

跟我学企业敏捷开发

Larry Cai¹

2012-10-30

¹This is the PDF file for the Agile Software Development book contents. It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. I hope you enjoy it, I hope it helps you learn the software development, and I hope you' ll continuously watch this : <http://github.com/larrycai/sdcamp>, will be happy if you follow my weibo <http://weibo.com/larrycai>

前言

对于一个刚加入企业的人来说，一个急切想知道的问题是：企业的软件开发到底是什么？

这本小册子就用几十页左右的内容把企业软件开发的方方面面给介绍一下。既然取名为“跟我学”，就表明它还是比较浅显的，很容易通过实践来了解。又由于比较薄，很多知识点只是入门而已，需要看许多的参考资料才能掌握的。

但同时也不代表里面的东西是很浅薄的，恰恰相反其中的内容应该是最前沿的，最能体现什么是真正的敏捷开发。

为什么写这本书

我在爱立信研发工作多年，从2001年开始研究软件开发的流程，先是极限编程，后来又敏捷。开始在小团队中推动，慢慢地扩大到整个部门，也因此喜欢上了敏捷。后来又参加了Scrum的培训，认识了很多敏捷社区的朋友，了解到了更多更好的企业软件开发方式。

我也一直战斗在第一线，以写代码为乐趣，有很多亲身经历。同时也在多媒体部创建和组织了无数次新人的专业培训，不断把我的很多经验传授给他们。因为我相信软件开发中的人是最重要的，我希望新人能够得到专业的培训，以此来快速适应企业的文化并成长。

在敏捷社区的一次讨论中，很多朋友建议我把经验分享出来。我突然意识到这是一个很不错的建议，虽然我痛恨写作（语文不好一直是我的一个心病），但或许能用这次机会来加强自己这方面的能力。

而且我是喜欢自由，也喜欢分享的技术爱好者，推崇用开源的技术方式来写技术类的书籍，希望能推动国内的开源技术书的发展。

不管怎样，我愿意分享我们公司培养新人敏捷开发的方式，期望你们能少走弯路。

1. 养成好的开发方式。站得高，看得远。不要一开始就纠缠于差的环境。
2. 养成团结开放的心态。软件是开放的世界，要勇于共享，不断进步。

总之，特别想用我们的经验来告诉埋头苦干的人，要站在巨人的肩膀上。

特色

这里边讲到的内容我们早已经在爱立信公司实践多次，特别适合刚加入企业的新员工，来快速地熟悉企业所使用的实践技术。

每一章内容都配备足够的练习，通过练习使员工掌握后面的理论知识，来一步步地了解软件开发的各个角落。

每章内容都是精心挑选的，难度适当。各章之间都有很好的衔接，也可独自成为一个知识点，可以独立章节学习，当然更加推荐连续学习。

本书结构

从第一章到第六章，每章都对应一门2-3小时的课，第七章是单独的一天来实践项目开发。附录A是如果在企业内部采用本书安排的一个模板。

- * 第一章：敏捷开发和Scrum
- * 第二章：版本控制Git和代码审阅Gerrit：使用分布式系统Git与Gerrit系统
- * 第三章：持续集成：使用Jenkins、Maven、Java项目
- * 第四章：如何写好Java程序：Java的覆盖率，单元测试，重构，测试驱动开发
- * 第五章：需求管理和实例化需求Specification by example
- * 第六章：用Cucumber来实例化需求
- * 第七章：项目实践：Game of life

适合读者

本书适合所有正在看的人，否则你也不会读到这里。作为作者来说，我认为此书尤其适合以下读者：

企业新人

进入企业，对如何正确地开发有点朦胧，觉得什么都重要，又看上去什么都不是那么要紧。入职的导师可能水平也不是很够，项目老是很紧张，需求要不是很清楚；有人抱怨敏捷是浆糊，又有人说敏捷是必须得。这些很常见的问题，很少在前期了解。

本书希望给你一个比较高的起点，告诉你优秀的软件开发是怎么做的，它会帮助你建立一个好的软件开发的价值观。

同时里面会讨论到相关的开发技术，可能你还没用上，这正好可以给你一个比较的机会。去了解一下各个技术的优缺点，或许你有机会推动你公司的软件开发水平。如果有更好的技术，别忘了通知一下。

技术经理

作为技术经理，已经在一个企业呆了一段时间了，可能对一切都习以为常；有时候有心改变，又无从下手。

本书希望给你一些启迪，让你了解企业软件开发可以怎么做，你可以翻阅你喜欢的章节，照着试试。相信会给你带了一些启发。

同时本书也给你介绍了如何在一个公司做专业的敏捷开发培训，你完全可以基于本书的内容，裁剪成适合你公司的内容。如果你真得这么做了，我会万分感想，因为这就是我写书的目的之一。

大学学生

快大学毕业了，也写了不少代码了，熟悉了很多算法、IDE用得也甬溜。突然发现找工作老问有没有经验；或者直接问什么是敏捷开发、持续集成等等。你或许突击了解过这些名词，但是具体为什么没有切身体会。

本书就是给你介绍一个企业软件开发碰到的方方面面，结合很多的实际例子，让你来通过实践的机会来体会企业中为什么老是谈到这些东西。

退一步说，看完后至少你简历上又多了好几条知识，不过千万不要说掌握，还是说“了解”或“知道”谦虚一点吧。

最好的方式就是到爱立信上海研发中心多媒体部来做实习生吧，我们会安排最专业的老师给你上课，有完备的培训材料 and 环境，保证你了解得更多。

还犹豫什么，找我们吧？当然对技术没追求的就算了。

如何使用本书

这本小册子最好是公司培训的参考手册，一般应该有专业的老师准备好对8-10人的团队进行指导。

当然你可以照着自学，多看看参考资料，毕竟这是一本软件开发的入门书，高度够了，深度不够。

几天的学习会很有意思，学完后，你会意识到在企业中你再也不可能什么都会，要抛弃单兵独斗的想法，要学会团队合作，更要持续地学习提高。而且要有激情，否则什么都学不到。

如何写作本书的

本书也是用敏捷的方式来写作的，我在Windows（希望早日能用MBA）的Notepad++编辑器里用Markdown的文本文件写内容，然后在共享目录的虚拟机中，一个命令生成所有的其他格式（如PDF、mobi、epub）；Git是我使用的版本控制系统。

本地测试通过后，我会提交到[GitHub](#)中去，同时架设了持续集成服务器，会对我最新内容再次检测，自动产生电子书。

这方面有兴趣的朋友可以继续翻看我在写的小书【跟我学开源技术书】 <https://github.com/larrycai/kaiyuanbook>。

自从花了一周时间写出第一版后，我就力争持续发布，同时我一直在纠结本书的深度和广度。

封面

因为自己爱好设计排版，就自己DIY了一下，封面是瑞典南部军港[卡尔斯克鲁纳](#) (Karlskrona) 市外面斯图尔克岛 (Sturko) 的一角，用Windows PowerPoint制作完成。

版本变化

下面是每次的版本变化清单，你也可以自己在Github上查看：<https://github.com/larrycai/sdcamp/compare/v1.0.0...v1.2.0>

2012年2月14日-1.2.0

一个比较完备的发行版，60页左右。

- * 加强了书的格式，解决了中文字体显示的问题，加上了色彩，加上了图书推荐页。
- * 加强了书的格式，前言和致谢挪到了目录前，附录区别于正常章节。
- * 补全了Git、持续集成、实例化需求、Cucumber章节。

2012年1月12日-1.0.0

第一次正式发布，40页左右，开始内部培训使用，包含完整的章节，但有些章节内容不完整。

致谢

首先应该感谢上海爱立信研发中心允许我把这些内部培训资料分享给大家。
特别感谢许晓斌 (@juvenxu) 在结构、方向和细节处给了我很多的建议。
特别感谢徐毅 (@徐毅-Kaveri) 给我写书的动力，也期待他的测试新书早日面世。
感谢李任、王艳、晁立山、梅一、姜信宝、卢绮闽对本书做了早期的审阅。
非常感谢北京的李小波从一开始就用开源的方式对文字细节进行不断地校对。

目录

前言	i
版本变化	v
致谢	vii
目录	ix
1 敏捷开发和Scrum	1
1.1 工作环境	1
1.2 简单历史	1
1.2.1 敏捷流派	2
1.3 Scrum 基本知识	2
1.3.1 基本角色	2
1.3.2 框架过程	3
1.3.3 常用的实践	3
1.4 相关知识	3
1.5 课后练习	4
1.6 小结	4
1.7 参考阅读	4
2 版本控制Git和代码审阅Gerrit	5
2.1 工作环境	5
2.2 什么是Git	5
2.2.1 集中式和分布式	5
2.3 Git基本用法	6
2.3.1 安装	6
2.3.2 配置 Git	6
2.3.3 建立本地 Git 仓库	7
2.3.4 第一个提交	7
2.3.5 Git分支 (Branch) 和合并 (Merge)	7
2.3.6 Git变基 (Rebase)	8
2.3.7 Git标记 (Tag)	8
2.4 Git远程仓库连接	8
2.4.1 在Gerrit中注册	9
2.4.2 Git克隆 (Clone)	9
2.4.3 Git推送/拉 (Push/Pull)	9

2.5	Git的良好使用习惯	10
2.5.1	提交注释的质量	10
2.6	常用的工作模式	10
2.6.1	本地特性分支	10
2.7	代码审阅和Gerrit	11
2.8	Git的缺点	11
2.9	相关知识	11
2.9.1	几种协议	12
2.10	课后练习	12
2.11	小结	12
2.12	参考阅读	12
3	持续集成	13
3.1	环境准备	13
3.2	持续集成流程	13
3.3	Maven	14
3.3.1	安装Maven	14
3.3.2	Maven仓库管理器: Nexus	14
3.3.3	第一个maven命令	15
3.3.4	体会两层缓存	15
3.4	持续集成服务器: Jenkins	16
3.4.1	安装	16
3.4.2	安装Git插件	16
3.4.3	系统配置Maven	16
3.4.4	设置构建任务	16
3.5	如何实施持续集成	17
3.6	相关知识	17
3.7	课后练习	17
3.8	小结	18
3.9	参考阅读	18
4	如何写好Java程序	19
4.1	环境准备	19
4.2	代码风格和编程规范	20
4.3	安全代码	20
4.4	单元测试	20
4.5	代码覆盖率	20
4.6	重构	20
4.7	测试驱动开发 TDD	21
4.7.1	测试驱动开发的步骤	21
4.7.2	单元测试就是文档	22
4.7.3	单元测试也要重构	22
4.8	测试模拟 (Mock)	22
4.9	课后练习	22
4.10	参考阅读	22

5 需求管理和实例化需求	25
5.1 环境准备	25
5.2 需求的困惑	25
5.2.1 测试人员的工作文档	26
5.3 用实例化来解决需求的问题	26
5.3.1 主要过程模式	27
5.3.2 网上书店一个例子	27
5.3.3 常见问题	29
5.4 如何实施	30
5.4.1 循序渐进和现有流程的结合	30
5.4.2 贴在墙上	31
5.5 相关知识	31
5.6 课后练习	31
5.7 小结	31
5.8 参考阅读	31
 6 用Cucumber来实例化需求	 33
6.1 环境准备	33
6.2 Cucumber 简介	33
6.3 安装	34
6.4 运行Cucumber	34
6.5 业务层: Gherkin语言	35
6.6 驱动层	36
6.7 常用的目录结构	36
6.8 继续网上书店的例子	36
6.9 常见问题	37
6.9.1 我们的系统没有接口能够被这么（或容易）测试得?	37
6.9.2 Cucumber用起来了，也自动化了，但是没人看?	37
6.10 相关知识	37
6.11 课后练习	37
6.12 小结	38
6.13 参考阅读	38
 7 项目实践: Game of life	 39
7.1 常见问题	39
7.1.1 过度承诺 (over commit)	39
7.1.2 没有团队精神	39
7.1.3 缺少计划和工作方式的制定	39
7.2 小结	40
 附录 A 培训示例模板	 41
A.1 时间安排	41
A.2 培训的材料	41
A.3 培训负责人准备工作	41
A.4 反馈	42

附录 B 企业版本控制的改革：走向Git	43
B.1 了解最新技术-分布式版本控制 (DVCS)	43
B.2 尝试在日常中使用分布式版本控制	43
B.3 宣扬和推广分布式版本控制	44
B.4 详细研究版本迁移	44
B.5 开始在小范围实施	44
B.6 推广、并引入Gerrit做代码审查	45
B.7 小结	45

第 1 章

敏捷开发和Scrum

敏捷软件开发越来越流行了，而且基本深入人心。技术水平高的人尤其推崇。

当我们学习编程时，本性都是敏捷的，谁都不想浪费时间。只是进入企业（不管大小）后，由于管理的需要，产生了不必要的浪费，也就显得不太敏捷了。

瀑布V模型在早期开发周期长，需求变化少的情况下还是很不错的，只是互联网时代软件开发技术日新月异，更新越来越快，这又不得不回到原来的思想，精简管理来降低浪费。

由此不要抱怨敏捷，它只是揭开了软件开发的遮羞布而已。

敏捷是由很多技术实践结合在一起的，依靠有经验的开发者去实施。

1.1 工作环境

不需要电脑，积极回答问题，并多多提问。

1.2 简单历史

敏捷这个术语早期有人叫轻量级（lightweight）软件过程，来区别于CMM、RUP等重量级软件过程。后来又觉得本质不是轻重的问题，所以又改成敏捷（Agile）。



图 1.1: 敏捷宣言

敏捷的技术实践在敏捷出现前就出现了，如持续集成，代码共享，结对编程。甚至是那些敏捷流派，如XP、Scrum、FDD都早就有了。只是这些技术先驱们觉得单打独斗太累，因此

在一次聚会中一起创建了敏捷宣言¹。

1.2.1 敏捷流派

从2004年起，敏捷开始展露锋芒，主要原因是恰好互联网企业需要快速开发，快速交付。他们就顺理成章地采用了敏捷的方式。

同时传统企业开始感受到了压力，碰到了问题，需要改进了，看看别人都敏捷了，开始跟风（褒义）了，这就碰到了选择的问题。记住，只有等你到了一定的水准后，才能无招胜有招。早期还是要学些固定套路的，这些套路就是不同的敏捷开发过程。

- * XP（极限编程）较早出现在中国的原因，得益于当初翻译的几本书（2001年），不过有点极端了，很多传统企业都不能适应。
- * Scrum是一个框架，概念清晰，比较容易上手（狡猾），当然它还是得和其他实践同步开展。不管怎么样，Scrum越来越流行了；当然骂声也不少，认为它什么都没讲，太虚了。实际上他们大多数人把自身的问题归结于Scrum了。
- * FDD（Feature Driven Development）等还有一些其他的过程，声音慢慢就越来越少了。猜想商业是一方面，推动者的能力或兴趣也是一方面。

这里，我们主要以Scrum来讲解敏捷，但千万别以为Scrum就是敏捷。可以阅读相关知识来了解更多的敏捷。

1.3 Scrum 基本知识

Scrum²是迭代式增量软件开发过程，也是一种敏捷软件开发的框架，通常用于敏捷软件开发。Scrum在英语的意思是橄榄球里的争球。

1.3.1 基本角色

Scrum是一个包括了一系列实践和预定义角色的过程框架。Scrum中的主要角色包括：

1. **Scrum Master**是来确保团队合理的运作Scrum，并帮助团队移除实施中的障碍。
2. **产品负责人(PO: Product Owner)**，确定产品的方向和愿景，定义产品发布的内容、优先级及交付时间，为产品负责。
3. **开发团队(Team)**，一个跨职能的小团队，人数5-9人，团队拥有交付可用软件需要的各种技能。

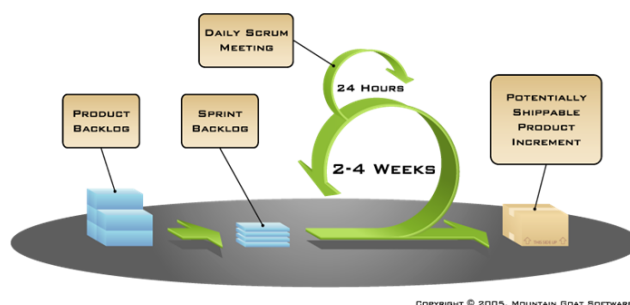


图 1.2: Scrum框架

¹<http://agilemanifesto.org/iso/zhchs/>

²<http://zh.wikipedia.org/wiki/Scrum>

1.3.2 框架过程

在每一个Sprint（两到四周的周期，其长度由开发团队决定）当中，开发团队创建可用的（可以随时推出）软件的一个增量。每一个Sprint所要实现的功能来自**产品待办事项列表（Product backlog）**。

产品待办事项列表是按照优先级排列的要完成的工作的概要需求，在团队的**计划会议（Planning meeting）**中，PO给出各个功能的优先级，开发团队一起决定在下一Sprint中他们能够承诺完成多少功能，这就形成了**Sprint待办事项列表（Sprint backlog）**。

在Sprint过程中，没有人能够变更Sprint待办事项列表，这意味着在Sprint中需求是被冻结的(No Change)。

每日站立会议（Daily standup meeting）一般定在早上，持续10分钟左右，每个团队成员需要回答三个问题来了解整个的运行情况和潜在的风险：

1. 昨天你完成了哪些工作？
2. 今天你打算做什么？
3. 完成你的目标是否存在什么障碍？

在Sprint结束时，会有一个**评审会议（Review meeting）**来检查一下功能是否按照产品负责人要求地完成了，质量应该是由团队保证的，而不是产品负责人或其他人负责。

评审会议后的**回顾会议（Retrospective meeting）**是团队自己帮助自己发现问题，并提出行之有效的方式进行提高，所以这不是由其他人来组织的。

1.3.3 常用的实践

管理Scrum过程有很多实施方法，如即时贴(yellow stick)、燃尽图(burndown chart)、白板(whiteboard)。Scrum最大的好处之一是它非常容易学习，而且启动Scrum应用并不需要太多的投入。

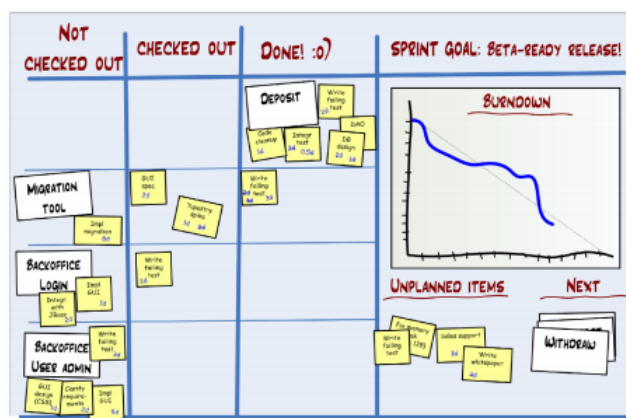


图 1.3: 每日例会中的任务白板（图来自《硝烟中的Scrum和XP》一书）

1.4 相关知识

敏捷这个范畴很大，在过程这个方面，建议看看XP和Lean、看板等内容。

1.5 课后练习

学会使用白板来做团队任务的分配，并且开始体会团队合作和企业中的工作任务和以前的不同。

1. 在Wiki系统中，各自创建自己的个人主页。
2. 敏捷团队到底要不要团队组长（Team leader），简单阐述想法，记录在Wiki中。
3. 一个敏捷团队几个人是最适合的，简单阐述想法，记录在Wiki中。
4. Scrum周期多长是最合适的，简单阐述想法，记录在Wiki中。

1.6 小结

敏捷听起来很虚，但是实际上回归了软件开发的本质，关注需求，团队合作，不断进步。

1.7 参考阅读

1. 硝烟中的Scrum和XP: <http://www.infoq.com/cn/minibooks/scrum-xp-from-the-trenches>
2. 看板和Scrum——相得益彰: <http://www.infoq.com/cn/minibooks/kanban-scrum-minibook-cn>
3. 你的Scrum检查列表: <http://www.infoq.com/cn/minibooks/scrum-checklists-cn>
4. What is scrum? http://www.scrumalliance.org/pages/what_is_scrum

第 2 章

版本控制Git和代码审阅Gerrit

如果你还停留在SVN阶段，或者从没有玩过Git，那太落伍了。Git是版本控制的一个飞跃，它极大的提高了软件开发的效率。

代码审阅有好几种方式，走读式效果不佳（有点事后诸葛亮的味道），结对编程（Pair Programming）一直是蛮多人推荐的方式，但真正在企业中实施成功的不是很多，不过还是值得推荐的。

基于Gerrit方式的代码审阅有很多的优点，能很好得满足企业的需要。

2.1 工作环境

- * 服务器端推荐用 Gerrit <http://code.google.com/p/gerrit/>
- * 客户端用Windows版的Git: <http://code.google.com/p/msysgit/>

2.2 什么是Git

Git最早是Linux用于Linux内核开发的版本控制工具。与常用的版本控制工具 CVS、Subversion 等不同，它采用了分布式版本库的方式，不需服务器端软件支持，使源代码的发布和交流极其方便。

Git的速度很快，既然它能应付Linux kernel这样的大项目，那么相信对大多数的企业软件的协作开发和代码量，它也是能胜任的。

Git最为出色的是它的合并跟踪（Merge tracing）能力和强大的社区支持。

2.2.1 集中式和分布式

企业常用的SVN和ClearCase是集中式版本控制系统，服务器架在IT环境中，本地只是签出代码的一个快照。很多操作如历史记录查询都必须连接到服务器才行。只要依赖网络，就会带来不必要的麻烦，比如在家办公。

分布式顾名思义就是代码仓库是可以分布在各处的，那样你就可以做很多以前必须要配置管理员参与的事情，如分支。当然它也带来一定的复杂性，早期可能还不太适应。有兴趣的朋友可以在附录B看看我的一些推广经验：“企业版本控制的改革：走向Git”。

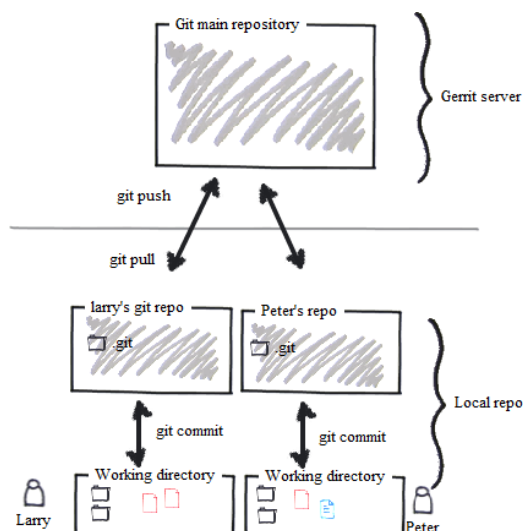


图 2.1: Git分布式版本控制

2.3 Git基本用法

Git的学习曲线相对来说还是有点陡的，但只要掌握了基本的一些命令，开始日常的工作就没有问题了。

2.3.1 安装

先装好Windows版的Git(“Git for windows”), 很多人老是说装msysgit, 实际上我们要的只是Git的工作环境, 而msysgit是一个含有整套源码环境的系统(如C编译器)完整包, 除非你是个Git极客, 否则别自寻麻烦。

缺省安装就可以了, 除非你是专家, 否则别选Putty(一种常用的远端控制台软件)的SSH。初学者80%的Git的问题出在SSH连接上。

2.3.2 配置 Git

首先要告诉Git你是谁, 怎么联系你, 这样在代码库中才能找到提交者; 同时界面也可设置成彩色来方便阅读。在SHELL环境下进行如下全局配置:

```
$ git config --global user.name "Your name"
$ git config --global user.email "Your email address"
$ git config --global color.ui auto
```

--global就是把全局配置放在你的HOME下 ~, 下面两条命令都可看到全局定义。

```
$ less ~/.gitconfig
$ git config -l --global
```

2.3.3 建立本地 Git 仓库

既然是分布式，就可以直接在本地创建Git仓库了。先生成一个干净目录helloworld并初始化成Git仓库。

```
$ cd ~
$ mkdir helloworld
$ cd helloworld
$ git init # 初始化本地仓库
Initialized empty Git repository in c:/Users/larrycai/helloworld/.git/
```

养成习惯经常看看目录下有什么变化了。

```
$ find .
.
./.git
./.git/config
./.git/hooks
...
./.git/hooks/update.sample
./.git/info
./.git/objects
./.git/refs
./.git/refs/heads
./.git/refs/tags
```

你会发现新建了.git目录，在下面还有很多东西，自己瞅瞅，琢磨琢磨，这也是平时自我提高的一个办法。不管怎样，这就是你的本地Git仓库了。

2.3.4 第一个提交

然后可以试着加入一些代码并签入本地版本库。

```
$ cat "Hello Git World" > README # 建一个空文件
$ git status # 会发现报告红色的未跟踪的文件
$ touch README # 创建空文件
$ git add README # 加入暂存 (stage)区
$ git status & find . # 变绿色，跟踪了。产生一个索引
$ git commit -am "add first empty file" # 签入代码到本地，要养成好习惯写好提交的注释。
$ git status & find . # 干净了，索引变化了。
$ git log
$ git blame # 查看谁改的
```

要细心体会每次的变化，就这么简单，也不那么容易。

2.3.5 Git分支 (Branch) 和合并 (Merge)

为了不影响团队其他成员的开发，常常建立一个分支 (Branch) 用来开发新功能和修改bug，等开发完成后，再合并 (merge) 到主分支 (master) 上供其他人使用。

分支和合并在其他大多数的版本控制系统中（如SVN，ClearCase）都是高级课程，而在Git中，一会儿就学到了。记住，在分布式版本控制系统中，这是一种很常用的工作方式。

一个Git仓库可以维护很多开发分支并快速切换，这是推荐的工作方式，而在SVN中，分支是尽量避免的。

```
$ git branch bug123 #创建关于 bug 123的分支
$ git branch # 看看有哪些分支，master是主分支。
  bug123
* master
$ git checkout bug123 # 切换到bug123分支。
Switched to branch 'bug123'
$ git checkout -b feature234 # 创建并直接切换到feature234分支
```

当需要合并时，切换到需要合并的分支上，如果需要，可以使用kdiff等软件。

```
$ git checkout master # 切换到主分支
$ git merge bug123 # bug123已解决，合并bug123
$ git branch -d bug123 # bug123没用了，可以删除这个分支了。
```

2.3.6 Git变基 (Rebase)

在两个分支之间同步的操作除了合并，还有一个类似的命令叫变基（rebase）。它就是把你的分支重新更新到新的基础之上。

```
$ git checkout master
$ git checkout -b bug123 # 从主分支工作在bug123分支上
$ git rebase master # 变基到最新的主分支的内容，继续修改bug123
```

2.3.7 Git标记 (Tag)

一般在发布前，我们需要打一个标记（Tag），表明这是一个重要的点，以后可以很方便地把当前的状态恢复，省得记录某个固定的签入了。

```
$ git tag -a v1.0.0 -m "official release for version 1.0.0" # 创建里程碑并加注释
$ git tag # 列出所有的里程碑
$ git checkout v1.0.0 # 以后可以很方便地签出里程碑 v1.0.0
```

2.4 Git远程仓库连接

到现在为止，我们一直在本地练习，该把代码上传到Git服务器了。Git服务器有好几种，如Gitolite、Gerrit。企业建议用Gerrit。

Gerrit是基于SSH协议用Java实现的Git服务器，谷歌Android开源项目就是使用Gerrit。

2.4.1 在Gerrit中注册

使用前，需要在Gerrit中注册，首先用正确的账号和密码登陆，然后上传你的SSH公钥。

SSH公钥是要用ssh命令产生的。运行ssh-keygen就会在根目录下创建.ssh目录和生成公私密钥文件id_rsa.pub，id_rsa。

```
$ ssh-keygen # 提示输入密码时回车用空密码就可以了！
```

id_rsa.pub就是公钥文件，上传并放在你的Gerrit账户下面。以后Git的相关命令就通过SSH来验证。

2.4.2 Git克隆 (Clone)

从远端Git服务器上把代码从远端Git仓库拿到本地的操作就叫克隆 (Clone)。

```
$ git clone ssh://larrycai@gerritserver.company.com:29418/gameoflife.git
```

如果一切正确，代码和它的全部历史就到了本地，记住你拿到的是完整的Git仓库，只是在本地而已，否则就不是分布式了。可以看看.git目录，或者打一下git log体会一下。

上面的命令中：

- * ssh:// 代表了访问的协议，后面的29418是SSH协议的通信端口，Gerrit不使用缺省端口22。
- * larrycai 是Gerrit中的账号ID，如果和你本地的ID相同可以省略。
- * gerritserver.company.com 是你企业使用Gerrit的Git服务器。
- * gameoflife.git 是Git仓库名字，一般习惯以.git作为后缀。

2.4.3 Git推送/拉(Push/Pull)

克隆后，你就可以在本地创建分支修改代码，并使用前面学习的Git命令来进行版本控制。

当完成一定的任务，代码修改完毕后，就可以考虑推送 (Push) 到远程仓库和别人共享。

```
$ git push
```

命令格式是：git push [remote-name] [branch-name]，缺省是origin和master。克隆操作会自动使用默认的master和 origin名字。可以看看.git/config文件。

同样得，为了同步其他人的最新代码，我们需要经常把最后的内容更新从远程仓库拉 (Pull) 下来，以此来更新本地仓库。

```
$ git pull
```

命令格式是：`git pull [remote-name] [branch-name]`，缺省也是origin和master。

2.5 Git的良好使用习惯

从一开始就需要养成良好的使用习惯，提交注释（commit message）的质量是一个经常被忽略的问题。

2.5.1 提交注释的质量

你的代码写完后是要让人看的，别人从版本库中查看代码的第一件事是读你提交的注释，因此一定要提高注释的质量，标准的做法¹是：

1. 第一行是简要介绍。让人明白做这个改变的原因？而不是你做了什么。
2. 接着一个空白的一行。
3. 有需要的话，然后用剩下的文本进行详细介绍。

如 [Linux Kernel commit 3db59dd9](#):

```
ima: fix cred sparse warning

Fix ima_policy.c sparse "warning: dereference of noderef expression"
message, by accessing cred->uid using current_cred().

Changelog v1:
- Change __cred to just cred (based on David Howell's comment)
```

如果简单，第2,3项可以省略。

要记住：**提交的注释能看出背后是否是一个专业的开发者**

2.6 常用的工作模式

有好几种Git工作模式可以学习，常用的是使用本地特性分支的方式。

2.6.1 本地特性分支

经常用本地主分支同步远端仓库，建立本地特性分支进行代码开发，可以同时存在多个分支。

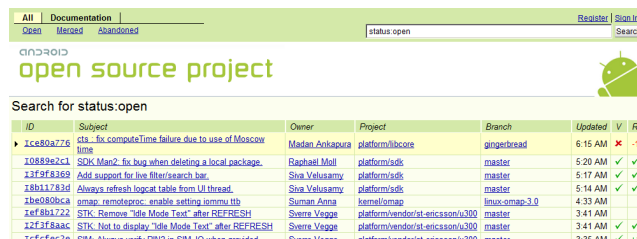
任务完成后，先在本地的主分支和远端仓库同步一次，再在特性分支和主分支变基一次，使得本地特性分支是基于最新代码开发的。

然后再切换到主分支，把本地特性分支合并上来，最后再推送到远端仓库。

¹Stackoverflow 上的解答 <http://stackoverflow.com/questions/2290016/git-commit-messages-50-72-formatting>

2.7 代码审阅和Gerrit

代码审阅是一个不错的敏捷开发实践，但实施却往往让人非常头疼。大企业中通常是制定出一大堆相关的规范和流程来指导代码审阅。谷歌的 Android 系统是现在非常热门的开源项目，它的代码审阅（包括贡献者的代码）就是基于Gerrit的流程，非常棒。



The screenshot shows the Gerrit web interface for the 'open source project'. It displays a search bar with 'status:open' and a table of open source projects. The table has columns for ID, Subject, Owner, Project, Branch, Updated, and a status column with checkmarks and minus signs.

ID	Subject	Owner	Project	Branch	Updated	✓	-
1ce80a776	cts - fix computeTime failure due to use of Moscow time	Madan Anithapara	platform/libcore	gingerbread	6:15 AM	✓	-1
10489e2c1	SDK Man2: fix bug when deleting a local package.	Raphael Moll	platform/adb	master	5:20 AM	✓	
13f2f8369	Add support for live filter/search bar.	Shiva Velusamy	platform/adb	master	5:17 AM	✓	
14b11783d	Always refresh logcat table from UI thread.	Shiva Velusamy	platform/adb	master	5:14 AM	✓	
1be080bca	omap remoteproc: enable setting jmmu lib	Suman Anna	kernel/omap	linux-omap-3.0	4:33 AM	✓	
1ef8b1722	STK: Remove "Idle Mode Text" after REFRESH	Svenne Vagge	platform/vendor/st-ericsson/u300	master	3:41 AM	✓	
12f1f8aac	STK: Not to display "Idle Mode Text" after REFRESH	Svenne Vagge	platform/vendor/st-ericsson/u300	master	3:41 AM	✓	
1c4cf8e2e	SIM: Always verify PIN2 in SIM_IO when provided	Svenne Vagge	platform/vendor/st-ericsson/u300	master	3:35 AM	✓	+1

图 2.2: Gerrit代码审阅系统

Gerrit是一个基于 Web 的代码评审和项目管理的工具，面向基于Git 版本控制系统的项目，所以如果你没用Git，就没法用Gerrit了，接下来看看在Gerrit中是怎么实施代码评审的。

- * 首先开发者（贡献者）的代码变更通过Git命令被推送到Gerrit管理下的Git 版本库，推送的提交转化为一个一个的代码审核任务
- * 代码审核者可以通过Web界面查看审核任务、代码变更，通过 Web 界面做出通过代码审核（Review）或者拒绝（Reject）等决定。
- * 测试者（一般可以设定为持续集成的服务器执行）可以通过访问来获取代码变更进行相应测试，如果测试通过，就可以把这个评审任务设置为校验通过（Verified）。
- * 最后经过了审核和校验的代码变更可以通过Gerrit界面中提交动作合并到版本库的对应分支。

相比代码走读，它的好处在于，审阅动作发生在向主干提交代码前，可以只看变更的部分显得很贴心，网上随时随地审阅起来也很方便，这也是有别于结对编程的一个好处。

任何人都可以审阅提交的代码，整个团队的代码都一目了然，审阅起来更方便，非常符合开放、透明的敏捷精神。使用之后能够显著提高代码质量，甚至于等到习惯了以后，代码不被审阅一下，都觉得实在是不好意思提交到主干上去。

Gerrit中通过特定分支，任何审核任务的代码变更都能访问，所以如果需要细看或是合并到本地都异常的方便。

2.8 Git的缺点

相比SVN、Mercurial，Git的学习还是需要花更多的时间，但是掌握基本的命令就可以畅通无阻了。

如果你喜欢上了她，你可能会喜欢她的一切，对Git也如此。作为技术人员，看到一些小命令、小技巧，会越来越有兴趣。

所以努力克服一点小问题，越过这个门槛，前途会更加美好，就是我的建议。

2.9 相关知识

GitHub、BitBucket、GoogleCode是非常流行的开源项目托管网站，也都支持Git，建议熟悉一下。

Mercurial (Hg) 也是一个和Git相类似的分布式版本控制系统，可以了解一下。

2.9.1 几种协议

访问远端仓库，大部分情况下使用SSH协议，实际上Git也可以用其他协议如git://和http://，这些都是由Git服务器提供的服务。

```
$ git clone ssh://git@gitserver/repo.git # 用git用户访问，常见于gitolite
$ git clone git@gitserver/repo.git # SSH协议，和上面一样，ssh://省略了
$ git clone git@github.com:larrycai/sdcamp.git # SSH协议，用git用户访问，转到larrycai用户，常见于github
$ git clone larrycai@gitserver/repo.git # SSH协议，直接larrycai用户
$ git clone ssh://larrycai@gitserver:29418/repo.git # SSH协议，一般是Gerrit服务器
$ git clone git://gitserver/repo.git # Git协议，一般用于克隆只读
$ git clone https://larrycai@gitserver/repo.git # HTTP协议，大部分情况是为了绕过防火墙
```

2.10 课后练习

- * 习惯使用Windows版的Git Bash环境。
- * 继续练习常用的例子：如熟练应用本地分支来开发任务、服务器同步。
- * 尝试用Gerrit给你所在产品的代码进行审阅。
- * 注册Github，并尝试提交本书的补丁。

2.11 小结

Git是一个分布式版本控制系统，不应该用以前集中式的版本控制系统的思路去考虑。要反复练习来熟悉一些基本的用法，慢慢提高使用水平。

随着Git的日益普及，网上的资料已经很多，多问多玩。

Gerrit的代码审阅特别适合提高企业产品的代码质量，而且又不花费很多额外的时间。

2.12 参考阅读

1. Git权威指南: <http://www.worldhello.net/gotgit/>
2. Pro Git中文: <http://progit.org/book/zh/>
3. Git Community Book 中文版 <http://gitbook.liuhui998.com/>
4. Gerrit <http://code.google.com/p/gerrit/>
5. Windows版的Git: <http://code.google.com/p/msysgit/>

第 3 章

持续集成

持续集成是一种软件开发实践，它是Martin Fowler先生提出¹的；一个在企业开发中必须的软件实践，谁也不会容忍企业的软件发布是在一台私人机器上完成的。

在持续集成中，团队成员频繁集成他们的工作成果，一般每人每天至少集成一次，在保证质量的同时也可以多次。每次集成会经过自动构建（包括自动测试）的验证，以尽快发现集成错误。许多团队发现这种方法可以显著减少集成引起的问题，并可以加快团队合作软件开发²的速度。

3.1 环境准备

服务器端准备好 Game of life项目的git仓库，客户端需要：

- * Maven 2.x 包 <http://maven.apache.org/download.html>
- * JDK 6 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- * Jenkins <http://jenkins-ci.org/>

3.2 持续集成流程

持续集成的一个通常的简单流程如下：

1. 将已集成的源代码复制一份到本地计算机。（`git clone/pull`）
2. 修改产品代码和添加修改自动化测试。
3. 把修改提交到源码仓库。（`git commit/git push`）
4. 在持续集成服务器上基于主干（`master`）的代码再做一次构建（编译，单元测试，构建，打包）。
5. 在持续集成服务器进行测试（验收）

如果上述所有操作没有任何错误，没有人工干预，并通过了所有测试，我们才可以认为这是一次成功的构建。

¹<http://martinfowler.com/articles/continuousIntegration.html>

²<http://www.infoq.com/cn/articles/ci-theory-practice>

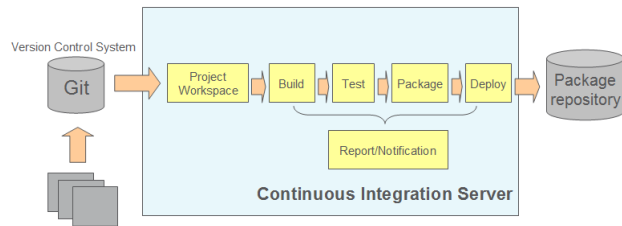


图 3.1: 持续集成流程

3.3 Maven

Maven是一个Java项目管理工具，就像Make对于c/c++项目。在Java的构建中和它“类似”的是Ant和Buildr。Maven比Ant的好处¹是：

- * 依赖包的管理只要写配置文件（pom.xml）就可以了，而Ant需要把第三方依赖的二进制包放在项目里。
- * 定义了标准集合，简单了项目的管理。

Maven主要还包括：

- * 一个项目对象模型（Project Object Model）
- * 一组标准集合
- * 一个项目生命周期(Project Lifecycle)
- * 一个依赖管理系统(Dependency Management System)
- * 用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。

3.3.1 安装Maven

装好JDK6，熟悉Unix环境，用Git bash安装Maven

```
$ cd /c # Windows C:/
$ tar -zxvf ~/Desktop/apache-maven-2.2.1-bin.tar.gz
$ mv apache-maven-2.2.1 maven
```

在系统中配好环境变量M2、M2_HOME、MAVEN_OPTS、PATH，如图3-2：
别忘了，需要重新打开bash后，配置才会起作用。

```
$ mvn --version
Apache Maven 2.2.1 (r801777; 2009-08-07 03:16:01+0800)
```

3.3.2 Maven仓库管理器：Nexus

不管怎么样，Java的包在编译时还是要下载下来的，在企业中，最方便的是架设一个管理Java的包的服务器。其中最著名的就是Nexus，它会缓存远程仓库的Jar包。如图3-3（源：<http://today.java.net/article/2010/01/04/maven-repository-managers-enterprise>）

¹所有的好处坏处都因人而异，请不要计较。

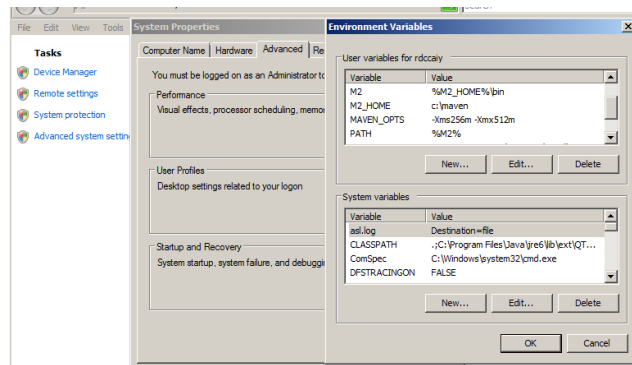


图 3.2: 系统中配好maven

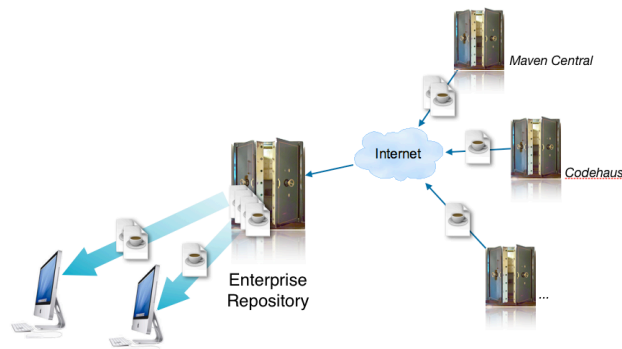


图 3.3: Maven仓库管理器

对于个人来说，你不需要安装，只要在~配置指向企业使用的Nexus服务器就好了，如

```
# ~/.m2/settings.xml
<settings>
  <mirrors>
    <mirror>
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>http://localhost:8081/nexus/content/groups/public</url>
    </mirror>
  </mirrors>
</settings>
```

3.3.3 第一个maven命令

在你的Game of life项目中，输入命令`mvn package`，观察命令行的输出，并且查看~目录的变化。

第一次执行还是比较慢的，大量的Jar包会下载到本地的缓存中，稍后会看见需要的依赖都在~中了。

3.3.4 体会两层缓存

实际上很容易理解，在个人机器上会有一个缓存，它在 ~，在Nexus服务器上是整个公司项目的缓存。

在你第二次编译时速度明显快了。

3.4 持续集成服务器：Jenkins

Jenkins是现在最流行也最有效的持续集成服务器，它的前身是著名的Hudson，后来由于Sun被Oracle收购以后，社区起了个新名字。

3.4.1 安装

不需要安装，直接在命令行启动。

```
$ java -jar ~/Desktop/jenkins.war --httpPort=7080
```

启动后就可以在你的浏览器中打开。<http://localhost:7080>，用7080端口只是为了防止可能的8080端口冲突。

3.4.2 安装Git插件

Jenkins的强大得益于它的插件系统（以.hpi结尾），大部分情况下，你要的插件早在社区存在了。

为了使用Jenkins和Git服务器相连，你要安装Git插件。你可以选择从Jenkins系统中下载Git插件（如果公司有防火墙的话需要配Proxy），也可以直接把它下载到本地后，拷到~下，别忘了重启Jenkins服务器。这样你就能看到Git选项了。

3.4.3 系统配置Maven

在系统中配置好maven目录，别忘了把自动安装选项去掉。



图 3.4: Jenkins 系统配置Maven

3.4.4 设置构建任务

新建一个任务game-of-life，选择自由风格（freestyle）。

1. 源码管理：配置好Git的远端仓库。
2. 构建触发器：设置轮询（Poll）策略：*/1 * * * *（每分钟一次）
3. 构建：用Invoke top-level maven targets构建，填上clean package
4. 构建后操作：Archive the artifacts选中后填上**/target/*.jar,**/target/*.war

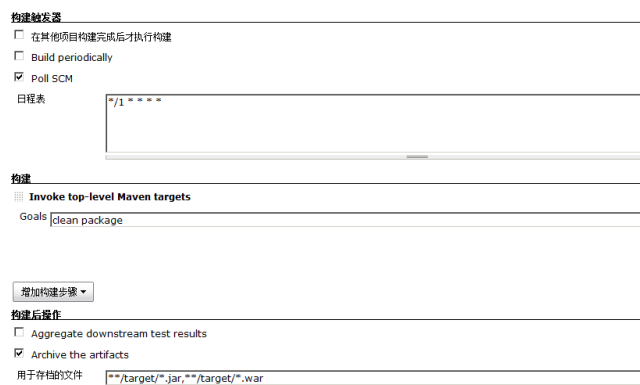


图 3.5: Jenkins game-of-life配置。

3.5 如何实施持续集成

首要一步是把服务器架设起来，然后把你的脚本放在任务中自动执行。经常你会发现本地好好的，到了持续集成服务器就不对了。这是很常见的问题，基本上都是环境的影响。不管怎么样，要让团队明白，持续集成服务器构建出来的产品结果才是有效的。

其次持续集成是有团队负责的，要把结果透明化得显示在公共地方（如显示在电视上）。要养成集成失败后立马修复的好习惯，因为只要有一次没人修，慢慢的就没人用了。



图 3.6: Jenkins监控显示屏（来自Extreme Feedback Panel插件）。

最后要养成持续提高的工作态度，随着越来越多的东西加入持续集成，速度会变慢、经常出错。要不断的找到薄弱环节（bottleneck），查找相关技术来不断提高。

3.6 相关知识

持续集成是十年前提出的东西，现在还有更多的好实践，建议看看持续交付一书。不仅产品代码需要持续集成，测试代码、基础设施的环境管理也需要持续集成。

在更多的持续集成上，要更进一步，达到持续交付。这并不一定代表你的产品一定需要在线运行，这更是一种软件开发的能力；你要很容易的部署到你的目标环境中，而且快速重现需要的配置。

3.7 课后练习

- 1. 装一些插件（Raditor, cobertura）体会一下。
- 2. 把JUnit的单元测试结果显示出来。

3. 查找构建输出在哪里。
4. 和其他团队成员一起修改代码，并提交到代码库中，看看变化。

3.8 小结

持续集成是敏捷软件开发的重中之重，一定要养成好的习惯。

3.9 参考阅读

- * Jenkins: The Definitive Guide: <http://www.wakaleo.com/books/jenkins-the-definitive-guide>
- * Maven实战: <http://www.juvenxu.com/mvn-in-action/>
- * 持续集成软件质量改进和风险降低之道: http://product.dangdang.com/product.aspx?product_id=20098017
- * 持续集成理论和实践的新进展: <http://www.infoq.com/cn/articles/ci-theory-practice>
- * Repository Management with Nexus : <http://www.sonatype.com/books/nexus-book/reference/index.html>
- * 持续交付: <http://www.continuousdelivery.info/>

第 4 章

如何写好Java程序

写程序对每个软件开发者是基本功，在企业中要考虑代码是给团队一起看的，因此质量的要求要更高一点。

首先你要有想把代码写好的意识，否则再怎么都说都没用。可以经常想想你的代码几年以后还容易读吗？

如果你在用一些比较新的语言那就很幸运，因为有很多有效的工具支撑，Java程序看起来就比C/C++容易一点。

在个人环境下，如果用Eclipse，可以安装下面的很多插件来帮你自动做很多事情。主要考虑：

1. 代码覆盖率，对Java来说，90%是一个不错的标杆，80%很勉强。
2. 为了实现代码覆盖率，当然需要用到单元测试，JUnit 3/4都可以，TestNG也不错。不用说，100%通过是必须的。
3. 代码风格和一些常见错误会有 Checkstyle和FindBUGs/PMD/CPD等工具。

怎么写好代码当然是更重要的事，你要熟悉基本的5大设计原则（SRP,OCP,LSP,DIP,ISP），理解这些以后，就可以学习如何用重构，如何用mock来隔离外部依赖写出好的单元测试。

测试驱动开发（TDD-Test Driven Development）是一个很有效的开发实践，要注重依据行为来驱动测试，而不是根据你的函数来写。

在服务器端，Sonar是个非常好用的工具，它把需要的质量信息都归纳起来了。一个产品架设好以后，代码的质量就一目了然了。

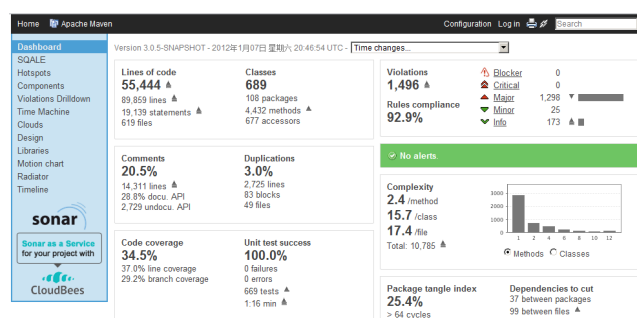


图 4.1: Sonar

4.1 环境准备

* Eclipse IDE和相关的插件EcmEmma

4.2 代码风格和编程规范

编程规范（代码风格）不需要自己去创立，继承别人现有代码就可以了，建议用checkstyle工具强制控制一些基本的。

下面列出一些常见的，不过重要的是理解，而不是事后检查。千万不要打印或制定厚厚的规范，没用的。可以翻翻[Java的编程规范](#)，下面列出几个：

- * 所有类的开头都要有Java文档的注释，而且写有用的内容。
- * 常量应该全部大写，单词之间由下划线分隔（例如，MAX_WORK_HOURS）。
- * 内部变量声明时应该对它进行初始化。
- * 不能有Magic数字，如if(input_length<8)。

4.3 安全代码

有些时候还要考虑一下写出安全的代码，这和产品的性质有关，web类产品会考虑得多一点。可以翻翻[Java的安全代码编程规范](#)，下面列出几个：

- * 要对输入信息（参数、特殊字符、SQL注入）进行检查验证。
- * 不要到处保存敏感数据（在数据库、文件等地方），如果必须要对它们进行保存，那就保存它们加密后的结果。
- * 要记录所有的或是有疑问的操作的日志。

4.4 单元测试

这个应该是最简单的，但还是发现好多人都不做，要养成习惯，每个公共方法都需要至少有一个测试用例。

单元测试是代码的一部分，要养成同时签入版本库的习惯。而且如果本地没有100%测试通过，也不允许提交代码。

良好的高度覆盖的单元测试是将来重构的保障。

4.5 代码覆盖率

有了单元测试后，就可以来看看代码覆盖率，建议在Eclipse中安装Ec1Emma，非常好用。

服务器端不用Sonar的话，cobertura不错。

4.6 重构

重构分好几层，这里主要考虑日常的重构，千万不要跟经理或项目经理要求时间，这个是对他们不可见的，当然团队可以了解你重构的内容。

不要把重构当成很复杂的东西，大多数是很简单的，在Eclipse中点几下就可以了。

最常见的重构有：

- * Rename（改名字）
- * Extract Method（提取函数）
- * Extract Interface（提取接口）

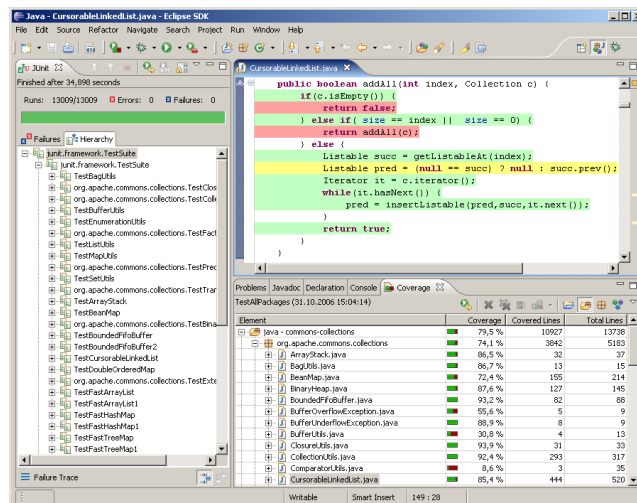


图 4.2: Eclipse插件Ec1Emma

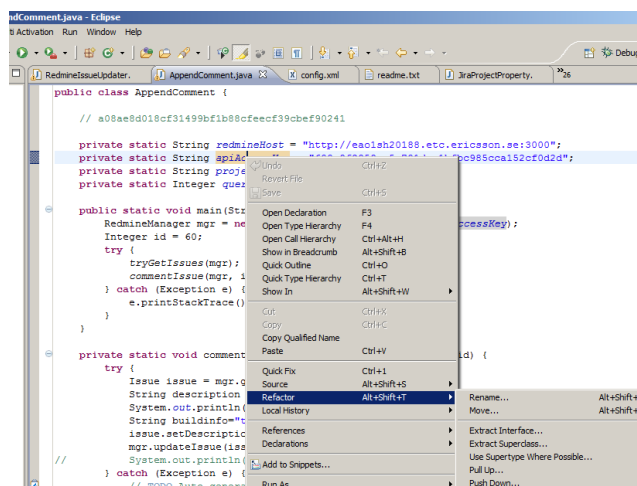


图 4.3: Eclipse中使用Refactor

关键的是看到坏代码，要养成习惯去重构，基础是有质量高的单元测试。

4.7 测试驱动开发 TDD

很多人认为测试驱动开发会浪费时间，如果写完代码再写测试不更快捷吗？反正把覆盖率达到就好了。

代码覆盖率只是TDD的一个好处而已，以上正确的一个前提就是代码实现的是正确的需求，很多很多时候，我们写了很多的代码，但有部分是不必要的。代码的产量不重要，重要的是质量。过多的不必要的代码是浪费。

而且覆盖率补上往往是一种借口，大部分的人代码写完了，认为任务就完成了，单元测试是额外的工作，至少我看到蛮多项目经理就这么认为的。

测试驱动开发会把你写代码的速度慢下来，但确保你写出来的代码是有用的、干净的。

因此TDD会帮助我们让代码一直工作，并且保持干净。

4.7.1 测试驱动开发的步骤

典型的步骤是这样的：

1. 根据任务，新增一个单元测试用例（TestCase）。
2. 编译代码，让刚刚写的那个测试编译通过，运行会失败，在IDE中显示红条(Red Bar)，因为还没有写实现代码。
3. 实现（implement）为了这个测试用例通过的代码，并编译通过。
4. 运行所有的测试，保证每个都能通过，在IDE中显示绿条(Green Bar)。
5. 重构代码，保持干净代码。

4.7.2 单元测试就是文档

如果你严格按照测试驱动开发方式来写代码，那么你的单元测试就应该是文档。

单元测试用例是为了实现某个功能而写的测试用例，因此它表明了代码的需求，读懂了这些需求，那就是很好的文档。

4.7.3 单元测试也要重构

很多人只知道对实现代码进行重构，忘了单元测试也是很需要的。

所以建议你看看你的单元测试，是不是明白你的代码的作用了呢，如果不是，请重构。

4.8 测试模拟（Mock）

既然叫单元测试，那么它的目标一般就是一个方法，因此需要尽量小步、细粒度的来进行测试。这样单元测试才可以快速完成。

在现实情况下，很多方法往往依赖于其他一些难以操控的东西，这时就需要进行隔离，如网路、数据库连接等，这就需要使用Mock模拟。

同时Mock也保证了代码的可测性。

Mock一般都是通过接口（interface）来实现的，有很多Java工具来帮你Mock。老字号的有JMock、EasyMock，现在比较不错的工具¹是Mockito、JMockit、PowerMock。后面解决了一些EasyMock不能对复杂方法处理的弱点，特别是静态函数。

记住工具不是主要的，但是好的工具确实是很重要的。

4.9 课后练习

1. 学习体会5大设计原则
2. 翻看代码规范，记住一些自己遗漏的地方。
3. 用TDD的方式实现一些功能，不要忘了重构和检查覆盖率。
4. 用JMockit工具来mock。

4.10 参考阅读

1. The Principles of OOD：<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
2. 重构：<http://book.douban.com/subject/4262627/>
3. 测试驱动的面向对象软件开发：<http://book.douban.com/subject/4910582/>
4. Ec1Emma：<http://www.eclemma.org/>

¹Stackoverflow 上的对这些工具的比较 <http://stackoverflow.com/questions/4105592/comparison-between-mockito-vs-jmockit-why-is-mockito-voted-better-than-jmockit>

5. 测试驱动开发全攻略@爬树的泡泡: <http://www.blogjava.net/briansun/archive/2005/07/31/8813.html>

第 5 章

需求管理和实例化需求

在一个产品开发中，需求是推动整个软件开发的源泉。产品经理制定需求，确定了方向，但是如果方向（需求）一开始就有问题，后面开发测试得再辛苦、再认真、再正确，对产品来说都是不合格的。

现在代码质量的检测（如覆盖率，静态代码检测等等），持续集成技术都已经积累了很多的经验，能很好得保证编码阶段代码本身的质量。需求的BUG比开发中的BUG更难发现。那有没有有效的类似的工程实践来解决需求的问题呢？

在本章，我们一起来学习需求管理中现在来看蛮有效果的一种实践 **实例化需求 (Specification by example)**。在下章同时会通过学习相关联的Cucumber软件来切身体会怎么将需求贯穿下去。

如果实施得好，相信你的产品的总体质量会上一个台阶。

5.1 环境准备

纸和笔就可以了。

5.2 需求的困惑

如果你做过开发，就知道软件开发的最大问题之一就是需求，而且它也很容易被作为替罪羊。在公司项目延迟和出大问题的最大借口（不过这也是事实），就是“需求不清楚”。

那把需求早点弄清楚不就行了嘛？听着挺容易，但要做好它却很难。

需求不清楚，不能开始项目，这一点大家都有共识。一种常见的解决办法就是：高级人员要对需求文档进行审核，完全通过后才能进行项目开发。但这实际上就是瀑布模型中的思路，大家已经知道它不大行得通，时间拉长了不算，早期把所有的需求都弄清楚也会变得纸上谈兵。

那敏捷迭代起来以后是否就好点呢。理论上会好点，因为需求在一个迭代中东西会少点，有机会理清一点。但就是因为一个迭代的周期短，在开完计划会议后（Planning meeting），团队会更愿意直接投入到代码开发中去，他们认为需求已经可以了；项目经理也觉得讨论需求会浪费点时间，我见过得很多人包括开发者都认为写代码才是干活。

这样的话，实际上往往到一个迭代的后面几天开始测试的时候才发现：测试人员、开发人员、产品负责人想的都不是很一样，但时间不够了，要不注册BUG，要不就是挪到下个迭代。这就是**技术债务 (Technical Debt)**的最大根源。

那是否有好的办法把需求质量有效得提高？

5.2.1 测试人员的工作文档

测试人员应该除了产品负责人对需求是最了解得，至少比开发者知道得更多¹。

再来看看现在的测试人员做了哪些事情，不同的公司可能不太一样，这里只是一种典型例子。

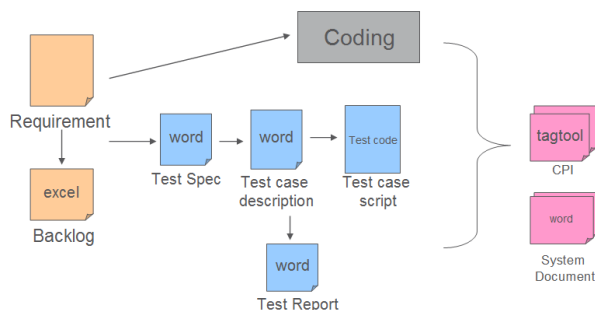


图 5.1: 测试人员的工作文档

得到“需求”后，首先可能要写的是**测试分析**：考虑一下这些功能大概的测试内容和范围。它存在的主要目的是为了让其他高级测试人员来把关，确保需求理解和相应的测试没有大的偏差，其中包括时间的估计。

紧接着是**测试用例**的描述：比较详细得用步骤的方式来阐述每一步测试。它很有可能是测试人员进行实际测试的主要文档，为了将来其他测试人员能重复测试。

如果运气好，敏捷实施不错的话，测试人员会被要求测试自动化，然后前一步骤的**测试用例**会被写成**测试脚本**。它来替换无谓的手工测试。再进一步，它还会在持续集成的服务器中被自动执行。

最后，一般都要求写一份**测试报告**：完成了多少测试用例等等。这个就是经理们经常有的一份文档，以此来作为项目实施总结报告。

上面的每一步看上去都很有必要，缺一不可。

但我建议你在旁观者角度来精益软件角度观察一下，有没有浪费²呢：

- * 这些文档都有哪些人读呢？作用大吗？3年后还能读懂吗？
- * 可以合并一些文档吗？只有一份行不行呢？

5.3 用实例化来解决需求的问题

解决需求当然有许许多多的办法，下面是几种常见的方式：

- * 从TDD（测试驱动开发）引申出来的**ATDD（Acceptance Test Driven Development: 验收测试驱动开发）**。就是把TDD对开发者的成功经验挪到测试团队中，让测试人员在项目中起主导地位。
- * **BDD（Behavior Driven Development: 行为驱动开发）**就是强调先搞清楚功能的业务需求，有它来指导后续的开发。
- * **实例化需求（Specification by Example）**：顾名思义就是要用例子的方式去阐述需求，这个概念是由Gojko Adzic提出的。

¹很可惜这一点很多测试人员都没有体会到，他们很多时候都是被动得测。开发者做出什么，他们测什么。

²精益软件开发的一大原则就是“消除浪费”，详见参考。

从本质上来说，实例化需求和ATDD、BDD包括其他的敏捷测试都是一个范畴，一样东西。但是实例化需求提出了更好得实践方式，减少了对工具的依赖，更容易被企业开发接受。

本章主要就是简单介绍实例化需求和实施的方法。

5.3.1 主要过程模式

在【实例化需求】一书¹中，Gojko提出了实例化需求的主要过程模式

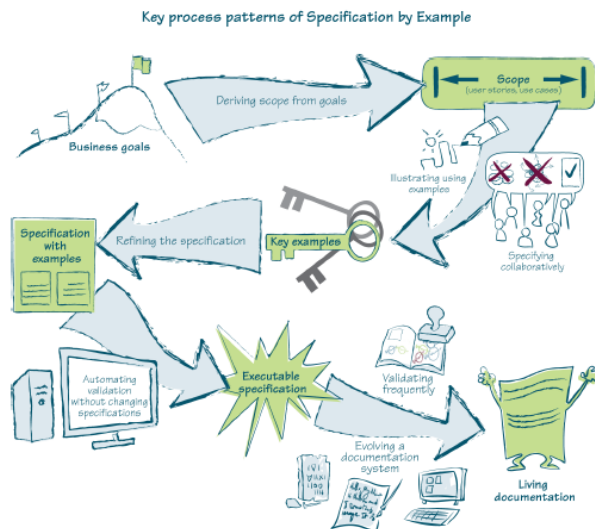


图 5.2: 实例化需求的主要过程模式

主要过程模式主要包括以下几个重要环节：

1. 从目标中获取范围：要一直牢记商业价值，为什么要做。很多时候执行项目时太关注怎么做了。
2. 用例子来协作探讨需求：例子能更好得把需求描述清楚，不能含糊。
3. 提炼需求说明：通过例子了解需求后就可以提炼出需要的需求说明。
4. 执行需求说明并自动化：需求说明如果能执行并放入到持续集成后，信息就不会过期。
5. 活文档：文档要长久，就必须容易维护，从需求说明中自动产生出的活文档是最有效的方式。

5.3.2 网上书店一个例子

让我们先从一个常见的例子来学习一下，这个例子也是Gojko常用的，我可能在一些环节稍微改变了一下。

假设某个网上书店为了提高用户的回头率，就提出了口号：**到年末，达到50%的用户的回头率**，这就是**目标 (Business Goal)**。一般是由CEO或最上层的市场经理根据市场决定的，他们关注的是战略方面的决策，也是整个产品开发的市場价值和背景。这一点经常被开发团队忽视，要让团队时刻知道项目的意义所在，这样在有冲突或对需求的理解产生怀疑时，就可以看看需求产生的具体原因，有时你可以想到不同的解决办法。条条道路通罗马—不过也要知道是去罗马对不对。

有了目标后，业务经理和领域专家们就可以从中提出一些要开发的项目**范围 (Scope)**，这个用头脑风暴 (Brainstorming) 的形式是最适当的，可能的产出点子如下：

¹图灵出版社已经引进出版，详见参考。

- * 给用户打折卡
- * 通过email主动推销优惠信息
- * 买书免运费
- * 限时打折
- *

这个阶段的关键是不要扼杀好点子，要集思广益。也不需要一开始就对点子进行细化，更多的是考虑投入产出比。在讨论充分的情况下，在会议的后期可以选定觉得最好的内容再进行推敲，以此来定出需求给研发部。

这一点就是关键，因为研发部一般和市场部不在一块儿（能在一块最好），通常是通过电子化的文档接受需求，这就一定要把需求说清楚，很显然用浅显易懂的语言来表示是最好的。先别看，你自己想想如果你来写会是什么样子。

好吧，让我们继续前面的例子。

假定**买书免运费**这个点子不错，现在就我们就可以试试用例子的方式来阐明需求。

- * 一个普通客户买一本书，免运费。

听着不太划算，业务经理认为要有一定数量，那就先定6本吧，然后加上5本的测试例子使它更容易理解。

- * 一个普通客户买6本书，免运费。
- * 一个普通客户买5本书，运费大于0。

蛮清楚的了，再来看看还有其他情况出现吗？

一个技术人员提出了运货范围，产品经理想了想竞争对手的情况，决定只给除西藏省，青海省的大陆地区免运费，差不多大家都清楚了。

- * 一个普通客户买6本书，送货地址到上海，免运费。
- * 一个普通客户买6本书，送货地址到西藏，运费大于0
- * 一个普通客户买5本书，运费大于0。

还有吗？想了想，对了，买书的人经常顺便买个小电子产品（或化妆品，哈哈），让我们加个U盘的情况吧。

产品经理想了想，物流也不会有损失，同意。

- * 一个普通客户买6本书，送货地址到上海，免运费。
- * 一个普通客户买6本书，送货地址到西藏，运费大于0
- * 一个普通客户买5本书，运费大于0。
- * 一个普通客户买6本书和一个U盘，送货地址到上海，免运费

反应快的立马想到了冰箱。

- * 一个普通客户买6本书和一台冰箱，送货地址到上海，免运费？？？

要不要免呢？产品经理觉得自己做不了主，问物流经理。负责物流的业务经理连忙说吃不消了，因为大件电子商品的快递费用很高，要亏本的。行吧，那就不免吧。

- * 一个普通客户买6本书和一台冰箱，送货地址到上海，运费大于0

这条和上面的U盘有冲突，需要解决。有人立马提出按重量来判断，大家一致同意。产品经理也定下了1公斤这个分界线。

- * 一个普通客户买6本书，送货地址到上海，免运费。
- * 一个普通客户买6本书，送货地址到西藏，运费大于0
- * 一个普通客户买5本书，运费大于0。
- * 一个普通客户买6本书和一个小于1公斤的U盘，送货地址到上海，免运费
- * 一个普通客户买6本书和一台大于1公斤的冰箱，送货地址到上海，运费大于0

现在我们经过**集体下定义（Specifying Collaboratively）**得到了一下**关键例子（Key Examples）**。明显得，这种奇异是最少的，一目了然。主要原因是最自然的业务描述，人人都明白。

有了这些例子，又可以回头去审阅最原始的需求“买书免运费”，就可以**提炼需求说明（Refine the Specification）**来得到完整的**实例化的需求说明（Specification with examples）**。

需求：买书免运费

提供读者买书优惠活动，买书超过（含）6本以上的而且重量小于1公斤，可以免费送货到除西藏自治区，青海省的大陆地区。

关键例子：

- * 一个普通客户买6本书，送货地址到上海，免运费。
- * 一个普通客户买6本书，送货地址到西藏，运费大于0
- * 一个普通客户买5本书，运费大于0。
- * 一个普通客户买6本书和一个小于1公斤的U盘，送货地址到上海，免运费
- * 一个普通客户买6本书和一台大于1公斤的冰箱，送货地址到上海，运费大于0

看上去很完美了，写入需求文档，放入Backlog，通知开发团队在下一个Spring中实现，争取早日上线。团队成员看了看，也觉得这次需求很清晰，也没问题，就承诺在这个Spring中完成。

孰料开发团队拿到这个需求准备实施没多久，架构师发现了个大问题：现在的数据库中没有“重量”一个字段，要加入这个字段的话，这个工作量会急剧上升。

怎么办呢？这是已经承诺好的需求，反悔还来得及吗？

软件开发中（生活中或亦如此）最重要的就是任何时候有问题，自己搞不定，不能完成自己的承诺时，都需要尽早提出来，和其他人一起商讨解决方案。

产品经理了解了技术难题后，认为这只是一个小的销售方案，没必要费那么多的开发时间。为了简化起见，大家一致决定只对图书免运费。

通过沟通，双方又都满意了。更新后的需求如下：

需求：买书免运费 提供读者买书优惠活动，买书超过（含）6本以上而且只含书的订单，可以免费送货到除西藏自治区，青海省的大陆地区。 关键例子： * 一个普通客户买6本书，送货地址到上海，免运费。 * 一个普通客户买6本书，送货地址到西藏，运费大于0 * 一个普通客户买5本书，运费大于0。 * 一个普通客户买6本书和一个U盘，送货地址到上海，运费大于0

这些是最重要的步骤，如果能实践它，已经可以得益颇多了。关于怎么去执行，那得结合工具来讲，我会在下章解释。

5.3.3 常见问题

在企业推动实例化需求，讲到这个例子时，经常会碰到这些问题：

6件可配吗？：例子中提到6件免运费，显然这个在系统中要可配的，不能在产品中硬编码（hard code）。但是这个需要在例子中讲清楚吗？

“否者我的开发团队又要说我需求没讲清楚，不过我总觉得这点意识他们应该有的？” ，一个有着很多痛苦经历的PO问道。

没有一定的说法。在这个案例中，建议没有必要把**免运费的数目可配**这个要求在例子中写出来。但是如果自己的团队这方面能力不够，提醒一下，或者放在基本要求中都是可以的。记住！！最重要的是沟通，把需求澄清出，不要有歧义。

这不是Waterwall吗？：这是一个简单的例子，看上去很快就弄明白了。但是在实际中，有太多的需求要澄清呀？

”这样看上去和以前没有两样，还是要花好多时间来搞清需求，只是快了一点而已？”

不是的，在实际运作中，这是一个迭代式得不断澄清的过程，可以把最重要的先澄清。这里面还有一个很重要：就算在优先级最高的需求中也有不重要的小功能，这个用例子的方式很容易体现出来。

和用例有区别吗？：很多人初次接触这个时总觉得和用例没区别。

可能，但是我很少看到好的用例，用例很多时候关注在技术实现的步骤上了，所以有很多不必要的技术细节，不太适合产品经理来阅读。这儿我们要求的是一份大家都容易读懂最少歧义的需求。

当然不排除你的用例很容易读，没有冗余，那往往就是实例化了。

5.4 如何实施

以前在估计时间时，团队很容易用含糊的原因解释。敏捷实施中又要求团队来估时间，常见的毛病就是：

我们需要1000人时（manhour），300人时是准备环境，400设计加开发，300人时是测试。

现在用实例化需求后，很容易得针对每个例子来估时间，PO也可以根据时间的代价来取舍。

认可了这种实践，实施起来也相对容易一点，下面列出几个要点（需要不断学习）。

5.4.1 循序渐进和现有流程的结合

我们可以不用改变现有流程的方式把实例化需求循序渐进地开展起来，这一点在企业中很重要，大的改变都很费周折。

在**测试分析**阶段，我们可以把简单的需求列出来，加上理解的一个最重要的例子就可以了。可以结合迭代式的方式，优先级高的先多花时间来写**测试用例**。

当然在写**测试用例**时，我们可以继续前面的格式，只是多加几个例子，并且持续地提炼需求说明，使得越来越清晰。

在下章中介绍如何把**测试用例**直接变成可测试的**测试脚本**。

这样子，你的现有流程就不需要改变，如果你已经有现成的框架来做**测试脚本**，那就继续用下去吧，只要需求明白了，什么都好解决。

5.4.2 贴在墙上

把需求贴在墙上可视化一点也是非常有效果的，它可以减少浪费。企业最大的毛病就是会多，每个人都是认为自己的会重要，这是很浪费时间的。

完全不一定每次审阅都要去参加，只要我们把需求贴在墙上，团队就随时随地得可以了解最新的进展。随手拿个即时贴提建议：时间估计，架构的影响…

如果对某个需求需要讨论了，召集相关人（千万不要整个团队）在小房间搞清楚，然后立即更新需求。

因为我们现在用上了自然语言的实例化的需求，任何时候团队成员应该很容易读懂，而且没有偏差；如果有，说明还不清楚。

5.5 相关知识

- * 了解ATDD, BDD
- * 阅读从用户体验中挖掘需求一种方式“体验设计7日谈之一：关于软件的体验设计”：<http://www.infoq.com/cn/articles/xzc-experience-design-software>
- * 了解特性注入 (Feature Injection)：<http://www.infoq.com/cn/news/2009/05/feature-injection-comics>

5.6 课后练习

- * 试着把身边最近工作过的一个需求用这种方式体会一下。

5.7 小结

实例化需求是一种很棒的协作探索需求的好办法，它强调了无时无刻地沟通的必要性，还有就是用例子来讲述需求更容易理解。但是要用熟练了还是很有难度得。

5.8 参考阅读

1. Book: Specification by example. <http://manning.com/adzic>
2. Specification by example: <http://specificationbyexample.com>
3. 精益软件开发: <http://zh.wikipedia.org/zh/%E7%B2%BE%E7%9B%8A%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91>
4. 实例化需求中文书: <http://www.ituring.com.cn/book/837>

第 6 章

用Cucumber来实例化需求

在上一章中我们已经了解如何把需求用实例化的方式弄清楚，现在继续考虑怎么用通用的记录下来，并且去执行它。

这方面实际上有很多工具，在本章，我们会通过学习一种现在最有效的Cucumber软件来切身体会怎么将需求贯穿下去，其他的可以看相关知识。

开始前，再强调一下，滥用工具还不如不用，Cucumber也是如此。

6.1 环境准备

* Windows下的Ruby <http://rubyinstaller.org>

6.2 Cucumber 简介

Cucumber（英文：黄瓜）（官方网站是<http://cukes.info/>）是一个实例化需求的极佳实现伴侣。它是基于Ruby的开源测试工具，得益于Ruby便于创建和使用DSL的特性，它可以通过自然语言（文本文字）来描述需求（业务层），并通过关键字驱动和正则表达式匹配告诉去做哪些事情（驱动层），在运行自动化测试结束以后，还会给出详细的报告。

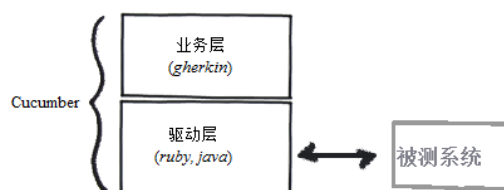


图 6.1: Cucumber的架构

下面就是一个加法例子的需求描述，Cucumber文件以`.feature`结尾。

```
# 加法 adding.feature Feature: Adding In order to avoid silly mistakes As a math idiot I want to be told the sum of two numbers
```

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
```

```

Scenario Outline: Add two numbers
  Given the input "<input>"
  When the calculator is run
  Then the output should be "<output>"
Examples:
  | input | output |
  | 2+2   | 4       |
  | 98+1  | 99      |

```

这就是业务层，它和上一章最后的例子很像。功能标题后面是它的简要描述，然后是详细的例子。

建议好好读读Dan North的文章：什么是故事<http://dannorth.net/whats-in-a-story/>。现在让我们试着来运行它，看看会怎么样。

6.3 安装

在Windows上，RubyInstaller提供了ruby的环境，下载安装包（如rubyinstaller-1.9.3-p0.exe），运行即可，别忘了把“Ruby放入PATH中”的选项选上。

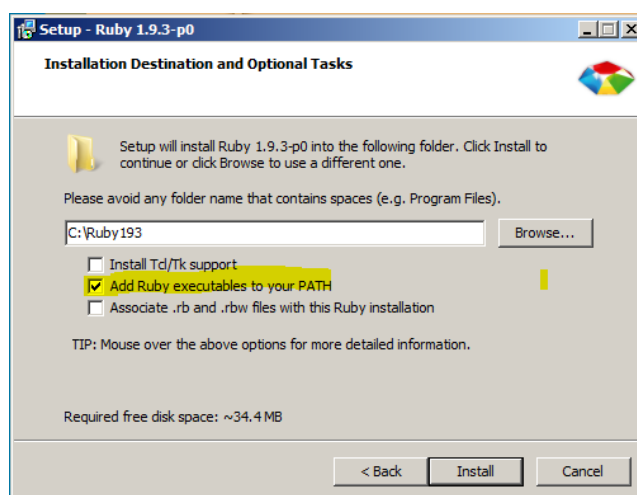


图 6.2: Windows平台安装Cucumber

```
$ gem install cucumber # 如果需要配代理, -p http://: $ gem install rspec #
cucumber 需要
```

6.4 运行Cucumber

一旦Cucumber装好了，我们就可以使用 `cucumber` 命令来运行feature文件。

feature文件放在**features**目录下，如果cucumber命令后不跟任何东西的话，那么它会执行所有的.feature文件。如果我们只想运行某一个.feature文件，我们可以使用命令 `cucumber features\feature_name`

```
$ cucumber features/adding.feature
Feature: Adding
  In order to avoid silly mistakes
  As a math idiot I want to be told the sum of two numbers
```



```

Scenario: Add two numbers      # features\adding.feature:3
  Given the input "2+2"        # features\adding.feature:4
  When the calculator is run    # features\adding.feature:5
  Then the output should be "4" # features\adding.feature:6

Scenario Outline: Add two numbers  # features\adding.feature:8
  Given the input "<input>"        # features\adding.feature:9
  When the calculator is run      # features\adding.feature:10
  Then the output should be "<output>" # features\adding.feature:11

Examples:
  | input | output |
  | 2+2   | 4      |
  | 98+1  | 99     |

```

3 scenarios (3 undefined) 9 steps (9 undefined) 0m0.046s

You can implement step definitions for undefined steps with these snippets:

```

Given / the input "([ ]*)" $/ do |arg1| pending # express the regexp
above with the code you wish you had end ...

```

你就可以看到它被正常执行了，发现了3个场景（scenarios），9个步骤（steps），这不就是我们需要的测试吗！！

让我们来解读一下吧。

6.5 业务层: Gherkin语言

业务层实际使用的是[Gherkin语言](#)，Cucumber是一个解释程序，它用来执行解释 .feature 文件里业务描述，它的关键字就是“Given”、“And”等等这样的字眼。

一个常见的Cucumber文件描述分为 **Feature（特性）**、**Scenario（场景）**、和**Step（步骤）**。让我们再来看看上面的例子：

1. **Feature: Adding:** 这是标题，每一个feature文件以关键字**Feature**开始，且紧跟着一个冒号和一个简单描述。
2. 在上面，你发现接下来的几行描述不会被解析，纯粹是描述用的（当然也很重要），强烈建议你照**用户故事（User Story）**的方式去写。
3. **Scenario: Add two numbers:** 关键字_Scenario_后面紧跟一个冒号和一个对应该场景的描述，也是简短的一句话。
4. 后面的以**Given/When/Then/And/But**开头（这些也是关键字）的都是步骤（步骤后面不需要跟冒号），用来阐述到底要的是什么样的需求。
5. **Scenario Outline: Add two numbers:** 关键字Scenario Outline，和Scenario不同的是它是支持表格的形式。
6. **Scenario** 和 **Scenario Outline**提供了特性的多个场景，可以出现多次。**Scenario Outline**提供了表格的形式，适合批量数据的处理。

具体怎么连到被测系统就靠驱动层了。

6.6 驱动层

驱动层的主要目的就是把业务层中的数据（如上“2+2”，“加”）提取出来，通过于应用程序进行交互，最后把返回结果和预期的值（“4”）进行比对，得出测试结果。

Cucumber的驱动层可以用Ruby，Java和其他语言来支持，很多时候语言的选择主要依赖团队的兴趣。这里以Ruby为例，当然不用担心，因为介绍的例子不需要很多深奥的知识。

在Cucumber中，第一次运行后，它会给出Ruby代码的模板，就是：

```
Given / the input "([ ]*)" $/ do |arg1| pending # express the regexp
above with the code you wish you had end ...
```

如果对脚本或Linux比较了解的话，很容易看出，这是一个正则表达式。

在features下面建一个step_definitions目录，把上面运行的代码模板片段写入calculator_steps.rb文件中，并且把pending那一行用#注释掉，再次运行cucumber，就很顺利通过了。

```
3 scenarios (3 passed) 9 steps (9 passed)
```

真实情况下，我们要写些代码匹配到关键字处理后，想办法传递到被测的系统，并和设定的期望值匹配来确定测试结果。

6.7 常用的目录结构

常用的目录结构组织方式是

```
$ find calculator calculator/ calculator/feature.html calculator/features cal-
culator/features/adding.feature calculator/features/division.feature calculator/
step_definitions calculator/step_definitions/calculator_steps.rb
```

1. features下面按功能放置各个业务。
2. step_definitions存放驱动层的脚本。

6.8 继续网上书店的例子

Cucumber虽然上是支持多语言包括中文¹的，但还是建议关键字用英文来写，以免其他工具的不支持。

用Cucumber重写的话，下面是一种方案。

```
# book.feature Feature: 买书免运费 提供读者(不管普通还是VIP客户) 买书优
惠活动, 买书超过(含)6本以上的, 可以免费货到除西藏省, 青海省的大陆地区。
Scenario Outline: Given 一个客户买了 And 买了 的东西 When 选好 Then 看见
```

Examples:

几本书	其他类别	送货地址	运费为0
6	n/a	上海	yes
6	n/a	西藏	no
5	n/a	上海	no
6		上海	no

它对应的用Ruby实现的驱动层的代码就可以类似：

¹就算中文，也建议用UTF-8格式。

```
def onlinebookstore(book_number,other_order_category,delivery_address) # 写
代码发往被测系统，得到运费 return 10 #模拟运费10元 end Given / 一个客户
买了 (+) /do|number|@book_number = numberendGiven/买了(.*?)的东西/ do |category|
@order_category = category end When / 选好 (.*?) /do|address|@delivery_address =
address@result = onlinebookstore(@book_number,@order_category,@delivery_address)endThen/看见(yes|no)/
do |expected_output| if(expected_output == 'yes') @result.should == 0 else @re-
sult.should == 1 end end
```

所以运行后，你应该能够从输出结果中看到3个Scenario测试通过了。

4 scenarios (1 failed, 3 passed) 16 steps (1 failed, 15 passed) 0m0.076s

如果你上过TDD了，就知道现在我演示的只是模拟的实现，现在就是驱动你把驱动层的代码写好使他被运行通过。

怎么样，有点感觉了，多多练习吧。

6.9 常见问题

6.9.1 我们的系统没有接口能够被这么（或容易）测试得？

好问题！！上面这个网上书店系统

- * 可能使用Flash写的，你根本没法用脚本填充数据，然后得到结果。
- * 或者就算是用HTML5写的，但是调用它也是很费时间周折的呀。

是的，没错测试是费时的，但再想想背后的原因！

这就是没有测试驱动开发的后果，任何技术的设计不仅要考虑实现，也要考虑测试，应用程序是否能够被自动化测试是一个衡量软件开发水平的重要标志。否者要架构师干嘛？

6.9.2 Cucumber用起来了，也自动化了，但是没人看？

正常。不要为了Cucumber而Cucumber；不要为了自动化而自动化。

首先要理解实例化需求，然后再用工具去支持，不能本末倒置。

另外把结果变成网页或者贴在墙上都是不错的建议，试试下面的命令吧？

```
$ cucumber format progress format html out=features_report.html
```

看看HTML的输出，你也可以自己定制你的报告。

6.10 相关知识

- * [FitNesse](#)也是ATDD中很著名的一种工具，在Cucumber前占有很大的地位。
- * [敏捷测试的思考和新发展](#)

6.11 课后练习

1. 把网上书店的例子，尝试用实例化需求说明的方式来描述清楚，并写成Cucumber的格式。
2. 阅读参考书，了解更多的Cucumber知识。
3. 了解Gherkin语言的详细内容，如tag，并结合Cucumber去执行。
4. 看看如何能够实施Cucumber，使它能够整合到持续集成中去。

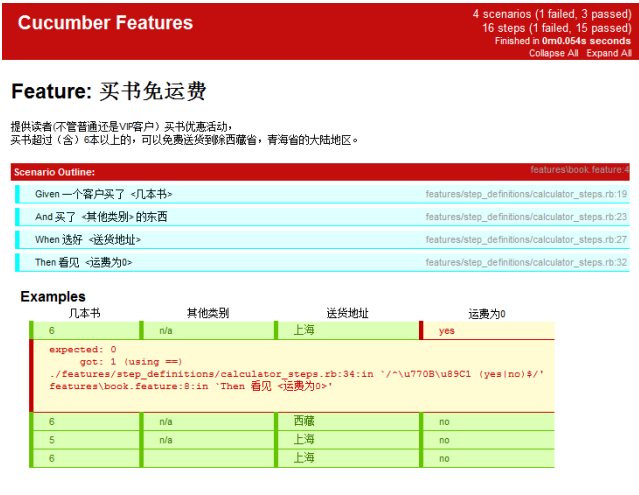


图 6.3: Cucumber的架构

6.12 小结

Cucumber也只是一种工具，如果不理解实例化需求说明的真正意义，它会被用得很累，好自为之。

6.13 参考阅读

1. Book: Specification by example. <http://manning.com/adzic>
2. Specification by example <http://specificationbyexample.com>
3. Cucumber <http://cukes.info>
4. Gherkin语言: <https://github.com/cucumber/cucumber/wiki/Gherkin>
5. Book: The Secret Ninja Cucumber Scrolls: <http://cuke4ninja.com/>

第 7 章

项目实践：Game of life

如果你是在自学，那么请用你自己的项目结合前面学到的东西练习一下，最好和几个同事一起来做。

如果你参加了公司的培训，这会是紧张和忙碌的一天。

作为团队，你们会收到产品负责人布置的任务，请用学到的东西去实践，记住团队合作！

练习的项目是取自John的[Jenkins指南](#)中的演示代码-[生命游戏 \(game of life\)](#)

7.1 常见问题

无论你多么强调团队合作的重要性，一碰到实际的项目，往往什么都忘了，下面是常见的两个毛病。

7.1.1 过度承诺 (over commit)

大部分人觉得任务很简单，一会儿就做完了，可往往一天都没完成，完成的最差¹的是0。

而且一般都不会主动和产品负责人去沟通，提前汇报风险，使得产品负责人早作准备。都要在产品评审会议的时候才会说不行了，太晚了。

7.1.2 没有团队精神

讨论任务的时候还有点团队精神，但是领完任务后又立马单枪匹马的干活了，最多是两人一组。一般一个两人的小组忘了他们是一个大团队，没有想到要帮助别人，关心其他团队成员。

7.1.3 缺少计划和工作方式的制定

没有规矩，不成方圆。很多情况下，团队常常匆忙开始开发工作，而忽略了必要的团队计划和工作方式制订。主要原因是时间很紧，需求很多，环境不熟悉。大家在如此巨大的压力下，往往会选择马上开始动手，而忘记了‘磨刀不误砍柴功’的古训。

这样的后果是，一旦有任何意外发生，团队会陷入混乱状态，完全没有办法给出相应的应对措施。同时，由于缺少统筹安排，团队分工不明，会导致不必要的浪费。

¹不过这个团队可能收获也是最多的。

7.2 小结

怎么样，经过一天的实战，体会到了企业开发的皮毛了吗？这还刚刚开始，享受你的企业敏捷开发之旅吧。

附录 A

培训示例模板

如果在公司培训，这是一个模板，建议根据实际情况修改。

A.1 时间安排

所有的培训都在二楼的哥德堡培训室，请带好自己的电脑，准时到达。

第一天 09:00-09:05 培训介绍
09:05-10:35 企业组织结构介绍、敏捷、Scrum
10:35-11:30 布置练习，在白板前实战学习。
13:00-16:00 Git入门，代码审阅与Gerrit。
第二天 09:00-09:15 站立会议，练习任务点评。
09:20-11:20 持续集成 (java, maven, jenkins)
13:00-16:00 Java质量, Game of life介绍
第三天 09:00-11:20 实例化需求
13:00-16:00 Cucumber
第四天 09:00-10:00 Game of life项目任务和计划会议
10:00-15:00 团队任务
15:00-16:00 任务审阅和回顾
16:00-16:30 结束，颁发毕业证书，拍照留念

A.2 培训的材料

培训的机器是Windows平台

- * 所有的培训Slides: <http://server/download/ppt>
- * 所有的需要安装的软件在: <http://server/download/software> (第一天用U盘拷贝到培训员工各自桌面)

这本手册只是一个参考，你需要花更多的时间去了解熟悉它们，4天的学习会很有意思，要有激情，否则什么都学不到。

A.3 培训负责人准备工作

- * 培训房间，网络，老师安排，会议通知，手册打印。

- * Gerrit服务器和Jenkins服务器和tomcat服务器
- * [生命游戏 \(game of life\)](#) Git仓库复原

A.4 反馈

作为企业中的一员，学会给别人反馈是第一要务，给这些课提些建议吧。谢谢。

附录 B

企业版本控制的改革：走向Git

在传统企业中，版本控制系统大都采用ClearCase或SVN。特别是ClearCase在早期提供了强大的企业应用的功能，我们部门也很早使用了。而且长久以来，在它周围建立了无数的应用和流程，同事们都觉得它是必须的了。

然而随着敏捷和开放的推动下，在有些产品用ClearCase开发碰到了很多局限，比如在家上班，远程团队开发。有人开始想到是否可以引入其他工具来解决，不过在大型企业要改变这种基础的工具是很难的。

我就想介绍一下我们是如何一步步地走向Git的。

特别声明：本文原为图灵社区活动“[唤醒你心中的布道师](#)”而写的文章：[企业版本控制的改革：从ClearCase到Git 我的布道之旅](#)，这里有所编辑。

这里想说的主要是如何在需要的时候推动技术的变革，而不是探讨技术的好坏。每个技术都有适应的场所，请勿生搬硬套。

关于版本控制的选择，也可以看看Martin Fowler写的[版本控制工具\(english\)](#)

B.1 了解最新技术-分布式版本控制（DVCS）

在推动技术改变的时候，首先要了解最新的技术状况，别学了一个旧了过时的了。

我们使用的主要是ClearCase，开始考虑这个转换的时候是在2009年初，SVN是第一个考虑的对象，因为它在开源中用的最多，[sourceforge](#)和Eclipse的很多项目多用它，但我总觉得缺了点什么。

恰好我有个同事提到SVN和ClearCase都是集中式的，推荐我看看一个分布式版本控制工具：Mercurial，说实话听了介绍不是很懂，没有眼前一亮的感觉。聊了一下，感觉和SVN的分支没有多大区别，何况DVCS还需要两层提交呢。

同时我也了解到还有其它的分布式版本控制工具Git，Bazaar可供选择。

不管怎么样，我了解到这块领域有了最新的技术，它或许能解决我们的问题（要不时地问问自己为什么）。

B.2 尝试在日常中使用分布式版本控制

为了尽快了解DVCS，我决定要在日常的开发中用用它，实践它，尽快地掌握它的关键。

由于同事对Mercurial很熟，我就踏踏实实地用Mercurial尝试了两个星期，不懂就问，顺便查查资料去比较一番。

DVCS真是很神奇，很好用，特别对我的胃口，感觉DVCS天生是为软件开发用的。

在同一时刻，我又比较深入的看了看其他的系统如Git，发现Git的生态圈更好一点。在软件开发中，生态圈会决定将来这个工具的发展趋势。

- * 如Eclipse插件开发邮件中开始讨论并决定用Git替代svn。
- * Git有很多的书可供选择（如 [ProGit](#)），[git在线网站](#)的内容也极其丰富。
- * [github](#)也漂亮得提供git的支持。补充一下，那时候[bitbucket](#)和[github](#)还在同一个水平线上。[google code](#)也还不支持git，只有Mercurial和svn。

通过这些实践和了解，发现DVCS-Git很适合我们所在的部门的企业产品软件开发。

B.3 宣扬和推广分布式版本控制

要在企业中换一个版本控制工具难度非常大，所以必须要布道，我采用了下面的方法：

1. 每月我们都有固定学习新东西的时间，我就推荐了Mercurial、Git两个课程，让大家一起来学习，了解它。顺便我要看看开发者对它的接受程度，有趣的是，水平越牛的人越是喜欢它，纷纷过来问什么时候能在产品开发中用上Git。
2. 除了开发者，管理者和其他的使用者（配置管理的同事）的想法也很重要。我经常抓住机会和这些人聊DVCS，聊Git，给他们介绍，看看他们有什么想法。当然他们有时候会不同意我的观点（有强势的，有委婉的），我就试图去说服他们，并从中挖掘出推动这个变化的关键因素。

慢慢得我就得到了很多如何推动这个变化的关键说服点，这个每次情况都不会一样。

B.4 详细研究版本迁移

开发者想使用分布式版本控制的呼声越来越高，管理者也开始认真考虑了。

在企业中，改变所需要的研究评测报告是必不可少的了，这也给了我一次重新认识集中式和分布式版本控制的过程，我花了更多的时间去想这个改变对企业带来的好处。实际上开发者有时候不会考虑到整个软件开发的所有方面，如安全，持续集成等等。报告的大致框架是：

- * 现在问题是什么？
- * 什么是DVCS，Git是什么？
- * 能改变什么？带来的好处？
- * 如果变化，计划是什么？

这一期间，使我静下心来更详细地了解了Git对企业可能的影响（有好的，有坏的），并制定了相应的对策。

B.5 开始在小范围实施

技术改变需要耐心和机遇，机缘巧合，迁移到Git的建议比较顺利地被管理层接受了。

然后就是要去认真真地实施了，这不是一个小问题，既然是软件开发，来不得半点的马虎，细节决定一切。而且实施得好坏还涉及到产品开发的正常运转。

企业中一般会从小范围开始实施，成功了才推广，下面是我们的一些实践。

- * 我们开始用[gitolite](#)作为Git服务器，架好试验平台，在一个小项目中开始尝试。

- * 人手一本Git的书，安排Git入门培训，提高驾驭Git的能力。
- * 不断收集资料，提高对Git的认识。

还好基本上没有出大的差错，虽然有蛮多技术难点的，不过最后都解决了。通过小范围的使用推广，我们的技术储备也加强了（特别是配置管理的人），对下一步的全面实施更有信心。

B.6 推广、并引入Gerrit做代码审查

早期我们用的是gitolite来架Git服务器，它很不错。不过后来发现Gerrit更好用，后来就切换过去使用了。这一点很重要，要不断探索这些新技术，争取在大规模推广前，用一个最适合的工具，否则一用上，在企业中就很难改变了。

Git开始在更多团队和更多产品中使用后，我们不断加强知识的培训，而且把相关的系统（如持续集成）都迁移到Git上去。一切都还不错，只是Git比想象中还复杂一点。

因为Gerrit有很强大的代码审查（code review）功能，不久以后这个功能也用上去了，代码提交的质量一下子上了一个档次，这是开始推动Git变革时没有想到的。

B.7 小结

技术的变化不是那么容易得，需要天时、地利、人和，缺一不可。如果你有什么好建议，欢迎一起探讨。