06 | 代码入库到产品上线：Facebook如何使用CI/CD满足业务要求?

2019-09-04 葛俊

讲述：葛俊
时长 17:50　大小 16.34M

你好，我是葛俊。

在上一篇文章中，我和你分享了代码入库前的流程优化，即持续开发。今天，我会继续与你介绍流程优化中，代码入库和入库后的 3 种持续工程方法，即持续集成（Continuous Integration, CI)、持续交付（Continuous Delivery, CD）和持续部署（Continuous Deployment, CD)。

在接下来的分享中，首先我会与你介绍这 3 种方法的定义和作用，帮助你深入理解这 3 个“持续”背后的原理和原则；然后我会以 Facebook 为参考，给你介绍基于这些原则的具体工程实践。我希望，这些内容能够帮助你用好这三个“持续”方法，提高团队的研发效能。

首先，我先来介绍一下，持续集成、持续交付和持续部署是用来解决什么问题的，也就是它们的定义和作用。

3 个“持续”的定义和作用

不知道你是否还记得，在开篇词中，我提到过一个低效能的情况，即产品发布上线时出现大量提交、合并，导致最后时刻出现很多问题。这个情况很常见，引起了很多用户的共鸣。产生这个问题最主要原因是，**代码合并太晚**。这里，我再与你详细解释一下原因。

当多个人同时开发一款产品的时候，很可能会出现代码冲突。而解决冲突，需要花费较多的时间；同时很可能出现冲突解决失败，导致产品问题。

如果没有一个机制督促我们尽早把代码推到主仓进行集成的话，我们通常会尽量先在自己的分支上进行开发。结果往往是，在冲刺快要结束，或者功能即将发布时，才出现大量的代码合并。

而这时因为很长时间没有进行过代码集成了，进行集成的代码量通常比较大，不同开发者的代码区别很大，冲突也很严重，难以解决。具体的负面影响是：发布推迟，产品质量不高，每一次发布时的熬夜和紧张影响团队士气。

而持续集成的根本出发点，就是为了解决这个问题。也就是说，它能够帮助开发人员尽量早、尽量频繁地把自己的改动推送到共享的代码仓库分支上，进行代码集成，从而减少大量代码冲突造成的低效能问题。

所以，**持续集成的定义就是：在团队协作中，一天内多次将所有开发人员的代码合并入同一条主干。**

代码入库后，剩下工作是把代码编译打包成可以发布的形式，先发布到测试环境进行测试，再发布到类生产环境进行测试，最终部署到生产环境。

在这个过程中，有两个问题需要特别注意。

首先，我们需要这个流程尽量频繁。如果我们能够把产品和功能尽快地发布到市场上，就能够更快地服务客户，以更快的试错速度寻找到用户，提供真正对客户有价值的功能。即使你的产品由于自身特性不会太频繁地部署给用户，但这种能够频繁生产出可以马上部署的产品的能力，也能让你在需要部署时，快速完成任务。

其次，在产品发布到不同环境的过程中，我们会发现一些在开发和持续集成中没有暴露的问题。如果在产品要正式发布时才发现这些问题，就会造成产品交付推迟，影响线上用户等情况，造成损失。针对这个情况，我们需要提前发现问题。

而解决这两个问题，正是持续交付和持续部署的出发点。

持续交付的目标是，对每一个进入主干分支的代码提交，构建打包成为可以发布的产品。它的定义是：**一种软件工程方法，在短周期内完成软件产品，以保证软件保持在随时可以发布的状态**。也就是说，对每一个提交，把集成后的代码部署到“类生产环境”中进行验证。如果代码没有问题，后续可以手动部署到生产环境中。

而持续部署，则更进一步。它把持续交付产生的产品立即自动部署给用户，定义就是：**将每一个代码提交，都构建出产品直接部署给用户使用。**

以上就是持续集成、持续交付与持续部署的作用和定义。在实现上，它们**共同的本质**是，让每一个变更都经过一条自动化的检验流水线，来检查每一个变更的质量，通过就进入下一个阶段。

这里的“下一个阶段”具体包括：代码并入主仓、产品进入测试环境、产品进入类生产环境、产品最终进入生产环境，如下图所示。

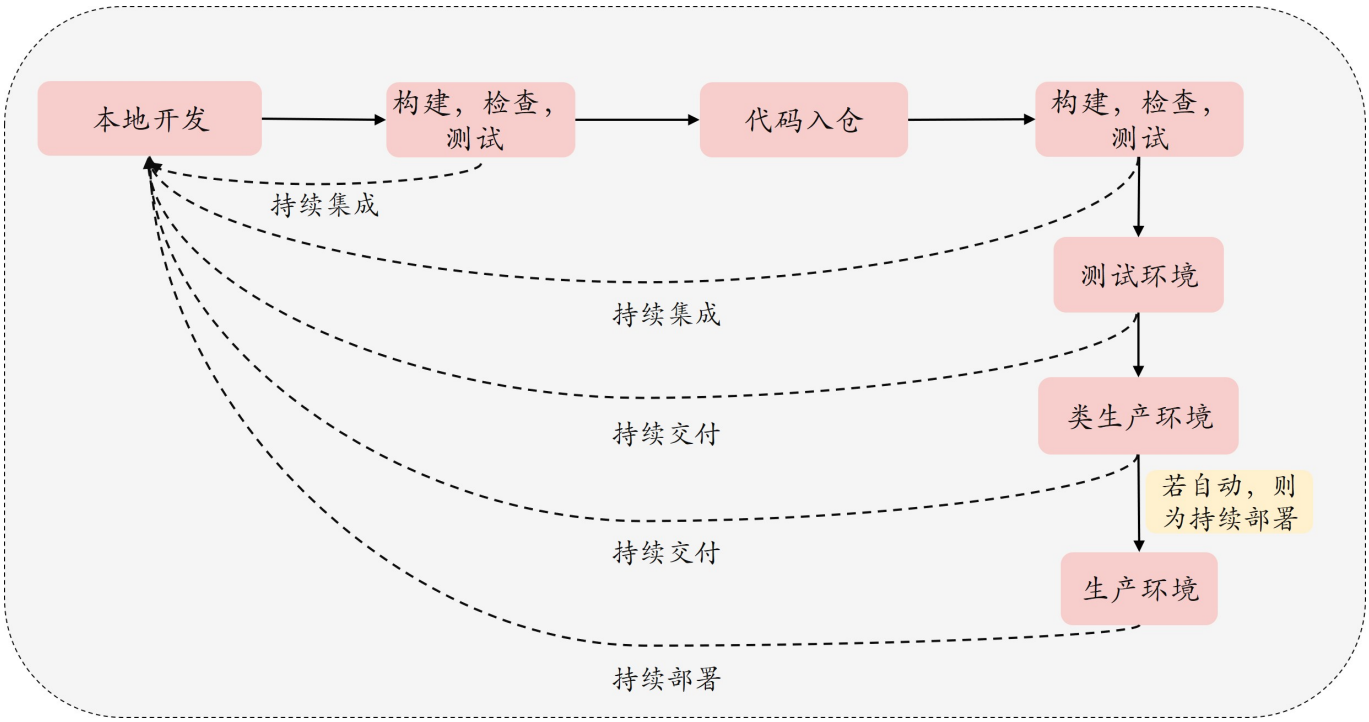

图 1 CI/CD 流水线示意图

你应该已经注意到了，整条流水线中，持续部署只是持续交付的最后一步，也就是自动化上线的那一步，前面的各种检查，都属于持续交付流水线。所以，**我在后面的内容中再提到流水线时，CI/CD 指的就是“持续集成 / 持续交付”。**

CI/CD 流水线，能够大大提高代码入库的速度和质量，是这几年硅谷互联网公司做到高效研发的必要条件。接下来，我就与你介绍 CI/CD 流水线的具体原则以及最佳实践，然后以 Facebook 的具体实践为例帮助你加深理解。

需要注意的是，在这篇文章中，我会重点与你分享 CI/CD 流水线的搭建原则。而关于具体的搭建方式，通常是持续集成工具 + 代码仓管理系统 + 检查工具 + 测试工具，比如 Jenkins+GitLab+SonarQube+Linter+UnitTest 的组合。你可以参考这个链接提供的方式去搭建。

CI/CD 流水线的具体原则以及最佳实践

根据上面提到的 3 个“持续”的本质，要做到高效，有 3 条基本原则：

流水线的测试要尽量完整；

流水线的运行速度一定要快；

流水线使用的环境，尽量和生产环境一致。

基本原则 1：流水线的测试要尽量完整

CI/CD 流水线的测试只有尽量完整，代码和产品的质量才能有保证。所以，最主要的工程实践，就是在流水线中运行大量高质量的测试和检查。

Facebook 就有大量的单元测试和集成测试用例、安全扫描，以及性能专项测试用例。如果某个验证在流水线中失败，开发人员会考虑是否要添加测试用例来防止再出现类似的问题。

另外，Facebook 持续在测试用例的开发上投入。在内部工具团队，有一个专门的测试工具团队来建设测试框架和测试流程，方便开发人员自己开发测试用例。比如，我在 Facebook 那几年，他们就一直在改进 JavaScript 的 Mock 框架，对开发人员写测试用例来说非常方便。

基本原则 2：流水线的运行速度一定要快

因为每一个变更都要通过 CI/CD 流水线的检验，所以流水线的速度关乎研发速度。而要提高这条流水线的速度，我们可以从以下两个方面考虑。

首先，从技术角度考虑。比如：

使用并行方式运行各种测试来提速；

投入硬件资源，使用水平扩展的方式来提速；

使用增量测试的方式进行精准验证。也就是说，只运行跟当前改动最相关的测试，以减少测试用例的运行数量。

其次，权衡流水线的运行速度、流水线资源和测试完整性的关系。不难理解，运行速度快、占用资源少、测试完整难以兼顾，因此我们必须做出权衡。这里我推荐几个方法：

如果通过增加硬件资源来提升运行速度需要的成本太高的话，可以对测试用例按优先级进行分类，每天运行流水线的时候，不用每次都运行所有测试用例，只选择其中几次进行全量测试。

提供支持，让开发人员在本地也能运行这些测试，从而使用本地资源尽早发现问题，这就避免了一些有问题的提交占用流水线的资源，进而提高整条流水线的运行速度。

运行测试的时候，按照一定的顺序运行测试用例。比如可以先运行速度快的用例，以及历史上容易发现问题的用例。这样可以尽早发现问题，避免耗费不必要的资源。

基本原则 3：流水线使用的环境，尽量和生产环境一致

这里的环境，包括机器环境、数据、软件包、网络环境等。环境不一致可能导致问题暴露在用户面前，损失严重；另外，在非生产环境上难以复现生产环境的问题，调试困难。

保证流水线环境与生产环境一致，具体方法包括：

软件包最好只构建一次，保证各种不同环境都用同一个包。如果不同的运行环境需要不同的参数，可以采用环境变量的方式把这些参数传递给软件包。

使用 Docker 镜像的方式，把发布的产品以及环境都打包进去，实现环境的一致性。在我看来，这正是 Docker 的最大好处。

尽量使用干净的环境。比如，测试时，使用刚从镜像产生的系统；又比如，使用蓝绿部署，每次产生新的部署时，直接丢弃旧的环境。

以上就是 CI/CD 流水线的 3 个基本原则和最佳实践。通过提高验证的完整性、速度，以及保证环境的一致性，我们可以降低成本，提高产品质量和验证产品价值假设的速度。

接下来，为了帮助你理解并正确运用这些原则和最佳实践，我们一起来看看 Facebook 是怎么做的。

具体案例：Facebook 是如何实施 CI/CD 来提高效能的?

Facebook 一直就非常注重 CI/CD，早在 2009 年就建设了顺畅的 CI/CD 流水线，而且一直在持续改进。

在 CI 方面，加强建设持续开发，让开发人员能在开发环境上进行大量的验证。本地的所有验证，与 CI 流水线上的验证方式保持一致，这就大大提高了开发人员在本地发现问题的能力，从而大量避免了有问题的代码提交到 CI 流水线，浪费资源。

在**代码入库的步骤**，采用 Phabricator 作为 CI 的驱动，并作为质量检查中枢，尽量提高入库前代码审查的流畅性。在这个过程中，Facebook 做到了以下几点：

测试的完整性。代码提交到 Phabricator 进行代码审查的同时，进行各种静态检查、单元测试、集成测试、安全测试，以及性能测试等。

工具的集成。Phabricator 提供的插件机制，可以跟其他系统和工具集成，以支持运行各种检查。

沙盒环境。代码在提交到 Phabricator 进行审查时，Phabricator 会自动产生一个沙盒环境。沙盒环境有两个好处：一是，可以让开发者之间进行联调；二是，可以让开发者并行地进行其他开发工作，因为在进行代码审查时，开发者的开发机器并没有被占用。

高效的代码审查。比如，代码审查不通过时，代码作者可以方便地在原来的提交之上进行修改，并在下一轮审查时只进行增量代码的审查。这就大大降低了每次代码审查的交易成本，从而保证了 CI 的顺畅性。

代码入库之后，进入持续交付步骤。Facebook 使用大仓，同一个仓中每天有几千个代码提交，所以持续交付的挑战很大。他们有一个专门的发布工具团队，自研了一套发布工具来实现自动化流水线，通过以下两点比较好地实现了流水线资源和测试完整性的平衡。

不针对每一个提交进行 CD 验证，而是按照一定时间间隔进行验证。因为提交太多，如果每个提交都进行构建打包，资源消耗实在太大，所以 Facebook 采用了按照一定时间

间隔，比如，每 10 分钟进行一次构建打包。这就大大降低了资源的消耗，不过这里有个问题，在验证步骤发现 Bug 时，因为验证的是最近 10 分钟的所有提交，所以不能精准定位造成问题的提交。
针对这个问题，Facebook 使用单主干开发分支方式，并强制在代码合并时，只能使用 git rebase 不能产生合并提交，所以提交历史是线性的，从而可以使用 git bisect 命令来自动化定位问题。这部分内容我会在下一篇文章中详细介绍。

对验证进行分级。也就是说，有几条不同的 CD 流水线，按照不同的时间间隔运行构建和检验。根据运行时间间隔的不同，它们运行的检验数量以及检查出来的 Bug 优先级也不同。间隔时间越长，运行的检验越全面，检查出来的 Bug 优先级越高。

这里需要说明的是，2017 年以前，Facebook 并没有把每一个在主干分支上成功通过流水线验证的软件包作为发布候选，而是在每周五的固定时间，从主干分支上拉出一个发布分支，稳定 3 天后上线。也就是说，这并不是严格意义上的持续交付。这是因为当时的自动化检验还不能确保产品达到上线要求。其实，这对很多公司来说都很常见，都需要一些额外的测试和检验来确保上线产品的质量。

最后，是**持续部署**的操作。在 2017 年以前，Facebook 并没有持续部署，而是采用的每周全量代码部署的方式。但到 2017 年，因为代码提交实在太多，每次周部署代码，处理的提交量会超过 10000，需要很长时间才能稳定发布分支，所以 Facebook 转向了持续部署。

具体的方法是，极致地进行自动化测试验证。关于实施细节，你可以参考 Facebook 的第一个发布工程师 Chuck Rossi 对持续部署流程的描述。

值得一提的是，跟持续交付一样，Facebook 的持续部署也不是纯粹的持续部署。因为代码提交太多，他们并没有每个提交都单独部署，而是采用类似持续交付的方法，把一段时间之内的提交一起部署。这种**不教条的方式，是我从 Facebook 学到的一个重要的做事方法。**

小结

Facebook 在 CI/CD 上做到了极致，对每一个代码提交都高效地运行大量的测试、验证，并采用测试分层、定时运行等方式尽量降低资源消耗。正因为如此，他们能够让几千名开发人员共同使用一个大代码仓，并使用主干开发，产生高质量的产品，从而实现了超大研发团队协同下的高效能。

在前面几篇文章中，我们多次提到“持续”。这个词，近些年在软件研发中比较流行，比如我今天与你分享的持续集成、持续交付、持续部署，加上持续开发，一共有 4 个了。

实际上，在 CI/CD 流水线中，**做为流水线的一部分，测试一直在运行并最快地给开发者提供反馈**。这正是另一个“持续”，也就是“持续测试”的定义。

“持续”如此重要的原因是，软件开发是一个流程，只有让这个流程持续运转才能高效。这里我把这 5 个持续都列举出来，方便你复习、参考。

	定义	关键点
持续开发	让开发者不受阻塞、不受不必要的干扰，从而全身心地聚焦在产品开发上	1. 规范化、自动化核心步骤 2. 快速反馈，增量开发
持续集成	在团队协作中，一天内多次将所有开发人员的代码合并入同一条主干	1. 流水线的测试要尽量完整 2. 流水线的运行速度一定要快 3. 流水线使用的环境，尽量和生产环境一致
持续交付	让软件产品的产出过程在一个短周期内完成，以保证软件保持在随时可以发布的状况	
持续部署	将每一个代码提交，都构建出产品直接部署给用户使用	
持续测试	测试是做为流水线的一部分，一直在运行并最快给开发者提供反馈	1. 服务化 2. 自动化 3. 分级

图 2 5 个“持续”方法定义与关键点对比

思考题

1. 在几千名开发人员共同使用一个大代码仓的工作方式下，做好 CI 有很大的挑战性。你觉得挑战在哪里，容易出现什么样的问题，又应该怎么解决呢？
2. 今天我提到了持续开发在 CI 中的作用，请你结合上一篇文章，思考一下持续开发和 CI/CD 是怎样互相促进的。

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再见！

上一篇　05 | 代码入库前：Facebook如何让开发人员聚焦于开发?

下一篇　07 | 分支管理：Facebook的策略，适合我的团队吗?

精选留言 (11)

写留言

刘晓光
2019-09-05

最大的困难有三个都在人的工作习惯上：有效且同期建设的单元测试;每天至少一次的push代码；轻量级频繁的code review

展开

作者回复: 第七篇文章会有一些相关讨论。希望对你能有些帮助。有问题继续讨论：）

1　1

Geek_1988
2019-09-05

使用git rebase的话，具体操作流程应该是

add /rm ->commit ->pull ->rebase ->push
这样子对吗

展开

作者回复: 对的。不过一般中间哪一个pull是使用fetch。因为pull会自动merge。所以，
add /rm ->commit ->fetch ->rebase ->push

1

师傅又被抓走了
2019-09-05

1 .在几千名开发人员共同使用一个大代码仓的工作方式下，做好 CI 有很大的挑战性。你觉得挑战在哪里，容易出现什么样的问题，又应该怎么解决呢?
容易出现代码合并冲突。出现冲突时的处理策略，回退or丢弃？冲突时，如何保证后续提交，可以正常合入?
功能依赖。功能模块大都不是独立的，如何保证双方一同合入？...

展开

作者回复: 请参考对Weining Cao问题的回复。

Weining Cao
2019-09-05

挑战在如何快速解决合并冲突。如何快速同步开发机器的代码保持最新。还有如果测试失败如何快速回滚。

展开

作者回复: > 挑战在如何快速解决合并冲突。
开发人员自己解决。提交做到原子性有很大帮助。

> 如何快速同步开发机器的代码保持最新。
经常`git fetch; git rebase origin/master`。

> 还有如果测试失败如何快速回滚。
1. 减少入库时的测试失败：多使用本地测试
2. 入origin/master库之后不会滚。写修复提交再赶紧push上去。实在不行，`git revert`命令产生一个新提交push上去。

Geek_1988
2019-09-04

怎样提高自动测试的覆盖率和性能，会是很大的挑战，各模块开发者对其他模块不熟悉，发生bug的机率较高，测试覆盖到位比较难。

业务变化过程中，及时发现不必要的测试，提高测试性能也比较重要。

展开

Johnson
2019-09-04

持续部署的描述那个链接打不开了

展开

作者回复: 这个可能需要科学上网

飞奔的骆驼
2019-09-04

持续序列流水线中进去下一个节点的标准是什么，实操中难点是，大家对自动化的结果没有信心，如何做到呢?

展开

高倩
2019-09-04

大公司对接的业务复杂多变，很多线上故障都出现在一些没有预估到的流程。如何提高CI，如何全面自动化覆盖回归，这个是个很大的挑战。

展开

作者回复: 能举一个“线上故障都出现在一些没有预估到的流程”具体又脱敏的例子吗?

Geek_93f953
2019-09-04

持续集成、持续交付、持续部署的核心区别在我看来是自动化测试能力：
CI 每天多次将多人开发的代码合并到主干，并进行构建、代码检查、冒烟测试
CDelivery 自动将CI的结果打包、在测试环境和类生产环境进行自动化测试
CDeploy 自动将CDelivery的结果进行回归测试，并按照预设的灰度策略部署到生产环境

展开


李双
2019-09-04

大公司的流程就是规范哈。如果一个小公司几十人的团队，是不是没必要搞这么多自动化，流程化，规范化。。。？

展开

作者回复: 赞同加牛不辣的观点。补充一点，在小公司的时候，流程和规范应该比大公司少一些，更加灵活一些。


5


Robert小七
2019-09-04

持续交付36讲专栏中说到持续交付包含持续集成和持续部署，老师你怎么看？

作者回复: 持续集成是在持续交付里面的。持续部署，要看具体语境中，持续部署具体怎么定义的（方法论的东西比较tricky）。