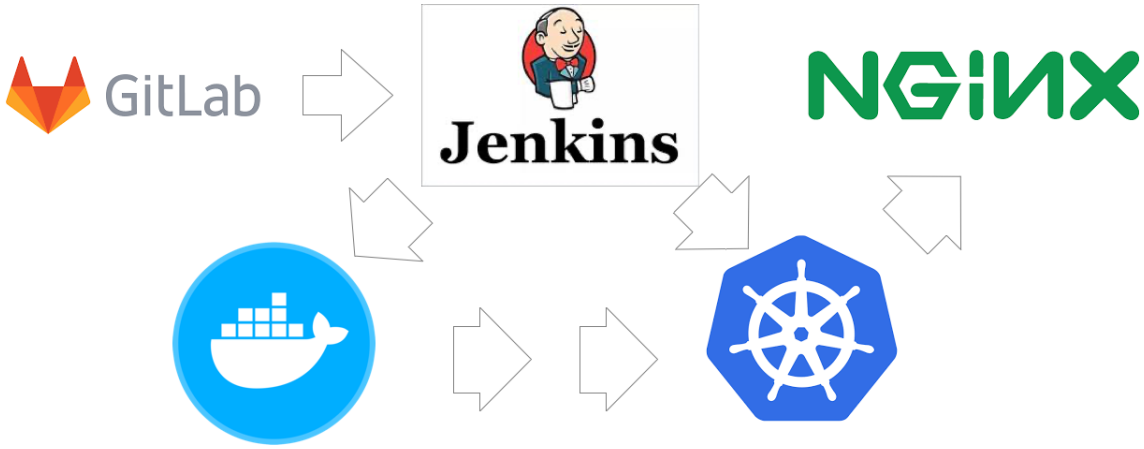
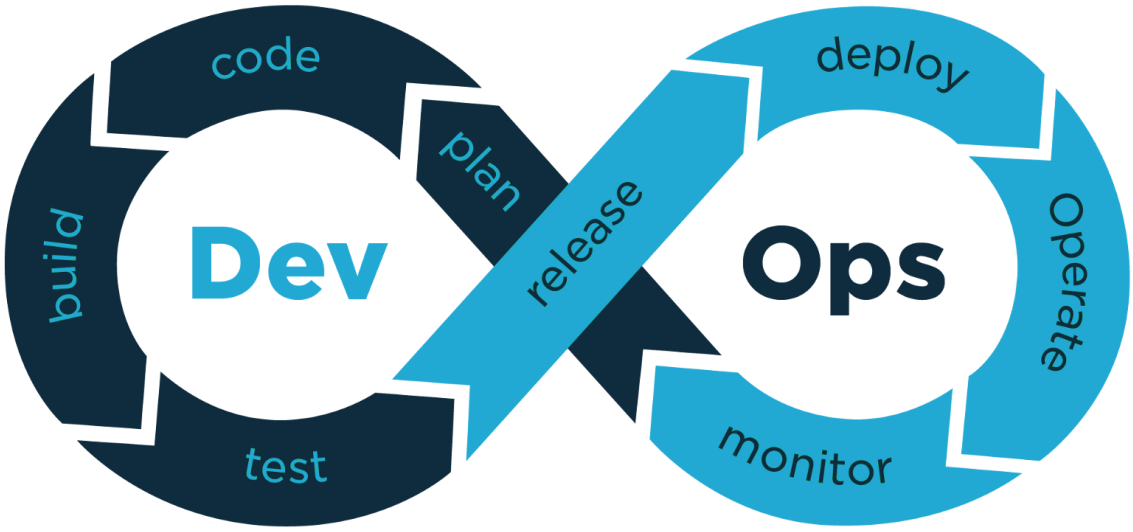
讲师(老司机)

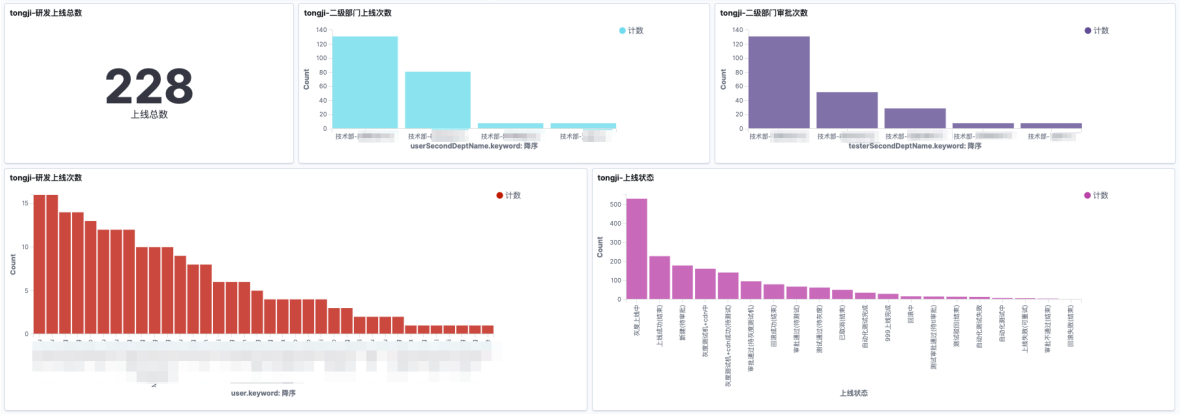
容器虚拟化技术和自动化部署



课程介绍

第一部分 Devops/CI/CD概述





从上线周报中可以看出，每周200+的构建成功，平均每天的构建次数在30+上线成功，各部门都有上线。究其原因是我们希望更好的对用户提供服务，希望对用户有益的想法，尽快的展现给用户，但是再展现给用户的时候，我们需要保证用户使用的稳定性和连续性。

那么怎样保证用户使用的稳定性和连续性，以此问题衍生出了很多实践方法。比较常见的入持续集成、极限编程、持续部署、持续交付、DevOps等等，接下来我们会粗略的过一下各个概念，以便让大家可以根据情况使用有益的实践

持续集成（CI）是一种**开发实践**，其中开发人员经常（最好每天几次）将代码集成到共享存储库中。然后可以通过自动构建和自动测试来验证每个集成。尽管自动化测试不是严格意义上的CI的一部分，但通常隐含了它。

定期集成的主要好处之一是，您可以快速检测到错误并更轻松地定位它们。由于引入的每个更改通常很小，因此可以快速查明引入缺陷的特定更改。

近年来，CI已成为软件开发的最佳实践，并遵循一系列关键原则。其中包括版本控制，构建自动化和自动化测试。

此外，持续部署和持续交付已成为**最佳实践**，可让您随时随地部署应用程序，甚至在每次引入新更改时甚至将主代码库自动推入生产环境。这使您的团队可以快速行动，同时保持可以自动检查的高质量标准。

> 持续集成并不能消除错误，但确实可以使查找和删除错误变得更加容易。 -- ThoughtWorks首席科学家Martin Fowler

在本指南中，我旨在揭开持续交付和DevOps的神秘面纱。我将解释这些做法，告诉您“在业务方面”它们对您有多重要，并帮助您参与其中。没那么复杂，我们拥有图片和所有内容。

> 您拥有技术，您是产品经理或MBA。您的团队A / B测试，功能切换和办公室里有只狗！当然，您了解什么是功能分支，什么是CD以及DevOps文化是什么样的。对？嗯...当然。
> 你已经敏捷了。现在，工程团队每周与您的产品人员见面，讨论故事和迭代。他们协作良好，正在建设的东西比以往任何时候都感觉更好。但是您的客户仍然无法更快地获得这些功能。您仍然必须等待释放火车离开车站。您听说过像Etsy，Flickr和Google这样的公司每天交付100次。他们是如何做到的呢?
> 您的开发团队希望“做CD”。您已经听到了一些好消息，但是您还担心更改将投入生产，而没有对其进行适当的测试或无法正确地将更改推向市场。这是什么CD东西?

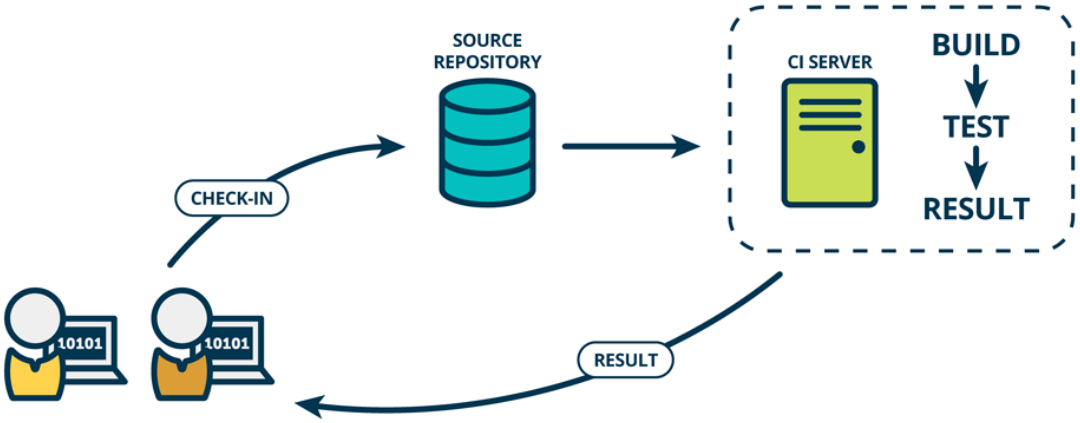
让我们从一些定义和示例开始

第1节 CI/CD

1.1 持续集成（CI）

在传统的软件开发中，集成过程通常在每个人完成工作后的项目结束时进行。整合通常需要数周或数月的时间，可能会非常痛苦。持续集成是一种将集成阶段置于开发周期中较早的做法，因此，构建，测试和集成代码的时间安排更为规则。

CI意味着一个开发人员（嗨，史蒂夫！）在家里用他的笔记本电脑编写代码，另一个开发人员（嗨，安妮！）在她办公室的台式机上编写代码，他们可以分别为同一产品编写软件，并将他们的更改集成到一起。一个叫做源库的地方。然后，他们可以根据各自编写的内容构建组合的软件，并测试其按预期方式工作。



开发人员通常使用称为CI Server的工具来进行构建和集成。CI要求Steve和Annie具有自检代码。这是用于自我测试以确保其按预期工作的代码，这些测试通常称为单元测试。集成代码后，当所有单元测试通过时，史蒂夫和安妮将得到一个绿色的版本。这表明他们已经验证了自己的更改已成功集成到一起，并且代码按测试期望的那样工作。但是，尽管集成代码可以成功地协同工作，但由于尚未经过测试和验证，可以在生产环境中使用，因此尚未投入生产。您可以在下面的“ 持续交付”部分中详细了解配置项之后的情况。

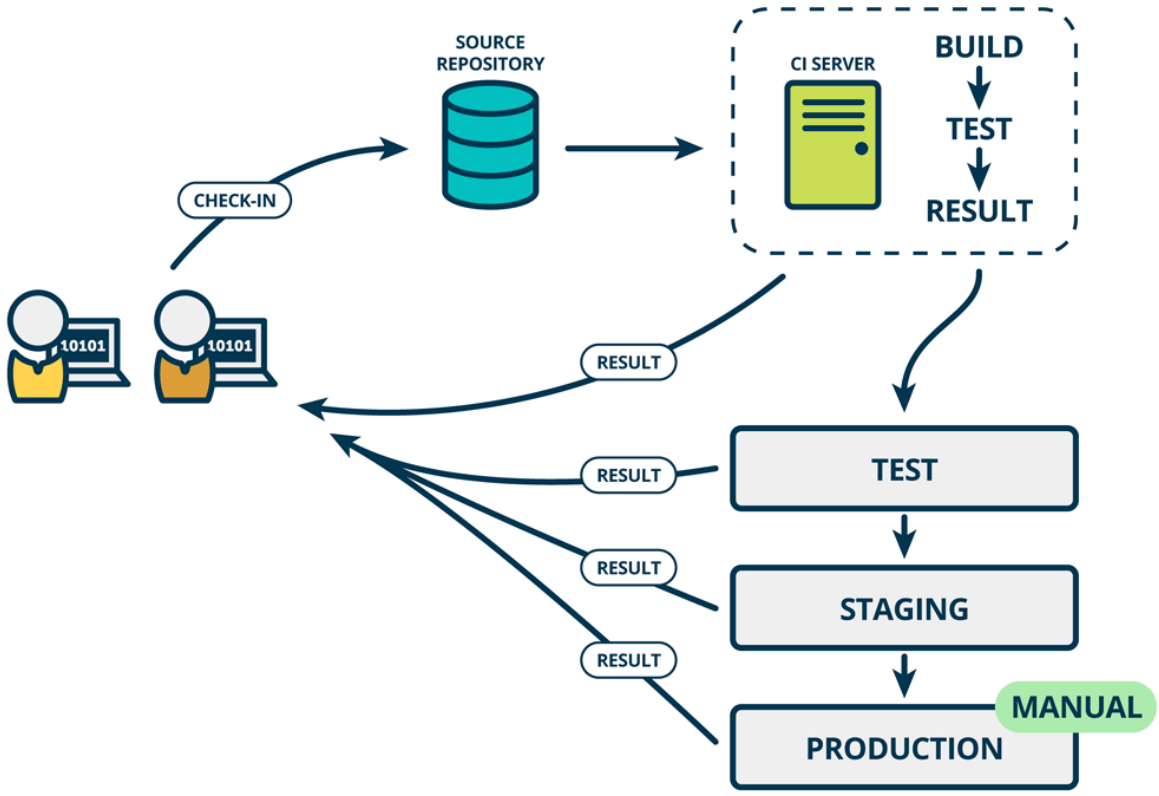
要被视为练习CI，Steve和Annie必须签入主源存储库，并频繁地集成和测试其代码。通常每小时一次，但每天至少一次。

CI的好处是集成成为非事件。一直在编写和集成软件。在CI之前，集成是在创建过程的最后一次完成的，并且花费的时间未知。现在有了CI，它每天都会发生，只需要几分钟，这只是“我们的工作方式”。

您的团队很可能正在执行CI（或者至少他们认为这样做）。您可以通过询问他们是否每天集成代码来进行确认-CI是进行持续交付所需的第一个实践。实际上，如果您曾经签入过帮助文本，文档或图像，那么您可能一直在进行整合！您还可以看到“您正在掌握持续集成的7个迹象”以获得更多确认。

1.2 连续交付（CD）

让我们回到我们的两个开发人员Steve和Annie。持续交付意味着Steve或Annie每次更改代码，集成并构建代码时，他们还将在与生产非常相似的环境中自动测试该代码。我们将此部署到不同环境并在不同环境上进行测试的过程称为部署管道。部署管道通常具有开发环境，测试环境和过渡环境，但是这些阶段因团队，产品和组织而异。例如，我们的Mingle团队有一个名为“ Cupcake”的阶段，这是一个登台环境，而Etsy的登台环境称为“公主”。

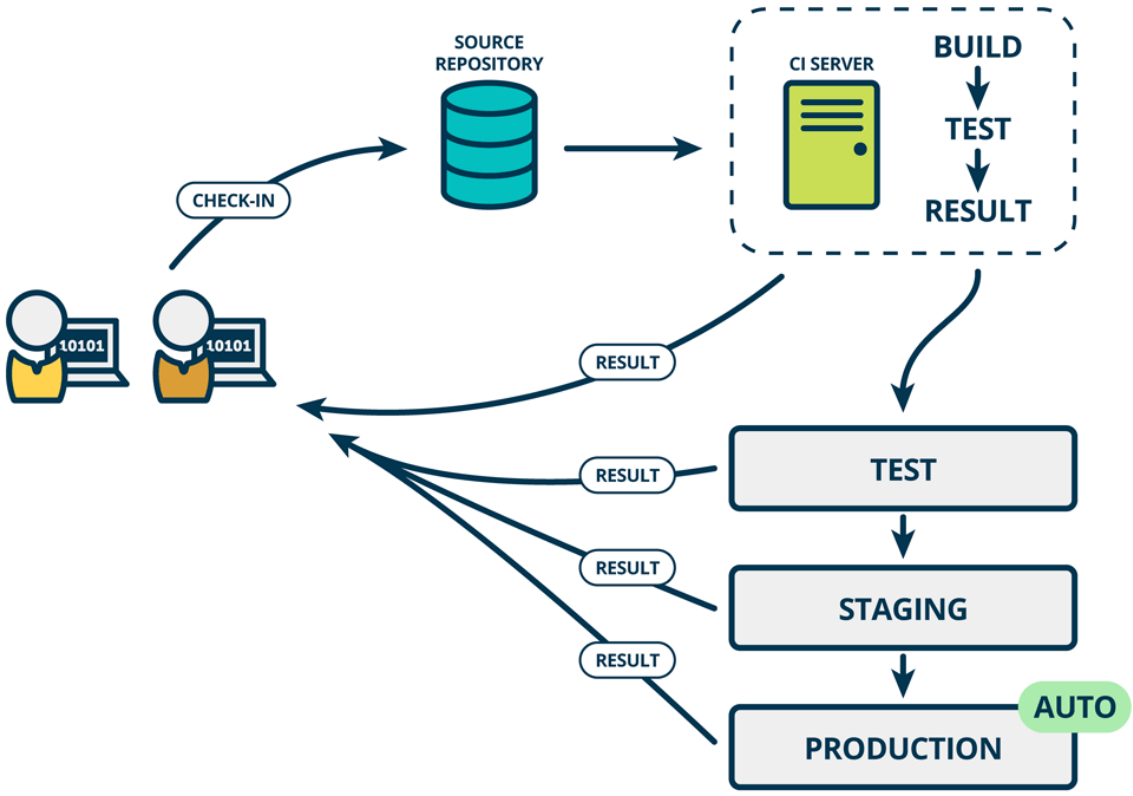


在每个不同的环境中，Annie或Steve编写的代码都经过不同的测试。这使他们和您越来越有信心，当代码在生产环境中部署时，它们将在生产环境中工作。至关重要的是，代码只提升到（上测试）在部署管道的下一个环境，如果 它通过以前的环境的考验。这样，Annie和Steve可以从每个环境中的测试中获得新的反馈，并且如果出现故障，他们可以更轻松地了解问题可能出在哪里，并在代码进入生产环境之前予以解决。

持续学习
这个过程对我们这个行业的人来说非常强大。这意味着，如果Annie的测试在所有环境中都通过了，您就会知道她的代码在投入生产时可能会按预期工作。一旦测试在所有环境中通过，您就可以立即决定最终用户是否通过测试。我们现在要在生产中使用这种绿色产品吗？是! 因此，一旦开发人员完成构建，便可以立即为客户提供经过全面测试的全新工作软件。！

1.3 持续部署

在这种实践中，Steve或Annie所做的每一项更改都通过了所有测试阶段，并 自动投入生产。蒂姆•菲茨（Tim Fitz）最初创造了很好的解释。一些公司这样做，而另一些则没有。要实现连续部署，您首先需要进行连续交付，因此在开始练习连续部署之前，先决定哪个对您更合适，这不是优先级。无论哪种方式，我认为持续交付都是为了增强整个业务的能力，因此至少您应该参与确定是否应该使用持续部署。毕竟，如果您正在阅读本文，那么您可能处于“业务方面”。

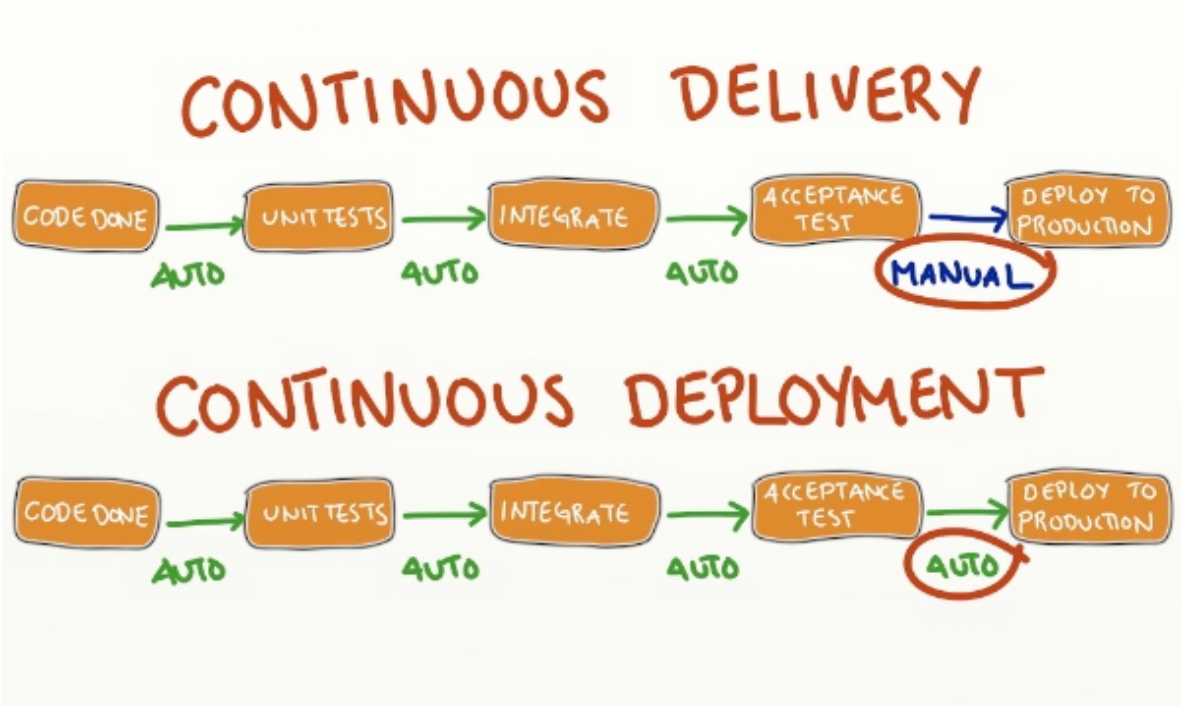


持续集成，持续部署和持续交付之间有什么区别?

持续集成（Continuous integration）
这种做法是将团队中不同开发人员的变更尽早集成到主线中，最好每天进行几次。这样可以确保各个开发人员处理的代码不会转移太多。当您将流程与自动化测试结合在一起时，持续集成可以使您的代码变得可靠。

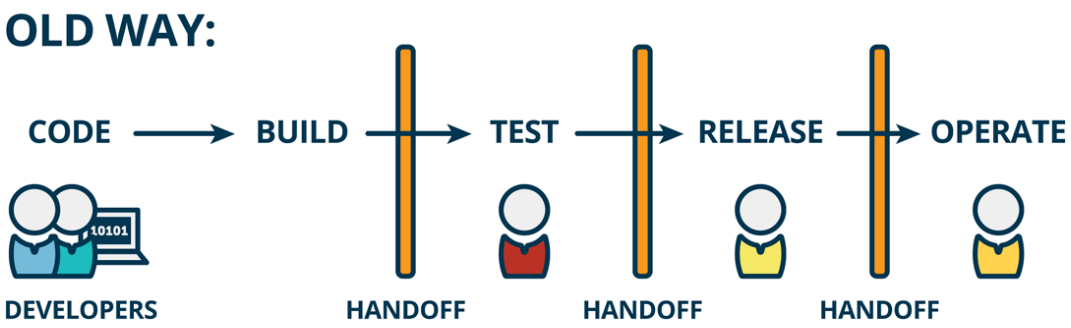
持续交付（Continuous delivery）
保持代码库随时可部署的做法。除了确保您的应用程序通过自动化测试外，它还必须具有将其投入生产所需的所有配置。然后，许多团队会进行推送更改，以立即将自动化测试传递到测试或生产环境中，以确保快速的开发周期。

持续部署（continuous deployment）
与持续集成密切相关，是指保持您的应用程序可随时部署，甚至在最新版本通过所有自动化测试的情况下，甚至自动发布到测试或生产环境。



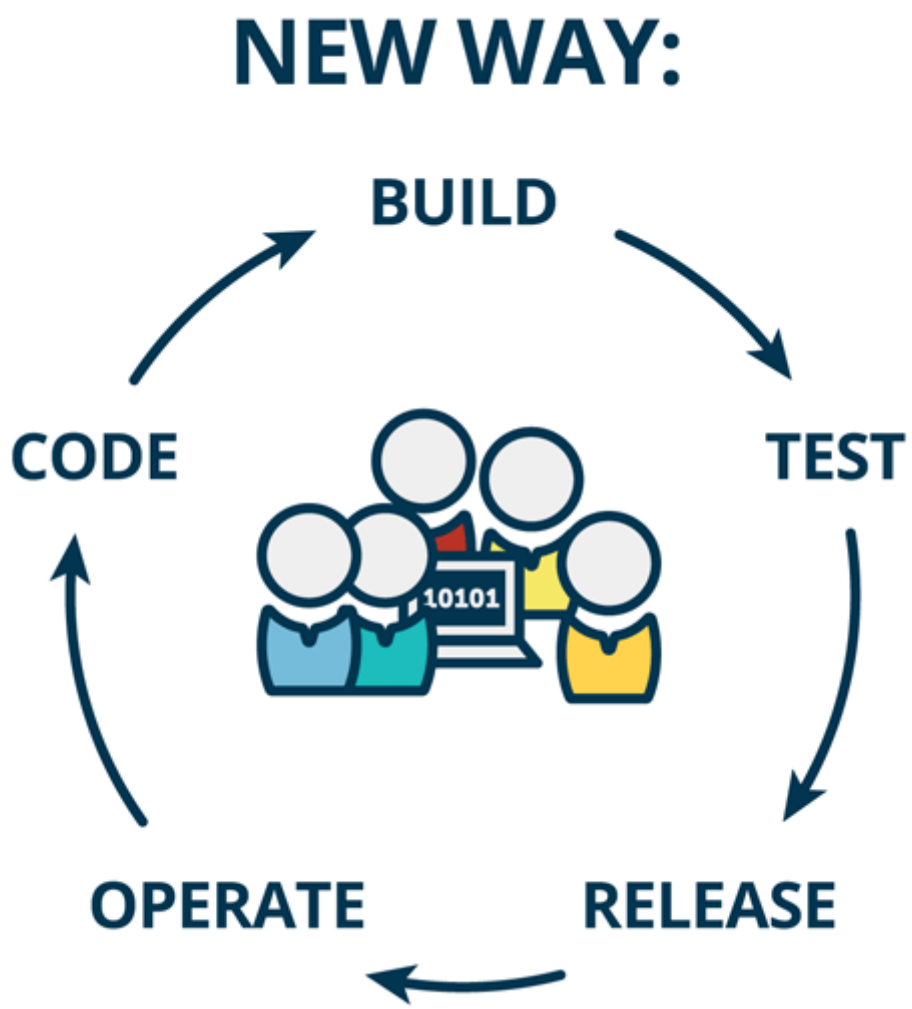
1.4 DevOps

“ DevOps”一词来自“Development”和“Operations”的组合。DevOps是一种文化，可促进开发人员与其他技术专业人员之间的协作，通常称是运维。具体来说，是在软件交付和部署过程中进行通信和协作，目的是更快，更可靠地发布质量更好的软件。
具有所谓DevOps文化的组织的共同特征是：自主的多技能团队（Steve，Annie和Joey都在同一团队中），高水平的测试和发布自动化（连续交付）以及两者之间的共同目标多技术成员。



您可能会看到此方法在您的组织中起作用的一种方式是，我们的开发人员朋友Steve和Annie将与 Joey 等操作人员合作，将软件交付生产，而不是在完成后将代码“交给” Joey进行发布。它。同样，史蒂夫（Steve），安妮（Annie）和乔伊（Joey）都将作为共同产品或服务团队的一部分，共同负责产品的支持和维护，而不是仅仅由运维团队负责。

您还将看到活动的自动化对于进行CD和DevOps的组织越来越重要。这是因为，为了实现CD和DevOps期望的可重复，定期且成功的软件发布过程，组织必须转向自动化流程。手动流程太容易出错并且效率低下。



DevOps文化通常与持续交付相关联，因为它们都旨在增强开发人员和运营团队之间的协作，并且都使用自动流程来更快，更频繁，更可靠地构建，测试和发布软件。这些都是像我们这样的人想要的东西。尽管开发团队经常看到流程改进的最直接好处，但CI，CD和DevOps对我们其他人来说却有很多好处。简而言之，我相信实践CD并拥护DevOps文化的组织将更频繁地向其客户提供更有价值，更可靠的软件。

1.5 有用的术语

签到
将本地开发代码推送到公共源存储库的过程。

CI服务器
用于构建和测试源代码的工具。CI服务器将告知开发人员最新的代码构建是否成功以及是否继续通过测试。

开发环境
开发人员在哪里创建，集成，构建和测试代码。

部署管道/管道
这是Steve和Annie的代码更改在完成并准备交付生产之前要经历的一组阶段。通常，这些将是“构建”，“单元测试”，“功能测试”，“性能测试”和“部署”。不同的自动化测试将在不同的阶段运行。一旦代码通过了整个部署管道，就可以将软件交付生产。

绿色建筑
绿色表示成功。绿色版本或内部版本已通过开发和交付过程的特定阶段的测试。通常，除非软件是“绿色”的，否则不会将其内部版本升级到部署管道的下一个阶段。绿色版本的反面是红色版本（请参见下文）。

增量发展
不要与迭代开发相混淆（见下文）。增量开发是一次构建一点产品直到全部完成的过程。在每个增量中添加片段，这些增量可以小也可以大。您可以将CI与增量开发结合使用，但是通过增量开发很难实现持续交付或连续部署，因为您必须等到所有增量完成后才能交付价值。Jeff Paton的《蒙娜丽莎》（Mona Lisa）很好地说明了增量开发与迭代开发之间的区别。

积分
个人或团队编写的所有代码都需要合并。我们称这种集成。在持续集成中，我们通常意味着需要定期整合个人的软件。在连续交付中，我们通常意味着将来自不同团队的软件集成在一起以创建整个产品。

迭代开发
不要与渐进式开发相混淆（请参见上文）。迭代开发是一次构建一点产品并对其进行完善直到完成的地方。该产品是迭代构建的，每次迭代都对相同的部分进行重新加工。预期并计划在不同迭代中的功能之间进行更改。您可以将CI，持续交付或持续部署与迭代开发结合使用。Jeff Paton的《蒙娜丽莎》（Mona Lisa）很好地说明了增量开发与迭代开发之间的区别。

主/干线/主线
“ master”，“ trunk”或“ mainline”分支是源存储库的主要分支。大多数人都进行基于主干的开发， 这意味着他们将始终将其更改集成到主线中。当个别开发人员将拥有自己的分支，或者团队将具有针对不同功能的分支时，其他人将进行基于分支的开发。

生产环境
这是部署或发布软件的地方。使用您的产品或网站的客户最有可能使用此环境。也称为“生产中”，“生产中”或“实时”。

红色打造
红色表示失败。红色版本或内部版本表示尚未通过开发和交付过程的特定阶段的测试。通常，如果软件的版本或版本为“红色”，则不会升级到部署管道的下一个阶段。红色构建的相反是绿色构建（请参见上文）。

源库
这就是源代码所在的地方。史蒂夫和安妮拥有自己正在处理的代码的本地版本（即在自己的计算机上），但是在开发人员签入对其所做的更改之后，源存储库将包含所有代码。

测试自动化
需要高质量的测试自动化来进行持续集成和持续交付。测试是检查软件是否按预期运行的方法。自动化测试是经过编码的测试，一旦将代码检入公共源存储库中，它们就会自动运行。

在CI领域，每次集成和构建软件时都会运行单元测试。如果测试未通过，则您的软件版本被确定为“不工作”，“红色”或“损坏”。在某些工作场所中，发生这种情况时会出现“红灯”或悲伤的声音。

如果构建损坏，则Steve或Annie（无论谁犯了故障代码）都需要“修复”，“使其绿色”或“使其正常工作”。他们可以通过对代码进行更改以修复它或删除破坏它的先前更改来实现。

单元测试
单元测试是代码中的自动化测试，用于测试低级的单个代码段，以确保它们可用并按预期工作。单元测试被认为是实践CI和CD的前提条件。

第4节 DevOps

前边我们大概讲解了CI/CD的概念，那么具体实施思想，我们通过DevOps展开来说

4.1 敏捷、持续交付和三步法

我们侧重于这些理论和原则，它们记录了制造业、高可靠性企业、高信任管理模型等十几年的经验，DevOps 实践正式基于这些经验衍生而来的。具体的原则和模式以及技术价值流中的应用，后再后面讲解。

4.1.1 制造业价值流

精益中的一个基本概念叫价值流。先在制造业的场景中定义它们，再讨论如何将它的应用到 DevOps 和技术价值流中。

> Jaren Martion 和 Mike Osterling 曾在 Value Stream Mapping 一书中把价值流定义为 **"一个组织基于客户的需求所执行的一系列有序的交付活动"**，或者是 **"为了给客户设计、生产和提供产品或服务所需从事的一系列活动，它包含了信息流和物料流的双重价值"**。

在制造业的流程中，价值流所处可见，它始于接收到客户订单并将原草料发往工厂。为了缩短和预测价值流中的前置时间，通常需要持续地关注如何建立一套流畅的工作流程，包含缩小批量尺寸、减少在制品（Work in Process, WIP）数量、避免返工等，同时还需要确保不会将次品传递到下游的工作中心，并持续不断地基于全局目标来优化整个系统。

4.1.2 技术价值流

在制造业中加速物理产品加工流程的原则和模式，同时可以应用到技术工作（及所有知识工作）中。在 DevOps 中，通常将技术价值流定义为 **"把业务构想转换为客户交付价值的、由技术驱动的服务所需要的流程"**。

流程的输入是既定的业务目标、概念、创意和假设，始于研发部门接受工作，并将它添加到待完成工作列表中。

接受了工作之后，研发团队将运用敏捷或迭代的开发流程，将那些想法转化为用户故事以及某种功能性说明，然后通过编写程序代码实现，再将代码签入到版本控制库中，接下来每次变更都将集成到软件系统并进行整体测试。

应用程序或服务只有在生产环境中按预期正常的运行，并为客户提供服务，所有的工作才产生价值。所以，我们不但要快速的交付，同时还要保证部署工作不会产生混乱和破坏，如中断客户服务、性能下降或者信息安全不合规等问题。

4.1.2.1 聚焦于部署的前置时间

部署的前置时间是价值流的一个子集，也是本书探讨的重点。价值流始于工程师（包括开发、QA、IT 运维和信息安全人员）向版本控制系统中提交了一个变更，止于变更成功地在生产环境中运行，为客户提供价值，并产生有效的反馈和监控信息。

一般分为2个阶段：

- 第一个阶段，工作只要包括设计和开发，它和精益产品开发有很多相似之处：都具有高度的变化性和不确定性，不仅需要创意，某些工作还可能无法重来，这导致无法确定总体处理时间。
- 第二个阶段，工作主要包括运维和测试，它类似于精益制造。相比前一个阶段，它需要创造性和专业技能，力求可预见性和自动化，将可变性降到最低（如短的和可预测的前置时间，接近零缺陷），并满足业务目标。

我们并不提倡在设计、开发串行的完成大批量的工作后，再转入测试、运维阶段（如使用大批量、基于瀑布模式的开发流程，工作在长生命周期的特性分支上）。恰恰相反，我们的目标是采用测试和运维与设计和开发同步的模式，从而产生更快的价值流和更高的质量。只有当工作任务是小批量的，并将质量内建到价值流的每个部分时，这种同步的模式才能实现。

> 实际上，使用类似测试驱动开发的技术，测试甚至可以发生在编写第一行程序代码之前

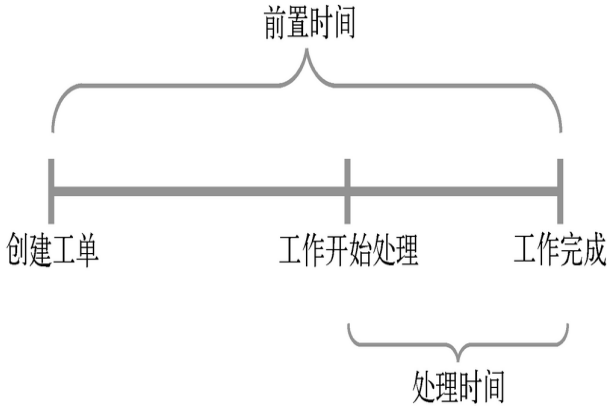
1.定义前置时间和处理时间

在精益社区里，前置时间与处理时间（有时候也被称为接触时间或者任务时间）是度量价值流性能的两个常用指标。

> Karen Martin 和 Mike Osterling 曾说：“为了避免混淆，我们不使用 “循环时间” 这个词，因为它还有其他的同义词——处理时间、输出速率等输出频率等。” 同理，本书中主要使用 “处理时间” 一词。

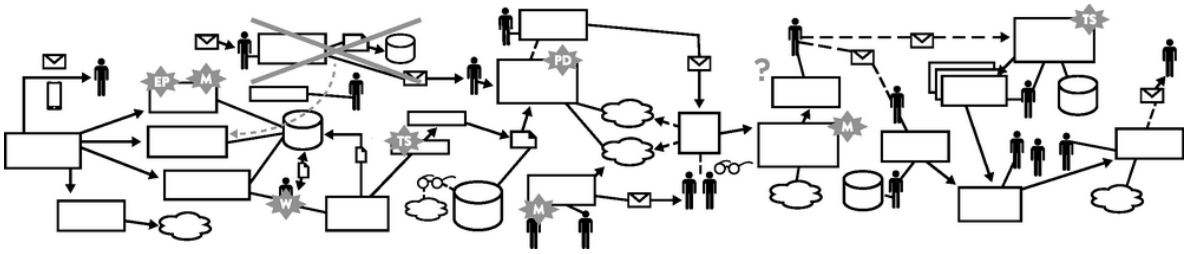
前置时间在工单创建后开始计时，到工作完成时结束；处理时间则从实际开始处理这个任务时才开始计时，它不包含这个工作在队列中排队等待的时间。

因为前置的实际是客户能够体验到的时间，所以包整点放在缩短前置时间而不是处理时间上。不过，处理时间与前置时间的比率是十分重要的效率指标，为了实现快速的流动并缩短前置时间，必须缩短工作在队列中的等待时间。



2.常见的场景: 为期数月的部署前置时间

通常，部署前置时间动则需要好几个月。在大型，复杂的企业里，使用着紧耦合的单体应用，少有集成测试的环境，测试和生产的前置时间很长，并且严重依赖于手动测试，或者需要各种审批流程，情况更是如此。



部署前置时间一旦变长，那么在价值流的每个阶段，几乎都需要 “填坑” 能手动来补救。通常很可能是项目结束前，将开发团队的变更合并到一起后，才发现整个系统根本无法正常工作，有时甚至会出现代码都无法通过编译和测试的情况。每一个问题都可能需要几天甚至几周的时间来定位和修复，因此导致了极其糟糕的客户体验。

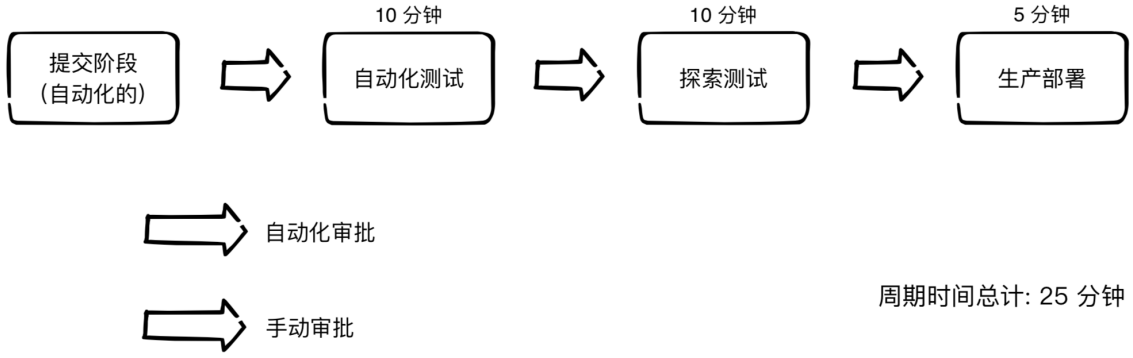
3.我们的目标: 分钟级别的部署前置时间

在 DevOps 的理想情况下，开发人员能够快速、持续地获得工作反馈，能快速和独立地开发集成和验证代码，并能将代码部署到生产环境中（自己部署或者他人部署）。

我们可以通过如下方式达到这个目标：向版本控制系统中不断地提交小批量的代码变更，并对代码做自动化测试和探索后，然后再将它部署到生产环境中。这样，就能对代码变更在生产环境中的成功运行保持高度自信，同时还能快速的发现并修复可能出现的问题。

为了更容易的实现上述目标，还需要通过模块化、高内聚、低耦合的方式优化架构设计，帮助小团队自治地工作。这样即使失败了，也能在可控范围内，而不至于对全局产生影响。

通过上述方式，能有效的将前置时间缩短至分钟级别；即使还在最坏的情况下，也不会超过小时级别。其价值流图为：



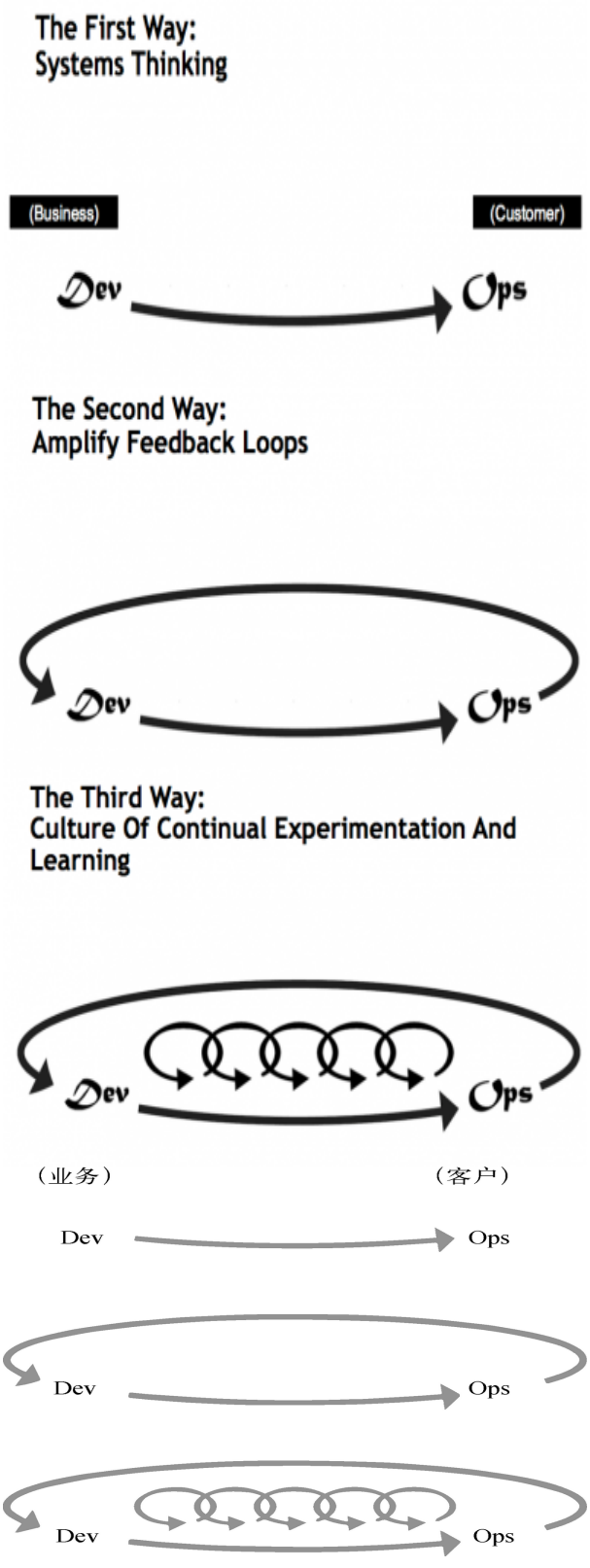
前置时间为分钟级别的技术价值流

4.1.2.2 关注返工指标——%C/A

除了前置时间和处理时间外，技术价值流中的第三个关键指标是完成时间和精确的总花费时间的百分比（%C/A）。该指标反映了价值流中的每个步骤的输出质量。Karen Martin 和 Mike Osterling 描述道: ”要获取 %C/A，可以询问下游客户他们有百分之多少的实际收到了 '真正有用的工作'，即使他们可以专心做有用的工作，而不必修复错误信息、补充信息、或者澄清那些本该确定的信息。“

4.1.3 三步工作法: DevOps 的基础原则

《凤凰项目》 把三步工作法作为基础的原则，并由此衍生除了 DevOps 的行为和模式:



第一步，实现开发到运维的工作快速地从左向右流动。为了最大程度的优化工作流，需要将工作可视化，减小每批次大小和等待间隔，通过内建质量杜绝向下游传递缺陷，并持续地优化全局目标。

通过加快技术价值流的流速，缩短满足内部或者外部客户需求所需的前置时间，尤其是缩短代码部署到生产环境所需的时间，可以有效地提高工作质量和产量，并使企业具有更强的外部竞争力。

相关的实践包括持续构建、集成、测试和部署，按需进行环境搭建，限制在制品数量，构建能够安全地实施变更的系统和组织。

第二步，在从右向左的每个阶段中，应用持续、快速的工作反馈机制。该方法通过放大反馈环放置问题复发，并能缩短问题检测周期，实现快速修复。通过这种发誓，能从源头控制质量，并在流程中嵌入相关的知识。这样不仅能创造出更安全的工作系统，还可以在灾难事故发生前就检测到并解决它。

即使发现并控制这些问题，直到拥有有效的对策，可以持续地缩短反馈周期和放大反馈环，这是所有现代流程优化方法的一个核心原则，能够创造出组织学习与改进的机会。

第三部，简历具有创意和高可信度的企业文化，支持动态的、严格的、科学的实验。通过主动承担风险，不但能从成功中学习，也能从失败中学习。通过持续地缩短和放大反馈环，不仅能创造更安全的工作系统，也能承担更多的风险，并进行测试帮助自己比竞争对手改进改进得更快，从而在市场竞争中战胜他们。

作为第三部的一部分，我们能够让工作系统事半功倍，将局部优化转化为全局优化。另外，不管是谁参与了工作，所有经验都可以持续地积累，组织里的人都可以互相借鉴彼此的经验和智慧。

4.2 第一步: 流动原则

在技术价值流中，工作通常是从开发人员流向运维人员，也就是业务和客户之间的所有职能部门。本章要讲述的第一步工作法，就是建立从开发到运维之间快速的、平滑的、能向客户交付价值的工作流。要为这个全局目标进行优化，而非围绕一系列全局目标，如功能开发的完成度、测试中问题的发生率和修正率、运维维护的可用性等。

通过持续加强工作内容的可视化，减小每批次大小和等待间隔，内建质量以防止缺陷向下游传递，从而增强流动性。通过加速技术价值流的流动，可以缩短满足内部客户和外部客户需求的前置时间，进一步提供工作质量，并使我们更加敏捷，能够比竞争对手更为出色。

我们的目标是在缩短代码从变更到生产环境上所需时间的同时，提高服务的质量和可靠性。实际上，可以在制造行业中找到价值流应用的相关线索，帮助我们将精益原则应用到技术价值流中。

4.2.1 使工作可见

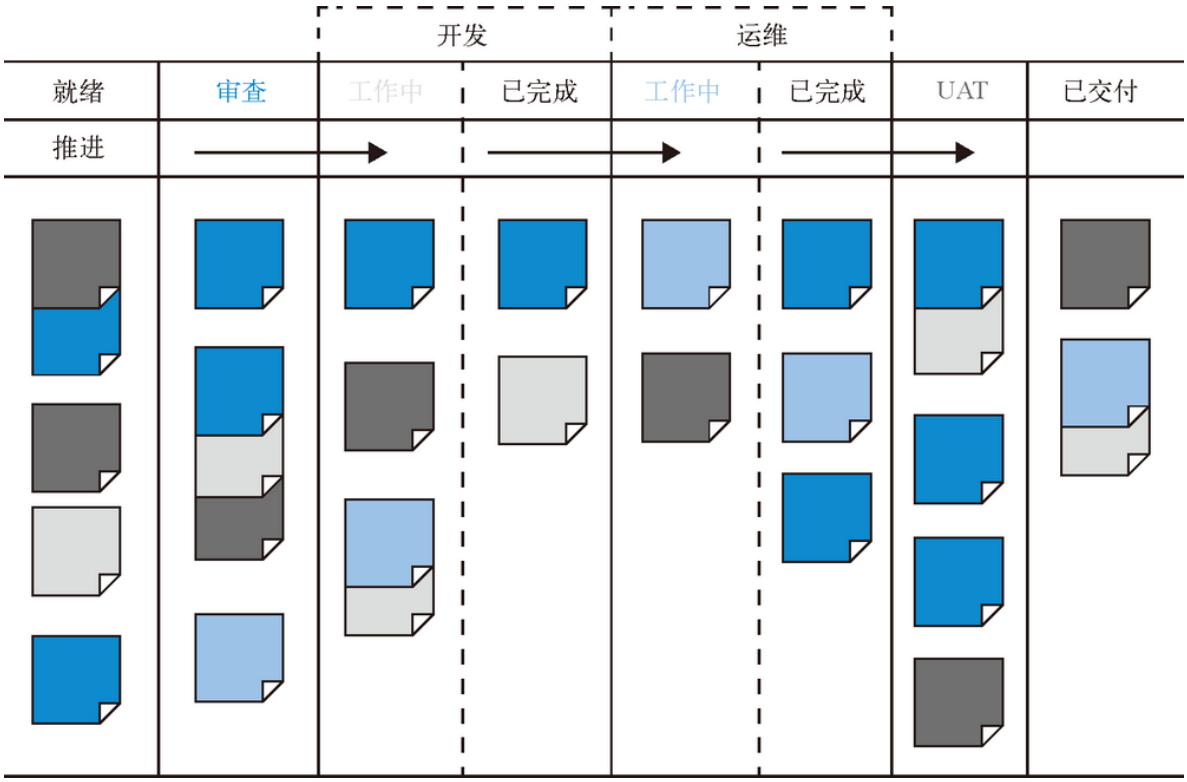
技术行业的工作内容是不可见的，这是其与制造业价值流相比的一个显著差异。相对于工业产品的生产过程而言，在技术价值流中很难发现工作过程中的阻塞点，比如，在哪里受阻了，在哪个环节产生了积压。而在制造业的价值流中，工作在不同工作中心间的转移通常是显而易见并且缓慢的，因为必须真正地转移库存产品。

另一方面，技术工作的流转通过点击一次鼠标就可以完成，譬如将工单重新指派给另一个团队。由于点击的操作太过容易，所以不同团队可能会因为信息不完整而将工作 "踢来踢去"，存在的问题也会被传递到下游工序，而这些问题完全是不易察觉的，知道无法按时间向客户交付产品，或者应用程序在生产环境中出了问题。

为了能辨别工作在哪里流动、排队或停歇，就需要将工作尽可能地可视化。可视化工作板是一种较好的工作方式，如在看板或 Sprint 计划板上，使用纸质或电子卡片将各项工作展示出来。工作通常从左侧发起（从代办事项中拉取），然后从一个工作中心拉取到下一个工作中心（用列表示），最后到达工作看板的最右侧，而这一列通常被标记为 "完成" 或 "已上线"。

通过这种方式，不仅能将工作内容可视化，还能有效地管理工作，加速其他从左至右的流动。此外，还可以通过卡片从在看板上创建到移动至 "完成" 这一列，度量出工作的前置时间。

理想情况下，看板应该覆盖整个价值流；仅当工作到达看板最右侧时，才能算是已完成。开发完成某个功能不能算是 "已完成"，只有应用程序在生产环境里成功地运行起来，并开始为客户提供价值的时候，才能算是 "已完成"。



通过将每个工作中心的所有工作都放进队列中，并且可视化地展示出来，利益干系人更容易从全局目标出发，确定各项工作的优先级。这样，每个工作中心都能采用单任务的处理方式，从优先级更高的任务开始，一次完成所有工作，以增加工作中心的吞吐量。

4.2.2 限制在制品数

这是一个及其重要的环节，而且往往是一个最容易让人忽略的环节
制造业的日常工作通常是由定期（如每天、每周）生成的产品计划决定的，根据客户订单，交货日期、零件库存等条件，确定执行哪些任务。

但技术工作通常是动态的——尤其是存在共享服务的情况下，团队必须要同时满足很多利益干系人的需求，这导致临时安排控制了日常工作。紧急的工作可能会来自于各种渠道，譬如工单系统，宕机告警、电子邮件、电话、即时通信的消息或管理层决定的事件。

生产中断在制造业里很显眼，且代价极高，当正进行中的工作戛然而止时，所有的半成品都将报废，然后再启动一批新作业。这种高昂的代价，让人们不希望中断频繁发生。

但是技术工作者很容易被打断，因为对所有人而言，这个中断的后果似乎是不可见的，即便它对生产效率的影响比制造业更甚。例如，将一个工程师同时分配到多个项目里，他不得不在多个任务，认知规则和目标之间来回切换，付出重新进入角色的成本。

研究证明，即便是完成简单任务，如将各种几何形状分类，当同时执行多个任务时，效率也会显著降低。从认知上看，技术价值流中的工作，显然要比分类几何形状复杂得多，所以多任务会导致更长的处理时间。

当使用看板管理工作时，可以限制多任务的出现，例如对看板的每一列或每个工作中心设置在制品数量的限制，并把卡片数量的上限标记在每一列上。

例如，将测试工作的在制品数量上限设置为 3。当测试队列中已有 3 张卡片时，除非某张卡片完成了，或将 3 张中的一张退回到前一个队列（左侧的那一列），否则禁止添加新卡片。另外，在把一项工作用卡片的形式显示在看板上之前，任何与之相关的功能都不能开展，这强调了任何工作都必须可视化。

Dominica DeGrandis 是在 DevOps 中运用看板的专家之一，他指出：“控制队列的长度（即在制品数）是一个非常强大的管理工具，因为这是影响前置时间的重要因素之一——对于大多数的工作条目而言，在它们完成以前，其实并无法预测到底需要多长时间。”

通过限制在制品数，还能更容易地发现工作中的阻碍。例如，当限制在制品时，可能会发现居然没什么工作可干，因为要等待其他人。虽然进行一项新工作（即 “干点什么总比什么都不干强”）可能很诱人，但此时更好的办法是查明导致等待的原因，并协助解决那个等待的问题。实际上，糟糕的多任务处理发生的原因，通常是同是给一个人分配多个项目，造成了很多优先级冲突问题。

正如《看板方法: 科技企业逐渐变革成功之道》的作者 David J. Anderson 所说: "停止开始，开始结束"。

4.2.3 减小批量大小

建立平滑而快速的工作流的另一个关键点，是通过小批量的模式完成工作。在精益革命之前，大批量（或规模）生产的方式在制造业司空见惯，在作业配置或作业之间的切换相当耗时且昂贵时尤其如此。例如，在生产大型汽车时，需要将巨大而沉重的模具放到金属冲压板上，这个过程可能需要好几天时间。鉴于如此高昂，通常会用大批量作业，一次冲压出尽可能多的车身版，从而减小模具的更换次数。

然而，大批量导致了在制品的暴涨，并在整个制造工厂中产生流量级联的变化。最后导致前置时间长、产品质量差的后果——如果发现了一个车身版有问题，整个批次都必须报废。

在精益中，一个重要的经验是：为了缩短前置时间和提高交付物质量，应当持续不断地追求小批量模式。理论上，最小的批量是单件流，也就是每次操作只执行一个单位产品的处理。

也称为“1的批量大小”或“1×1流量”，该术语表示批量大小和在制品都限制为1。

关于小批量和大批量之间的巨大差异，James P. Womack 和 Daniel T. Jones 在《精益思想》一书中，通过 “模拟邮寄宣传册” 的经典案例进行了说明。

这个例子假设要邮寄出 10 本宣传册。邮寄之前，每本宣传册都必须经历 4 个步骤：折叠，插入信封，给信封封口，盖戳。

如果采用大批量策略（即 “大规模生产”），我们会对每个宣传册按顺序进行上述 4 个步骤。换句话说，首先要将 10 张纸全部折叠完，再将每张纸分别插入信封，然后给所有的信封封口，最后全部盖章。

另一种方式是小批量策略（即 “单件流”），即对每本宣传册顺序地执行所需的所有步骤，然后在开始处理下一本宣传册。换句话说，先折叠一张纸，将其插入信封，再给信封封口，之后盖章；然后，取下一张纸，并重复以上过程。

赛用大批量和小批量之间的差距是巨大的如下图。假设对所有 10 个信封都必须草去如上述 4 个步骤，并且每一步操作需要 10 秒。如果使用每批 5 个的大批量策略处理，则完成第一个盖戳的信封需要用 310 秒。

> 这里是“5×1流量”，即批量大小为5，在制品为1。由于这里是单人模拟的场景，并且一双手同时只能加工一个信封，所以在制品数量也只能为1。310秒 = (5 × 10 + 5 × 10) + (5 × 10 + 5 × 10) + (5 × 10 + 5 × 10) + 10。 ——译者注

大批量

单件流

更糟糕的是，假设我们在信封封口操作中发现第一步的折叠做错了，在这种情况下，我们能发现错误的最早时间在 200 秒之后，这样我们就不得不将这个批次的 10 个小册子再重新折叠并装回信封中。

相比之下，使用下批量策略时，仅用 40 秒就完成了第一个盖戳信的生产，比大批量决策块 8 倍。如果第一步出错了，只需要返工一个小册子。小批量生产的在制品更少，前置时间更短，错误检测更快，返工量更少。

对于价值流而言，大批量的副作用和制造业一样。我们制定了软件发布的年度计划，将一整年的开发成功一次性地都发布到生产环境中。这种大批量的发布会造成突发的、大量的在制品，导致所有下游工作中心大规模的混乱，其结构是流动性变差，质量下降。这和我们阐述的经验是类似的，即对生产环境 的变更越大，问题的定位和修复就越困难，修复时间也就越长。

> 多是指数据中心的运维部门。——译者注

Eric Ries 在 “创业经验教训”（Startup Lessons Learned）这篇文章中说：“在开发（或DevOps）流程中，批量大小是工作产品在不同阶段间移动的单位数。对于软件而言，最容易看到的是代码。当工程师签入代码时，他们就批量地处理了一定数量的工作。有许多控制批处理的方式，从持续部署要求的小批量，到相对传统的基于分支的大型模块开发，都是聚合多个开发人员几周或几个月所工作的代码。”

在技术价值流中，单件流可以通过持续部署实现。其中，每一个提交到版本控制系统的变更都会集成、测试并部署到生产环境。具体的实现方法，将在第四部分中进行详细描述。

本书强调的是端到端的价值流，只有在部署之后，把价值交付给客户了，一项工作才算完成，因此是持续部署。 ——译者注

4.2.4 减少交接次数

在技术价值流中，如果部署的前置时间以月作为周期单位，通常是因为要将版本控制系统中的代码部署到生产环境需要数百甚至数千个操作。实际上，代码在价值流流转的过程中，需要各个不同部门的协同才能完成相关任务，包括国能测试、集成测试、环境测试、配置服务器、存储管理、网络、负载均衡和信息安全加固等。

一项工作在团队之间交接时，需要大量的沟通——请求、委派、通知、协调，而且经常需要排优先级、调度、消除突破、测试和验证。这些工作可能还需要使用不同的工单系统或项目管理系统，编写技术模范文档，用会议室、电子邮件或电话的形式进行沟通，可能还涉及文件共享服务器、FTP 服务器和 Wiki 页面的使用。

实际上，上述流程中的每个环节都有其潜在的队列，当依赖不同价值流共享的资源（例如集中式操作）时，就会出现工作等待。这些请求的前置时间通常会很长，从而导致那些本应按按期操作完的工作持续地延期。

即使在最好的情况下，有些信息或者知识也不可能避免地在交接过程中丢失。经历了多次的交接后，问题的上下文和所支持的组织目标可能会完全丢失。例如，服务器管理员可能会收到一个关于创建用户账号的新工单，但是他并不知道是什么应用程序或服务会使用这个账号，为什么需要新建账号，其他的依赖关系是什么，或者这到底是不是一个重复劳动。

为了减少这类问题的出现，要么努力减少交接次数，要么用自动化方式执行大部分操作，要么重新调整组织结构，让团队不必依赖其他人就可以独立的为客户提供价值。因此，要通过减少队列的等待时间以及非增值工作的实际来增加流动性。

4.2.5 持续识别和改善约束点

为了缩短前置时间、提高吞吐量，需要不断地识别系统中的约束点，提高工作产能。 Goldratt 博士在《 Beyond the Goal 》一书中提到：“在任何价值流中，总是有一个流动方向、一个约束点，任何不针对此约束点而做的优化都是假象。”如果我们优化约束点之前的那个工作重心，那么工作必将在这个约束点上更快的积压起来。

反之，如果优化约束点之后的工作中心，那么它还会处于饥饿状态，等待约束点处于工作的约束。对于这种现象，Goldratt 博士给出了解决方案，定义了如下 “5 个关键步骤”：

- 识别系统的约束点；
- 决定如何利用这个系统约束点；
- 基于上述决定，考虑全局工作；
- 改善系统的约束点；
- 如果约束点已经突破了，请回到第一步，但要杜绝惯性导致的系统约束。

在 DevOps 的转型工程中，如果希望前置时间从月或季度缩短为几分钟，那么一般需要依次优化下面的约束点。

- 环境搭建：如果生产或测试环境的搭建总是需要数周或数月，则按需部署就无法实现。解决实施是按需建立完全自服务的环境，保证团队需要环境的时候，能同过自动化方式创建。
- 代码部署：如果代码的部署需要花数周或更长时间（譬如每次部署需要 1300 个手动，易出错的操作，涉及多达 300 名工程师），那么就无法按需部署。解决措施是尽可能自动化部署的过程，以便让任何开发人员都需要按需自动化地部署。
- 测试的准备和执行：如果每次代码部署都需要两周的实际来完成测试环境的准备和数据集的配置，手动执行所有的回归测试还需要另外四周时间，那么就无法实现按需部署。解决措施是实现自动化测试，这样才能在安全、并行地执行部署的同时，使测试的速度能跟上代码开发的速度。
- 紧密耦合的架构：如果架构是紧密耦合的，那也无法实现按需部署，因为每次要做代码变更时，工程师都不得不从变更审评委员会哪里获得执行变更的许可。解决措施是创建松散耦合的架构，这样开发人员才能安全、自主地进行变更，提高生产力。

如果能突破以上的约束点，那么接下来的约束有可能是开发部门或产品经理。因为我们的目标是让小型开发团队可以独立、快速、可靠地开发、测试和部署，并持续为客户创造价值，所以这些环节应该是约束点集中的所在。对于高绩效者来说，不管工程师处于开发、QA、运维还是信息安全岗位，他们的目标都是尽量提高生产力。

当约束点出现在开发阶段时，我们将仅受限于有多少创意精良的业务假设，以及能否开发必要的代码来用真实客户来测试这些假设。

以上所述的约束点在 DevOps 转型中是相当普遍的——在价值流中识别约束点的技术，诸如如何使用价值流映射和度量的方法，以后会详细描述。

4.2.6 消费价值流中的困境和浪费

丰田生产系统的先去之一新乡重夫认为，浪费是业务兴盛的最大威胁——精益中对浪费的常用定义是“使用了超过客户需求和他们愿意支付范围的任何资料或资源的行为”。他定义了制造业里 7 中主要的浪费类型：库存、过量生产、过度加工、运输、等待、移动和缺陷。

现代化的精益理念解释道：**"消除浪费"** 会有点贬义和不近人情的意味，我们的目标其实是想通过持续的学习来破除日常工作中的困境，从而更好地实现组织的目标。在本书的后续内容里，“浪费” 一次以为和这个更具现代感的定义，因为它更符合 DevOps 所期望的理想境界。

> Mary 和 Tom Poppendieck 在 《Implementing lean software Development: From Concept to Cash》一书中描述道：浪费和困境是软件开发过程中导致交付延迟的主要因素。

下面是该书中描述的关于浪费和困境的部分类型。

- 半成品：它值的是价值流里任何还没有彻底完成的任务（例如，需求文档或尚未审核的变更单）、处于队列中的工作（如等待 QA 审核或服务器管理员审核的工单）。部分完成的工作会逐渐地过期，随着时间的推移最终失去了价值。
- 额外工序：在交付过程中执行的、并未给客户增值的额外工作，可能包括哪些在下游工作中心从没使用过的文档，或是对输出结果做出的并不增值的评审或审批。额外工序不仅增加了处理的工作量，还增加了前置时间。
- 额外功能：在交付过程中构建的那些组织或客户完全不需要的功能（如 “镀金”）。额外功能增加了功能测试和管理的复杂度和工作量。
- 任务切换：将人员分配到多个项目和价值流里后，他们需要进行上下文切换，并管理工作之间的依赖关系，这会在价值流中耗费额外的工作量和时间。

- 等待：由于资源的竞争而在工作之间产生了等待，这将增加周期时间，延迟了向客户交付价值。
- 移动：信息或数据在工作重心之间移动的工作量。例如，在一个需要频繁沟通的项目里，团队成员实际上不在一起办公，无法坐在一起紧密协作，这时人员移动的浪费就产生了。另外，工作交接也会产生移动的浪费，需要额外的沟通来澄清所有歧义的部分。
- 缺陷：由于信息、材料或产品的错误、残缺或模糊，而需要一定的工作量来确认。残缺的生产和被检验出来的时间间隔越长，解决问题就越困难。
- 非标准或手动操作：需要依赖其他人的非标准的或手动的工作，例如使用不能自动化反复重建的服务器、测试环境和配置。理想情况下，任何依赖运维团队手动完成的操作，都应该配置自动化的、按需提供的，或者自助服务。
- 填坑侠：为了实现组织的目标，不得不把有些人和团队置于不太合理的处境，者甚至会成为他们的家常便饭（如半夜两点生产环境出现事故，连夜给软件版本提交了上百个工单）。

4.3 第二步: 反馈原则

第一步工作法描述的原则，使得工作能够在价值流中从左向右快速的流动。第二步工作法描述的原则，则使得在从右向左的每个阶段中能够快速、持续地获得工作反馈。我们的目标是建立安全和可靠的工作系统。

这一点对于复杂的系统尤其重要，在这种情况下，发现和纠正错误的最早时机通常是灾难性事件发生时，例如制造业工人在工作过程中受伤，或核反应堆的堆芯熔毁。

在技术行业，我们的工作几乎都发生在灾难性后果如影随形的复杂系统里。和制造业相似，通常只有在发生重大故障的时候，才能发现问题所在，例如遇到大规模用户服务终端，或安全漏洞导致客户数据泄露。

通过在整个价值流和组织中建立快速、频繁、高质量的信息流，包括反馈和前馈回路，可以让系统更安全。这样，就可以在规模较小、修复成本较低的情况下发现并修复问题，在灾难发生前翘楚问题，并创造出组织性学习氛围。同时，应该把失败和事故的发生视为宝贵的学习机会，而不是惩罚和责备的理由。为了实现上述目标，我们先探索复杂系统的本质，以及怎样才能使它更安全。

4.3.1 在复杂系统中安全的工作

复杂系统的一个重要特征是，无法将系统视为一个整体，去理解各个部分是如何结合在一起的。复杂系统的组件之间通常是紧耦合且紧密关联的，不能仅仅依据组件的行为来解释系统的行为。

Charles Perrow 博士研究了三里岛核事故，他发现没有人能了解核反应堆在所有情况下的行为，以及在何种情况下会发生故障。当核反应堆的一个组件出现故障时，很难将其与其他组件隔离，以不可预测的方式快速地流过阻力最小的路径。

Sidney Dekker 博士提出了一些关于安全的重要元素，他发现了复杂系统的另一个特点：相同的实际做两次，结果未必相同。也正是因为这个特点，即便施行了有价值的静态检查和最佳实践，还是不足以防止灾难发生。

复杂系统中的故障是存在且不可避免的。因此，无论在制造业还是信息技术行业，我们都必须设计出一个安全的工作系统，让员工能无所畏惧地开展工作，确保早在灾难性后果（例如员工伤害、产品残缺或负面的客户影响）发生之前，能快速检测出错误。

Steven Spear 博士在他的哈佛商学院博士论文中揭晓了丰田生产系统背后的因果机制。他认为，我们可能无法设计出绝对安全的系统，但是可以通过采取以下 4 项错误让复杂系统更安全地工作：

- 管理复杂的工作，从中识别出设计和操作的问题；
- 群策群力解决问题，从而快速地构建新知识；
- 在整个组织中，将区域性的新知识应用到全局范围；
- 领导者要持续培养有以上才能的人。

要在复杂系统中安全地工作，必须具备上述 4 种能力。接下来，我们将描述前两种能力及它们的重要性，同时还会探讨其他领域是如何实现这些能力的，以及如何在技术价值流中实现它们。第三项和第四项能力会在第 4 章中描述。

4.3.2 及时发现问题

在安全的工作系统中，要不断地对设计和假设进行验证。目标是更早、更快、以尽可能低的成本、从尽可能多的纬度增加系统的信息流，并尽可能清晰地确定问题的前因后果。能排除的假设越多，定位和解决问题的速度就越快，从而提高我们的顺应里、敏捷性以及学习和创新能力。

我们通过在工作系统中建立反馈和前馈回路的方式体现这一点。Peter Senge 博士在《第五项修炼: 学习型组织的艺术和实践》一书中，描述了反馈回路是学习型组织和系统思维的重要组成部分。反馈和前馈回路能让系统内各部件之间的关系增强或抵消。

在制造业，如果缺乏有效的反馈机制，往往会酿成重大质量和安全问题。有这样一个典型的案例：通用汽车的弗里蒙特制造厂既没有有效的流程来检测装配过程中的问题，也没有明确的步骤来解决问题。结果导致了各种问题，比如发动机倒置，汽车缺少方向盘或轮胎，甚至是由于根本无法启动，不得不把汽车拖出装配流水线。

相比之下，在高绩效的制造业运营中，整个价值流里存在着快速、频繁和高质量的信息流——每个工序的操作都会被质量和监控，任何缺陷或严重偏差都能快速发现和处理。这些是保证质量、安全和持续学习与改进的基础。

在技术价值流中，由于缺少快速反馈机制，我们经常会得到糟糕的工作结果。例如，在瀑布型软件项目中，代码的开发可能花上一整年，在开始测试之前（甚至在向客户发布软件前），我们得不到任何质量反馈。在反馈稀少且滞后的情况下，工作结果是很难达到预期的。

相反，我们的工作应该是在技术价值流的每个阶段（包括产品管理、开发、QA、信息安全和运维），在所有工作执行的过程中，建立快速的反馈和前置回路。这包括创建自动化的构建、集成和测试过程，以便尽早检测出那些可能导致缺陷的代码变更。

我们还要建立全方位的监控系统，监控服务组件在生产环境中的运行状态，以便快速探测到服务的意外情况。监控系统还能帮助我们度量是否偏离了预期目标，并将监控结果辐射到整个价值流中，这样能看到我们的行为如何影响系统里的其他部分。

反馈回路不但能让问题的快速探测和修复为可能，而且还能告诉我们如何防止问题复发。这样做不但提高了工作系统的质量和安全，还创造了组织性知识。

Pivotal 软件公司工程副总裁、《探索吧！深入理解探索式软件测试》一书的作者 Elisabeth Hendrickson 说：“在我认责质量验证的时候，我将自己的工作描述为 ‘建立反馈回路’。反馈直观重要，因为它是我们工作的 向导。必须不断地验证目标，验证实施是否满足了客户的需求，而测试仅仅是一种反馈。”

4.3.3 群策群力, 战胜问题获取新知

显然，仅仅检测出意外的发生是远远不够的。一旦问题出现了，还必须群策群力，发动所有相关的人员解决问题。

Spear 博士认为，群策群力的目的是遏制住问题，放置蔓延，确定定位和处理问题，避免复发。他说：“这样做可以让所有参与者都得到更深入的知识，理解如何管理系统，把无法规避的、早起的无知阶段变成学习的过程”。

这个原则的典范是丰田的 “安全绳”。在丰田制造工厂里，每个工作中心都是一条安全绳索，每个工人和经理都受过培训，他们会在出现问题时拉下安全绳，比如，当零件有缺陷时，当需要的零件用光时，或者是加工时间比文档中描述的长时。

在安灯绳被拉动时，团队领导就能第一时间得知并着手解决问题。如果问题不能再指定的时间（如 55 秒）内解决，就会停掉整个生产线，调动整个企业一起协作，直到成功地找出解决问题的对策。

我们不能绕开问题，也不应该用 “有更多时间再解决” 来搪塞，而要立刻群策群力修复问题——这与通用汽车弗里蒙特工厂的做法几乎完全相反。群策群力的原因如下：

防止把问题带入下游的处理环节，否则不但修复的成本和工作量会呈指数级增加，而且还会欠下技术债；
防止工作重心启动新的工作，否则不但修复的系统中引入新的错误；
如果问题还没有得到解决，那么工作重心在下一次操作（如 55 秒后）中，可能还会遇到相同的问题，需要更高的修复成本。
这种全民总动员的做法似乎违背了常规管理方法，因为局部问题扰乱了整体的运营。然而，全民总动员让学习成为了可能。它还能防止由于记忆模糊的情况变化导致了关键信息遗失，这在复杂系统中显得尤为重要。在复杂的系统里，由于人员、流程、产品、地点中存在着很多意想不到的、特殊的相互作用，会出现很多问题。随着时间推移，谁都不可能精确地重视问题发生时的场景。

正如 Spear 博士所说，全民总动员是 “实时的问题识别、定位和处理（在制造业称为对策或纠正措施）循环的一部分。这就是休哈特提出的循环（即 PDCA 环）——计划（Plan）、实施（Do）、检查（Check）、改进（Act），后来由爱德华兹.戴明推广并得到了迅猛发展”。

只有尽可能在早起阶段，通过全民总动员的方式来解决小问题，才能把灾难性能事故消灭在萌芽状态。换句话说，当核反应堆的熔芯熔毁了，那就太迟了，已经回天乏术。

为了在技术价值流中实施快速反馈，我们必须建立等同与安全绳和全民响应的机制。这要求我们也创造出这样一种文化，让人们在发生问题时就去拉动安全绳，无论是在生产事故发生时还是在价值流的早期出现错误时，并且这个行为是安全的甚至是受鼓励的。例如，当有人提交了一个代码变更，而这导致了持续构建或测试过程失败的时候。

触发了安全绳时，我们就聚集在一起解决问题，停止开展任何新工作，直到问题解决。这给价值流中的每个人提供了快速反馈（特别是哪个导致系统故障的人），让我们能够快速地隔离和定位问题，避免出现复杂的情况，导致问题的因果关系变得模糊。

组织开展新工作有助于实现持续集成和部署，这就是技术价值流中的单件流。能通过持续构建和集成测试的所有变更都可以部署到生产环境中，任何导致测试失败的变更都会触发安全绳，并且会将大家聚集起来解决问题。

4.3.4 在源头保障质量

基于对意外和事故的处理模式，我们可能会在无意中把某种不安全的工作系统固化下来。在复杂的系统中，通过加入更多的检查步骤和审批流程，实际上还增加了故障发生的可能性。做决策的地方一般远离执行工作的地方，这导致审批流程的有效性有所下降。这样做不仅降低了决策质量，而且还增加了决策周期，进而减弱了因果关系之间反馈的强度，降低了在成功和失败中学习的能力。

在一些相对较小的简单系统中也存在这种情况。通常因为清晰度和及时性不足，自上而下的官僚主义和控制系统变得无效，导致了 “应该做事的人” 和 “实际做事的人” 之间的存在巨大差异。

质量控制无效的例子如下：

- 需要其它团队帮忙完成一系列乏味、易出错和手动执行的任务，这些任务本来应该由需求方自己采用自动化方式完成。
- 需要那些远离实际工作场所且公务繁忙的人批准，迫使他们在不了解工作情况和潜在影响的情况下做出决策，或者仅仅是例行公事式地盖章批准。
- 编写大量含有可疑细节，且在写后不久就过时了的文档。
- 将大量工作推给运维团队和专家委员会审批和处理，然后等待回复。

相反，在日常工作中，我们需要价值流中的每个人在他们的控制领域里发现并解决问题。通过这种方式，可以把质量控制、安全责任和决策都置于开展工作的场景里，而不是依赖于外围高层领导者的审批。

根据同行评审来评定所提出的变更，确保这些变更会按照设计运行。尽可能多用自动化方式执行通常由 QA 和信息安全人员来进行的质量检查。按需执行自动化测试，而无需开发人员向测试团队请求或发起测试工作。这样，开发人员能够快速地测试自己的代码，甚至把代码的变更都部署到生产环境中。

我们用这种方式真正地让所有人都负担起了质量责任，而不是仅让一个部门来负责。信息安全并不是信息安全部门专属的工作，正如可用性不仅仅是运维部门的专属工作一样。

让开发人员也对系统质量负责，不但能提高系统的质量，而且还能加速学习。这对于开发人员来说尤为重要，因为通常他们是距离客户最远的团队。正如 Gary Gruver 所说："当有人因为 6 个月前开发人员所造成的事故而对着他们咆哮时，开发人员其实学不到任何东西——这就是我们必须尽可能快地（几分钟之后，而不是几个月后）向所有人提供反馈的原因。"

4.3.5 为下游工作重心而优化

20 世纪 80 年代，可制造型设计（Designing for Manufacturability）原则旨在设计零件和工艺过程，让成品能够以最低的成本、最高的质量和最快的流程生产出来。例如，设计出非对称的部件以防止装反，设计出螺丝紧固件以免部件被拧的太紧。

这偏离了通常做设计的方式，即过度重视外部客户，而忽略了内部利益干系人，如生产线工人。

精益定义了我们必须为两类客户而设计：**外部客户**（最有可能为我们提供的服务付费的人）和**内部客户**（紧随我们立刻接收和处理工作的人）。根据精益原则，我们最重要的客户使我们的下游。为他们而优化我们的工作，需要我们对他们的问题给予同情心，从而更好地识别出会阻碍快速和平滑游动的设计问题。

在技术价值流中，我们通过为运维而设计来为下游工作重心做优化，包括运维的非功能性需求（如架构、性能、稳定性、可测试性、可配置性和安全性）与用户功能同样重要。

这样，我们就在源头保障了质量，并形成了一套非功能性需求，可以主动地将它们集成到构建的所有服务中。

4.4 第三部: 持续学习与实验原则

第一步建立了从左到右的工作流，第二步建立了从右到左的快速、持续的反馈，第三步要建立持续学习与实验的文化。通过应用三步工作法能持续提升个人技能，进而转化为团队和组织的财富。

在制造业的生产流程中，由于存在系统性的质量和安全问题，所以往往需要严格定义并完成工作内容。例如，在上一章描述的通用汽车的费里蒙特制造厂里，工人几乎无法在日常工作中做出改进或融入所学，即使提出改进建议也很少被接受。

在这样的环境里，通常会弥漫着恐惧和不信任感。工人犯了错就会受到惩罚，那些提出建议或指出问题的人则会被认为是告密或者管闲事的人。当发生了上述情况时，领导层会故意压制，甚至进行惩罚，这导致了质量和安全问题的进一步恶化。

相反，在那些高绩效的组织中，则要求积极地促进学习。工作不是严格意定义的，相反，工作系统时动态的，生产线工人在日常工作中通过实验来做出新的改建。工作流程是严格标准化的，工作结果都会写入文档。

技术价值流的核心是建立高度信任的文化。它强调每个人都是持续学习者，必须在日常工作中承担风险；通过科学的方式改进流程和开发产品，从成功和失败中积累经验教训，从而识别有价值的想法，摒弃无用的想法。另外，所有局部的经验都快速转化为全局的改进，从而帮助整个组织尝试和实践新技术。

为日常工作的改进预留时间，从而进一步促进和保障学习。通过不断向系统加压的方式，来强化持续改进。在可控的情况下，我们甚至通过在生产环境里模拟或者注入故障来增强弹性（resilience）。

通过建立持续、动态的学习机制，帮助团队快速并自动地适应不断变化的环境，进而帮助企业在市场经济竞争中脱颖而出。

4.4.1 建立学习型组织和安全文化

在复杂系统中工作时，精确地预测出结果是不现实的。因此，在日常工作中，即便未雨绸缪、小心谨慎，意外依然会发生，甚至有时还会发生灾难性的事故。

当某些意外影响到客户时，我们努力追本溯源，但根本原因通常会被认定为认为错误，而管理层的做法往往是点名、责备，甚至羞辱责任人。而且，管理者会暗示，犯错的人应当受到惩罚。他们因此会指定出更多防范错误复发的流程和审批环节。

Sideny Dekker 博士曾定义了安全文化的一些关键要素，并创造了 “正义文化” 这一术语。他写到：“不公正的事故和意外处理会阻碍安全调查，让安全工作者感到恐惧（而不是专注），让整个组织更加官僚（而非更加细致），甚至还会导致信息封闭、责任逃避和滋生自我保全意识。”

这些问题在技术价值流中显得特别突出——技术类工作几乎都是在复杂系统中进行的，管理层对事故责任人进行惩罚不但会引起恐惧感，还会导致问题和故障的隐瞒不报，直到下一个灾难性事故的发生。

Ron Westrum 博士是研究企业文化中安全和绩效重要性的鼻祖之一。他曾指出，在医疗机构中，患者的生命安危高度依赖于医疗机构的 “生机” 文化。他定义了三种类型的文化，如下表：

- 病态型：病态性组织的特点是组织中存在大量恐惧和威胁。由于政治原因，个体为了保全自身利益，通常会隐瞒真相或者歪曲事实。在这种组织中，故障和事故经常被隐瞒。
- 官僚型：官僚型组织的特点是规则合流程僵化，所有部门通常都 “自扫门前雪”。在这种组织中，通过评判系统处理事故，结果往往恩威兼施。
- 生机型：生机型组织的特点是积极探索和分享信息，让组织更好地履行使命。在这种组织中，整个价值流中所有的员工共同承担责任，对事故进行积极反思，并进行真正的根因调查。

Ron Westrum 的组织类型学模型：组织如何处理信息

病态型	官僚型	生机型
隐瞒信息	忽略信息	积极探索信息
消灭信使	不重视信使	训练信使
逃避责任	各自担责	共担责任
阻碍团队的互动	容忍团队的互动	鼓励团队间结盟
隐瞒事故	组织是公道和宽容的	调查事故原因
压制新想法	认为新想法会造成麻烦	接纳新想法

与 Westrum 博士在医疗机构中的发现类似，在技术价值流中，高度信任的生机型文化对组织的绩效也有异曲同工的作用。

在技术价值流中，通过努力打造安全的工作系统，我们能建立起生机文化的基础。在意外和故障发生时，关注如何重新设计系统，从而防止事故复发，而不是去追究人的问题。

例如，可以在每次事故发生后进行不指责的回顾，对事故发生的原因和过程做出客观解释，并就优化系统的最佳措施达成一致。在理想情况下，这不但能防止问题复发，还有助于实现更快的故障定位和恢复。

我们能够通过以上的方式建立学习型组织。Etsy 的工程师 Bethany Macri 发起并开发了事故回顾记录工具 Morgue，他说：“如果不指责，员工就没有了恐惧，没有了恐惧，就能够做到坦诚，而坦诚能够有效预防事故。”

Spear 博士认为，消除指责能够有效实现学习型组织，使 “组织自我诊断和自我优化，并能熟练地定位和解决问题”。

Senge 博士描述了学习型组织的诸多特质。他在《第五项修炼：学习型组织的艺术与实践》一书中写道，这些特质有助于实现客户价值，保证服务质量，揭示事故真相，并打造具有竞争优势、充满活力和高度忠诚的组织。

4.4.2 将日常工作的改进制度化

如果团队没有能力或者意愿去改进现有的流程，那么就会持续饱受眼前问题的困扰和折磨，而且痛苦指数还会与日俱增。Mike Rother 在《丰田套路》一书中指出，就算不去优化现状，流程也不会是一成不变的——由于混乱和无序，流程会随着时间的推移持续恶化。

在技术价值流中，为了防止灾难性事故的发生，团队陷于实施各种临时解决方案的工作中，反而没有时间去完成那些有价值的工作。因此，用临时方案解决问题的模式往往还会导致问题和技术债务的积累。所以《精益企业》的作者 Mike Orzen 说："比日常工作更重要的，是对日常工作的持续改进。"

通过明确预留时间来改善日常工作，包括预留时间来催还技术债、修复缺陷、重构和优化代码环境。可以在每个开发周期的间歇中预留一段时间，或者安排改善闪电战（kaizen blitze）时段，让工程师通过自组团队的方式来解决他们感兴趣的问题。

通过采取以上措施，在日常工作中，所有人都始终能在可控的范围内发现和解决问题。在解决了困扰团队数月甚至几年的重大问题后，接下来就可以消除系统中其他的潜在问题。及早定位和识别在那些潜在的问题，不但能降低解决问题的成本，而且系统承担的风险也会更小。

1987 年，镁铝公司（一家铅制造商，当时有 90000 名雇员，年收入 78 亿美元）改进了工作场所的安全性。生产铝是需要高温、高压和强腐蚀性化学品的。美铝 1987 年的事故居高不下，每年有近 2% 的员工受伤，平均每天 7 人左右。当 PaulO'Neil 担任首席执行官时，他设定的第一个目标是让员工、协作方和方可的伤害数降为零。

Paul O'Neil 要求所有受伤事故必须在 24 小时内通报——这不是为了惩罚，而是为了持续改进，创造出更安全的工作场所。通过这项举措，美铝在 10 年里将伤害率降低了 95%。

随着伤害率的降低，美铝就可以把精力放在那些较小的问题和潜在的风险上——他们开始识别潜在的风险，而不仅是在事故发生后通知 Paul O'Neil。通过这种模式，美铝改善了工作场所的安全性，在接下来的 20 年里，他们的安全记录在业内一直保持着领先。

Spear 博士写到："美铝通过识别自身的处境、困难和障碍，逐渐用动态的持续改进模式替代了传统的应急和救火队模式。在识别了风险、定位并处理了问题之后，他们就去反思其他容易被忽视的风险，并持续改进。这帮助公司在市场上获得了更大的竞争优势。"

类似地，对于技术价值流而言，让工作系统更加安全也一样有助于发现和解决潜在风险。例如，一开始我们可能只是对影响客户服务的事故做不指责的事后调查，随着时间的推移，我们将逐渐地识别其他潜在的风险。

4.3 把局部发现转化为全局优化
一旦在局部范围内取得了成果，就应当把它分享给组织里的其他人，让更多的人从中获益。换句话说，当单个团队或个人获得了独有的专业知识或经验时，我们的目标是把这些隐形知识（即很难通过文档或沟通的方式传达的知识）转换为显性知识，从而帮助他人吸取这些专业支持并在实践中应用。

这样，当其他人也要完成类似的工作时，就可以参照以往集体的经验和智慧。把局部知识转化为全局知识的一个著名案例是 "美国海军核动力推进计划"（又称海军反应堆，Naval Reators，NR），它稳定运行的时间超过了 5700 堆年，反应堆至今尚未发生过一次伤亡或核泄漏事故。

NR 以恪守标准化流程而闻名于世，出现任何流程或操作偏差都要写故障报告，以便积累经验。不管故障信号强弱，或者风险大小如何，都会基于这些经验持续更新流程和系统设计。

这样带来的结果是：在任何一组新海员出海时，他们都能迅速地从集体长期积累的智慧中获得成长。同样令人印象深刻的是，他们在海上的经验也会持续添加到这个智慧库中，以帮助以后的海员安全地执行任务。

在技术价值流中，我们也应该通过类似的机制建立全局知识库。例如，把所有事故报告转化为可搜索的知识库，让有需要的团队能更加方便地使用它去解决类似问题，同时建立起组织级的共享源代码库，让所有人可以方便地使用整个组织的代码、库和配置。这些机制有助于把个人的专业知识转化为服务更多成员的集体智慧。

4.4.4 在日常工作中注入弹性模式

低绩效组织想方设法缓解问题，换句话说，他们疲于应付问题。例如，为了降低任务空闲的风险（货物未到场，或者当前的库存配件都是报废品），管理者可能会在每个工作重心里存放更大量的在制品。然而，这个缓冲也增加了在制品数量，前文所提到的各种不良后果也会接踵而来。

类似的，为了降低由于机械故障所导致的生产中断的风险，管理者可能会通过购买更多的设备、雇佣更多的员工或者扩大厂房的方式来增加产能。可能所有这些做法同时也增加了成本。

相对而言，高绩效组织通过改善日常运营，持续地引入张力提高生产效率，同时在系统中注入更大的弹性，来实现或达到更佳的效果。

另外一个经典案例是丰田的一个顶级供应商——爱信精机公司的一家工厂。假设他们有两条生产线，每条生产线每天能产出 100 个单位的产品。在订单不紧急的时候，他们把所有的生产任务都发送到其中一条生产线上，并尝试不同的方式来增加产量和识别生产流程中的瓶颈；如果这条生产线由于过载而发生了故障，他们就会把生产任务发往另外一条生产线。

通过在日常工作中持续不断地试验，他们能够持续地提高产能，并且不需要增加任何新设备或人员。引入这种类型的改进不仅提高了生产效率，而且还挺高了弹性，因为组织总是处在紧张和变化的状态中。知名作者和风险分析师 Nassim Nicholas Taleb 将这种通过加压来增强弹性的做法称为抗脆弱性（antifragility）。

在技术价值流中，通过缩短部署的前置时间、提高测试覆盖率、缩短测试执行时间，甚至在必要的解耦架构，都属于在系统中引入类似张力的做法，也都能够提高开发人员的生产效率及可靠性。

另外，还可以通过演习的方式来预演大规模故障，比如关闭某个数据中心。或者在生产环境中注入大规模故障（如 Netfix 著名的 “捣乱猴”，它会随机杀死生产环境中的进程和服务器），来验证系统的可靠性是否达到了预期。

4.4.5 领导层强化学习文化

按照传统的管理模式，领导者负责制定目标，分配资源，建立正确的激励机制，同时还要为组织简历情感基调。换句话说，领导者通过 “做出所有正确的决定” 来领导团队。

然而，有证据表明，领导力的优秀并非体现在做出的所有决定都是对的。相反，更卓越的领导力其实是为团队创造条件，让团队能在日常工作中感受到这种卓越。换句话说，这需要领导者和员工们共同的努力，每个人都相互依存，缺一不可。

《走动管理》的作者 Jim Womack 描述了领导者和一线工作者之间的互补的工作关系，必须互相尊重。根据 Womack 的说法，这种关系时必要的，因为谁都无法独立解决问题——领导者不会亲自从事解决问题所需的一线工作，而一线工作者也不了解大的组织环境，或不具备在工作领域以外做出改变的权利。

领导者必须强调解决问题的能力和学习的价值。Mike Rother 在他所谓的教练套路（coaching kata）中规范化了这些方法。结果体现了一种科学的方法——明确地描述我们的《真北》目标，就像美铝案例中的 “零事故” 或爱信案例中的 “一年吞吐量翻番”。

在这些战略目标的指导下，我们建立相互嵌套的迭代的短期目标，然后在价值流或工作重心级别设立目标条件（如在接下来的两周里缩短 10% 的前置时间），以实现这些目标。