

架构师

5月 ARCHITECT



特别专题

测试的未来

为功能测试构建通用mock server

黑天鹅发生的前后

让断言不再成为自动化测试的负担

JavaOne

关于Java性能的9个谬论

Java 8发布时间推迟到2014年

使用Java进行跨平台移动开发

书评：Java应用架构

goroutine背后的系统知识

JAVA线程池的分析和使用

技术突破可能性和高效能Geek团队





促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

本期主编

侯伯薇

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

变与不变

哲学基本原理中有一条：世间万事万物都处于变化之中。而古语中也说：穷则变，变则通，通则利。然而，还有一种说法：万变不离其宗。一件事物不论如何变化，总是会有不变的内容存在其中。

人生之中多变化，每天早上从睡梦中苏醒，我们都是全新的自己。在人生的每个阶段中，我们总是会根据身处的环境和具体情况不断做出选择，“衣带渐宽终不悔，为伊消得人憔悴。”然而，在回顾自己走过的路时，却经常会发现，时空虽然变幻，但最终往往找到的感觉和多年之前非常类似，似乎又回到了原点，正所谓“蓦然回首，那人却在灯火阑珊处。”

计算机领域发展之迅速，大家有目共睹。各种各样新的技术和新的产品层出不穷。然而，至少到现在为止，各种发展还是基于最基础的计算机基本原理。而与用户体验相关的各种设计，更是要以各种与其他领域共通的知识来支持。在那种情况下，计算机技术只是工具，其中体现的是一直没有变化的设计理念。

作为一名程序员，我们也要不断地应对业界日新月异的变化，新理念、新技术、新语言、新架构、新项目等等，我们都需要了解和学习，不断更新和完善自身。然而，在学习了众多知识和技能之后，往往你会发现，其中所蕴含的基本原理始终如一。所以，我们不仅要学习各种各样的“招式”来应对不同的情况，更需要练好内功，那样才可以以不变应万变。

InfoQ 中文站本身也在不断变化，从原来主要翻译英文站的新闻，到现在更多地注重原创；从原来单篇约稿，到现在创建多个专栏，刊登系列文章；就连页面的样式，也经历了多个版本，尽管暂时看来还不太习惯，但相信以后我们还会根据读者们的意見不断调整和完善。然而，一直不变的是为高端技术人员服务的理念，以及“促进软件开发领域知识与创新的传播”的宗旨。

一切都在变化，正是因为有变化才有发展；但还总会有不变的，那也正是事物存在的根本。希望我们都可以在适应变化的同时，保留自己不变的本心，可能是原则，可能是梦想，但都是内心深处的那一份执着。

本期主编：侯伯薇

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...

QClub

邀请
业内知名专家

自由开放的
讨论氛围

定期举办的线下活动

结识
圈内技术好友

InfoQ



中文 | 英文 | 日文 | 葡文 |

目录

卷首语	3
人物专访 INTERVIEW	7
性能测试面面观——HP 性能测试专家宗刚访谈	7
热点新闻 NEWS	18
Netflix 公布个性化和推荐系统架构	18
淘宝开源其系统监控工具 Tsar	23
开发人员如何有效地进行数据库设计	26
Apache Struts 1 宣告退出舞台	31
由中行 IBM 大型机宕机谈银行系统运维	34
特别专题 TOPIC	38
特别专题 TOPIC	39
自动化测试基础设施（一）——为功能测试构建通用 mock server 系统	39
软件测试中的黑天鹅（二）：黑天鹅发生的前后	45
让断言不再成为自动化测试的负担	52
企业系统集成点测试策略	58
本期专栏 COLUMN	76
关于 Java 性能的 9 个谬论	76
受困于连续出现的安全问题，Java 8 发布时间推迟到 2014 年	84

Tabris 1.0 : 使用 Java 进行跨平台移动开发	86
推荐文章 ARTICLES	90
书评 : Java 应用架构	90
goroutine 背后的系统知识	98
聊聊并发 (三) ——JAVA 线程池的分析和使用	109
豌豆荚王俊煜 : 技术突破可能性和高效能 Geek 团队	117
推荐编辑 翻译团队编辑方盛	124
封面植物 鸢尾花	126

人物专访 | Interview

性能测试面面观——HP 性能测试专家宗刚访谈

宗刚是 2013 北京 QCon 测试专题的受邀讲师。在 QCon 开始之前，专题出品人高楼对宗刚就性能测试领域内的许多问题做了深入的访谈。以下是访谈内容：

高楼：能否先简单谈谈您在测试领域的经验？和您对此领域的理解？

宗刚：我的工作经验主要分成三个阶段：

第一阶段：民企开发 Leader

毕业后做程序员，负责开发维护 6 个产品。解决过多次关键性能问题，其中有一次系统跑批 2 小时后死机，通过我的优化最后只需要 15 分钟完成，协助业务部门打了一个大胜战。06 年开始在项目组自下而上推一些敏捷实践。因为有开发编程以及敏捷工程的基础，为我后期进行大型系统性能测试、优化、规划以及提出全生命周期敏捷性能管理体系打下了坚实的基础。

第二阶段：创业公司负责人

和几位朋友一起创业，负责公司两个产品的运作，测试、开发、需求、业务、销售、人力各种工作都干过。虽然结果不太理想，但这个阶段磨练我从整体去看一个产品，从开发、测试、产品系统看问题，而不会局限于研发。

第三阶段：惠普非功能技术负责人

到惠普后，作为团队非功能技术负责人，推行敏捷软件开发，推行安全测试，提出性能测试优化建模整体服务整体解决系统上线过程中的技术难题，提出全生命周期敏捷性能与容量管理体系解决系统整个生命周期的性能与容量难题，提出交维服务系统解决从研发到运维的技术、流程等问题。主要为国内金融、电信、政府核心系统进行非功能服务。曾在创金融领域全球最高 TPS (HP 开放平台) 的项目担任性能技术专家。惠普是一个很大的平台，各种资源都有，性能测试工具、监控工具、刀片、小机、存储、网络、操作系统以及各种领域专家，最重要的是有很多大型应用系统的客户，非常有利于性能技术成长。

对于测试领域的理解：

- A. 从测试领域职业发展来看，将来的测试有两条比较好的出路，一条为业务专家，懂得某业务领域知识如金融、电信、建筑等等有行业壁垒的知识；另一条为测试技术专家，会编程是基础，能够做技术含量比较高的测试。原来主要点击几下鼠标的黑盒测试竞争会越来越激烈，由于知识壁垒有限，大量的高校毕业生轻易进入这个领域，很难出高薪。前几年是测试领域的原始积累期，很多技术能力不强的人能成为领导、经理，将来这种可能性越来越小。
- B. 性能测试领域现在还缺少标准，市面上的培训以及书籍多数以工具为主，没有系统解决性能问题关注性能测试为主。

高楼：能否简述企业中性能测试现状？

宗刚：从 09 年的淘宝双十一导致多家银行网银系统宕机 到 12306 购票难，再到前不久聚美优品促销活动刚开始就遭秒杀。根据 Google 的统计，如果网站打开慢每 500 毫秒，用户访问量将下降 20%。根据 Amazon 统计，每慢 100 毫秒，交易额下降 1%。这些事件和统计数据为大家敲响了警钟，企业也会越来越重视性能测试。

企业性能测试常见问题：

- A. 缺少整体性能与容量管理策略，常常临时抱佛脚，见过用 20 人年开发的系统上线之后系统性能完全无法满足要求，重新开发
- B. UAT 阶段才做测试。为时太晚，很多问题这个阶段无法解决或解决成本非常高
- C. 运维与研发缺乏互动。没有形成生产与研发的闭环，测试结果脱离实际，见过通过大量性能测试的系统上线之后就宕机
- D. 缺少性能优化和规划。只有性能测试报告，定位不了问题，提出不了建议，对于生产系统的容量和性能缺少规划，不清楚系统的容量，无法支持有效的业务决策。

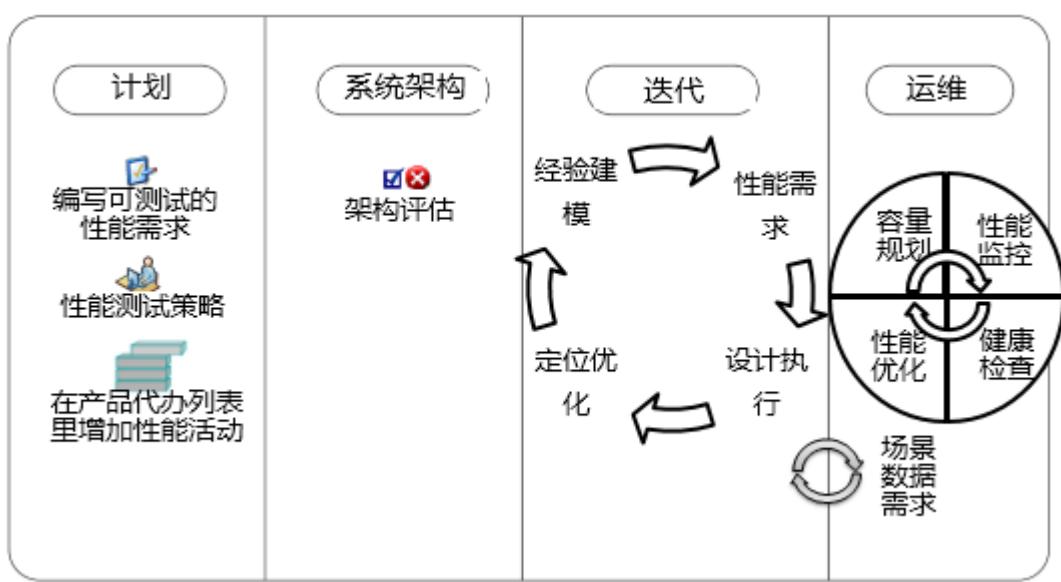
高楼：能否描述一下性能测试人员的现状？

宗刚：11 和 12 年分别在北京、上海、深圳面试了近 100 位性能测试人员，主要的特点如下：

- A. 性能测试人员出身比较复杂，有开发经验的人员能力和潜力都强于其他。由于性能测试项目比较少，所以不同角色的人遇到这种项目，就成为性能测试人员。由于性能测试对人员要求的技术比较高，相对之下有开发经验的人员学习速度要快得多。就拿我自己举例，由于有 4 年的开发经验，通过两个项目的实践就可以灵活掌握性能测试，1 年掌握的东西相当于没有开发经验的 3 年。
- B. 多数性能测试人员都以工具为主，缺少系统解决性能问题的能力。见过一个项目的性能测试人员只懂通过 loadrunner 设置场景发起压力，根本不会性能监控和瓶颈定位，测试的数据压倒生产库都不知道。
- C. 从面试的整体来看北京的技术能力稍强于其它地区，基本上为北京>上海>深圳。
- D. 很多“资深”性能测试的人员由于停留于几年前会 loadrunner 就是专家的时代，技能没有提升，陷入“上不去，下不来”的尴尬境地。
- E. 多数人员习惯于从测试看问题，缺少整体视角解决性能问题。个人建议从产品经理的角度看问题，因为一个产品其实就是一个小型企业，从这个角度看成本、创新、流程、质量就比较有意义，抓住了本质。
- F. 性能测试领域非常缺人才，缺少精通性能测试，同时熟悉各层性能优化的人才。见过好几个企业有若干个 OS、中间件、DB 性能优化专家，但是解决不了性能问题，缺少整体贯通的人员。

高楼：你如何理解性能测试在软件生命周期中的位置？

宗刚：性能测试应该贯穿整个软件的生命周期，从需求到架构到迭代到上线再到运维都和性能测试息息相关。下图为借鉴了敏捷性能工程的思路整理出来的一个全生命周期性能管理图。



主要分成 4 个大阶段：

A. 计划阶段：

编写可测试的性能需求：详细说明可落地可测试的需求，而不是笼统的写着一天支持 1.5 亿的交易，支持 1 亿的用户。

性能测试策略：需要提前考虑怎么进行性能测试，用什么工具？需要哪些培训等等
 在产品代办列表里增加性能活动：由于性能测试一般实施周期比较长，建议单独成为一个列表项。

B. 架构评估：

在系统架构阶段，在实现部分关键功能的情况下评估系统性能、容量、安全、可扩展性、稳定性等等是否满足系统设计的需要，我们常常缺少这个阶段的实践，等系统开发结束才进行，常常为时已晚。在系统规划时常常在缺少实际测试数据的时候拍脑袋规划硬件，出现“大马拉小车”的局面，架构评估的另一个作用是通过架构阶段的评估为规划提供数据支持。

C. 迭代阶段：

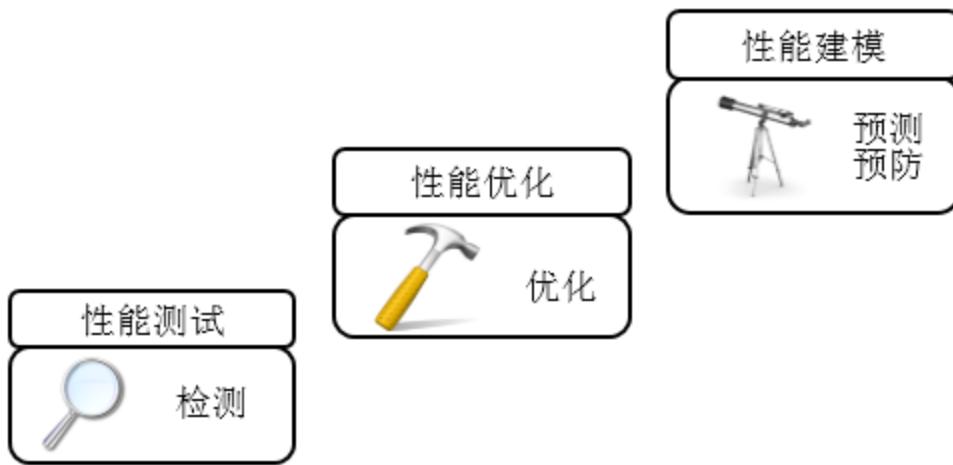
系统不断的增加新特性、新需求，需要迭代验证系统的性能与容量

D. 运维阶段：

研发人员常常觉得系统交给运维就可以了，由于运维人员对应用本身不够清楚，所以常常是盲人摸象，抓不住根本。见过业务高峰期，运维人员就看着

CPU 在往上涨，不知道应该怎么办，不清楚系统的容量点会在哪里出现，系统宕掉一台服务器会怎么样？多长时间能够恢复？到底能够支持多少的业务量？什么业务比较消耗时间？怎么优雅降级？在研发环境中，获得这些数据和手段都是比较容易的，运维人员是研发的第一个客户，应该多为他们考虑。

上面介绍了整个生命周期性能的管理，从广度角度讲。那么从深度角度讲，性能管理应该包括：性能测试、性能优化和性能建模容量规划。

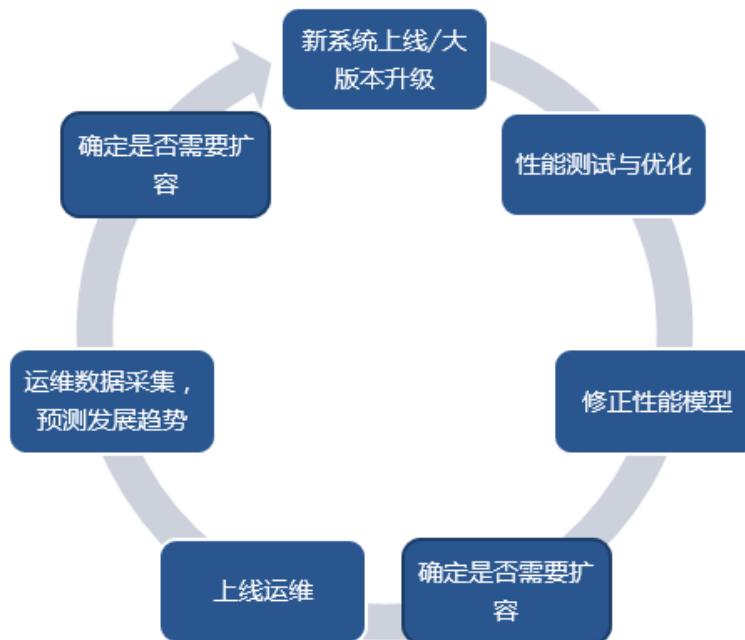


性能测试：验证系统性能是否满足需求。

性能优化：优化性能瓶颈，提升系统处理能力，测试和优化会有若干次的迭代。
性能建模容量规划：生产环境可能出现各种场景，应该怎么预测与预防。

如果比喻整个过程为病人看病，那么性能测试就是体检，性能优化就是对病下药，性能建模容量规划就是保健。

由于系统总在变化，新业务、扩容、软硬件版本升级等等，所以需要不断的迭代，如下图：



高楼：你如何判断性能测试在项目或产品中的实际价值？

宗刚：实际价值：

- A. 业务部门：支持业务决策和促销，提高客户满意度
- B. 运维部门：清楚系统容量，提升系统可用性、稳定性，降低硬件采购成本，提前预测预防提高响应速度，睡觉可以安心
- C. 规划部门：有理有据进行规划
- D. 研发部门：减少运维部门给研发部门的压力

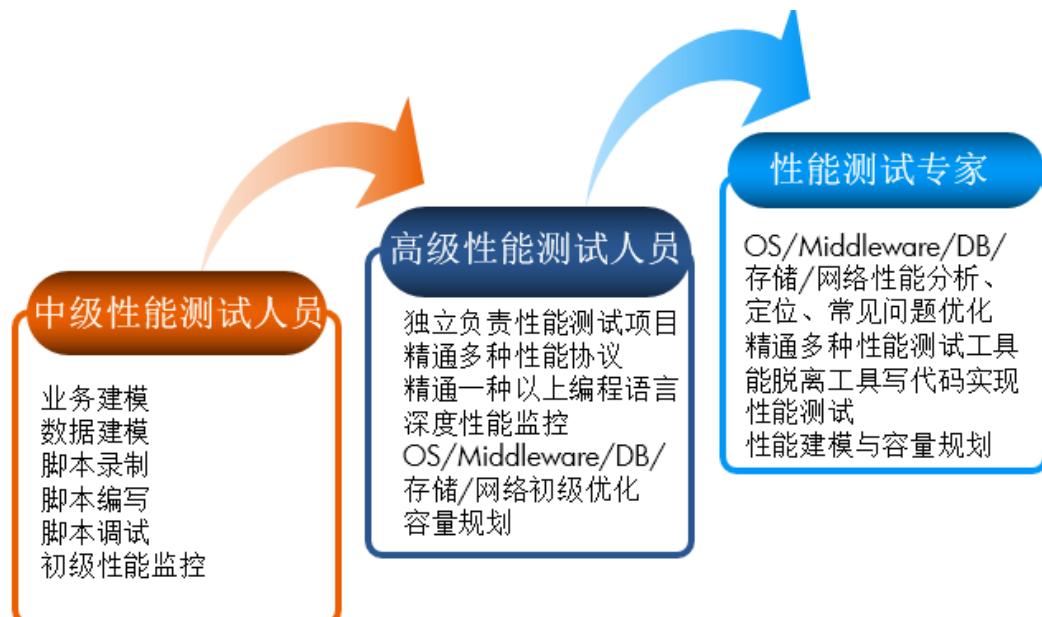
高楼：你觉得高级性能测试专家需要什么样的个人能力和素质？

宗刚：高级性能测试专家需要的能力模型，如下图所示，成为博学的专家。

OS、中间件、数据库、应用 存储、网络、沟通、敏捷方法

性 能 测 试 分 析 建 模

精通于性能测试分析建模，熟悉系统各层的监控与优化。同时需要较强的沟通能力，为了实施好项目需要有过程领域的知识如敏捷、CMMI 等。性能测试技术发展路线参考如下：



成为一个高级性能测试人员需要掌握的东西非常多，如何学习这些知识。通过多年的实践归纳，我的一点学习方法和大家分享：

成长=3 本书+领域专家+实验+实战+持续提升。

3 本书：1 本基础书籍+1 本全面的理论书籍+1 本实战书籍。所有的书一定要经典书籍，我们常常开始学习知识的时候通过论坛的方法，这种方法入门比较容易，但是很难系统，也会占据非常多的时间。为什么要经典书？读书的最大成本不是买书的钱，而是读书的时间，所以看书一定要看最经典的书，怎么找经典书？可以到豆瓣、专业论坛、当当上看看评论。每个领域有每个领域的书籍，学习 Oracle 优化有 oracle 的书籍，学习 loadrunner 有 loadrunner 的书籍，千万不要以为做性能测试就学 3 本性能测试的书籍就够了。

领域专家：成长过程中如果有领域专家的支持，就会少走不少弯路。当我开始学习 Oracle 性能调优的时候，刚好认识一位 Oracle 调优同事，和他沟通请他推荐一些资料，讲讲实操的技巧。这里需要注意的一点是不要所有毛皮小事都去找专家，人家也有自己的工作。一些问题可以通过 google 搜索、或专业论坛就可以解决。前段时间有项目需要用 informix 数据库，就请一位 informix 专家给我指导，大多数技术小问题在技术论坛上都有。如果大家不认识专家，那也没有关系，通过微博、论坛认识他们，大多数人还是愿意帮忙的。

实验：性能项目不是那么多，所以自己要找一些实验的内容。技术书籍一般都会有一些实战的内容，如果不实际操作一遍往往过一段时间就忘了。当我学习 Tuxedo 调优的时候，在自己的虚拟机上搭建 Tuxedo 环境，使用修改后的 Demo 应用进行压力测试，设计不同的压力场景，测试过程不断去调优应用，这个学习过程成长会很快。我的好多同事为了学习好 hp-ux，自己购买退役的小机搭环境，进行实验调优。很多时候不是技术难，而是没有机会接触这样的环境。有过实验的经历，在就职面试的时候也算是经验啊。

实战：通过实验之后，基本有经验了，如果再通过几个实战项目，不断总结归纳，基本就成为一个中级的性能测试人员了。以战养战，没有一个人开始就会所有的东西，每个项目都会用一些新的技术，所以在不同的项目中需要有很强的学习能力，能够快速学习新的技术并用于实战。

持续提升：想成为高级性能测试人员或专家，就需要不断更新学习新的知识和技术。通过论坛、活动、微博、读书等方式不断提升，也要常常和大家一起分享，分享是非常好的学习手段，还可以提高自己的知名度。

高楼：如何从业务目标分析得到性能测试需求、性能指标？

宗刚：常见的业务需求如下：日交易量支持 1.4 亿，响应时间小于 2 秒，支持用户 2 亿。我们需要把这些指标转化为可以测试的指标和场景。通过分析历史交易的波峰波谷，把 1.4 亿的交易量折换为每秒钟的交易量；响应时间也可以分类，比如本地业务多长时间，跨省业务多长时间，跨行业务多长时间等等；我们常常把支持多少用户作为衡量指标，这是一个误区，大量用户导致产生大量的业务才会消耗系统的利用率，所以关键是业务量。这里有个例外，如果要验证支持多少在线用户，以及长连接的系统就需要考虑支持的用户数量更精确的说法应该是支持的 Session。从业务需求到性能测试需要一般要经历这些过程：评估性能风险、确定关键用例、选择关键性能场景、建立可测试性能目标。

性能指标一般会有：交易响应时间、交易成功率、资源利用率、每秒钟的交易量（TPS）。这几个指标是相互约束的，如果低成功率下的 TPS 是没有意义的。多数运维部门对资源利用率都会有一些硬规定，比如 CPU 不能高于 85%，内存不能高于 90% 等等，所以在测试之前需要清楚这些约束。除了上面的通用指标，各个应用系统会依据技术特点有一些特殊的指标。更全面的指标应该是分层的，从终端用户的体验—>业务流程—>中间件数据库的响应—>基础架构的利用。

高楼：如何进行性能测试建模？在性能测试过程中要建立哪些模型？

宗刚：性能测试过程需要考虑的模型有：业务模型、测试模型、用户模型 TPS 模型、数据模型、失效模型、性能模型

业务模型：依据应用系统特点分析出来的不同的业务场景和业务配比。性能测试常常会通过历史数据分析业务的重要性、交易频率、易耗资源的交易以及未来的发展趋势，最后确定一种业务配比。依据这个业务配比设计测试场景。这往往是不够的，一个线上系统往往有多个业务模型，需要考虑时间驱动的如：白天、晚上、月末、过年过节，事件驱动的如：本拉登去世黄金业务突发变化、业务部门促销等，第三方驱动：永远不要相信第三方的内容，所以需要考虑第三方接口的业务突变，延时等等。

测试模型：如何将业务模型的内容转化为可以测试的内容，就是测试建模需要解决的问题。通过业务建模分析出来的业务需要过滤，剔除一些不易执行的、相互包含的等等业务。最后落地为各种可执行的业务配比，业务配比完

成后，需要考虑的是如何和测试工具映射起来，这个就牵涉到用户模型和 TPS 模型。用户模型是指按照业务配比设置发起压力的用户比例，这种方法存在一定的局限性，因为不同的交易响应时间是不同的，长交易完成 1 笔交易，短交易可能是 5 笔，特别是在较大压力时，测试结果的业务配比会和真实的业务配比差距很大。所以一般情况下需要考虑 TPS 模型，这个是和业务模型相同的配比，这个模型的一个劣势就是需要不断调整并发用户数。

数据模型：一个系统的大多数性能问题是数据库问题，所以垫底数据或参数化数据是否和实际相符将直接影响性能测试的有效性。一般建议性能测试使用清洗后的生产数据，参数化数据尽量采集生产系统一天的交易数据。以前见过一个项目，说有的数据都是通过 loadrunner 压进去的，所有数据都集中在一块，测试结果和实际生产差距巨大，整个测试无效。

失效模型：主要是总结了大量以往生产系统经常出错的模式，在设计测试案例的时候需要着重考虑。这部分要依据实际情况来定，如果能够从运维部门获得更多的事故数据就更有价值。

性能模型：不同的交易对系统的性能要求是不同的，依据测试的数据以及生产环境的数据建立模型，主要解决以下问题：测试环境中测试的数据如何映射到生产环境？生产环境中出现性能问题应该如何预测预防和优雅降级？生产环境应该如何扩容等等。

高楼：这次 QCon 测试专题中，您的专题讲座会给听众带来哪些收获？

宗刚：这次演讲我会分享如何进行大型交易系统性能测试与分析，并讨论有哪些需要注意的地方。在内容中我会以一个大型案例为主线，并通过多个成功与失败案例为辅系统解决性能问题。这里面大家将会了解到：

- A. 风险驱动的性能测试与分析计划
- B. 历史数据与未来业务并重的业务模型分析
- C. 注重实效的场景设计
- D. 符合实际的测试数据
- E. 被测系统和压力系统并重的性能监控
- F. 快速的性能瓶颈定位

G. What-IF 分析与优雅降级

原文链接：<http://www.infoq.com/cn/articles/performance-test>

相关内容

- [性能测试面面观——HP 性能测试专家宗刚访谈](#)
- [软件测试魅力何在——QCon 北京测试专题出品人高楼访谈](#)
- [不喜欢绘画的木工不是好测试人员——QCon 北京测试专题讲师王东刚访谈](#)
- [软件测试魅力何在——QCon 北京测试专题出品人高楼访谈](#)
- [百度技术沙龙第 18 期回顾：大型网站的性能测试实践及结果分析（含资料下载）](#)
- [云测试探索实践](#)

ThoughtWorks®

JOIN US

测试工程师 / QA

软件开发工程师 / Developer

移动开发工程师 / Mobile Developer

前端开发工程师 / UI Developer

业务分析师 / Business Analyst

ThoughtWorks 北京 | 西安 | 成都
同步招聘，等你加入！

Join.thoughtworks.com

热点新闻 | News

Netflix 公布个性化和推荐系统架构

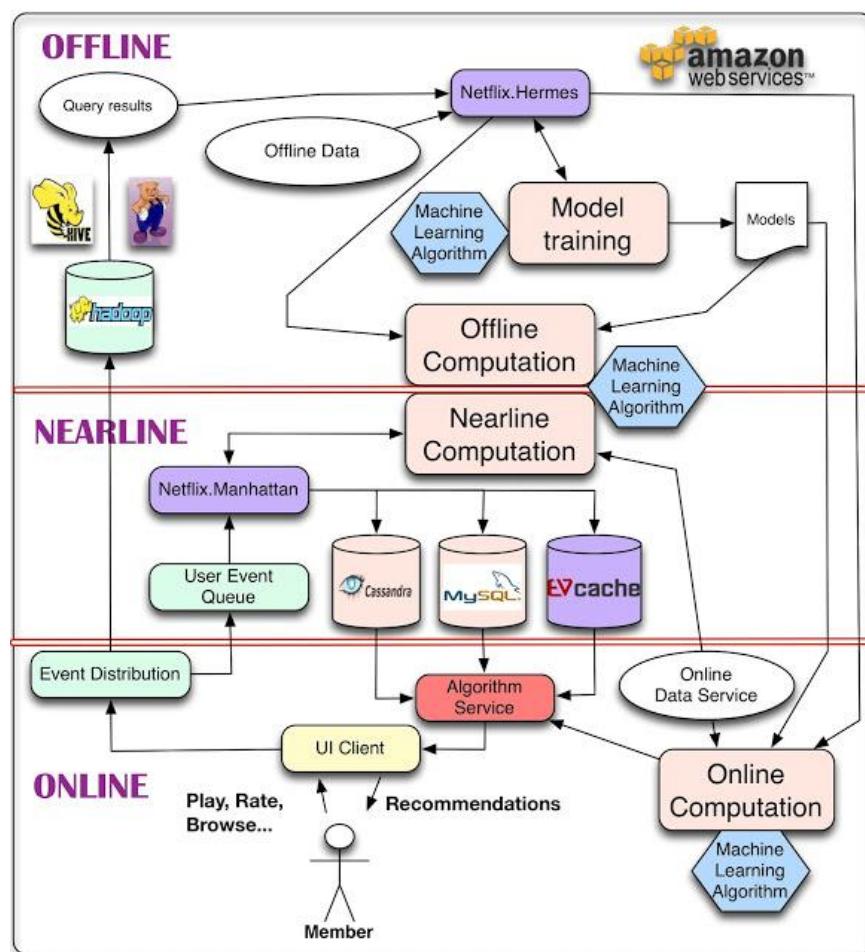
作者 [郑柯](#)

Netflix 的推荐和个性化功能向来精准，前不久，他们公布了自己在这方面的系统架构。

3月27日，Netflix 的工程师 [Xavier Amatraining](#) 和 [Justin Basilico](#) 在官方博客[发布文章](#)，介绍了自己的个性化和推荐系统架构。文章开头，他们指出：

要开发出这样的一个软件架构，能够处理海量现有数据、响应用户交互，还要易于尝试新的推荐方法，这可不一点都不容易。

接下来，文章贴出了他们的系统框架图，其中的主要组件包括多种机器学习算法。



他们这样解释其中的组件和处理过程：

对于数据，最简单的方法是存下来，留作后续离线处理，这就是我们用来管理离线作业（Offline jobs）的部分架构。计算可以以离线、接近在线或是在线方式完成。在线计算（Online computation）能更快地响应最近的事件和用户交互，但必须实时完成。这会限制使用算法的复杂性和处理的数据量。离线计算（Offline computation）对于数据数量和算法复杂度限制更少，因为它以批量方式完成，没有很强的时间要求。不过，由于没有及时加入最新的数据，所以很容易过时。个性化架构的关键问题，就是如何以无缝方式结合、管理在线和离线计算过程。接近在线计算（Nearline computation）介于两种方法之间，可以执行类似于在线计算的方法，但又不必以实时方式完成。模型训练（Model training）是另一种计算，使用现有数据来产生模型，便于以后在对实际结果计算中使用。另一块架构是如何使用事件和数据分发系统（Event and Data Distribution）处理不同类型的数据和事件。与之相关的问题，是如何组合在离线、接近在线和在线之间跨越的不同的信号和模型（Signals and Models）。最后，需要找出如何组合推荐结果（Recommendation Results），让其对用户有意义。

接下来，文章分析了在线、接近在线和离线计算。

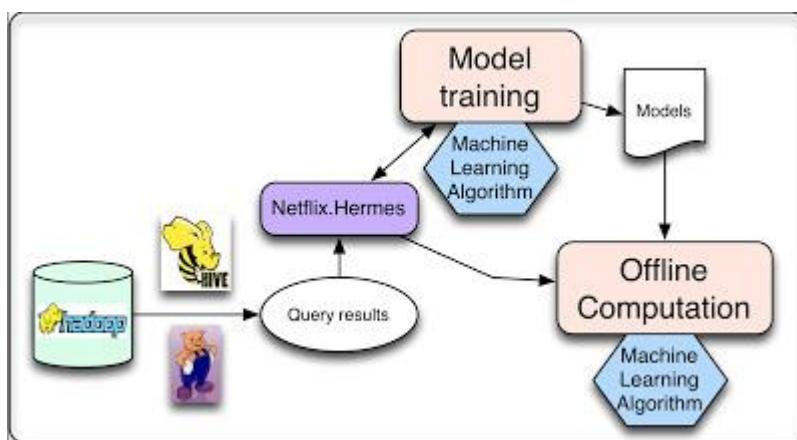
对于在线计算，相关组件需要满足 SLA 对可用性和响应时间的要求，而且纯粹的在线计算在某些情形下可能无法满足 SLA，因此，快速的备用方案就很重要，比如返回预先计算好的结果等。在线计算还需要不同的数据源确保在线可用，这需要额外的基础设施。

离线计算在算法上可相对灵活，工程方面的需求也简单。客户端的 SLA 响应时间要求也不高。在部署新算法到生产环境时，对于性能调优的需求也不高。Netflix 利用这种灵活性来完成快速实验：如果某个新的实验算法执行较慢，他们会部署更多 Amazon EC2 实例来达成吞吐处理目标，而不是花费宝贵的工程师时间去优化性能，因为业务价值可能不是很高。

接近在线计算与在线计算执行方式相同，但计算结果不是马上提供，而是暂时存储起来，使其具备异步性。接近在线计算的完成是为了响应用户事件，这样系统在请求之间响应速度更快。这样一来，针对每个事件就有可能完成更复杂的处理。增量学习算法很适合应用在接近在线计算中。

不管什么情况，选择在线、接近在线、还是离线处理，这都不是非此即彼的决策。所有的方式都可以、而且应该结合使用。…… 即使是建模部分也可

以用在线和离线的混合方式完成。这可能不适合传统的监督分类法 (supervised classification) 应用，因为分类器必须从有标记的数据中批量培训，而且只能以在线方式使用，对新输入分类。不过，诸如矩阵因子分解这样的方法更适合混合离线和在线建模方法：有些因子可以预先以离线方式计算，有些因子可以实时更新，创建更新的结果。其他诸如集群处理这样的非监督方法，也可以对集群中心进行离线计算，对集群节点进行在线作业。这些例子说明：模型训练可以分解为大规模和复杂的全局模型训练，以及轻量级的用户指定模型训练或更新阶段，以在线方式完成。

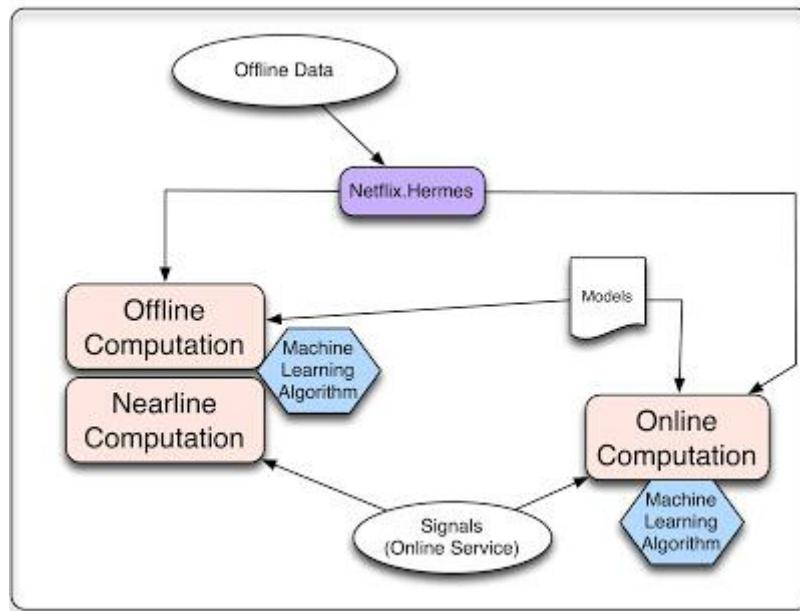


对于离线作业 (Offline jobs)，主要用来运行个性化机器学习算法。这些作业会定期执行，而且不必与结果的请求和展示同步。主要有两种任务这样处理：模型训练和中间与最终结果批量计算 (batch computation of intermediate or final results)。不过，他们也有一些学习算法是以在线增量方式完成的。

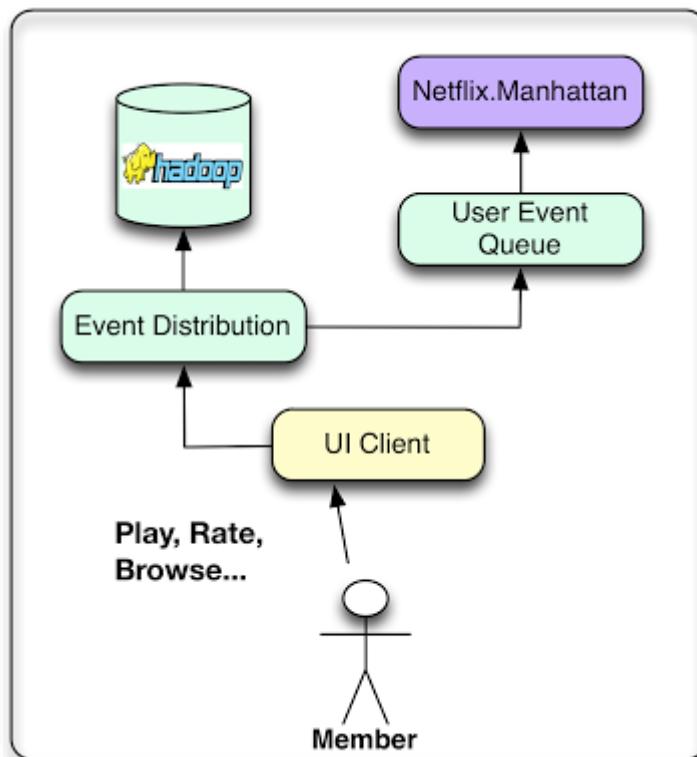
这两种任务都需要改善数据，通常是由数据库查询完成。由于这些查询要操作大量数据，以分布式方式完成更方便，因此通过 Hadoop 或是 Hive、Pig 作业就是自然而然的事情。一旦查询完成，就需要某种机制发布产生的数据。对于这样的机制，Netflix 有如下需求：

- 可以通知订阅者查询完成。
- 支持不同存储方式（不只是 HDFS，还有 S3 或是 Cassandra 等等）
- 应该透明处理错误，允许监控和报警。

Netflix 使用内部的工具 Hermes 完成这些功能，它将数据以接近实时的方式交付给订阅者，在某些方面接近 [Apache Kafka](#)，但它不是消息/事件队列系统。



无论是离线还是在线计算，都需要处理三种输入：模型、数据和信号。模型是以离线方式训练完成的参数文件，数据是已完成处理的信息，存在某种数据库中。在 Netflix，信号是指输入到算法中的新鲜信息。这些数据来自实时服务，可用其产生用户相关数据。



对于事件和数据分发，Netflix 会从多种设备和应用中收集尽可能多的用户事件，并将其集中起来为算法提供基础数据。他们区分了数据和事件。事件是对时间敏

感的信息，需要尽快处理。事件会路由、触发后续行动或流程。而数据需要处理和存储，便于以后使用，延迟不是重要，重要的是信息质量和数量。有些用户事件也会被作为数据处理。

Netflix 使用内部框架 Manhattan 处理接近实时的事件流。该分布式计算系统是推荐算法架构的中心。它类似 Twitter 的 [Storm](#)，但是用处不同，而且响应不同的内部需求。数据流主要通过 [Chukwa](#)，输入到 Hadoop，进行处理的初步阶段。此后使用 Hermes 作为发布-订阅机制。

Netflix 使用 Cassandra、EVCache 和 MySQL 存储离线和中间结果。它们各有利弊。MySQL 存储结构化关系数据，但会面临分布式环境中的扩展性问题。当需要大量写操作时，他们使用 EVCache 更合适。关键问题在于，如何满足查询复杂度、读写延迟、事务一致性等彼此冲突的需求，要对于各种情况到达某个最优点。

在总结中，他们指出：

我们需要具备使用复杂机器学习算法的能力，这些算法要可以适应高度复杂性，可以处理大量数据。我们还要能够提供灵活、敏捷创新的架构，新的方法可以很容易在其基础上开发和插入。而且，我们需要我们的推荐结果足够新，能快速响应新的数据和用户行为。找到这些要求之间恰当的平衡不容易，需要深思熟虑的需求分析，细心的技术选择，战略性的推荐算法分解，最终才能为客户达成最佳的结果。

原文链接：<http://www.infoq.com/cn/news/2013/04/netflix-ml-architecture>

相关内容

- [Netflix 公布个性化和推荐系统架构](#)
- [推荐系统的工程挑战](#)
- [Twitter 研发人员 John Oskasson 分析 Twitter 后台软件栈](#)
- [Apache 基金会主席 Doug Cutting 谈 Hadoop 和开源](#)
- [Hortonworks 宣布一款 Hadoop 数据平台](#)
- [Windows Azure 更新，支持 Hadoop、HTML5/JS、CORS、PhoneGap、Mercurial 和 Dropbox](#)

热点新闻 | News

淘宝开源其系统监控工具 Tsar

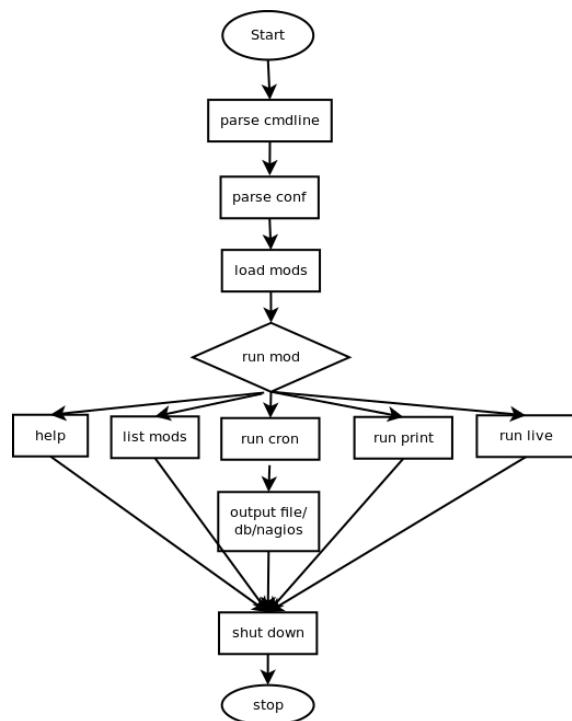
作者 贾国清

淘宝在开发社区的贡献可谓功不可没，近日又将其系统监控工具 [Tsar](#) 代码上传至 [GitHub](#)。据@淘叔度介绍，Tsar 在淘宝内部已经被大量使用，它不仅可以监控 CPU、IO、内存、TCP 等系统状态，也可监控 Apache、Nginx/Tengine、Squid 等服务器状态。

Tsar (Taobao System Activity Reporter) 可将收集到的数据存储在磁盘上，另外可以支持将数据存储到 MySQL 中，也可将数据发送到 Nagios 报警服务器。在展示数据层面，可以指定模块，并且支持对多条信息的数据进行 Merge 输出，如果带--live 参数，还可以输出秒级的实时信息。

从架构上来看，Tsar 基于模块化设计，源代码上来看主要包含两部分：框架和模块。框架源代码主要存放在 src 目录，模块源代码主要在 modules 目录中。框架提供对配置文件的解析、模块的加载、命令行参数的解析、应用模块的接口对模块原始数据的解析与输出。

Tsar 的运行流程图如下：



主要执行流程：

- 解析输入
- 读取配置文件信息
- 加载相应模块
- Tsar 的三种模式 (print、live 和 cron) 执行
- 释放资源

本次开源消息公布后，受到了社区开发者的欢迎：

TriChaos：喜欢没有浮华的字符界面，像和没有废话的人对话。

_Yuen：代码少逻辑清晰模块化又好，不得不说是一个好工具的典范。

淘木名：事后线上问题分析必备利器。

淘叔度：目前只支持 C 语言开发的插件。不过写个模块非常简单。

得益于淘宝开源

[淘宝开源平台](#)于 2010 年 6 月上线，至今，该平台已经发布了若干开源项目，其中不乏来自于淘宝之外的项目在此落户。目前注册会员数 13002 人，收录项目已达到 2875 个。目前，该平台关注度最高的 5 个项目分别是 [tfs](#)、[tair](#)、[webx](#)、[OceanBase](#)、[athrun](#)。

扩展阅读

- [GitHub 引起开源“平民化”变革](#)
- [走近淘宝开源平台](#)
- [腾讯前端 Alloy 团队访谈：HTML5 开源图像处理框架 AlloyImage](#)
- [58 同城开源其轻量级 Web 框架 Argo](#)

原文链接：<http://www.infoq.com/cn/news/2013/04/taobao-open-tsar>

相关内容

- [淘宝开源其系统监控工具 Tsar](#)
- [GitHub 引起开源“平民化”变革](#)

- [腾讯 AlloyTeam 再次发力：开源 HTML5 图像处理引擎 AlloyImage](#)
- [58 同城开源其轻量级 Web 框架 Argo](#)
- [Redis 开源文档《Redis 设计与实现》](#)
- [淘宝开源分布式消息中间件 Metamorphosis](#)

热点新闻 | News

开发人员如何有效地进行数据库设计

作者 [崔康](#)

数据库设计在软件开发过程中占有重要的地位，国内开发者 MeteorSeed 在[博客](#)中结合自己的实际经历全面总结了关系型数据库设计需要注意的各个方面，包括 Codd 的基本法则、设计阶段、设计原则和命名规则。

MeteorSeed 认为在项目早期应该由开发者进行数据库设计，后期调优则需要 DBA：“一个精通 OOP 和 ORM 的开发者，设计的数据库往往更为合理，更能适应需求的变化”。他引用了关系数据库之父 Codd 的 12 条法则，作为数据库设计的指导性方针：

1. 信息法则

关系数据库中的所有信息都用唯一的一种方式表示——表中的值。

2. 保证访问法则

依靠表名、主键值和列名的组合，保证能访问每个数据项。

3. 空值的系统化处理

支持空值（NULL），以系统化的方式处理空值，空值不依赖于数据类型。

4. 基于关系模型的动态联机目录

数据库的描述应该是自描述的，在逻辑级别上和普通数据采用同样的表示方式，即数据库必须含有描述该数据库结构的系统表或者数据库描述信息应该包含在用户可以访问的表中。

5. 统一的数据子语言法则

一个关系数据库系统可以支持几种语言和多种终端使用方式，但必须至少有一种语言，它的语句能够以某种定义良好的语法表示为字符串，并能全面地支持以下所有规则：数据定义、视图定义、数据操作、约束、授权以及事务。

（这种语言就是 SQL）

6. 视图更新法则

所有理论上可以更新的视图也可以由系统更新。

7. 高级的插入、更新和删除操作

把一个基础关系或派生关系作为单个操作对象处理的能力不仅适应于数据

的检索，还适用于数据的插入、修改和删除，即在插入、修改和删除操作中数据行被视作集合。

8. 数据的物理独立性

不管数据库的数据在存储表示或访问方式上怎么变化，应用程序和终端活动都保持着逻辑上的不变性。

9. 数据的逻辑独立性

当对表做了理论上不会损害信息的改变时，应用程序和终端活动都会保持逻辑上的不变性。

10. 数据完整性的独立性

专用于某个关系型数据库的完整性约束必须可以用关系数据库子语言定义，而且可以存储在数据目录中，而非程序中。

11. 分布独立性

不管数据在物理是否分布式存储，或者任何时候改变分布策略，RDBMS 的数据操纵子语言必须能使应用程序和终端活动保持逻辑上的不变性。

12. 非破坏性法则

如果一个关系数据库系统支持某种低级（一次处理单个记录）语言，那么这个低级语言不能违反或绕过更高级语言（一次处理多个记录）规定的完整性法则或约束，即用户不能以任何方式违反数据库的约束。

MeteorSeed 把数据库设计阶段分为规划阶段、概念阶段、逻辑阶段、实现阶段和物理阶段。关于设计原则，他从以下几个方面阐述了自己的经验：

- 降低对数据库功能的依赖

功能应该由程序实现，而非 DB 实现。原因在于，如果功能由 DB 实现时，一旦更换的 DBMS 不如之前的系统强大，不能实现某些功能，这时我们将不得不去修改代码。所以，为了杜绝此类情况的发生，功能应该有程序实现，数据库仅仅负责数据的存储，以达到最低的耦合。

- 定义实体关系的原则

当定义一个实体与其他实体之间的关系时，需要考量如下：

- 牵涉到的实体 识别出关系所涉及的所有实体。
- 所有权 考虑一个实体“拥有”另一个实体的情况。
- 基数 考量一个实体的实例和另一个实体实例关联的数量。

关系与表数量

- 描述 1:1 关系最少需要 1 张表。
- 描述 1:n 关系最少需要 2 张表。
- 描述 n:n 关系最少需要 3 张表。
- 列意味着唯一的值

如果表示坐标 (0,0) , 应该使用两列表示 , 而不是将“0,0”放在 1 个列中。
- 列的顺序

列的顺序对于表来说无关紧要 , 但是从习惯上来说 , 采用“主键+外键+实体数据+非实体数据”这样的顺序对列进行排序显然能得到比较好的可读性。
- 定义主键和外键

数据表必须定义主键和外键 (如果有外键) 。定义主键和外键不仅是 RDBMS 的要求 , 同时也是开发的要求。几乎所有的代码生成器都需要这些信息来生成常用方法的代码 (包括 SQL 文和引用) , 所以 , 定义主键和外键在开发阶段是必须的。之所以说在开发阶段是必须的是因为 , 有不少团队出于性能考虑会在进行大量测试后 , 在保证参照完整性不会出现大的缺陷后 , 会删除掉 DB 的所有外键 , 以达到最优化能。 MeteorSeed 认为 , 在性能没有出现问题时应该保留外键 , 而即便性能真的出现问题 , 也应该对 SQL 文进行优化 , 而非放弃外键约束。
- 选择键

人工键与自然键。人工键——实体的非自然属性 , 根据需要由人强加的 , 如 GUID , 其对实体毫无意义 ; 自然键——实体的自然属性 , 如身份证编号。人工键的好处 : 键值永远不变 ; 永远是单列存储。人工键的缺点 : 因为人工键是没有实际意义的唯一值 , 所以不能通过人工键来避免重复行。 MeteorSeed 建议全部使用人工键。原因如下 :

- 在设计阶段我们无法预测到代码真正需要的值 , 所以干脆放弃猜测键 , 而使用人工键。
- 人工键复杂处理实体关系 , 而不负任何属性描述 , 这样的设计使得实体关系与实体内容得到高度解耦 , 这样做的设计思路更加清晰。

MeteorSeed 的另一个建议是——每张表都需要有一个对用户而言有意义的自然键 , 在特殊情况下也许找不到这样一个项 , 此时可以使用

复合键。这个键我在程序中并不会使用其作为唯一标识，但是却可以在对数据库直接进行查询时使用。使用人工键的另一个弊端，主要源自对查询性能的考量，因此选择人工键的形式（列的类型）很重要：

- 自增值类型，由于类型轻巧查询效率更好，但取值有限。
- GUID 查询效率不如值类型，但是取值无限，且对开发人员更加亲切。

智能键与非智能键。智能键——键值包含额外信息，其根据某种约定好的编码规范进行编码，从键值本身可以获取某些信息；非智能键，单纯的无意义键值，如自增的数字或 GUID。智能键是一把双刃剑，开发人员偏爱这种包含信息的键值，程序盼望着其中潜在的数据；数据库管理员或者设计者则讨厌这种智能键，原因也是很显然的，智能键对数据库是潜在的风险。前面提到，数据库设计的原则之一是不要把具有独立意义的值的组合实现到一个单一的列中，应该使用多个独立的列。数据库设计者，更希望开发人员通过拼接多个列来得到智能键，即以复合主键的形式给开发人员使用，而不是将一个列的值分解后使用。开发人员应该接受这种数据库设计，但是很多开发者却想不明白两者的优略。MeteorSeed 认为，使用单一列实现智能键存在这样一个风险，就是我们可能在设计阶段无法预期到编码规则可能会在后期发生变化。比如，构成智能键的局部键的值用完而引起规则变化或者长度变化，这种编码规则的变化对于程序的有效性验证与智能键解析是破坏性的，这是系统运维人员最不希望看到的。所以 MeteorSeed 建议如果需要智能键，请在业务逻辑层封装（使用只读属性），不要再持久化层实现，以避免上述问题。

除此之外，MeteorSeed 还从“是否允许 NULL”、属性切割、规范化（范式）、选择数据类型、优化并行等几个方面谈了设计原则。有关详细内容，可以查看 MeteorSeed 的博客[原文](#)。

原文链接：<http://www.infoq.com/cn/news/2013/04/db-design-principle>

相关内容

- [开发人员如何有效地进行数据库设计](#)
- [2012.4.16 微博热报：Web 开发新趋势、11 个数据库设计规则](#)

- [James Phillips 谈从关系型数据库转到 NoSQL](#)
- [MySQL 数据库开发的三十六条军规](#)
- [带有基于 Smalltalk 的 Ruby VM 的 NoSQL OODB :MagLev 1.0 发布了](#)
- [云时代的列式数据库——Sybase IQ15.3 新特性](#)

热点新闻 | News

Apache Struts 1 宣告退出舞台

作者 贾国清

近日，Apache 官方网站发布了关于 Apache Struts 1 EOL (End-Of-Life) [新闻稿及通告](#)。该新闻稿指出，2013 年 4 月 5 日，Apache Struts 项目团队正式通知广大开发者，Struts 1.x 开发框架结束使命，并且官方将不会继续提供支持。

Struts 1.x 项目创建于 2000 年，最新版本 1.3.10 发布于 2008 年 12 月。同期，Struts 社区将精力专注于推动 Struts 2 框架的发展，截止到 2013 年 4 月，已发布了 23 个版本。据通告称：

此次宣告 Struts 1.x 退出舞台并不再提供支持，主要是因为缺少足够的志愿者来提供支持。

与此同时，在[新闻稿](#)中，Apache Struts 团队强烈建议大家学习 Struts 2 框架，Struts 2 更加现代、高度解耦、功能丰富且易于维护。就在 3 月，Apache Struts 刚刚发布了 2.3.12 版，这个版本为维护版本，包含了一些很小的改进，如：

- 重构所有验证器，可通过 OGNL 设置参数
- Tag 的 required 属性改名为 requiredLabel 支持 HTML5 的 required 属性
- 三个新的 Tiles 插件，用于支持 Tiles 3 结果类型
- 改进支持 JBoss 5 的 Convention 插件

此外，对于正在使用 Struts 1.x 的开发者或团队，Apache Struts 团队就常见问题给出了回答：

Struts 1.x 不再提供支持后，现有资源如何处理？

所有资源将会保留，Apache Struts 首页将会提供相关文档的链接，同时也会保留 Struts 1.x 各个版本的下载地址。所有的 Struts 1 源代码均可在 Apache Struts 代码仓库中找到，并且永久保留。所有发布的 Maven 构件（Maven artifacts）均可通过 Maven Central 访问。

如果以后发现和 Struts 1.x 相关的安全问题或严重的 Bug，是否还会有相应的修复？

目前来看，是不会的，这也是要宣布 EOL 的原因。既然现在已经宣布不再提供支持，开发者也需要寻找移植方案，将现有的 Struts 1 代码移植到其他 Web 框架上。

现在是不是就需要将 Struts 1 从我的项目中删除？

就目前 Struts 团队了解的情况来看，不必立即删除。然而需要意识到的是，未来将不会有针对安全和 Bug 问题的修复，一旦未来发现上述情况，开发团队需要自己来应对。

如果需要将现有项目从 Struts 1 移植到其他 Web 框架，有什么推荐？

您也许会意识到，目前为止 Struts 1 还没有直接的替代品。您需要根据现有代码的情况，来选择新的开发框架，并通过调整代码来适应新的框架。虽然有很多基于 Action 的 Java Web 开发框架都可以实现这个效果，但我们还是会推荐 Struts 2。因为他更先进、高度解耦、功能丰富且更易于维护。他继承了 Struts 1 的核心理念，但在架构和 API 设计上要比 Struts 1.x 进步很多。此外，其他的替代框架也不错，如 [Spring Web MVC](#)、[Grails](#) 或 [Stripes](#)。

如果想继续对 Struts 1 进行维护，我们可以做些什么？

可以随意的为 Struts 1 进行贡献。有两种方法：从现有代码建立分支并进行改进或吸引社区人士来继续推动 Apache Struts 项目。如果有足够的人愿意并且有能力来继续提供补丁、进行维护以及长期管理的话，我们认为有这样的支援者的情况下，或许 Struts 1 还有希望。

此消息一出，微博人士也纷纷感慨：

蒼氳：还记得进公司的第一件事就是把纯 JSP 的实现的功能用 Struts1 实现，好怀念啊.....感谢 Struts1。

猫砂西瓜：忘不了那一屏幕的 form bean。

KDS-黑暗浪子：逝去的总归要逝去，一切向前看。仔细想想，从 03 到 07 年我也用这个 5 年了。

Kaloo2010：面试再问这个就有的说了。识时务的马：怀念一下，当初那个堆砌 ActionForm 的青葱岁月。

此外，为了给读者提供更好的参考，InfoQ 中文站上也为您准备了相关内容：

- [Spring 相关的内容](#)
- [Grails 相关的内容](#)
- [迷你书：Spring 的 IoC 容器](#)
- [迷你书：Grails 入门指南](#)

原文链接：<http://www.infoq.com/cn/news/2013/04/apache-struts1-eol>

相关内容

- [Apache Struts 1 宣告退出舞台](#)
- [快讯：今日 9:45 分将直播 SpringOne 大会“Spring 框架的未来”等主题演讲](#)
- [Spring Framework 4.0 相关计划公布---包括对于 Java SE 8 和 Groovy2 的支持](#)
- [SpringOne 专访：Chris Richardson 谈云时代的 Spring 发展](#)
- [基于 Spring Batch 的大数据量并行处理](#)
- [SpringOne 会前访谈：Josh Long 谈 Spring 发展](#)

热点新闻 | News

由中行 IBM 大型机宕机谈银行系统运维

作者 [郑柯](#)

12月15日[中行 IBM 大型机宕机](#)，系统没有第一时间切换到热备或者异地容灾上，直接影响中行的信用卡支付相关业务，直到4小时之后才恢复服务。由于银行业务的特殊性，对于系统的可用性要求极高，就此事件，我们采访了兴业银行系统分析师周伟然、支付宝应用运维架构师陆惟凯（花名：近南），请他们谈一下对于银行系统运维的一些看法。

InfoQ：作为一名银行金融行业的 IT 技术专家，您认为本次中行 IBM 大型机宕机的体现出哪些问题和教训？

陆惟凯：主要的问题是灾备或大型故障的演练与决策，对于硬件或者机房故障的大型故障，需要有经过验证演练的切换方案来保证切换风险可控。对于故障决策来说是否启动灾备切换是个艰难的决定，不过确实也要能够下决策去切换。其实一切的根源还是在切换方案是否足够可靠、是否经过演练。只要切换风险可控，切换得决策其实不会太纠结。

周伟然：对于本次中行事件，具体原因不了解的情况下不好直接评论。但所谓相关金融系统的运维是一个复杂的系统功能，不能单纯的从 main frame 的稳定性一概而论。设备运行的稳定性也只是整体系统稳定性的很小部分。除了环境保障中包含的网络环境、硬件资源、存储设备、操作系统数据库等基础软件环境以外，应用运行、系统间互操作等事件都可能产生重大影响。而风险是无法完全避免的，这才显示的出灾难备份和应急预案的重要性，最大程度降低风险暴露后的影响是验证应急体系有效性的重要指标。

InfoQ：ITIL 流程是否在您所在的组织中使用？对于类似事故，ITIL 流程的处理应该是什么样子？

陆惟凯：使用，不过不是标准的 ITIL 流程。我们有一个应急响应的 Team 在处理相关决策以及应急事务。对于特别重大的问题会在应急响应 TEAM 内进行决策。

周伟然 我行使用 ITIL。无论是 ITIL 还是各级监管机构，乃是内部风险机构，对于银行应急处理的流程均有严格的要求，基本上是系统分类，根据不同等级重要性提出不同的风险要求。对于重要系统，需要建设完备的灾备体系，建立完善的应急预案 并且需要确保灾备和应急预案的有效性。对此，监管和内部审计通过演练进行确认。 所谓的演练非模拟实际环境的演练，而是在实际的生产环境进行的模拟灾难，各机构对演练的频度和内容均有严格的要求，并且重大演练时，监管官员将进行现场检查 通过各银行每年发出的停业公告可以看到这些演练信息。

InfoQ：在你们的系统中，“桌面模拟演练”和“Call Tree 演练”是如何进行的？

陆惟凯：模拟演练比较少吧。方案定了之后模拟其实都是没问题的，定期的 review 是需要的。演练相关主要是定期组织运维的容灾演练与应急演练以及网购节（双 11 大促）之前的演练。

周伟然：据我所知，在股份制银行或规模以上银行，重要系统演练多以实际生产系统的方式进行，模拟演练主要用于系统正式上线之前的验证，在实际生产运行时并不采用也不符合监管要求。所有实际生产系统，即实际生产后台、实际渠道系统，但限定范围，例如，在演练时，可能关闭网银入口，使用户无法直接登录，控制演练本身造成的二次风险。

InfoQ：相对互联网行业来说，银行金融行业的 IT 运维人员的素质和技能具体有哪些不同？

陆惟凯：个人感觉是比较接近的。可能是我在支付宝工作的缘故，IT 相关企业的运维人员根据企业的性质不同（门户，电商，游戏，SNS）等会有一些各自有特色的容灾以及流控方案。所以需要相关的运维人员更多的了解前端业务，能够根据不同的故障情况进行不同的处理。（例进行功能的删减控制，流量开关，流量切换等）。另外 IT 企业运维人员遇到的外部故障会更多一些比方外部攻击，或运营商，或应用异常出现的故障。。另外传统 IT 业的系统更新频率会比金融业快上很多。相关应用发布带来的一些故障处理也会对运维人员提出更高的需求。传统金融行业的容灾方案相对来说就比较简单一些。在数据备份方面 IT 企业根据企业特性不同，数据备份的重要性也会不同。金融行业对可用率以及数据备份的要求会更高。

周伟然：由于不太了解互联网的运维素质所以不好比较。但对于金融行业运维，制度性准确性和规范性是很重要的。由于银行设计大量资金和重要隐私，

在制度规范上有着较为严格的规定，例如业务、研发人员与生产系统严格分离、生产数据完全无法接触的到、需要检查分析时需要通过严格的审批流程。在研发软件下发生产也必须严格进行内容审查和审批，操作步骤必须清晰描写，而对于运维把控的是对于审批结果的执行，精确执行审批结果而不能自行改动丁点，而且执行过程被记录，可被审计 在风险发生时，则应依照预案进行各项操作。运维人员对于应急预案的制定的维护，需要基于大量运维经验，并且通过不断优化验证的。

InfoQ：能否介绍下：在您所在的组织中，关键业务系统的备份是怎么做的？

陆惟凯：同城容灾加异地灾备吧..同城容灾包括机房内单点容灾（备份）以及机房间的相互备份。

周伟然：备份方式对于重要系统均需多方面考虑，例如某关键系统，首先在运行时就使用应用集群的方式确保可用性，通讯接入采用端口和地址复用进行多重备份。运行体系基本需要确保无单点故障，即单一功能点在 2 个或以上并行运行的节点。其他设备采用热备或冷备方式。该数据库备份基于数据库引擎和高端引擎进行远程灾备同步的功能，为单数据源热备份，数据的保存备份对于非监管要求数据，根据内部管理规定制定备份保存时间，备份至专用数据平台、对于监管要求的数据，在一定时间内在线保存至数据平台，长时间后转磁带长期保存。

InfoQ：在网友评论中看到一句话：“最关键的是一般都是只有设备容灾，没有人员组织架构的容灾。”请问您觉得“人员组织架构的容灾”应该如何理解？

陆惟凯：人员组织架构的容灾分两部分来看，一部分是操作以及一线的处理人员的备份，这块要保证相关的运维的操作技能与权限到位，在第一联系人没有联系到的情况下可以联系第二联系人来进行处理。

第二是决策人员的备份对于决策的人员存在联系不上的情况下，可以联系备份决策人员来进行决策。

当然这里的人员组织架构容灾基本还没有考虑到一个异地或者其他成分，如果遇到毁天灭地型的地震或者更极端的灾难的时候，可能会缺乏异地的人手来处理问题。。

周伟然：人员组织的架构在银行来说有着明确的规定。首先对于每个系统对应的负责人员需要报送管理，并且做到 A、B 角等多角定义，在系统故障和

重大事件保障时均遵循流程对应具体人员。日常工作时，大家对 ab 角等也有一定的注意，例如某集体全体不宜同一趟飞机出行等来降低风险。

InfoQ：能否介绍一些国外银行金融企业对类似问题和事故的处理经验？

陆惟凯：没有相关的经验。

周伟然：处理经验其实之上各题中均有提到，即功夫在平时。好的应急预案和备份需要大量前期工作和定期优化维护，并且验证，每次处理之后通过仔細的分析、审计、故障报告等方式探讨不足，不断地优化和改进。

原文链接：<http://www.infoq.com/cn/news/2013/04/BOC-Downtime>

相关内容

- [由中行 IBM 大型机宕机谈银行系统运维](#)
- [IBM Workload Deployer—更成熟的私有云设备](#)
- [敏捷、转变、成长——IBM WebSphere 软件全面升级，全新举措加速敏捷进程](#)
[MySQL 运维感悟](#)
- [“软件创新”引领大数据和分析、企业移动、智慧城市等 IBM2013 业务重点](#)
- [IBM 软件公布最新在华战略，以“软实力”启商业新局](#)

特别专题 | Topic

曾经，测试在开发过程中只是“二等公民”，得不到足够的重视，在遇到时间紧任务重的时候，是时间被压缩的首选环节。

然而，随着业界对软件质量要求的不断提高，测试的重要性也不断提高。特别是随着敏捷开发的发展，测试已经渐渐融入到开发环节之中，出现了自动化测试、测试驱动开发等技术，已经成为很多开发人员必备的知识。

在本期专栏中，我们期望可以借助多篇文章，从多个不同的角度来谈“测试”这个话题，既包括企业系统集成测试、如何更好地实施敏捷自动化测试、如何构建最适合的自动化测试基础设施三篇注重实践的文章，也包括如何发现软件测试中的“黑天鹅”这篇更偏重于理念的文章。希望开发团队和测试团队都能从这些文章中有所收获，更好、更有效地完成测试工作，从而保证软件项目或者产品的质量。

特别专题 | Topic

自动化测试基础设施（一）——为功能测试构建通用 mock server 系统

作者 [余朝晖](#)

mock 在单元测试中已经众所周知。现今我们有各种功能强大而又好用的 mock 框架，可以很方便的解除单元测试中各种依赖，这大大的降低了编写单元测试的难度。而测试驱动开发 (TDD) 更进一步将 mock 作为一种设计手段，来辅助识别出元素之间交互的接口和职责。

那么在功能测试(这里提到的功能测试指的是用户级测试)这个层次，是否有必要使用 mock 呢？如果有必要又将如何构建呢？或者说是否有可能像单元测试中那样构建一个通用的 mock server 系统呢？本文将根据我的实践经验，向大家介绍一个通用 mock server 系统的主要组成部分以及设计思路。

Why

现今的业务系统很少孤立存在，它们或多或少需要使用兄弟团队或是其他公司提供的服务，这为我们的联调和测试造成了麻烦。对于这种情况，我们常见的解决方案是搭建一个临时的 server，模拟那些服务，提供数据进行联调和测试。这就是 mock server 的雏形。一般来讲，搭建这种 mock server 系统比较简单，不过它的功能也比较简单，而且往往需要针对不同的接口重复开发。那有没有可能像单元测试中使用的 mock 框架那样构建一个通用的 mock server 系统呢？

How

观察单元测试中的 mock 框架，我们会发现一般使用 mock 的流程是：

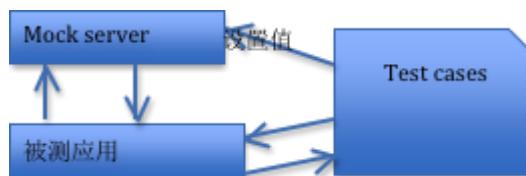
```

init mock //创建mock对象
config mock //设置mock期望
setup mock //将mock对象设置给被测对象
call //调用被测接口，被测接口里的代码会调用mock对象
verify //验证
拿mockito举例：
User expected = new User("admin", "12345");
//init
UserDAO dao = mock(UserDAO.class);
//config
when(dao.findByName("admin")).thenReturn(expected);
//setup
UserService service = new UserService(dao);
//call
User actual = service.login("admin", "12345");
//verify or assert
    
```

借鉴这种做法，我么就可以构建一个简单的 mock server 系统。接下来的内容中，我们会在这个 mock server 的基础上演化出比较完善的版本。

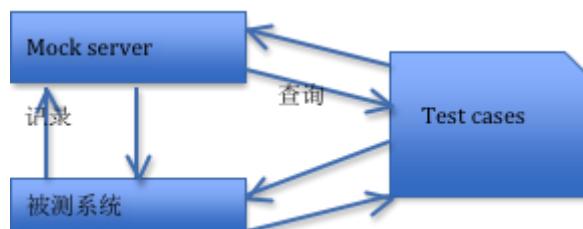
版本 1(简单的模拟值)

假设我们需要 mock 的是 HTTP 接口。我们的 mock server 提供一个配置接口(对应着上面的 config mock 步骤)，测试运行之前调用配置接口将需要 mock 的 HTTP 接口 URL 以及需要返回的值传递过去，mock server 内部建立 url 到返回值的关联(这里就类似存在一个哈希表一样)。Mock server 还提供另外一个接口供被测系统调用。这是一个通用的接口，所有原先指向真实服务的地址全部被指向到该接口(可以通过修改配置或修改系统 hosts 文件)。当该接口被调用时会寻找刚才建立的 url 到返回值的关联，并将 mock 的值返回。这样一个非常简单的 mock server 就构建出来了，对于一些简单的联调场景基本够用。这个时候我们的 mock server 的原理图如下面所示：



版本 2(提供调用参数的查询)

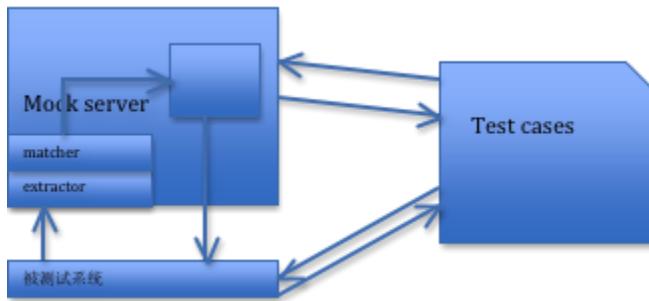
有了第一版的 mock server，对一些只需要模拟值的场景是够用了。但是，mock 的作用仅限于此么？想想单元测试中的 mock。在单元测试中 mock 框架除了能够为被测系统构建输入值外，还能捕获到被测程序传出的值，然后对这些值进行校验。举一个实际的例子：在我们的系统中经常需要向用户发送短信，为了对发送的短信功能以及短信内容进行验证，在没有 mock 之前我们可能真的需要向真实手机发送短信，然后验证。这不仅降低了测试效率，也增加了不少不可控因素。为此我给 mock server 添加第三个接口：query 接口。在被测系统调用 mock 接口时，mock 会记录下被测系统传递给 mock 接口的参数值，然后测试中可以调用 query 接口查询到记录的值，在测试中可以对其断言，而且这里记录下的调用记录还可以作为日志提供出来，提高系统的诊断能力。这样我们的 mock server 的结构就变成：



版本 3(可以根据参数模拟)

现在我们的 mock server 已经可以提供类似 verify 的功能了，但实际上它还不能算一个真正的 mock。它也不能处理哪怕稍微复杂一点的情况。在实际中，我们经常需要针对不同的参数返回不同的值。举个简单的例子：我们 mock 一个支付接口，对于订单号 123 我们期望支付成功，对于订单 234 期望支付失败。这就引入了我的 mock server 的两个核心组件：extractor 和 matcher。Extractor 组件主要用于从调用的参数上提取出参数值，然后转换成 key/value 的格式提供给后续的环节使用；因为请求的参数格式多种多样(json, xml 等)，extractor 为此提供了统一的接口。

Matcher 组件的作用是在拿到 extractor 传递过来的 key/value 值后利用一些匹配器匹配到具体设置的期望上，所有匹配到的调用会返回对应的值。也就是说 mock server 内部不再是简单的映射了。后面再介绍 DSL 部分的时候会介绍 matcher 使用的语法。这样我们的 mock server 结构就演化成下图：



版本 4(提供多种协议的支持)

估计有人在抱怨，说了这么多这个 mock server 还只能 mock HTTP 接口啊，我们的系统中存在 HTTP 接口，RPC 接口，SMTP 接口等等。这是 mock server 中协议组件的职责。协议组件是 mock server 的入口，它提供多种协议的服务，并且解析出协议包数据，然后将数据交给 extractor 组件；除此之外，协议组件在收到上层的返回值后，会按照协议的格式返回给被测系统。利用一些开源的类库，我们可以很容易对一些通用协议提供支持，但对一些私有的二进制协议如果没有现成的库支持，要重新开发成本很大，不过我们可以从客户端来解决这个问题，这在后续的文章中会有介绍。

版本 5(模拟行为)

基本上一个功能还算完善的 mock server 成型了。但这就够了么？对于要模拟各种场景的测试还远远不够。我们很多接口有回调的功能，我们通常还需要模拟接口超时的情况，而对于一些支付相关的接口经常需要对参数进行加密解密，而且这些情况都需要是可配置的。有没有发现，前面我们介绍的所有实际上都是 mock 值。也就是我们设置一些值，然后调用的时候将值返回。但是很多时候我们不仅需要 mock 值，更要 mock 行为。这样我们有了 mock server 中最核心的组件：命令执行引擎(好牛的名字，其实就那样)。在设置 mock 的时候我们不再是设置一个值，而是设置一个预定义命令组合成的流水线(即按照类似下面 xml 的配置一步一步执行，并且可以将上一步的执行结果传递给下一步)：

```

<delay>1000</delay>
<callback url=http://localhost/callback.do>{"ret":"true"}</callback>
<return>{"ret":"true"}</return>

```

上面的流水线被命令执行引擎解析执行后就是按顺序执行对应的 DelayCommand, CallbackCommand 以及 ReturnCommand 命令了，具体命令就不介绍了。采取这种方式给我们 mock server 带来了很大的灵活性：只需要简

单的扩展一个子命令，就可以扩充 mock server 的行为。比如 mock 某网关接口时需要使用 MD5 加密，只需要扩展一个 MD5Command(下面代码中的\$result 表示前一步骤 <md5 /> 加密后的结果)：

```
<md5 />
<return>$result</return>
```

DSL

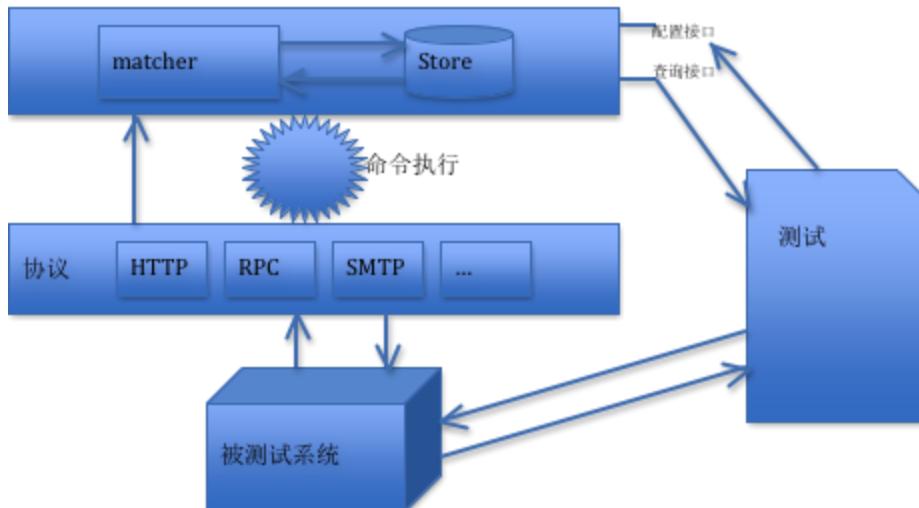
现在我们的 mock 不仅可以 mock 值了，对于各种行为的模拟也得心应手。但是要使用方便，还要提供便于使用的接口。Mock server 提供两类接口：针对自动化测试的 DSL，以及针对手工测试使用的管理界面。这里主要介绍这种 DSL(因为我们的测试用例是使用 xml 编写，转换成编程语言也很容易)：

```
<mock service="http://test.json" matcher="hasKey($param.orderNo)">
  <delay>1000</delay>
  <md5 />
  <return>$result</return>
</mock>
```

service 是对被 mock 的服务的描述，比如对于 SMTP，我们可以这样定义：service="smtp:9000"。这个表示在 9000 端口上监听 smtp 协议。而 matcher 即前面介绍的 matcher 组件所使用的各种匹配器，用于匹配被测系统调用 mock server 时传递的数据。比如上面的例子表示的就是如果被测系统调用 http 接口 /ticket.jsp，并且参数里包含 orderNo 则延迟 1 秒钟，然后返回一个 json 值。

总结

前面几节介绍了一个比较完善的通用 mock server 从简到繁演化的设计思路，希望可以为想要构建类似设施的读者提供一个参照。



这个 mock 系统包含两个主要部分：mock admin 和 mock server。Mock admin 是管理界面，主要提供监控(可以在界面上实时看到被测系统与 mock server 交互)以及手工测试时的配置界面。Mock server 即前面介绍的主体，其架构如上图所示。Mock server 包含几个核心组件：协议、extractor、matcher、命令执行引擎、存储(即 mock server 中使用的各种数据的存储)。Mock server 提供三类接口：配置、被 mock 接口(各种服务，通过协议组件提供)、查询。

原文链接：<http://www.infoq.com/cn/articles/auto-test-mock-server>

相关内容：

- [Robot Framework 作者建议如何选择自动化测试框架](#)
- [在敏捷项目中实施自动化测试之我见](#)
- [Android 自动化测试解决方案](#)
- [敏捷自动化测试（2）——像用户使用软件一样享受自动化测试](#)
- [敏捷自动化测试\(1\) —— 我们的测试为什么不够敏捷？](#)
- [不喜欢绘画的木工不是好测试人员——QCon 北京测试专题讲师王东刚访谈](#)

特别专题 | Topic

软件测试中的黑天鹅（二）：黑天鹅发生的前后

作者 邵晓梅

1. 历史与三重迷雾

在“[认识软件测试中黑天鹅](#)”一文中，我描述了什么是软件测试中的黑天鹅及其特点，本文将探讨测试中的黑天鹅发生之前、之后、以及正在发生之中的故事。《黑天鹅》一书的作者 Nassim 指出“历史是模糊的。你看到了结果，但看不到导致历史事件发生的幕后原因。”其实，测试何尝不是这样，假如把测试看成一个盒子，这个盒子也是模糊的，你看不到盒子里面是什么，整个机制是如何运行的。书中描述：“对待历史问题，人类思维会犯三个毛病，我称之为三重迷雾。他们是：

1. 假想的理解，也就是人们都以为自己知道在一个超出他们认知的更为复杂（或更具随机性）的世界中正在发生什么。
2. 反省的偏差，也就是我们只能在事后评价事务，就像只能从后视镜里看东西（历史在历史书中比在经验现实中显得更加清晰和有调理）。
3. 高估事实性信息的价值，同时权威和饱学之士本身有缺陷，尤其是在他们进行分类的时候，也就是进行‘柏拉图化’的时候。”我很容易联想到，这三重迷雾分别对应测试中的黑天鹅发生之前、之后、以及黑天鹅形成之中的故事，即：发生之前的“盲目预测”、发生之后的“假想解释”、以及黑天鹅形成之中的“柏拉图化”。

2. 黑天鹅发生之前的“盲目预测”

“第一重迷雾就是我们认为我们生活的这个世界比它实际上更加可理解、可解释、可预测”。打开收音机或电视机，你就会听到或看到，每天都有无数的人在信心满满地预测着各种各样的事情：股市的走势、房价的走势、战争是否会爆发、疾病是否会流行……

正如 Nassim 指出的那样，“几乎所有关心事态发展的人似乎都确信自己明白正在发生什么。每一天都发生着完全出乎他们预料的事情，但他们就是认识不到自

己没有预测到这些事。很多发生过的事情本来应该被认为是完全疯狂的，但在事情发生之后，看上去就没有那么疯狂。这种事后合理性在表面上降低了事件的稀有性，并使事件看上去具有可理解性。”就拿我所生活的这座城市-上海来说，近来的许多事件都在证实着这点：黄浦江上打捞出数千头病死猪、H7N9 的流行，昨天突然看到一条“上海自来水中添加了 XX”的微博更是令人触目惊心，我们内心都预测这些事情不应该发生，可是实际上却发生了。

这种黑天鹅发生之前的“盲目预测”让我想到软件测试中版本发布之前的“测试评估(test evaluation)”。一个产品经过测试团队的集中测试后，发布到用户那里，谁能准确预测是否会出现“黑天鹅”呢？在你的团队，在版本对外发布之前，是否需要测试团队填写一个关于产品质量的测试评估表？下图是测试评估表的一个样例。

特性 ↗	质量评估 (A/B/C/D) ↗	质量评估说明 ↗	测试 投入 充分 性 ↗	关键遗留问题和风险 ↗	后续测试 策略 ↗	是否适 宜发布 (Y/N) ↗

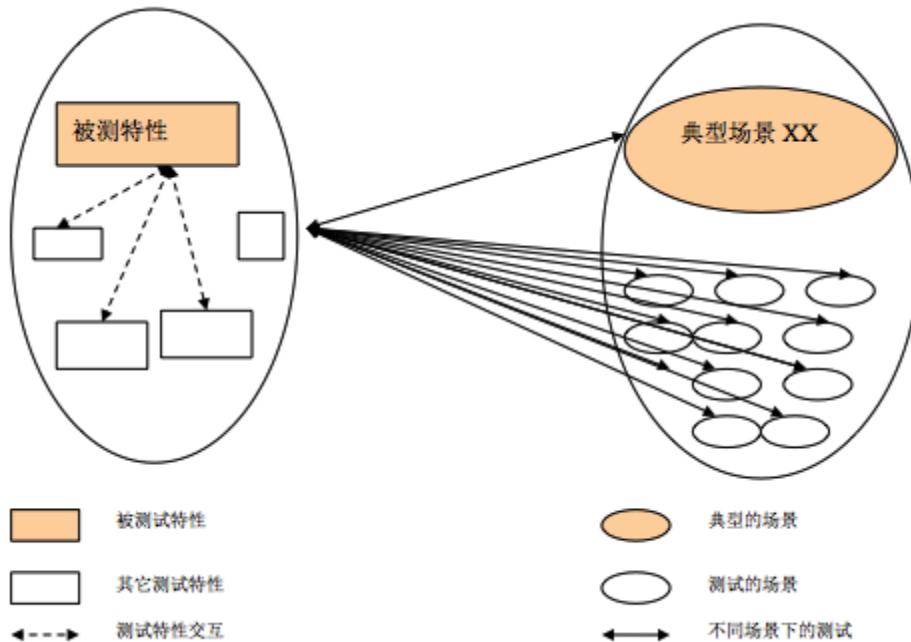
这里的特性指的是“测试特性 (test feature) ”。根据各团队上下文的不同，这个测试特性可能与开发特性直接对应，也可能不与开发特性一一对应，每个测试特性对应一条或多条需求（用户需求或系统设计需求）。我最常看见的质量评估说明是“XX 特性基本功能正常。”有些人在后面附上所发现的比较严重的 bug。这样的描述显然并不令人满意。

问题是：

每个特性质量的 A/B/C/D 的结论是否准确？所有特性的 A/B/C/D 的结论加一起又如何判别整个系统的质量？是否所有特性的质量都是 A 或 B，版本就可以对外发布，并且发布以后不会出现令人意想不到的“黑天鹅”现象？测试人员给出的 A/B/C/D 有多大成分是基于一种 feeling 给出的？

对于任何一条需求，开发人员的任务就是实现它，无论是由一个项目组实现，还是多个项目组配合实现。但是，对于测试人员，考虑的事情就复杂一些。我们除了要验证这条需求本身的功能实现是否正确，还要验证该需求和其它功能之间的交互，还要考虑客户可能使用的各种场景（ Scenario ），包括各种组网场景、各种参数的配置等。如果把测试场景和测试特性交互起来，测试就无穷无尽了，并

且也没有必要在每一种测试场景下，都验证被测特性的基本功能、异常功能、与其它功能的交互、非功能属性等各方面。如何设计更有效的、有限数目的用例尽量做到最大的测试覆盖，这属于另外一个话题，本文不做探讨。



毫无疑问，尽管测试设计人员和测试执行人员精疲力尽的试图覆盖该特性的所有可能的场景，测试人员仍然只覆盖了一小部分的场景下的该特性的部分测试用例，如果没有 Fatal 的遗留问题，是否该特性的质量就可以评价为 A 或 B，并且认为可以对外发布了呢？这种测试评估工作，与其说是让测试人员对被测对象的质量进行测试评价，不如说是让测试人员对被测对象进行质量预测，因为测试人员所了解的只是部分信息，而不是全部。Nassim 说“在这些错误的预测和盲目的希望中，有一些愿望的成分，但也有知识的问题。”我更愿意相信，测试人员对产品质量的预测主要是“知识的问题”，毕竟完整测试是不可能的。既然测试无法做到完整覆盖，不如在有限的时间和资源内，尽可能覆盖典型的场景，测试评估中针对已经覆盖到的典型场景进行评估，具体地，James Bach 和 Michael Bolton 在 RST (Rapid Software Testing) 课程里提出的三步法可以供参考：

- 测试中发现了哪些问题；
- 做了哪些测试活动发现了这些问题；
- 测试活动本身的有效性如何，哪些地方可以改善。

那么对于典型场景之外的其他场景又如何评估呢，这与测试评价之前的那些测试活动息息相关，也就是与黑天鹅正在形成之中的那些故事有关，我们在后面第四部分再探讨。

3. 黑天鹅发生之后的“假想解释”

不管怎样，那些你认为不应该发生的“黑天鹅”经常如期而至。很多组织要求缺陷发生后开展缺陷根因分析(RCA)(我总是在尽力避免使用“缺陷回溯”这个词)。尤其对黑天鹅这样的重要的 bug，更要仔细开展缺陷根因分析了。对黑天鹅开展 RCA 的目的是利用这个黑天鹅，挖掘它带给我们的信息，从而尽可能在以后的测试中发现更多类似的缺陷，对开发而言则是在以后尽可能避免引入此类的 bug.

RCA 的目的是做到基于缺陷的过程改进，而不是解释黑天鹅的发生这个动作本身。不要试图去解释所有的黑天鹅，尤其是那些在实验室很难重现的黑天鹅。我看到有些测试团队，当黑天鹅发生了很紧张，又是一个严重的 bug 漏测了，赶紧组织人力调查分析，尽快写出一个看起来像样的缺陷分析报告。而阅读这个 RCA 报告，很难从中找出真正有力的措施可以有效避免今后这类 bug 不再漏测。举几个现实中的“RCA 现象”：RCA 报告中，错误的原因有“人为错误”、“沟通不畅”、“缺乏相应的测试用例”等；避免漏测的措施有“加强代码走读”、“加强白盒测试”、“提高测试设计能力”、“加强与开发人员的沟通”等。Nassim 发现，“我们的头脑是非常了不起的解释机器，能够从几乎所有事物中分析出道理，能够对各种各样的现象罗列出各种解释，并且通常不能接受某件事是不可预测的想法。”

我曾对多个团队进行 T-RCA (我提出的一个缺陷根因分析的方法) 引导，发现一个有趣的现象，尽管各个团队所测试的产品是不同的，但是他们 RCA 分析的结论却是惊人的相似，很多团队都存在我上述所列的“RCA 现象”。我分析原因大致有二，一是没有找到有效开展 RCA 的方法，没有找到缺陷发生的根本原因；二是很多团队使用了比较“细致”的 RCA 模板，模板里对每一项可能的情况进行了细致的分类，比如该缺陷所处的测试级别、涉及的测试活动、所属的测试类型、可能的原因分类等等。缺陷根因分析工作仿佛变成了测试人员只要对照模板逐一打钩去筛选就可以了，但实际上 RCA 是个高度探索性的过程，需要与缺陷相关的各干系人去沟通，需要从纷繁复杂的种种因素中创造性地找到改进的措施。面对着电脑，填写那些 RCA 模板中的空白项不是最主要的工作。实际上，某种程度上讲，过细的 RCA 模板简化了缺陷分析过程，掩盖了缺陷根因分析过程的复杂性，也比较容易导致分析结果的雷同。《黑天鹅》里的这句话也许可以给我们

更多启示：“我们对周围世界的任何简化都可能产生爆炸性后果，因为它不考虑不确定性的来源，它使我们错误地理解世界的构成。”

4. 黑天鹅形成之中的“柏拉图化”

既然，测试中的黑天鹅的发生是个经常性的事件，那么测试的过程不也正处于黑天鹅形成的过程吗？想象一下，当前我们正在测试一个产品，我们了解黑天鹅理论，我们知道这个产品发布给用户后极有可能会冒出“黑天鹅”来，那么我们当前可以采取什么措施呢？

我在“认识软件测试中黑天鹅”一文中解释了什么是“测试的柏拉图化”：只注重外在的形式、尤其是针对具体明确的事情进行简单分类的时候，犯“柏拉图化”错误的人容易高估他们已经掌握的事实性信息的价值，而对大量的他们还不知晓的并且非常重要的信息视而不见。我是在 2006 年发现这个“我所不知道的大量信息的”，也同时发现了之前我所犯的“测试的柏拉图化”的错误，即特别重视诸如“测试用例设计个数、测试用例执行个数、发现的 bug 数”等这些数据，也许在这些明确的、外在的形式之外，有更多值得我们思考的东西。

那一年，我负责一个特性(大体就是在手机上通过蜂窝小区广播的形式收看电视) 的测试，这个特性是个全新开发的特性，我是第一个、也是当时唯一一个测试人员，测试任务很紧张，因为这个特性已经定于一、两个月以后在香港某个运营商网络里首次商用。像大多数测试人员一样，我很辛勤地、按照既定方法和策略测试这个特性，在有限的测试时间里：我熟悉了这个特性、设计了很多测试用例、执行了大量的测试用例、发现了很多 bug、对这个特性相关的每一个 bug 都了如指掌、通过和开发人员不断地交涉和定位 bug 还对该特性当前存在的缺陷非常清楚、熟悉这个特性的代码和问题定位手段。当我奔赴香港作为测试人员辅助开通这个特性的商用之前，研发团队解决了所有严重的缺陷，我很有信心应对各种可能的状况，我几乎认为我对这个特性的了解已经很完整了。

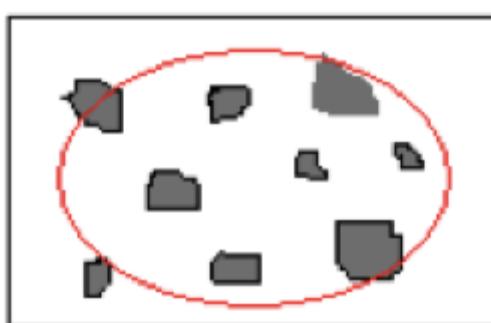
可是，当我抵达用户那里，看到我们的产品在真实网络中的运行环境、配置、使用方式，与用户和当地技术支持人员的各种交流，坐在地铁里看到身边的人员拿着手机正在开启使用我刚刚参与开通商用的特性，从真实网络环境中获取的大量错误报告和跟踪消息和后台日志。。。短短几天内，有关这个特性的、以前我所不知道的、大量的信息冲进了我的大脑，我像突然捡到宝藏一样地忙着分析日志、记录 bug，生怕遗漏了哪个 bug 没有记录，然后回到实验室都不知道如何触发。

我发现了一个重要的事实，这一回，我只用了几天时间，没有设计任何测试用例，没有执行任何测试用例，却在一个我几乎认为没有什么严重缺陷的特性上，“不费吹灰之力”似的又发现了很多严重的缺陷。而且这些缺陷不是实验室触发的，而是就发生在用户的身上，有些遭到用户的投诉，有些用户还不知晓，换句话说，这些缺陷都是优先级很高的非常重要的缺陷。

从香港回来，我一直在思考，测试团队如何发现这些重要的缺陷？甚至如何让平常我们在实验室中的测试也能这么高效、这么有效？您可能已经猜到了，这就是后来备受重视的 UBT (Usage Based Testing)，或者有的行业叫做 TiP (Test in Production)，也有人称为 Testing-in-the-wild。

再回到前面曾提到的一个关于测试评估的问题上：假如平常的测试更关注典型场景的测试，那么对于非典型场景如何测试以及如何评估呢？我想 UBT 是个不错的选择。既然无法做到全覆盖的测试，就不去做，不要试图在现有的测试模式下，测试设计和测试执行都投入很大精力去覆盖各种场景和交互，因为这样做收效甚微，依然达不到目的。平常的功能测试只做最普通、最典型、最重要场景下的功能验证，保证每个测试特性的基本功能 OK。此外，还要开展 UBT 或 TiP，主要考虑各种配置和场景，用模拟器模拟真实商用组网环境和业务模型，或者直接在用户使用产品的真实环境中开展测试。

假如被测系统如下图的方框，灰色的小块就是我们平常的功能测试覆盖，可能很多测试团队的做法是试图尽最大能力增加这些灰色小块的覆盖，但依然会有很多覆盖不到的地方。而 UBT 就相当于红色的范围，虽然没有针对性的设计测试用例，但由于模拟了可能使用的商用场景和业务，业务之间交互的测试在这种测试环境下自动进行，潜在的一般的 bug 都被自动发现（比较难触发的异常 bug 依然发现不了），如果这样的 UBT 测试连续执行几天或数周都没有问题，此时测试的评估中就可以很有信心地写到：在 XXX UBT 环境下连续运行 XX 天，没有发现严重问题。这样，也大大减少了版本发布后“黑天鹅”出现的几率。



5. 结论

如果你认同“测试的黑天鹅”就在我们身边，那么：

- 在“测试的黑天鹅”发生之前，不要在信息不完整的情况下对全局做“盲目预测”，因为你只掌握了部分信息，你要做的是更准确地陈述这些“部分信息”；
- 在“测试的黑天鹅”发生之后，不要试图对所有黑天鹅都做“假想解释”，更重要的是从已经发生的黑天鹅身上挖掘更有价值的信息，以减少更多类似黑天鹅事件的发生；
- 在“测试的黑天鹅”形成过程之中，不要犯“测试的柏拉图化”错误，重视你知道的信息，更重视那些你所不知道的大量的更有价值的信息。

References

Nassim Nicholas Taleb, The Black Swan: Second Edition: The Impact of the Highly Improbable, 2008

原文链接：

<http://www.infoq.com/cn/articles/before-after-the-occurrence-of-black-swan>

相关内容：

- [采访和书评：Google 如何做测试](#)
- [2012.4.25 微博热报：测试用例、硅谷热点](#)
- [敏捷测试推进工作笔记（一）](#)
- [百度技术沙龙第 18 期回顾：大型网站的性能测试实践及结果分析（含资料下载）](#)
- [软件测试魅力何在——QCon 北京测试专题出品人高楼访谈](#)
- [不喜欢绘画的木工不是好测试人员——QCon 北京测试专题讲师王东刚访谈](#)

特别专题 | Topic

让断言不再成为自动化测试的负担

作者 [殷坤](#)

在本系列的第一篇文章“我们的测试为什么不够敏捷”中，根据实例总结出敏捷自动化测试的两大阻碍：“脚本维护困难”、“断言条件繁琐”。本文针对在不失自动化测试有效性的前提下如何降低断言成本来分享一些实践经验。

目前业界常见的自动化测试工具在断言方面大多都是采用“指哪儿打哪儿”的细粒度模式，即，在自动化测试过程中只能对设置为断言条件的字段（页面元素）进行断言。而且这些测试工具即使提供录制脚本的功能，但对于断言往往还需要测试人员手工补充插入。

除了补充、维护断言条件的工作量大之外，这种断言模式还存在一些明显的不足，下面结合一个实际的例子（如下图）进行分析：



The screenshot shows a software interface for managing users. On the left is a sidebar menu titled "菜单树" (Menu Tree) with the following structure:

- console
 - 工作流
 - 组织机构
 - 用户管理
 - 管理员
 - 业务用户
 - 用户信息
 - 密码修改
 - 角色管理
 - 岗位管理
 - 维度
 - 资源管理
 - 安全管理
 - 报表

On the right is a table titled "用户列表" (User List) displaying the following data:

	员工编号	姓名	部门	入职日期	职位
1	250	杨作仲	财务部		项目经理
2	319	赵斌	财务部	2008-03-19	软件工程师
3	216	陈旭杰	开发部	2008-03-19	软件工程师
4	100	张卫滨	销售部	2008-03-19	RIA主架构师
5	10000	赵磊	销售部	2008-03-19	产品经理

Below the table are navigation buttons: 前 (Previous), 后 (Next), 第 (Page 1), /1页 (1 page), New, Delete, Save, and status information: 本页共5条记录 共5条记录.

- 无法感知未设为断言对象的字段上发生的错误

以上图为例，如果在“增加用户”的测试脚本之后只对列表中的“用户名是否存在”进行断言，那么就可能遗漏“入职日期没有提交成功”的错误。而且由于页面的信息量往往很大，我们是不可能对所有字段都设置为断言条件的。

- 难以对于 UI 样式或 UI 逻辑进行断言

以上图为例，有一个 UI 样式类的缺陷（左侧菜单树的根节点“console”下面多出来一条虚线）和一个 UI 逻辑类的缺陷（右侧用户列表只有一页，但“下

“一页”和“最后一页”图标依然可以点击的，即没有灰显）。此类缺陷即使对于经验丰富、心思缜密的测试人员，在人工测试时也是很可能发现不了的，并且在自动化测试过程中也很难进行断言。

即使存在上述问题，测试脚本中是否有充分的断言，依然是评判自动化测试有效性的一个重要指标。但实施过自动化测试的人应该都会有这样的体会：“大部分断言在大部分情况下只是佐证软件是运行正常的”。

当然，所有人都应该是非常期待看到这样的结果，毕竟谁也不希望每次回归测试时都是用例大面积不通过。只是辛辛苦苦写这些断言语句的测试人员心里未免有些“小遗憾”。

[本系列上篇文章](#)中谈到“很多人一提到自动化测试脚本，马上就想到需要提供录制工具”，但如果换个角度思考，很可能就是“柳暗花明又一村”。

在这里，我们同样换个角度思考，假设我们的自动化测试主要目标是为了证明软件运行正常，那么我们会怎么做？

笔者这边的一个经验就是“按照完整的业务流程来组织测试用例，只对少量、必要的关键点进行断言”。以“租户对虚拟化资源的申请使用”为例，来具体看看测试用例的组织方式：

1. 新租户注册；
2. 管理员登录系统，对注册租户进行审批，然后退出系统；
3. 审批后的租户登录系统；
4. 租户申请所需要的虚拟化资源(比如 ,40G 硬盘、2 核 CPU、2G 内存)，然后退出系统；
5. 管理员登录系统，对租户申请的资源进行审批，然后退出系统；
6. 租户登录系统，在已申请资源的基础上创建安装指定操作系统的虚拟机；
7. 断言虚拟机是否创建成功；
8. 租户退出系统；
9. 管理员登录系统，删除租户；
10. 断言租户之前申请的资源是否被完全释放；
11. 租户再次登录系统，断言是否无法登录；

上述测试用例就是按照完整的业务流程进行组织，并且只对少量关键点(7、10、11)进行断言，如果整个用例可以运行通过，就能证明这个业务是没有问题的。

另外还有一个值得考虑的现象，就是相对于自动化测试而言，一个优秀的测试人员在人工测试时是如何判断功能正确与否的呢？他不会死板的只盯着某几个输入域的值，他一定还会同时关注页面上所有数据的正确性、会更加关注业务流程是否正确、会更敏锐的发现页面样式或 UI 逻辑类的缺陷。

为了兼顾“证明软件正常运行”和“人性化的识别软件缺陷”，一个优秀的测试工具应该考虑提供以下多种断言机制。

一、控件级细粒度断言

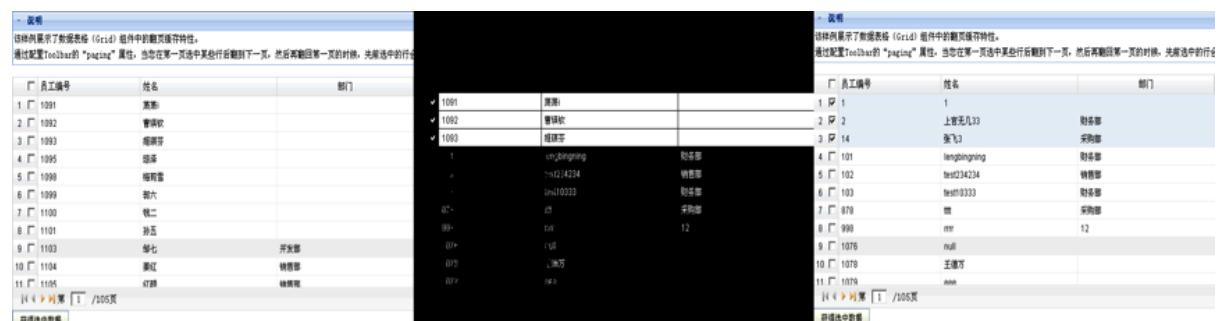
即前面提到的最常见的断言方式。在测试过程中，可以在任何位置增加断言脚本，来判断页面指定控件是否存在、控件显示值是否为预期结果等。通常建议只对关键校验点进行断言。

二、页面级粗粒度断言

通过对比前（之前测试通过）后（后续持续发布）版本在测试用例路径和输入参数相同的情况下，整个页面内容（包括截图和数据）是否严格相同来做粗粒度断言。

通过页面截图进行断言有两个实现要点：首先要选择一个合适的截图方案（笔者推荐采用 [Selenium WebDriver](#) 提供的 TakesScreenshot 接口）；其次需要提供图片对比工具，以便测试人员可以一眼看出两个版本页面截图的差异。

下面是笔者在测试框架中实现的截图自动化对比功能的实际效果。下图中左侧部分是“实际结果截图”、右侧是“预期结果截图”、中间部分是差异对比，测试人员一眼便可看出其中的 Bug：“表格行选中的翻页缓存（在当前页选中几条记录，翻到下一页再翻回本页，需要保持之前的行选中状态）功能丢失了”。



通过页面数据进行断言的实现方式相对简单一些，首先要提取页面上所有的数据（或文本），接着进行格式化，然后再自动化对比。“页面级粗粒度断言”的特点及应用场景如下：

无需编写任何断言语句；

需要能够提供可用于自动对比的历史正确版本，特别适用于可以持续构建的项目；

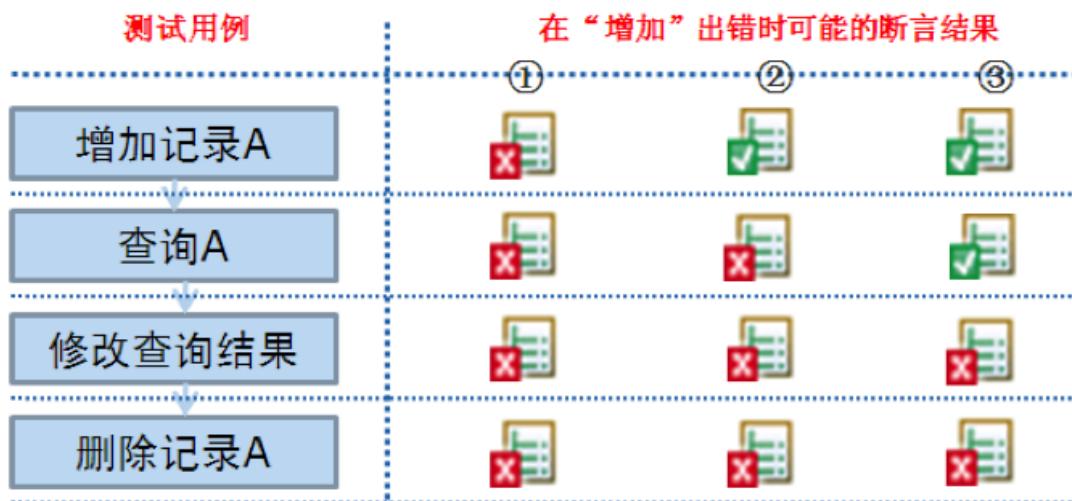
能判断出 UI 样式和 UI 逻辑类的错误；

由于对比绝对精准，导致可能存在误判，因此需要人工对差异图片进行排查；【注】由于很多 Web 页面都有渐入渐出、点击时按钮变色等很炫的效果，所以在两次截图的瞬间可能页面的动态样式是不一样的，这就是所谓的“误判”。笔者对于一个“动态样式”适中的项目采用这种断言方式，统计结果表明误判率在 20% 左右。

鉴于回归测试的时候，通常大部分用例应该是可以通过的，所以“页面级粗粒度断言”的投入产出比非常占优势！

三、基于业务逻辑断言

在测试设计时把有依赖关系的用例一起执行，如果某个步骤出现问题，即便不设置任何断言语句，在当前步骤或后续步骤的测试用例也会执行失败。下面以“增加、查询、修改、删除”这个最典型的流程来说明（如下图所示）。



假定在“增加”环节出现问题，那么我们的测试用例执行情况可能出现如下几种结果：

1. 如果在“增加记录 A”的用例中包含对是否增加成功的断言，那么测试用例从“增加记录 A”开始出错；
2. 如果在“增加记录 A”中不包含断言，而是在“查询 A”的用例中包含是否有查询结果的断言，那么测试用例会从“查询 A”开始出错；
3. 如果在“查询 A”中也不包含断言，那么测试用例会从“修改查询结果”开始出错。

所谓“基于业务逻辑断言”，就是指上述第三种情况，其特点及应用场景如下：

- 无需编写任何断言语句，但测试设计要考虑业务逻辑顺序；
- 与“页面级粗粒度断言”相比，不需要提供可用于对比的历史正确版本，通常适用于项目刚开发或样式做整体调整等情况；
- 断言错误的位置不精准，可能延后；
- 执行过程每一步都做截图备份（通过 [Selenium WebDriver](#) 可以很方便的实现），可以非常有效的辅助定位准确的出错原因；
- 鉴于回归测试的时候，通常大部分用例应该是可以通过的，所以“基于业务逻辑断言”的投入产出比非常占优势！

四、自定义扩展断言

在人工测试时经常有些操作结果的正确与否在当前页面无法做出判断，需要到其它页面甚至系统外部（比如，数据库、输出日志）获取信息来做出判断。以最常见的“基于数据库进行断言”为例，测试工具需要支持把断言时用到“预期结果”和“实际结果”配置为对应的 SQL 语句。

以上介绍了从测试工具的角度可以提供的多种断言机制，在自动化测试过程中应该根据项目实际情况，考虑采用上述多种断言的组合，以弥补控件级细粒度断言的不足。

本系列文章至此，已经分享了[如何降低测试脚本的编写、维护成本](#)，如何在不失测试有效性的前提下减少断言成本。改善上述两大问题之后，自动化测试基本上就可以比较顺利的开展了。接下来如何让自动化测试的价值最大化呢？答案就是频繁的执行测试脚本。因此下一步要做的就是持续集成（或者称为每日构建）。下一篇篇文章会分享一个由测试团队主导的持续集成案例，敬请关注！

作者简介

殷坤，东软集团资深测试经理、技术讲师，10年软件研发、实施、测试及项目管理工作经验。目前专注于敏捷项目管理及质量控制、过程改善、自动化测试、持续集成、用户体验提升等方面。

原文链接：<http://www.infoq.com/cn/articles/Agile-test-automation-3>

相关内容：

- [敏捷自动化测试（2）——像用户使用软件一样享受自动化测试](#)
- [敏捷自动化测试\(1\) —— 我们的测试为什么不够敏捷？](#)
- [Poang，基于 Node.js 的自动化测试范例](#)
- [在敏捷项目中实施自动化测试之我见](#)
- [获取自动化测试的最大价值](#)
- [社区热议自动化测试发展前景](#)



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**5月31**日前

精彩看点：

- 鹰眼下的淘宝
- Java SE 8的55个新特性
- 使用Vert.x实现JVM异步编程
- 享用甜点时要不要来杯咖啡？Java 与树莓派 (Raspberry Pi)
- JCP.next：重振 Java 标准
- JSR 354：货币单位和货币金额 API
- Java 战略与技术主题演讲

特别专题 | Topic

企业系统集成点测试策略

作者 [熊节](#)

集成是企业应用系统中绕不开的话题。与外部系统的集成点不仅实现起来麻烦，更是难以测试。本文介绍了一种普遍适用的集成点测试策略，兼顾测试的覆盖程度、速度、可靠性和可重复性，为集成点的实现与测试建立一个通用的参考。

背景

本文作为例子介绍的系统是一个典型的 JavaEE Web 应用，基于 Java 6 和 Spring 开发，采用 Maven 构建。该系统需要以 XML over HTTP 的方式集成两个外部系统。

该系统由一支典型的分布式团队交付：业务代表平常在墨尔本工作，交付团队则分布在悉尼和成都。笔者作为技术领导者带领一支成都的团队承担主要交付任务。

痛点

由于需要集成两个外部系统，我们的 Maven 构建[1]过程中有一部分测试（使用 JUnit）是与集成相关的。这部分测试给构建过程造成了一些麻烦。

首先是依赖系统的可靠性问题。在被依赖的两个服务之中，有一个服务部署在开发环境中的实例经常会关机维护，而它一旦关机就会导致与其集成的测试无法通过，进而导致整个构建失败。我们的交付团队严格遵守持续集成实践：构建失败时不允许提交代码。这么一来，当我们依赖的服务关机维护时，交付团队正常的工作节奏就会被打乱。

即使没有关机维护，由于开发环境中部署的服务实例仍在不断测试和调优，被依赖的服务实例也不时出现运行性能低、响应时间长等问题，使我们的构建过程也变得很慢，有时甚至会出现随机的构建失败。

被依赖的服务在开发环境下不可靠、性能低，会使应用程序的构建过程也随之变得脆弱而缓慢，从而打击程序员频繁进行构建的积极性，甚至损害持续集成的有

效性。作为团队的技术领导者，我希望解决这个问题，使构建可靠而快速地运行，以确保所有人都愿意频繁执行构建。

如何测试集成点

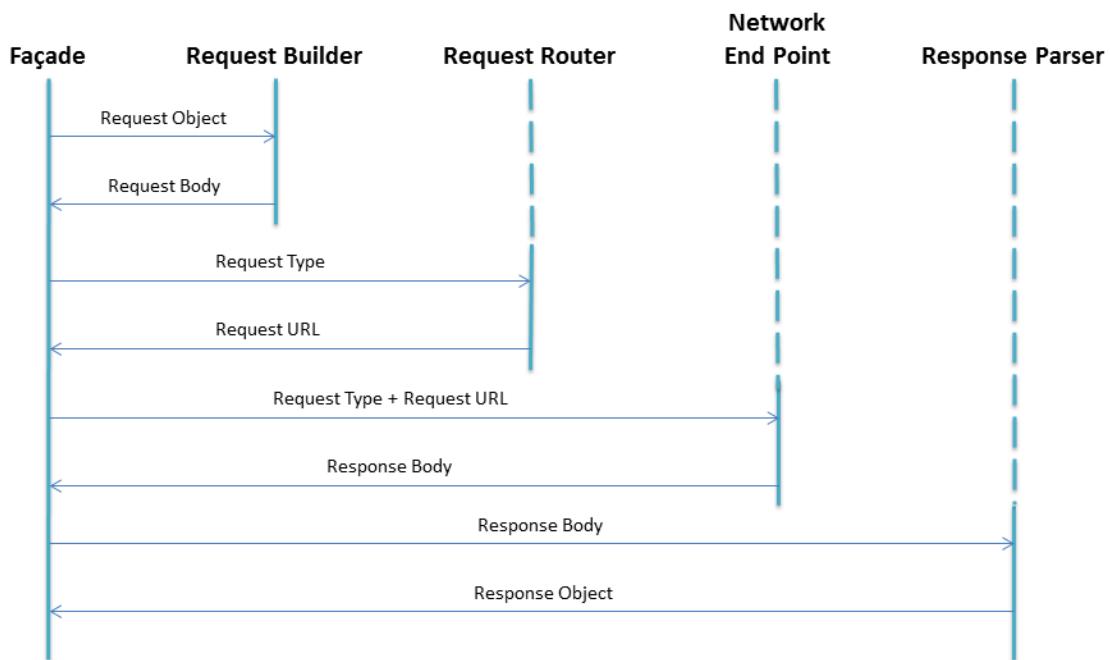
在一个基于 Spring 的应用中，与外部服务的集成通常会被封装为一个 Java 接口以及其中的若干方法。例如“创建某品牌的用户”的服务很可能如下呈现：

```
public interface IdentityService {  
    Customer create(Brand brand, Customer customer);
```

一个实现了 IdentityService 接口的对象会被 Spring 实例化并放入应用上下文，需要使用该服务的客户代码可以通过依赖注入获得该对象的引用，从而调用它的 create 方法。在测试这些客户代码时，始终可以 mock 一个 IdentityService 对象，将其注入被测对象，从而解耦对外部服务的依赖。这是使用依赖注入带来的收益。

因此，我们的问题主要聚焦于集成点本身的测试。

用面向对象的语言来集成一个基于 HTTP 的服务，集成点的设计经常会出现这样一个模式，其中涉及五个主要的组成部分：门面（ Façade ）；请求构造器（ Request Builder ）；请求路由器（ Request Router ）；网络端点（ Network End Point ）；应答解析器（ Response Parser ）。它们之间的交互关系如下图：



显而易见，在这个模式中，真正需要发出网络请求的只有网络端点这个组件。该组件的作用即是“按照预先规定好的通信方式，向给定的网络地址发出给定的请求，返回应答内容”。对于基于 HTTP 的服务集成而言，网络端点的接口大致如下呈现：

```

public interface EndPoint {
    Response get(String url);
    Response post(String url, String requestBody);
    Response put(String url, String requestBody);
}
    
```

其中 Response 类包含两项主要信息：HTTP 返回码，以及应答正文。

```
public class Response {  
    private final int statusCode;  
    private final String responseBody;
```

不难注意到，EndPoint 类所关心的是把正确的请求发送到正确的地址、取回正确的应答。它并不关心这个地址究竟是什么（这是请求路由器组件的责任），也不关心请求与应答包含什么信息（这是请求构造器和应答解析器的责任）。这一特点使得EndPoint 类的测试完全不需要依赖真实服务的存在。

网络端点的测试

如前所述，EndPoint 类并不关心发送请求的地址，也不关心请求与应答的内容，只关心以正确的方式来发送请求并拿回应答——“正确的方式”可能包括身份认证与授权、必要的 HTTP 头信息等。为了测试这样一个类，我们不需要朝真正的网络服务地址发送请求，也不需要遵循真实的请求/应答协议，完全可以自己创造一个 HTTP 服务，用最简单的请求/应答文本来进行测试。

Moco[2]就是专门用于这种场合的测试工具。按照作者的介绍，Moco 是“一个非常容易设置的 stub 框架，主要用于测试与集成”。在 JUnit 测试中，只需要两行代码就可以声明一个 HTTP 服务器，该服务器监听 12306 端口，对一切请求都会以字符串“foo”作为应答：

```
MocoHttpServer server = httpserver(12306);  
server.reponse("foo");
```

接下来就可以像访问正常的服务器一样，用 Apache Commons HTTP Client 来访问这个服务器。唯一需要注意的是，访问服务器的代码需要放在 running 块中，以确保服务器能被正常关闭：

```
running(server, new Runnable() {
```

```

@Override

public void run() throws IOException {

    Content content =
Request.Get("http://localhost:12306").execute().returnContent();

    assertThat(content.asString(), is("foo"));

}
    
```

当然，作为一个测试辅助工具，Moco 支持很多灵活的配置，感兴趣的读者可以自行查阅文档。接下来我们就来看如何用 Moco 来测试我们系统中的网络端点组件。作为例子，我们这里需要集成的是用于管理用户身份信息的 OpenPTK[3]。OpenPTK 使用自定义的 XML 通信协议，并且每次请求之前要求客户端程序先向/openptk-server/login 地址发送应用名称和密码以确认应用程序的合法身份。为此，我们先准备一个 Moco server 供测试之用：

```

server = httpserver(12306);
server.post(and(
    by(uri("/openptk-server/login")),
    by("clientid=test_app&clientcred=fake_password"))).response(status(200));
    
```

接下来我们告诉要测试的网络端点，应该访问位于 localhost:12306 的服务器，并提供用户名和密码：

```

configuration = new IdentityServiceConfiguration();
configuration.setHost("http://localhost:12306");

configuration.setClientId("test_app");
    
```

```
configuration.setClientCredential("fake_password");
xmlEndPoint = new XmlEndPoint(configuration);
```

然后就可以正式开始测试了。首先我们测试 XmlEndPoint 可以用 GET 方法访问一个指定的 URL，取回应答正文：

```
@Test
public void shouldBeAbleToCarryGetRequest() throws Exception {
    final String expectedResponse = "<message>SUCCESS</message>";
    server.get(by(uri("/get_path"))).response(expectedResponse);
    running(server, new Runnable() {
        @Override
        public void run() {
            XmlEndPointResponse response =
                xmlEndPoint.get("http://localhost:12306/get_path");
            assertThat(response.getStatusCode(),
            equalTo(STATUS_SUCCESS));
            assertThat(response.getResponseBody(),
            equalTo(expectedResponse));
        }
    });
}
```

实现了这个测试以后，我们再添加一个测试，描述“应用程序登录失败”的场景，这样我们就得到了对 XmlEndPoint 类的 get 方法的完全测试覆盖：

```
@Test(expected = IdentityServiceSystemException.class)
public void shouldRaiseExceptionIfLoginFails() throws Exception {
    configuration.setClientCredential("wrong_password");
    running(server, new Runnable() {
```

```

@Override
public void run() {
    xmlEndPoint.get("http://localhost:12306/get_pa
th");
}
}
    
```

以此类推，也很容易给 post 和 put 方法添加测试。于是，在 Moco 的帮助下，我们就完成了对网络端点的测试。虽然这部分测试真的发起了 HTTP 请求，但只是针对位于 localhost 的 Moco 服务器，并且测试的内容也只是最基本的 GET/POST/PUT 请求，因此测试仍然快且稳定。

Moco 的前世今生

在 ThoughtWorks 成都分公司，我们为一家保险企业开发在线应用。由于该企业的数据与核心保险业务逻辑存在于 COBOL 开发的后端系统中，我们所开发的在线应用都有大量集成工作。不止一个项目组发出这样的抱怨：因为依赖了被集成的远程服务，我们的测试变得缓慢而不稳定。于是，我们的一位同事郑晔[4]开发了 Moco 框架，用它来简化集成点的测试。

除了我们已经看到的 API 模式（在测试用例中使用 Moco 提供的 API）以外，Moco 还支持 standalone 模式，用于快速创建一个测试用的服务器。例如下列配置（位于名为“foo.json”的文件中）就描述了一个最基本的 HTTP 服务器：

```

[
{
    "response" : {
        "text" : "Hello, Moco"
    }
}
    
```

```
}
```

```
]
```

把这个服务器运行起来：

```
java -jar moco-runner-<version>-standalone.jar -p 12306 foo.json
```

再访问“<http://localhost:12306>”下面的任意 URL，都会看到“Hello, Moco”的字样。结合各种灵活的配置，我们就可以很快地模拟出需要被集成的远程服务，用于本地的开发与功能测试。

感谢开源社区的力量，来自澳大利亚的 Garrett Heel 给 Moco 开发了一个 Maven 插件[5]，让我们可以在构建过程中适时地打开和关闭 Moco 服务器（例如在运行 Cucumber[6]功能测试之前启动 Moco 服务器，运行完功能测试之后关闭），从而更好地把 Moco 结合到构建过程中。

目前 Moco 已经被 ThoughtWorks 成都分公司的几个项目使用，并且根据这些项目提出的需求继续演进。如果你有兴趣参与这个开源项目，不论是使用它并给它提出改进建议，还是为它贡献代码，郑晔都会非常开心。

其它组件的测试

有了针对网络端点的测试之后，其他几个组件的测试已经可以不必发起网络请求。理论上来说，每个组件都应该独自隔离进行单元测试；但个人而言，对于没有外部依赖的对象，笔者并不特别强求分别独立测试。只要有效地覆盖所有逻辑，将几个对象联合在一起测试也并无不可。

出于这样的考虑，我们可以针对整个集成点的 façade（即 IdentityService）进行测试。在实例化 IdentityService 对象时，需要 mock[7]其中使用的 XmlEndPoint 对象，以隔离“发起网络请求”的逻辑：

```
xmlEndPoint = mock(XmlEndPoint.class);
identityService = new IdentityServiceImpl(xmlEndPoint);
```

然后我们就需要 mock 的 XmlEndPoint 对象表现出几种不同的行为，以便测试 IdentityService(及其内部使用的其他对象) 在这些情况下都做出了正确的行为。以“查找用户”为例，XmlEndPoint 的两种行为都是 OpenPTK 的文档里所描述的：

1. 找到用户：HTTP 状态码为“200 FOUND”，应答正文为包含用户信息的 XML；
2. 找不到用户：HTTP 状态码为“204 NO CONTENT”，应答正文为空。

针对第一种（“找到用户”）情况，我们对 mock 的 XmlEndPoint 对象提出期望，要求它在 get 方法被调用时返回一个代表 HTTP 应答的对象，其中返回码为 200、正文为包含用户信息的 XML：

```
when(xmlEndPoint.get(anyString())).thenReturn(
    new XmlEndPointResponse(STATUS_SUCCESS,
    userFoundResponse));
```

当 mock 的 XmlEndPoint 对象被设置为这样的行为，“查找用户”操作就应该能找到用户，并组装出合法的结果对象：

```
Customer customer =
identityService.findByEmail("gigix1980@gmail.com");
assertThat(customer.getFirstName(), equalTo("Jeff"));
assertThat(customer.getLastName(), equalTo("Xiong"));
```

userFoundResponse 所引用的 XML 字符串中包含了用户信息，当 XmlEndPoint 返回这样一个字符串时，IdentityService 就能把它转换成一个 Customer 对象。这样我们就验证了 IdentityService (以及它内部所使用的其他对象) 的功能。

第二种场景（“找不到用户”）的测试也与此相似：

```
@Test
public void shouldReturnNullWhenUserDoesNotExist() throws Exception {
    when(xmlEndPoint.get(anyString())).thenReturn(
        new XmlEndPointResponse(STATUS_NO_CONTENT, null));
    Customer nonExistCustomer =
```

```

        identityService.findByEmail("not.exist@gmail.com");
        assertThat(nonExistCustomer, nullValue());
    }
}
    
```

其他操作的测试也与此相似。

集成测试

有了上述两个层面的测试，我们已经能够对集成点的五个组件完全覆盖。但是请勿掉以轻心：100% 测试覆盖率并不等于所有可能出错的地方都被覆盖。例如我们前述的两组测试就留下了两个重要的点没有得到验证：

1. 真实的服务所在的 URL；
2. 真实的服务其行为是否与文档描述一致。

这两个点都是与真实服务直接相关的，必须结合真实服务来测试。另一方面，对这两个点的测试实际上描述功能重于验证功能：第一，外部服务很少变化，只要找到了正确的用法，在相当长的时间内不会改变；第二，外部服务如果出错（例如服务器宕机），从项目本身而言并没有修复的办法。所以真正触碰到被集成的外部服务的集成测试，其主要价值是准确描述外部服务的行为，提供一个可执行的、精确的文档。

为了提供这样一份文档，我们在集成测试中应该尽量避免使用应用程序内实现的集成点（例如前面出现过的 IdentityService），因为如果程序出错，我们希望自动化测试能告诉我们：出错的究竟是被集成的外部服务，还是我们自己编写的程序。我更倾向于使用标准的、接近底层的库来直接访问外部服务：

```

System.out.println("== 2. Find that user out ==");
GetMethod getToSearchUser = new GetMethod(
    configuration.getUrlForSearchUser("gigix1980@gmail.com"));
getToSearchUser.setRequestHeader("Accept", "application/xml");
httpClient.executeMethod(getToSearchUser);
assertThat(getToSearchUser.getStatusCode(), equalTo(200));
System.out.println(getResponseBody(getToSearchUser));
    
```

可以看到，在这段测试中，我们直接使用 Apache Commons HTTP Client 来发起网络请求。对于应答结果我们也并不验证，只是确认服务仍然可用、并把应答正文（ XML 格式）直接打印出来以供参考。如前所述，集成测试主要是在描述外部服务的行为，而非验证外部服务的正确性。这种粒度的测试已经足够起到“可执行文档”的作用了。

持续集成

在上面介绍的几类测试中，只有集成测试会真正访问被集成的外部服务，因此集成测试也是耗时最长的。幸运的是，如前所述，集成测试只是用于描述外部服务，所有的功能验证都在网络端点测试（使用 Moco）及其他组件的单元测试中覆盖，因此集成测试并不需要像其他测试那样频繁运行。

Maven 已经对这种情形提供了支持。在 Maven 定义的构建生命周期[8]中，我们可以看到有“test”和“integration-test”两个阶段（ phase ）。而且在 Maven 项目网站上我们还可以看到一个叫“Failsafe”的插件[9]，其中的介绍这样说道：

The Failsafe Plugin is designed to run integration tests while the Surefire Plugins is designed to run unit tests. The name (failsafe) was chosen both because it is a synonym of surefire and because it implies that when it fails, it does so in a safe way.

按照 Maven 的推荐，我们应该用 Surefire 插件来运行单元测试，用 Failsafe 插件来运行集成测试。为此，我们首先把所有集成测试放在“integration”包里，然后在 pom.xml 中配置 Surefire 插件不要执行这个包里的测试：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${maven-surefire-plugin.version}</version>
  <executions>
    <execution>
      <id>default-test</id>
```

```
<phase>test</phase>

<goals>

    <goal>test</goal>

</goals>

<configuration>

    <excludes>

        <exclude>**/integration/**/*Test.java</exclude>

    </excludes>

</configuration>

</execution>

</executions>

</plugin>
```

再指定用 Failsafe 插件执行所有集成测试：

```
<plugin>

    <artifactId>maven-failsafe-plugin</artifactId>

    <version>2.12</version>

    <configuration>

        <includes>

            <include>**/integration/**/*Test.java</include>

        </includes>

    </configuration>
```

```
<executions>

    <execution>

        <id>failsafe-integration-tests</id>

        <phase>integration-test</phase>

        <goals>

            <goal>integration-test</goal>

        </goals>

    </execution>

    <execution>

        <id>failsafe-verify</id>

        <phase>verify</phase>

        <goals>

            <goal>verify</goal>

        </goals>

    </execution>

</executions>

</plugin>
```

这时如果执行“mvn test”，集成测试已经不会运行；如果执行“mvn integration-test”，由于“integration-test”是在“test”之后的一个阶段，因此两组测试都会运行。这样我们就可以在持续集成服务器（例如 Jenkins）上创建两个不同的构建任务：一个是提交构建，每次有代码修改时执行，其中不运行集成测试；

另一个是完整构建，每天定时执行一次，其中运行集成测试。如此，我们便做到了速度与质量兼顾：平时提交时执行的构建足以覆盖我们开发的功能，执行速度飞快，而且不会因为外部服务宕机而失败；每日一次的完整构建覆盖了被集成的外部服务，确保我们足够及时地知晓外部服务是否仍然如我们期望地正常运行。

对已有系统的重构

如果一开始就按照前文所述的模式来设计集成点，自然很容易保障系统的可测试性；但如果一开始没有做好设计，没有抽象出“网络端点”的概念，而是把网络访问的逻辑与其他逻辑耦合在一起，自然也就难以写出专门针对网络访问的测试，从而使得大量测试会发起真实的网络访问，使构建变得缓慢而不可靠。

下面就是一段典型的代码结构，其中杂糅了几种不同的职责：准备请求正文；发起网络请求；处理应答内容。

```
PostMethod postMethod = getPostMethod(  
  
    velocityContext, templateName, soapAction);  
  
new HttpClient().executeMethod(postMethod);  
  
String responseBodyAsString = postMethod.getResponseBodyAsString();  
  
  
  
if (responseBodyAsString.contains("faultstring")) {  
  
    throw new WmbException();  
  
}  
  
  
  
Document document;  
  
try {  
  
    LOGGER.info("request:\n" + responseBodyAsString);  
  
    document = DocumentHelper.parseText(responseBodyAsString);  
}
```

```
    } catch (Exception e) {  
  
        throw new WmbParseException(  
  
            e.getMessage() + "\nresponse:\n" + responseBodyAsString);  
  
    }  
  
    return document;  
}
```

针对每个要集成的服务方法，类似的代码结构都会出现，从而出现了“重复代码”的坏味道。由于准备请求正文、处理应答内容等逻辑各处不同（例如上面的代码使用 Velocity[10] 来生成请求正文、使用 JDOM[11] 来解析应答），这里的重复并不那么直观，自动化的代码检视工具（例如 Sonar）通常也不能发现。因此第一步的重构是让重复的结构浮现出来。

使用抽取函数（Extract Method）、添加参数（Add Parameter）、删除参数（Remove Parameter）等重构手法，我们可以把上述代码整理成如下形状：

```
// 1. prepare request body  
  
    String requestBody = renderTemplate(velocityContext,  
templateName);  
  
// 2. execute a post method and get back response body  
PostMethod postMethod = getPostMethod(soapAction, requestBody);  
new HttpClient().executeMethod(postMethod);  
String responseBody = postMethod.getResponseBodyAsString();  
if (responseBodyAsString.contains("faultstring")) {  
    throw new WmbException();  
}
```

```
// 3. deal with response body

Document document = parseResponse(responseBody);

return document;
```

这时，第 2 段代码（使用预先准备好的请求正文执行一个 POST 请求，并拿回应答正文）的重复就变得明显了。《重构》对这种情况做了介绍[12]：

```
public class SOAPEndPoint {
    public String post(String soapAction, String requestBody) {
        PostMethod postMethod = getPostMethod(soapAction, requestBody);
        new HttpClient().executeMethod(postMethod);
        String responseBody = postMethod.getResponseBodyAsString();
        if (responseBodyAsString.contains("faultstring")) {
            throw new WmbException();
        }
        return responseBody;
    }
}
```

原来的代码如果两个毫不相关的类出现 Duplicated Code，你应该考虑对其中一个使用 Extract Class，将重复代码提炼到一个独立类中，然后在另一个类内使用这个新类。但是，重复代码所在的函数也可能的确只应该属于某个类，另一个类只能调用它，抑或这个函数可能属于第三个类，而另两个类应该引用这第三个类。你必须决定这个函数放在哪儿最合适，并确保它被安置后就不会再在其他任何地方出现。

这正是我们面对的情况，也正是“网络端点”这个概念应该出现的时候。使用抽取函数和抽取类（Extract Class）的重构手法，我们就能得到名为 SOAPEndPoint 的类：

再按照前文所述的测试策略，使用 Moco 给 SOAPEndPoint 类添加测试。可以看到，SOAPEndPoint 的逻辑相当简单：把指定的请求文本 POST 到指定的 URL；如果应答文本包含“faultstring”字符串，则抛出异常；否则直接返回应答文本。尽管名为“SOAPEndPoint”，post 这个方法其实根本不关心请求与应答是否符合 SOAP 协议，因此在测试这个方法时我们也不需要让 Moco 返回符合 SOAP 协议的应答文本，只要覆盖应答中是否包含“faultstring”字符串的两种情况即可。

读者或许会问：既然 post 方法并不介意请求与应答正文是否符合 SOAP 协议，为什么这个类叫 SOAPEndPoint？答案是：在本文没有给出实现代码的 getPostMethod 方法中，我们需要填入一些 HTTP 头信息，这些信息是与提供 Web Services 的被集成服务相关的。这些 HTTP 头信息（例如应用程序的身份认证、Content-Type 等）适用于所有服务方法，因此可以抽取到通用的 getPostMethod 方法中。

随后，我们可以编写一些描述性的集成测试，并用 mock 的方式使所有“使用 SOAPEndPoint 的类”的测试不再发起网络请求。至此，我们就完成了对已有的集成点的重构，并得到了一组符合前文所述的测试策略的测试用例。当然读者可以继续重构，将请求构造器与应答解析器也分离出来，在此不再赘述。

小结

在开发一个“重集成”的 JavaEE Web 应用的过程中，自动化测试中对被集成服务的依赖使得构建过程变得缓慢而脆弱。通过对集成点实现的考察，我们识别出一个典型的集成点设计模式。基于此模式以及与之对应的测试策略，借助 Moco 这个测试工具，我们能够很好地隔离对被集成服务的依赖，使构建过程快速而可靠。

随后我们还考察了已有的集成点实现，并将其重构成为前文所述的结构，从而将同样的测试策略应用于其上。通过这个过程，我们验证了：本文所述的测试策略是普遍适用的，遗留系统同样可以通过文中的重构过程达到解耦实现、从而分层测试的目标。

[1] “构建”一词在本文中是指使用自动化的构建工具（例如 Maven）将源代码变为可交付的软件的过程。一般而言，JavaEE 系统的构建过程通常包括编译、代码检查、单元测试、集成测试、打包、功能测试等环节。

[2] <https://github.com/dreamhead/moco>

[3] <http://www.openptk.org/>

-
- [4] <http://dreamhead.blogbus.com/>
 - [5] <https://github.com/GarrettHeel/moco-maven-plugin>
 - [6] <http://cukes.info/>
 - [7] 笔者使用的 mock 框架是 Mockito : <https://code.google.com/p/mockito/>
 - [8] <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>
 - [9] <http://maven.apache.org/plugins-archives/maven-failsafe-plugin-2.12.4/>
 - [10] <http://velocity.apache.org/>
 - [11] <http://jdom.org/>
 - [12] 《重构》 , 3.1 小节。
-

原文链接 :

<http://www.infoq.com/cn/articles/enterprise-systems-integration-points>

相关内容 :

- [2011 企业信息架构设计：换位思考的一年](#)
- [Dan Allen 谈 Arquillian 测试框架](#)
- [CloudBees 发布 Jenkins 企业版](#)
- [组合测试法中的全对偶测试法](#)
- [Gorilla Logic 发布 FlexMonkey 5——一款面向 Adobe Flex 和 AIR 的开源测试工具](#)
- [使用 Test 262 测试浏览器的 JavaScript 兼容性](#)

本期专栏 | Column

关于 Java 性能的 9 个谬论

作者 [Ben Evans](#) , 译者 [臧秀涛](#)

Java 的性能有某种黑魔法之称。部分原因在于 Java 平台非常复杂，很多情况下问题难以定位。然而在历史上还有一种趋势，人们靠智慧和经验来研究 Java 性能，而不是靠应用统计和实证推理。在这篇文章中，我希望拆穿一些最荒谬的技术神话。

1. Java 很慢

关于 Java 的性能有很多谬论，这一条是最过时的，可能也是最为明显的。

确实，在上世纪 90 年代和本世纪初处，Java 有时是很慢。

然而从那以后，虚拟机和 JIT 技术已经有了十多年的改进，Java 的整体性能现在已经非常好了。

在 6 个独立的 [Web 性能基准测试](#) 中，Java 框架在 24 项测试中有 22 项位列前四。

尽管 JVM 利用性能剖析仅优化常用的代码路径，但这种优化效果很明显。很多情况下，JIT 编译的 Java 代码和 C++一样快，而且这样的情况越来越多了。

尽管如此，依然有人认为 Java 平台很慢，这或许源自体验过 Java 平台早期版本的人的历史偏见。

在下结论之前，我们建议保持客观的态度，并且评估一下最新的性能结果。

2. 可以孤立地看待单行 Java 代码

考虑下面这行短小的代码：

对 Java 开发者而言，看似很明显，这行代码一定会分配一个对象并调用适当的构造器。

我们也许可以据此推出性能边界了。我们认为这行代码一定会导致执行一定量的工作，基于这种推定，就可以尝试计算其性能影响了。

其实这种认识是错误的，它让我们先入为主地认为，不管什么工作，在任何情况下都会进行。

事实上，javac 和 JIT 编译器都能够将死代码优化掉。就 JIT 编译器而言，基于性能剖析数据，甚至可以通过预测将代码优化掉。在这样的情况下，这行代码根本不会运行，所以不会影响性能。

此外，在某些 JVM 中——比如 JRockit——JIT 编译器甚至可以将对象上的操作分解，这样即便代码路径还有效，分配操作也可以避免。

这里的寓意是，在处理 Java 性能问题时，上下文非常重要，过早的优化有可能产生违反直觉的结果。所以最好不好过早优化。相反，应该总是构建代码，并且使用性能调校技术来定位性能热点，然后加以改进。

3.微基准测试和你想象的一样

正如我们上面看到的那样，检查一小段代码不如分析应用的整体性能来的准确。

尽管如此，开发者还是喜欢编写微基准测试。似乎对平台底层的某些方面进行修修补补会带来无穷的乐趣。

理查德·费曼曾经说过：“不要欺骗自己，你自己正是最容易被欺骗的人。”这句话用来说说明编写 Java 微基准测试这件事是再合适不过了。

编写良好的微基准测试极其困难。Java 平台非常复杂，而且很多微基准测试只能用于测量瞬时效应，或是 Java 平台的其他意想不到的方面。

例如，如果没有经验，编写的微基准测试往往就是测一下时间或垃圾收集，却没有抓住真正的影响因素。

只有那些有实际需求的开发者和开发团队才应该编写微基准测试。这些基准测试应该完全公开（包括源代码），而且是可以复现的，还应接受同行评审及进一步的审查。

Java 平台的很多优化表明统计运行和单次运行对结果影响很大。要得到真实可靠的答案，应该将一个单独的基准测试运行多次，然后把结果汇总到一起。

如果读者感觉有必要编写微基准测试，Georges、Buytaert 和 Eeckhout 等人的论文《利用严格的统计方法评测 Java 性能（Statistically Rigorous Java Performance Evaluation）》是个不错的开始。缺乏适当的统计分析，我们很容易被误导。

有很多开发好的工具以及围绕这些工具的社区（比如 Google 的 Caliper）。如果确实有必要编写微基准测试，那也不要自己编写，这时需要的是同行的意见和经验。

4. 算法慢是性能问题的最常见原因

在开发者之间有一个很常见的认知错误（普通大众也是如此），即认为系统中他们控制的那部分很重要。

在探讨 Java 性能时，这种认知错误也有所体现：Java 开发者认为算法的质量是性能问题的主要原因。开发者考虑的是代码，因此他们会偏向于考虑自己的算法。

实际上在处理一系列现实中的性能问题时，人们发现算法设计是根本问题的几率不足 10%。

相反，与算法相比，垃圾收集、数据库访问和配置错误导致应用程序缓慢的可能性更大。

大部分应用处理的数据量相对较小，因此，即使主要算法效率不高，通常也不会导致严重的性能问题。可以肯定，我们的算法不是最优的；尽管如此，算法带来的性能问题还是算小的，更多性能问题是应用栈的其他部分导致的。

因此我们的最佳建议是，使用实际生产数据来揭开性能问题的真正原因。要测量性能数据，而不是凭空猜测！

5. 缓存可以解决所有问题

“计算机科学中的所有问题都可以通过引入一个中间层来解决。”

David Wheeler 的这句程序员格言（在互联网上，这句话至少还被认为是其他两位计算机科学家说的）非常常见，尤其是在 Web 开发者之中很流行。

如果未能透彻理解现有的架构，而且分析也已停顿，往往就是“缓存可以解决所有问题”这种谬论抬头的时候了。

在开发者看来，与其处理吓人的现有系统，还不如在前面加一层缓存，将现有系统隐藏起来，以此期待最好的情况。无疑，这种方式只是让整体架构更复杂了，当下一个接手的开发者打算了解系统现状时，情况会更糟糕。

规模庞大、设计拙劣的系统往往缺乏整体的设计，是一次一行代码、一个子系统这样写出来的。然而很多情况下，简化并重构架构会带来更好的性能，而且几乎总是更容易让人理解。

所以当评估是否真的有必要加入缓存时，应该先计划收集一些基本的使用统计信息（比如命中率和未命中率等），以此证明缓存层带来的真正价值。

6.所有应用都需要关注 Stop-The-World 问题

Java 平台存在一个无法改变的事实：为运行垃圾收集，所有应用线程必须周期性停顿。有时这被当作 Java 的一个严重缺点，即使没有任何真凭实据。

实证研究表明，如果数字数据（如价格波动）变化的频率超过 200 毫秒一次，人就无法正常感知了。

应用主要是给人用的，因此我们有一个有用的经验法则，200 毫秒或低于 200 毫秒的 Stop-The-World (STW) 通常是没有影响的。有些应用可能有更高的要求（如流媒体），但很多 GUI 应用是不需要的。

少数应用（比如低延迟交易或机械控制系统）无法接受 200 毫秒的停顿。除非编写的就是这类应用，否则用户基本感觉不到垃圾收集器的影响。

值得一提的是，在应用线程数量超过物理核数的任何系统中，操作系统必须控制对 CPU 的分时访问。Stop-The-World 听着可怕，但实际上任何应用（不管是 JVM 还是其他应用）都要面对稀缺计算资源的争用问题。

如果不去测量，JVM 对应用性能有何附加影响是不清楚的。

总之，请打开 GC 日志，以此来确定停顿时间是否真的影响了应用。通过分析日志来确定停顿时间，这里既可以手工分析，也可以利用脚本或工具分析。然后再判定它们是否真的给应用带来了问题。最重要的是，问自己一个关键的问题：确实有用户抱怨吗？

7. 手写对象池适合一大类应用

认为 Stop-The-World 停顿在某种程度上是不好的，应用开发团队的一个常见反应就是在 Java 堆内实现自己的内存管理技术。这往往归结为实现一个对象池（甚至是全面的引用计数），而且需要使用了领域对象的任何代码都参与进来。

这种技术几乎总是具有误导性的。它基于过去认知，那时对象分配非常昂贵，而修改对象则廉价的多。现在的情况已经完全不同了。

现在的硬件在分配时非常高效；最新的桌面或服务器硬件，内存带宽至少是 2 到 3GB。这是一个很大的数字，除非专门编写的应用，否则要充分利用这么大的带宽还真不容易。

一般来说，正确实现对象池非常困难（尤其是有多个线程工作时），而且对象池还带来了一些负面的要求，使这种技术不是一个通用的良好选择：

所有接触到对象池代码的开发者必须了解对象池，而且能正确处理

哪些代码知道对象池，哪些代码不知道对象池，其界限必须让大家知道，并且写在文档中

这些额外的复杂性要保持更新，而且定期复审

如果有一条不满足，悄然出现问题（类似于 C 中的指针复用）的风险就又回来了

总之，只有 GC 停顿不能接受，而且调校和重构也未能将停顿减小到可以接受的水平时，才能使用对象池。

8. 在垃圾收集中，相对于 Parallel Old，CMS 总是更好的选择

Oracle JDK 默认使用一个并行的 Stop-The-World 收集器来收集老年代，即 Parallel Old 收集器。

Concurrent-Mark-Sweep (CMS) 是一个备选方案，在大部分垃圾收集周期，它允许应用线程继续运行，但这是有代价的，而且有一些注意事项。

允许应用线程与垃圾收集线程一起运行，不可避免地带来一个问题：应用线程修改了对象图，可能会影响对象的存活性。这种情况必须在事后加以清理，因此 CMS 实际上有两个 STW 阶段（通常非常短）。

这会带来一些后果：

1. 必须将所有应用线程带到安全点，每次 Full GC 期间会停顿两次；
2. 尽管垃圾收集与应用同时执行，但应用的吞吐量会降低（通常是 50%）；
3. 在使用 CMS 进行垃圾收集时，JVM 所用的簿记信息（和 CPU 周期）远高于其他的并行收集器。

这些代价是不是物有所值，取决于应用的情况。但是天下没有免费的午餐。CMS 收集器在设计上值得称道，但它不是万能的。

所以在确定 CMS 是正确的垃圾收集策略之前，首先应该确认 Parallel Old 的 STW 停顿确实不能接受，而且已经无法调校。最后，我重点强调一下，所有指标必须从与生产系统等价的系统中获得。

9.增加堆的大小可以解决内存问题

当应用陷入困境，并且怀疑是 GC 的问题时，很多应用团队的反应就是增加堆的大小。在某些情况下，这样做可以快速见效，而且为我们留出了时间来考虑更周详的解决方案。然而，如果没有充分理解性能问题的原因，这种策略反而会让事情变得更糟糕。

考虑一个编码非常糟糕的应用程序，它正在产生很多领域对象（它们的生存时间很有代表性，比如说是 2-3 秒）。如果分配率高到一定程度，垃圾收集会频繁进行，这样领域对象会被提升到老年带。领域对象几乎是一进入老年带，生存时间就结束了，从而直接死亡，但它们直到下一次 Full GC 时才会被回收。

如果增加了应用的堆大小，我们所做的不过是增加了相对短命的对象进入和死亡所用的空间。这会导致 Stop-The-World 停顿时间更长，对应用并无益处。

在修改堆大小或者调校其他参数之前，理解对象的分配和生存时间的动态是很必要的。没有测量性能数据就盲目行动，只会使情况更糟糕。在这里，垃圾收集器的老年带分布情况特别重要。

结论

当谈到 Java 的性能调校时，直觉常常起误导作用。我们需要实验数据和工具来帮助我们将平台的行为可视化并加强理解。

垃圾收集就是最好的例子。对于调校或者生成指导调校的数据而言，GC 子系统拥有无限的潜力；但是对于产品应用而言，不使用工具很难理解所产生数据的意义。

默认情况下，运行任意 Java 进程（包括开发环境和产品环境），应该至少总是使用如下参数：

- verbose:gc (打印 GC 日志)
- Xloggc: (更全面的 GC 日志)
- XX:+PrintGCDetails (更详细的输出)
- XX:+PrintTenuringDistribution(显示 JVM 所使用的将对象提升进入老年代的年龄阈值)

然后使用工具来分析日志，这里可以利用手写的脚本，可以用图生成，还可以使用 GCViewer (开源的) 或 jClarity Censum 这样的可视化工具。

『号外』：JavaOne 2013 大会将于 7 月 22–25 日在上海世博中心举行，内容涵盖使用 Java SE 构建现代应用程序、打造针对下一代智能设备的移动和嵌入式 Java 应用程序、编制基于 Java EE 的复杂企业解决方案以及在云环境中安全、无缝地构建和部署业务应用程序等，[报名或查看详情请点击](#)。

关于作者



Ben Evans 是 jClarity (这是一家创业公司，主要设计辅助开发和运维团队的性能工具) 的 CEO。他是 LJC (伦敦 Java 用户组) 的组织者之一，也是 JCP 执行委员会的成员之一，JCP 执行委员会负责帮助定义 Java 生态系统中的相关标准。他还是 Java Champion 和 JavaOne Rockstar。他与人合著了《[The Well-Grounded Java Developer](#)》一书。此外，他还经常进行公开演讲，探讨 Java 平台、性能、并发及相关话题。

原文链接：http://www.infoq.com/cn/articles/9_Fallacies_Java_Performance

相关内容：

- [关于 Java 性能的 9 个谬论](#)
- [Tabris 1.0：使用 JAVA 进行跨平台移动开发](#)

- [处理 JSON 的 Java API 标准发布](#)
- [受困于连续出现的安全问题，Java 8 发布时间推迟到 2014 年](#)
- [IntelliJ IDEA 12.1 发布 支持 JavaFX 2.0](#)
- [书评：Java 应用架构](#)

本期专栏 | Column

受困于连续出现的安全问题，Java 8 发布时间推迟到 2014 年

作者 [Vikram Gupta](#)，译者 [臧秀涛](#)

Oracle 的 Java 平台组首席架构师 Mark Reinhold 在其[博客](#)上宣布，Oracle 决定将 Java 8 的发布时间推迟 4-6 个月。

Reinhold 提到，Oracle 正在全力解决近来引发公众关注的这一波安全漏洞问题，Java 8 项目组不可避免地被抽调走了很多工程师。他说，“维护 Java 平台的安全性，其优先级总是比开发新功能要高，所以这些工作不免会抽调一些正在进行 Java 8 开发的工程师。”此外他还强调，为改进代码质量并减少缺陷，Oracle“升级”了开发流程。

Reinhold 还提到，最主要的延误在 JSR 335 中，也就是负责向 Java 中加入闭包(亦称“Lambda 表达式”的 Lambda 项目。他认为，“如果放弃 Lambda，剩下的特性尽管也比较有趣，但总体上看就没那么吸引力了。假如今年发布一个没有 Lambda 的版本，得到广泛应用的可能性很小，那又何故如此呢？”

当面临推迟发布或减少特性的选择时，Reinhold 拒绝了第三种选择：“如果为了保证按时发布而牺牲质量，那几乎可以肯定，我们会重复过去总犯的错误，将不完整的语言变更和 API 设计构建在虚拟的基石之上，会致使无数开发者在未来数年内将围绕其缺陷工作，直到这些特性——甚至整个平台——被新事物替代为止。”

看起来 Java 8 GA (General Availability) 版本的发布时间会从原计划的 2013 年 9 月推迟到 2014 年第一季度。这已经是该版本的第二次跳票了。Java 8 最初计划于[2012年底](#)发布，后来受 Java 7 开发延期的影响，被推迟到了 2013 年 9 月，并且放弃了 Jigsaw 项目。目前这次延期也会波及到 JDK 9，我们预计 JDK 8 将于 2014 年第一季度发布，而 JDK 9 则从 2015 年推迟到 2016 年初。InfoQ 去年 7 月曾[报道](#)过，Java 8 放弃了 Jigsaw 项目，而且 Reinhold 否定了该项目再次进入 Java 8 的可能性。

看一下 Java 8 保留特性的集合，显而易见，Java 8 就是通过 Lambda 项目定义的。

Lambda 项目也需要和其他大型模块进行很多协调，尤其是并发框架和泛型框架。这也印证了 Oracle 腾出开发力量去解决安全问题的解释。

Lambda 表达式的引入是在 2004 年 Java 5 引入泛型之后最大的语法增强。

闭包是越来越流行的“[函数式编程](#)”范型的基础。（纵然术语“闭包”和“Lambda 表达式”在理论上有所差别，但在 Java 语境中它们经常交替使用。）为使闭包的价值最大化，Java 还将修改相关的库。其中包括新的[流 API](#)，它提供了一种将指令操作流水线化的机制，这对函数式编程是至关重要的。再就是[Option](#) 类，它包装了可能出现的空值，消除了某些情况下测试空引用的必要性，以便简化闭包开发。

包括 Scala 和 Clojure 在内的大多数流行的 JVM 语言，Ruby，以及像 C# 和 F#（一种基于 .Net 的函数式语言）等流行的 .Net 语言都已经支持闭包，所以很多开发者希望 Java 也加入该特性。

『号外』：JavaOne 2013 大会将于 7 月 22–25 日在上海世博中心举行，内容涵盖使用 Java SE 构建现代应用程序、打造针对下一代智能设备的移动和嵌入式 Java 应用程序、编制基于 Java EE 的复杂企业解决方案以及在云环境中安全、无缝地构建和部署业务应用程序等，[报名或查看详情请点击](#)。

原文链接：http://www.infoq.com/cn/news/2013/04/Java_8_Delayed

相关内容：

- [受困于连续出现的安全问题，Java 8 发布时间推迟到 2014 年](#)
- [深入理解 Java 内存模型（七）——总结](#)
- [深入理解 Java 内存模型（六）——final](#)
- [深入理解 Java 内存模型（五）——锁](#)
- [聊聊并发（三）——JAVA 线程池的分析和使用](#)
- [关于 Java 性能的 9 个谬论](#)

本期专栏 | Column

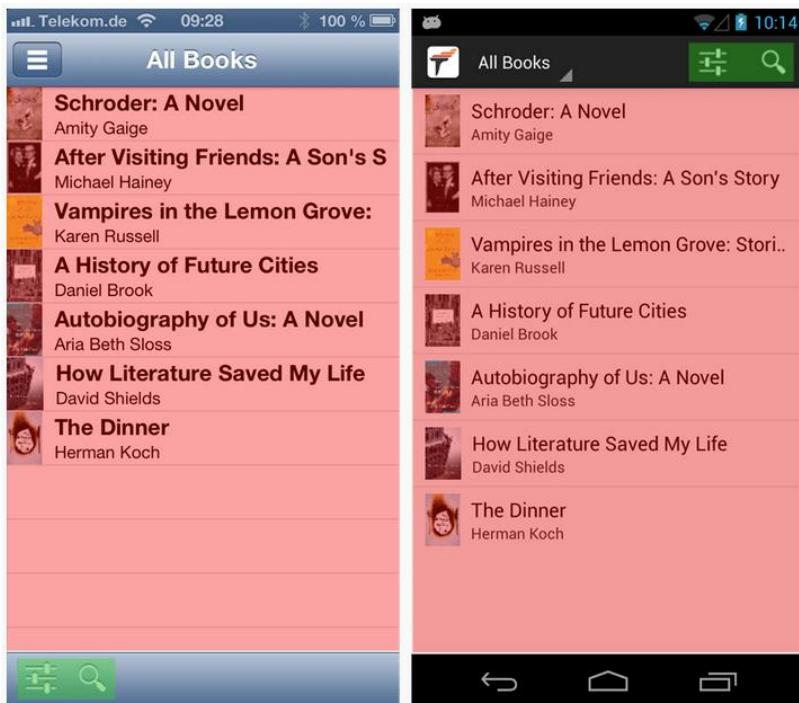
Tabris 1.0：使用 Java 进行跨平台移动开发

作者 [Abel Avram](#) , 译者 [廖煜嵘](#)

经过三年的开发，EclipseSource 终于发布了 Tabris 1.0，这是一个支持 iOS 和 Android 的跨平台 Java 移动开发框架。Tabris 定位于企业，与其他移动解决方案不同的是，它使用了不同的方法：

- 大多数编程工作都使用 JAVA 完成
- 业务逻辑和客户端 UI 的二进制表示运行在服务器端的 [Eclipse RAP](#) 上
- 一个很薄的客户端应用运行在移动设备上
- 服务器端通过 JSON 格式与客户端通信，发送数据和命令让客户端 创建可视的 UI
- 客户端使用原生组件生成界面
- iOS 的客户端使用 Object-C 编写，而 Android 的客户端则使用 Java 编写

Tabris 自带了一个构建于的 Java SWT API 之上的 [UI 工具包](#)。该工具包增加了两个主要功能部件：页面（Page）和动作（Action）。前者包含的是应用的基本内容，而后者主要用来执行用户的命令。下面是一个 Tabris 示例应用的截图，其中页面显示为红色，而动作显示为绿色：



页面可以互相链接并导航，而动作既可以是整个应用的全局动作，也可以是当前页面的局部动作。

在处理完一个组件后，应用的服务器端会将一个 JSON 片段发送到客户端，比如下面的例子是创建一个按钮：

```
{
  "head": {
    "requestCounter": 3
  },
  "operations": [
    [ "create", "w22", "rwt.widgets.Button",
      {
        "parent": "w6",
        "style": [ "CHECK" ],
        "bounds": [ 103, 231, 107, 23 ],
        "tabIndex": 10,
        "foreground": [ 0, 0, 0, 255 ],
        "text": "Sample Button",
        "alignment": "left",
        "selection": true
      }
    ]
  ]
}
```

客户端应用则会使用 iOS 或 Android 的原生组件渲染该按钮。

InfoQ 采访了 EclipseSource 的 Tabris 服务器端的团队负责人 [Holger Staudacher](#)，以了解关于该框架的更多信息。据 Staudacher 介绍，Taris 定位于企业，而且需要一直连接到服务端：

Tabris 用于现场移动类应用。这意味着它最好工作在受控的环境中。我所说的是使用固定的网络连接，诸如此类。我们的客户——如医院——所有的设备都连接到同一网络。此类应用程序通常是由企业开发的。

Tabris 不支持脱机工作。移动用户界面基本就是一个会话。因此，如果用户的设备断开了连接，则会话会变为无效。在移动客户端可以进行错误处理。我们实现了标准的错误处理，比如重新发送 http 请求等等。但应用开发人员可以使用原生扩展加以扩充。

当被问及 Tabris 是否适合开发一般应用时，Staudacher 说理论上是可以的，但那并非它的最佳使用场景：

对于一般的应用，这意味着服务器的负载会很高。我们使用的是标准的 Java EE 技术。所以，面对这样的高负载，可以使用 Java EE 集群机制。目前我们尚未测试数以百万计的用户负载。但从技术的角度看应该是可行的。

Tabris 支持[许多 SWT 组件](#)，还增加了对一些原生感应器（如摄像头和地理定位设备）的支持。

目前，Tabris 支持平板电脑，但这样的应用每次只能显示一个页面。将来会添加对多个页面的支持。此外，Tabris 可以扩展到其他移动平台，如果有动力的话，该框架或许会支持 Windows。

在不久的将来，Tabris 团队打算增加如下功能：

- 支持地址簿
- 支持设备的不同方位状态（垂直或水平）
- 支持用于处理客户端事件的客户端脚本
- 支持 [XCallbackUrl](#)

根据 [FAQ](#) 所述，使用 Tabris 创建的应用能通过 App Store 部署，“只要你遵守 App Store 的规则”。在 Google Play 上有一个针对 Android 的[示例应用](#)。

Tabris 并不开源，但企业授权用户能获得源代码。更多详情，请访问
[EclipseSource 的价格体系页面。](#)

『号外』：JavaOne 2013 大会将于 7 月 22–25 日在上海世博中心举行，内容涵盖使用 Java SE 构建现代应用程序、打造针对下一代智能设备的移动和嵌入式 Java 应用程序、编制基于 Java EE 的复杂企业解决方案以及在云环境中安全、无缝地构建和部署业务应用程序等，[报名或查看详情请点击。](#)

原文链接：<http://www.infoq.com/cn/news/2013/05/Tabris-mobile-Java>

推荐文章 | Articles

书评：Java 应用架构

作者 [Alex Blewitt](#) , 译者 [张卫滨](#)

《Java应用架构 以OSGi为例讲解模块化模式》(Java Application Architecture: Modularity Patterns with Examples using OSGi)是 Kirk Knoernschild 编写的一本优秀图书，它通过 18 个模式列表来讲解怎样在应用程序的设计中实现模块化设计。

本书的第一部分介绍了使用模块化的理由，这是通过介绍运行时和开发时的模块化支持进行的。接下来，介绍了它是如何在设计时能够有利于架构并展现了怎样与面向服务的架构进行集成。最后，会有一个参考样例来展示这些功能如何组合在一起。

本书的第二部分是模式的列表，它分成五个不同的部分，这些模式可以在[该站点的模式目录中找到](#):

- 基本模式 (Base Pattern)
 - [管理关系 \(Manage Relationships \)](#) – 设计模块关系。
 - [模块重用 \(Module Reuse \)](#) – 强调模块级别的重用。
 - [模块内聚 \(Cohesive Modules \)](#) – 模块的行为应该只服务于一个目的。
- 依赖模式 (Dependency Pattern)
 - [非循环关系 \(Acyclic Relationships \)](#) – 模块关系必须是非循环的。
 - [等级化模块 \(Levelize Modules \)](#) – 模块关系应该是等级化的。
 - [物理分层 \(Physical Layers \)](#) – 模块关系不应该违反概念上的分层。
 - [容器独立 \(Container Independence \)](#) – 模块应该独立于运行时容器。
 - [独立部署 \(Independent Deployment \)](#) – 模块应该是独立的可部署单元。
- 可用性模式
 - [发布接口 \(Published Interface \)](#) – 使模块的发布接口众所周知。
 - [外部配置 \(External Configuration \)](#) – 模块应该可以在外部进行配置。
 - [默认实现 \(Default Implementation \)](#) – 为模块提供一个默认实现。
 - [模块门面 \(Module Facade \)](#) – 创建一个门面，使其担当细粒度模块底层实现的粗粒度入口。

- 扩展性模式

- [抽象化模块 \(Abstract Modules\)](#) – 依赖于模块的抽象元素。
- [实现工厂 \(Implementation Factory\)](#) – 使用工厂来创建模块的实现类。
- [分离抽象 \(Separate Abstraction\)](#) – 将抽象与实现它们的类放在各自独立的模块之中。

- 通用模式

- [就近异常 \(Colocate Exceptions\)](#) – 异常应该接近抛出它们的类或接口。
- [等级化构建 \(Levelize Build:\)](#) – 按照模块的等级执行构建。
- [测试模块 \(Test Module\)](#) – 每个模块应该有一个对应的测试模块，用来校验模块的行为并阐述其用法。

本书的最后一部分涵盖了模式化的应用程序，它使用 OSGi 作为 Java 模块系统的样例。尽管这一部分是特定于 OSGi 的，但是本书其他部分所讲解的模式可以用于 Java 的任意模块系统，包括即将到来的 Jigsaw 项目。对于那些没有运行时模块系统但又想构建模块化软件的人们来说，这也是一个很好的指导。

InfoQ 联系到了 Kirk，并询问了他关于软件和模块化的一些想法，我们的讨论是从过去几十年来软件的复杂性是如何变化开始的：

Kirk: 毫无疑问，在过去的几十年中软件获得了持续的增长。更加强大的硬件以及更具表现力的编程语言使得创建强大的软件系统更为容易。

如今，大多数的开发人员不用再过多担心要应对严格的内存限制以及有限的处理能力。例如，一项由 Jyväskylä 进行的研究显示，在 1990 年存在 1200 亿行代码，到 2000 年这个数字已经增长了一倍多到了 2500 亿行。这个研究也声明代码行数每七年就翻一倍。

这是很令人吃惊的并且需要对此有清醒的认识，它意味着相对于某些人编写第一行代码的时候，我们在未来的七年中将会编写比以前更多的代码。在很多方面来看，这种代码上的增长并不是一件坏事。如果软件没有增长和进化，它最终就会消亡。使用软件的人们很自然地希望从它身上得到更多，所以我们要推出新的版本并进行更新来支持我们的用户。但是更多的代码会意味着更多的维护工作。更大的系统天生就会比那些较小的系统维护起来更为困难。

InfoQ：在软件的设计和架构方面，有什么影响吗？

Kirk 令人遗憾的是，目前还没有明显的影响。在很多方面，我们依然还是以相同的方式来设计软件，就像 15 或 20 年前所做的那样。但是，这不会持续太久，有些事情需要发生变化。

开发庞大的软件已经不合适了。SOA 以及服务在正确的方向上迈出了一步，但是只解决了所面临挑战的一部分。服务设计中最复杂的一个方面就是确定服务的粒度。如果太粗粒度的话，在跨各种上下文方面服务的用处就不大(例如可重用性)，而太细粒度的话，对于发挥作用来说服务所做的就有所不足 (如可用性)。

正如我在本书中所讨论的，最大化重用会使可用变得复杂。单个抽象层次并不能解决这个问题。更多的层级是必要的，而模块化就是下一个等级，模块化就是下一步，它会帮助你实现“一直向下的”架构。（译者注：作者在书中借鉴了“海龟背地球”的故事，原文是“turtles all the way down”。如想更多地了解英文中的这个俗语，读者可以参考维基百科的如下地址：

http://en.wikipedia.org/wiki/Turtles_all_the_way_down。在本书中，作者使用了类似的词汇“architecture all the way down”，指的是架构不仅要关注高层的服务，还要关注中间的模块和包甚至底层的代码，我们将这个短语译为“一直向下的架构”。）

最终形成的结果就是，你在合适等级的粒度上使用已有的模块来组装服务，模块是与服务不同等级的粒度，这有助于你实现一直向下的架构。

InfoQ：JVM 中模块化的未来如何？

Kirk : 构建于 JVM 之上的模块化方面，有很多的事情正在进行。目前，OSGi 正在应用于各个主要厂商的产品之中并且在很多的组织中都是可以使用的。

令人遗憾的是，OSGi 在企业中还没有获得驱动力。有些人说它太复杂，但他们真正想说的是设计模块化的软件是很困难的。正如你所熟悉的 Frederick Brooks 所言，这个问题是以软件开发的本质复杂性为核心的。架构和设计是十分困难的。

然而，有弹性的、高适应性的、灵活的以及可维护的软件是成功的关键，尤其是今天快节奏的企业环境中，在企业中交付需要比以往更大的软件开发团队。当然，OSGi 并不是唯一的模块系统。Oracle 也在努力将 Jigsaw 加入到 Java SE 之中，这将会在 2013 年可用（译者注：Jigsaw 已经推迟到了预

计 2015 年发布的 Java SE9 之中，本文的最初发表日期为 2012 年 7 月）。Java 平台将会支持模块化，它将会改变设计软件系统的方式。

InfoQ：是什么促使你决定编写《Java 应用架构》这本书？

Kirk：大约在 10 年以前，我认识到我所设计的软件有一个严重的问题。我们将所有的精力放在设计良好的类结构上，然后我们将所有的东西打包到一个巨大的可部署单元中，如 WAR 或 EAR 文件。这感觉起来并不好。所有的努力是用来创建灵活的软件，但是我们在重用、可维护性以及灵活性方面并没有看到明显的收益。

与此同时，我签署了本人第二本书的合约并计划编写关于软件模式的内容。我将自己在软件设计方面所面临的严重问题带到了写作之中；感觉有些东西丢失了，而这使得我苦苦追寻的收益不可能实现。我一直以来就是 Bob Martin 的 [SOLID 原则](#) 和 [GOF 模式](#)（译者注：参见机械工业出版社的《设计模式：可复用面向对象软件的基础》）的忠实粉丝。我还开始研究 Clemens Szyperski 的作品，此人是[《Component Software: Beyond Object-Oriented Programming》](#)（译者注：该书中文版由电子工业出版社引进，中文书名为《构件化软件--超越面向对象编程》）一书的作者，以及 John Lakos 的作品，此人编写了[《Large Scale C++ Software Design》](#)（译者注：该书中文版由中国电力出版社引进，中文书名为《大规模 C++ 程序设计》）。我接受了所有的这些理念，并且改变了自己设计软件系统的方式，也就是关注 JAR 文件并将其视为模块化单元。

在过去的几年中，我完善了这种方式。通过将 JAR 文件作为模块化单元，我突然能够实现很多以前苦苦追寻的收益。你在本书中看到的很多模式都是 10 年前开始的结果。同时，我曾经还有一本完成大约 70% 的书，但是它里面没有包含我在同一时期所使用的很多理念。所以我暂时停止了那本书，而是花费时间继续证明这些模式，大约在两三年前，遇到到一个很有耐心的出版人，让我相信一切已经就绪。可能这更多地描述了这本书是怎么来的，而不是我为什么要写它。

最终，我写这本书是因为这些模式异乎寻常地改善了我开发的系统，我希望将它们与全世界共享。

InfoQ：这本书是特定于 OSGi 的吗？

Kirk：绝对不是这样的。它不特定于任何的模块框架。这些模式甚至整本书在设计之时就是要借助标准 Java 所能提供的特性来进行使用的。实际上，在 2006 年之前，我并没有接触过 OSGi，这已经是我开始探索这个问题四年之后的事情了。

我发现 OSGi 后感觉茅塞顿开。我发现了一个志同道合的社区以及对众多模式提供运行时支持的框架。当我学习 OSGi 的时候，我开始思考将已有的应用迁移至 OSGi 都需要些什么。

本书提供了一些关于 OSGi 的样例，但是主要是用来讲述 OSGi 的威力以及让大家简单了解将系统从标准 Java 迁移到 OSGi 都需要做什么。

InfoQ：如果你不使用 OSGi 或 Jigsaw，能进行模块化吗？

Kirk：绝对是可行的。在第二章中，我讨论了模块化的两个方面——运行时模型（runtime model）和开发模型（development model）。开发模型又可以进一步划分为编程模型（programming model）和设计范式（design paradigm）。

模块框架会为你提供编程模型和运行时模型。但是不会有什么框架帮助你设计很好的一组模块。这个任务留给了开发人员，这也是本书关注的焦点。

我想与面向对象范式做一个类比。仅仅因为语言支持继承、多态、封装以及动态绑定并不能保证产生伟大的面向对象软件。设计模式会帮助你做到这一点。

对于模块化来说是同样的道理。即便你使用了模块化框架，并不能保证会有模块化的软件。所以本书的目的在于帮助你首先设计模块化的软件，不管你使用的是 OSGi、Jigsaw 还是标准的 Java。按照这种方式，你可以马上使用本书的理念，而且没有必要采用任何新的框架也不用任何额外的基础设施投资。

大多数的样例代码使用标准的 Java 来论证理念。我也增加了一些基于 OSGi 的代码样例，它们用来阐述你能够从运行时模块化中所获得的收益。所以，书中的理念除了帮助你在当下设计模块化的软件以外，当运行时模块化可用并且你决定使用它的时候，这些理念也会为你做好准备。

InfoQ：模块化只是有关于重用吗？

Kirk：重用是有意思的，因为它具有卖点。我的意思是在过去几十年主要的技术趋势中，都将重用作为其主要的优势。在 90 年代，它是面向对象编程。稍后就是基于组件的开发（component-based development）。然后我们接触到的就是面向服务的架构。

具有讽刺意味的是，它们中的每一个都没有达到预期，众多的开发团队依然在重用中挣扎。重用当然是模块化的优势之一，但它并不是灵丹妙药。模块化填补了其他技术手段无法填补的空白，因为模块化允许你在不同等级的抽象和粒度上来关注设计。如果你阅读本书第二章和第五章的话，你会发现这个空白在哪里以及模块化如何提供帮助。

但是，模块化也有其他的优势。这包括更好的可维护性、依赖管理以及增强理解复杂软件系统的能力。在本书第一部分有几章详细讨论了这些理念。

InfoQ：模块化能够用于已有的系统吗，或者它需要进行预先设计吗？系统能不能重构得更加模块化？

Kirk：如果你在考虑架构和设计的时候，就开始思考模块化的话。这样会更加容易一些。但是，如果你面临的是已有系统，你依然可以对其进行模块化，如果你具备良好的类设计更是如此。实际上，本书的第八章，就会带你做一个练习，我们会使用模式对并不那么模块化的已有系统进行几次重构。最后能够得到高度模块化的系统，这是阐述模块化收益的一个很有力的例子。

在你做这些练习的时候，通常会发现有意思的事情。你会发现在类设计中存在一些你以前所不知道的缺陷。例如，几乎所有的团队都想要对软件系统进行分层。通常，这意味着他们具有展现层、领域层以及数据访问层。但这是逻辑分层。也就是，它们在理念上存在于类结构中，但并不是物理上的可部署单元。

当试图将你的软件划分为模块的时候，你会开始关注可部署单元。当你这样做的时候，你一般会发现这其中存在一些我们事先并不知道的违反逻辑分层的地方。关于这个问题，在物理分层模式中我进行了更多的讨论。

InfoQ：对于 Java 模块化的未来，你怎么看？

Kirk：模块化将会改变我们在 Java 平台上开发和交付软件系统的方式。在开发的角度来看，它填补了其他主要的设计范式所遗留的空白。在软件交付的

角度来看，它标志着庞大和静态平台的终结。这将会促使更为强大的生态系统涌现，开发人员能够很容易地通过仓库提供模块。

鉴于在移动生态系统中应用商店获得了令人兴奋的成功，这种仓库将有可能会成为“企业模块商店”。有些模块可能会是免费的，但是另外一些可能会付费才能使用。这就会允许通过不同的来源将运行时环境拼凑完整。在有些时候，有可能环境本身也是根据应用程序的基础设施需求来进行组装的。

本书中完整的模式列表可以在[本书的站点上](#)找到（以及移动优化版本）。本书的一个样例可以在线获取，书中模式的样例可以在[GitHub 上得到](#)。本书可以在Amazon 上[购买印刷版](#)以及[Kindle](#) 和 [iBooks](#) 格式的电子版。

这些 Q&A 基于 Kirk Knoernschild 编写的《Java 应用架构：以 OSGi 为例讲解模块化模式》一书，该书由 Pearson/Prentice Hall Professional 在 2012 年 3 月出版，ISBN 为 0321247132。关于更多的信息，可以访问该[站点](#)。（译者注：本书的中文版由机械工业出版社华章图书引进，目前本人正在翻译，预计今年 9 月份可以出版，关注模块化和 OSGi 技术的读者敬请期待。最终的中文版书名有待编辑和出版社确认。关于内容和术语的翻译，您如果有好的建议，欢迎与我联系交流。）

关于本书作者



Kirk Knoernschild 是一位软件工程师，热衷于构建伟大的软件。他对设计、架构、应用开发平台、敏捷开发以及 IT 产业的整体情况都有很大的兴趣，尤其是与软件开发相关的内容。他的新书《Java 应用架构：以 OSGi 为例讲解模块化模式》出版于 2012 年，介绍了 18 个帮助你设计模块化软件的模式。你可以访问他的[个人站点](#)。

关于评论者



Dr Alex Blewitt 工作于伦敦的一家投资银行，但依然在抽时间了解 OSGi 与 Eclipse 的新闻。尽管曾经作为 EclipseZone 的编辑和 Eclipse Ambassador 的候选人，但是他的日常角色与 Eclipse 或

Java 毫无关系。除此以外的时间他会用来陪伴自己的家人，并在天气晴好的时候带他们乘飞机旅行。

原文链接 : <http://www.infoq.com/cn/articles/java-application-architecture>

相关内容 :

- [书评 : Java 应用架构](#)
- [标准 Java 模块系统的需求](#)
- [Jigsaw 被推迟到了 Java SE 9](#)
- [Apache Geronimo 3 通过 Java EE 6 Full Profile 认证](#)
- [JavaOne 演讲亮点 : JavaFX 2.0 发布 , Java 9 登场](#)
- [关于 Java 性能的 9 个谬论](#)

欢迎加入

图灵社区 ituring.com.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

推荐文章 | Articles

goroutine 背后的系统知识

作者 孙振南

[Go 语言](#)从诞生到普及已经三年了，先行者大都是 Web 开发的背景，也有了一些普及型的书籍，可系统开发背景的人在学习这些书籍的时候，总有语焉不详的感觉，网上也有若干流传甚广的文章，可其中或多或少总有些与事实不符的技术描述。希望这篇文章能为比较缺少系统编程背景的 Web 开发人员介绍一下 [goroutine](#) 背后的系统知识。

1. 操作系统与运行库

对于普通的电脑用户来说，能理解应用程序是运行在操作系统之上就足够了，可对于开发者，我们还需要了解我们写的程序是如何在操作系统之上运行起来的，操作系统如何为应用程序提供服务，这样我们才能分清楚哪些服务是操作系统提供的，而哪些服务是由我们所使用的语言的运行库提供的。

除了内存管理、文件管理、进程管理、外设管理等等内部模块以外，操作系统还提供了许多外部接口供应用程序使用，这些接口就是所谓的“系统调用”。从 DOS 时代开始，系统调用就是通过软中断的形式来提供，也就是著名的 [INT 21](#)，程序把需要调用的功能编号放入 AH 寄存器，把参数放入其他指定的寄存器，然后调用 INT 21，中断返回后，程序从指定的寄存器(通常是 AL)里取得返回值。这样的做法一直到奔腾 2 也就是 P6 出来之前都没有变，譬如 windows 通过 INT 2E 提供系统调用，Linux 则是 INT 80，只不过后来的寄存器比以前大一些，而且可能再多一层跳转表查询。后来，Intel 和 AMD 分别提供了效率更高的 [SYSENTER/SYSEXIT](#) 和 [SYSCALL/SYSRET](#) 指令来代替之前的中断方式，略过了耗时的特权级别检查以及寄存器压栈出栈的操作，直接完成从 RING 3 代码段到 RING 0 代码段的转换。

系统调用都提供什么功能呢？用操作系统的名字加上对应的中断编号到谷歌上一查就可以得到完整的列表 ([Windows](#), [Linux](#))，这个列表就是操作系统和应用程序之间沟通的协议，如果需要超出此协议的功能，我们就只能在自己的代码里去实现，譬如，对于内存管理，操作系统只提供进程级别的内存段的管理，譬如

Windows 的 [virtualmemory](#) 系列，或是 Linux 的 [brk](#)，操作系统不会去在乎应用程序如何为新建对象分配内存，或是如何做垃圾回收，这些都需要应用程序自己去实现。如果超出此协议的功能无法自己实现，那我们就说该操作系统不支持该功能，举个例子，Linux 在 2.6 之前是不支持多线程的，无论如何在程序里模拟，我们都无法做出多个可以同时运行的并符合 POSIX 1003.1c 语义标准的调度单元。

可是，我们写程序并不需要去调用中断或是 SYSCALL 指令，这是因为操作系统提供了一层封装，在 Windows 上，它是 NTDLL.DLL，也就是常说的 Native API，我们不但不需要去直接调用 INT 2E 或 SYSCALL，准确的说，我们不能直接去调用 INT 2E 或 SYSCALL，因为 Windows 并没有公开其调用规范，直接使用 INT 2E 或 SYSCALL 无法保证未来的兼容性。在 Linux 上则没有这个问题，系统调用的列表都是公开的，而且 Linus 非常看重兼容性，不会去做任何更改，glibc 里甚至专门提供了 [syscall\(2\)](#) 来方便用户直接用编号调用，不过，为了解决 glibc 和内核之间不同版本兼容性带来的麻烦，以及为了提高某些调用的效率（譬如 `_NR gettimeofday`），Linux 上还是对部分系统调用做了一层封装，就是 [VDSO](#)（早期叫 [linux-gate.so](#)）。

可是，我们写程序也很少直接调用 NTDLL 或者 VDSO，而是通过更上一层的封装，这一层处理了参数准备和返回值格式转换、以及出错处理和错误代码转换，这就是我们所使用语言的运行库，对于 C 语言，Linux 上是 glibc，Windows 上是 kernel32（或调用 msvcrt），对于其他语言，譬如 Java，则是 JRE，这些“其他语言”的运行库通常最终还是调用 glibc 或 kernel32。

“运行库”这个词其实不止包括用于和编译后的目标执行程序进行链接的库文件，也包括了脚本语言或字节码解释型语言的运行环境，譬如 Python，C# 的 CLR，Java 的 JRE。

对系统调用的封装只是运行库的很小一部分功能，运行库通常还提供了诸如字符串处理、数学计算、常用数据结构容器等等不需要操作系统支持的功能，同时，运行库也会对操作系统支持的功能提供更易用更高级的封装，譬如带缓存和格式的 IO、线程池。

所以，在我们说“某某语言新增了某某功能”的时候，通常是这么几种可能：

1. 支持新的语义或语法，从而便于我们描述和解决问题。譬如 Java 的泛型、Annotation、lambda 表达式。

2. 提供了新的工具或类库，减少了我们开发的代码量。譬如 Python 2.7 的 argparse
3. 对系统调用有了更良好更全面的封装，使我们可以做到以前在这个语言环境里做不到或很难做到的事情。譬如 Java NIO

但任何一门语言，包括其运行库和运行环境，都不可能创造出操作系统不支持的功能，Go 语言也是这样，不管它的特性描述看起来多么炫丽，那必然都是其他语言也可以做到的，只不过 Go 提供了更方便更清晰的语义和支持，提高了开发的效率。

2. 并发与并行 (Concurrency and Parallelism)

并发是指程序的逻辑结构。非并发的程序就是一根竹竿捅到底，只有一个逻辑控制流，也就是顺序执行的(Sequential)程序，在任何时刻，程序只会处在这个逻辑控制流的某个位置。而如果某个程序有多个独立的逻辑控制流，也就是可以同时处理(deal)多件事情，我们就说这个程序是并发的。这里的“同时”，并不一定要是真正在时钟的某一时刻(那是运行状态而不是逻辑结构)，而是指：如果把这些逻辑控制流画成时序流程图，它们在时间线上是可以重叠的。

并行是指程序的运行状态。如果一个程序在某一时刻被多个 CPU 流水线同时进行处理，那么我们就说这个程序是以并行的形式在运行。(严格意义上讲，我们不能说某程序是“并行”的，因为“并行”不是描述程序本身，而是描述程序的运行状态，但这篇小文里就不那么咬文嚼字，以下说到“并行”的时候，就是指代“以并行的形式运行”) 显然，并行一定是需要硬件支持的。

而且不难理解：

1. 并发是并行的必要条件，如果一个程序本身就不是并发的，也就是只有一个逻辑控制流，那么我们不可能让其被并行处理。
2. 并发不是并行的充分条件，一个并发的程序，如果只被一个 CPU 流水线进行处理(通过分时)，那么它就不是并行的。
3. 并发只是更符合现实问题本质的表达方式，并发的最初目的是简化代码逻辑，而不是使程序运行的更快；

这几段略微抽象，我们可以用一个最简单的例子来把这些概念实例化：用 C 语言写一个最简单的 HelloWorld，它就是非并发的，如果我们建立多个线程，每

个线程里打印一个 HelloWorld，那么这个程序就是并发的，如果这个程序运行在老式的单核 CPU 上，那么这个并发程序还不是并行的。如果我们用多核多 CPU 且支持多任务的操作系统来运行它，那么这个并发程序就是并行的。

还有一个略微复杂的例子，更能说明并发不一定可以并行，而且并发不是为了效率，就是 Go 语言例子里计算素数的 [sieve.go](#)。我们从小到大针对每一个因子启动一个代码片段，如果当前验证的数能被当前因子除尽，则该数不是素数，如果不能，则把该数发送给下一个因子的代码片段，直到最后一个因子也无法除尽，则该数为素数，我们再启动一个它的代码片段，用于验证更大的数字。这是符合我们计算素数的逻辑的，而且每个因子的代码处理片段都是相同的，所以程序非常的简洁，但它无法被并行，因为每个片段都依赖于前一个片段的处理结果和输出。

并发可以通过以下方式做到：

1. 显式地定义并触发多个代码片段，也就是逻辑控制流，由应用程序或操作系统对它们进行调度。它们可以是独立无关的，也可以是相互依赖需要交互的，譬如上面提到的素数计算，其实它也是个经典的生产者和消费者的问题：两个逻辑控制流 A 和 B，A 产生输出，当有了输出后，B 取得 A 的输出进行处理。线程只是实现并发的其中一个手段，除此之外，运行库或是应用程序本身也有多种手段来实现并发，这是下节的主要内容。
2. 隐式地放置多个代码片段，在系统事件发生时触发执行相应的代码片段，也就是事件驱动的方式，譬如某个端口或管道接收到数据(多路 IO 的情况下)，再譬如进程接收到某个信号(signal)。

并行可以在四个层面上做到：

1. 多台机器。自然我们就有了多个 CPU 流水线，譬如 Hadoop 集群里的 MapReduce 任务。
2. 多 CPU。不管是真的多颗 CPU 还是多核还是超线程，总之我们有了多个 CPU 流水线。
3. 单 CPU 核里的 ILP(Instruction-level parallelism)，指令级并行。通过复杂的制造工艺和对指令的解析以及分支预测和乱序执行，现在的 CPU

可以在单个时钟周期内执行多条指令，从而，即使是并发的程序，也可能是以并行的形式执行。

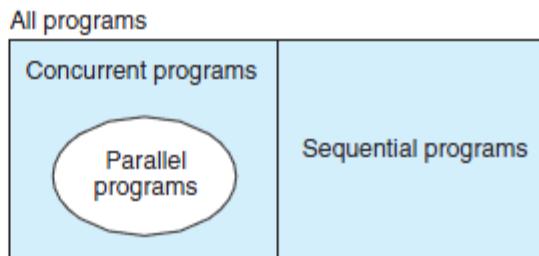
4. 单指令多数据(Single instruction, multiple data. SIMD),为了多媒体数据的处理，现在的CPU的指令集支持单条指令对多条数据进行操作。

其中，1 牵涉到分布式处理，包括数据的分布和任务的同步等等，而且是基于网络的。3 和 4 通常是编译器和 CPU 的开发人员需要考虑的。这里我们说的并行主要针对第 2 种：单台机器内的多核 CPU 并行。

关于并发与并行的问题，Go 语言的作者 Rob Pike 专门就此写过一个幻灯片：
<http://talks.golang.org/2012/waza.slide>

在 CMU 那本著名的《Computer Systems: A Programmer's Perspective》里的这张图也非常直观清晰：

Figure 12.30
 Relationships between
 the sets of sequential,
 concurrent, and parallel
 programs.



3. 线程的调度

上一节主要说的是并发和并行的概念，而线程是最直观的并发的实现，这一节我们主要说操作系统如何让多个线程并发的执行，当然在多 CPU 的时候，也就是并行的执行。我们不讨论进程，进程的意义是“隔离的执行环境”，而不是“单独的执行序列”。

我们首先需要理解 IA-32 CPU 的指令控制方式，这样才能理解如何在多个指令序列(也就是逻辑控制流)之间进行切换。CPU 通过 CS:EIP 寄存器的值确定下一条指令的位置，但是 CPU 并不允许直接使用 MOV 指令来更改 EIP 的值，必须通过 JMP 系列指令、CALL/RET 指令、或 INT 中断指令来实现代码的跳转；在指令序列间切换的时候，除了更改 EIP 之外，我们还要保证代码可能会使用到的各个寄存器的值，尤其是栈指针 SS:ESP，以及 EFLAGS 标志位等，都能够恢复到目标指令序列上次执行到这个位置时候的状态。

线程是操作系统对外提供的服务，应用程序可以通过系统调用让操作系统启动线程，并负责随后的线程调度和切换。我们先考虑单颗单核 CPU，操作系统内核与应用程序其实是在共享同一个 CPU，当 EIP 在应用程序代码段的时候，内核并没有控制权，内核并不是一个进程或线程，内核只是以保护模式运行的，代码段权限为 RING 0 的内存中的程序，只有当产生中断或是应用程序呼叫系统调用的时候，控制权才转移到内核，在内核里，所有代码都在同一个地址空间，为了给不同的线程提供服务，内核会为每一个线程建立一个内核堆栈，这是线程切换的关键。通常，内核会在时钟中断里或系统调用返回前(考虑到性能，通常是在不频繁发生的系统调用返回前)，对整个系统的线程进行调度，计算当前线程的剩余时间片，如果需要切换，就在“可运行”的线程队列里计算优先级，选出目标线程后，则保存当前线程的运行环境，并恢复目标线程的运行环境，其中最重要的，就是切换堆栈指针 ESP，然后再把 EIP 指向目标线程上次被移出 CPU 时的指令。Linux 内核在实现线程切换时，耍了个花枪，它并不是直接 JMP，而是先把 ESP 切换为目标线程的内核栈，把目标线程的代码地址压栈，然后 JMP 到 [switch_to\(\)](#)，相当于伪造了一个 CALL switch_to() 指令，然后，在 switch_to() 的最后使用 RET 指令返回，这样就把栈里的目标线程的代码地址放入了 EIP，接下来 CPU 就开始执行目标线程的代码了，其实也就是上次停在 [switch_to](#) 这个宏展开的地方。

这里需要补充几点：(1) 虽然 IA-32 提供了 TSS (Task State Segment)，试图简化操作系统进行线程调度的流程，但由于其效率低下，而且并不是通用标准，不利于移植，所以主流操作系统都没有去利用 TSS。更严格的说，其实还是用了 TSS，因为只有通过 TSS 才能把堆栈切换到内核堆栈指针 SS0:ESP0，但除此之外的 TSS 的功能就完全没有被使用了。(2) 线程从用户态进入内核的时候，相关的寄存器以及用户态代码段的 EIP 已经保存了一次，所以，在上面所说的内核态线程切换时，需要保存和恢复的内容并不多。(3) 以上描述的都是抢占式 (preemptively)的调度方式，内核以及其中的硬件驱动也会在等待外部资源可用的时候主动调用 [schedule\(\)](#)，用户态的代码也可以通过 [sched_yield\(\)](#) 系统调用主动发起调度，让出 CPU。

现在我们一台普通的 PC 或服务器里通常都有多颗 CPU (physical package)，每颗 CPU 又有多个核 (processor core)，每个核又可以支持超线程 (two logical processors for each core)，也就是逻辑处理器。每个逻辑处理器都有自己的一套完整的寄存器，其中包括了 CS:EIP 和 SS:ESP，从而，以操作系统和应用的

角度来看，每个逻辑处理器都是一个单独的流水线。在多处理器的情况下，线程切换的原理和流程其实和单处理器时是基本一致的，内核代码只有一份，当某个 CPU 上发生时钟中断或是系统调用时，该 CPU 的 CS:EIP 和控制权又回到了内核，内核根据调度策略的结果进行线程切换。但在这个时候，如果我们的程序用线程实现了并发，那么操作系统可以使我们的程序在多个 CPU 上实现并行。

这里也需要补充两点：(1) 多核的场景里，各个核之间并不是完全对等的，譬如在同一个核上的两个超线程是共享 L1/L2 缓存的；在有 NUMA 支持的场景里，每个核访问内存不同区域的延迟是不一样的；所以，多核场景里的线程调度又引入了“调度域”([scheduling domains](#))的概念，但这不影响我们理解线程切换机制。(2) 多核的场景下，中断发给哪个 CPU？软中断(包括除以 0，缺页异常，INT 指令)自然是在触发该中断的 CPU 上产生，而硬中断则又分两种情况，一种是每个 CPU 自己产生的中断，譬如时钟，这是每个 CPU 处理自己的，还有一种是外部中断，譬如 IO，可以通过 APIC 来指定其送给哪个 CPU；因为调度程序只能控制当前的 CPU，所以，如果 IO 中断没有进行均匀的分配的话，那么和 IO 相关的线程就只能在某些 CPU 上运行，导致 CPU 负载不均，进而影响整个系统的效率。

4. 并发编程框架

以上大概介绍了一个用多线程来实现并发的程序是如何被操作系统调度以及并行执行(在有多个逻辑处理器时)，同时大家也可以看到，代码片段或者说逻辑控制流的调度和切换其实并不神秘，理论上，我们也可以不依赖操作系统及其提供的线程，在自己程序的代码段里定义多个片段，然后在我们自己程序里对其进行调度和切换。

为了描述方便，我们接下来把“代码片段”称为“任务”。

和内核的实现类似，只是我们不需要考虑中断和系统调用，那么，我们的程序本质上就是一个循环，这个循环本身就是调度程序 `schedule()`，我们需要维护一个任务的列表，根据我们定义的策略，先进先出或是有优先级等等，每次从列表里挑选出一个任务，然后恢复各个寄存器的值，并且 `JMP` 到该任务上次被暂停的地方，所有这些需要保存的信息都可以作为该任务的属性，存放在任务列表里。

看起来很简单啊，可是我们还需要解决几个问题：

(1) 我们运行在用户态，是没有中断或系统调用这样的机制来打断代码执行的，那么，一旦我们的 schedule() 代码把控制权交给了任务的代码，我们下次的调度在什么时候发生？答案是，不会发生，只有靠任务主动调用 schedule()，我们才有机会进行调度，所以，这里的任务不能像线程一样依赖内核调度从而毫无顾忌的执行，我们的任务里一定要显式的调用 schedule()，这就是所谓的协作式 (cooperative) 调度。（虽然我们可以通过注册信号处理函数来模拟内核里的时钟中断并取得控制权，可问题在于，信号处理函数是由内核调用的，在其结束的时候，内核重新获得控制权，随后返回用户态并继续沿着信号发生时被中断的代码路径执行，从而我们无法在信号处理函数内进行任务切换）

(2) 堆栈。和内核调度线程的原理一样，我们也需要为每个任务单独分配堆栈，并且把其堆栈信息保存在任务属性里，在任务切换时也保存或恢复当前的 SS:ESP。任务堆栈的空间可以是在当前线程的堆栈上分配，也可以是在堆上分配，但通常是在堆上分配比较好：几乎没有大小或任务总数的限制、堆栈大小可以动态扩展(gcc 有 split stack，但太复杂了)、便于把任务切换到其他线程。

到这里，我们大概知道了如何构造一个并发的编程框架，可如何让任务可以并行的在多个逻辑处理器上执行呢？只有内核才有调度 CPU 的权限，所以，我们还是必须通过系统调用创建线程，才可以实现并行。在多线程处理多任务的时候，我们还需要考虑几个问题：

(1) 如果某个任务发起了一个系统调用，譬如长时间等待 IO，那当前线程就被内核放入了等待调度的队列，岂不是让其他任务都没有机会执行？

在单线程的情况下，我们只有一个解决办法，就是使用非阻塞的 IO 系统调用，并让出 CPU，然后在 schedule() 里统一进行轮询，有数据时切换回该 fd 对应的任务；效率略低的做法是不进行统一轮询，让各个任务在轮到自己执行时再次用非阻塞方式进行 IO，直到有数据可用。

如果我们采用多线程来构造我们整个的程序，那么我们可以封装系统调用的接口，当某个任务进入系统调用时，我们就把当前线程留给它（暂时）独享，并开启新的线程来处理其他任务。

(2) 任务同步。譬如我们上节提到的生产者和消费者的例子，如何让消费者在数据还没有被生产出来的时候进入等待，并且在数据可用时触发消费者继续执行呢？

在单线程的情况下，我们可以定义一个结构，其中有变量用于存放交互数据本身，以及数据的当前可用状态，以及负责读写此数据的两个任务的编号。然后我们的并发编程框架再提供 read 和 write 方法供任务调用，在 read 方法里，我们循环检查数据是否可用，如果数据还不可用，我们就调用 schedule() 让出 CPU 进入等待；在 write 方法里，我们往结构里写入数据，更改数据可用状态，然后返回；在 schedule() 里，我们检查数据可用状态，如果可用，则激活需要读取此数据的任务，该任务继续循环检测数据是否可用，发现可用，读取，更改状态为不可用，返回。代码的简单逻辑如下：

```

struct chan {
    bool ready,
    int data
};

int read (struct chan *c) {
    while (1) {
        if (c->ready) {
            c->ready = false;
            return c->data;
        } else {
            schedule();
        }
    }
}

void write (struct chan *c, int i) {
    while (1) {
        if (c->ready) {
            schedule();
        } else {
            c->data = i;
            c->ready = true;
            schedule(); // optional
            return;
        }
    }
}

```

很显然，如果是多线程的话，我们需要通过线程库或系统调用提供的同步机制来保护对这个结构体内数据的访问。

以上就是最简化的一个并发框架的设计考虑，在我们实际开发工作中遇到的并发框架可能由于语言和运行库的不同而有所不同，在功能和易用性上也可能各有取舍，但底层的原理都是殊途同归。

譬如，glibc 里的 [getcontext/setcontext/swapcontext](#) 系列库函数可以方便的用来保存和恢复任务执行状态；Windows 提供了 Fiber 系列的 SDK API；这二者都不是系统调用，[getcontext](#) 和 [setcontext](#) 的 man page 虽然是在 section 2，但那只是 SVR4 时的历史遗留问题，其实现代码是在 glibc 而不是 kernel；[CreateFiber.aspx](#) 是在 kernel32 里提供的，NTDLL 里并没有对应的 NtCreateFiber。

在其他语言里，我们所谓的“任务”更多时候被称为“协程”，也就是 Coroutine。譬如 C++ 里最常用的是 Boost.Coroutine；Java 因为有一层字节码解释，比较麻烦，但也有支持协程的 JVM 补丁，或是动态修改字节码以支持协程的项目；PHP 和 Python 的 generator 和 yield 其实已经是协程的支持，在此之上可以封装出更通用的协程接口和调度；另外还有原生支持协程的 Erlang 等，笔者不懂，就不说了，具体可参见 Wikipedia 的页面：<http://en.wikipedia.org/wiki/Coroutine>

由于保存和恢复任务执行状态需要访问 CPU 寄存器，所以相关的运行库也都会列出所支持的 CPU 列表。

从操作系统层面提供协程以及其并行调度的，好像只有 OS X 和 iOS 的 [Grand Central Dispatch](#)，其大部分功能也是在运行库里实现的。

5. goroutine

Go 语言通过 goroutine 提供了目前为止所有(我所了解的)语言里对于并发编程的最清晰最直接的支持，Go 语言的文档里对其特性也描述的非常全面甚至超过了，在这里，基于我们上面的系统知识介绍，列举一下 goroutine 的特性，算是小结：

1. goroutine 是 Go 语言运行库的功能，不是操作系统提供的功能。goroutine 不是用线程实现的。具体可参见 Go 语言源码里的 [pkg/runtime/proc.c](#)
2. goroutine 就是一段代码，一个函数入口，以及在堆上为其分配的一个堆栈。所以它非常廉价，我们可以很轻松的创建上万个 goroutine，但它们并不是被操作系统所调度执行
3. 除了被系统调用阻塞的线程外，Go 运行库最多会启动 \$GOMAXPROCS 个线程来运行 goroutine

4. goroutine 是协作式调度的，如果 goroutine 会执行很长时间，而且不是通过等待读取或写入 channel 的数据来同步的话，就需要主动调用 [Gosched\(\)](#) 来让出 CPU
5. 和所有其他并发框架里的协程一样，goroutine 里所谓“无锁”的优点只在单线程下有效，如果 \$GOMAXPROCS > 1 并且协程间需要通信，Go 运行库会负责加锁保护数据，这也是为什么 sieve.go 这样的例子在多 CPU 多线程时反而更慢的原因
6. Web 等服务端程序要处理的请求从本质上讲是并行处理的问题，每个请求基本独立，互不依赖，几乎没有数据交互，这不是一个并发编程的模型，而并发编程框架只是解决了其语义表述的复杂性，并不是从根本上提高处理的效率，也许是并发连接和并发编程的英文都是 concurrent 吧，很容易产生“并发编程框架和 coroutine 可以高效处理大量并发连接”的误解。
7. Go 语言运行库封装了异步 IO，所以可以写出貌似并发数很多的服务端，可即使我们通过调整 \$GOMAXPROCS 来充分利用多核 CPU 并行处理，其效率也不如我们利用 IO 事件驱动设计的、按照事务类型划分好合适比例的线程池。在响应时间上，协作式调度是硬伤。
8. goroutine 最大的价值是其实现了并发协程和实际并行执行的线程的映射以及动态扩展，随着其运行库的不断发展和完善，其性能一定会越来越好，尤其是在 CPU 核数越来越多的未来，终有一天我们会为了代码的简洁和可维护性而放弃那一点点性能的差别。

感谢[孙振南](#)的投稿，大家可以在[这里](#)查看[原文](#)。

原文链接：<http://www.infoq.com/cn/articles/knowledge-behind-goroutine>

相关内容：

- [goroutine 背后的系统知识](#)
- [Google 即将发布 Go 语言 1.1 版，含多项重大更新](#)
- [Iron.io 从 Ruby 迁移到 Go：减少了 28 台服务器并避免了连锁故障](#)
- [百度基础体系首席架构师林仕鼎分享系统架构领域论文精要](#)
- [Go，基于连接与组合的语言（上）](#)
- [Go 语言 Interface 漫谈](#)

推荐文章 | Articles

聊聊并发（三）——JAVA 线程池的分析和使用

作者 方腾飞

1. 引言

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

2. 线程池的使用

线程池的创建

我们可以通过 ThreadPoolExecutor 来创建一个线程池。

```
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds, runnableTaskQueue, handler);
```

创建一个线程池需要输入几个参数：

- corePoolSize (线程池的基本大小) : 当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的 prestartAllCoreThreads 方法，线程池会提前创建并启动所有基本线程。
- runnableTaskQueue (任务队列) : 用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。
 - ArrayBlockingQueue : 是一个基于数组结构的有界阻塞队列，此队列按 FIFO (先进先出) 原则对元素进行排序。

- LinkedBlockingQueue : 一个基于链表结构的阻塞队列 , 此队列按 FIFO (先进先出) 排序元素 , 吞吐量通常要高于 ArrayBlockingQueue 。静态工厂方法 Executors.newFixedThreadPool() 使用了这个队列。
- SynchronousQueue : 一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作 , 否则插入操作一直处于阻塞状态 , 吞吐量通常要高于 LinkedBlockingQueue , 静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。
- PriorityBlockingQueue : 一个具有优先级的无限阻塞队列。
- maximumPoolSize (线程池最大大小) : 线程池允许创建的最大线程数。如果队列满了 , 并且已创建的线程数小于最大线程数 , 则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
- ThreadFactory : 用于设置创建线程的工厂 , 可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
- RejectedExecutionHandler (饱和策略) : 当队列和线程池都满了 , 说明线程池处于饱和状态 , 那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 AbortPolicy , 表示无法处理新任务时抛出异常。以下是 JDK1.5 提供的四种策略。
 - AbortPolicy : 直接抛出异常。
 - CallerRunsPolicy : 只用调用者所在线程来运行任务。
 - DiscardOldestPolicy : 丢弃队列里最近的一个任务 , 并执行当前任务。
 - DiscardPolicy : 不处理 , 丢弃掉。
 - 当然也可以根据应用场景需要来实现 RejectedExecutionHandler 接口自定义策略。如记录日志或持久化不能处理的任务。
- keepAliveTime (线程活动保持时间) : 线程池的工作线程空闲后 , 保持存活的时间。所以如果任务很多 , 并且每个任务执行的时间比较短 , 可以调大这个时间 , 提高线程的利用率。
- TimeUnit (线程活动保持时间的单位) : 可选的单位有天 (DAYS) , 小时 (HOURS) , 分钟 (MINUTES) , 毫秒 (MILLISECONDS) , 微秒 (MICROSECONDS , 千分之一毫秒) 和毫微秒 (NANOSECONDS , 千分之一微秒) 。

向线程池提交任务

我们可以使用 execute 提交的任务，但是 execute 方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知 execute 方法输入的任务是一个 Runnable 类的实例。

```
threadsPool.execute(new Runnable() {
    @Override
    public void run() {
        // TODO Auto-generated method stub
    }
});
```

我们也可以使用 submit 方法来提交任务，它会返回一个 future，那么我们可以通过这个 future 来判断任务是否执行成功，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用 get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

```
Future<Object> future = executor.submit(harReturnValuetask);
try {
    Object s = future.get();
} catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
    // 处理无法执行任务异常
} finally {
    // 关闭线程池
    executor.shutdown();
}
```

线程池的关闭

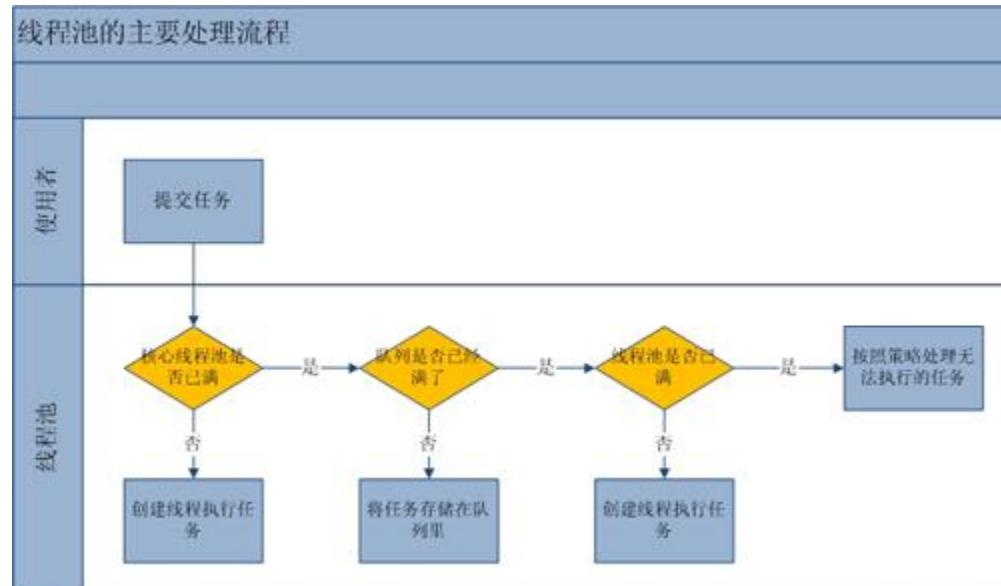
我们可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow 首先将线程池的状态设置成 STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而 shutdown 只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown 方法就会返回 true。当所有的任务都已关闭后，才表示线程池关闭成功，这时调用 isTerminated 方法会返

回 true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用 shutdown 来关闭线程池，如果任务不一定要执行完，则可以调用 shutdownNow。

3. 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

- 首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
- 其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
- 最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

源码分析。上面的流程分析让我们很直观的了解了线程池的工作原理，让我们再通过源代码来看看是如何实现的。线程池执行任务的方法如下：

```

public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    //如果线程数小于基本线程数，则创建线程并执行当前任务
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        //如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        //如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，则创建一个线程执行任务。
        else if (!addIfUnderMaximumPoolSize(command))
            //抛出RejectedExecutionException异常
            reject(command); // is shutdown or saturated
    }
}
    
```

工作线程。线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从 Worker 的 run 方法里看到这点：

```

public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    } finally {
        workerDone(this);
    }
}
    
```

4. 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

1. 任务的性质：CPU 密集型任务，IO 密集型任务和混合型任务。
2. 任务的优先级：高，中和低。
3. 任务的执行时间：长，中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU 密集型任务配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。IO 密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如 2*Ncpu。混合型的任务，

如果可以拆分，则将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过 Runtime.getRuntime().availableProcessors()方法获得当前设备的 CPU 个数。

优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，如果等待的时间越长 CPU 空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用 CPU。

建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行 SQL 变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

5. 线程池的监控

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

1. taskCount：线程池需要执行的任务数量。
2. completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于 taskCount。

3. largestPoolSize : 线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
4. getPoolSize: 线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不减。getActiveCount : 获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的 beforeExecute , afterExecute 和 terminated 方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。如：

```
protected void beforeExecute(Thread t, Runnable r) { }
```

6. 参考资料

Java 并发编程实战。

JDK1.6 源码

作者介绍

方腾飞，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：<http://ifeve.com> 微博：<http://weibo.com/kirals> 欢迎通过我的微博进行技术交流。

原文链接：<http://www.infoq.com/cn/articles/java-threadPool>

相关内容：

- [聊聊并发（三）——JAVA 线程池的分析和使用](#)
- [深入理解 Java 内存模型（七）——总结](#)
- [深入理解 Java 内存模型（六）——final](#)
- [深入理解 Java 内存模型（五）——锁](#)
- [聊聊并发（六）——ConcurrentLinkedQueue 的实现原理分析](#)

● [聊聊并发（五）——原子操作的实现原理](#)

推荐文章 | Articles

豌豆英王俊煜：技术突破可能性和高效能 Geek 团队

作者 贾国清

豌豆们最近在忙些什么，在组建团队的过程中，他们是如何来判断并找到正确的人，豌豆英的 Geek 或 Hacker 文化又是怎样的，他们又是如何激励豌豆们的成长呢？希望通过本文的内容，像「豌豆洗白白」一样——让一切“真相大白”。

了解豌豆英的朋友想必都会知道，帮助用户面对复杂 Android 应用环境、更简单地获取到优质应用，一直是[豌豆英](#)致力解决的问题。对产品精益求精的他们偶尔也会开些比较 Geek 的玩笑，比如今年愚人节期间发布「空气洗白白」来解决 PM2.5 的问题，笔者就被“愚弄”了一下，收到了 2 包装的新鲜空气，正是这样的一个团队，在无时无刻的通过各种方式，来表达他们的愿望和价值观。

今年年初推出的[「豌豆洗白白」](#)，也是为了帮助用户通过简单的方式一键“洗白”手机上的山寨和广告应用。同时，豌豆英一直在改善和扩充「豌豆洗白白」的功能。包括不断提高对山寨和广告应用辨识的准确度、提高应用替换方案的用户接受度，以及在山寨、广告的维度基础上增加对有隐私风险、有性能问题的应用的判断和解决。根据王俊煜描述：

「豌豆洗白白」是我们在解决应用获取的问题上同用户沟通最直接的一环，但其实它所运用的标准和逻辑在我们的应用推荐和搜索环节也都会完整地运用。

另外，最近我们依然在不断改进产品的基础体验，比如说提升 Windows 客户端的性能问题，比如提高电脑和手机的连接效率、传输速度等基本性的指标。这些可能都不是足以让人“眼前一亮”的新功能，但是是每一个用户都能切实感受到豌豆英“简单”“好用”的产品细节。

在工作中，几乎每个团队经常会有一些“为了能够完成工作而需要做的工作”，例如各种以沟通为目的会议、写各种周报、做各种汇报、绩效评估等等。这些工作不可能完全消除，其中的一些是会对效率有促进作用的，也有很大一部分是通过工具、更好的配合等等可以消除的、实际上不必要的“损耗”。王俊煜认为：

高效率的团队，应该是越少这种“损耗”越好。从团队的角度来讲，最终是要解决有没有邀请正确的同学加入的问题。团队成长得越大，反而越意识到组建“A-team”的重要性，发展过程中有时候因为短期压力对团队组建的标准有妥协，事后一看往往都是因小失大的，反而需要花费更多的时间来止损。偶尔有不合适同学加入，早期不下决心请人离开，时间越长、破坏越大。这些其实都是经常听别人分享经验的时候会谈到的，不过确实只有自己都尝试过，才会有体会。

关于 Geek 和 Hacker

关于 Geek 和 Hacker，王俊煜引用了李如一在某次采访中抛出的概念：“用技术去 make impact 的一群人”。首先“Geek”是一个正面的定义，他们最终的目标是要做一些有影响力的事情。当然，他们做事的方式有点古怪，可能是用很 Technical 的方法去解决不怎么 Technical 的问题。

豌豆莢内部有很多这样的事情，比如为了提高阿姨的工作效率，一位豌豆为她开发了一个订饭网站叫「喂豌豆」，每天下午两点会发邮件询问每个人吃什么，并自动汇总订单。

真正的 Geek 往往是这样的，聊天的时候只要从项目进入技术细节，你甚至只需要稍微多问几句，他都会变得很兴奋。相信在中国这样的人是足够多的，可能之前没有太多地方为他们提供好的机会。当然，即使员工变得越来越多，豌豆莢也会将 Geek 文化一直保持下去。

那么，从哪些途径可以找到想要的 Geek 呢？豌豆莢现在主要靠猎头，以前熟人推荐是最有用的。不过猎头往往通过非常硬性的标准去判断人选，推荐的质量确实没有熟人推荐要好。豌豆莢也会在一些特定的渠道放出招聘，比如从 Twitter 过来的人通常会比从微博过来的 Geek 一点；包括在知乎看问答，也会找到合适的人选。

关于如何制定一个适合 Geek 文化的工作目标？王俊煜认为这点很难做到，因为 Geek 多数很难“manage”，就好像很多人都是白天不干活，晚上拼命干一样。豌豆莢整个的目标体系（OKRs）基本还是由员工自己给自己制定为主，起到一个提醒作用，帮助大家变得更可预期一点点。

怎样让一枚 Geek 走上管理者的位置 , 怎样才能解除他可能的不适应 ? 暂时看来 , 还没有更好的办法 , 王俊煜如是说 , 100 人的公司还不需要很多传统意义上的“管理”的样子。再者 , Geek 也可以用 Geek 的方式来达到他要的目标。

在豌豆荚内部 , 现在已经保持了很好的 Geek 文化 , Hackday 就是其中之一。

作为公司 , 组织 Hackday 的目的是给工程师们一个特定的时间 , 去实现他们平时工作中想到的一个很牛逼的技术和想法。这些技术和想法可能由于平时工作时间太紧 , 或者可行性没有得到验证 , 工程师不能直接实施到商业产品中 (或者和他做的产品根本没有关系) 。但在 Hackday 期间 , 公司允许工程师去尝试任何疯狂 , 酷炫 , 或者看起来可笑的想法 , 只要有人愿意去尝试。

这些想法不一定和实际的工作相关 , 每次 Hackday 需要有个主题 , 但作为组织者 , 不要限定参与者只做和工作相关的事情。说白了就是这段时间就是工程师“不务正业”的时间。是他们完全自己掌控的时间 , 他们编程的所有动机 , 应该来自于兴趣和乐趣。他们的组队也是按照自己的意愿。

比如说豌豆荚最近一次的 Hackday 的主题是“技术突破可能性”。这个主题非常开放 , 总体就是要大家去尝试以前一直认为不可能实现的技术和想法。豌豆荚还有一个主题的 candidate 是“愚人节” , 大家做一些愚人节能够“愚人” , 但又好玩的东西。当然 奖励也是很 Geek 的东西 , 比如豌豆荚这次的一等奖是 Google Glass 。

嗯 , 最后我们为第一名的 Team 打了最有诚意的白条 , 因为 Google Glass 还没量产呢。

愚人节 Hack 已经逐渐成为豌豆荚的传统 , 每年都会有策划。对于科技公司而言 , 愚人节是一个不容错过的机会 , 豌豆荚也希望借这个机会把他们的情感和价值观真实的表达出来。不仅是对外 , 还有对内。

豌豆荚的价值观是「简单、有爱」 , 这需要传递给每一个用户 , 以及每一颗豌豆。今年的「空气洗白白 plus 」就是这样一次尝试 (视频 : [空气洗白白发布会](#)) , 我们制作 [视频](#)、 [Landing Page](#) 并且通过微博传播 , 让用户被小小的“愚弄”一次 , 同时也为内部员工准备了 4 月 1 日的活动。

内部活动的视频我们也通过官方微博放了出来 , 形式是骗参与者吸入氦气和六氟化硫说话变声 , 给他们一个 surprise 。大家玩得很开心 , 同时也再一次能感受到豌豆荚的文化和价值观。

找到对的人并激励他们

趋势专家、畅销书作者丹尼尔·平克 (Daniel H. Pink) 曾在 [《驱动力》](#) 中写到：

奖励有时候很奇怪，它就像对人的行为施了魔法：把有意思的工作变成了苦工，把游戏变成了工作。——奖励让内在动机（激励因素）消失了。奖励会使人关注面变窄，遮蔽他们宽广的视野，让他们没法看到常见事物的新用法。

那么，豌豆荚又是如何找到正确的人，又是如何帮助他们找到合适的岗位，并进行有效的激励呢？

王俊煜比较关心每个人的内在驱动力是从何而来，是不是和众多豌豆们一样是抱着希望创造伟大事物的目标和热情而来，是不是一样喜欢用创造式的方式解决新问题。

在面试时，他经常会问到对方的问题是：你做过的最棒的 Project 是什么？它为什么是最棒的？最后你可能会得到很多种答案，比如它曾经被表扬过，或者说帮助同事解决了什么问题，也有可能是某个技术或者挑战性的难题被解决掉。这些回答会反映出每个人内在驱动力的不同。

在分工上，豌豆荚没有特别明确的界限，所以大家会有比较自由的空间去试试不是自己的本职工作，但又感兴趣的那部分。

在很多公司中，员工做的事情是一一对应的，比如前端就是前端，前后的步骤都会交给相应职位的人员去做。但是在豌豆荚，如果你愿意多做一些，并且可以证明自己做得不错，一点问题都没有。

有不止一位豌豆在工程师和产品设计师之间有过相互转换的经历。唯一不希望出现的情况是，工程师凡是和产品有关的决定都要等产品设计师做，产品设计师一行代码都不写，豌豆荚一定是鼓励混搭的。

我们就有一位前端工程师因此将更多的精力放到了产品设计，现在 [wandoujia.com](#) 的所有事情几乎都是他自己负责，包括前端和产品。在这种情况下，豌豆荚不会特别刻意的宣布职位调动，这些都是很自然的过渡。我们的 Marketing 甚至也会做一些产品设计师的事，大家边界都模糊一点，只要你想做更多的事情还是蛮方便的。当然，愿意专注自己的领域也完全没有问题。

对了，刚才提到的这名前端豌豆还将工作地点放到了杭州。因为家庭原因他不得不离开北京，当时也在杭州找好了工作，但他本人还是很想继续留在豌豆荚。那我们就一起找一个解决方案吧，最后的决定是，他可以异地工作，每个月回来与大家见一次。目前这个状态已经维持至少半年了，我们之间的合作也没有什么问题。

在极客公园年会上，关于如何通过工具来提高生产力，记得王俊煜当时是这么说的：“生产力”其实更像是一门“副科”，你平时不会关心，但又不能不及格。

对于团队成员的激励和成长，豌豆荚会根据自己的实际情况制定一些简单的激励体系，不过归根结底，包括对王俊煜自己来说，最大的激励还是来自于能不能在这里创造出对人类有价值的事物，并在这其中提升自己的能力。他相信这是最大、最自然的激励。

谈到具体的激励体系，就要从每月的一次 Nb 奖说起，Nb 奖会颁发给当月对公司或者行业特别有贡献的事情，得奖对象是参与到这个事情中的每一个人。

Peer Bonus，具体的解释是这样的：如果你觉得某人在他的职责之外或者预期之外做多了很多，特别是给你自己的工作提供了巨大的帮助，那么申请一个 500 元的 Peer Bonus 给这个帮助了你的豌豆，这种方式说一声“真的很感谢”。所以应该有两个要素，一个是这件事情本身应该是极大超出对此人工作的预期和职责，不管是在广度上还是在深度上；另外一个是，提名人是直接受益人。

Perf Review，每年一次。每年年底豌豆荚会做一个非常多维度的 Review，从每个人、每个项目到公司整体都有涉及。它包含很多部分，包括判断公司、项目和个人在这一年的目标是否达成，并且根据自己的 OKR(OKR，Objective and Key Result，也就是豌豆荚的目标体系)去做打分。

简单介绍一下，比如每个人都会有三个人去为他做 Review，有相同 Team 的人，也有其他 Team 但同一个项目的合作者，这样做的目的是让每个人找出自己这一年做对了什么，以及还可以改进什么，然后综合这些评价，Team Leader 会为这名豌豆写一个 Review，并且打出综合的分数。

OKRs 和 Peer Bonus 都是从 Google 借鉴来的，管理上灵活的方式则是从 Facebook 得到的灵感。

前两天看到一篇文章，标题大概是“所谓管理，是你招人烂的表现”，可能有点极端，但我还蛮同意的。

员工增加之后豌豆荚也在找平衡的点，不是大家完全想做什么就做什么。主要的目的是，让豌豆们明白自己的贡献和突破的方向，而 Leader 们会在其中帮忙沟通，砍掉边角的事情，帮助他们把工作理得更 focus 一点。

说实在的，这些其实都不是“管理”的感觉。

其次，大家自己从用户反馈、数字增长里是看得见价值的，豌豆们认同这个价值，因此也会有激励的作用。

我们去年夏天做过一个 Workshop，那个时候公司面临一些选择和困惑，创始人们开始寻找解决问题的方式。最具体的问题是，对于豌豆荚真正的 mission 和价值观，似乎大家都知道，但并没有非常明确的一致性的细节。做 Workshop 的契机就是想让大家达成统一，然后我们找到了[青年志](#)，他们帮诺基亚、Nike 都做过一些策略性的事情。

青年志的方法是，他们并不试图告诉你什么，而是引导大家把心里想的东西说出来。当时豌豆荚有一半的人去参加了这个活动，关在一个屋子里做了一整天。首先用一上午的时间去思考自己想成为什么样的人，希望为这个社会和世界带来什么，并且用一幅画去表示自己的愿景。最后的结果表示出大部分人还是希望在自己喜欢的领域做出贡献，并且成为很牛逼的存在。所以现在当他们听到身边的人在用豌豆荚，看到用户量上涨，都会蛮开心的。

当时第二个问题是：你认为豌豆荚能为这个世界带来什么？把类似答案的人分成了不同的 Team，让大家去 Brainstorm 豌豆荚这家公司的 mission 和 vision。最后经过投票，大家最认同的词是“pioneer”。也就是说，大家希望豌豆荚能够成为科技领域的先行者，探索和发现新的东西，并让行业更加向前。

很难讲这个 Workshop 是不是真的对所有人都输出了东西，发挥了什么效果，这些本来就无法被量化。但至少它让相对比较核心的豌豆荚成员更清楚了员工和自己的想法，在之后的前进方向中，我们也会根据这些想法做出判断和调整。

在访谈最后，王俊煜提到早年时每次遇到困难，心里面都会想，过了这关以后就会好走一些了。没有这样的事情。如果是一支一直都不想停下来的团队，在持续的快速成长当中一定会不断遇到困难——而且都是有生命危险的困难。要做好这个准备。

接下来豌豆荚会做些什么

豌豆荚一直致力解决的是智能设备内容下载和消费中会遇到的方方面面的问题，让那些看似复杂的智能设备简单、好用。

据王俊煜透露，短期内仍会继续聚焦在 Android 领域，对内容的定义主要也还是应用。豌豆洗白白就是在解决了应用全、质量好之后，彻底帮助用户摆脱「问题应用」的产品。一方面通过更加简单的产品帮助用户更好的使用 Android 手机；另一方面，豌豆荚也一直相信 Android 的世界或许是复杂的，但一定不是危险的。他们不希望用户因为所谓的「山寨、隐私」问题，就彻底放弃使用高质量的应用甚至 Android 手机。

最后，祝愿这支有理想、肯坚持、不断探索的年轻团队，能够梦想成真，在其他内容领域能够进行更多的探索性尝试，让用户能更简单的管理手机、获取内容！

原文链接：<http://www.infoq.com/cn/articles/wandoujia-a-geek-team>

相关内容：

- [豌豆荚王俊煜：技术突破可能性和高效能 Geek 团队](#)
- [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》](#)
- [敏捷咨询师许晓斌指出结对编程顺利推行的四个原因](#)
- [为完全分散式团队提高生产率和效率](#)
- [访谈《iTeams——把个人放回团队中》作者 William E. Perry](#)

推荐编辑 | 翻译团队编辑方盛



Hi All。我叫方盛，能受邀成为本期《架构师》的推荐编辑真是让我受宠若惊，但是在惶恐之余更多的还是兴奋，能够通过这次机会跟更多的朋友相识，真是小弟毕生之幸。

结识 InfoQ 是在一个很偶然的机会，在这里我需要好好感谢我的好哥们——老雷，是他带我走进了 InfoQ 的世界，让我知道除了每天写代码，打 Dota 之外，还有更有意义的事情可以做。其实自己以前就有在业余时间做点事情的想法，但是一直都没有机会，结识 InfoQ 以后，我觉得我的业余生活找到了目标。在 InfoQ 的站点上不仅可以获取更多的业内前沿知识和动态，也在专业技能提升上给自己带来不小的帮助，还可以结交许多来自五湖四海的朋友，以及社区里的名人，真是人生一大幸事。

自从 12 年 10 月份加入 InfoQ 以来，我就感觉这是一个严谨务实、团结活泼的团队，每次大家在 QQ 群里的“咬文嚼字”，那都是认真务实的一种表现，都是对翻译事业的一种尊重和敬畏。特别是臧秀涛童鞋等人，每次审校我的译文都是批注得“面目全非”，虽然文档上是花花绿绿的，但是他们在做翻译时的精益求精之心却跟这些批注一样，是一目了然的。就是这样一个在学术上极其严谨的团队，其实在生活中也是群团结活泼的人。

在前段时间刚刚闭幕的 QCon 北京 2013 上，我有幸来到北京参与了这场技术盛宴，在会议的休息时间，我见到了传说中的水羽哲——水哥和杨赛——赛姐，以及臧秀涛和李彬等人（希望“等人”不要怪我，有点数不过来了），从看大家第一眼就知道这是一个活力四射的群体，看到大家在会场上忙碌的身影，就知道这是一个团结敬业的团队。在社区里大家群策群力，每次针对一个词一句话的激烈讨论都是为了将更贴近原文的意思传递给大家，从而发扬 InfoQ “促进软件开发领域知识和创新的传播”的精神；在会场上，每一场会议的精心准备都是为了更好的服务大家，让大家可以尽情的享受讲师们带给大家的视听盛宴。此时此刻，我觉得，成为 InfoQ 的一员，我是幸运的。

“促进软件开发领域知识和创新的传播”，这是一句很多人会喊，但是却鲜有人会为之付出行动的口号，但是 InfoQ 做到了，每次我在进行翻译工作的时候，我觉得我的热情和积极性都超过了编写代码的时候，甚至翻译到深夜也不知疲倦，我觉得自己不仅仅是简单的将 ABC 翻译成你我他，而是将思想在进行传递，将

国外的优秀思想传递到国内。作为 InfoQ 的译者，我更觉得自己是一名思想传递者，如果此时你也心动了，就请加入 InfoQ 的大家庭。

以上就是本人在深夜写下的一些闲言碎语，如果大家希望可以跟我进行更多的交流可以通过新浪微博@InfoQ_Stifler 或者邮箱：fangsheng1987@gmail.com 联系到我。如果你有意加入我们成为一名思想传递者请邮件至：
editors@cn.infoq.com。

封面植物 | 鸢尾花



鸢尾花是鸢尾属植物，是对一族草本开花植物的统称。“鸢尾”之名来源于希腊语，意思是彩虹。它表明天上彩虹的颜色尽可在这个属的花朵颜色中看到。鸢尾花在我国常用以象征爱情和友谊，鹏程万里，前途无量明察秋毫。这种花由 6 个花瓣状的叶片构成的包膜，3 个或 6 个雄蕊和由花蒂包着的子房组成。香气淡雅，可以调制香水。分布于日本、中国中部、西伯利亚、法国和几乎整个温带世界。是法国国花。形态特征：年生矮小草本，植株基部淡绿色，围有 3-5 枚鞘状叶及少量的老叶残留纤维。根状茎细长，坚韧，二歧状分枝，横走，棕黄色，节处膨大；须根细弱，生于节处，棕黄色。叶狭条形，黄绿色，花期叶长 5-20 厘米，宽 1-2.5 毫米，果期长可达 40 厘米，宽达 7 毫米，顶端长渐尖，基部鞘状，有 1-2 条纵脉。花茎高 5-7 厘米，中下部有 1-2 枚鞘状的茎生叶；苞片 2 枚，草质，绿色，狭披针形，长 3.5-5.5 厘米，宽约 6 毫米，顶端渐尖，内包含有 1 朵花；花淡蓝紫色，直径 3.5-4 厘米；花梗长 0.6-1 厘米；花被管长 2.5-3 (5) 厘米，外花被裂片倒卵形，长约 2.5 厘米，宽 1-1.2 厘米，盛开时上部平展，有马蹄形的斑纹，爪部楔形，中脉上有黄色的鸡冠状附属物，表面平坦似毡绒状，内花被裂片倒披针形，长 2.2-2.5 厘米，宽约 7 毫米，直立；雄蕊长约 1 厘米，花丝及花药皆为白色；花柱分枝淡蓝紫色，长约 1.8 厘米，宽约 4 毫米，顶端裂片长三角形，外缘有不明显的疏齿，子房绿色，圆柱形，长 4-5 毫米。蒴果圆球形，直径 1.2-1.5 厘米，顶端有短喙；果梗长 1-1.3 厘米，苞片宿存于果实基部。花期 3-4 月，果期 5-7 月。药效作用：1. 消炎作用：鸢尾黄酮甙，在试管中有抗透明质酸酶的作用，而且不为半胱氨酸所阻断，它还能抑制大鼠的透明质酸性的浮肿而不抑制角叉菜胶性浮肿，对鼠因腹腔注射氮芥引起的腹水渗出亦有抑制作用。2. 其他作用：鸢尾黄酮甙，能促进家兔唾液分泌，注射较口服的作用更快而强。

架构师

www.infoq.com/cn/architect

每月8号出版

时刻关注软件开发领域的变化与创新

架构师

11月 ARCHITECT

特别专题
光棍节狂欢的背后——
电商系统深探
1号店B2C电商系统深造之路
百万点推荐引擎——从需求到架构
麦当劳购物系统浅谈分享
REST的远程API设计案例
大型Rails与VoIP系统架构
与部署实践
什么是Node.js
扩展Oozie
浅谈dojox中的一些小工具

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

10月 ARCHITECT

特别专题
大数据时代
大数据
大数据时代的数据管理
阿里巴巴数据架构设计经验与挑战
大数据时代的创新者们
关系数据库还是NoSQL数据库
向Java开发者介绍Scala
HTML 5 or Silverlight?
解析JDK 7的Garbage-First收集器
了解云计算的基础

Steve Jobs
1955-2011

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

9月 ARCHITECT

特别专题
QCon全球企业大会精华点滴
QCon在中国的三年回顾
麦肯锡对阿里巴巴国际站架构演进
畅销书《IPS》和《技术流年》
新浪微博团队建设的虚与实
沐泽宁谈主观决策架构
跟着李树学Oozie
如何查看我的订单—
REST的远程API设计案例
通用系统思考，走上改善之路
Redis内存使用优化与存储

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

8月 ARCHITECT

特别专题
云计算的安全风险
圆桌会议：云计算的安全风险
设计一种云级别的身份认证结构
云应用和平台的现状：
云采纳者如是说...

Java虚拟机家族考
专家视角看IT转型构
为什么使用 Redis及高产品定位
架构演化之谜

InfoQ 每月8号出版

时刻关注企业软件开发领域的变化与创新

架构师

7月 ARCHITECT

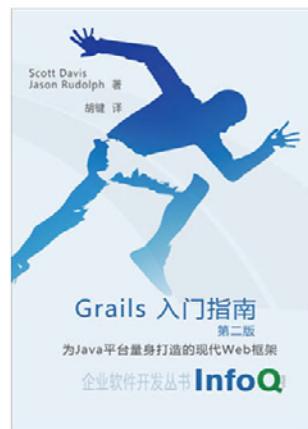
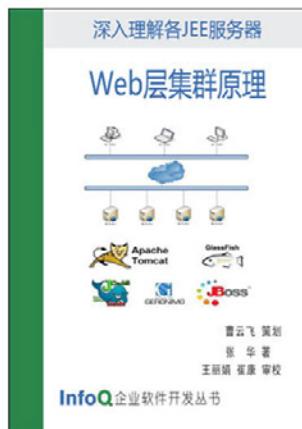
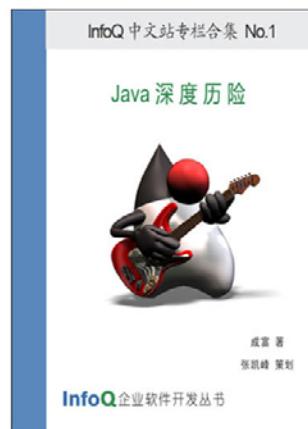
特别专题
深入理解Node.js
为什么要用后端工程语言Node.js
虚拟研讨会：Node.js生态系统之
概览、棒、最佳实践
使用JavaScript和Node.js构建Web应用
Node.js的起源和实践应用—
专访Node.js创始人Ryan Dahl

Java深度剖析十：Java对集群划分与热切换
将数据打散之一：关于松散的数据设计
微服务平台部署与PaaS
社区驱动的源码控制
来自Padmanab的真言良言

InfoQ 每月8号出版

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 5 月刊

每月 8 日出版

本期主编：侯伯薇

美术/流程编辑：水羽哲

总编辑：侯伯薇

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：侯伯薇，中文站翻译团队主编

侯伯薇，生于丹东凤城，学在春城长春，工作在滨城大连；虽已年过而立，但自问童心未泯；对代码热情不减，愿与天下程序员共同修炼，不断提升。译有《[学习 WCF](#)》、《[Expert C# 2008 Business Objects](#)》。

1kg.org 多背一公斤

爱自然 | 更爱孩子

