InfoQ Maven专栏（四）——基于Maven的持续集成实践

February 4th, 2011

Martin的《持续集成》

重温大师经典 — Martin Fowler的《持续集成》 - 滕云

相信很多读者和我一样，最早接触到持续集成的概念是来自Martin的著名文章《持续集成》，该文最早发布于2000年9月，之后在2006年进行了一次修订，它清晰地解释了持续集成的概念，并总结了10条实践，它们分别为：

- 只维护一个源码仓库
- 自动化构建
- 让构建自行测试
- 每人每天向主干提交代码
- 每次提交都应在持续集成机器上构建主干
- 保持快速的构建
- 在模拟生产环境中测试
- 让每个人都能轻易获得最新的可执行文件
- 每个人都能看到进度
- 自动化部署

原始文章距今已10年有余，这在软件行业中算是很长的时间了，但我们都能看到Martin总结的这些实践依旧闪耀着光芒，依旧有很多团队在努力实践它们并得到了丰厚的回报，当然也有很多团队因为各种原因拒绝实践持续集成从而无法体会到个中好处。

从这10条实践中我们能找到很多流行开源工具的影子，例如版本控制工具cvs、svn、git，自动化构建工具 Gradle Maven、Ant，自动化测试框架JUnit、TestNG，以及持续集成服务器CruiseControl和Hudson等等。其实不论 Jenkins 你是否实践持续集成，单独使用这其中的很多工具都能发挥极大的价值，持续集成的一大意义在于它引入了一个有效的流程，能让这些工具有机融合，并相互促进。 关于持续集成还有一本获得Jolt大奖的图书，名为《持续集成——软件质量改进和风险降低之道》。但无论是Martin的文章，还是这本图书，都没有阐述使用Maven作为自动化构建工具实施持续集成的细节。本文旨在介绍一些基于Maven实施持续集成的实践，希望这些经验能从具体处帮助到读者。

架设私有Maven仓库

Martin的文章并没有涉及到依赖管理的内容，但在Java的世界中，依赖管理是开发人员不得不面对的问题。无论是外部的开源类库依赖，还是项目内部的模块间依赖，都需要有效地管理。可以说依赖管理是持续集成核心的内容之一。Maven通过其依赖管理机制和随处可用的中央仓库有效地解决了这个问题，用户只需要在POM中声明项目所需要的依赖，Maven就能在构建的时候自动从仓库解析依赖。

不过仅仅这样是不够的，我们知道，持续集成的最大好处在于降低风险，简单地来说就是尽早暴露问题，能让开发人员及早发现并修复，从而降低修复成本。可是，如果每个人都从中央仓库重复下载依赖，这是非常耗时的，集成的反馈周期肯定会延长。我已经无数次听到有人抱怨“Maven在下载整个Internet！”。构建要快！持续集成反馈要快！Maven你不能拖慢这个流程。

幸运的是开源世界有很好的解决方案，只要使用Maven仓库管理器软件如Nexus建立一个私有的Maven仓库，问题就能迎刃而解。原理很简单，这个位于局域网内的Maven仓库能够代理所有外部仓库，从而避免所有人从Internet重复下载依赖文件。这样Maven解析依赖的时候仅限于局域网，构建速度就大大地加快了。例如大家都需要使用junit-4.8.2.jar，当第一个人向私有仓库请求的时候，私有仓库从中央库下载并缓存下来，假设耗时10s，之后其他人需要junit-4.8.2.jar的时候，私有仓库直接使用缓存的文件，这个耗时可能就是1s。如果有100个开发人员使用该文件，那节省的时间就是 100 * 10 – (10 + 99 * 1) = 891s ，实际情况中依赖的数量可能会是成百上千，那节省的时间就变得非常的可观。

也许有人会说，我也完全可以将项目依赖加入到版本控制中，这一点甚至在《卓有成效的程序员》中都被明确提及，在该书第5章的”DRY版本控制”一节中，Neal Ford有这么一段话：“所有用来构建项目的东西都应该被放入版本控制，包括二进制文件（类库，框架，JAR文件，构建脚本等等）”。作者进一步解释了其目的，这么做能够保

证项目不受外部因素影响（如依赖版本变化，甚至丢失），保证构建的稳定，作者也同时提及了一般版本控制工具处理二进制文件的性能问题。抛开这条结论性的实践，仔细考虑其目的，我们就能发现，私有Maven仓库同样能保证构建的稳定，而且能避免版本控制工具处理二进制文件而造成的潜在性能问题。所以，我斗胆说一句，Neal Ford所提的这条实践OUT了！

私有Maven的仓库的意义还不仅限于此，结合自动化部署和Maven的SNAPSHOT机制，它能大大促进项目集成的效率。

在模块化的开发环境中，大家各司其职，专注于自己所负责的模块，持续集成的规则是，在往版本控制提交代码前，需要先保证本地构建没有问题，那一般的做法就是更新所有模块的代码并构建。可是，真的需要构建那些其实你并不怎么关心的模块么？且不谈一旦构建他人代码时出错，你往往会不知所措，这种做法同时也增加了本地构建的时间。

Maven有SNAPSHOT版本的概念，其目的就是让你能够构建一个临时的版本，供团队他人使用，这样他们就不必在代码的层次关心自己的依赖。于是私有Maven仓库就充当了一个中介的作用，而持续集成服务器就多了一个职责，每次它成功构建一个模块，都应该将该模块的SNAPSHOT版本发布到Maven仓库中。现在，大家就不用去构建别人的代码了，Maven能自动帮你从私有仓库解析下载依赖的最新SNAPSHOT（使用**mvn**命令的**-U**参数强制更新）。注意，除了持续集成服务器外，任何其他人都不应该发布SNAPSHOT版本到Maven仓库，因为只有持续集成服务器的环境是可信任的，你能在本地成功执行**mvn clean install**并不代表持续集成服务器上该命令能成功，由于每个人的本地环境各有差异，因此集成的成功与否应当以持续集成服务器为准，而只有集成成功后，SNAPSHOT才可以被部署到私有仓库供他人使用。

鉴于上述的原因分析，我认为在基于Maven的持续集成环境中，再怎么强调私有Maven仓库的重要性都是不为过的。

正确的集成命令

在持续集成服务器上使用怎样的 mvn 命令集成项目，这个问题乍一看答案很显然，不就是 **mvn clean install** 么？事实上比较好的集成命令会稍微复杂些，下面是一些总结：

- **不要忘了clean：** clean能够保证上一次构建的输出不会影响到本次构建。
- **使用deploy而不是install：** 构建的SNAPSHOT输出应当被自动部署到私有Maven仓库供他人使用，这一点在前面已经详细论述。
- **使用-U参数：** 该参数能强制让Maven检查所有SNAPSHOT依赖更新，确保集成基于最新的状态，如果没有该参数，Maven默认以天为单位检查更新，而持续集成的频率应该比这高很多。
- **使用-e参数：** 如果构建出现异常，该参数能让Maven打印完整的stack trace，以方便分析错误原因。
- **使用-Dmaven.repo.local参数：** 如果持续集成服务器有很多任务，每个任务都会使用本地仓库，下载依赖至本地仓库，为了避免这种多线程使用本地仓库可能会引起的冲突，可以使用 **-Dmaven.repo.local=/home/juven/ci/foo-repo/** 这样的参数为每个任务分配本地仓库。
- **使用-B参数：** 该参数表示让Maven使用批处理模式构建项目，能够避免一些需要人工参与交互而造成的挂起状态。

综上，持续集成服务器上的集成命令应该为 **mvn clean deploy -B -e -U -Dmaven.repo.local=xxx** 。此外，定期清理持续集成服务器的本地Maven仓库也是个很好的习惯，这样可以避免浪费磁盘资源，几乎所有的持续集成服务器软件都支持本地的脚本任务，你可以写一行简单的shell或bat脚本，然后配置以天为单位自动清理仓库。需要注意的是，这么做的前提是你有私有Maven仓库，否则每次都从Internet下载所有依赖会是一场噩梦。

用好Profile

如果不需要考虑各种不同的环境， 而且你的自动测试（包括集成测试）跑得飞快，那你就不用为项目建立多个集成任务。但实际的情况是，集成的时候可能要考虑各种环境，例如开发环境、测试环境、产品环境。而当项目越来越大，测试越来越多，控制构建时间在一个可接受的范围内（例如10分钟）变得越来越不现实。《持续集成——软件质量改进和风险降低之道》中介绍了一种名为**分阶段构建（staged build）**的解决方案，例如你可以将构建分为两个部分，第一部分包括了编译和单元测试等能够快速结束的任务，第二个部分包括集成测试等耗时较长的任务，只有第一部分成功完成后，才触发第二部分集成。这么做的意义在于让持续集成的反馈尽可能的快。

Maven的Profile机制能够很好的支持分阶段构建。例如，借助Maven Surefire Plugin，你可以统一单元测试命名为**UT，统一集成测试命名为**IT，然后配置Maven Surefire Plugin默认只运行单元测试，然后再编写一个名为integrationTest的Profile，在其中配置Maven Surefire Plugin运行集成测试。然后再以此为基础分阶段构建项目，第一个构建为 **mvn clean install -B -e -U** ，第二个构建任务为 **mvn clean deploy -B -e -U -PintegrationTest** 。前一个构建成功后再触发第二个构建，然后才部署至Maven仓库。值得一提的是，Maven Surefire Plugin能够很好支持JUnit 3、JUnit 4和TestNG，你可以按照最适合自己的方式来划分单元测试和集成测试。

另一个常见的分阶段构建案例是生成Maven站点，使用 **mvn clean site** 生成站点往往比较耗时且耗资源，这样的任务对应的持续集成中的*持续审查*阶段，该阶段往往不需要很高的集成频率。你会希望每10分钟就检查源代码变更并编译测试，但很少有人会希望每10分钟让系统生成一次测试覆盖率报告、CheckStyle报告等内容，因此合理的做法是使用一个较低的频率，例如每天，这样可以避免无谓的资源消耗，更重要的是，这样不会拖慢本该很快的编译和单元测试等反馈内容。

还有一些情况是系统需要基于不同环境进行集成，这时候就需要用到Maven的属性机制、资源过滤、以及前面提到的Profile。篇幅原因，这里不再展开。

小结

持续集成是敏捷最重要的实践之一，但如何在基于Maven的环境下实践持续集成却鲜有文章详述，本文介绍了一些该主题的最佳实践，包括架设私有仓库、使用正确的集成命令、利用Profile等技术处理分阶段构建等等。本文旨在让广大Maven用户认识到这些实践的存在及重要性，并没有详细解释一些诸如Nexus安装配置、Maven Surefire Plugin配置、或者说Profile配置使用方面的细节，如果你希望看到更细节的介绍，可以参考我的《Maven实战》一书。除了上面的内容之外，该书还详细解释了如何使用Hudson（也许该改称Jenkins了）这一最流行的开源持续集成服务器。当然，如果你有关于Maven和持续集成方面的经验，也请不吝分享。

本文已经首发于InfoQ中文站，版权所有，原文为《Maven实战（四）——基于Maven的持续集成实践
