持续集成（第二版）

作者：Martin Fowler

译者：雷镇

持续集成 是一种软件开发实践。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，也可以多次。每次集成会经过自动构建（包括自动测试）的 检验，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的速度。这篇文章简要介绍了持续集成的技巧和它 最新的应用。

最后更改于：2006 年 5 月 1 日

目录

我还可以生动记起第一次看到大型软件工程的情景。我当时在一家大型英国电子公司的 QA 部门实习。我的经理带我熟悉公司环境，我们进到一间巨大的，充满了压抑感和格子间的的仓库。我被告知这个项目已经 开发了好几年，现在正在集成阶段，并已经集成了好几个月。我的向导还告诉我没人知道集成要多久才能结束。从此我学到了软件开发的一个惯例：集成是一个很耗 时并难以预测的过程。但是事实并非总是如此，我的 ThoughWorks 同事所做的项目，以及很多其它遍布世界各地的软件项目，都不会把集成当回事。任何一个开发者本地的代码和项目共享基准代码的差别仅仅只有几小时的工作而已，而且这只要几分钟的时间就可以被集成回去。任何集成错误都可以很快被发现，并被快速修复。这鲜明的差别并非源于昂贵和复杂的工具。其中的精华蕴含于一个简单的实践：使用统一的代码仓库并频繁集成（通常每天一次）。

当我向别人介绍持续集成方法时，人们通常会有 两种反应："这（在我们这儿）不管用"和 "做了也不可能有什么不同"。但如果他们真的试过了，就会发现持续集成其实比听起来要

简单 ,并且能给开发过程带来 巨大的改变。因此第三种常见的反应是 :"我们就是这么做的，做开发怎可能不用它呢？"

"持续集成"一词来源于 极限编程（Extreme Programming），作为它的 12 个实践之一出现。当我开始在 ThoughWorks 开始顾问职业生涯时，我鼓励我所参与的项目使用这种技巧。Matthew Foemmel 将我抽象的指导思想转化为具体的行动。我们看到了项目从少而繁杂的集成进步到我所描述的不当回事。Metthew 和我将我们的经验写进了这篇论文的第一版 里。这篇论文后来成了我网站里最受欢迎的文章之一。

尽管持续集成不需要什么特别的工具 ,我们发现使用一个持续集成服务器软件还是很有效果的。最出名的持续集成服务器软件是 CruiseControl，这是一个开源工具，最早由 ThoughWorks 的几个人开发，现在由社区维护。之后还有许多其他持续集成服务器软件出现，有些是开源的，有些则是商业软件，比如 ThoughtWorks Studio 的 Cruise。

用持续集成构建特性

对我来说，解释持续集成最简单的方法就是用一个简单的例子来示范开发一个小 feature。现在假设我要完成一个软件的一部分功能，具体任务是什么并不重要，我们先假设这个 feature 很小，只用几个小时就可以完成。（我们稍后会研究更大的任务的情况。）

一开始，我将已集成的源代码复制一份到 本地计算机。这可以通过从源码管理系统的

mainline 上 check out 一份源代码做到。

如果 你用过任何源代码管理系统，理解上面的文字应该不成问题。但如果你没用过，可能会有读天书的感觉。所以我们先快速解释一个这些概念。源代码管理系统将项目 的所有源代码都保存在一个“仓库（repository）”中。系统的当前状态通常被称为“mainline”。开发者随时都可以把 mainline 复制 一份到他们自己的计算机 ,这个过程被称为“check out”。开发者计算机上的拷贝被称为“工作拷贝（working copy）”。（绝大部分情况下，你最终都会把工作拷贝的内容提交到 mainline 上去，所以两者实际上应该差不多。）

现在我拿到了工作拷贝 ,接下来需要做一些事情来完成任务。这包括修改产品代码和添加修改自动化测试。在持续集成中，软件应该包含完善的可自动运行 的测试，我称之为自测试代码。这一般需要用到某一个流行的 XUnit 测试框架。

一旦完成了修改，我就会在 自己的计算机上启动一个自动化 build。这会将我的工作拷贝中的源代码编译并链接成为一个可执行文件，并在之上运行自动化测试。只有当所有的 build 和测试都完成并没有任何错误时，这个 build 过程才可以认为是成功的。

当我 build 成功后，我就可以考虑将改动提交到源码仓库。但麻烦的情况在于别人可能已经在我之前修改过 mainline。这时我需要首先把别人的修改更新到我的工作拷贝中，再重新做 build。如果别人的代码和我的有冲突，就会在编译或测试的过程中引起错误。我有责任改正这些问题，并重复这一过程，直到我的工作拷贝能通过 build 并和 mainline 的代码同步。

一旦我本地的代码能通过 build，并和 mainline 同步，我就可以把我的修改提交到源码仓库。

然而，提交完代码不表示就完事大吉了。我们还要 做一遍集成 build，这次在集成计算机上并要基于 mainline 的代码。只有这次 build 成功了，我的修改才算告一段落。因为总有可能我忘了什么东西在自己的机器上而没有更新到源码仓库。只有我提交的改动被成功的集成了，我的工作才能结束。这 可以由我手工运行，也可以由 Cruise 自动运行。

如果两个开发者的修改存在冲突，这通常会被第二个人提 交代码前本地做 build 时发现。即使这时侥幸过关，接下来的集成 build 也会失败掉。不管怎样，错误都会被很快检测出来。此时首要的任务就是改正错误并让 build 恢复正常。在持续集成环境里，你必须尽可能快地修复每一个集成 build。好的团队应该每天都有多个成功的 build。错误的 build 可以出现，但必须尽快得到修复。

这样做的结果是你总能得到一个稳定的软件，它可能有一些 bug，但可以正常工作。每个人都基于相同的稳定代码进行开发，而且不会离得太远，否则就会不得不花很长时间集成回去。Bug 被发现得越快，花在改正上的 时间就越短。

持续集成实践

从上面的故事我们大概了解了持续集 成是如何在我们的日常工作中发挥作用的。但让一切

正常运行起来还需要掌握更多的知识。我接下来会集中讲解一下高效持续集成的关键实践。

只维护一个源码仓库

在软件项目里需要很多文件协调一致才能 build 出产品。跟踪所有这些文件是一项困难的工作，尤其是当有很多人一起工作时。所以，一点也不奇怪，软件开发者们这些年一直在研发这方面的工具。这些工具称为 源代码管理工具，或配置管理，或版本管理系统，或源码仓库，或各种其它名字。大部分开发项目中它们是不可分割的一部分。但可惜的是，并非所有项目都是如 此。虽然很罕见，但我确实参加过一些项目，它们直接把代码存到本地驱动器和共享目录中，乱得一塌糊涂。

所以， 作为一个最基本的要求，你必须有一个起码的源代码管理系统。成本不会是问题，因为有很多优秀的开源工具可用。当前较好的开源工具是 Subversion。（更 老的同样开源的 CVS 仍被广泛使用，即使是 CVS 也比什么都不用强得多，但 Subversion 更先进也更强大。）有趣的是，我从与开发者们的交谈中了解到，很多商业源代码管理工具其实不比 Subversion 更好。只有一个商业软件是大家一致同意值得花钱的，这就是 Perforce。

一旦你有了源代码管理系统 ,你要确保所有人都知道到哪里去取代码。不应出现这样的问题：“我应该到哪里去找 xxx 文件？” 所有东西都应该存在源码仓库里。

即便对于用了源码仓库的团队，我还是观察到一个很普遍的错误，就是他们没有把 所有东

西都放在源码仓库里。一般人们都会把代码放进去，但还有许多其它文件，包括测试脚本，配置文件，数据库 Schema，安装脚本，还有第三方的库，所 有这些 build 时需要的文件都应该放在源码仓库里。我知道一些项目甚至把编译器也放到源码仓库里（用来对付早年间那些莫名其妙的 C++编译器很有效）。 一个基本原则是：你必须能够在一台干净的计算机上重做所有过程，包括 checkout 和完全 build。只有极少量的软件需要被预装在这台干净机器上，通 常是那些又大又稳定，安装起来很复杂的软件，比如操作系统，Java 开发环境，或数据库系统。

你必须把 build 需要的所有文件都放进源代码管理系统，此外还要把人们工作需要的其他东西也放进去。IDE 配置文件就很适合放进去，因为大家共享同样的 IDE 配 置可以让工作更简单。

版本控制系统的主要功能之一就是创建 branch 以管理开发流。这是个很有用的功能，甚至可以说是一个基础特性，但它却经常被滥用。你最好还是尽量少用 branch。一般有一个 mainline 就够 了，这是一条能反映项目当前开发状况的 branch。大部分情况下，大家都应该从 mainline 出发开始自己的工作。（合理的创建 branch 的 理由主要包括给已发布的产品做维护和临时性的实验。）

一般来说，你要把 build 依赖的所有文件放进代码管理 系统中，但不要放 build 的结果。有些人习惯把最终产品也都放进代码管理系统中，我认为这是一种坏味道——这意味着可能有一些深层次的问题，很可能是无 法可靠地重新 build 一个产品。

自动化 build

通常来 说，由源代码转变成一个可运行的系统是一个复杂的过程，牵扯到编译，移动文件，将 schema 装载到数据库，诸如此类。但是，同软件开发中的其它类似任务一样，这也可以被自动化，也必须被自动化。要人工来键入各种奇怪的命令和点击各种对话框纯粹是 浪费时间，也容易滋生错误。

在大部分开发平台上都能找到自动化 build 环境的影子。比如 make，这在 Unix 社区已经用了几十年了，Java 社区也开发出了 Ant，.NET 社区以前用 Nant，现在用 MSBuild。不管你在什么平台上，都要确保只用一条命令就可以运行这些脚本，从而 build 并运行系统。

一 个常见的错误是没有把所有事都放进自动化 build。比如：Build 也应该包括从源码仓库中取出数据库 schema 并在执行环境中设置的过程。我要重申一下前面说过的原则：任何人都应该能从一个干净的计算机上 check out 源代码，然后敲入一条命令，就可以得到能在这台机器上运行的系统。

Build 脚本有很多不同的选择，依它们所属的平台和社区而定，但也没有什么定势。尽管大部分的 Java 项目都用 Ant，还是有一些项目用 Ruby（Ruby Rake 是一个不错的 build 脚本工具）。我们也曾经用 Ant 自动化早期的 Microsoft COM 项目，事实证明很有价值。

一个大型 build 通常会很耗时，如果只做了很小的修改，你不会想花时间去重复所有的步

骤。所以一个好的 build 工具应该会分析哪些步骤可以跳过。一个通用的办法是比较源文件和目标文件的修改时间，并只编译那些较新的源文件。处理依赖关系要麻烦一些：如果一个目标文 件修改了，所有依赖它的部分都要重新生成。编译器可能会帮你处理这些事情，也可能不会。

根据你的需要，你可能 会想 build 出各种不同的东西。你可以同时 build 系统代码和测试代码，也可以只 build 系统代码。一些组件可以被单独 build。Build 脚本应该允许你在不同的情况中 build 不同的 target。

我们许多人都用 IDE，许多 IDE 都内置包含某种 build 管理功能。然而，相应的配置文件往往是这些 IDE 的专有格式，而且往往不够健壮，它们离了 IDE 就无法工作。如果只是 IDE 用户自己一个人开发的话，这还能够接受。但在团队里，一个工作于服务器上的主 build 环境和从其它脚本里运行的能力更重要。我们认为，在 Java 项目里，开发者可以用自己的 IDE 做 build，但主 build 必须用 Ant 来做，以保证它可以在开发服务器上运行。

让你的 build 自行测试

传统意义上的 build 指编译，链接，和一些其它能让程序运行起来的步骤。程序可以运行并不意味着它也工作正常。现代静态语言可以在编译时检测出许多 bug，但还是有更多的漏网之鱼。

一种又快又省的查 bug 的方法是在 build 过程中包含自动测试。当然，测试并非完美解

决方案，但它确实能抓住很多 bug——多到可以让软件真正可用。极限编程（XP）和测试驱动开发（TDD）的出现很好地普及了自测试代码的概念，现在已经有很多人意识到了这种技巧的 价值。

经常读我的著作的读者都知道我是 TDD 和 XP 的坚定追随者。但是我想要强调你不需要这两者中任何一个就能享受自测试代码的好处。两者都要求你先写测试，再写代码以通过测试，在这种工作模式里测试更多 着重于探索设计而不是发现 bug。这绝对是一个好方法，但对于持续集成而言它并不必要，因为这里对自测试代码的要求没有那么高。（尽管我肯定会选择用 TDD 的方式。）

自测试代码需要包含一套自动化测试用例，这些测试用例可以检查大部分代码并找出 bug。测试要能够从一条简单的命令启动。测试结果必须能指出哪些测试失败了。对于包含测试的 build，测试失败必须导致 build 也失败。

在过去的几年里，TDD 的崛起普及了开源的 XUnit 系列工具，这些工具用作以上用途非常理想。对于我们在 ThoughWorks 工作的人来说，XUnit 工具已经证明了它们的价值。我总是建议人们使用它们。这些最早由 Kent Beck 发明的工具使得设置一个完全自测试环境的工作变得非常简单。

毋庸置疑，对于自动测试的工作而言，XUnit 工具只是一个起点。你还必须自己寻找其他更适合端对端测试的工具。现在有很多此类工具，包括 FIT，Selenium，Sahi，Watir，FITnesse，和许多其它我无法列在这里的工具。

当然你不能指望测试发现所有问题。就像人们经常说的：测试通过不能证明没有 bug。然而，完美并非是你要通过自测试 build 达到的唯一目标。经常运行不完美的测试要远远好过梦想着完美的测试，但实际什么也不做。

每人每天都要向 mainline 提交代码

集成的主要工作其实是沟通。集成可以让开发者告诉其他人他们都改了什么东西。频繁的沟通可以让人们更快地了解变化。

让开发者提交到 mainline 的一个先决条件是他们必须能够正确地 build 他们的代码。这当然也包括通过 build 包含的测试。在每个提交迭代里，开发者首先更新他们的工作拷贝以与 mainline 一致，解决任何可能的冲突，然后在自己的机器上做 build。在 build 通过后，他们就可以随便向 mainline 提交了。

通过频繁重复上述过程，开发者可以发现 两个人之间的代码冲突。解决问题的关键是尽早发现问题。如果开发者每过几个小时就会提交一次，那冲突也会在出现的几个小时之内被发现，从这一点来说，因为 还没有做太多事，解决起来也容易。如果让冲突待上几个星期，它就会变得非常难解决。

因为你在更新工作拷贝时也 会做 build，这意味着你除了解决源代码冲突外也会检查编译冲突。因为 build 是自测试的，你也可以查出代码运行时的冲突。后者如果在一段较长的

时间还没被查出的话会变得尤其麻烦。因为两次提交之间只有几个小时的修改，产生这些问题 只可能在很有限的几个地方。此外 ,因为没改太多东西 ,你还可以用 diff-debugging 的技巧来找 bug。

总的来说，我 的原则是每个开发者每天都必须提交代码。实践中，如果开发者提交的更为频繁效果也会更好。你提交的越多，你需要查找冲突错误的地方就越少，改起来也越快。

频繁提交客观上会鼓励开发者将工作分解成以小时计的小块。这可以帮助跟踪进度和让大家感受到进展。经常会有人一开始根 本无法找到可以在几小时内完成的像样的工作，但我们发现辅导和练习可以帮助他们学习其中的技巧。

每次提交都应在集成计算机上重新构建 mainline

使用每日提交的策略后，团队就能得到很多经过测试的 build。这应该意味着 mainline 应该总是处于一种健康的状态。但在实践中，事情并非总是如此。一个原因跟纪律有关，人们没有严格遵守在提交之前在本地更新并做 build 的要求。另一个原因是开发者的计算机之间环境配置的不同。

结论是你必须保证日常的 build 发生在专用的集成计算机上，只有集成 build 成功了，提交的过程才算结束。本着“谁提交，谁负责”的原则，开发者必须监视 mainline 上的 build 以便失败时及时修复。一个推论是如果你在下班前提交了代码，那你在 mainline build 成功之前就不能回家。

我知道主要有两种方法可以使用：手动 build，或持续集成服务器软件。

手动 build 描述起来比较简单。基本上它跟提交代码之前在本地所做的那次 build 差不多。开发者登录到集成计算机，check out 出 mainline 上最新的源码（已包含最新的提交），并启动一个集成 build。他要留意 build 的进程，只有 build 成功了他的提交才算成功。（请查看 Jim Shore 的描述。）

持续集成服务器软件就像一个监视着源码仓库的监视器。每 次源码仓库中有新的提交，服务器就会自动 check out 出源代码并启动一次 build，并且把 build 的结果通知提交者。这种情况下，提交者的工作直到收到通知（通常是 email）才算结束。

在 ThoughtWorks，我们都是持续集成服务器软件的坚定支持者，实际上我们引领了 CruiseControl 和 CruiseControl.NET 最 早期的开发，两者都是被广泛使用的开源软件。此后，我们还做了商业版的 Cruise 持续集成服务器。我们几乎在每一个项目里都会用持续集成服务器，并且对结果非常满意。

不是每个人都会用持续集成服务器。Jim Shore 就清楚地表达了为什么他更偏好手动的办法。我同意他的看法中的持续集成并不仅仅是安装几个软件而已，所有的实践 都必须为了能让持续集成更有效率。但同样的，许多持续集成执行得很好的团队也会发现持续集成服务器是个很有用的工具。

许多组织根据安排好的日程表做例行 build，如每天晚上。这其实跟持续集成是两码事，而且做得远远不够。持续集成的最终目标就是要尽可能快地发现问题。Nightly build 意味着 bug 被发现之前可能会待上整整一天。一旦 bug 能在系统里呆这么久，找到并修复它们也会花较长的时间。

做好持续集成的一个关键因素是一旦 mainline 上的 build 失败了，它必须被马上修复。而在持续集成环境中工作最大的好处是，你总能在一个稳定的基础上做开发。mainline 上 build 失败并不总是坏事，但如果它经常出错，就意味着人们没有认真地在提交代码前先在本地更新代码和做 build。当 mainline 上 build 真的失败时，第一时间修复就成了头等大事。为了防止在 mainline 上的问题，你也可以考虑用 pending head 的方法。

当团队引入持续集成时，这通常是最难搞定的事情之一。在初期，团队会非常难以接 受频繁在 mainline 上做 build 的习惯，特别当他们工作在一个已存在的代码基础上时更是如此。但最后耐心和坚定不移的实践常常会起作用，所以不要气馁。

保持快速 build

持续集成的重点就是快速反馈。没有什么比缓慢的 build 更能危害持续集成活动。这里我必须承认一个奇思怪想的老家伙关于 build 快慢标准的的玩笑（译者注：原文如此，不知作者所指）。我的大部分同事认为超过 1 小时的 build 是不能忍受的。团队们都梦想着把 build 搞得飞快，但有时我们也确实会发现很难让它达到理想的速度。

对大多数项目来说，XP 的 10 分钟 build 的指导方针非常合理。我们现在做的大多数项目都能达到这个要求。这值得花些力气去做，因为你在这里省下的每一分钟都能体现在每个开发者每次提交的时候。持 续集成要求频繁提交，所以这积累下来能节省很多时间。如果你一开始就要花 1 小时的时间做 build，想加快这个过程会相当有挑战。即使在一个从头开始的新项目里，想让 build 始终保持快速也是很有挑战的。至少在企业应用里，我们发现常见的瓶颈出现在测试时，尤其当测试涉及到外部服务如数据库。

也许最关键的一步是开始使用分阶段 build（staged build）。分阶段 build（也被称作 build 生产线）的基本想法是多个 build 按一定顺序执行。向 mainline 提交代码会引发第一个 build，我称之为提交 build（commit build）。提交 build 是当有人向 mainline 提交时引发的 build。提交 build 要足够快，因此它会跳过一些步骤，检测 bug 的能力也较弱。提交 build 是为了平衡质量检测和速度，因此一个好的提交 build 至少也要足够稳定以供他人基于此工作。

一旦提交 build 成功，其他人就可以放心地基于这些代码工作了。但别忘了你还有更多更慢的测试要做，可以另找一台计算机来运行运行这些测试。

一个简单的例子是两阶段 build。第一阶段会编译和运行一些本地测试，与数据库相关的单元测试会被完全隔离掉（stub out）。这些测试可以运行得非常快，符合我们的 10 分钟指导方针。但是所有跟大规模交互，尤其是真正的数据库交互的 bug 都无法被发现。第二阶段的 build 运行一组不同的测试，这些测试会调用真正的数据库并涉及更多的端到端的行为。这些测试会跑上好几小时。

这种情况下，人们用第一阶段作为提交 build，并把这作为主要的持续集成工作。第二阶段 build 是次级 build，只有 在需要的时候才运行，从最后一次成功的提交 build 中取出可执行文件作进一步测试。如果次级 build 失败了 ,大家不会立刻停下手中所有工作去修复，但团队也要在保证提交 build 正常运行的同时尽快修正 bug。实际上次级 build 并非一定要正常运行 ,只要 bug 都能够被检查出来并且能尽快得到解决就好。在两阶段 build 的例子里，次级 build 经常只是纯粹的测试，因为通常只是测试拖慢了速度。

如果次级 build 检查到了 bug，这是一个信号，意味着提交 build 需要添加一个新测试了。你应该尽可能把次级 build 失败过的测试用例都添加到提交 build 中，使得提交 build 有能力验证这些 bug。每当有 bug 绕过提交测试，提交测试总能通过这种方法被加强。有时候确实无法找到测试速度和 bug 验证兼顾的方法，你不得不决定把这个测试放回到次级 build 里。但大部分情况下都应该可以找到合适加入提交 build 的测试。

上面这个例子是关于两阶段 build，但基本原则可以被推广到任意数量的后阶段 build。提交 build 之后的其它 build 都可以同时进行，所以如果你的次级测试要两小时才能完成，你可以通过用两台机器各运行一半测试来快一点拿到结果。通过这个并行次级 build 技巧，你可以向日常 build 流程中引入包括性能测试在内的各种自动化测试。(当我过去几年内参加 Thoughtworks 的各种项目时，我碰到了很多有趣的技巧，我希望能够说服一些开发者把这些经验写出来。)

在模拟生产环境中进行测试

测试的关键在于在受控条件下找出系统内可能在实际生产中出现的任何问题。这里一个明显的因素是生产系 统的运行环境。如果你不在生产环境做测试，所有环境差异都是风险，可能最终造成测试环境中运行正常的软件在生产环境中无法正常运行。

自然你会想到建立一个与生产环境尽可能完全相同的测试环境。用相同的数据库软件，还要同一个版本；用相同版本的操作系统；把所有生产环 境用到的库文件都放进测试环境中，即使你的系统没有真正用到它们；使用相同的 IP 地址和端口；以及相同的硬件；

好吧，现实中还是有很多限制的。如果你在写一个桌面应用软件，想要模拟所有型号的装有不同第三方软件的台式机来测试显然是不现实的。类似的，有些生产环境可 能因为过于昂贵而无法复制(尽管我常碰到出于经济考虑拒绝复制不算太贵的环境 结果得不偿失的例子)。即使有这些限制，你的目标仍然是尽可能地复制生产环 境，并且要理解并接受因测试环境和生产环境不同带来的风险。

如果你的安装步骤足够简单，无需太多交互，你也许 能在一个模拟生产环境里运行提交 build。但事实上系统经常反应缓慢或不够稳定，这可以用 test double 来解决。结果常常是提交测试为了速度原因在一个假环境内运行，而次级测试运行在模拟真实的生产环境中。

我注意到越来越多人用虚拟化来搭建测试环境。虚拟机的状态可以被保存，因此安装并测试最新版本的 build 相对简单。此外，这可以让你 在一台机器上运行多个测试，或在一台机器上模拟网络里的多台主机。随着虚拟化性能的提升，这种选择看起来越来越可行。

让每个人都能轻易获得最新的可执行文件

软件开发中最困难的部分是确定你的软件行为符合预期。我们 发现事先清楚并正确描述需求非常困难。对人们而言，在一个有缺陷的东西上指出需要修改的地方要容易得多。敏捷开发过程认可这种行为，并从中受益。

为了以这种方式工作，项目中的每个人都应该能拿到最新的可执行文件并运行。目的可以为了 demo，也可以为了探索性测试，或者只是为了看看这周有什么进展。

这做起来其实相当简单：只要找到一个大家 都知道的地方来放置可执行文件即可。可以同时保存多份可执行文件以备使用。每次放进去的可执行文件应该要通过提交测试，提交测试越健壮，可执行文件就会越 稳定。

如果你采用的过程是一个足够好的迭代过程，把每次迭代中最后一个 build 放进去通常是明智的决定。Demo 是一个特例，被 demo 的软件特性都应该是演示者熟悉的特性。为了 demo 的效果值得牺牲掉最新的 build，转而找一个早一点但演示者更熟悉的版本。

每个人都能看到进度

持续集成中最重要的是沟通。你需要保证每个人都能轻易看到系统的状态和最新的修改。沟通的最重要的途 径之一是 mainline build。如果你用 Cruise，一个内建的网站会告诉

你是否正有 build 在进行，和最近一次 mainline build 的状态。许多团队喜欢把一个持续工作的状态显示设备连接到 build 系统来让这个过程更加引人注目，最受欢迎的显示设备是灯光，绿灯闪亮表示 build 成功，红灯表示失败。一种常见的选择是红色和绿色的熔岩灯，这不仅仅指示 build 的状态，还能指示它停留在这个状态的时间长短，红灯里出现气泡表示 build 出问题已经太长时间了。每一个团队都会选择他们自己的 build 传感器。如果你的选择带点幽默性和娱乐性效果会更好（最近我看到有人在实验跳舞兔）。

即使你在使用手动持续集 成，可见程度依然很重要。Build 计算机的显示器可以用来显示 mainline build 的状态。你很可能需要一个 build 令牌放在正在做 build 那人的桌子上（橡皮鸡这种看上去傻傻的东西最好，原因同上）。有时人们会想在 build 成功时弄出一点噪音来，比如摇铃的声音。

持续集成服务器软件的网页可以承载更多信息。Cruise 不仅显示谁在做 build，还能指出他们都改了什么。Cruise 还提供了一个历史修改记录，以便团队成员能够对最近项目里的情况有所了解。我知道 team leader 喜欢用这个功能了解大家手头的工作和追踪系统的更改。

使用网站的另一大优点是便于那些 远程工作的人了解项目的状态。一般来说，我倾向于让项目中发挥作用的成员都坐在一起工作，但通常也会有一些外围人员想要了解项目的动态。如果组织想要把多 个项目的 build 情况聚合起来以提供自动更新的简单状态时，这也会很有用。

好的信息展示方式不仅仅依赖于 电脑显示器。我最喜欢的方式出现于一个中途转入持续集成的项目。很长时间它都无法拿出一个稳定的 build。我们在墙上贴了一整年的日历，每一天都是一个小方块。每一天如果 QA 团队收到了一个能通过提交测试的稳定 build，他们都会贴一张绿色的贴纸，否则就是红色的贴纸。日积月累，从日历上能看出 build 过程在稳定地进步。直到绿色的小方块已经占据了大部分的空间时，日历被撤掉了，因为它的使命已经完成了。

自动化部署

自动化集成需要多个环境，一个运行提交测试，一个或多个运行次级测试。每天在这些环境之间频繁拷贝 可执行文件可不轻松，自动化是一个更好的方案。为实现自动化，你必须有几个帮你将应用轻松部署到各个环境中的脚本。有了脚本之后，自然而然的结果是你也要 用类似的方式部署到生产环境中。你可能不需要每天都部署到生产环境（尽管我见过这么做的项目），但自动化能够加快速度并减少错误。它的代价也很低，因为它 基本上和你部署到测试环境是一回事。

如果你部署到生产环境，你需要多考虑一件事情：自动化回滚。坏事情随时可 能发生，如果情况不妙，最好的办法是尽快回到上一个已知的正常状态。能够自动回滚也会减轻部署的压力，从而鼓励人们更频繁地部署，使得新功能更快发布给用 户。（Ruby on Rails 社区开发了一个名为 Capistrano 的工具，是这类工具很好的代表。）

我还在服 务器集群环境中见过滚动部署的方法，新软件每次被部署到一个节点上，在几小

时时间内逐步替换掉原有的软件。

参见相关文章：进化式数据库设计

许多人在频繁发布时都遇到的一个障碍是数据库的迁移。数据库的改动很麻烦，因为你不能仅仅修改 schema，你还要保证数据本身也被顺利迁移。这篇文章描述了我的同事 Pramod Sadalage 自动化重构和迁移数据库的技巧。这篇文章只是一个简单的记录，其中的信息在 Pramod 和 Scott Amblers 关于数据库重构的书中有更详细的阐述 [ambler-sadalage]。

在 web 应用开发中，我碰到的一个有趣的想法是把一个试验性的 build 部署到用户的一个子集。团队可以观察这个试验 build 被使用的情况，以决定是否将它部署到全体用户。你可以在做出最终决定之前试验新的功能和新的 UI。自动化部署加上良好的持续集成的纪律是这项工作的基础。

持续集成的益处

我认为持续集成最显著也最广泛的益处是降低风险。说到这里，我的脑海中还是会浮现出第一段描述的早期软件项目。他们已经到了一个漫长项 目的末期（至少他们期望如此），但还是不知道距离真正的结束有多远。

延迟集成的问题在于时间难以估计，你甚至 无法得知你的进展。结果是你在项目最紧张的阶段之一把自己置入了一个盲区，此时即使没有拖延（这很罕见）也轻松不了多少。

持续集成巧妙的解决了这个问题。长时间的集成不再存在，盲区被彻底消除了。在任何时间你都知道你自己的进展，什么能运转，什么不能运转，你系统里 有什么明显的 bug，这些都一目了然。

Bug 让人恶心，它摧毁人的自信，搞乱时间表，还破坏团队形象。已部署软件里的 bug 招致用户的怒气。未完成软件里的 bug 让你接下来的开发工作受阻。

持续集成不能防止 bug 的产生，但它能明显让寻找和修改 bug 的工作变简单。从这个方面看，它更像自测试代码。如果你引入 bug 后能很快发现，改正也会简单得多。因为你只改了系统中很小的一部分，你无需看很多代码就能找到问题所在。因为这一小部分你刚刚改过，你的记忆还很新鲜，也 会让找 bug 的工作简单不少。你还可以用差异调试——比较当前版本和之前没有 bug 的版本。

Bug 也会积累。你的 bug 越多，解决掉任何一个都会越困难。这部分原因是 bug 之间的互相作用，你看到的失败实际上是多个问题叠加的结果，这使得检查其中任何一个问题都更加困难。还有部分原因是心理层面的因素，当人们面对大量 bug 时，他们寻找和解决 bug 的动力就会减弱。《Pragmatic Programmer》一书中称之为“破窗综合症“。

使用持续集成的项目的通常结果是 bug 数目明显更少，不管在产品里还是开发过程中都是如此。然而，我必须强调，你受益的程度跟你测试的完善程度直接相关。其实建立测试系统并非想象中那么困难， 但带来的区别却显而易见。一般来说，团队需要花一定时间才能把

bug 数量减少到理想的地步。做到这一点意味着不断添加和改进测试代码。

如果你用了持续集成，你就解决了频繁部署的最大障碍之一。频繁部署很有价值，因为它可以让你的用户尽快用到新功能，从而快速提供反馈， 这样他们在开发过程中可以有更多的互动。这可以帮助打破我心目中成功的软件开发最大的障碍——客户与开发团队之间的障碍。

引入持续集成

看到这里，你一定想要尝试一下持续集成了。但是从哪里开始呢？我在上面描述了一整套 的实践，这些可以让你体验到所有的好处。但你也不必一开始就照单全收，你有自己的选择余地，基本上取决于你的环境和团队的特性。我们也从过去的实践中吸取 了一些经验和教训，做好下面这些事会对于你的持续集成运作有重要的意义。

最早的几步之一是实现 build 自动化。把所有需要的东西都放进版本控制系统里，这样你就可以用一条命令 build 整个系统。这对许多项目而言不是什么小任务，这对于其他东西正常工作非常重要。刚开始你可能只是偶尔需要的时候做一个 build，或者只是做一个自动的 nightly build。当你还没有开始持续集成时，自动 nightly build 也是一个不错的开始。

其次是引入一些自动化测试到你的 build 中。试着指出主要出错的地方，并要让自动化测试暴露这些错误。建立又快又好的测试集合会比较困难，特别在已存在的项目中，建立测试需要时间。你必须找个地 方开始动手，就像俗话说的，罗马不是一天建成的。

还要试着加快提交 build 的速度。虽然需要几个小时 build 的持续集成也比什么都没有好，但能做到传说中的 10 分钟会更好。这通常要对你的代码动一些大手术，以剥离对运行缓慢那部分的依赖。

如果你刚刚开始一个新项目，从一开始就用持续集成。对 build 时间保持关注，当慢于 10 分钟时就立即采取行动。通过快速行动，你可以 在代码变得太大之前做一些必要的架构调整。

比所有事情都重要的是寻找帮助。找一个以前做过持续集成的人来帮 你。像所有新技巧一样，当你不知道最终结果怎样的时候会非常难以实施。请一个导师（mentor）可能会花些钱，但如果你不做，你会付出时间和生产效率损 失的代价。（免责声明/广告：是的，我们 ThoughtWorks 在这个领域提供咨询服务。不管怎样，我们曾经犯过你可能会犯的大多数错误。）

最后的思考

在我和 Matt 写完最初那篇论文后的几年，持续集成已经成为了软件开发的一个主流方法。ThoughtWorks 的项目很少有不用到它的。我们也能看到世界各地的人们在用持续集成。与极限编程中一些充满争议的实践不同，我很少听到关于持续集成的负面消息。

如果你还没用持续集成，我强烈建议你试一下。如果你已经在做了，可能这篇文章中的某些

方法可以帮你得到更好的效果。过 去几年中，我们已经了解了很多关于持续集成的知识，我希望还有更多的知识可以让我们学习和提高。

延伸阅读

本文篇幅所限，只能覆盖部分内容。想了解持续集成的更多细节，我建议看一下 Paul Duvall 的书（这本书得到了 Jolt 大奖）。目前为止还没有多少关于分阶段 build 的文章，但有一篇 Dave Farley 发表在《ThoughtWorks 文选》中的文章还不错（你也可以在这里找到）。

感谢，Thanks！