来自软件界实践者的经验心得

敏捷实践的秘密

ThoughtWorks文集II

李剑 策划

熊节 乔梁 陈金洲 覃其惠 等著

InfoQ企业软件开发丛书

前言 Preface

传播知识、推进社区建设，是 ThoughtWorks 企业文化的核心之一。因为我们的目标并不仅仅是通过帮助客户开发最好的软件系统，为他们带来更多的业务价值，而且还要提升软件开发行业的水准。

我们有许多员工或是在各类会议上担任讲师，或是积极参与开源项目建设，或是在博客和杂志上发表文章，分享传播经验。于是便有了 CruiseControl、有了 Selenium，有了《卓有成效的程序员》、《软件开发沉思录》……当然，还有手头这册迷你书。

自 09 年 3 月份起，“ThoughtWorks 实践集锦 “这个专题已经在 InfoQ 上连载了十余篇文章，其中包括了团队协作、数据迁移、版本控制、测试、富客户端开发、持续集成等内容。这些文章均以具体的技术/管理实践为核心，希冀为读者提供行之有效的指导，在开发过程中充分发挥价值。

覃其慧的文章介绍了，如何让测试人员的工作更有效率，与其他人协作配合。章昱恒以 DBA 的视角，讲述了在数据迁移的工作中，利用敏捷开发常用的测试驱动、持续集成，并引入精益软件的开发技巧，高质量地保障数据迁移的顺利进行，为软件开发打造坚实的数据质量基础。陈金洲以他多个富客户端项目的开发经验为基础，总结了富客户端开发的多项原则与实践。

胡凯和乔梁同是 Cruise 开发团队的一员，他们的文章均来源于在 Cruise 产品中的多年开发经验：乔梁讲述了持续集成实践本身的重构与演进，胡凯的第一篇文章分析了 SVN 的局限性和分布式版本管理工具的种种特性，第二篇提出了一些原则，来帮助团队在不使用 Mock 的前提下，交付具有良好健壮性、可以快速运行的测试。

李光磊的 “TDD 实践之实用主义”，总结了他在 TDD 之路上披荆斩棘前行时，提炼出来的一个个实践"变种"，以解决项目中遇到的一个个难题。纪律和规范是持续集成的基础，如果我们的项目组中存在着不一致的、不可复制的环境，则生产效率始终存在阻碍。李光磊在他的文章 “环境无关的环境 “中，讲述了如何建立规范，解决环境问题。

在 “Tech Lead 的三重人格 “一文中，熊节描述了一个优秀的 Tech Lead 应该承担起哪些职责——技术决策、流程监督、干扰过滤——才能充分发挥其能力，带领团队取得成功。李贝在 “自动化测试的分层结构 “中，将自动化测试分成测试用例层、逻辑层、待测系统层，分离测试逻辑与测试支撑代码，保证了测试代码的易理解、易维护、健壮。

这本书的主旨跟 09 年敏捷中国大会一样，紧扣 “实效 “二字。在敏捷已经得到越来越多的推广实施之后，我们更希望人们能够清楚的认识到目的和手段的区别，关注于如何使用实践、创造实践来不断解决问题，不断改进。

希望这本书能够对你有所帮助。

郭晓

ThoughtWorks 中国公司总经理

目　录

Powered by ThoughtWorks

2010 第五届 敏捷中国大会

AGILE CHINA 2010

10月14-15日 中国 • 北京新世纪日航饭店

由 ThoughtWorks 主办，CSDN 承办，InfoQ 提供票务支持的第五届敏捷软件开发大会“敏捷中国 2010”将于 10 月 14-15 日在新世纪日航酒店举行。

大会将分四个主题：业务、技术和实践、领导力与组织及工具。敏捷中国 2010 旨在为敏捷爱好者搭建开放的交流平台，向社会公开征集演讲话题及讲师。

- Martin Fowler，著名作家，软件咨询师及演讲家，ThoughtWorks 首席科学家
- Mary Poppendieck，电信领域精益软件开发方法权威，著有《 Lean Software Development: An Agile Toolkit 》（曾获 Jolt 大奖）
- James Grenning, Renaissance Software Consulting 公司创始人，全球知名讲师，过程教练及咨询师
- Jean Tabaka，拥有 30 年软件开发行业经验，著有《Collaboration Explained: Facilitation Skills for Software Project Leaders 》
- 何勉，上海贝尔高级项目经理，长期从事电信产品软件开发、产品项目管理和软件部门能力建设等工作
- 张为民，曾任爱立信（中国）项目经理，现任中国移动通信研究院业务拓展经理
- ……

点击报名，享有优惠

10 月 14 日至 15 中国·北京新世纪日航饭店

InfoQ Enterprise Software Development Community

InfoQ 中文站作为第四届敏捷中国大会合作方，共同组织策划帮助 ThoughtWorks 实施了以“实效敏捷”为主题的敏捷中国 2009 大会。

第一章 我和敏捷团队的五个约定 1

覃其慧 ThoughtWorks 公司敏捷咨询师

我——作为一名测试人员——有一个与众不同的习惯：每当要加入一个新项目的时候，我总会找到项目中的同伴，真诚而亲切地说："为了更好地合作，我有 5 个约定，希望大家能尽量遵守"。

约定 1. 业务分析师们，我们其实是同一个角色的两种面孔，请叫上我们参加客户需求会议

我们的团队需要让客户频繁的得到可用的软件，客户的不断反馈会给软件的未来做出最正确的方向指引。

如果我们交付的软件有很多质量的问题，存在大量的缺陷，客户会被这些缺陷的奇怪行为干扰，没有办法把注意力放在软件本身的价值是否符合他们的真正需求上，不能给出最有价值的反馈。所以，我们只有频繁的做测试，在每次交付之前都把质量问题找出来告诉我们的团队，问题才能及时的得到改正。

而我坚信"prevention is better than cure"（预防胜于治疗），我会要把工作的重点放在预防缺陷上，这样可以节省 Dev 们很多修复缺陷的时间与精力。

为了达到这个目的，我需要跟你一起参加客户需求会议，尽早的了解客户需求与使用软件的惯常行为。那么在你完成需求的验收条件的定义的时候，我也基本完成了测试用例的准备。

我们可以赶在开发人员们写代码之前就告诉他们我要测什么，让他们减少因为过于乐观而漏掉的一些重要的有破坏性的情况，减少缺陷的发生。这是我测试的一项重要任务。

如果你们在大部分需求都整理好了再交给我们，我会浪费掉等待的时间。更重要的是，开发好的软件里面已经有很多本来可以不存在的缺陷在里面了，开发人员们可能需要加班加点来保证在项目最终交付时间之前把它们改好。这样很容易产生新的缺陷的。

所以，请让我尽早了解需求，请不要让我到项目后期才能开始测试。

约定 2. 开发人员们，虽然你们是编写自动化测试的专家，但请听听我们意见

我知道，对于你们，自动化测试不过是利用 junit, rspec, selenium，watir，uiautomation 等等

写出的“另一段程序”而已。而对于 80%的 QA 来说，编写自动化测试并不是一件简单的事情。

不过我仍然相信，有测试人员介入的自动化测试更有价值。

你们用单元测试，集成测试来保证代码的质量。然而你们的这些日常测试离代码更近，离最终用户还点远。很多测试都不是在测软件功能。

你们可以把功能测试写的又快又多 ,而我们可以指出什么功能测试最有必要加到自动化测试中。

你们平时大部分精力都在编码上，没有太多时间去查都有什么缺陷。而我们可以指出什么地方缺陷可能会出现的比较频繁，建议在这些脆弱的地方加自动化测试。

所以请听听我们的意见，我们可以给你们提供这些信息。

约定 3. 项目经理们，请不要要求我们测试软件的所有路径

软件测试是一个永无止尽的任务。基本上没有什么软件简单到我们能够尝试完它的每一个可能的路径的。就连一个看似简单的微软计算器都有无穷尽的路径，无止尽的输入，更何况比这个更复杂的商用软件。

如果你们担心没有尝试过全部的路径不可靠，疑惑我们怎么敢说这个软件质量是好的还是坏，都有什么风险。请你们先注意，我们是跟业务分析师一样，都了解软件的价值的。价值可以帮我们做出判断，什么时候可以停止测试并对客户说我们的软件已经满足您的要求了，请放心使用。

因为我们了解价值，我们可以肯定的说哪些软件的使用方式是至关重要的，哪些是不太可能出现的。我们会在全面测试了软件以后，把主要精力放在价值高的功能点上。合理的利用项目有限的时间。

因为我们了解价值，我们可以正确的把发现的问题分类。我们可以帮助 dev 们把精力放在重要的缺陷上，避免把时间放在对于客户微不足道却不得不花费大量精力才能修正的问题上。

所以，请不要要求我们无止尽的测试一个软件。我们了解价值，请相信我们的判断。

约定 4. 迭代经理们，如果对于交付风险有任何疑问，请来询问我

BA 和 Dev 们都是关注一个软件在什么情况是可以良好的工作。而我们除了验证这些情况以外，大量的时候都用在寻找什么样的情况软件不能正常的运行。所以除了针对定义好的软件行为进行测试，我们还会做很多探索性测试。我们通常可以通过这样的测试发现一些没有定

义的、不曾预期的行为。这些行为往往将会构成软件交付的风险。

我们会告诉你们现在都发生了什么问题，分别分布在哪里。

我们会告诉你们，在什么情况下软件可能会有异常行为，是不是会牵连到其他的部分，是否可以绕过去。

我们会告诉你们，哪些部分功能比较不稳定，需要更多的留意。

约定 5. 测试人员们，那些敏捷实践对于我们也是有用的。

结对不是 dev 们的专利。我不希望总见到你们独自坐在自己的位置上冥思苦想。走出去，跟其他队友多多交流！

多跟测试队友交流，pair 看看设计的测试用例是不是够全面，独自一个人想到的未必足够好。他们会给你诚恳的意见的。对他们，也请一样认真对待。

如果你发现开发人员们做出的架构决定使测试工作变得更困难。那么请大声地告诉他们，design for testability（提高你们设计的可测性）。

如果你发现业务分析师写的需求无法验证，定义的客户行为不够具体，一个用户故事中包含太多了功能点，等等，那么也请大声地告诉他，INVEST（独立，可协商，价值，可估算，短小，可测）。

也请你们多跟开发人员结对写自动化测试，既可以帮助你们学习怎样更好的编写自动化测试，也能帮助开发人员们结对更多的了解用户行为。

这就是我的五个约定，它们是我在团队中顺利展开工作的基础。

作者简介：覃其慧，ThoughtWorks 敏捷咨询师。她参与了大量的敏捷软件开发的实践和敏捷咨询。目前主要关注以价值为驱动的敏捷测试。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-parti

第二章 如何在敏捷开发中做好数据迁移 2

章显恒 ThoughtWorks 公司敏捷咨询师

数据迁移是指在系统软件开发中，将具有实际业务价值的数据，依据功能需求或系统开发的要求，在不同存储媒介、存储形式或计算机系统之间转移的过程。

数据迁移是系统开发经常涉及到的一项工作。在企业级应用系统中，新系统的开发，新旧系统的升级换代，以及正常的系统维护，不可避免地涉及到大量的迁移工作。而在一个以数据为核心的业务系统中，数据的迁移更是无处不在。比如：在以数据仓库为架构原型的系统设计中，ETL（抽取，转换，装载）部分的实现就是一种数据迁移；对大型数据系统的分布式实施，数据迁移就是整个实施过程的主要部分。而在敏捷实践中，渐进式的数据库开发，更是涉及到大量的数据迁移和同步工作。

我们时常会听到用户提出这样的要求“我们并不过于关心应用的好坏，但需务必保证数据准确”。的确，在以数据为运营基础的行业里，数据质量本身就是软件质量的权重部分，尤其在电信、金融和控制领域里，这一特征表现的格外明显。数据迁移也是敏捷开发中相当重要的环节，它影响着各个发布版本的数据质量，而数据质量又决定着系统的有效性和可靠性，因此高质量地完成数据迁移不容忽视。

数据迁移往往被视为一件很简单的工作。在很多人眼里，数据迁移仅仅是用 sql 语句向相应数据表装载数据的过程。但在实际操作中，数据迁移涉及到很多层面的因素，如用户需求，系统功能，数据库建模等，若出现问题，将导致开发进展缓慢或质量不高。常见问题有业务系统逻辑模糊、脏数据、遗留系统的技术债和管理债等。那么如何有效的避免这些问题，提高迁移质量呢？

本文将以 ThoughtWorks 中国公司与客户合作的 CRM 项目为背景，为读者介绍如何在敏捷开发中高质量地处理数据迁移工作，从而在数据层面提高系统质量。

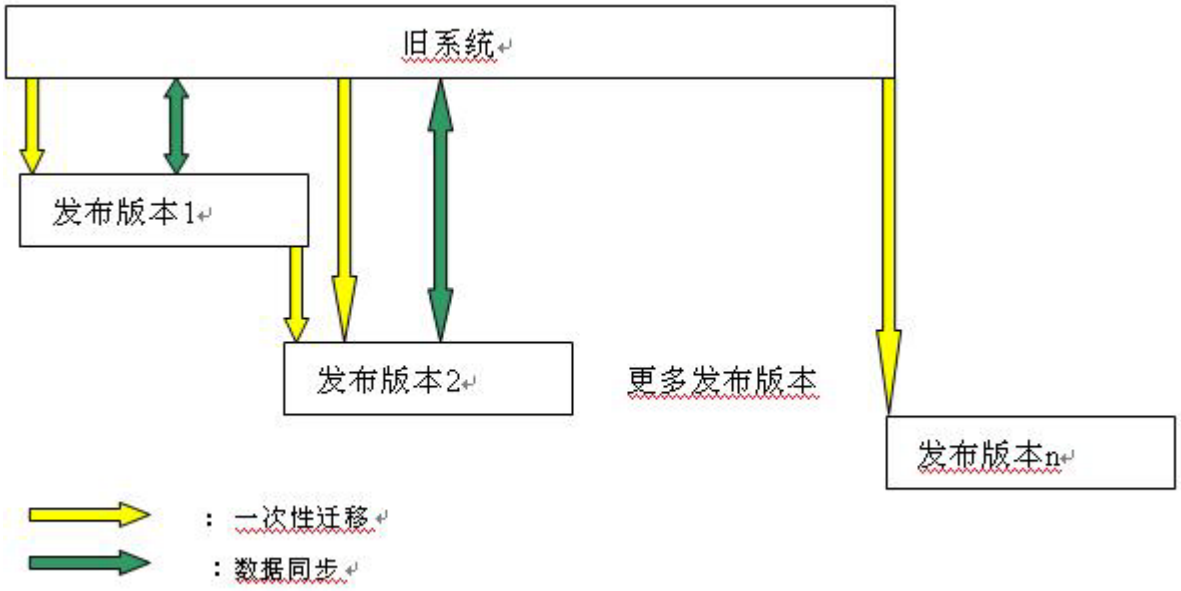
开发背景

A 系统（旧系统）是客户原有的一套 CRM（客户关系管理）系统。系统采用 B/S 架构，使用 sql server 2005 做为后台数据库。旧系统的数据建模设计采用了高度范式化的设计思路，

其目的是极度追求灵活性。业务数据被大量拆分并散布存储在上百张数据表里。数据表内和表之间不存在参照约束。大量的业务逻辑采用存储过程封装以提高效率。存储过程体系相当庞大，且存在复杂的相互调用。数据库中存在一些脏数据，可能是长期的使用、维护或误操作导致，但没人知道它们有多少，具体存在哪里。应用界面可用性不理想而且系统效率较低，用户常抱怨系统反映迟缓或无反应。数据库 存储的业务数据约 50G 左右。

ThoughtWorks 团队将为客户提供一套新的 CRM 系统用以替换旧系统主要功能。新系统精简整理旧系统功能，并整合了客户的最新需求。在设计上做了巨大变更，以改善界面可用性，同时为了保障终端用户对系统服务的需要，新旧系统要求能够同时运行并实现数据同步，当终端用户全部过度到新系统后，终止旧系统。在这个过程中，DBA 团队需给予足够的数据保障。

以下为项目版本的发布图。



数据迁移开发方法

1. DBA 需要制定目标并且管理自己的任务

尽管在每个迭代中，团队都会讨论决定如何组织‘需求故事’（story），但是 DBA 仍然需要有自己的‘故事墙’（story wall），并且花时间组织自己的 story。在实际开发中，数据迁移仅仅是 DBA 工作的一部分，DBA 还要完成相应的 story 开发和数据分析，有时还要给开发人员提供数据支持。混乱的管理会带来开发上的冲突。因此，有效管理任务是做

好数据迁移的首要环节。

故事墙是管理这些任务的最好方法。尽管这个故事墙对客户提供的商业价值是间接的，但从整个团队角度来看，任何需要数据的人或程序都是 DBA 的用户，故事墙有利于管理每个 story 包含的数据需求，避免数据迁移任务与其它数据库开发任务之间的冲突，从而减少重复性工作或修复性工作。DBA 有必要将这种 方法引入到数据库开发中。

DBA 要从商业价值角度决策数据迁移的需求。系统开发中，客户和开发员常常会向 DBA 提出自己的数据迁移要求，但往往这些要求并不具有全局性和决定性，毕竟他们仅仅是针对一个 story 的需要而提出。如果 DBA 盲目执行，将起到事倍功半的效果。DBA 应当积极参加 IPM（迭代计划会议。它是在每个迭代开始时的会议，全体成员共同讨论 story 计划完成数量）。无论是直接与用户交互，还是参与团队合作，DBA 有必要将每个 story 内容了解的清清楚楚。通常，DBA 可以不必像开发人员一样去了解 story 的开发细节，但通过与业务分析师和开发员的沟通，潜在的数据需求自然浮出水面。针对这些数据需求，通过再次组织并加以优先级，我们很容易回答这些问题：接下来应该完成的任务是什么？它的实际商业价值是什么？谁将需要它？什么时候需要？实践证明，多花些时间和团队或客户沟通是事半功倍的好方法，而且 DBA 通过了解业务数据可以给开发员更好的指导，减少开发员对数据的误解，有利于提高整体团队的开发效率。

通过对每个 story 的了解，我们总结并制定了针对当前发布版本需要的 7 个数据迁移 story，并且确认了它们的确不存在任务上的重复，也邀请项目经理和客户一起确认了这份计划。如此我们的目标已经制定。

2. **思考实施策略**

我们已经管理好所有数据迁移的任务，接下来考虑如何实现。通过以往的经验，我们发现如果没有仔细思考全局和细节问题而直接编写代码，带来的后果是无法控制的。我们应该首先充分了解这个过程可能存在的风险，然后决定采用什么样的策略，是否可以借助工具提高效率。这里的潜在风险主要包括：

2.1 数据质量

> 旧系统的数据库建模是一个高度范式化的结构，每个表之间存在相当大的依赖关系。一旦一个表存在脏数据，我们如何保证得到正确的查询结果？

2.2 对原有系统的了解

> 旧系统的应用程序引入了面向对象的设计方法，并且继承关系数据也被存储在若干张数据表里，如何正确区分这些业务对象和关系，保证在迁移过程中不会制造脏数据？

2.3 业务数据映射

> 旧系统和新系统之间存在着相当大的业务逻辑差异，我们是否能够将业务逻辑、数据映射到新系统？是否存在不可实现的转换？

在未充分了解这些问题之前，我们无法进一步制定计划，即时给予客户反馈是解决这些问题的最好方法。经过进一步沟通后，我们发现问题的复杂程度远远超过想象，尽管客户对旧系统非常了解，但他们对于某些数据也不能给出明确答案。鉴于这些情况，我们制定了初步的解决策略：

1. 更多的了解旧系统，即时给予反馈。对于那些无法找到答案的问题，考虑是否可以寻求其它资源或忽略没有价值的数据。
2. 尽量细化分割每一个复杂需求，形成多个任务。小粒度任务能够帮助暴露更多问题。
3. 采用测试驱动，确保一套可靠的测试机制。
4. 制定实现框架和阶段性目标。
5. 不要过于乐观的估计进展，每一阶段要留有充分的单元测试。
6. 调整每个迭代的内容，对有较强依赖关系的任务可以放在今后的迭代周期里。

3. 实施数据迁移

新系统的数据迁移包含两个部分：一次性数据迁移和数据同步迁移

一次性数据迁移

一次性数据迁移指仅仅发生在某一个发布版本上线安装时 ,新旧系统同时处于脱机状态，一部分数据将从旧系统中转移到新系统的过程。

数据同步迁移

数据同步迁移过程发生在新系统运行时，新旧系统同时处于工作状态，双方通过交换数

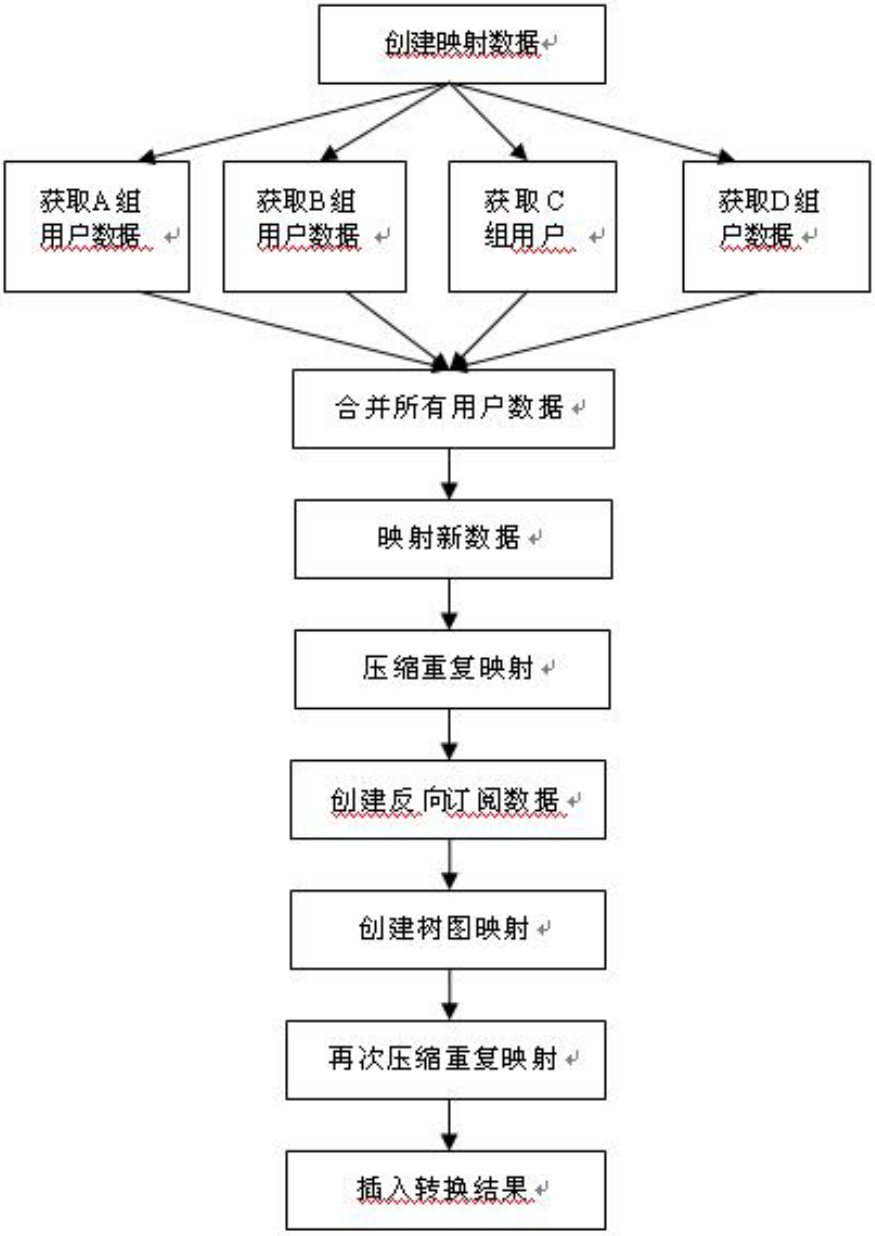
据保证彼此数据的一致性。

同为数据迁移，但因两类迁移各具特点，因此在共同的处理方式上也略有不同。

	一次性数据迁移	数据同步迁移
特点	1. 数据量大。 2. 使用频率低（一次性使用）。 3. 转换逻辑复杂，需大量定制映射转换数据。	1. 数据量小 2. 使用频率高（以分钟为单位，周期性运行）。 3. 转换逻辑复杂，少量定制映射转换数据。 4. 需要事务处理以保证数据一致性
共同处理方式	1. 细化任务。 2. 测试驱动。 3. 持续集成	
不同处理方式	1. 在执行测试驱动中，应侧重数据质量的测试。应依据不同环境的测试结果 ,增强测试体系。 2. 工具选择。避免使用第三方工具，直接使用 sql 脚本以提高迁移效率。 3. 保留中间处理结果	1. 在执行测试驱动中，应侧重逻辑映射方面的测试。 2. 工具选择。可考虑使用第三方工具，增强事务控制。 3. 可不保留中间结果

- 细化任务

  依据最初制定的开发策略，当我们遇到复杂的迁移需求时，首先分解每个需求为若干个模块，然后画出整体结构图。以下是某一处数据迁移脚本的模块分割：



最初由于这个部分的迁移逻辑过于复杂，以至于客户对它的处理结果没有信心。但当共同完成这个图表后，大家一致认为它没有像想象中的困难。总而言之，立刻解决一个复杂的问题很困难，但解决其中一个小问题却很容易。

- 测试驱动

如同编写程序代码一样，我们不仅为实现数据迁移脚本使用了测试驱动，还引入了针对数据库设计的一些方法。在程序设计中，当代码本身结构良好，单元（类、方法）之间关系清晰，可以直接添加单元测试。现在，我们有了很好的脚本逻辑结构，可以很容易添加每一步结果的单元测试，这就如同形成了一道安全网，保证异常数据出现

时，能够立即发现并加以处理。在实际编写迁移脚本之前，应首先明确测试内容，准备好测试脚本。

测试内容包括：

应产生的符合期望的数据

基于给予的原始测试数据，这一测试过程测试脚本的数据转换逻辑是否正确。以下举例说明：

测试环境：旧系统中存在某个名为'Jason'的客户信息，他的 personId 是 1000101 。

测试目的：当某一客户的信息迁移到新系统的 CUSTOMERS 表后，新系统应该存在该客户信息。

新系统上要运行的测试代码：

```
DECLARE @personName NVARCHAR(250),


SELECT
@personName = personName
FROM
CUSTOMERS
WHERE
personId = 1000101
IF (@personName <> 'Jason') or (@personName is NULL)
BEGIN
INSERT INTO LoadTestErrorLog (errorDescription)
VALUES ('personName for personId 1000101 is not Jason')
END
Go
```

这里常用的原则是：一段 sql 语句仅用来测试一处期望数据，这样可以减少代码之间的相互依赖性，更准确的定位错误数据。

不应当产生的异常数据

异常数据指在迁移过程中出现的不符合逻辑的数据。理论上讲，迁移过程不应当出现

异常数据，然而现实情况中，迁移结果总会出现我们不需要的数据。其原因包括 数据源出现异常、实现过程中的误操作、系统应用的 bug 等。总而言之，为了保证这些错误不会出现在最终结果，相应的测试脚本必不可少，也是防止问题进一步扩大的有效举措。这一测试过程常被用来发现在生产环境中可能出现的问题。以下举例如何测试异常数据：

测试环境：全部或部分生产环境数据

测试目的：将某个客户的信息迁移到新系统的 CUSTOMERS 表后，数据表不应该具有顾客名字为空的记录，如果出现将视为迁移过程的错误。

新系统上要运行的测试代码：

```
DECLARE @isExistPersonNameWithNULL INTEGER


SELECT
@isExistPersonNameWithNULL = count(*)
FROM
CUSTOMERS
where personName is null


IF (@isExistPersonNameWithNULL> 0)
BEGIN
INSERT INTO LoadTestErrorLog (errorDescription)
VALUES ('personName doesn't contain legal information')
END


Go
```

数据表的数据量是否符合期望

当数据被迁移至新系统后，应当确保迁移数据量符合应期望值。实现方法多种多样，较简单的方法是直接比较数据迁移前后的数据记录数是否在数值上相等。以下举例说明：

测试环境：全部或部分生产环境数据。

测试目的：客户数据被迁移后，应当确保客户数据没有丢失。

新系统上要运行的测试代码：

```
DECLARE @NumberofCustomerinOldDB INTEGER
DECLARE @NumberofCustomerinNewDB INTEGER

SELECT
@NumberofCustomerinOldDB = count(*)
FROM
oldDB.dbo.persons -- 这是在旧系统中定义的客户表
...
```

--省略复杂的过滤逻辑

```
SELECT
@NumberofCustomerinNewDB = count(*)
FROM
newdb.dbo.CUSTOMERS -- 这是在新系统中定义的客户表
where personName is null
IF (@NumberofCustomerinOldDB<>@NumberofCustomerinNewDB )
BEGIN
INSERT INTO LoadTestErrorLog (errorDescription)
VALUES ('not all customers are migrated ')
END
Go
```

最终当把测试 sql 代码片段组装在一起后，我们获得了一批测试脚本，并按照以下流程，通过使用 NANT 工具实现自动化：

NANT 中的实现方法：

<target name="-init " … />

该任务负责初始化测试环境

<target name="-parseDbScripts " … />

该任务负责编译并部署迁移脚本

<target name="-resetTestData " … />

该任务负责重置测试数据

<target name="-executeMigrationScripts " … />

该任务负责执行迁移脚本

<target name="-testMigration " … />

该任务负责执行迁移测试脚本

<target name="testDataMigration" depends="-init, -parseDbScripts, -resetTestData, executeMigrationScripts, -testMigration" />

该任务将成为持续集成调用的入口

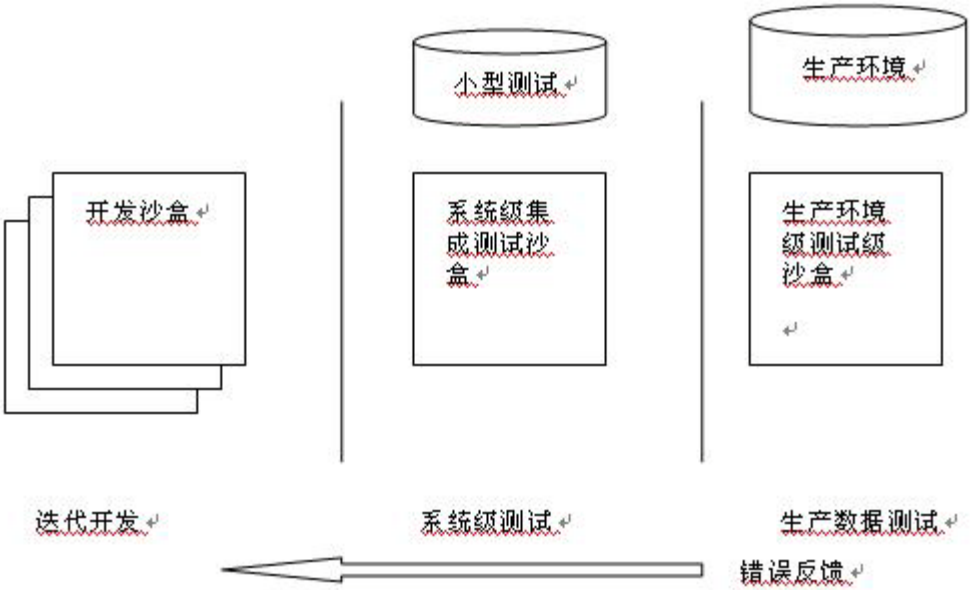
- 持续集成

为完成持续集成测试，测试沙盒必不可少。“沙盒“是一个完整的功能环境，在这里脚本能够被编译，测试和运行。

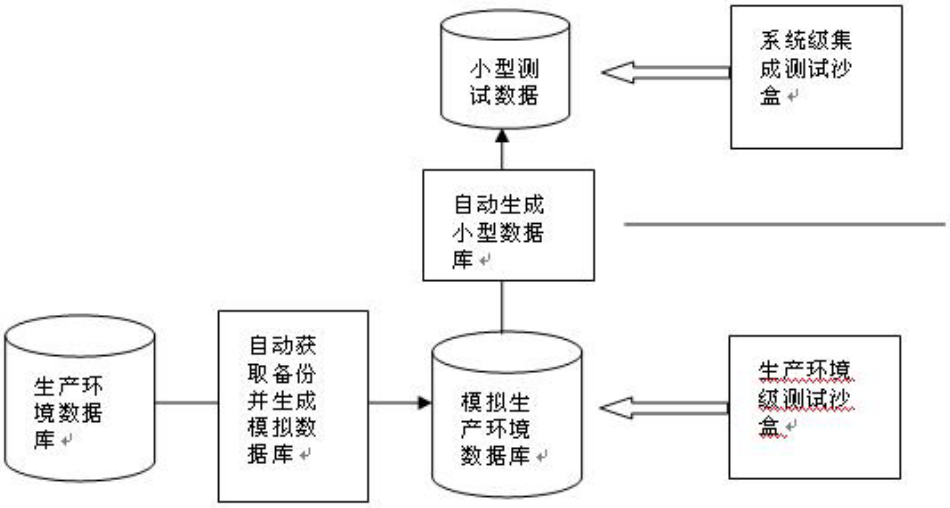
a) 在开发沙盒中，我们准备了少量的核心数据，用以测试 sql 脚本的质量。

b) 在系统级集成测试沙盒中，我们还准备了一个小型数据库，这个数据库包含了一部分核心数据，着重测试数据迁移过程的逻辑转换。

c) 在生产环境级测试沙盒中，由于数据库来源于实际数据备份，因此数据处于不断变化状态 ,这就更需要不断运行测试脚本 ,避免脏数据和数据丢失。由于生产环境数 据量相对大了许多，我们可以适当减少测试次数以减少对开发资源的消耗。同时，其它测试脚本，如变更数据库结构的脚本，都可以和数据迁移脚本组织在一起，一 次性完成测试。



同样，我们采用自动化机制维护这些开发测试沙盒。



将测试置于持续集成环境中，下图是处于持续集成环境的测试任务。

- 工具选择

  选择数据迁移工具应当以帮助提高工作效率和数据迁移运行效率为原则。通常最直接的方法是编写 sql 脚本，借助其它工具也能起到很好的效果，比如 MS SSIS 等。然而我们发现，过多的引入第三方工具往往带来的麻烦也多，例如，我们不得不花时间来学习这些工具的某些特殊用法，有时工具也会产生 bug，以 至于不得不再花时间解决这些 bug，而这与最初的开发目标相背离。因此，有效的方法是尽量使用 sql 脚本执行所有的迁移工作，同时也得到了最佳的执行效 率。

- 保留中间结果用于脚本调试

  相比设计语言，Sql 语句较难调试，即使有些数据库产品提供了调试工具，但是调试数据结果集仍然是项挑战性的工作。尤其在旧系统到新系统的迁移过程中，业务逻辑发生巨大变化，客户经常要求提供某些证据，来解释他们对数据迁移结果的怀疑。保留中 间环节数据,不仅方便调试，也方便数据追溯，为开发带来更高效率。以下举例说明：

```
SELECT

...

into debug_allpersonhistroy

FROM

oldDB.dbo.personhistory -- 这是在旧系统中定义的业务存储表

...

--省略复杂的过滤逻辑

select column1...columnN

into debug_allpersonhistroy_aftermapping --保留这一步数据集合
```

```
from debug_allpersonhistroy inner join mappingtableBtwOldandNew

...

--省略复杂的过滤逻辑

SELECT

...

FROM

newdb.dbo.contactHistory -- 这是在新系统中定义的业务存储表

...

--省略复杂的过滤逻辑

Go
```

典型问题

数据迁移在不同的场景往往出现不同的问题，单凭经验也不能全部解决。运用头脑风暴，集中团队中所有力量思考所有可能出现的问题并加以避免。有时开发员遇到的问题也帮助 DBA 少走弯路。最终，头脑风暴能够提供我们的是一份有价值的列表，里面包含各种问题和注意事项：

1. 一致性检查

   一致性检查包括：字符编码检查、语言设置、环境参数设置等。

   迁移过程常出问题的是字符集，它带来的问题是数据乱码。不同系统在最初设计时应用的字符集或编码格式未必相同。在迁移过程中，单凭缺省设置是不够不安全的。有效的办法是在项目伊始，即确认系统间环境一致性。在新系统中采用兼容性的 unicode 编码也能够解决这些问题。

2. 控制 NULL 的使用

   由于旧系统本身很少使用约束，以至于在表连接查询中出现大量无法得到正确匹配的数据。在 sql 中，当我们试图使用自然连接，我们发现某些数据丢失了，如果使用外连接，这将会带来一种新的脏数据：NULL。从数据库设计角度，NULL 不代表任何含义，而实际情况中，很多数据库建模往往给 NULL 赋予含义，甚至多种含义，以至于不同的查询

需求要视不同的业务逻辑对待。在旧系统里，这种现象比比皆是，无疑给迁移带来了不少麻烦。

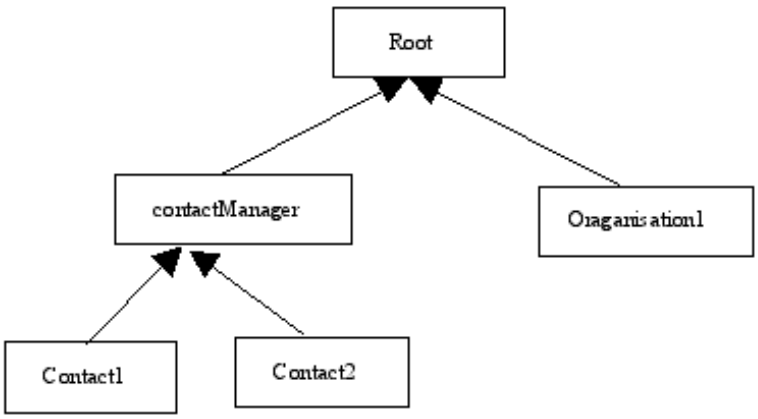
解决方法：不为 NULL 赋予逻辑上的定义。尽量少使用外连接运算。

例如：

旧系统定义如下父子结构表：

```
objectId, parentObjectId,objectType …

-------------------------------------

Null Null ‘root’

1 Null ‘contactManager’

2 1 ‘contact’

3 1 ‘contact’

4 Null ‘orgnisation1’

…
```

显然，系统希望构建如下对象树图：



然而，当程序试图遍历所有对象时发现：NULL 无法参与计算。因为 NULL 与任何数据的计算结果都是 NULL。程序必须增加额外代码来处理特殊情况。

3. 代码复用，降低依赖性

迁移脚本应当遵循与编码同样的规则，高内聚，低耦合，能够被重复利用的代码需尽量被封装成单元，重复拷贝并不是迁移脚本应当采用的方法。

解决方法：使用临时存储过程实现某些公用代码的复用，简化调用接口。

4. 新问题，新测试

当我们遇到新的问题时，常忙于解决问题，给出解释。然而当这一切完成后，并不意味着问题已经全部被解决。因为这些问题仍然可能再次发生，也说明目前测试不足。

解决方法：当新问题出现后，暂停当前的工作，立刻针对这种情况写出测试。为其花费些时间意味着不会让技术问题债台高垒。

例如：在新系统的数据库里，QA 发现了一组不符合逻辑的数据：记录的结束时间(EndTimestamp)早于开始时间(startTimeStamp)8 个小时。它的实际期望结果是：记录的结束时间必须晚于开始时间。

```
ID，  startTimestamp, EndTimestamp, createDate …

-----------------------------------------------------

11020011 2008-12-14 09:23:00 2008-12-14 01:23:00 2008-12-14 09:23:00
```

显然程序在插入数据时用错了时区。在 bug 被修复之前，立刻加入一个数据库测试以保障今后不会再次出现。

测试代码如下：

```
DECLARE @CNT INTEGER


select @CNT=COUNT(*) from tableA where startTimestamp> EndTimestamp

IF @CNT>0

BEGIN

INSERT INTO LoadTestErrorLog (errorDescription)

VALUES (' EndTimestamp should be late than startTimestamp ')

END

GO
```

5. 目标制定者和开发者应该保留的心态

数据迁移是一件看似简单但具有挑战的工作。因此，我们常常过于乐观估计开发效率。然而这里的风险在于我们仅仅看到了处理逻辑，而没有看清楚数据质量，以至于盲目写出的迁移脚本可以在测试环境中工作，但无法在生产环境中运行。

解决方法：无论多么简单的数据迁移，应首先与客户或业务分析师沟通业务逻辑，确保对数据质量的了解。

结论

数据迁移是一项看似简单却蕴含巨大挑战的工作。它不仅包含了具体技术问题，而且要求 DBA 具有较好的沟通能力，深入的了解业务逻辑。通过旧系统到新系统的 数据迁移工作，我们逐渐地将精益软件设计思想深入到细节，并且取得了很好的效果。当数据迁移完成后，我们完成了近 6000 行的迁移脚本，迁移结果通过了客户方的抽样测试，最终确保了整个系统的正常运行。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-partii

第三章 RichClient/RIA 原则与实践(上) 3

陈金洲 ThoughtWorks 公司咨询师

Web 领域的经验在过去十多年的不断的使用和锤炼中，整个开发领域的技术、理念、缺陷已经趋于成熟。JavaEE Stack, .NET Stack, Ruby On Rails 等框架代表了目前这个技术领域的所有经验积累。这样我们在开始一个新的项目的时候，只需要选择对应语言的最佳实践，基本上不会犯大的错误。例如，如果使用 Java 开发一个新的 Web 应用，那么基本上 Spring/Guice+Hibernate/iBatis/+Struts /SpringMVC 这种架构是不会产生重大的架构问题的；如果使用 RoR 那么你已经在使用最佳实践了；系统的分层：领域层，数据库层，服务层，表现层等等；为了保证系统的可扩展性，服务器端应当是无状态架构，等等。总而言之，web 开发领域，它丰富的积累使得开发者逐渐将更多的精力投入到应用本身。

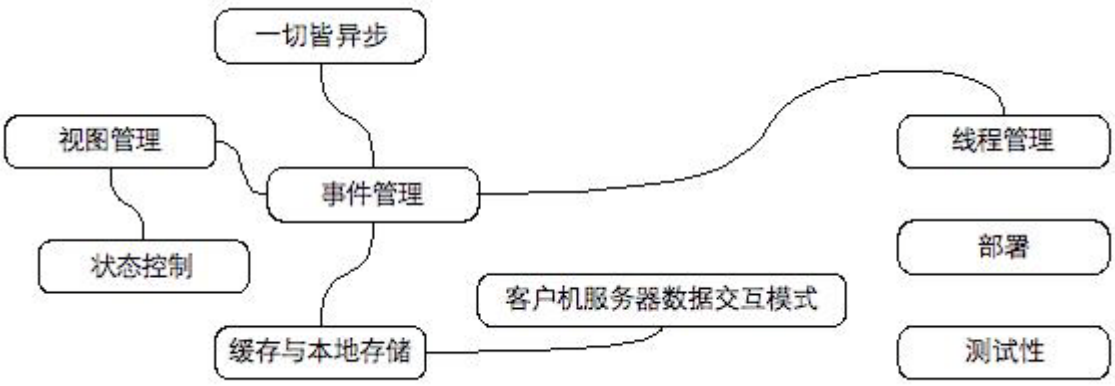
来看富客户端，或者富互联网应用。在我看来，今天的 RichClient 与 RIA 已经没有分别：只要代表着丰富界面元素和丰富用户体验，需要与服务器进行交互的应用都可以称为 RichClient 或者 RIA，虽然感觉上 RichClient 更“企业化”一些（服务器往往在企业内部），RIA 更“个人化”一些（服务器往往处于公网）。从最小的层面来说，我现在正在使用的离线模式的 GoogleDoc 就是一个 RichClient 应用——虽然它没有那么 Rich，采用和 microsoft office 一样土的界面；我现在正在听音乐的 Last.fm 客户端显然是一个非常典型的 RIA——它所有的个人喜好信息、音乐全都来自远在美国的服务器。本地的这个界面，只是提供收集个人和音乐信息，以及控制音乐的播放和停止；目前拥有 1150 万玩家的魔兽世界，则是一个挣钱最多的，最“富”的客户端，10 多 G 的客户端包含了电影品质的广阔场景，华丽的魔法效果和极其复杂的人机交互。

如今的用户需求已经达到了一个新的高度，那些灰色的，方方正正的界面已经逐渐不能够满足客户的需求。从我们工作的客户看来，他们除了对“完成功能”有着基本的期待外，对于将应用做得“酷”，也抱有极大的热情。我工作的上一个项目是一个 CRM 系统，它是基于.NET Framework 3.5 的一个 RichClient 应用。它的主窗口是一个带着红色渐变背景的无边框窗口，还有请专业美工制作的图标，点击某一个菜单还有华丽的二级菜单滑动效果。我们在这个项

目中获得了很多，有些值得借鉴，有些仍然值得反思。我仍然记得我们在项目的不同阶段，做一个技术决定是如此的彷徨和忐忑：因为在当时的 RichClient 企业开发领域，几乎没有任何丰富的经验可以借鉴，我们重新发明了一些轮子，然后又推翻它；我们偏离了 UI 框架给我们提供的各种便利而自己实现种种基础特性 ,只是因为他们偏离了我们所倡导的测试性的原则。在写下本文的时候，我尝试搜索了一下，仍然没有比较深入的实践性文章来介绍企业环 境下 RichClient 开发。大多数的书，如 Swing、JavaFX、.NET WPF 开发等等，偏向于小规模特性介绍，而在大规模的企业应用中，这些小的技巧对于架构决策往往帮助很小。

我的工作经历应当是和大多数开始进行 RichClient 开发的开发者类似 :有着丰富的 Web 开发的经验之后开始进行 RichClient 开发。加入 ThoughtWorks 之后参加了多个不同的 RichClient 项目的开发工作 ,使用/尝试过的语言包括 Java Swing, Flex/Adobe Air, .NET WinForm/.NET WPF. 对于不同平台之间的种种有些体会。在这里我将这些实践和原则总结如下。例子很可能过时，毕竟华丽的界面框架层出不穷，但原则应当通用的。使用和遵循这些原 则将会帮助你少犯错误——至少比我们过去犯的错误要少。如果你拥有一定的 web 开发经验，那么这篇文章你读起来会很亲切。

这些原则/实践往往不是孤立的，我尝试将他们之间用图的方式关联起来，帮助你在使用的过程中进行选择。例如，你遵循了“一切皆异步”的原则，那么很可能你 需要进行“线程管理”和“事件管理”；如果你需要引入“缓存与本地存储”，那么“数据交互模式”你也需要进行考虑。希望这张图能够帮助读者理解不同原则之间的联系。



下面列出的这些原则或者实践没有严格意义上的区分。按照上面的图，我推荐是，一旦你考虑到了某一个实践，那么与它直接关联的实践你最好也要实现。它会使得你的架构更全面，经得起用户功能的需求和交互的需求。

为了让这些实践更加通用，我采用伪代码书写。相信读者能够转化成相应的语言——Java, C#,

ActionScript 或者其他。这些实践并非与某一种语言相关。在某些特定的例子中，我会采用特定语言，但大多数都是伪代码描述的。

1 一切皆异步

所有耗时的操作都应当异步进行。这是第一条、也是最重要的原则，违背了这条原则将会导致你的应用完全不可用。

考虑这样的一个功能：点击一个"更新股票信息"按钮，系统会从股票市场（第三方应用）获得最新的股票信息，并将信息更新到主界面。丝毫不考虑用户体验的写法：

```
void updateStockDataButton_clicked() {
    stockData = stockDataService.getLatest(); // 从远程获取股票信息
    updateUI(stockData); // 这个方法会更新界面
}
```

那么，当用户点击 updateStockDataButton 的时候，会有什么反应？难说。如果是一个无限带宽、无限计算资源的世界，这段代码直观又易懂，而且工作的非常好：它会从第三方股票系统读到股票数据，并且更新到界面上。可惜不是。这段代码在现实世界工作的时候，当用户点击这个按钮，整个界面会冻结一一知道那种感觉吗？就是点完这个按钮，界面不动了；如果你在使用 Windows，然后尝试拽住窗口到处移动，你会发现这个窗口经过的地方都是白的。你的客户不会理解你的程序实际上在很努力的从股票市场获得数据，他们只会很愤怒的说，这 个东西把我的机器弄死了！他们的思路被打断了。于是他们不再使用你的程序，你们的合作没了。你没钱了。你的狗也跑了。

出现界面冻结的原因是，耗时操作阻塞了 UI 线程。UI 线程一般负责着渲染界面，响应用户交互，如果这个线程被阻塞，它将无法响应所有的用户交互请求，甚至包括拖拽窗口这样简单的操作。所有的界面框架，无论是 Java/.NET/ActionScript/JavaScript，都只有一个 UI 线程，这个估计永远都不会变。

用户看到的应用通常与程序员大相径庭。用户对应用的期待级别分别是：能用、可用、好用、好看。而我观察到的大多数程序员停留在第一阶段：能用。“一切皆异步”这个原则说来简单，做起来也不会很难。把上面的代码稍作改动，如下：

```
void updateStockDataButton_clicked() {
    runInAnotherThread( function () {
        stockData = stockDataService.getLatest(); // 从远程获取股票信息
        updateUI(stockData); // 这个方法在UI线程更新界面
    }
}
```

注意加粗部分。runInAnotherThread 是跟语言平台特定的。对于.net C#，可以是一个 Dispatcher+delegate 或者 ThreadPool.QueueUserWorkItem；对于 Java，可以干脆是一个 Runable。对于 AJAX，可以是 XMLHttpRequest 或者把这个计算扔到一个 IFrame 中；对于 ActionScript，似乎没有什么好的方法，把获取数据的部分交给 XML.load 然后通过事件回调的方式来进行界面刷新吧。

耗时操作一般两种来源产生：网络带来的延迟以及大规模运算。两者对应的异步实现方式有所不同。前者往往可以通过特定语言、平台的获取数据的方式来进行异步，特别是缺乏多线程特性的动态语言。例如典型的 AJAX 方式：

```
xhr = new XmlHttpRequest()
xhr.send("POST", '/stockData/MSFT', function() {
    doSomethingWith(xhr.responseText);  // 只有当数据返回的时候，才会调用
})
```

大规模运算带来的耗时在 Java/C#等支持多线程的语言环境中很容易实现，而对于 JavaScript/ActionScript 等很难，折衷的方式是将复杂运算延迟到服务器端进行；或者将复杂运算拆解成若干个耗时较少的小运算，例如 ActionScript 的伪多线程实现方式。

“一切皆异步”这个原则说来容易，但要在企业应用中以一种一致的方式进行实现很难。上例中 runInAnotherThread 的方式貌似简单，也可能出现在各种 GUI 框架的介绍中，但绝不是一个稍具规模的 RichClient 应当采用的方式。它很难作为一种编程范式被遵循，你绝不会希望看到在你的代码中所有用到异步的地方都 new Runnable(){...}。这样带来的问题不仅仅是异步被不被管理的到处乱扔，还带来了测试的复杂性。为了解决这些只有在至少有点规模的 RichClient 中才出现的问题，你最好也实现了“4 线程管理”（见下篇），能够实现“3 事件管理”（见下篇）更好。终极方式是将这些抽象到应用的基础框架中，使得所有的开发人员以一种一致的方式进行编程。

2 视图管理

2.1 视图生命周期管理

视图这个概念在 WEB 开发中几乎被忽略。这里所说的视图是指页面、页面块等界面元素。在 WEB 开发中，视图的生命周期很短：在进入页面的时候创建，在离开页面的时候销毁。一不小心页面被弄糟了，或者不能按照预期的渲染了，点下**刷新**按钮，整个世界一片清净。

WEB 下的视图导航也是如此自然。基于超链接的方式，每点击一次，就能够打开一个新的页面，旧的页面被浏览器销毁，新的页面诞生。（这里不考虑 AJAX 或者其他 JavaScript 特效）

如果把这种想法带入到 RichClient 开发，后果会很糟糕。每当点击按钮或者进行其他操作需要导航到新的窗口，你不加任何限制的创建新窗口或者新的视图。然而 CPU 不是无限的。创建一个新的视图通常是很耗 CPU 和内存的。系统响应会变慢。用户会抱怨，拒绝付钱，于是因为饥饿，你的狗再次离开了你。

每次新创建视图产生的严重后果并不仅仅是非功能性的，还包括功能性的缺失。如果你用过 Skype，当你在给张三通话的时候，再次点击张三并且进行通话，你会发现刚刚的通话界面会弹出来，而不是开启新窗口。在我们的一个项目中，有一个功能：点击软件界面上的电话号码就能开启一个新窗口，并直接连到桌上的电话拨号通话。可以想象，如果每次都会弹出新的窗口，软件的逻辑是根本错误的。

如何解决这个问题？最简单的方式是将所有已知的视图全都保存到本地的一个缓存中，我们命名为 ViewFactory，当需要进行获取某个视图的时候，直接从 ViewFactory 拿到，如果没有创建，那么创建，并放到 Cache 中：

```
class ViewFactory {
    cache = {}
    View getView(Object key) {
       if cache.contains(key) {
          return cache[key]
       }
       cache[key] = createView(key)
       return cache[key]
   }
}
```

需要注意的是，ViewFactory 中 key 的选择。对于简单的应用，key 可以干脆就是某个单独窗口的类名。例如整个系统中往往只有一个配置窗口，那么 key 就是这个类名；对于需要复用的窗口，往往需要根据其业务主键来创建相应的视图。例如代码中只有一个 UserDetailWindow，需要用来展示不同用户的信息。当需要同时显示两个以上的用户信息的时候，用同一个窗口实例显然不对。这时候 key 的选择可以是类名+用户 ID。

2.2 视图导航

上面的方案并没有解决导航的问题。导航需要解决的问题有两个，如何导航以及如何在导航时传递数据。这时候不得不羡慕 WEB 的解决方式。我要访问 ID 为 1 的用户信息，只需要访问类似于 users/1 的页面就好；需要访问搜索结果第 5 页，只需要访问 /search?q=someword&page=5 就好。这里/search 是视图，q=someword 和 page=5 是传递的数据。目前我还没有发现任何一本书来讲述如何进行视图导航。我们的方式是实现一个 Navigator 类用来导航，Navigator 依赖于前面提到的 ViewFactory：

```
class Navigator {

    Navigator(ViewFactory viewFactory) {
        this.viewFactory = viewFactory;
    }

    void goTo(Object viewKey) {
        this.viewFactory.getView(viewKey).show()
    }

}
```

（这个类看起来跟 ViewFactory 没什么大的差别，但他们逻辑上是完全不同，并且下面的扩展中会增强）

这样是可以解决问题的。如果要在不同的视图之间传递数据，只需要对 Navigator.goTo 方法稍加扩展，多添加一个参数就能够传递参数了。例如，在用户列表窗口点击用户名，发送一条消息并打开聊天窗口，可以写为：

```
void messageButton_clicked() {
    Navigator.goTo("ChatWindow#userId", "聊天消息")
}
```

然而这种方式并不完美。当你发现大量的数据在窗口之间交互的时候，这种将主动权交给调用方控制的方式，会给状态同步带来不少麻烦；如果你使用了本地存储，它越过存储层直接与服务器交互的方式也会带来不少的不便之处。更好的方式是使用“3 事件管理”（见下篇）。当然，如果窗口之间导航不存在数据传递，基于 Navigator 的方式仍然简单并且可用。

作者简介：陈金洲，Buffalo Ajax Framework 作者，ThoughtWorks 咨询师，现居北京。目前的工作主要集中在 RichClient 开发，同时一直对 Web 可用性进行观察，并对其实现保持兴趣。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-partiii

第四章

RichClient/RIA 原则与实践(下) 4

陈金洲 ThoughtWorks 公司咨询师

3 事件管理

事件管理应当是整个 RichClient/RIA 开发中的最难以把握的部分。这部分控制的好，你的程序用起来将如行云流水，用户的思维不会被打断。任何一个做 RichClient 开发的程序员，可以对其他方面毫无所知，但这部分应当非常熟悉。事件是 RichClient 的核心，是“**一切皆异步**”的终极实现。前面所说的例子，实际上可以被抽象为事件，例如第一个，获取股票数据，从事件的观点看，应该是：

- 开始获取股票数据
- 正在获取股票数据
- 获取数据完成
- 获取数据失败

看起来相当复杂。然而这样去考虑的时候，你可以将执行计算与界面展现清晰的分开。界面只需要响应事件，运算可以在另外的地方悄悄的进行，并当任务完成或者失败的是时候报告相应的事件。从经验看来，往往同样的数据会在不同的地方进行不同的展示，例如 skype 在通话的时候这个人的头像会显示为占线，而具体的通话窗口中又是另外不同的展现；MSN 的个人签名在好友列表窗口中显示为一个点击可以编辑控件，而同时在聊天窗口显示为一个不能点击只能看的标签。这是 RichClient 的特性，你永远不知道同一份数据会以什么形式来展现，更要命的是，当数据在一个地方更新的时候，其他所有能展现的地方都需要同时做相应的更新。如果我们仍然以第一部分的例子，简单采用 runInAnoterThread 是完全不能解决这个问题的。

我们曾经犯过一些很严重的错误，导致最终即便重构都积重难返。无视事件的抽象带来的影响是架构级别的，小修小补将无济于事。

事件的实现方式可以有很多种。对于没有事件支持的语言，接口或者干脆某一个约束的方法就可以。有事件支持的语言能够享受到好处，但仍然是语法级别的，根本是一样的。观察者模式在这里很好用。仍然以股票为例，被观察的对象就是获取股票数据对象 StockDataRetriver，观察的就是 StockWindow：

```
StockDataRetriver {
    observers: []

    retrieve() {
        try {
            theData = ...// 从远程获取数据
            observers.each {|o| o.stockDataReady(theData)}// 触发数据获取成功事件
        } catch {
            observers.each { |o| o.stockDataFailed() }  // 触发事件获取失败事件
        }
    }
}


StockDataRetriver.observers.add(StockWindow)  // 将StockWindow加入到观察者队列


StockWindow {
    stockDataReady(theData) {
        showDataInUIThread(); // 在UI线程显示数据
    }
    stockDataFailed() {
        showErrorInUIThread(); // 在UI线程显示错误
    }
}
```

你会发现代码变得简单。UI 与计算之间的耦合被事件解开，并且区分 UI 线程与运算线程之间也变得容易。当尝试以事件的视角去观察整个应用程序的时候，你会更关注于用户与界面之间的交互。

让我们继续抽象。如果把“获取股票数据”这个按钮点击，让 StockDataRetriver 去获取数据当作事件来处理，应该怎么写呢？将按钮作为被观察者，StockDataRetriver 作为观察者显然不好，好不容易分开的耦合又黏在一起。引入一个中间的 Events 看起来不错：

```
Events {
    listeners: {}

    register(eventId, listener) {
        listeners[eventId].add(listener)
    }

    broadcast(eventId) {
        listeners[eventId].observers.each{|o| o.doSomething(); }
```

```
    }
}
```

Events 中维护了一个 listeners 的列表，它是一个简单的 Hash 结构，key 是 eventId，value 是 observer 的列表；它提供了两个方法，用来注册事件监听以及通知事件产生。对于上面的案例，可以先注册 StockDataRetriver 为一个观察者，观察 start_retrive_stock_data 事件：

```
  Events.register('start_retrive_stock_data', StockDataRetriever)
```

当点击“获取股票数据”按钮的时候，可以是这样：

```
Events.broadcast('start_retrive_stock_data')
```

你会发现 StockDataRetriver 能够老老实实的开始获取数据了。

需要注意的是，并非将所有事件定义为全局事件是一个好的实践。在更大规模的系统中，将事件进行有效整理和分级是有好处的。在强类型的语言（如 Java/C#）中，抽象出强类型的 EventId，能够帮助理解系统和进行编程，避免到处进行强制类型转换。例如，StockEvent：

```
StockDataLoadedEvent {
    StockData theData;
    StockDataLoadedEvent(StockData theData);
}

Event.broadcast(new StockDataLoadedEvent(loadedData))
```

这个事件的监听者能够不加类型转换的获得 StockData 数据。上面的例子是不支持事件的语言，C#语言支持自定义强类型的事件，用起来要自然一些：

```
delegate void StockDataLoaded(StockData theData)
```

事件管理原则我相信并不难理解。然而困难的是具体实现。对一个新的 UI 框架不熟悉的时候，我们经常在“代码的优美”与“界面提供的特性”之间徘徊。实现这样的一个事件架构需要在项目一开始就稍具雏形，并且所有的事件都有良好的命名和管理。避免在命名、使用事件的时候的随意性，对于让代码可读、应用稳定有非常大的意义。一个好的事件管理、通知机制是一个良好 RichClient 应用的根本基础。一般说来，你正在使用的编程平台如 Swing/WinForm /WPF/Flex 等能够提供良好的事件响应机制，即监听事件、onXXX 等，但一般没有统一的事件的监听和管理机制。对于架构师，对于要使用的编程平台 对于这些的原生支持要了熟于心 ,在编写这样的事件架构的时候也能兼顾这些语言、平台提供给你的支持。

采用了事件的事件后，你不得不同时实践“**线程管理**”，因为事件一般来说意味着将耗时的操作放到别的地方完成，当完成的时候进行事件通知。简单的模式下，你可以在所有需要进行异步运算的地方，将运算放到另外一个线程，如 ThreadPool.QueueUserWorkItem，在运算完成的时候通知事件。但从资源的角度考虑，将这些线程资源有效的管理也是很重要的，在“线程管理”部分有详细的阐述。另外，如果能将你的应用转变为数据驱动的，你需要关注

“**缓存以及本地存储**”。

4 线程管理

在 WEB 开发几乎无需考虑线程，所有的页面渲染由浏览器完成，浏览器会异步的进行文字和图片的渲染。我们只需要写界面和 JavaScript 就好。如果你认同“一切皆异步”，你一定得考虑线程管理。

毫无管理的线程处理是这样的 :凡是需要进行异步调用的地方 ,都新起一个线程来进行运算，例如前面提到的 runInThread 的实现。这种方式如果托管在“事件管理”之下，问题不大，只会给测试带来一些麻烦：你不得不 wait 一段时间来确定是否耗时操作完成。这种方式很山寨，也无法实现更高级功能。更好 的的方式是将这些线程资源进行统筹管理。

线程的管理的核心功能是用来统一化所有的耗时操作，最简单的 TaskExecutor 如下：

```
TaskExecutor {
    void pendTask(task) { //task: 耗时操作任务
        runInThread {
            task.run(); // 运行任务
        }
    }
}

RetrieveStockDataTask extends Task {
    void run() {
        theData = ... // 直接获取远程数据，不用在另外线程中执行
        Events.broadcast(new StockDataLoadedEvent(theData)) // 广播事件
    }
}
```

需要进行这个操作的时候，只需要执行类似于下面的代码：

```
TaskExecutor.pendTask(new RetrieveStockDataTask())
```

好处很明显。通过引入 TaskExecutor，所有线程管理放在同一个地方，耗时操作不需要自行维护线程的生命周期。你可以在 TaskExecutor 中灵活定义线程策略实现一些有趣的效果，如暂停执行，监控任务状况等，如果你愿意，为了更好的进行调试跟踪，你甚至可以将所有的任务以同步的方式执行。

耗时任务的定义与执行被分开，使得在任务内部能够按照正常的方式进行编码。测试也很容易写了。

不同的语言平台会提供不同的线程管理能力。.NET2.0 提供了 BackgroundWorker，提供了一

序列对多线程调用的封装，事件如开始调用，调用，跨线程返回值，报告运算进度等等。它内部也实现了对线程的调度处理。在你要开始实现类似的 TaskExecutor 时，参考一下它的 API 设计会有参考价值。Java 6 提供的 Executor 也不错。

一个完善的 TaskExecutor 可以包含如下功能：

- Task 的定义：一个通用的任务定义。最简单的就是 run()，复杂的可以加上生命周期的管理：start()、end()、success()、fail()..取决于要控制到多么细致的粒度。
- pendTask，将任务放入运算线程中
- reportStatus，报告运算状态
- 事件：任务完成
- 事件：任务失败

写这样的一个线程管理的不难。最简单的实现就是每当 pendTask 的时候新开线程，当运算结束的时候报告状态。或者使用像 BackgroundWorker 或者 Executor 这样的高级 API。对于像 ActionScript/JavaScript 这样的，只能用伪线程，或者干脆将无法拆解的任务扔到服务器端完成。

5 缓存与本地存储

纯粹的 B/S 结构 ,浏览器不持有任何数据 ,包括基本不变的界面和实际展现的数据。RichClient 的一大进步是将界面部分本地持有，与服务器只作数据通讯，从而降低数据流量。像《魔兽世界》10 多 G 的超大型客户端，在普通的拨号网络都可以顺畅的游戏。

缓存与本地存储之间的差别在于，前者是在线模式下，将一段时间不变的数据缓存，最少的与服务器进行交互，更快的响应客户；后者是在离线模式下，应用仍然能够完成某些功能。一般来说，凡是需要类似于“查看 XXX 历史”功能的，需要“点击列表查看详细信息”的，都会存在本地存储的必要，无论这个功能是否需要向 用户开放。

无论是缓存还是本地存储 ,最需要处理的问题如何处理本地数据与服务器数据之间的更新机制。当新数据来的时候，当旧数据更新的时候，当数据被删除的时候，等等。一般来说，引入这个实践，最好也实现基于数据变化的“事件管理”。如果能够实现“客户机-服务器数据交互模式”那就更完美了。

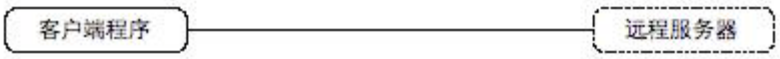
我们犯过这样一个错误。系统启动的时候 ,将当前用户的联系人列表读取出来 ,放到内存中。

当用户双击这个联系人的时候，弹出这个联系人的详细信息窗口。由于没有本地存储，由于采用了 Navigator 方式的导航，于是很自然的采用了 Navigator.goTo('ContactDetailWindow', theContactInfo)。由于列表页面一般是不变的，因此显示出来的永远是那份旧的数据。后来有了编辑联系人信息的功能，为了总是显示更新的数据，我们将调用更改为 Navigator.goTo('ContactDetailWindow', 'contactId')，然后在 ContactDetailWindow 中按照 contactId 把联系人信息重新读取一次。远在南非的用户抱怨慢。还好我没养狗，没有狗离开我。后来我们慢慢的实现了本地存储，所有的数据读取都从这个地方获得。当数据需要更新的时候，直接更新这个本地存储。

本地存储会在根本上影响 RichClient 程序的架构。除非本地不保存任何信息，否则本地存储一定需要优先考虑。某些编程平台需要你在本地存储界面和数据，如 Google Gears 的本地存储，置于 Adobe Air 的 AJAX 应用等，某些编程平台只需要存储数据，因为界面完全是本地绘制的，如 Java/JavaFX/WinForm/WPF 等。缓存界面与缓存数据在实现上差别很大。

本地存储的存储机制最好是采用某一种基于文件的关系数据库，如 SQLite、H2（HypersonicSQL）、Firebird 等。一旦确定要采用本地存储，就从成熟的数据库中选择一个，而不要尝试着自己写基于文件的某种缓存机制。你会发现到最后你实现了一个山寨版的数据库。

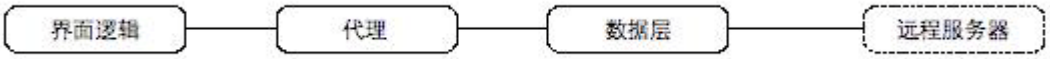
在没有考虑本地存储之前，与远端的数据访问是直接连接的：



我们上面的例子说明，一旦考虑使用本地存储，就不能直接访问远程服务器，那么就需要一个中间的数据层：

数据层的主要职责是维护本地存储与远程服务器之间的数据同步，并提供与应用相关的数据缓存、更新机制。数据更新机制有两种，一种是 Proxy（代理）模式，一种是自动同步模式。

代理模式比较容易理解。每当需要访问数据的时候，将请求发送到这个代理。这个代理会检查本地是否可用，如果可用，如缓存处于有效期，那么直接从本地读取数据，否则它会真正去访问远端服务器，获取数据，更新缓存并返回数据。这种手工处理同步的方式简单并且容易控制。当应用处于离线模式的时候仍然可以工作的很好。



自动同步模式下，客户端变成都针对本地数据层。有一个健壮的自动同步机制与服务器的保持长连接，保证数据一直都是更新的。这种方式在应用需要完全本地可运行的时候工作的非常好。如果设计得好，自动同步方式健壮的话，这种方式会给编程带来极大的便利。

说到同步，很多人会考虑数据库自带的自动同步机制。我完全不推荐数据库自带的机制。他们的设计初衷本身是为了数据库备份，以及可扩展性 （ Scalability ）的考虑。在应用层面，数据库的同步机制往往不知道具体应用需要进行哪些数据的同步，同步周期等等。更致命的是，这种机制或多或少会要求客户端与服务器端具备类似的数据库表结构，迁就这样的设计会给客户端的缓存表设计带来很大的局限。另外，它对客户机-服务器连接也存在一定的局限 性，例如需要开放特定端口，特定服务等等。对于纯粹的 Internet 应用，这种方式更是完全不可行的，你根本不知道远程数据库的结构，例如 Flickr, Google Docs.

当本地存储+自动同步机制与“事件管理”都实现的时候，应用会是一种全新的架构：基于数据驱动的事件结构。对于所有本地数据的增删改都定义为事件，将关心这些数据的视图都注册为响应的观察者 ,彻底将数据的变化于展现隔离。界面永远只是被动的响应数据的变化，在我看来，这是最极致的方式。

结尾

限于篇幅，这篇文章并没有很深入的讨论每一种原则/实践。同时还有一些在 RichClient 中需要考虑的东西我们并没有讨论：

- **纯 Internat 应用离线模式的实现。**像 AdobeAir/Google Gears 都有离线模式和本地存储的支持，他们的特点是缓存的不仅仅是数据，还包括界面。虽然常规的企业应用不太可能包含这些特性，但也具备借鉴意义。
- **状态的控制。**例如管理员能够看到编辑按钮而普通用户无法看见，例如不同操作系统下的快捷键不同。简单情况下，通过 if- else 或者对应编程平台下提供的绑定能够完成，然

而涉及到更复杂的情况时，特别是网络游戏中大量互斥状态时，一个设计良好的分层状态机模型能够解决这些问题。如何定义、分析这些状态之间的互斥、并行关系，也是处理超复杂

- **测试性。**如何对 RichClient 进行测试？特别是像 WPF、JavaFX、Adobe Air 等用 Runtime+编程实现的框架。它们控制了视图的创建过程，并且倾向于绑定来进行界面更新。采用传统的 MVP/MVC 方式会带来巨大的不必要的工作量（我们这么做过！），而且测试带来的价值并没有想象那么高。
- **客户机-服务器数据交互模式。**如何进行客户机服务器之间的数据交互？最简单的方式是类似于 Http Request/Response。这种方式对于单用户程序工作得很好，但当用户之间需要进行交互的时候，会面临巨大挑战。例如，股票代理人关注亚洲银行板块，刚好有一篇新的关于这方面的评论出现，股票代理人需要在最多 5 分钟内知道这个消息。如果是 Http Request/Response，你不得不做每隔 5 分钟刷一次的蠢事，虽然大多数时候都不会给你数据。项目一旦开始，就应当仔细考虑是否存在这样的需求来选择如何进行交互。这部分与本地存储也有密切的关系。
- **部署方式。**RichClient 与 B/S 直接最大的差异就是，它需要本地安装。如何进行版本检测以及自动升级？如何进行分发？在大规模访问的时候如何进行服务器端分布式部署？这些问题有些被新技术解决了，例如 Adobe Air 以及 Google Gears，但仍然存在考虑的空间。如果是一个安全要求较高的应用，还需要考虑两端之间的安全加密以及客户端正确性验证。新的 UI 框架层出不穷。开始一个新的 RichClient 项目的时候，作为架构师/Tech Lead 首先应当关注的不是华丽的界面和效果，应当观察如何将上述原则和时间华丽的界面框架结合起来。就像我们开始一个 web 项目就会考虑 domain 层、持久层、服务层、web 层的技术选型一样，这些原则和实践也是项目一开始就考虑的问题。

感谢

感谢我的同事周小强、付莹在我写作过程中提供的无私的建议和帮助。小强推荐了介绍 Google Gears 架构的链接，让我能够写作“本地存储”部分有了更深的体会。

这篇文章是我近两年来在 RichClient 工作、网络游戏、WebGame 众多思考的一个集合。我尝试过 JavaFX/WPF/AdobAir 以及相关的文章，然而大多数的例子都是从华丽的界面入手，没有实践相关的内容。有意思的反而是《大型多人在线游戏开发》这本书，给了我在企业 RichClient 开发很多启发。我们曾经犯了很多错误，也获得了许多经验，以后我

们应当能做得更好。

参考

- .NET 的 BackgroundWorker 定义：http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx
- ActionScript 的伪线程：http://blogs.adobe.com/aharui/2008/01/threads_in_actionscript_3.html
- Google Gears 架构：http://code.google.com/apis/gears/architecture.html

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-partiii-ii

第五章 为什么我们要放弃 Subversion 5

胡凯 ThoughtWorks 公司敏捷咨询师

Subversion 曾经是我们亲密无间的战友，但自从一年前部分团队成员去了美国，我们和 Subversion 的关系就开始出现了裂痕，首先是将 Subversion 服务器架设在美国后，中国开发人员频繁进行的一些操作变得非常缓慢，本来通过追溯代码历史便可找出原因的问题，却因为网速缓慢，导致开发者将大量的时间耗费在等待服务器响应，而不是分析问题上。其次，由于缺乏 IT 基础设施方面的投资以及完善的备份策略，数次因为网络原因或者服务器宕机，导致团队无法从中国访问版本管理服务器，正常的提交、更新操作都无法进行，最严重的是版本管理服务器曾经在发布之前出现故障，导致服务器上的数据不得不回滚到九天以前，给发布带来了很大的风险。

从现象上看，版本管理服务器不在本地是遭遇速度瓶颈的主要原因，本质却是由于版本管理工具不能很好的根据团队的规模和结构伸缩。对我们而言，比较理想的版本管理解决方案是在中美两地架设服务器，加快各个操作的执行速度，服务器之间自动同步来平衡两地对于速度和代码集成的要求。然而采用 Subversion 作为版本管理工具，决定了服务器仅能架设在一地。SVK 可以解决部分问题，但它的缺陷太多，操作起来非常不便。我们所面临的备份问题则是由于在 Subversion 的设计中，所有的元数据仅仅保存在服务器上，一旦服务器出现意外，元数据所包含的宝贵信息便无从恢复。之前的教训让我们认识到如果采用 Subversion 作为版本管理工具，就不能仅仅乐观的假设服务器不会出错，必须有详尽可行的备份计划，通过不断备份来规避风险。

Subversion 的这些天生缺陷让我们把目光投向了 DVCS（分布式版本管理工具），在这个家族中，比较成熟的产品有 Git、Mercurial 和 Bzr，相比之下，由于 Mercurial 对 Linux，Mac 和 Windows 平台有良好的支持，支持通过 Web 方式访问代码库，并存在成熟的 IntelliJ、Eclipse 插件，最终成为了胜出者，时至今日，它已经在我们团队服役超过 1 年了，从 0.9.4 到 1.1.2，每一次版本的更新，都让我们愈加喜欢这个设计精巧产品。那么较之 Subversion，Mercurial 究竟胜在哪里？

快速可靠

Mercurial 带给团队的第一个体验就是快，原因很简单，由于 DVCS 的工作目录与中央仓库（Central Repository）别无二致，同样保存了全部的元数据，那么 Subversion 需要通过网络完成的操作(诸如提交、追溯历史、更新等)，Mercurial 可以在离线条件下通过操作本地仓库完成（图-1）。

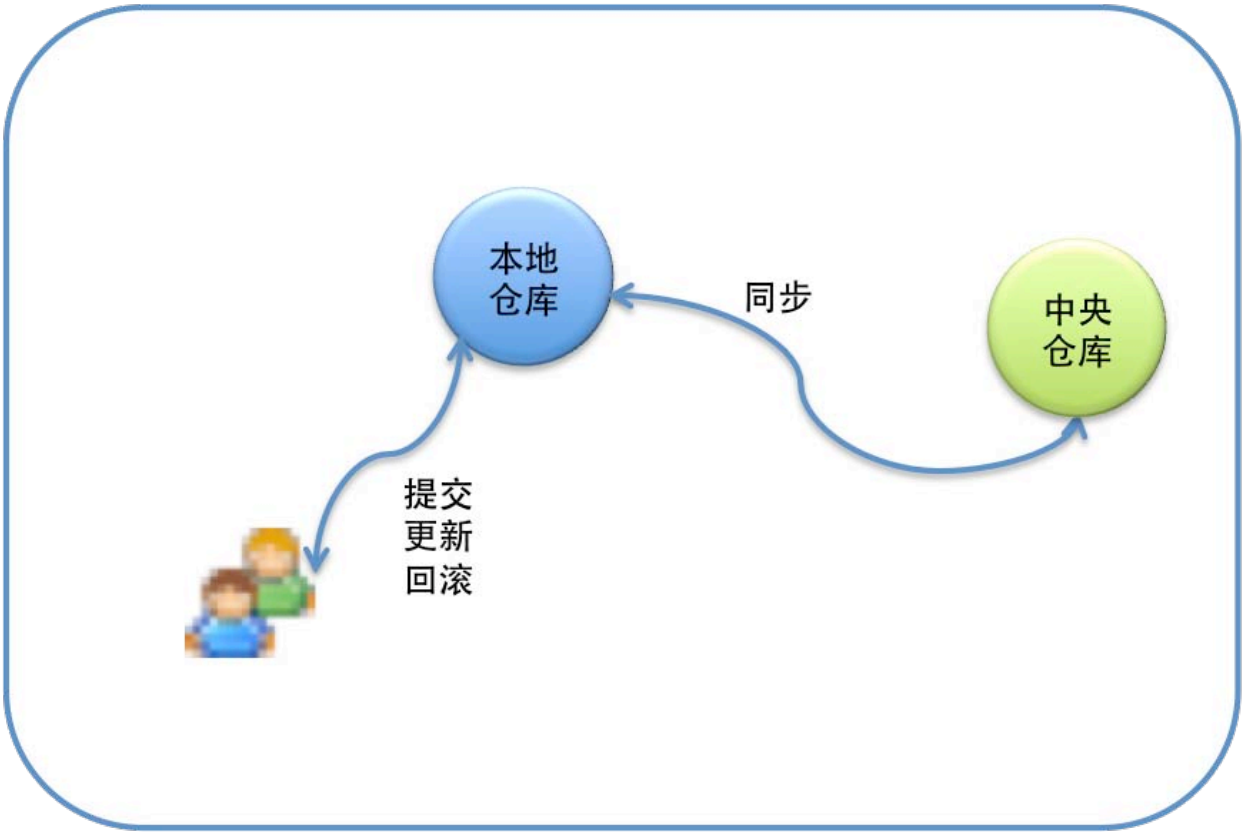


图-1

通过减少与中央仓库的通信, Mercurial 加快了操作速度，减小了网络环境对团队的影响，非常符合我们的需求。这种速度和可靠性的提高，对于时刻与版本管理工具打交道的开发者是一种非常愉悦的工作体验。此外，包含了全部元数据的工作目录可以在中央仓库出现问题时（图 2-b）成为备用仓库（图 2-c），而整个过程只需运行一条命令 即可。

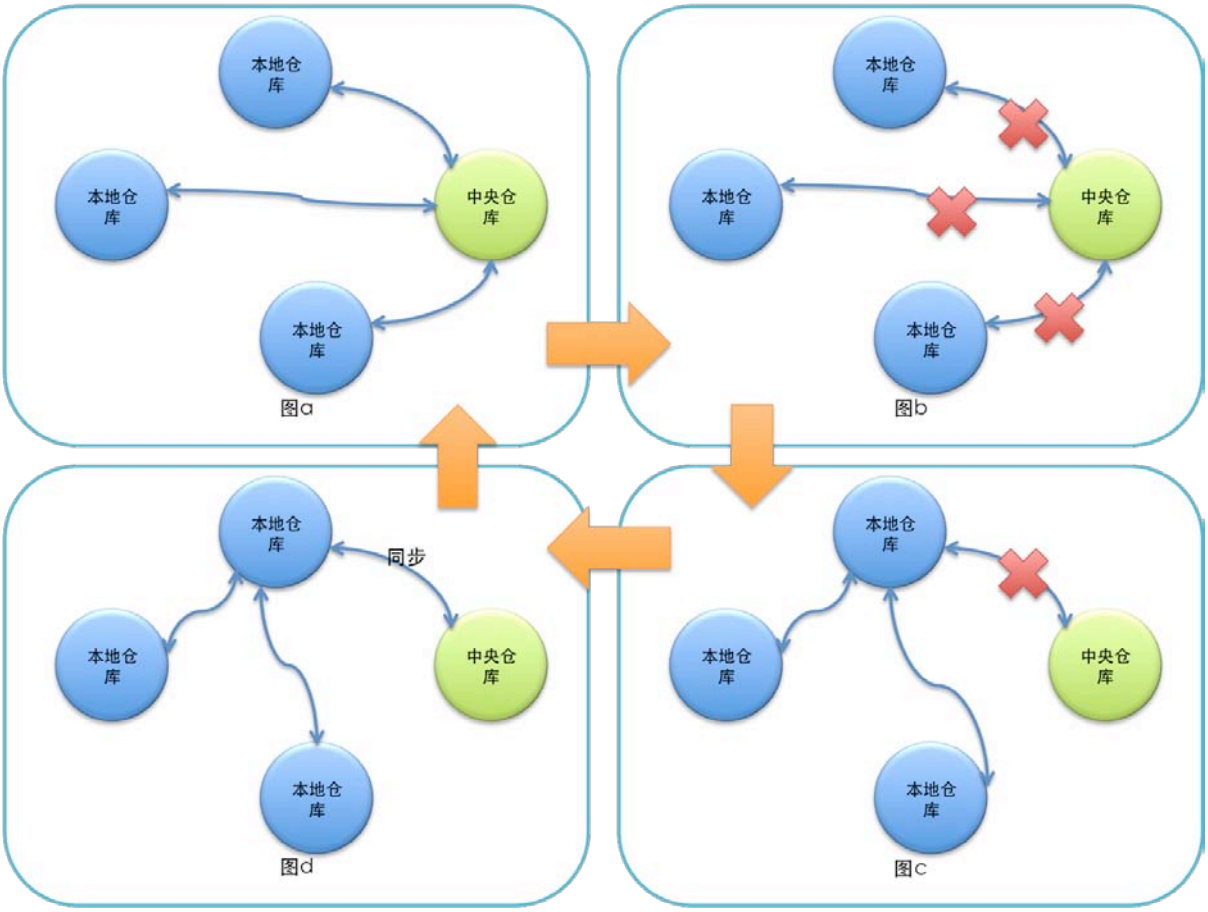


图-2

这样，在 IS 部门修复中央仓库的过程中，开发团队依然可以通过备用仓库交换修订，日常工作在没有中央仓库的情况下依然可以正常开展，中央仓库恢复后，再将宕机期间所有的修订通过备用仓库同步到中央仓库上（图 2-d），这套机制可以作为经费和硬件设施有限团队的备份方案。即便中央仓库完全损毁，所造成的损失也非常有限，避免了使用 CVCS 时将“所有鸡蛋放在一个篮子里”的风险。

便于协同工作

在日常的工作中，我们常常利用 Mercurial 灵活的分支合并来共享修改，协同工作。几个月前在印度发布产品时，我需要在新的工作站上安装开发环境，由于代码库庞大而且网速缓慢，克隆中央仓库的操作需要花费数小时才能完成（图 3-a），Mercurial 的灵活性使我可以将工作站指向已经存有代码的笔记本电脑来执行克隆操作（图 3-b），在数分钟后工作站就完成了全部的克隆操作，之后再将它指向中央仓库（图 3-c），即可正常提交/更新代码，大大节省了时间，提高了效率。

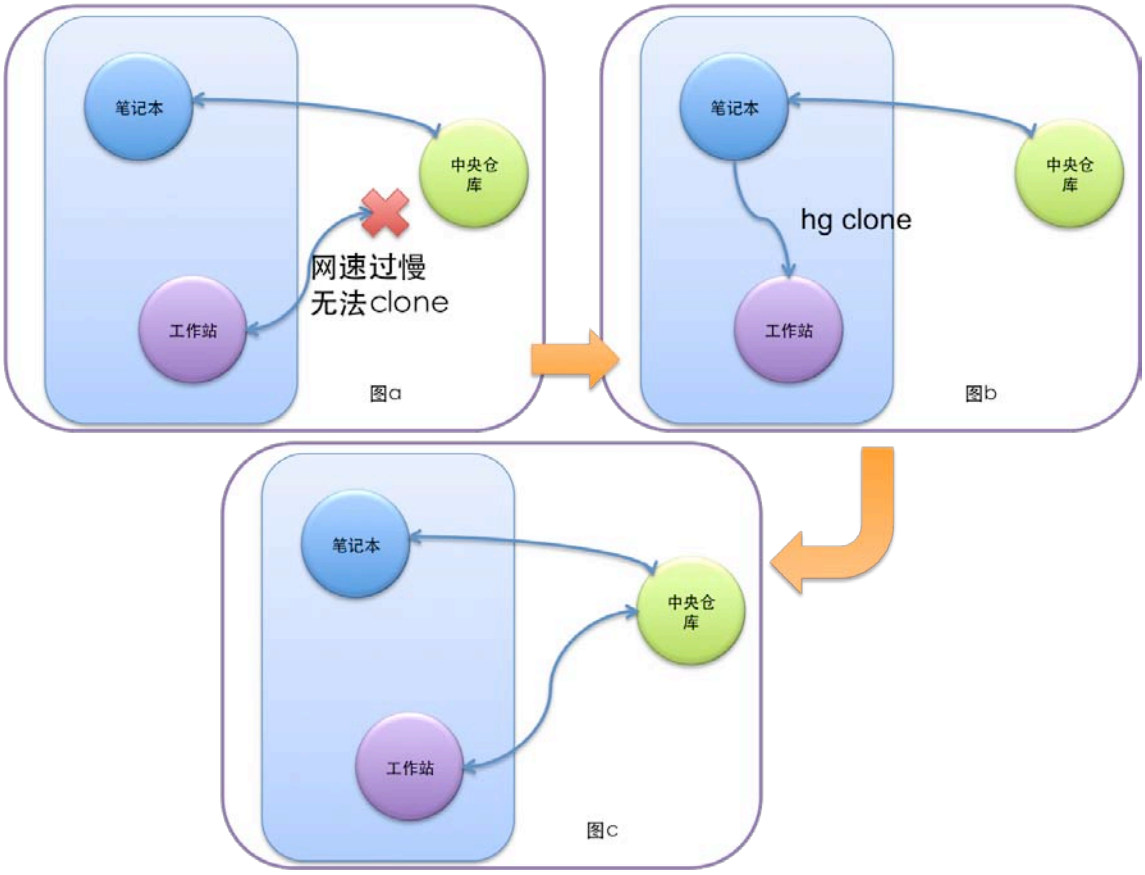


图-3

在另一个场景中，由于我所在团队使用 Linux 作为开发环境，在急于验证某些功能是否在 Windows 平台可以正常工作时，我们会将代码在 Linux 工作站本地提交，再将 Windows 工作站的工作目录指向 Linux 工作站，获取更新（图 4-b）。之后，在 Windows 平台验证功能，如果功能存在问题，可以修复后再将修订从 Windows 工作站提交到 Linux 工作站（图 4-c），最终由 Linux 工作站运行测试并将全部更新同步到中央仓库（图 4-d）。Mercurial 的分布式特性让开发团队敏捷的分享修订，更有效率的开发。

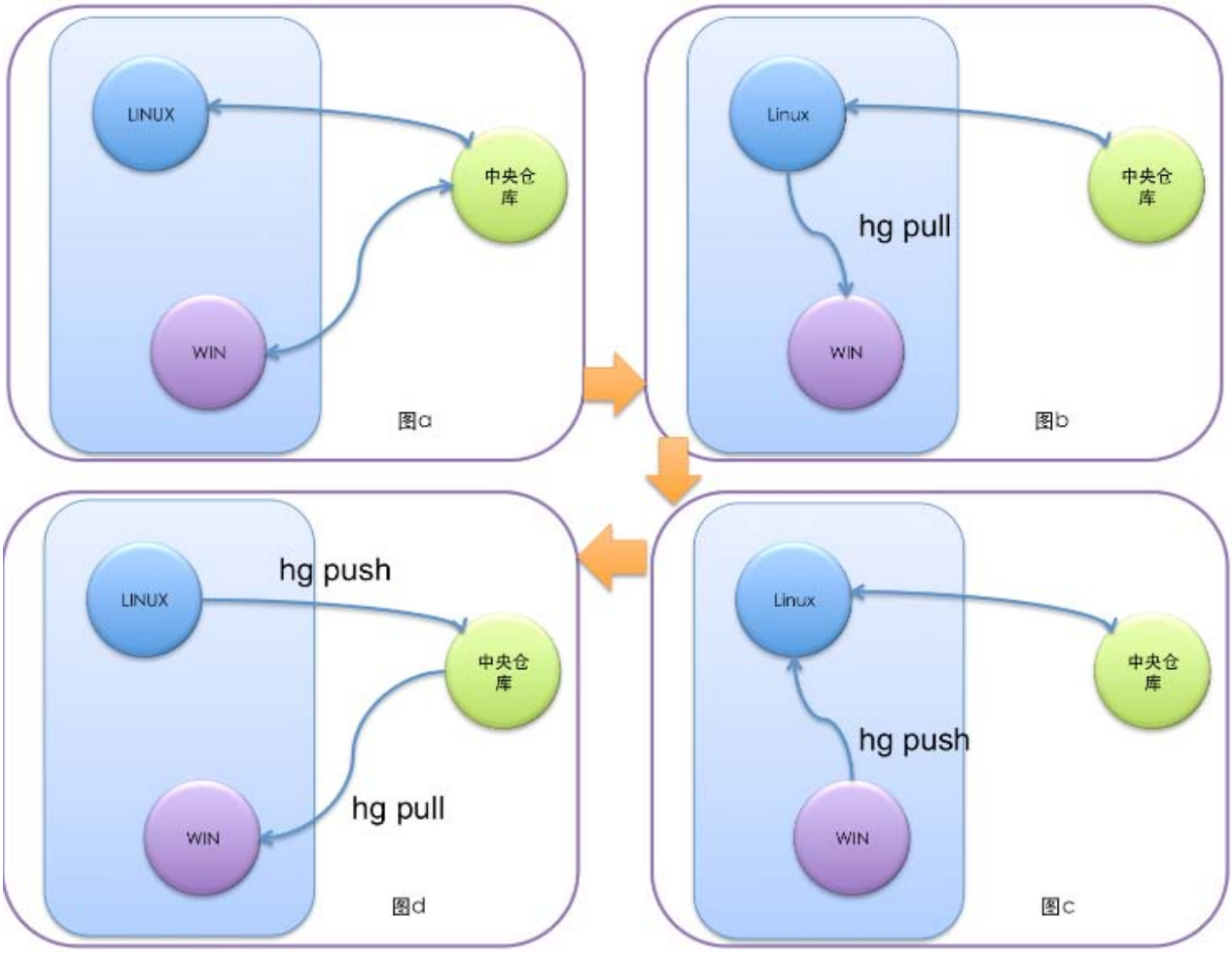


图-4

对小步前进友好

本地仓库的存在，使 Mercurial 对小步前进更加友好 。小步前进意味着开发者在不破坏任何现有功能的前提下，每次修改少量代码并提交。这两个需求让使用集中式版本管理工具的开发者常常处于两难的境地，”不破坏现有功能“与“每次修改少量代码并提交”意味着存在便于分析的细粒度需求以及开发人员必须掌握增量式的对象建模、重构，数据库设计、迁移等技术。难于小步前进体现的是团队成员经验和技术的欠缺，然而解决这些问题不是一朝一夕之功，本地仓库的存在给了开发者更大的自由，允许开发者频繁提交而无需顾忌是否每 一次提交都不会破坏现有功能，在代码经过若干次提交到达稳定状态时再与中央数据库同步。通过使用 Mercurial，使得小步前进这个实践得以在团队开展，在大家体会到实践带来的好处后，再追求高质量的小步前进。

学习曲线低

Mecurial 灵活的分支合并策略使我们可以选择与 CVCS(集中式版本管理工具如 Subversion，CVS 等)非常相似的架构（如图-1 所示），这样，团队在更换版本管理工具后依然可以工作在相对熟悉的环境中。在(图-1)所示的结构中，开发者需首先架设中央仓库，再从中央仓库克隆出工作目录，在开发过程中，开发者将修订提交到本地仓库，最后，在功能完成后将本地仓库的所有修改同步到中央仓库。除了最后一步，其余步骤和 CVCS 完全一致，开发者可以很快对 Mercurial 总体架构建立初步的认识。Mercurial 的基本命令与 CVS/Subversion 非常类似，熟悉 CVS /Subversion 的团队可以依然工作在熟悉的命令行环境。从结构到命令，Mercurial 做到了对 CVCS 用户友好，降低了学习曲线，让开发者可以相对轻松的跨出从 CVCS 到 DVCS 的第一步。如果仅仅想作一下尝试，又或者公司的政策不允许将版本管理工具从 Subversion 迁移到 Mercurial，Mercurial 还提供了 HgSubversion 插件可供选择，它可以将 Mercurial 作为 Subverion 的客户端使用，这样，既可以保留 Subversion 版本管理服务器，又可以在本地采用 Mercurial 来享受 DVCS 的种种好处，使开发者可以非常安全过渡到 DVCS 环境。

总结

毫无疑问 Subversion 是非常优秀的版本管理工具，但是它有自己的适用范围，并不是银弹。抛弃 Subversion，也不因为我们是新技术的狂热分子，而是它无法伸缩来适应团队的结构变化。对于希望尝试 DVCS 的团队，我的几个建议是：决策者首先要识别团队的痛点，对问题域有清醒的认识，而不能仅仅追赶技术潮流，其次是使用它、慢慢的接受它，如果团队仅仅止于理论方面的学习，各种方案的论证，是无法掌握 DVCS 并利用它来提高团队效率的，最后整个团队需要持续学习 DVCS 背后的设计思想，对于问题域的抽象以及丰富的插件的使用方法。这些知识将直接或间接的帮助团队提高进行代码版本管理的能力，更有效律的管理代码。

感谢

感谢我的同事杨哈达，初悦欣，乔梁和陈金洲在我写作过程中提供的无私的建议和帮助。和杨哈达的讨论让我对于 DVCS 的理论有了更清楚的认识，初悦欣和我一起重构了插图，乔梁和陈金洲对于文章的结构提出了很多意见和建议。

这篇文章是我在工作中从熟悉的 Subversion 迁移到 Mercurial 的经验分享，在这个过程中有不适应，也曾经犯了很多错误并获得了许多经验。希望读者能从这篇文章中有所收获，从

DVCS 在我们团队的实践了解到 DVCS 可能带给自己团队的价值 ,更有信心的进入 DVCS 领域。

作者简介：胡凯，ThoughtWorks 敏捷咨询师，近 2 年一直从事持续集成工具 Cruise 以及 CruiseControl 的设计开发工作。 对于 Web 开发，敏捷实践，开源与社区活动有浓厚的兴趣，可以访问他的个人博客进行更多的了解。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-partiv

第六章

"持续集成"也需要重构

--持续集成实践在 Cruise 开发过程中的演讲

6

乔梁 ThoughtWorks 公司资深咨询师/敏捷过程教练/产品 Cruise 的项目经理

前言

持续集成是极限编程十二实践之一（1999 年 Kent Beck 编写的《解析极限编程》），最初被使用极限编程方法的开发人员所推捧，并在过去的几年中得到广泛应用，成为业界广为人知的软件开发实践。该实践用于解决软件开发过程中一个具体且重要的问题，即“确保当某个开发人员完成新的功能或修改代码后，整个软件仍旧能正常工作。”

然而，持续集成并非像大多数人想像的那样，首次部署好持续集成环境后就大功告成，一劳永逸了。恰恰相反，它与你项目中的其它产品代码一样，需要改进与重构，否则，就会使你进入一种“持续闹心”的状态，甚至可能让你觉得这件事根本不应该做，如何解决这一问题呢？对“持续集成”应用“Retrospective”和“重构”。本文将结合 Cruise 团队一年多的实际历程，讲述持续集成实践在软件开发过程中的演进。

友情提示：请读者在阅读本文时，注重文中所述的思考过程与“持续集成”的重构方式，而非产品本身。

基本持续集成——万里长征第一步

实际问题：我们需要一个持续集成环境

为什么要做持续集成不在本文讨论之内。

理论基础

持续集成基 于这样一个假设：如果两次代码集成的间隔时间越长，最终集成时痛苦的经历就会越多。而其目标有两个：一是“频繁集成”，二是 “反映代码质量”。为了做到“ 频繁集成”，

就要求任何开发人员在每次向中央代码库提交代码时，就将所有代码进行编译，打包，部署，以确保能够产生交付物。然而，“频繁集成”仅仅证明了每次提交是否可以得到交付物，而我们真正需要知道的是“这个交付物的质量如何”。如果交付物有问题，引起问题的代码则要么被回滚，要么被修正。当然，这样的任务也可以通过手工来完成，但手工工作的特点就是易出错且耗时，所以需要将其自动化。为了做好“持续集成实践”，写一个自动化构建脚本来自动构建并运行 一些自动化测试套件还是必要的。这种传统的持续集成方式常被固化于开发环节，即：开发人员安装一个专门的持续集成服务器来自动化运行这些单元测试，然后通过各种各样自动生成的测试结果分析代码的质量，这也是 CruiseControl 诞生的原因。

解决方案

最初，我们的代码并不多，自动化脚本比较简单，在一台机器上运行所有的测试也仅需要几分钟，该构建套件的运行顺序是：CheckStyle -> Compile -> UnitTest -> FunctionTest -> Report。示意如下：

```
<project name="Cruise" default="all" basedir=".">
   <target  name="all"  depends="checkStyle,  Compile,  UnitTest,FunctionTest,
Report"/>
   <target name="checkStyle">
      ....
   </target>
   <target name="Compile" >
      ...
   </target>
   <target name="UnitTest" depends="Compile">
      ...
   </target>
   <target name="FunctionTest" depends="Compile">
      ...
   </target>
   <target name="Report" depends="FunctionTest">
     ...
   </target>
 </project>
```

这也是大部分软件团队进行持续集成的起点。

目前，有很多种持续集成工具，其中不乏开源产品，如 CruiseControl 家族。项目伊始，我们使用 CruiseControl 建立了自己的持续集成服务器，整个项目的持续集成基础结构如图所示：

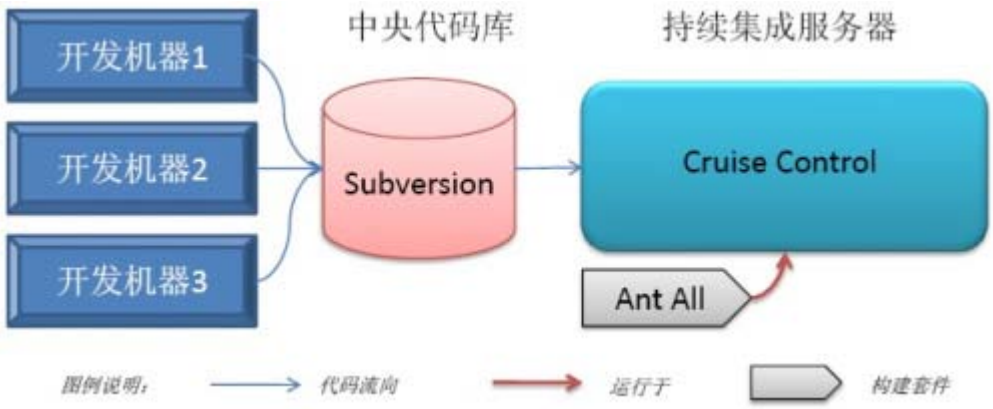


图 1 基础持续集成模式

注：得到快速反馈是开发好产品的关键。因此，我们自己就做了 Cruise 的第一个用户，首先解决自己的持续集成需求。项目开始后仅几周，我们的产品 Cruise 已经基本满足自身团队持续集成的需求，因此，我们就将 CruiseControl 替换成 Cruise，但持续集成基本结构并没有变化。

阶段化持续集成 —— 平衡的艺术

实际问题：测试越来越慢，开发人员等得不耐烦。

随着时间的推移，Cruise 每次做持续集成的运行时间很快就超过了 15 分钟（开发人员能够忍受的最大限度）。然而，为了保证 Cruise 支持大多数浏览器，我们还打算增加 Cruise 运行于不同操作系统及不同浏览器的功能测试。

理论基础

一般来说，测试代码越多，越能够正确反映代码的质量［前提：你写的测试是有意义的:)］。所以在整个生命周期中，大家都试图增加更多的测试代码。然而，越多的测试代码也意味着更长的运行时间，更慢的反馈速度。因此，我们不得不在“反馈时间”与“判断质量准确性”两者之间找到一种平衡，而“阶段化持续集成”就有了用武之地。

所谓的“阶段化持续集成”是指为不同的构建测试套件（以下称构建计划）建立不同的持续集成循环周期，由于单元测试运行时间短，反馈较快，所以可以频繁进行，而功能测试、性能测试的时间比较长，占用资源比较多，比较昂贵，所以适当减少集成次数，但一定要保证其周期性运行。因此，我们的持续集成方案很自然地分成了多个构建计划：快速构建计划让

开发人员尽可能快地得到反馈，此时我们牺牲了准确性来换取时间。但当你得到了快速反馈以后，再花多一点儿时间从其它构建计划中得到更详细的反馈。这样一来，对于同一个项目，你有多个构建计划，每个对应一种构建类型（单元测试，功能测试、功能测试等），越靠后的构建计划在运行时需要的时间就越长。如下图。

单元测试构建计划运行了多次， U1,U2...U7，其中 U4 失败了。功能测试构建计划时间较长，在该时间段内仅运行两次，F1 和 F2(失败)，而性能测试构建计划 P1 时间最长。整个持续集成系统由多个机器组成，包括一个中心服务器（Server）和多台工作站（Agent），每个构建计划都在一台工作站（agent）独立执行，以便无相互影响。

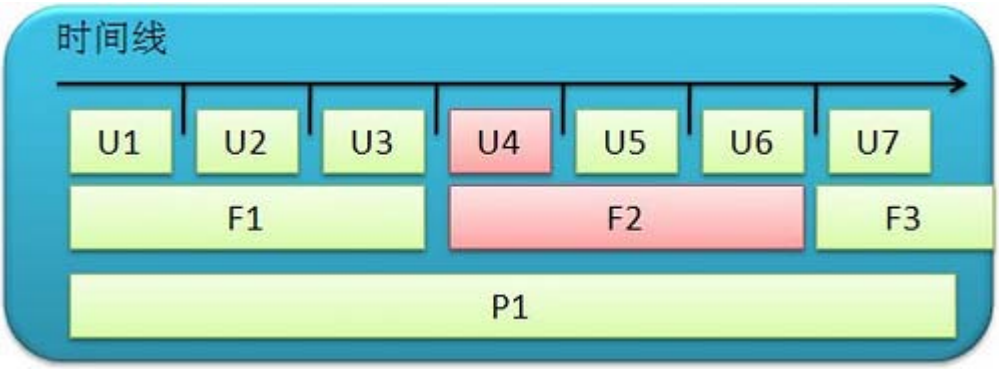


图 2 阶段式并行构建图解

解决方案

由于 Cruise 本身已经支持这种持续集成模式，所以我们的持续集成基本框架也演变成下图：

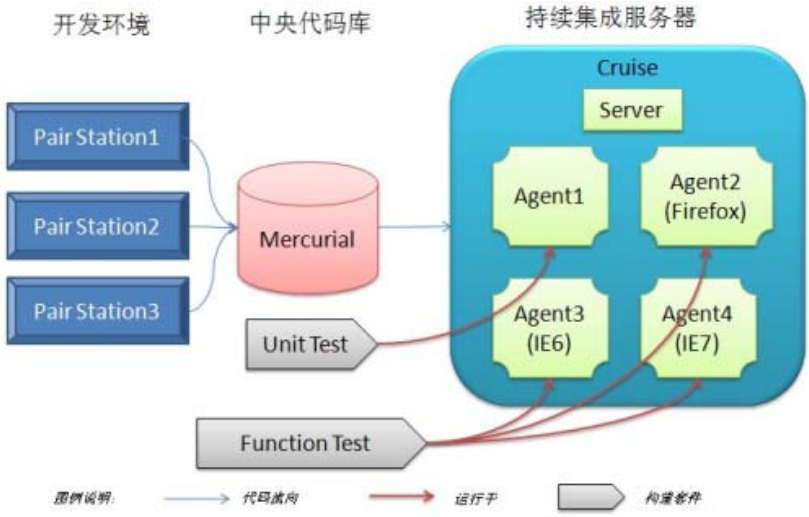


图 3 阶段式持续集成模式图解

中心服务器(Server)的责任是（1）检查中央代码库的代码是否发生变化，(2）如果代码发生了变化，将相应的构建类型分配到各自的工作站 （Agent）上运行；(3）中心仓库，保存所有信息。上图中，单元测试(Unit Test)构建类型运行于工作站 Agent1 上，每次执行需要 15 分钟左右，而功能测试(Function Test)构建类型分别运行于三个工作站（Agent2,Agent3 和 Agent4）上，每次执行约 30 分钟（其中 Agent1 为 Debian 操作系统，Agent2 是 Ubuntu8.04 操作系统，安装有 Firefox2.0，Agent3 是 WindowXP + IE6，而 Agent4 是 Windows 2003 + IE7）。

细心的读者会发现，中央代码库从 Subversion 变成了 Mercurial，我的同事胡凯在《为什么我们要放弃 Subversion》一文中讲述了缘由，其根本收益就是提高了 Cruise 生产效率。构建计划：是根据功能划分的自动构建循环周期，由一系列的步骤组成。下图为三种不同的构建计划：

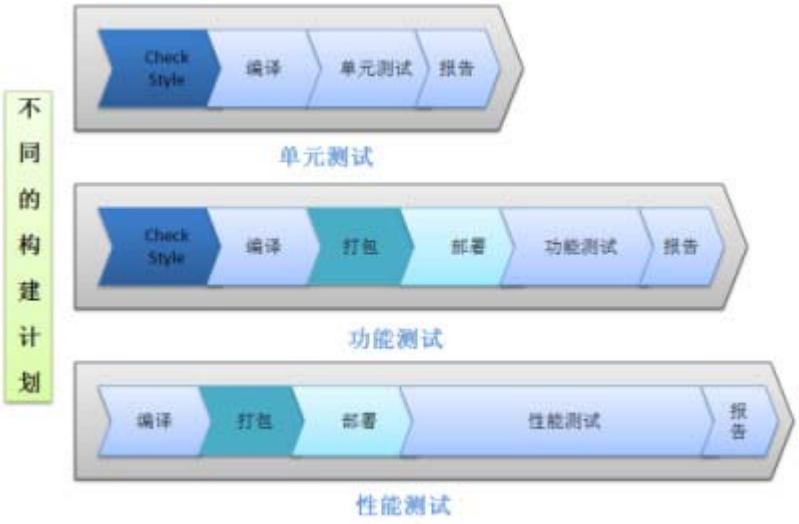


图 4 不同类型的构建计划

利弊分析

益处：暂时解决反馈时间长的问题。

缺点：仍存在浪费，提交频繁时，问题定位有时较困难。

在执行每个构建计划前，都需要从中央服务器上检出代码。不同的构建计划中，有很多步骤都是重复的，例如图 4 所示的编译与打包工作。当代码库比较小时，这并不是一个大问题。然而，对于此时的 Cruise 来说，就已经是“浪费”了。在图 2 中，单元测试 U4 已经以失败而告终，可功能测试还是在另外三台工作站 （Agent）上运行着，一直占用着机器，直至结

束。想像一下，如果构建计划的数量多于工作站（Agent）的个数时，这种资源的浪费又是多么严重呢？

另外，如图 2 所示，由于单元测试运行较快，所以功能测试 F2 中包含多个单元测试（U2,U3,U4）所覆盖的代码变化。如果 F2 失败，很难判断是哪次 Checkin 使 F2 失败的，因为前面已经成功结束的 U2 和 U3 也可能使 F2 失败。

过程化持续集成 —— 消除浪费

实际问题

开始使用阶段式持续集成时，由于不用改太多的构建脚本，而且也初步达到了目的。可是随着时间的推移，代码越来越多，逻辑越来越复杂，问题定位令我们头痛，而且每个构建计划都要重新编译和打包等重复性工作也让团队不爽。

理论基础

过程化则是将每一个步骤单元化，并顺序执行。如第一个单元是 CheckStyle，编译及单元测试，第二个单元是打包生成软件，第三个单元是“部署并进行功能测试”，第三个单元是“性能测试”。这种过程化分解因去除了重复步骤，所以更有效率，而且每次执行时，代码可以顺序经历一系列的测试。

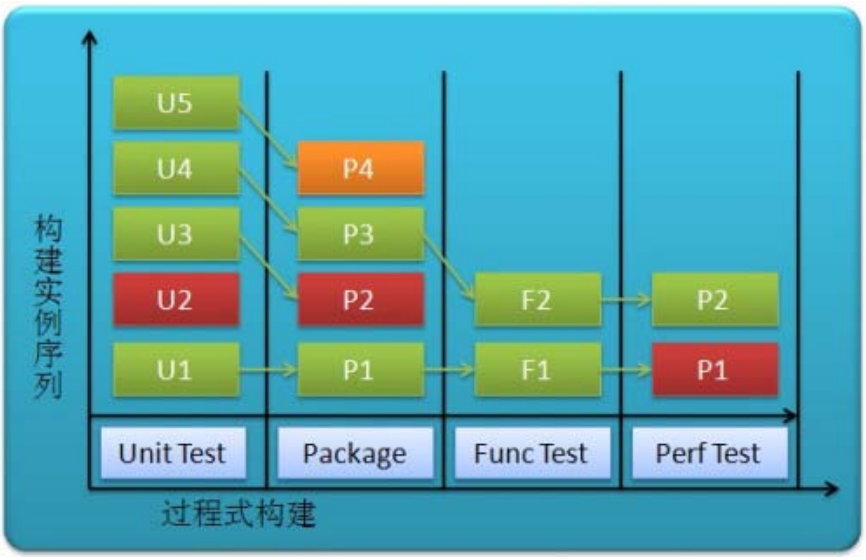


图 5 过程化构建图解

该类型集成的关键在于让所有这些分离的单元可以顺序执行。首先，编译过程被调用，一旦编译单元成功结束，下一单元（快速测试）即开始。正是此刻，我们遇到了第一个复杂性。由于第二个单元（如快速测试）并不是一个完整的过程，所以并不能用它自己产生的产物来测试。而这一点恰恰是过程化持续集成的的关键：将构建与测试分离以节省时间，这也是其与阶段性集成的不同之处。由于后续单元并不产生测试所需要的产物，就要从外部获取，而这些产物正好产生于前面的单元。因此，运行前面单元时，它要将这些产物放在一个已知位置，而后续的单元从这个已知位置拿到产物。图 5 所示，U4 成功后，P3 从 U4 处拿到代码进行打包；P3 结束后，F2 从 P3 处拿到交付物进行部署和功能测试，F2 成功后，P2 再开始运行。

利弊分析

优点：消除和重复工作，提供了持续的信息反馈，反映了整个的过程。

前一个构建计划 U2 失败后，后续的构建计划 P2 也不会从 U2 中拿取产物来运行，而是等待拿 U3 的结果来运行，从而也有效的利用的资源。

缺点：团队自己管理的内容多一些，复杂一些。

由于各个构建计划之间是各自独立的，所以它们之间的依赖关系由它们之间传递的参 数表现。当您想追踪某个后续构建计划为什么失败时，你必须找到这个参数。可这种过程化持续集成并没有提供一个内在功能来完成它，所以你必须自己再多做一点儿工作，而且前后构建单元之间的产物传递也要自己来完成。产物传递看上去要复杂一点儿。因为最近一次构建的结果很容易将前一次的结果覆盖掉。解决这一问题的一种方法是将每次构建的结果分开存放，而后续的单元要知道在哪里可以找到它们所需的文件。利用构建编号做为目录名来保存构建结果时，后续的单元就需要知道这个构建编号，以便找到它所需要的文件。尽管这不难做 到，但还是需要做一点儿工作才能做到的。

另外，由于这种方式需要准备产品的中央存储空间，建立命名规则，修改我们的构建代码，以便在各过程之间传参数。另外，这种模式还是不能从根本解决问题定位这个难题。

团队结论

由于采用这种方式的管理复杂性较高，在我们的环境中，潜在的时间消耗也不少，所以我们放弃了这种集成方式。

管道式持续集成——企业级持续集成的解决方案

实际问题

阶段化持续集成重复任务多，而过程化集成的管理复杂性太高了，任何过程化上的变化都要修改已经写好的脚本，而这些脚本维护比较困难。既然以上两种模式都不灵了，那就再想别的办法吧。

理论基础

管道式持续集成形式上与过程化持续集成相类似，但却在概念上有显著不同。在管道式持续集成中，所有的过程单元都运行在同一管道的上下文中，即各单元所使用的原材料都是完成相同的，即代码基线相同。当持续集成服务器发现有新的代码时，会创建新的一个管道，所有的过程单元都在这一个管道中运行。而每个单元产生 的产物也在该管道中有效。如图所示：

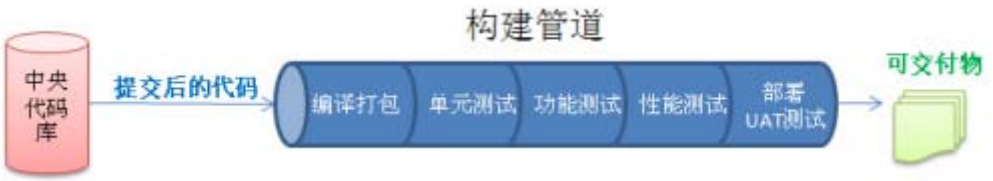


图 6 管道化持续集成

利弊分析

管道式持续集成综合了阶段化与过程化的优点，而带来的复杂性却不用你操心，因为 Cruise 为你管理这一切。

也就是说，你很容易就掌握，哪些产品代码是在哪个管道中生成的，是由哪些原材料（源代码）生成的，而与其它管道产生的产品代码有什么不同。在管道式中，每次构建都会试图从管道的一端走到另一端。因此，你不会遗漏任何一个版本的成功产品代码。

解决方案

Cruise 持续集成类似于下面的持续集成构建管道：

```
<pipeline name="Cruise">
    <stage name="UnitTest">
       <job name = "windows">
 <artifacts>
            <artifact src="testreport"  dest="report"/>
            <artifact src="pkg"  dest="package"/>
          </artifacts>
          <tasks>
            <ant />
          </tasks>
       </job>
       <job name="linux"/>
    </stage>
    <stage name="FuncTest">
       <job name = "windows"/>
       <job name="linux"/>
    </stage>
    <stage name="TwistTest">
       <job name = "windows"/>
       <job name="linux"/>
    </stage>
    <stage name="Package">
       <job name = "Solaris"/>
       <job name="linux"/>
    </stage>
 </pipeline>
```

如图 7 所示，最后一次构建版本（1.2.128）从管道的一头成功走到了另一头，正在进行最后一步（部署到生产环境，黄色方块代表正在运行），版本 1.2.127 失败于打包交付物，虽然我们让版本 1.2.126 走到了管道的尽头，但您可能注意到这是一个有瑕疵的版本（Package 有点问题），而且被很快在 1.2.128 中修复了。而其它版本都有问题而未能走到管道的尽头。

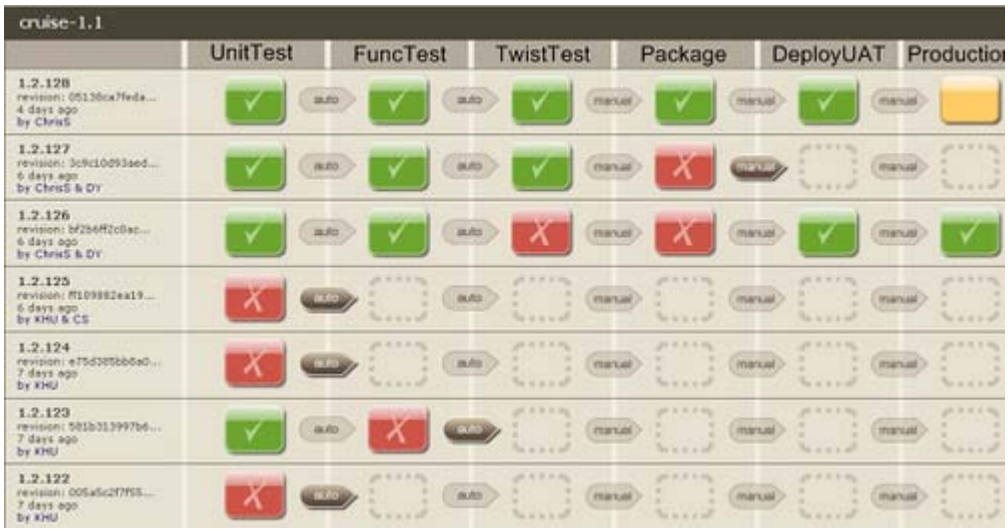


图 7 构建管道图解

并发执行——时间就是金钱，资源也是金钱

实际问题

尽管我们将整个持续集成过程中所需完成的工作根据具体任务分成了很多独立的步骤，并管道化运行，但是并未能解决真正的“时间运行超长”问题。因为，我们的测试代码已经达到相当数量，运行一次所有的单元测试要在 40 分钟以上，而在一台机器上运行所有功能测试大约需要 2 小时以上。而且可以预见到，这一时间量会快速增加。那么，要如何解决这个问题呢？当然不可能因为时间长而不增加测试。

理论基础

将同一类型的测试分成几组，同时运行在配置相同的 Agent 上。这样，虽然不能减少总时间，但是完成所有同类型测试所需要的时间周期会缩短。

解决方案

Cruise 团队成员利用业余时间创建了一个开源项目，名为 Test-Load-Balance， 利用 Cruise 的特性，根据 Cruise 中 Job 的配置，自动将测试分成多组，运行于不同的 Agent 中。当测试运行时间到了无法接受的时候，只需要在 Cruise 的配置文件中增加一个 Job，它就会自动感知这一变化，将测试再多分一份出来。只要有足够的 Agent，那么你的测试时间一定会缩短到 15 分 钟以内。目前，Cruise 的单元测试被 Test-load-Balance 自动等分成四组运行于 Windows，以及五组运行于 Linux，而功能测试也一样被自动分成四组，分别运行于 Windows 和 Linux。目前用于 Cruise 持续集成环境中的工作站共计 11 台虚拟机，使 Cruise 的所有单元测试可以在 15 分钟内完成，所有的功能测试可以在 35 分钟内完成。

我们的持续集成基本结构变为：

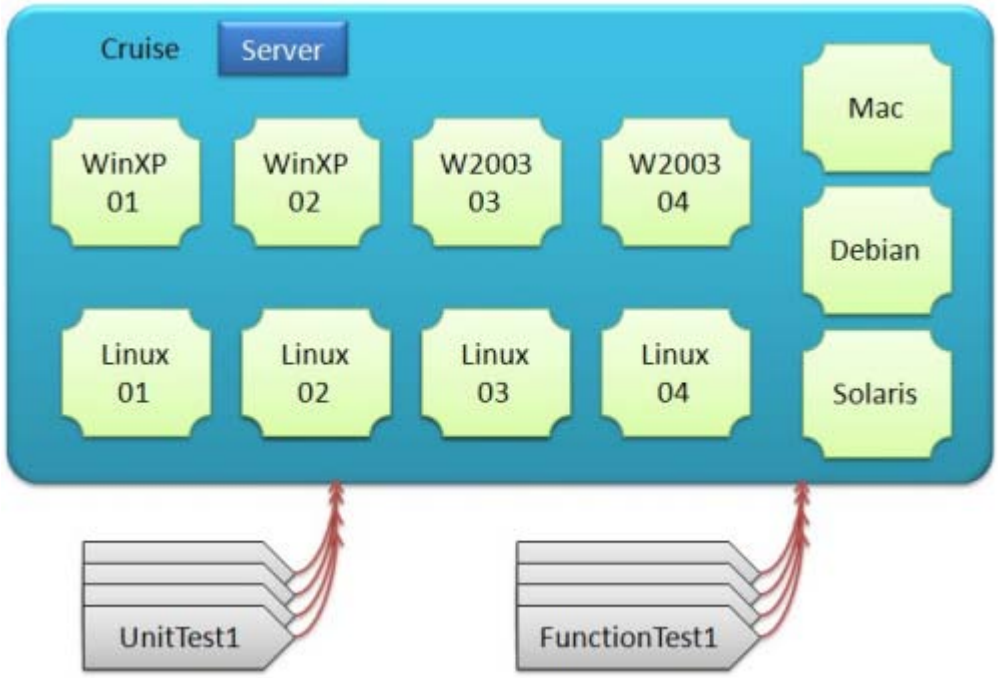


图 8 Build Cloud

Cruise 团队的持续集成类似下面的配置文件：

```
<pipeline name="Cruise">
    <stage name="UnitTest">
       <job name = "windows-01"/>
       <job name = "windows-02"/>
       <job name = "windows-03"/>
       <job name = "windows-04"/>
       <job name="linux-01"/>
       <job name="linux-02"/>
       <job name="linux-03"/>
       <job name="linux-04"/>
    </stage>
    <stage name="FuncTest">
       <job name = "windows"/>
       <job name="linux"/>
    </stage>
    <stage name="TwistTest">
       <job name = "windows-01"/>
       <job name = "windows-02"/>
       <job name = "windows-03"/>
       <job name = "windows-04"/>
       <job name="linux-01"/>
       <job name="linux-02"/>
       <job name="linux-03"/>
       <job name="linux-04"/>
    </stage>
    <stage name="Package">
       <job name = "Solaris"/>
       <job name="linux"/>
    </stage>
```

```
    <stage name="UAT">
       <job name = "deployUAT"/>
    </stage>
    <stage name="Production">
       <job name = "deployProd"/>
    </stage>
</pipeline>
```

利弊分析

我们已无需投入过多的精力在持续集成环境上了，除非我们的硬件不足（可现在的硬件成本要比人工成本低多了）。因为 Cruise Server 自动会将构建计划平均分成多组，并分配到相应的工作站(Agent)上运行。如果测试代码进一步膨胀，我们只需要在相应的测试阶段加增加 Job 的个数，再克隆出几台虚拟机扔到我们的 Agent grid 中。而这仅需很少的人工操作，根本不必修改我们的自动构建代码。

个人持续集成——最大化利用资源

实际问题

测试越来越多，运行时间越来越长。开发人员在本地执行单元测试的时候，开发人员就要看着屏幕等待吗？

解决方案

在 PairStation 上安装一个虚拟机，在其上建立自己的代码仓库。需要运行测试时，把 PairStation 物理机器的代码提交到这个虚拟机的代码仓库中，并让其运行测试。然后，继续在 PairStation 上写代码。如图所示：

利弊分析

益处显而易见，节省时间，提高开发效率。团队（至少个人）要使用分布式版本控制系统。另外，用于开发的物理机性能不能太差。其实，这也算不上大问题，因为在你开发软件时，思考的时间应该多于你写代码的时间。

小结

在敏捷团队中，我们所要做的只不过是：不断的回顾、找出问题与瓶颈、不断地重构。通过不断重构持续集成基础结构以及自动化构建脚本，使其达到我们对“反馈时间”和“判断质量准确性”的要求。

另外，我们已将“持续集成”扩展到整个软件开发周期，涵盖了持续部署及发布。在上面的配置文件中，细心的你会发现最后的两个 Stage 分别名为“UAT”和“Production”，它们一个用于部署新版本到我们自己的持续集成服务器，另一个用于部署新版本到一个公用的持续集成服务器。部署‘UAT’的频率为两天到一周之间，‘Production’的频率为一周。这样，我们可以得到快速反馈，改进自己的产品，同时其它团队可以尽早地使用我们开发的新功能。

持续集成并不是一蹴而就的工作，需要根据团队的实际情况来实施（但这并不能成会“偷赖”的另一个说法）。仅管 Cruise 团队的持续集成尚属“简单”之列，但并不能说明复杂项目无法做持续集成。俗语道，“没有做不到，只有想不到”。只要不断反思与重构，一样可以硕果累累。

注：这些持续集成方式早有出处，例如 2004 年的《通过“产物依赖”实现企业级持续集成》，而 Cruise 团队仅是勇于实践者，并进行着不断的思考和优化。

关于 Cruise

2008 年 7 月 Thoughtworks 的 Studios 部门首次发布一款持续成与发布管理的系统工具（名为 Cruise），该产品将持续集成延伸到了应用的测试、部署与发布阶段。Cruise 可以运行在多种操作系统上（包括 Windows，Mac OS X，和 Linux），并为.NET，Java 和 Ruby 提供了使用上的便捷。 它使复杂软件应用的发布管理变得容易、可靠，为你的团队争取更多的时间，以便集中精力开发新的功能。

作者简介：乔梁，ThoughtWorks 公司资深咨询师及敏捷过程教练；产品 Cruise 的项目经理。

他在 IT 业界有十余年的工作经验，从事过开发、系统管理、培训、项目管理等工作，对于企业从 CMMI 到 Lean/Agile 的转换，业务分析，企业 IT 管 理多有心得。目前致力于获取和传播构建、测试、部署方面的最佳实践，将 Cruise 开发成为持续集成和发布管理的第一商业服务器。

原文链接： http://www.infoq.com/cn/articles/thoughtworks-practice-partv

第七章 Mock 不是测试的银弹 7

胡凯 ThoughtWorks 公司敏捷咨询师

开发者编写高质量测试的征途上可谓布满荆棘，数据库、中间件、不同的文件系统等复杂外部系统的存在，令开发者在编写、运行测试时觉得苦恼异常。由于外部系统常常运行在不同机器上或者本地单独的进程中，开发者很难在测试中操作和控制它们。外部系统以及网络连接的不稳定性（外部系统停止响应或者网络连接超时），将有可能导致测试运行过程随机失败。另外，外部系统缓慢的响应速度（HTTP 访问、启动服务、创建删除文件等），还可能会造成测试运行时间过长、成本过高。种种问题使开发者不断寻找一种更廉价的方式来进行测试，mock 便是开发人员解决上述问题时祭出的法宝。mock 对象运行在本地完全可控环境内，利用 mock 对象模拟被依赖的资源，使开发者可以轻易的创建一个稳定的测试环境。mock 对象本地创建，本地运行的特性更是加快测试的不二法门。

我所在团队设计开发的产品是持续集成服务器，产品特性决定了它需要在各个平台（Windows, Mac, Linux 等）与各种版本管理工具(svn, mercurial,git 等)、构建工具(ant, nant, rake 等)进行集成，对于外部系统的严重依赖让我们在编写测试时遇到了很多困难，我们自然而然的选用了 JMock 作为测试框架，利用它来隔离外部系统对于测试的影响，的的确确在使用 JMock 框架后测试编写起来更容易，运行速度更快，也更稳定，然而出乎意料的是产品质量并没有如我们所预期的随着不断添加的测试而变得愈加健壮，虽然产品代码的单元测试覆盖率超过了 80%，然而在发布前进行全面测试时，常常发现严重的功能缺陷而不得不一轮轮的修复缺陷、回归测试。为什么编写了大量的测试还会频繁出现这些问题呢？ 在讨论之前先来看一个真实的例子：

我们的产品需要与 Perforce(一种版本管理工具)进行集成，检测某段时间内 Perforce 服务器上是否存在更新，如果有，将更新解析为 Modification 对象。将这个需求反应在代码中，便是首先通过 Perforce 对象检测服务器更新，然后将标准输出(stdout)进行解析：

```
public class PerforceMaterial {
    private Perforce perforce;
    .....
    .....
    public List findModifications(Date start, Date end) {
        String changes = perforce.changes(start, end); //检测更新，返回命令行标准
```

输出（stdout）

```
        List modifications = parseChanges(changes);//将标准输出解析为Modification
        return modifications;
    }

    private List parseChanges(String output) {
         //通过正则表达式将stdout解析为Modification对象
    }
}

public class Perforce {
    public String changes(Date start, Date end) {
          //通过命令行检测更新，将命令行标准输出（stdout）返回
    }
}
```

相应的 mock 测试也非常容易理解：

```
.....
@Test
public void shouldCreateModifiationWhenChangesAreFound() {
    final String stdout = "Chang 4 on 2008/09/01 by p4user@Dev01 'Added
build.xml'"; //设置标准输出样本
    final Date start = new Date();
    final Date end = new Date();
    context.checking(new Expectations() {{
       one(perforce).changes(start, end);
       will(returnValue(stdout));
    }});//设置perforce对象的行为，令其返回设定好的stdout
    //调用被测方法
    List list = perforceMaterial.findModifications(start, end);

    assertThat(list.get(0).revision(), is("4"));
    assertThat(list.get(0).user(), is("p4user@Dev01"));
    assertThat(list.get(0).modifiedTime(), is("2008/09/01"));
}
```

测试中的 stdout 是在真实环境下运行 Perforce 命令行所采集的标准输出(stdout)样本， 通过 mock perforce 对象，我们可以轻易的控制 changes 方法的返回值，让验证解析逻辑的正确性变得非常容易，采用 mock 技术使开发者无需顾忌 Perforce 服务器的存在与否，而且可以采用不同的 stdout 来覆盖不同的情况。然而危机就在这看似完美的测试过程中被埋下了，事实上 Perforce stdout 中的时间格式会依用户环境的设定而变化，从而进一步导致 parseChanges 方法中的解析逻辑出现异常。由于测试中的 stdout 全由假设得来，并不会依照环境变化，即便我们将测试跑在多种不同的环境中也没能发现问题，最终在产品环境才由客户发现并报告了这个缺陷。

真实 perforce 对象的行为与测试所使用的 mock 对象行为不一致是出现上述问题的根本原因，被模拟对象的行为与真实对象的行为必须完全一致称之为 **mock 对象的行为依赖风险。**

开发者对 API 的了解不够、被模拟对象的行为发生变化（重构、添加新功能等修改等都可能引起被被模拟对象的行为变化）都可能导致错误假设（与真实对象行为不一致），错误假设会悄无声息的引入缺陷并留下非法测试。非法测试在这里所代表的含义是，它看起来很像测试，它运行起来很像测试，它几乎没有价值，它几乎不会失败。在开发中，规避行为依赖风险最常见的方法是编写功能测试，由于在进行 mock 测试时，开发者在层与层之间不断做出假设，而端到端的功能测试由于贯穿了所有层，可以验证开发者是否做出了正确的假设，然而由于功能测试编写复杂、运行速度慢、维护难度高，大部分产品的功能测试都非常有限。那些通过 mock 测试的逻辑，便如埋下的一颗颗定时炸弹，如何能叫人安心的发布产品呢？

《UNIX 编程艺术》中有一句话“先求运行，再求正确，最后求快”，正确运行的测试是高质量、可以快速运行测试的基础，离开了正确性，速度和隔离性都是无根之木，无源之水。那么采用真实环境就意味着必须承受脆弱而缓慢的测试么？经历了一段时间的摸索，这个问题的答案渐渐清晰起来了，真实环境的测试之所以痛苦，很大程度上是由于我们在多进程、多线程的环境下对编写测试没有经验，不了解如何合理的使用资源（所谓的资源可能是文件、数据库中的记录、也可能是一个 新的进程等），对于我们，mock 测试作为“银弹”的作用更多的体现在通过屏蔽运行在单独进程或者线程中的资源，将测试简化为对大脑友好的单线程运行环境。在修复过足够多的脆弱测试后，我们发现了编写健壮测试的秘密：

要设计合理的等待策略来保守的使用外部系统。很多情况下，外部系统处于某种特定的状态是测试得以通过的条件，譬如 HTTP 服务必须启动完毕，某个文件必须存在等。在编写测试时，开发者常常对外部系统的估计过于乐观，认为外部系统可以迅速处于就绪状态，而运行时由于机器和环境的差异，结果往往不如开发者所愿，为了确保测试的稳定性，一定要设计合理的等待策略保证外部系统处于所需状态，之所以使用"等待策略"这个词，是因为最常见”保证外部系统处于所需状态“的方法是万恶的"Thread.sleep"，当测试运行在运算速度/网络连接速度差异较大的机器上时，它会引起随机失败。而比较合理的方法是利用轮询的方式查看外部系统是否处于所需状态（譬如某个文 件存在、端口打开等），只有当状态满足时，才运行测试或者进行 Assertion，为了避免进入无限等待的状态，还应该设计合理的 timeout 策略，帮助确定测试失败的原因。

要正确的创建和销毁资源漠视测试环境的清理也常常是产生脆弱测试的原因，它主要表现在测试之间互相影响，测试只有按照某种顺序运行时才会成功/失败，这种问题一旦出现会变的非常棘手，开发者必须逐一对有嫌疑的测试运行并分析。因此，有必要在开始时就处理好资源的创建和销毁，使用资源时应当本着这样一个原则：谁创建，谁销毁。 junit 在环境清理方面所提供的支持有它的局限性，下面的代码是使用资源最普遍的方式：

```
@After
```

```
public void teardown() {
   //销毁资源A
   //销毁资源B
}

@Test
public void test1() {
   //创建资源A
}

@Test
public void test2() {
   //创建资源B
}
```

为了确保资源 A 与资源 B 被正确销毁，开发者必须将销毁资源的逻辑写在 teardown 方法中，然而运行用例 test1 时，资源 B 并未被创建，所以必须在 teardown 中同时处理资源 A 或 B 没有被创建的情况，由于需要销毁的资源是用例中所使用资源的并集，teardown 方法会快速得膨胀。由于这样的原因，我在开源项目 junit-ext 中加入了对 Precondition 的支持，在测试用例运行前，其利用标注所声明的多个 Precondition 的 setup 方法会被逐一调用来创建资源，而测试结束时则调用 teardown 方法销毁资源。

```
@Preconditions({ResourceIsCreated.class, ServiceIsStarted.class})
@Test
public void test1() {
     //在测试中使用资源
}

public class ResourceIsCreated implements Precondition {
    public void setup() {
          //创建资源
    }
    public void teardown() {
          //回收资源
    }
}

public class ServiceIsStarted implements Precondition {
     public void setup() {
          //创建资源
     }
     public void teardown() {
          //回收资源
     }
}

public interface Precondition {
     void setup();
```

```
    void teardown();
}
```

这个框架可以更好的规范资源的创建和销毁的过程，减少因为测试环境可能引起的随机失败，当然这个框架也有其局限性，在 ResourceIsCreated 和 ServiceIsStarted 之间共享状态会比较复杂，在我们的产品中，Precondition 大多用于启动新进程，对于共享状态的要求比较低， 这样一套机制就非常适合。每个项目都有其特殊性，面对的困难和解决方案也不尽相同，但在使用资源时如果能遵守“谁创建，谁销毁”的原则，将会大大减小测试 之间的依赖性，减少脆弱的测试。

要设计合理的过滤策略来忽略某些测试。我们很容易在项目中发现只能在特定环境下通过的测试，这个特定环境可能是特定的操作系统，也可能是特 定的浏览器等，之所以会产生这些测试通常是开发者需要在源码中进行一些特定环境的 hack，它们并不适合在所有环境下运行 ,也无法在所有环境中稳定的通过 ,因此应该设计一套机制可以有选择的运行这些测试，junit 的 assumeThat 的机制让我再次有点失望，本着自己动手丰衣足食的原则，在 junit-ext 我添加了利用标注来过滤测试的机制，标注了 RunIf 的测试仅当条件满足时才会运行，除了内置一些 Checker，开发者也可以很方便的开发自己的 Checker 来适应项目的需要。

```
  @RunWith(JunitExtRunner.class)
  public class Tests {
    @Test
    @RunIf(PlatformIsWindows.class) //test1仅运行在Windows环境下
    public void test1() {
    }
}

public class PlatformIsWindows implements Checker {
    public boolean satisfy() {
        //检测是否WINDOWS平台
    }
}
```

要充分利用计算资源而不是人力资源来加快测试。对于加快测试，最普遍也最脆弱的方法是利用多线程来同时运行多个测试，这个方法之所以脆弱， 是因为它会让编写测试/分析失败测试变的异常复杂，开发者必须考虑到当前线程在使用资源时，可能有另一个线程正要销毁同一个资源，而测试失败时，也会由于线程的不确定性，导致问题难于重现而增加了解决问题的成本。我认为一个更好的实践是在多台机器上并发运行测试，每台机器只需要运行(总测试数/机器数)个 测试，这样所花费时间会近似减少为(原本测试时间/机器数)。相对与购置机器的一次性投入，手工优化的不断投入成本更高，而且很多公司都会有闲置的计算资源，利用旧机器或者在多核的机器上安装虚拟机的方式，可以很经济的增加计算资源。在项目开发的业余时间，我和我的同事们一起开发了开源的测试辅助工具 test-load-balancer。在

我们的项目中 ,通过它将需要 90 分钟的测试自动划分为数个 10 分钟左右的测试在多台虚拟机上并发运行，很好的解决了速度问题。

对 mock 追本溯源，我们会发现它更多扮演的是设计工具的角色而不是测试工具的角色，mock 框架在设计方面的局限性李晓在《不要把 Mock 当作你的设计利器》 一文中已经谈的很透彻，在此不再赘述。Mock 不是测试的银弹，世上也并无银弹存在，测试实践能否正常开展的决定性因素是团队成员对测试流程，测试方法的不断改进而不是先进的测试框架。不要去依赖 mock 框架，它的强制约定常常是你改进设计和添加功能的绊脚石，改善设计，依赖一个简洁的代码环境，依赖一套可靠的测试方法才是正途。从意识到 mock 测试带来的负面影响，到从滥用 mock 的泥潭中挣扎出来，我们花费了很多时间和经历，希望这些经验可以对同行们能有所借鉴，有所启发。

Mock，还请慎用。

写在最后

在 ThoughtWorks 工作的三年经历中，对于 mock 框架的使用从无到有，到滥用到谨慎。三年间，和李晓、陶文，李彦辉，Chris Stevenson 等人反复讨论和辩论使我对 mock 有了更多的理解。这篇文章反反复复改了许多稿，在此对陈金洲和李默的耐心帮助致谢。最后，没有与 InfoQ 的编辑李剑和霍泰稳在 Twitter 上的一番谈话，也就没有这篇文章。

参考

Mock Roles, not Objects， Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes

不要把 Mock 当作你的设计利器，李晓

作者简介：胡凯，ThoughtWorks 公司的敏捷咨询师，近 2 年一直从事持续集成工具 Cruise 以及 CruiseControl 的设计开发工作。创造和参与了开源测试框架 junit-ext 和 test-load-balancer，对于 Web 开发，敏捷实践，开源软件与社区活动有浓厚的兴趣，可以访问他的个人博客进行更多的了解。

原文链接： http://www.infoq.com/cn/articles/thoughtworks-practice-partvi

第八章 环境无关的环境 8

李光磊 ThoughtWorks 公司软件工程师

软件开发过程中常常需要搭建各种环境：开发环境、测试环境，集成构建环境等等。一个不可复制的环境是低效的根源，它引起的常见问题比如：

1. 产品只能在你的机器上编译通过
2. 产品在你机器上运行正常，可在测试环境中总是出错
3. 新加入一个项目成员，需要一天时间来为其建立开发环境
4. 把测试环境和集成环境迁移到另外一台服务器上花了几天时间

这些问题的原因以及解决方案，在最新出版《The Productive Programmer》（卓有成效的程序员）中，Neal Ford 给出了详细的介绍。我们列举几种细化的方案，作为书中提到的“间接”、“规范性”等原则的实践。

试图解决的问题：环境的各个部分散落在不同角落，不是少了这个就是少了那个，或不同机器上版本不一样；环境配置依赖全局环境变量或属性，硬编码的绝对路径等。

目标：机器环境虽然各有各的不同，但依然有可能创建一个“环境无关的环境”。

1. 使用相对路径代替绝对路径

关键是如何获得当前路径，如何确定根路径，如何确保目录结构。

获得当前路径

1. Windows 和 Unix 都有内置的环境变量来表示当前路径，分别是%cd%和$PWD
2. Windows 批处理脚本中，还可以使用%~dp0%获得脚本所在路径
3. 而在 Unix Bash 脚本中，则可以使用“pwd”，即获得 pwd 命令的输出

4. Makefile 中，也可以获得 shell 命令的输出，比如 this_dir = $(shell pwd)
5. Ant 脚本中，内置的 basedir 属性缺省代表的是脚本文件所在的路径

%cd%和%~dp0%的区别:

1. %cd%是脚本运行时的当前工作路径，与脚本所在位置无关
2. %~dp0%则相反，是脚本所在路径，与运行时的工作路径无关

举例来说，有如下内容的 D:\project\run.bat：

```
echo %cd%
echo %~dp0%
```

一般来说%~dp0%比%cd%更常用。

下面是一个环境无关的 makefile 的例子（只列出了变量定义部分）:

```
PROJ_ROOT=$(shell pwd)
INCLUDE += $(PROJ_ROOT)/include
LIB += $(PROJ_ROOT)/lib
```

当然也可以根本不定义变量表示当前路径，而直接以相对路径的形式引用子目录，和通过“..”来引用父目录及兄弟目录，然而显式的变量定义提供了一层间接，你可以通过多种方式覆盖它的缺省值，从而适应不同的环境。参见后面的“缺省值 + 用户自定义属性”。

确定根目录

如果总是需要引用某个根路径的话，则可以使用环境变量来定义根路径（参见后面的环境变量）。其实相对路径配合固定的目录结构，大大削减了对显式定义根路径的需求。

前面 makefile 的例子可以改写如下

```
ifeq "$(origin PROJ_ROOT)" "undefined"
PROJ_ROOT                                                                    =
You_should_firstly_specify_$${PROJ_ROOT}_in_your_environment_or_by_command_
line
endif

INCLUDE += $(PROJ_ROOT)/include
LIB += $(PROJ_ROOT)/lib
```

保证目录结构的固定，自然是使用配置管理系统。

2. 使用配置管理系统(版本控制系统)

这是一个“规范性”或者“标准化”的问题。配置管理系统不只是放源代码的，只要有配置管理需求或配置管理能带来好处，或需要有唯一的官方来源，都可以使用配置管理系统来管理；环境的配置文件，环境本身，都可以置入配置管理之下。有些公司的版本控制系统只放源代码，连测试代码都分开另放，耗费很多精力来维护源代码之外的文档。

1. 强制使用配置管理可解决固定的目录结构的问题
2. 使用配置管理还可以解决丢三落四的问题
3. 自然也可以解决所用工具版本不一致的问题

这里有几个常见的问题：

1. 大型的系统软件如何处理，比如 JDK、VC++编译器等，是否也需要置入版本控制：这个基本不用，如果对其特定的版本有需求，可在脚本中加入检查其版本的逻辑，不满足则提示并退出即可
2. 依赖的大量二进制库如何处理：如果有必要可使用依赖管理工具如 Maven，Ivy 等，而用于存放依赖的本地仓库依然应该置入配置管理，哪怕不用其版本控制功能，而只是利用其官方来源/备份存档等好处
3. 必须放在特定位置的文件如何处理，比如某个文件必须放在/etc 目录下：这个文件还是可以放在配置管理下的目录中，而在/etc 下创建符号链接来指向它；并提供脚本来干这件事。

3. 环境变量

必要时使用环境变量来引用环境。全局的环境变量可用作缺省值，在脚本中覆盖它(基本上，这是“用户自定义属性”的一个实例)。

4. 缺省值 + 用户自定义属性

这是创建“环境无关的环境”的核心机制。无论如何，环境要在不同机器上部署，总会需要修改某些配置，以适应宿主机器。然而前面我们提到，所有配置都已置入版本控制。如果我们直接修改，则每个环境中都会存在未提交的本地修改。这是我们不希望看到的，因为当我

们升级配置并更新到所有部署时，可能会产生冲突。这里其实是两个层面的问题：

1. 提供一种机制，当环境与缺省配置不一致时，允许用户修改
2. 用户修改的文件应避免与官方文件更新的冲突

先说第一个。

缺省值当然可以直接定义在脚本或配置文件中。而多数常用的脚本和配置系统都提供了用户定义属性覆盖缺省值的机制。比如：

1. Windows 批处理：set path=my_extra_path;%path%
2. Bash：export PATH=my_extra_path:$PATH
3. Ant：

   <property file="user.properties"/> <!-- user.properties 中可定义任何后面引用到的属性，以覆盖其缺省值-->

   <property name="src.dir" path="${basedir}" /> <!-- 定义"src.dir"的缺省值-->
4. CruiseControl：

   <property name="src.dir" value="." /> <!-- 定义"src.dir"的缺省值-->

   <property file="user.properties"/> <!-- user.properties 中可定义任何后面引用到的属性，以覆盖其缺省值-->
5. 注意 Ant 的属性是只读的，先入为主。CruiseControl 的属性则是后发制人。
6. Makefile 则可以直接在命令行覆盖文件里面定义的缺省属性。如覆盖前面例子中的 PROJ_ROOT：

   make PROJ_ROOT=/home/mike/project

再说第二个。

有一个很简单的解决办法，就是把用户自定义属性置入单独的文件，并且不要把它提交到版本控制系统中（一个理由是这一部分相对整个组织来说，不存在也不需要唯一的官方来源）

前面例子中的 user.properties 就是一个用户自定义属性文件，只存在每个用户自己的机器上，不在配置库中。在 Windows 和 Bash 脚本中也可以类似处理：

1. Windows: call user_env.bat

2. Bash: source ./user_env.sh 或 . ./user_env.sh

随之而来的一个问题是，这个用户相关的文件应该放在何处。这里其实约定好就可以了，比如当前路径，根路径，甚至 user 的 home 路径都可以。

参考借鉴

1. 至此，这套东西在不同的机器上部署时，只要从版本控制系统中 check out 出来即可使用了。更进一步，还可以提供脚本，即生成器，自动探测用户的环境，来生成全套配置，类似 Rails 生成应用框架那样。
2. 其实，这是一个规范性或标准性的问题。Neal Ford 在《卓有成效的程序员》中，用一章的篇幅详述了各种解决方案，包括配置管理，符号链接等。除此之外，他还建议可以/应该使用虚拟机来统一项目组的开发环境等。参见《卓有成效的程序员》第五章。
3. 另请参阅《CruiseControl Enterprise 最佳实践 (3)：Configuring CruiseControl the CruiseControl way》，是创建环境无关的持续集成环境的实例。

作者简介：李光磊，软件工程师，同时还是一位敏捷教练，就职于 ThoughtWorks。他还是活跃的 blog 作者，了解他最新的想法，请访问 http://blog.csdn.net/chelsea。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-part7

第九章 Tech Lead 的三重人格 9

熊节 ThoughtWorks 公司高级咨询师

很多团队都有 tech lead 这个角色的存在，但同时很多团队对这个角色都缺乏明确的定义。大多数时候，团队只是指派其中经验最丰富、技术最精熟的开发者来担当 tech lead。但除了“tech”的成分之外，这个角色还有“lead”的成分，这就决定了他不仅需要技术上的能力，还要眼观六路耳听八方，才能带领团队—— 至少是开发者们——取得成功。

Tech lead 需要关注的事情可谓纷繁芜杂。把这些事情分门别类，我们可以看到，这个角色大致有三方面的职责：技术决策者、流程监督人、干扰过滤器。

技术决策者

从技术的角度，tech lead 需要关注以下三个方面：架构设计、局部设计和关键技术点。

参与架构设计

系统的架构通常是在项目初期的 quickstart 阶段设计出来的。这个架构设计包含的都是高层面的内容，例如 C/S 或 B/S 的选择、开发平台、编程语言、数据迁移策略、集成点、部署结构等。

尽管架构师（architect）是架构设计的主要负责人，tech lead 还是会参与设计过程，以获得对系统全局的清晰了解，并且尽早发现架构设计中不合理或者风险较高的部分。

主要关注点：

- 选择什么平台和编程语言？
- 需要和什么系统集成？需要使用哪些第三方软件或工具？

  给自己画一张集成架构图：先用虚线圈出为用户提供服务所需的整个系统范围，再用实线圈出将用主要编程工具（例如 J2EE 或者 Ruby on Rails）开发的应用程序，两者之间的

其他东西就是你需要集成的目标。找出集成点和集成方式。留住这张图并保持更新，你会经常用到它。

- 如何部署？如何迁移现有数据？

  开发团队日常部署到什么环境？UAT（用户验收测试）和性能测试的环境由谁部署？生产环境由谁部署？部署的频度如何？开发和测试使用的数据集从何而来？数据迁移是部署过程的一部分吗？除了自动化部署脚本之外还需要什么额外的环节吗？

- 团队的技能是否与项目需求匹配？

攻克技术难题

作为“tech” lead，解决技术上的难题、为团队扫清前进道路，自然是题中应有之义。由于具备对系统全局架构的了解，tech lead 能够识别出项目中可能遇到的技术挑战，及时进行研究和实验，尽力使其不对开发工作造成阻碍。

主要关注点：

- 集成点是否得到妥善处理？

  能成功集成吗？用于集成测试的环境到位了吗？出错的情况妥善处理了吗？

- 如果技术栈中的某一部分出现问题怎么办？

  项目中用到的第三方组件有谁熟悉？如果是开源软件，相关的社群在哪里？如果是闭源软件，我们能得到其生产厂商的技术支持吗？

确定设计方案

敏捷项目中，大部分设计都是在项目过程中做出的，其中有些设计会对系统整体产生较大的影响，有时甚至会改变起初的架构设计。所以贯穿整个项目，tech lead 需要保持对系统整体和细部的把握，选择适当的设计方案，或是通过重构让好的设计浮现出来。

主要关注点：

- 代码中是否出现明显的 bad smell？

  是否看到明显的大类、长方法或者重复代码？条件逻辑嵌套太深了吗？本应属于模型的逻辑在视图或者控制器里堆积了吗？是否应该考虑做一做对象健身操？（参见

《ThoughtWorks 文集》，第 6 章）

- 局部设计是否会对系统造成明显的损害？

  这个设计会严重损害性能吗？它是一个安全漏洞吗？它是难以测试的吗？它损害了系统遵循的概念一致性吗？

- 团队是否正在进行关于设计和重构的讨论？

要在局部设计上具备发言权，tech lead 就不能脱离开发工作——不写代码的人是无权评价代码的。在讨论细部设计时，tech lead 需要把握一个微妙的平衡：既不要过于民主而耗费太多时间，也不要过于集中而使团队成员失去学习和思考的机会。一个好的实践是：每天早上把所有开发者召集起来，用 15 ~ 20 分钟浏览前一天编写的所有代码，这样每个人都有机会在看到 bad smell 时指出。

流程监督人

为了保障开发工作顺畅进行，tech lead 需要从以下三方面入手来保持对开发流程的关注：开发环境、持续集成和测试。

开发环境

Tech lead 需要保障良好的开发实践得到贯彻，尤其是当团队中有较多缺乏经验的成员时这一点就更显重要。一家公司内部很可能有一些成熟的开发套路，这也使得每个 tech lead 的责任更重大：如果团队成员在一个项目中没有养成好习惯，受损害的不仅是这一个项目，还有整个公司的习惯套路。

主要关注点：

- 如何使开发团队拥有统一的环境？

  IDE 的设置和快捷键如何同步？shell 设置和别名如何同步？数据库设置如何保持一致？需要将开发环境虚拟化吗？

- 如何实现一段代码并提交？

  开发者从哪里获取验收条件？编写代码时应该遵循哪些惯例？功能完成之后是否需要邀请 BA/QA 快速确认？提交代码之前应该如何进行本地构建和测试？提交时的注释格式有什么要求？

ThoughtWorks 的很多 Ruby on Rails 项目都采用“rake commit”这个任务来提交代码。这个任务会从 svn 服务器更新代码、执行本地构建、运行测试、要求开发者输入符合格式的注释内容、然后提交修改内容。我们通常还会给这个命令指定一个 shell 别名“rc”，这样我们只需要敲三下键盘就可以开始一次标准的提交。

- 如何结对编程？

  哪些任务需要结对？哪些任务可以不必结对？结对的轮换合理吗？是否有谁在某一个任务上做了太长时间？是否有谁对某一部分完全不了解？结对过程中大家都全心投入了吗？是否需要用特别的结对方法来引导经验较少的团队成员？需要开发者和 QA 结对吗？

  Tech lead 同样需要与团队成员结对，有时是为了理解某一部分的实现，有时是为了指导和帮助队友。Tech lead 应该始终牢记自己是开发团队的一份子，每天都应该尽量安排出时间参与结对；同时其他开发者也应该谅解 tech lead 还有其他职责，不要因此拒绝和他结对。

持续集成

每个人都需要对持续集成负责，有一个人需要负更多的责，这个人就是 tech lead：他要清楚持续集成中每个阶段的用意，当集成失败时他要知道这意味着什么。关于持续集成的设计，我强烈推荐 Dave Farley 的文章“一键发布”（《ThoughtWorks》文集，第 12 章）。

主要关注点：

- 持续集成环境和生产环境有多大差异？
- 持续集成覆盖哪些阶段？

  项目各方的关注点在持续集成中都有体现吗？性能测试被覆盖到了吗？打包部署呢？拿到持续集成产出的安装包，我们一定有信心将它部署到生产环境吗？

- 持续集成给开发团队快速而有用的反馈了吗？

  经常失败的是哪些阶段？随机失败的概率大吗？随机失败会掩盖真正的问题吗？从提交代码到走完所有集成阶段通常需要多长时间？需要使用更大规模的并行集成吗？（例如引入更多的 Cruise Agent。）

测试

测试就是开发者要满足的目标。没有良好的测试，就等于没有良好的目标。作为 tech lead，要关注不仅是单元测试，还包括功能测试和各种非功能性需求的测试；不仅是开发者编写的测试，还包括 QA 乃至客户的测试。当然，从技术的角度出发，tech lead 更关注的还是自动化的测试。

主要关注点：

- 团队如何实施 TDD？

  测试的粒度和层面合理吗？是否有适当的系统测试？是否有滥用系统测试取代单元测试的倾向？修复 bug 时先用测试来描述 bug 了吗？集成点都有测试覆盖吗？数据迁移都有测试覆盖吗？

- 功能测试/验收测试的质量和进度如何？

  QA 和开发者如何借助功能测试沟通？如果功能测试由 QA 编写，能赶上开发的进度吗？测试代码的质量如何？如果由开发者编写，测试覆盖到所有验收条件了吗？QA 能否理解和维护测试代码？是否需要安排 QA 和开发者结对编写功能测试？

- 性能测试得到足够关注了吗？

  有清晰的性能需求吗？有性能测试描述这些需求吗？如何得到性能测试的结果？

我的同事 James Bull 在“实用主义的性能测试”（《ThoughtWorks 文集》，第 14 章）一文中对性能测试的做法有精彩的论述，此处不再赘述。

干扰过滤器

在大部分软件项目中，开发者的工作——细部设计和编程实现——都位于关键路径上：它们未必是最有价值的工作（尽管我个人坚持这样认为），但它们一定是最耗时间的工作。换句话说，开发者的时间是否充分用于开发，将决定项目能否按时交付。所以，tech lead 的很大一部分责任就是过滤各种干扰，使开发者们全神贯注地编程。尽管项目经理和 BA 也起到这样的作用，但还是有很多编程之外的事需要“技术人员”来做。

与客户技术团队沟通

大多数定制开发项目都会涉及到客户方的技术团队：开发、测试、DBA、运维、支持，等等。光把系统做好还不够，你还得把做好的系统交到他们手上，项目才真算完成——“交到他们手上”这件事就得由 tech lead 来负责。

主要关注点：

- 如何与客户的开发团队交换知识？

  有哪些惯例需要遵循？有哪些既有的工具可以利用？如果双方开发者同时开发，如何协作？如何结对编程？如果只需要在开发结束后移交产品，如何做知识传递？如何确保对方保持关注？如果双方不在同一地点，是否需要安排定期的电话会议或视频会议？

- 如何与客户的支持团队协作？

  生产或 UAT 环境的服务器由谁管理？如何交付部署包？客户的测试人员如何进行测试？测试结果如何获取？

  运维团队在软件项目中常常被忽视，但他们对产品的成功上线至关重要。通过搭建和维护 UAT/性能测试环境，可以尽早地让运维团队参与到项目之中，并且了解部署和日常管理的相关信息，从而使最终的上线变得相对容易。

- 如何与客户的 DBA 协作？

  数据库迁移计划经过 DBA 复审了吗？DBA 是否能获得系统运行时的数据库访问日志？如何与 DBA 讨论解决数据库相关的问题？

  一个好的实践是在每次部署到 UAT 环境之前将近期的数据库改变总结出来告知 DBA，并请他持续关注 UAT 环境被使用时的数据库日志。如果 DBA 指出一些明显性能低下或者有其他问题的 SQL，应该重视他的意见。

与 BA/QA 协作

团队内部的“干扰源”主要是 BA 和 QA。BA 和 QA 往往缺乏技术背景，如果他们经常用一些“愚蠢”的问题去打断开发者的工作，开发者们可能会觉得他们添乱多过帮忙。这时 tech lead 就得表现出更多的耐心，先过滤掉那些没有营养的问题，从而让开发者们觉得与 BA/QA 的沟通是有帮助的。

主要关注点：

- Story 的内容和工作量估计合理吗？

  Story 涉及的功能实现起来有多困难？是否有更简单的方式来实现同样的目标？相关的风险大吗？

  理论上，工作量估算是由开发者来做的。但有两种原因使得 tech lead 需要代表开发者来做这件事：（1）找开发者做估算可能打断他们的工作节奏；（2）项目初期可能其他开发者不了解情况，甚至还没有加入项目。同样，如何尽量让所有开发者都感到自己的意见得到尊重，又不过多占用他们的时间，这也是需要平衡的。而代表开发者做估算的准确度也将直接影响 tech lead 在他们心里的地位。

- QA 的工作需要帮助吗？

  QA 发现 bug 时会如何处理？他发现的 bug 经常是“误报”吗？需要帮助他编写自动化测试吗？需要帮助他做性能测试吗？

打杂

其他所有需要由“技术人员”来做的事，tech lead 都应该有自己一肩挑的准备——例如查一下数据库里有哪些不符合业务规则的数据，生成一份 CSV 文件，小小改动一下界面，甚至倾听一下客户和项目经理的抱怨再给他们一点“技术性”的安慰，等等。

但这不表示你就总是应该把这些事一肩挑。Tech lead 这个角色的微妙之处就在于：他仍然是“技术人员”，和所有的开发者一样。换句话说，tech lead 能做的事，其他开发者也应该能做。所以，再一次地，你应该有一个平衡：把一部分杂事自己消化掉，不让它们干扰开发者的正常工作；另一部分杂事则分配给开发者去做，让他们感到自己除了编程之外还参与了项目的其他方面，同时也给自己挤出一点时间做其他更重要的事。

小结

在我所经历过的大部分项目里，tech lead 都是个超级忙的家伙——看这篇文章就不难理解为什么。人们期望 tech lead 做的事很多，而且在项目的各个阶段还有所不同，一个人要肩负这样的期望确实不容易。

不过只要意识到“tech lead”只是一顶很多开发者都可以戴的帽子，随着项目的进展，你就

可以慢慢地把更多的责任放在整个团队的肩上——只要不至于造成损害，于是自己也有了更多的时间来编程。最终你可能会得到一支这样的团队：其中每个开发者相当平均地扮演一部分 tech lead 的角色，各种任务随优先级被有效地处理。这时你就可以说，这个项目的 tech lead 确实做好了他的工作。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-part8

第十章 自动化测试的分层结构 10

李贝 ThoughtWorks 公司咨询师

摘要

在测试自动化中，测试代码中不仅仅包含测试逻辑，还包含许多其他代码，比如 URL 拼接、html/xml 解析、访问 UI 控件，等等。若把测试逻辑与这些无关代码混在一起，测试逻辑将会很难理解， 也不容易维护。本文会介绍如何用分层结构来解决测试自动化中遇到的这些问题。在这个分层结构中，测试自动化代码会被分成三层：（1）测试用例层，表达应用程序的测试逻辑。（2）领域层， 用业务领域术语来给待测系统建模，封装 HTTP 请求、浏览器控制、结果解析逻辑等，给测试用例层提供一个接口。（3）待测系统层，第 2 层构建在这一层之上。

问题

QA 的工作包括设计测试用例、探索性测试（exploratory testing）及回归测试，等等。这些工作有的依靠 QA 的聪明才智，而有的却只是重复劳动（例如回归测试）。随着系统中不断地加入新功能，回归测试这类工作耗费的时间也越来越多。

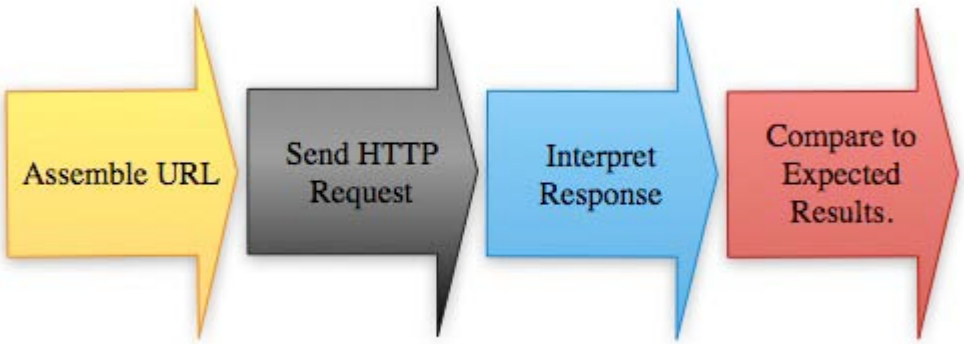
测试自动化可以解决这个问题。测试自动化后，重复性的劳动会由计算机来做，而测试用例都用计算机程序来表述， 因此 QA 可以从重复劳动中解脱出来，有更多的时间用在创造性的工作上来。

在测试自动化中，测试代码并不仅仅包含测试逻辑，也包含许多其他的支撑代码，例如 URL 拼接、HTML/XML 解析、UI 控件访问等。例如要测试一个能接受不同搜索参数，并返回包含特定信息的 XML（例如用户数据）的 web 服务，测试代码需要：

1. 根据待测操作拼接 URL
2. 使用 HTTP 库发起 HTTP 请求

3. 读取 web 服务器返回的信息，并解析数据

4. 对比返回的数据与期望数据



有的测试自动化代码，会把 URL 拼接、HTML/XML 解析、XPath 表达式，和测试逻辑写在一起，通常在同一个类或方法中。

这种方法很容易入手，且很直观，因为其反映了测试人员手工测试的过程。但是这种方法存在一定的问题：

1. 测试逻辑难以理解及修改。当测试逻辑与一大堆无关代码混在一起时，很难辨别出测试逻辑。要添加新测试用例，通常需要重读这些支撑代码才能找到需要修改的代码。测试逻辑也会很难理解。

2. 测试变得很脆弱。因为测试逻辑和 html 解析等支撑代码混在一起，待测系统和自动化测试直接的‘契约’若稍有变化，自动化测试将无法运行。例如，若 UI 发生变化，比如把 input 元素挪到另一个 div 元素下，或者改变某个 UI 元素的 ID，所有相关的测试自动化代码都会受到影响。

3. 维护开销大。一组完备的测试用例会对系统的某个部分进行多组测试，而每组测试间都会存在重复的代码。例如这些代码可能都要 （1）根据待测操作拼装 URL，（2）发出 HTTP 请求，（3）解析 web 服务器返回的信息，（4）比较实际结果及期望结果。因为在各个测试用例间存在 重复代码，如果这个过程发生任何改变，则需要修改各个测试用例的代码。

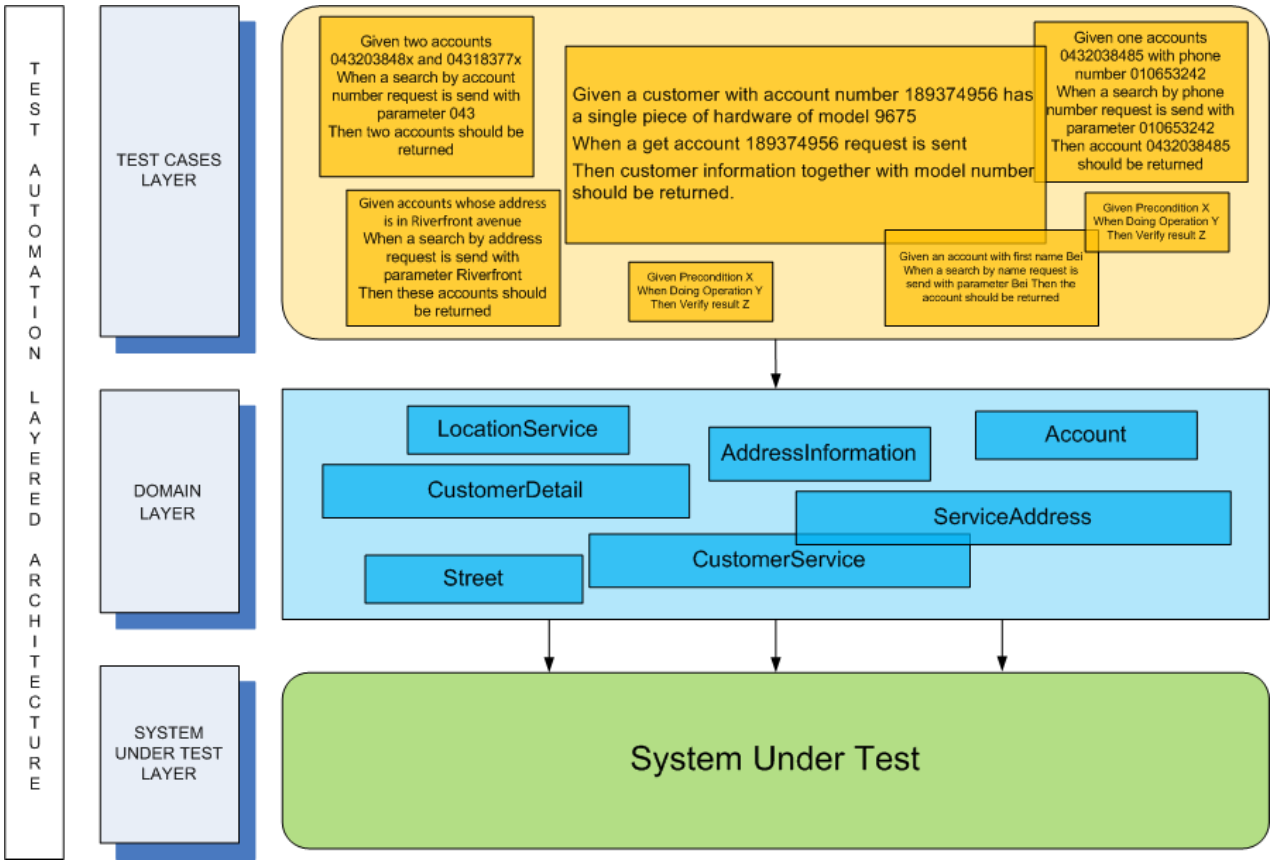
解决方法

软件开发领域曾遇到过同样的问题，并找到了解决方法，即‘层次结构’(Layered Architecture)。引用《领域驱动设计--软件核心复杂性应对之道》('Domain-driven design: tackling complexity in the heart of software') 一书：

> “分层结构的价值在于每一层只关注于程序的特定方面。这使得每个方面的设计都很紧凑，也更容易理解。当然，使用层次结构的最重要原因是把各个重要的方面都分隔开。

虽然测试自动化领域关注的是测试领域，但是所遇到的问题的本质却是一样的，因此可以应用相似的解决方案：

测试用例层	这一层包含所有（并只有）测试逻辑。有了下一层即领域层帮忙，测试逻辑可以很清晰、简洁地表达出来。不同用户故事、场景及边界条件 都构建领域层之上，区别只在于测试数据。
领域层	这一层封装了对待测系统的所有操作，例如 URL 拼接、XML 或 HTML 解析，富客户端或浏览器的控制，等等。通过这一层 包装，待测系统可以以业务领域语言的形式供调用者使用，而非以 xpath、sql 或者 html 等技术“语言”形式。这层的目的在于提高抽象层次。测试的目的是验证业务逻辑是否实现地正确。若测试能用业务领域的语言编写，那么测试目的就一目了然了。
待测系统层	即要测试的系统



测试用例层包含许多测试用例。这些测试用例都是基于领域层的。领域层用领域语言封装了待测系统。领域层直接访问待测系统。

例子

假设我们要测试一个 restful web 服务。通过这个 web 服务，我们可以用电话作为关键字搜索客户信息。

要调用这个 web 服务，需要发起以下格式的 HTTP 请求：

```
http://{endpoint}/subscribers?telephoneNumber={telephoneNumber}
```

服务端返回的以竖线分割的数据包含客户的姓名、电话、地址及其他信息：

```
13120205504|ST|C|SQ|112|||FIRST|ST|W|Riverfront|BC|010|68930432|
```

测试这个服务的用例为：(1)用能精确匹配一个用户的电话作为关键字搜索，(2)用能精确匹配多个用户的电话作为关键字搜索，(3)用 不完整电话作为关键字搜索等。用例的完整程度完全取决于 QA 的想象能力。

对于每个测试用例，执行的数据基本上都一样：(1)拼装包含电话号码关键字的 URL，(2)用 HTTP 库发出 HTTP GET 请求，(3)解析数据， (4)把真实值与期望值做比较。为了避

免上面提到的问题，我们在这里采用分层结构：

测试用例层

这一层的具体实现方式与采用的测试框架有关。在这个例子中，我们采用 C#及 NBehave。

```
[Story]
   public class SearchCustomerbyTelephoneNumberStory: TestBase {
   [Scenario]
      public void SearchWithAPhoneNumberWhichHasAnExactMatch()  {
         story.WithScenario("Search with a phone number which has a exact
match")
          .Given(AN_ACCOUNT_WITH_PHONE_NUMBER, "01068930432", EMPTY_ACTION)
          .When(SEARCH_WITH, "01068930432",SEARCH_WITH_ACTION)
          .Then(ACCOUNT_INFORMATION_SHOULD_BE_RETURNED, "13120205504",
                 ACCOUNT_INFORMATION_SHOULD_BE_RETURNED_ACTION)

          .Given(AN_ACCOUNT_WITH_PHONE_NUMBER, "01062736745")
          .When(SEARCH_WITH, "01062736745")
          .Then(ACCOUNT_INFORMATION_SHOULD_BE_RETURNED, "12666056628");
      }

   [Scenario]
      public void SearchWithPartialPhoneNumber() {
         story.WithScenario("Search with partial phone number")
           .Given(THREE_ACCOUNTS_WITH_PHONE_NUMBER_STARTS_WITH,     "0106",
EMPTY_ACTION)
          .When(SEARCH_WITH, "0106", SEARCH_WITH_ACTION)
          .Then(ACCOUNT_INFORMATION_SHOULD_BE_RETURNED, "13120205504",
                 ACCOUNT_INFORMATION_SHOULD_BE_RETURNED_ACTION)
          .And(ACCOUNT_INFORMATION_SHOULD_BE_RETURNED, "12666056628")
          .And(ACCOUNT_INFORMATION_SHOULD_BE_RETURNED, "17948552843");
      }

   [Scenario]
      public void SearchWithAPhoneNumberWhichHasSeveralExactMatches() {...}

   [Scenario]
      public void SearchWithNonExistentPhoneNumbers() {...}

   [Scenario]
      public void SearchWithInvalidPhoneNumberValues() {...}

      ...
      ...
   }
```

这些测试用例用 C#写成，但是很接近英语，即使非技术人员也可以读懂。（请参照 Martin Fowler 的 BusinessReadableDSL ）。这样，其他的团队成员，特别是对领域更熟悉的业务人员，可以很容易的读懂测试用例，因此也更可能指出测试中遗漏的案例及场景。

若采用支持以自然语言形式书写测试用例的框架(例如 Ruby 平台下的 Cucumber)则会更好。

以"ACTION"结尾的变量为 lambda 表达式。他们是真正的测试逻辑。

SEARCH_WITH_ACTION 会向 web 服务发出请求，并会解析返回的以竖线分割的数据。类 CustomerService 和 Subscriber 在领域层中，他们会在多个测试中重复使用。

```
SEARCH_WITH_ACTION =
          phoneNumber =>
              {
                  subscribers                                                    =
customerService.SearchWithTelephoneNumber(phoneNumber);
              };
```

ACCOUNT_INFORMATION_SHOULD_BE_RETURNED_ACTION 用于验证数据：

```
ACCOUNT_INFORMATION_SHOULD_BE_RETURNED_ACTION =
          accountNumber =>
              {
                  //Get expected subscriber from fixture
                  Subscriber expected = SubscriberFixture.Get(accountNumber);
                  CustomAssert.Contains(expected, subscribers);
              };
```

领域层

CustomerService 类以真实 web 服务的名称命名。在需求文档、日常对话、架构图以及代码中，都用这个名称来指代此 web 服务。使用统一的名称，能除去二义，提高沟通效率。

```
public class CustomerService
   {
       public Subscriber SearchWithTelephoneNumber(string telephoneNumber)
       {
           string url =
               string.Format(
                   "{0}/subscribers?telephoneNumber={1}",
                   endpoint, telephoneNumber);

           //Send http request to web service, parse the xml returned,
           //populate the subscriber object and etc.
           return GetResponse(url);
       }
       ...
   }
```

Subscriber 类建模了用户。比起用竖线分割的字符串，增加一层数据抽象，用对象表示返回的数据，能使测试更容易理解（你应该不会偏好用 pipedData[101]表示电话号码吧？ ）。

```
  public class Subscriber
    {
       public string AccountNumber { get; set; }
```

```
        public string FirstName { get; set; }
        public string Surname { get; set; }
        public string TelephoneNumber { get; set; }
        ...
    }
```

有了这些领域模型，测试就能直接构建在这些对象上了。例如，可以如此验证所返回的用户名为'Bei'：

```
Assert.AreEqual("Bei", subscriber.FirstName);
```

或者电话号码以'010'开始：

```
Assert.IsTrue(subscriber.TelephoneNumber.StartsWith("010"));
```

点击这里可 以下载到样例代码。代码中演示了如何用分层架构组织测试自动化代码。 你可以在 Visual Studio 2008 中打开项目，也可以在命令行运行执行‘go.bat’来运行所有测试。‘go.bat’运行完后会将测试结果保存在‘artifacts’文件夹。源代码中包含三个项目。名称以 with ‘Client’的项目包含领域层。以‘Client.Spec’结尾的项目为领域层对应的单元测试（ TDD ）。‘Stories’项目包含测试用例层。这份源代码由真实项目中来，并作了相应修改。某些 类返回了硬编码的值，是为了不访问真实的 web 服务。

这如何能解决问题？

1. 问题：'测试逻辑难以理解和修改'。现在我们有了一个单独的层表示测试逻辑。这层构建在领域层之上，因此测试可以很用简洁、紧凑的自然语言形式表述，因此阅读、理解、推理和修改测试用例的难度，更取决于编码人员的语言能力，而非编码水平。
2. 问题：'测试很脆弱'。因为我们有一个单独的层把测试用例和待测系统隔离开，若待测系统有任何变化，只有此层会受到影响。只要在此层做相应修改，构建于此层之上的测试用例仍然可以执行。
3. 问题: '维护开销大'。因为有了领域层的封装，各个测试用例中不会再有重复代码。要做修改，也只需修改一处。此外，因为领域模型直接针对待测系统建模，代码也跟容易理解和修改。

常见问题解答

问题：这个方法看起来有些复杂，必须要这么做吗？

回答：这主要取决于待测系统的规模和复杂程度。如果系统规模较小、业务逻辑相对简单，这个方法就过于笨重了。在这种情况下，甚至连测试自动化都可能是浪费时间。如果只花几

分钟时间就能手动测试整个系统，那还自动化干什么呢？若系统较为复杂，把测试逻辑和支持代码混合在一起问题应该不大。而对业务逻辑复杂、规模庞大的系统（也就是说，大部分企业级应用） 我偏好这种方式。

问题：若采用这种结构，那么在开始‘真正’的测试前，需要投入一定时间搭建整个结构，会不会很浪费时间？

回答：这只是另外一种组织代码的方式。即使代码不按照这种方式组织，还是要写代码拼装 URL、解析 XML / HTML、验证测试结果。采用这种结构，只需要把代码拆分到不同的类及方法中。此外，没有必要一次完成整个结构。可以根据当前的测试需要，逐步完成整个结构。

问题：完成这个结构需要相当的面向对象知识，并不是所有 QA 都可以做。

回答：实际上测试自动化并不只是 QA 的职责。项目中其他成员，包括开发人员，也可以参与。

开发人员有很强的编程功底，编写出的代码质量也相对较高，因此可以负责领域层。而 QA 擅长设计测试用例、找出各种边界测试条件，因此可以负责测试用例层。

作者简介：李贝，ThoughtWorks 的咨询师，主要兴趣在于领域驱动设计、测试自动化及领域专属语言。本文原文《Layered Architecture for Test Automation》于 2009 年 8 月 11 日发表在 InfoQ 英文站。

原文链接：http://www.infoq.com/cn/articles/thoughtworks-practice-part9

番外篇

TDD 实践之实用主义

李光磊 ThoughtWorks 公司软件工程师

TDD 的概念和实践依然在被怀疑和争议。然而，质疑者驻足不前，实践者却不再理会争论的口水，而是脚踏实地的实践，并获取高效回报。在这个过程中，我们发现 TDD 和敏捷倡导的理念，帮助我们解决了一个又一个的工程问题。

1. 为沟通选择语言

我们在一个海员管理系统的开发中遇到了问题，这个领域的专业术语我们很难翻译。即使勉强翻译出了，也感觉辞不达意，无论是初看上去，还是过一段时间再看都一头雾水。比如，我们写出了下面的测试用例：

```
public void
test_should_return_NOT_pass_if_duty_higher_than_second_mate_or_second_engin
eer_and_education_level_is_secondary_and_guraduated_after_2002_02_01() {
   ……
}
```

但其中 second mate/second engineer 是什么意思呢? secondary 的 education level 具体又是什么?

还有：

```
public void test_should_return_third_mate_course_for_jianxi_third_mate() {
  ……
}
```

jianxi_third_mate 是什么？等等。

当然，我们可以制定一个术语表，请专业人士先帮我们翻译好，然后在代码中遵循这个术语表。然而随着需求的增加，术语层出不穷，并且有特定中国特色的名词根本就没有对应的翻译，于是这个问题就一直困扰着我们。

而直到有一天，在一次重构中我们把上面的第一个测试用例重命名了一下，一切似乎突然间开朗了：

```
public void test_应该算未通过_if_职务高于二副二管轮_而_学历只是中专_并且_毕业时间晚于2002
```

```
年2月1日() {
    ……
}
```

Team 里的人纷纷围过来，看着这个跟需求描述里的验收条件几乎一模一样的测试用例名称，感受到一种前所未有的清澈。大家几乎在几秒钟之内就做出了选择：这种形式是可以接受的，而且表达能力更强，交流效果不错。

什么?使用中文作为函数名?这似乎只是那些被主流舆论鄙视的"汉语编程"研究者才搞的东西，我们一直就被教育离这些东西远点，甚至汉语拼音都不推荐使用，一个经常拿来做反面教材的例子就是数据库表的列名使用汉语拼音，这被看作不专业的表现。又或者，以后团队中加入外国开发者怎么办?

幸运的是，我们是软件工程师，不是计算机科学家。学术理论可以极端，而工程一定是某种折衷。定理由自然界精确遵守，而工程却是各种应力的人为平衡。

具体到这个案例，让我们正视现实：

1. 团队成员并不善长本项目领域的专业英语。
2. 任何翻译都会造成一定的信息损失，尤其在一些具有中国特色的领域，比如"中专"翻译为英语就很难像中文一样简洁直观。
3. 在可预见的将来，不会有老外加入开发团队。

而选用中文却能够让我们更好的坚持以下原则：

1. 代码除了完成功能，另外一个重要的功能是交流。(我们选择了对团队来说最有效的交流方式)
2. 用测试用例的名字来描述需求。(用中文描述更精确， 易于理解)

当然我们也会失去一些东西，比如对上面提到的"应该坚持使用英文"原则的放弃。在这里我们认为放弃这条原则的收益大于损失。一种损失就是失去了学习英文的机会，比如上面最后一个测试用例，用中文写出来就是：

```
public void test_见习三副应该参加三副的培训() {
    ……
}
```

或者有人会说："见习"的英语是"intern"，常用词啊。然而系统中还有另外一类角色，叫"实习三副"等，那才是"intern"。实习是实际动手，担当实际的职责；见习是只看不练，跟在后面观摩学习。见习的英文单词是"noviciate"，并不为项目组所熟悉，而我们也不再关心它。

总之，在实践中应当权衡各种利弊，选择对你来说最有效的方式。

2. 用大量测试来驱动

在项目组中曾经发生过自以为完成了某个特性，后来却发现漏掉了某些验收条件，甚至是比较重要验收条件的事情。而这也是 TDD 经常被质疑的一点，就是如何保证测试的完备性。因为总是想一点写一点，不经过深入的思考，不可避免的会漏掉某些测试条件。

然而，TDD 并不妨碍你深入思考，只是劝你在实现时步子不要太大，小步前进获得对问题的进一步认识以随时调整设计和实现。不过今天不争论 TDD 的哲学问题，我们只是关注用什么样的实践来消除对于 TDD 测试完备性的疑虑。

事实上，完全可以根据需求文档，验收条件，进行"深入思考"，从一开始就写下所有能想到的单元测试用例，就跟测试人员在产品出来前就对着需求准备测试用例一样。

哦，等等，这还叫 TDD 吗? TDD 所强调的小步前进，随时应变哪里去了？为全面的测试用例所花费的大量努力岂不是有浪费的风险?

嗯，不错，TDD 强调小步前进的原因就是要避免浪费。如果我们能找到一种方法，既能够提醒我们不要忘记需求，又让我们在需求变化时不致浪费太多，岂不是皆大欢喜?

想想，我们用什么来描述需求?是测试用例名称，而不是测试用例的函数体，而名称的书写几乎是没有成本的。从需求文档中把验收条件抠出来即可。如：

```
   public void test_会被认为不服从调配_if_the_seaman_在当前职位上曾经旷工() {
       // TODO
    }

    public void test_会被认为不服从调配_if_the_seaman_在当前职位上曾经请过病假() {
       // TODO
    }

    public void test_会被认为不服从调配_if_the_seaman_在当前职位上曾经请过事假() {
       // TODO
    }

    public void test_会被认为不服从调配_if_the_seaman_在当前职位上曾经被遣返() {
       // TODO
    }

    public void test_不会被认为不服从调配_if_the_seaman_在当前职位上从未旷工_请病假_
请事假_和遣返() {
       // TODO
```

```
    }
```

两分钟，我们就把这个用户故事的测试用例按照验收条件里说的全部描述出来了。函数体全部都是空的，因此所有的测试都是通过的，不会强迫你一次性把所有的测试都实现。

每次你流览或修改这段代码，空的函数体或里面的// TODO 都会提醒你还有测试没有完成。即使你不在这个特性上工作了，切换过来的 Pair 也会从你遗留的测试用例名称中迅速的了解需要做什么。

这种方法是怎么解决问题的呢?

第一，还是让我们正视现实：如果需求描述不和代码放在一起，开发人员很少会在开发过程中去翻阅需求文档，甚至是特性编码结束后。这在成熟的开发团队中会有改善，但仍然不可避免。把需求描述以测试用例名称的方式放进代码，便会无时无刻不在提醒开发者，还有这个这个这个验收条件没满足。

第二，我们依然坚持了以下原则：

1. 用测试用例的名字来描述需求。

2. 小步前进，编写一个测试用例，实现一段产品代码，编写下一个测试用例，实现下一段产品代码。（因为所有的未完成测试都是通过的，不妨碍你运行测试，提交代码和持续集成）

3. 当实现过程中发现事情并不是当初想的那样时，随时更改或删除之前写的测试用例，不会造成大的浪费。（因为只是函数名加空的函数体，成本很低）

在开始写下"所有"的测试用例名称并不意味着一劳永逸。中间当需求发生变化，我们需要对应的添加或删除一部分测试用例。在实现的过程中，发现某些条件不在验收范围内，或许是之前没考虑到，那么跟 BA/QA 确认后，需要添加到用例列表中。

(细心的读者可能发现，这里面存在着重复。就是以普通文本形式存在的验收条件和以测试用例名称存在的验收条件。或许应该有类似 SVN2Wiki 的工具，来消除这类重复)

当然，完备性还有其它的含义和检测条件，不是说你提前多写几个用例就是完备了。这里只是用一种成本最低的方式来解决问题。

3. 一个环境，多个断言

随着时间的推移，项目组发现测试运行的越来越慢。当然，这是一个普遍现象，也已经有很

多方法来加速测试的运行速度。但很多需要新工具的支持，而项目组暂时没时间去切换工具。有没有其它更方便的做法?

像其它性能问题一样，我们首先需要确定瓶颈在哪里。我们发现主要是每个测试用例运行前搭建测试所需的环境相对较为耗时，尤其是 Selenium 测试，它需要启动和关闭浏览器。并且，很多测试用例其实使用相同的环境设置，只是每个用例仅仅去断言其中一个需求。

我们可以修改 Runner，让一组测试使用同一个浏览器实例，每次环境的清理通过清理 Session 来完成。而我们也可以采取另外一种方法，就是合并使用相同环境设置的测试用例，把它们的断言都放进同一个用例。

哦，这又违反了 Kent Beck 为 TDD 制定的原则：每个测试用例最好只有一个断言。

好，让我们再一次分析原则背后的理念。一个用例一个断言，是为了让测试更清晰，更精确的描述需求，测试失败更容易定位。那么有没有一种方法，既能让多个断言共享相同的环境设置，又能清晰精确的描述需求呢?

想想，我们用什么来描述需求?是测试用例的名称，确切的说，是函数的名称。只要我们把一组组相关的断言封装到一个个函数里，给它们一个能够清晰精确的描述这组断言对应的需求的名称，然后在测试用例里面调用这些函数就可以了。这样我们只需为多个断言设置一次环境，而同时又保留了清晰精确的表达需求的能力。

```
@Test
    public void test_should_show_step_details_info_in_todo_item_page() throws
Exception {
        TodoItemPage page = navigator.gotoTodoItemPage( );

        should_show_step_name_as_page_title(activeStepOfNonStartedInstance ,
page);

should_show_start_processing_button_if_current_step_status_is_waiting(page)
;
        should_show_transition_buttons(activeStepOfNonStartedInstance, page);

should_NOT_ask_user_to_input_his_opinion_if_current_step_status_is_NOT_proc
essing(page);
        should_show_comment_box_after_click_start_process(page);
    }

    private    void    should_show_step_name_as_page_title(FlowStep    step,
TodoItemPage page) {
        assertEquals(step.getName(),  page.title());
    }

    private                                                              void
should_show_start_processing_button_if_current_step_status_is_waiting(TodoI
temPage page) {
```

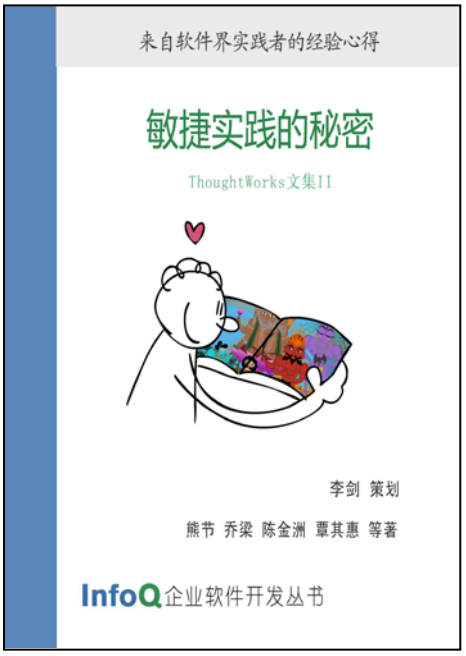
```
        assertTrue(page.isStartProcessingButtonVisible());
    }

    private                                                                 void
should_show_comment_box_after_click_start_process(TodoItemPage page) {
        page。clickStartProcessingButton();
        assertTrue(page.isCommentBoxAppear());
    }

    private                                                                 void
should_ask_user_to_input_his_opinion_if_current_step_status_is_processing(T
odoItemPage page) {
        assertTrue(page.isCommentBoxVisible());
        assertTrue(page.isActionButtonsVisible());
    }

    private                                                                 void
should_ask_user_to_select_next_step_operators(FlowTransitionDefinition
nextTransitonOfStep,
                                                        TodoItemPage page) {
        assertTrue(page。isUserGroupVisible(nextTransitonOfStep.getId()));
    }
    ……
```

小结

回过头来我们看看上面的三个实践，它们如出一辙的，一次又一次的"违反"了某种原则。它们分别是"不能用汉语"，"不能一次编写多个测试用例"，和"不能在一个用例里面使用多组断言"，而实际上，我们违反的只是这些原则的外在形式，但却坚持了这些原则背后的思想，如最有效的沟通，注重实效而不是形式。以此 为基石，我们可以在出现新的约束的情况下，灵活运用，发明各种实践，并享受由此带来的效率提升。

作者简介：李光磊，软件工程师，同时还是一位敏捷教练，就职于 ThoughtWorks。他还是活跃的 blog 作者，了解他最新的想法，请访问 http://blog.csdn.net/chelsea。

原文链接：http://www.infoq.com/cn/articles/tdd-practice



敏捷实践的秘密

----ThoughWorks 文集 II

专题策划：李剑

责任编辑：霍泰稳

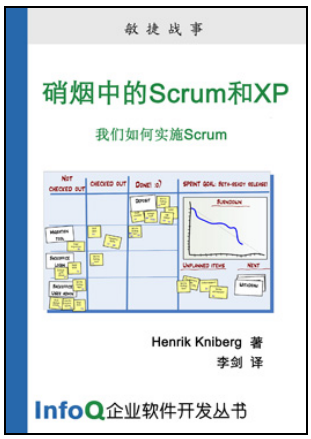
美术编辑：胡伟红

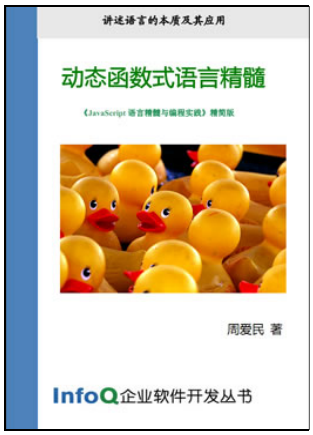
本迷你书主页为

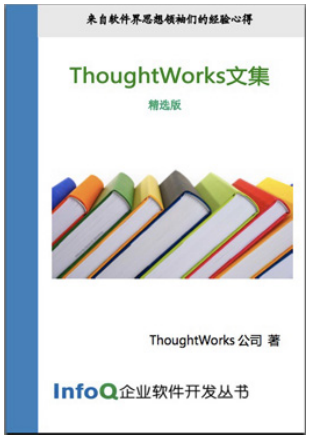
http://www.infoq.com/cn/minibooks/thoughtworks-anthology-ii

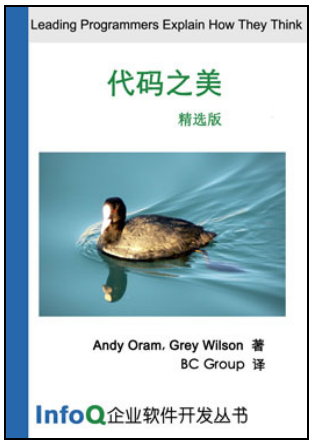
InfoQ企业软件开发丛书

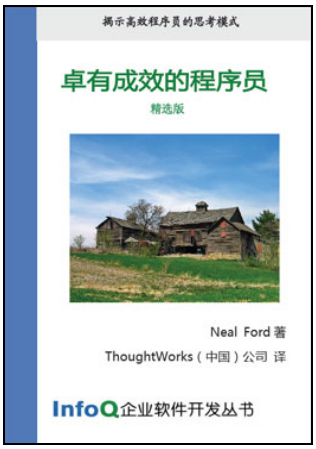
欢迎免费下载

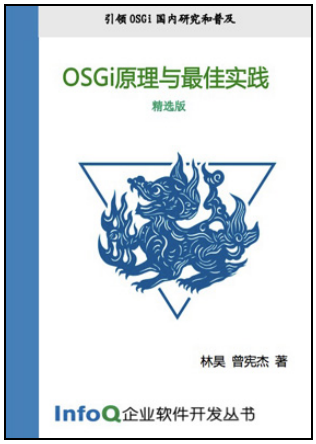












商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com