软件质量报道 QualityReport

MeterSphere

白皮书 v1.0

持续测试

The White Paper of Continuous Testing

“软件质量报道”公众号及 MeterSphere 开源项目联合出品

2021 年 1 月

持续测试

让测试不再成为持续交付的瓶颈

目录 CONTENTS

1. 引言

1.1 编写本白皮书的目的

软件测试作为软件质量保证的关键手段，自从软件工程诞生之日起，就一直伴随着软件研发模式的迭代升级而持续自我更新。软件工程诞生五十多年来，历经瀑布模型、V 模型、敏捷模型等一系列的变革，慢慢形成业界认可的软件工程理论，并由此产生了代码重构、测试驱动开发、持续集成、持续交付等一系列优秀实践。这其中，持续交付是企业最为关注的实践，也是敏捷和 DevOps 的核心诉求。它能够帮助企业更好地满足日益迅速变化的业务需求。持续交付的实践过程倒逼软件测试，最终形成了与之适配的持续测试。

当前，随着企业数字化转型、“软件定义一切”的深入发展，越来越多的企业面临着自身业务数字化转型的挑战，而这个转型过程高度依赖企业自身软件研发能力的构建和持续提升。因此，越来越多的非传统软件公司需要进入软件研发领域，并着手构建符合当下企业数字化转型要求的软件研发能力，包括与之适配的软件测试能力。

与此同时，关于持续测试（Continuous Testing）的最佳实践和方法还主要停留在大、中型互联网企业的内部，并未形成整个行业的共识。为此，我们希望通过本白皮书系统、深入地介绍“持续测试”理念的内涵，以及如何在企业和团队内实施，并给出相关的流程、方法和工具，真正推动持续交付在更广泛范围内的落地，持续提高大部分企业的数字业务竞争力。

1.2 如何组织、发布和维护本白皮书

本白皮书由“软件质量报道”微信公众号和 MeterSphere 开源项目组共同编写而成。编写团队基于贴近企业实践的视角和态度进行策划和组织，结合自身对软件测试发展历程以及持续测试理念的理解，积极听取行业内其他专家的观点，并认真借鉴优秀公司所积累的经验，在此基础上认真完成白皮书的编写任务。

全文从持续测试产生的背景和价值、企业如何实践持续测试，以及持续测试成熟度模型（Continuous Testing Maturity Model，简称为 CTMM）等三个方面展开讨论，并介绍了持续测试产生的原因、持续测试落地的关键要素，以及如何借助 CTMM 来完成对团队测试能力的评判等内容。

本白皮书采用线上渠道（网站、微信公众号等）为主的分发模式，编写团队通过线上渠道广泛收集大家的意见并定期更新白皮书内容。更新版本的白皮书仍然通过线上渠道再次对外提

供下载。欢迎业界各位同仁积极提交修正和改进意见，持续完善本白皮书，使之更加准确、全面和深入。

1.3 本白皮书给企业带来的价值

本白皮书为企业数字业务的资深测试人员、资深开发人员、高级质量管理人员、项目管理人员以及高层管理人员所准备。希望能够通过本白皮书让相关读者理解持续测试的理念，以及在企业数字业务中积极采纳持续测试的价值和必要性。当然，持续测试包含的范围非常广泛。从面向开发人员的测试实践到传统测试团队的功能测试，乃至业务系统发布后的线上测试。一本薄薄的白皮书肯定无法详尽方方面面，所以本白皮书主要关注于持续测试中的系统功能测试部分（也就是传统测试团队主要的工作范围），对于其他环节的持续测试实践则不会深入展开。期待本白皮书能作为一个引子，帮助大家推开持续测试这扇大门，并给大家在徜徉这扇大门背后的世界时提供一个相对合理的指引。

2. 持续测试产生的背景与意义

2.1 软件测试的起源与发展

如同任何一个行业内的系统和产品一样，软件系统和产品也同样有可能存在缺陷和故障。所以，有了计算机软件就需要有计算机软件测试。虽然软件测试诞生得很早，但是软件测试的发展并不是一帆风顺。特别是在早期阶段，软件测试得不到重视，测试技术发展比较慢。进入二十世纪八十年代后，软件测试得以快速发展，成为软件业的热门领域之一，也达到了较高的水准。回顾整个软件测试的发展历程，大体可以分成以下几个大的阶段：

▶ **初级阶段（1982 年之前）：**1968 年提出“软件工程”的概念，而在这之前，软件危机愈演愈烈，整个 IT 行业意识到必须系统化地思考软件质量问题，将软件测试和软件调试区别开来，初步形成软件测试这个学科方向。这个阶段的测试通常被认为是对软件进行事后检验的手段，以验证软件是否满足业务需求设计为主，基本停留在对程序进行验证，测试只是软件设计和编程之后的一个阶段。而且，那时的软件测试还普遍缺乏有效的测试方法和理论指导，导致软件产品存在比较多的质量问题。

▶ **发展阶段（1982 至 2000 年）：**在这个阶段，软件测试的内涵发生了变化，软件测试被看作是软件质量保证（SQA）的重要手段，不再单纯是一个程序验证的阶段，而是贯穿整个软件生命周期，从需求评审、设计评审、单元测试到集成测试、系统测试、验收测试等完整的过程，而且包含了发现缺陷、评价质量、预防缺陷等内容。作为软件工程中的重要角色，拥有数量众多的软件测试专业人员，而且软件测试也有了自己的国际标准，成为软件工程学科中的一个重要组成部分。软件测试的理论、方法和技术也逐渐建立起来了，例如，结构化测试方法、面向对象的测试方法。

▶ **敏捷阶段（2001 年之后）：**随着敏捷宣言的发布，软件测试作为软件研发的组成部分，自然需要适应敏捷开发模式以进行软件测试的转型。业界开始提倡测试驱动开发、验收测试驱动开发，开始关注自动化测试，加强测试工具和测试框架的开发，尽可能做到回归测试自动化测试、测试管理自动化。之后，DevOps 思想也逐渐兴起，测试不仅要左移，而且要右移，进行更多的在线测试（Test in Production，TiP）。在持续交付的大背景下，持续集成、持续测试将成为测试的主旋律。

最近几年，国内软件测试行业也掀起了以“敏捷测试”为代表的新一轮理论和实践运动，这场运动给软件测试从业人员带来更大挑战，也被越来越多需要数字化转型的企业所关注。

2.2 企业的数字化转型与数字业务

随着过去 20 年互联网和移动互联网的普及，互联网和移动互联网已经深入到人类生活的每一个角落，众多企业不得不进行数字化转型来保持与其客户、合作伙伴的交互和协作。为此，企业数字业务变得越来越重要。由于企业自身业务的复杂性和多变性，企业数字业务也必然面临着复杂性和多变性的挑战。这种挑战让越来越多企业意识到必须要建立自己的数字业务运营和发展能力，而不能仅仅依赖于外部的某些超级数字化平台和特定供应商。

企业需要将其日常业务运营的方方面面迁移到数字世界，并保持在线运营。显然，这种迁移高度依赖企业业务软件的支撑。软件在企业内的角色定位由此从后台走到前台，不再仅仅是企业生产运营中的一个工具，而是融入企业业务运营的各个角落。如果企业需要建立数字业务运营和发展能力，企业自身就必须具备软件研发和运营能力。为此，企业不得不自己组建软件研发团队，发展自己的软件研发和交付能力。

但是，如果回顾过去几十年软件行业的发展，软件研发对于很多专业软件企业来说都是一种高风险的投入，面临着非常多的不确定因素和较高的失败率。何况现在有很多非软件企业需要在零基础上建设如此复杂的企业业务软件研发和交付能力。幸运的是，通过过去 20 多年软件及互联网行业的发展，软件行业初步寻找到了降低软件生产风险和提升软件生产效率的有效方法。这种方法体现在软件研发模式和软件交付模式两个方面：

▶ 软件研发模式实现了从传统瀑布模型向敏捷模型转换，以适应越来越快速的软件研发迭代要求；

▶ 软件的交付模式从线下模式（盒装软件）转为线上模式（SaaS，软件即服务）。由于整个社会网络带宽、容量和通信能力的大幅度提升，软件在线访问和使用成为可能。

接下来我们将从软件研发模式的转型来阐述敏捷与 DevOps 的内涵，以及它对软件测试提出的要求。

2.3 敏捷与 DevOps：软件研发模式的变革

敏捷和 DevOps 模式借鉴新产品研发模式、传统制造业精益生产的理念，打破了传统“瀑布模型”软件生产模式中各个阶段及角色的边界，将软件研发中各个不同角色紧密融合到一个短周期的软件研发过程中，从而实现快速迭代、持续发布的目标。相对于传统的瀑布模型来说，其有如下几个方面的关键变化：

▶ 敏捷与 DevOps 模式更加积极地去面对软件研发过程中频繁变化的业务需求，认为需要具备支撑软件业务需求不断变化的能力。为此，其在软件研发和运维过程中引入短周期持续迭代模式，通过拆分业务需求、反复迭代的方式来响应业务的需求变化；

▶ 敏捷与 DevOps 模式特别强调“业务价值交付”这一关键要点，认为软件研发过程中所有的设计都应该服务于“软件业务价值更快、更好地交付给最终业务用户”这一宗旨。而且，这个业务价值的交付必须经过最终业务用户认可才行。所以，软件研发过程要能够更加高效地交付业务价值，并支持在最终用户那里快速验证交付的业务价值；

▶ 敏捷与 DevOps 模式始终追求激发软件研发流程中人的主动性，践行人和互动比流程和规章更加重要的原则。所以，其研发流程设计中突出为人员提供环境和支持，相信团队自身的能力来完成软件业务价值的最终交付。

通过以上对比可以发现，这种模式成功的关键在于，团队能够以“业务价值的持续交付”为核心追求。为适应这种模式，软件研发进入了迭代周期通常为 1 至 4 周的持续迭代模式。软件研发团队的各个角色需要通力合作，在如此短的迭代周期内完成从需求分析、任务规划、代码实现、质量测试乃至上线发布的全过程。显然，这也意味着软件生产流水线上的各个环节都必须要进行彻底的变革以满足如此高频的迭代节奏。而这种持续不断的业务价值交付过程也被定义为敏捷与 DevOps 最为核心的实践模型，即持续交付。

2.4 持续交付催生持续测试

如前所述，持续交付是企业对其软件研发能力的核心诉求，其持续交付的关键抓手就是实现软件业务价值持续、高效地向业务用户交付这一闭环。软件行业内大家经常使用如下的“∞”模型来描述这个闭环：

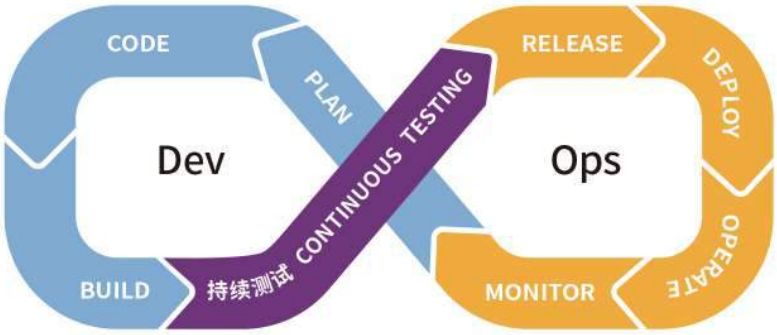

图1 DevOps 与持续测试

从图 1 可以看出，整个持续交付的流程包括计划、编码、构建、测试、发布、部署、运维和监控环节，并实现整个过程的持续迭代闭环。对于持续交付中的任何一个环节，为了满足敏捷及 DevOps 的要求，其都要遵循以下几个方面的要求：

▶执行过程需要“足够得快”，以满足整体迭代的效率要求。这要求每个阶段都需要引入尽可能多的自动化工具和能力；

▶执行效果需要“准确且有效”，以保障整体迭代的质量要求。这要求每个阶段都准确且有效达到最核心的质量诉求，而不是追求“大而全”的完整质量管理逻辑；

▶执行衔接需要“平滑而有序”，以保障整个迭代闭环能够高速运转。这要求每个阶段在设计过程中都要考虑前后衔接，并在团队组织和工具链上实现打通。

按照以上要求，软件行业先后引入了 Scrum 敏捷研发模型、持续构建、持续部署、持续监控等一系列最佳实践。而在当下，相比于其他环节，测试领域成为企业落地敏捷及 DevOps 实施的最大瓶颈。具体来说，这个瓶颈体现在两个问题：一是速度问题，一是价值问题。

速度问题主要体现在执行效率低，反馈时间长。

首先，传统软件测试的执行方式以手动测试为主。如今，软件自动化技术已经非常普及，软件生产流程中的代码检查、构建和部署等多个环节都已经大规模自动化，但软件测试的自动化比例仍然低得惊人。来自《中国 DevOps 现状调查报告 2019》的数据（如图 2）就说明了这一点：

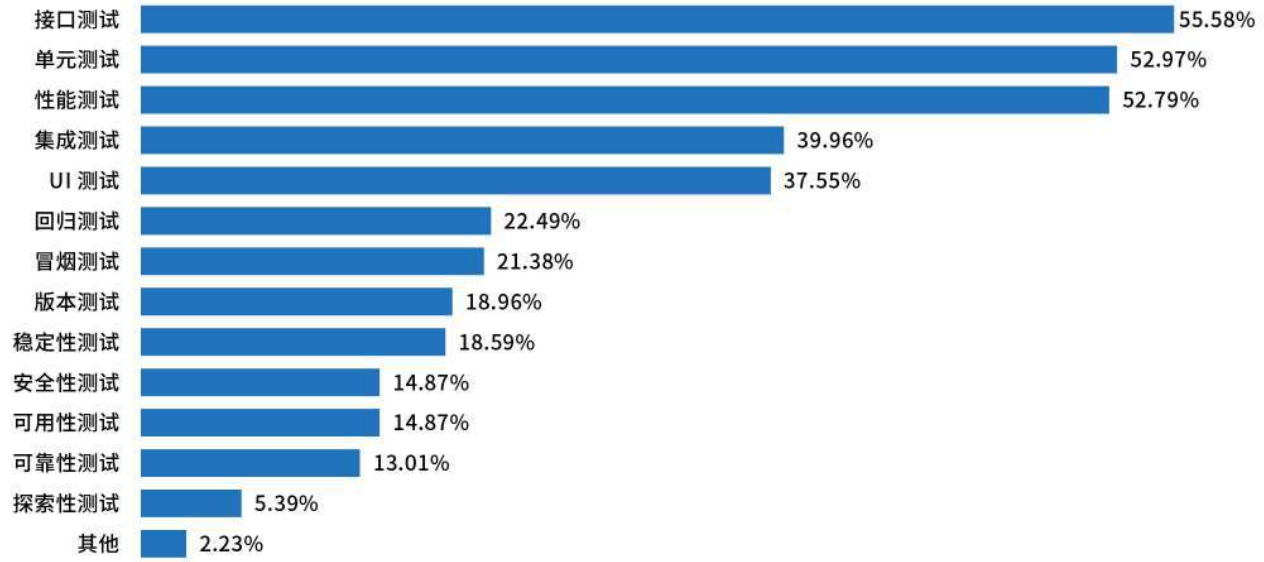

图2 国内软件测试行业各测试类型自动化占比情况

如图 2 所示，即使相对容易自动化的接口测试自动化占比也未达六成，而其他如集成测试、UI 测试、回归测试等测试类型的自动化占比就更低。没有足够自动化测试的支持，测试的迭代效率必然会成为瓶颈。尤其是当下企业对持续迭代的周期要求越来越短，这个瓶颈会变得更加突出。众所周知，当下很多时候测试被当成影响整个研发效率中的首要问题。不仅在国内，来自 GitHub 的《2018 年全球开发者报告》(如图 3）也显示有 55% 的被调查者认为，测试是导致项目延期最多的原因，位居各原因之首。

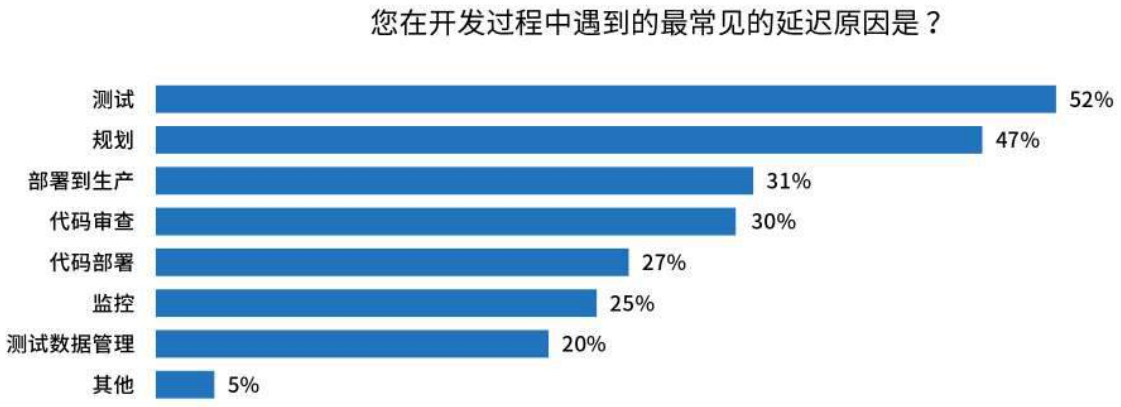

图3 导致软件延期的研发流程环节

当然，执行效率低会带来另外一个问题，就是反馈时间太长。在测试没有执行完之前，开发团队几乎得不到测试关于当前产品质量和风险的任何反馈。而测试在产品没有最终完成编码前，也无法介入其中开始工作。一旦到测试介入测试并发现产品缺陷的时候，很多时候都已经为时已晚（要么就是没有足够时间来修复，要么就是缺陷修复的代价太高）。

无论是执行效率慢，还是反馈时间晚，最终都体现为测试工作未能很好地完成，导致产品的发布延期，或者无奈之下只能带着巨大业务风险发布产品。

价值问题主要体现在缺少足够的“认可”，无论是从业务部门还是从研发部门。

在实际工作中，测试的结果往往得不到足够的“认可”，从而让整个团队对测试工作的价值产生怀疑。这种不能取得信任的原因主要有以下两个方面：

1. 测试结果的稳定性不高，频繁出现明显的“误报”现象。这种误报有可能是因为测试执行过程中的错误导致，也有可能和测试数据或者测试环境相关。频繁出现的“误报”导致大家对测试结果的怀疑，进而不信任测试质量；

2. 测试过程未能充分反馈软件中的业务风险。现代软件涉及到的功能和场景越来越复杂，但现实条件又决定了测试无法覆盖软件中的所有功能点和可能存在的应用场景。从商业角度来看，不同功能点及其应用场景的商业价值是不一样的，甚至相差巨大。传统软件测试用例设计的指导理念过于强调测试用例的逻辑完备性，而忽略了不同测试用例覆盖的业务风险差异巨大。于是，实际工作中经常出现重视逻辑和代码覆盖率，而忽略业务风险覆盖率的问题。测试工作花费了大量时间设计和维护了众多的无太大业务风险价值的测试用例，而忽略了对关键路径和重大业务风险点的覆盖。这种问题最终呈现出来的结果就是，业务团队无法从测试报告明确判断当前版本的业务风险到底有多大。

随着敏捷和 DevOps 研发模式的推广，以上这些问题变得越来越明显。测试越来越成为企业软件研发流程敏捷化的障碍。为了解决上面问题，各种测试理念和测试工具不断出现。在这其中，为适配“持续交付”这一敏捷和 DevOps 核心目标的“持续测试”理念最为关注，并且得到了越来越多企业的采纳。

2.5 什么是持续测试?

为了适应敏捷和 DevOps 软件研发新模式，“持续测试”这个理念被业界提出。目前，大家普遍认为持续测试是指软件持续交付流水线中的一种可随时开展且具有连续性的测试自动化测试流程。它基于强调全方位的测试能力，以及测试、开发和运维良好的融合自动化测试能力，但它更关注利用自动化测试能力在持续交付流水线全过程中及时、准确地给团队提供当前版本的质量和使用体验反馈，从而切实保障软件持续交付过程中的质量，如图 4 所示。形象地来说，它不仅需要有助于让软件交付流水线运行得更快，还需要能够以最低的代价覆盖交付所面临的业务风险，保障生产所开发的软件产品足够安全可靠。

图4 持续测试在敏捷与 DevOps 实践中的定位

从持续测试的定义可以看出，持续测试是基于自动化测试能力，但更是一种融入持续交付实践的测试活动运行方式。与持续交付其他阶段的实践类似，持续测试实践最关键的特点在于“持续”二字。其特征包括以下四点：

▶ **全流程平滑有序：**将传统瀑布模型下的测试活动分别向软件研发运维流水线的左侧和右侧进行彻底地移动，以让测试活动在覆盖软件交付流水线的全过程上没有停顿、没有阻塞。

▶ **准确且有效：**被测系统往往很复杂，不可能做全回归测试，而是要推行精准测试，提升测试效率。

▶ **足够快：**以快速反馈为主要导向，整个测试过程要快，一方面依赖高度自动化测试（自动化测试占比应该超过 85%），另一方面依赖业务端到端的探索式测试。

▶ **高度集成：**以融入持续交付流水线为载体，测试活动将伴随软件研发流水线的每一次流动、每一个版本而频繁发生，实现测试与持续构建、持续集成、持续部署、持续运维等全面集成。

除此之外，为了让测试反馈的结果准确且有用，持续测试还强调通过各种手段避免软件测试中的“误报”现象，保证测试反馈准确。同时，通过聚焦测试用例的业务风险覆盖率确保测试能真正反馈软件的业务风险。

综上讨论，持续测试包含的内涵非常丰富。除了传统意义上的测试活动外，还包含“测试左移”和“测试右移”的实践。其中“测试左移”强调把测试活动引入到需求、设计和编码等开发环节，让测试活动与开发活动同时进行。甚至是测试在前、开发在后，即测试驱动开发，这其中典型的实践有需求评审、设计评审、单元测试、代码扫描分析与检查等。

而“测试右移”则强调，测试活动不应该随着软件上线发布而结束。相反，对于线上环境持续的测试和监控也是重要的测试活动。其中典型的“测试右移”活动有生产环境上的流量回放、全链路压测、用户体验的 A/B 测试等。

如引言所述，本白皮书主要关注与系统功能测试环节的持续测试实践。对于持续测试的左移和右移行为会有所涉及，但不会展开论述。如果你想详细了解这方面的相关需求，建议参考其他相关资料。

2.6 持续测试对企业的价值

当前，企业数字业务拓展的关键是实现用户、业务和数据的在线化，并基于以上三者实现业务运营的持续优化和创新。为此，敏捷和DevOps成为了企业数字业务中的主流软件研发模式，而持续测试则在其中承担着企业数字业务拓展中的质量保障责任。具体来说，这种特殊质量保障的价值体现在以下几个方面：

▶ **它是保障企业业务数字化转型可以在业务风险可控情况下推进的关键。**显然，数字业务的业务风险控制是企业数字化转型中的底线。没有这个保障，企业不敢也无力进行彻底的数字化转型变革。快速迭代的数字业务更加重了企业对于这个问题的担心。持续测试实践强调风险控制内建在软件交付流水线的全过程，并强调通过合理的测试用例设计方式覆盖尽可能多的业务风险。同时，持续测试会将测试活动融入到交付流程中，通过自动化手段及时高效地发现软件业务风险。这样，快速迭代的数字业务全过程都有不断展开、有效设计的测试活动来控制其业务风险。

▶ **它是保障企业数字业务创新的基石之一。**业务创新意味着需要尝试全新的东西。不管是全新的领域和全新的模式，都意味着带来更多未知的风险。业务风险的控制方式多种多样，但对于数字业务来说，其业务软件的风险保障是最关键的一环。持续测试的理念是把测试的活动尽早发生，及早发现质量和业务风险，纠正产品设计和规划中的问题，帮助业务创新渡过最容易夭折的孵化阶段；同时，当业务创新进入高速成长阶段，持续测试整个机制能有效降低快速成长项目的质量风险，避免高速成长项目因为质量风险而遭遇突然死亡。

▶ **它是帮助研发团队进行测试转型升级的重要抓手。**企业要想大力拓展数字业务，其关键在于整个数字业务团队自身的转型升级。在大部分企业内，其研发团队中的质量保障由于历史遗留业务或者团队认知的原因，仍然在按传统方式运营。如果需要改变这种现状，企业需要一种明确的方法论和指导实践帮助团队形成新认知、达到新共识、执行新变革。

具体来说，这种抓手体现在三个方面：其一，持续测试强调全流水线都需要展开测试活动，这对于提升研发和运维团队的测试意识有很大帮助；其二，持续测试强调自动化测试能力的提升，这对传统测试团队实现自动化测试转型有巨大帮助；其三：持续测试强调基于交付流水线的协同，这让不同环节测试活动的衔接变得非常关键，从而能够有效促进研发团队内部不同团队在测试活动上的协同和价值认同。

3. 如何实施持续测试

3.1 落地持续测试的整体框架

作为软件测试领域当前最为推崇的理念之一，持续测试在越来越多的企业内部推广和落地。尽管每家企业落地实践都有其自身特点，但是总结起来仍然可以归纳为如图 5 的整体实践框架。

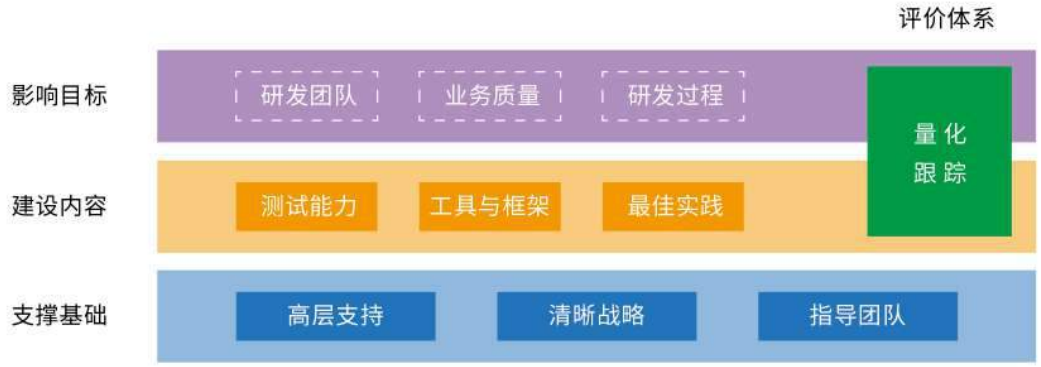

图5 持续测试落地整体框架

首先，整个持续测试实践的支持基础在于，取得公司高层支持并能制定清晰的转型战略、建立指导团队。其次，在完成支撑基础工作后的建设内容包括，研发团队测试能力建设、工具与框架建设以及最佳实践落地。最后，通过这些工作形成对于研发团队、业务质量和软件研发过程的改变和影响，将持续测试实践融入到公司研发人员、软件研发流程和每个业务系统的质量保障之中。同时，整个过程需要通过合理的量化跟踪体系来帮助我们量化持续测试建设的进展和已经产生的相关影响进度。接下来我们将逐一讨论这个框架中各个要素的内涵。

3.1.1 取得高层支持

与任何一项重要的企业实践一样，持续测试落地的第一个关键要素也是需要取得企业高层的支持。这一点之所以如此重要，是因为如下几个原因。

1. 首先，团队要意识到持续测试实践的落地并不是一个简单事情。它的起因可以是公司内一个小团队的尝试，但是要想在整个公司层面上进行推广和落地，高层支持是帮助内部团队解决实践落地过程中各种问题的有力保障，尤其是在面对跨部门的沟通和协调时候，高层支持是这些工作得以推动的关键所在。

2. 其次，高层支持对于持续落地的意义还在于，高层更容易将具体的软件工程实践价值和企业业务发展价值建立正确的关联，能让公司从业务拓展的角度来意识到持续测试实践落地的意义，更容易形成公司内部广泛的共识，可以大幅度地减小在实际落地过程中的摩擦和阻力。

所以，作为该项实践在企业内落地的推动者，其需要做的第一件事情就是——寻求企业内高层的支持。不过，在寻求高层支持时候，推动者一定要避免把持续测试描述成为一个“银弹”，让企业高层和非技术人员对此产生误解和不切实际的预期。推动者需要明确告知高层这个实践在推动过程中的困难、挑战和风险，需要得到的具体支持，以及这个实践给企业带来的具体收益（例如成本收益、质量收益、效率收益等），并最好能够提供相对可量化的指标。一旦有可以持续跟踪的量化指标，推动者则更容易得到高层的有力支持。

3.1.2 制定清晰的战略

在得到高层支持后，制定清晰的战略就变成另外一个非常重要的因素。一般来说，这个战略包括业务层面、技术层面和团队组织架构层面。

▶ 从业务层面看，战略一定要紧紧围绕企业整体的业务目标展开，优先选择能够带来明确业务价值的项目和团队展开，尤其要避免落入纯粹技术层面上的“先易后难”逻辑中。这样容易导致实践推动过程中业务价值无感的尴尬境况；

▶ 从技术层面上看，转型战略一定要抓住持续测试中的核心技术手段（即自动化测试）的能力建设。无论是测试工具还是业务可测试性上，都需要向自动化测试手段倾斜。只有在实践过程中不断加强测试环节的自动化占比，才有可能不断释放持续测试实践的价值红利，得到公司高层和业务团队的持续支持；

▶ 从团队组织架构层面看，战略需要考虑两个问题：一是研发团队内测试活动的组织安排和调整；二是持续测试实践如何影响软件研发流程中的跨团队协作。

作为持续测试实践的推动者，制定好清晰的战略是其推动具体实践落实的关键指引，也有利于整个公司在具体实践过程中保持一致的理解。

3.1.3 成立指导团队

成立某种形式的指导团队是一个普遍的实践，也被认为是企业持续测试落地的关键要素。这个团队可能是在企业组织架构上真实存在的团队，但更可能是一种“委员会”性质的虚拟团队。总结来说，这个团队主要的职责包括如下几个方面：

▶ 帮助企业制定清晰的战略落地路径和执行计划；

▶ 提供必要的组织协调、资源支持和技术支持能力；

▶ 持续跟进实践落地的进度，及时反馈出现的问题并推动解决。

之所以需要建立这样的团队，是因为持续测试实践并不仅仅是一个技术方案或者工具平台的采纳，而是一种新的理念和运作模式的改变。这个过程较难很好地嵌入到已经存在的日常软件团队工作中，或者说简单嵌入会因为当前运作模式的惯性导致落地实践变形。

通常来说，这个指导团队是由以下几种类型的人员所构成的：

▶ 来自公司或者软件研发团队的高层代表。其主要职责是为团队持续取得高层的支持，并确保整个实践落地过程中不背离企业最核心的业务价值；

▶ 来自持续测试领域的专家。其主要职责是为实践落地过程中提供专业的技术培训和沟通协调支持，确保持续测试中的关键理念在落地过程中不走样。这类专家可以是来自企业内部，也可以是来自外部的领域内专家；

▶ 来自早期实践团队的领导。早期实践团队一般是持续测试实践落地的拥护者，而且也是企业实践落地的直接操作者。这类人员可以及时反馈实践落地过程中遇到的企业特有问题，并能结合企业实际情况给出建议。而且，在实践推广环节，这类成员会是最好的“现身说法”案例，可以大大提升整个实践的推广速度。

3.1.4 提升团队测试能力

企业内任何实践的落地，都依赖于企业内部人员观念的转变和能力的提升，持续测试也不例外。在这一点上，研发团队人员能力的提升尤为重要。由于之前长期的错误观点引导，导致国内研发团队在测试能力的投资明显落后于开发和运维能力，大量测试外包的采纳又让广泛的测试能力提升变得不太现实。但是，在持续测试实践过程中，内建质量变成最核心的质量管理手段，这要求对于整个软件研发团队整体全面地提升测试能力。为了达成这一目标，企业研发团队需要从培训和实践两个方面入手。

能力培训

如同任何一项组织转型，能力培训都是不可或缺的环节。由于持续测试强调内建质量的能力，这要求整个持续交付流水线上的不同角色都参与其中，所以持续测试的培训工作不仅仅指测试团队的岗位，还包括开发团队、配置管理团队等岗位。当然，培训工作需要针对不同岗位有不同的侧重点。一般来说，常见岗位对于持续测试能力培训的侧重点如表 1 所示。

表1 不同岗位持续测试能力培训的侧重点

岗位名称	培训重点
开发人员	重点关注在开发环节有效提升软件内建质量的实践，包括单元测试、组件集成测试，以及相应的工具、自动化运行 / 回归测试方法等；
测试人员	重点关注测试用例设计与管理、接口与 UI 测试自动化、性能测试设计与自动化，以及测试数据管理、服务虚拟化相关工具平台使用等；
运维人员	重点关注线上压力测试、流量数据脱敏和存储、线上问题反馈，以及相关工具和自动化运行 / 回归方法等；
配置管理人员	重点关注持续交付流水线如何支持持续测试实践，包括如何集成单元测试、静态代码检测、如何在日常构建和部署中触发自动化测试等。

实践考核

任何一项能力的提升仅仅靠培训和理论学习是远远不够，更需要在日常工作中反复实践和积累经验。这就需要研发团队各个不同角色都在测试能力上投入时间和精力，也意味着在考核体系上需要额外认可这部分的投入回报。为此，需要鼓励将持续测试实践融入到研发团队的日常工作考核过程中。这种考核需要包括软件内建质量提升的最终效果考核，也需要包括其中关键提升过程的考核。

▶ 正常情况下，有效持续测试实践的落地必然会带来比较明显的线上缺陷率的下降。因此，这个指标可以作为考核整体研发团队持续测试实践落地一个效果指标；

▶ 针对软件研发过程的不同阶段制定相应的过程性考核指标，包括如单元测试覆盖度、测试用例业务风险覆盖度、测试自动化占比等考核指标；

▶ 对于跨团队的持续测试整体运行效率设立过程考核指标，包括回归测试频率、执行时长、执行成功率等考核指标。

以上这些考核指标多以项目团队为维度进行考核。通过团队实践考核来肯定团队在持续测试方面的投入。当然，这些考核指标也会反向推动团队更积极地投入资源和精力到持续测试实践的落地过程之中。

3.1.5 选择合适的工具和框架

在解决团队和人员能力的同时，选择一个合适的支撑平台和框架同样也是落地持续测试转型实践的关键因素。当前，软件测试工具及框架非常多，无论是商业的还是开源的都可以供大家选择。由于持续测试强调测试质量管理是一个贯穿整个软件生产过程的活动，企业很多时候都需要选择多个工具和框架来完成这个目标。如果按照工具使用的目标阶段进行划分，可以把持续测试的工具按照表 2 进行划分。

表2 不同测试工具和框架的分类

目标阶段	工具与框架定位	典型工具与框架
编码阶段	帮助团队完成包括单元测试、集成测试和代码检查等。	JUnit、PyUnit、CppUnit、SonarQube
测试阶段	帮助团队完成功能测试、接口测试、界面测试、兼容性测试、验收测试，以及性能测试等。	RobotFramework、Postman、Selenium、Appium、MeterSphere
运营阶段	帮助团队完成线上系统的功能、性能、容量问题的测试和验证等。	JMeter、LoadRunner、阿里云 PTS、MeterSphere
测试管理	帮助团队对不同阶段的测试进行统一管理和协调。	JIRA、TestLink、MeterSphere

尽管持续测试涉及的工具和框架非常多，但是企业进行工具选型还是要回到持续测试理念的关键诉求上来，重点需要考量如下几个方面：

▶ **工具和框架是否可以提供良好的自动化测试能力？**自动化测试是持续测试的基础，离开这个能力谈持续测试的工具都面临着无法落地的窘境。而且，由于持续测试强调不同阶段都需要内嵌测试质量保障，所以不同阶段的工具和框架需要按照测试活动的类型融入不同的自动化测试能力；

▶ **工具和框架是否可以很好地和团队持续交付工具链进行融合？**持续测试是持续交付体系中的质量保障理念，其需要融合到持续交付工具链里面工作才能够达到最终的目标；

▶ **工具和框架需要提供合适的使用门槛。**过高的使用门槛，例如需要专业编程技能、复杂操作界面等，经常成为工具从企业内精英团队向其他团队推广的事实障碍。

考虑到开发团队日常工作的主要环境是集成开发环境（IDE），面向开发团队的持续集成工具和框架普遍都需要融入到 IDE 中。但是，市面上其他类型的持续集成工具相对分散，缺乏类似于 IDE 这样的集成整合平台。为此，这些工具及框架当下一个明显的发展方向就是“平台化”。一般来说，这种类型的持续测试平台需要具备以下几个方面的特征：

▶ **整合当前最普遍使用的测试能力工具。**相对于各种工具提供的单一能力，持续测试期待能够整合多种测试需要的能力，并且能实现不同能力之间的合理衔接和转换。通常，这种平台需要整合的工具和能力包括用例管理能力、接口测试能力、性能测试能力、UI 测试能力等；

▶ **实现测试活动的全生命周期线上化管理。**除了整合各种测试能力工具外，同样需要提供完整的测试流程跟踪能力，并且实现测试设计、测试计划、测试执行和测试报告等不同阶段工作的关联和整合，从而实现测试活动的全面线上化管理，增强人员之间的沟通协调效率；

▶ **支持不同团队的协同。**如前所述，持续测试强调测试活动需要在不同环节持续发生，这就需要平台能够支持包括开发团队、测试团队、测试管理团队等不同角色的协同；

▶ **支持和外部系统的集成。**具体表现为对于企业内包括需求管理、缺陷跟踪、构建工具、部署工具等持续交付工具链的内置对接，以及二次开发的支持能力。

当前市面上定位于持续测试平台的项目和产品在不断涌现。其中最具代表性的产品有来自国外的 Tricentis Tosca 和来自国内的一站式开源持续测试平台 MeterSphere。

表3 两款具有代表性的持续测试平台对比

	Tricentis Tosca	MeterSphere
自动化能力	支持基于模型的 API 和 UI 测试 支持服务虚拟化能力 支持自动化性能测试能力 支持测试数据管理能力	支持类似 Postman 体验的接口测试 支持接口管理功能 支持兼容 JMeter 的性能测试能力 支持测试数据管理功能
测试管理能力	支持测试用例管理、测试计划规划、测试报告生成 支持基于风险的测试需求分析与测试用例设计	支持测试用例管理、测试计划规划、测试报告生成
团队协同	支持多租户体系 支持基于角色的权限管理体系	支持多租户体系 支持基于角色的权限管理体系
外部系统集成能力	支持主流持续集成工具对接 支持主流缺陷管理工具对接	支持主流持续集成工具对接 支持主流缺陷管理工具对接
产品分发方式	SaaS 分发为主	开源 + 企业版软件订阅 + SaaS
产品支持方式	线上支持	社区支持、线上支持、线下专业服务

从表3可以看出，Tricentis Tosca 平台的功能完整性要强于 MeterSphere 项目。而在产品分发和支持方面，由于 MeterSphere 来自国内团队，且采用开源方式运作，有明显的本土优势和社区化优势。按照 MeterSphere 项目的社区规划，接下来会逐步完善功能缺口，添加包括 UI 测试、服务虚拟化在内的各种能力工具。

3.1.6 重视行业最佳实践

尽管不同企业的测试活动运营情况以及人员能力各有不同，但仍然有很多大家共同认可的持续测试落地最佳实践。成功落地这些行业最佳实践会使得企业持续测试推进过程变得事半功倍。目前，行业内持续测试最佳实践主要集中在如下几个方面：

▶ **如何提升持续测试中的自动化占比？**显然，这个问题是大家最为关注的最佳实践。团队可以通过测试类型迁移、自动化测试工具采纳等手段提升持续测试的自动化占比。按照行业经验，自动化测试用例比较理想的占比需要超过 85%；

▶ **如何提升持续测试中的测试稳定性？**这里的测试稳定主要指自动化测试的稳定性。无论自动化测试占比有多高，如果自动化测试不能长期、稳定地运行，其价值都是值得怀疑的。在这方面，行业也积累了很多相关实践，例如测试环境管理、测试数据管理、服务虚拟化技术等最佳实践；

▶ **如何提升持续测试中的风险反馈质量？**作为质量保障实践，其根本目的在于高效反馈当前软件的质量风险。而软件质量风险的反馈能力很大程度上依赖于测试用例自身的设计。类似，行业为此也提出了基于业务风险覆盖度的测试用例设计实践等；

▶ **如何保持测试的持续性？**持续测试最终是为持续交付服务，其相关实践也需要融合到持续交付流水线中，从而成为持续交付过程中的质量保障体系。而且，持续测试强调测试活动是伴随整个业务系统全生命周期。即使系统已经交付上线，测试活动也不应该停止。这个融入过程也诞生了相关的最佳实践操作，值得团队学习和参考。

关于持续测试落地最佳实践的详细介绍，本白皮书后面还会深入展开讨论。具体内容可以参照后面相关章节的内容。

3.1.7 进行持续的量化跟踪

在推动持续测试实践的过程中，持续量化跟踪是另外一个非常重要的关键因素。只有可以量化的跟踪数据才能够给团队持续明确的反馈，也是持续争取到高层支持的最有效手段。这些量化指标可以是反馈效率、质量、成本或者业务风险覆盖等维度。以下是实践过程中常见的量化指标：

1. 测试用例业务风险覆盖度：根据业务系统进行业务风险评估后，设计的测试用例覆盖整个系统的业务风险比例。这是一个不同于传统测试用例逻辑覆盖度的指标，它用于反馈测试用户对于软件业务风险的管理，是一个比测试用例逻辑覆盖度更加重要的指标；

2. 自动化测试覆盖率：即自动化测试用例占所有测试用例的的比例。这个指标反映公司和团队向自动化测试转型的进度。一般来说，这个指标达到 85% 是一个比较理想的状态；

3. 测试用例回归通过率：即每次测试计划中回归用例的通过率。这一指标主要反映整个测试用例运行的稳定性。持续稳定的回归测试是帮助团队提升对持续测试实践信心的关键；

4. 测试成本节约率：通过自动化测试执行的用例产生的成本节约。这一指标直接反映了公司在自动化测试投入的产出，是持续获取高层投入的重要指标；

5. 测试平均执行时间：测试用例的平均执行时间。这个指标一方面反映自动化测试的效率提升，另外一方面还能指引团队将自动化测试实践和企业内 CI/CD 流水线进行合理结合，落实提交即测试和每日构建即测试等最佳实践；

6. 缺陷发现率：即每个测试用例平均能发现的缺陷数。这个指标反映出测试用例设计的有效性，以及持续测试执行的效率。更有效的测试用例设计和更高效的持续测试运行都可以有效提升缺陷发现率。

当然，企业和团队在实践过程中还可以发现更多、更适合自己的量化指标。持续测试落地过程中不断积累这些指标是其重要工作之一。但是，在制定这些量化指标时候，团队需要注意两个方面的问题：

▶ **量化指标一定是要有利于指导整个研发团队向正确的方向转型。**错误的量化指标只会比没有量化指标还要糟糕，因为它会把团队带向更加错误的境况；

▶ **量化指标一定是要能够通过技术手段持续进行跟踪。**永远靠人工进行统计的量化指标基本上都难以持续。即使可以持续，也存在量化跟踪代价大、准确度低和反馈慢的问题。

另外，在本白皮书最后的部分会提出一个基于行业普遍实践的持续测试成熟度衡量模型。这个模型框架对于团队进行量化跟踪也有比较好的指导作用，建议大家可以参考。

3.2 持续测试落地过程中的最佳实践

如前所述，持续测试的最佳实践主要关注解决测试自动化、测试稳定性、测试有效性和持续交付融合这几个方面。本章节将从这几方面深入展开探讨部分最佳实践内涵，以及落地过程中的具体操作。

表4 持续测试落地过程中的最佳实践

最佳实践类型	最佳实践主题
测试自动化	向接口测试自动化迁移；降低自动化测试维护成本。
测试稳定性	实现测试数据集中管理；提供服务虚拟化能力；管理好测试环境与测试基础设施。
测试有效性	基于业务风险覆盖度的用例有效性分析；建设精准测试能力。
测试持续性	最佳方式融入持续交付流水线；探索式测试；彻底地回归测试自动化。

3.2.1 向接口测试自动化迁移

如前所述，持续测试团队自动化测试占比超过 85% 是一个追求的目标。但是，如何达到这个目标就面临着测试用例类型的选择问题。当大家讨论自动化测试，通常想到的是端到端的 UI 自动化测试。使用类似于 Selenium、Robot Framework 或者 Appium 为主的框架编制脚本并运行。当然，这类自动化测试是非常有价值和必要的。但如大家所知，它也面临一些现实的问题：

▶ **自动化执行稳定性不够。**这类以 UI 为主的自动化测试脚本稳定性高度耦合用户系统的 UI 操作界面，不幸的是，UI 用户界面端是业务系统中变化最为频繁的部分之一。每次业务系统大的迭代都很大可能导致基于 UI 界面元素的自动化脚本失效，甚至有些时候需要完全重写自动化测试脚本；

▶ **自动化执行效率较低。**UI 自动化测试需要启动完整的应用客户端，模拟用户操作，获得界面反馈。这个过程运行链路较长，一个测试用例以分钟为单位运行是常见情况。当测试用例数量较多时候，一次小规模的自动化回归测试也需要几个小时，甚至以天计算的时长。这样的效率会带来一个明显的副作用，就是它很难友好地集成到持续交付流水线中的每次构建中，从而实现高频率的持续运行，以便给研发团队提供及时的测试反馈；

▶ **测试用例粒度过大。**UI 测试一般以端到端的测试场景为主，而验证手段也以界面表现为主。一旦测试用例执行失败，其并不能给出相对精确的测试失败原因，给研发团队修复问题提供明确指导。

如上所述，利用 UI 自动化为主的测试用例类型很难达成推动测试自动化占比大幅提升的目标。相比于 UI 自动化测试，接口自动化测试更加适合成为测试自动化的主要测试用例类型，因为它非常适合持续测试理念下的自动化需求（如表 5）。

表5 UI 测试与接口测试的对比

类型特征	UI 测试	接口测试
执行效率	端到端执行，每个用例典型执行时间在数分钟级别。	单接口或者基于接口编排的场景执行，每个用例典型执行时间在 1 分钟以下。
实现难度	依赖前端客户端录制技术，需要较为复杂的脚本编写能力。	基于接口规范进行设计，设计以界面操作为主，实现难度较低。
执行稳定性	高度依赖普遍频繁变化的前端 UI 界面，用例失效风险大。	依赖接口稳定性。大部分应用系统都会保持接口向前兼容，实践过程中稳定性有保障。
发现问题有效性	失败用例只能给出大体排查方向，需要团队深入寻找失败原因。	基于接口进行良好设计的用例能做到精准定位问题场景和原因。

基于以上分析可以看出，接口测试是整个测试自动化中最应该追求的测试类型。如果需要推动整个测试自动化占比，应该优先推动接口自动化的实践。一般来说，接口自动化测试用例应该占持续测试团队自动化用例超过 85%。

当前，对于大部分测试团队，为了推动测试自动化实践的落地，一个关键的工作就是实现测试自动化的工作重心从 UI 自动化向接口自动化迁移。这个迁移过程的典型过程如下：

▶ **首先，面向存量系统接口进行测试。**测试团队与开发团队需要在系统接口定义上形成更多的沟通，乃至形成统一的接口管理平台。测

试团队需要基于当前接口定义进行接口测试用例设计、实现和执行，并通过接口测试自动化形成和开发团队的高效反馈机制。比如，实现接口测试每日回归，并及时和开发团队反馈回归结果。在这一阶段，接口测试用例的典型形式是单接口测试（围绕一个单接口形成正反结合的用例组合）；

▶ **其次，拆分现有自动化测试场景中的接口测试场景。**很多测试团队已经有成千上万的端到端 UI 自动测试脚本，并在日常过程中每天运行。测试团队需要去分析这些自动化测试场景，判断是否可以从其中剥离出相应的接口自动化测试场景。通常来说，优先选择通过接口测试去验证后端业务逻辑，将 UI 测试保留用于界面展现验证。通过这个拆分工作可以发掘出大量进行接口编排测试的场景；

▶ **最后，配合业务系统接口改造进行接口测试。**这是一个需要开发和测试团队共同努力的工作。不过好在随着前后端分离和微服务框架技术的普及，越来越多的开发团队也有很强意愿进行相关的改造（这个改造工作对开发团队的价值也同样明显）。在这个改造过程中，测试团队需要尽早参与，争取按双方约定的接口规范尽早实现必要的接口测试自动化，从而通过接口自动化测试反向驱动开发业务改造过程中的质量保障，实现业务改造过程中的“测试驱动开发”。

总结来说，持续测试中的测试自动化实现高度依赖于接口测试自动化的实现。研发团队需要将其测试自动化的主战场切换到接口测试上，并通过以上介绍的相关实践方式推动接口测试自动化落地。

3.2.2 降低自动化测试维护成本

自动化测试另外一大挑战就是维护成本问题。这个维护成本包括两个典型原因。**其一是自动化测试用例自身实现不稳定。**被测对象并没有发生变化，但测试用例因为自身实现不稳定而带来维护成本。解决这个问题一般需要去寻找更加稳定的方式设计和运行当前测试用例。比如，测试用例脚本中的断言判断规则是否可以更灵活，UI 测试中的元素 ID 寻找方式是否可以更智能等。当然，如果测试用例运行高度依赖外部测试数据或者服务，后面讨论的测试数据管理和服务虚拟化也会对降低这类测试用例维护成本有很大帮助。

其二是用例更新困难。随着业务系统的不断更新，自动化测试用例也要保持更新以维持有效性。如前所述，解决这个问题最主要的途径是实现测试用例类型的迁移，让自动化测试用例更多关联相对稳定的接口定义，而不是频繁变化的 UI。另外，模块化复用也是解决这个问题非常关键的手段。测试用例管理自身也可以分层。实践中可以把简单独立的原子场景进行编排，形成一个测试用例模块。更高级的测试场景利用这些测试用例模块进行组装。

例如，一个典型的电商系统通常包括如图 6 所示的完整流程：

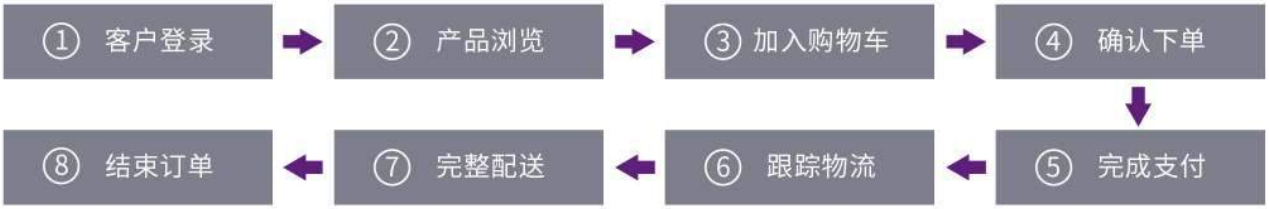

图6 典型电商系统的示例流程

假设需要对以上流程进行接口测试设计，比较合理的设计方案是对其中每个环节内的接口测试用例进行模块化管理，然后基于这些模块灵活组装不同新的复杂场景。以上每个环节内部也可以包括多个接口测试用例。例如“加入购物车”环节不仅仅要测试加入购物车的接口是否工作，还要验证购物车内查询接口运行是否符合预期。通过这种模块化管理和复用的方式，用户每个环节的更新变化大多时候都只关联到对应模块内的用例更新，而不需要更新上层依赖这个模块的复杂用例场景。

Codeless 自动化测试也被认为是降低自动化测试维护成本的重要手段之一，业界也在这些方面做不断的尝试。这类手段最主要的努力方向主要有以下几个方面：

▶ 通过引入更加友好的自动化测试交互界面，让大量的测试用例和脚本的维护可以用直观的界面交互方式完成，从而降低维护这些用例和脚本的成本。同时，这种模式也能有效降低测试人员的技术门槛；

▶ 通过引入用例执行过程中的高级“自愈”功能，让用例执行过程对于其背后依赖的环境和被测系统有更好的适配性。比如，Web UI 测试能够自动化尝试多种模式匹配界面上的目标元素；

▶ 通过引入更加友好的“录制 - 回放”机制支持用例和脚本的自动生成过程。例如，“录制 - 回放”工具越来越能和其他工具结合，实现基于数据驱动的用例生成机制，对降低用例维护成本的效果非常明显。

改进自动化测试用例维护成本的另外一个重要手段在于测试用例的设计思路。基于业务风险的测试用例设计强调用尽可能少的测试用例覆盖尽可能高的业务风险，从而避免维护大量无实质业务风险价值的无效用例，整体上减少测试用例的维护成本。关于如何有效进行基于业务风险的测试用例设计，请参考本白皮书后面的章节。

3.2.3 实现测试数据集中管理

测试数据的准备和管理是自动化测试落地的另外一个难题。在自动化测试过程中，测试初始环境的数据准备、测试过程数据的输入，以及测试结果的验证都离不开测试数据。为此，测试数据经常分成以下几类：

▶ **状态性数据：**描述测试用例开始前的系统状态数据。包括如数据库初始数据、系统依赖的磁盘数据、内存数据等。这类数据除了被系统自身使用外，还有可能用于系统依赖的外部服务虚拟化环境中进行状态初始化；

▶ **输入性数据：**描述测试用例执行的时候需要输入的数据。这类数据有可能直接嵌入到测试用例设计内部，也有可能需要依赖外部输入（比如文件等）；

▶ **验证性数据：**描述测试用例执行验证过程中的比对数据。类似于输入性数据，这类数据可能直接嵌入到测试用例设计内部，也有可能来自外部加载。

不管是以上哪种测试数据，随着测试用例规模的增加，其规模也在不断膨胀，维护成本也在快速上升。所以，集中化测试数据管理会成为很多持续测试团队必须要面对的问题。一般来说，这个集中管理平台需要具备以下几个方面的能力：

▶ **测试数据的输入和合成能力。**测试数据集中管理平台内的数据要么来自外部现成数据的输入，要么来自内部的自动合成。从外部输入角度看，需要支持实际使用到的各种数据和文件格式，并能够支持必要的数据格式检查，以保障输入数据的质量。而内部自动化合成能力也同样重要，因为实践过程中设计良好的合成数据能够帮测试团队覆盖大部分的测试场景。以最常见的“手机号”字段数据随机生成器，其应该具备批量生成包括不同运营商正确号码的能力，同时也应该能够伪造出一些常见的不合法格式的手机号。这样的数据合成能力会大幅度提升测试效率；

▶ **测试数据的版本管理能力。**测试数据集中管理平台需要支持对一个数据对象进行版本管理的能力，以适应不同版本测试用例和测试脚本的需要。对照测试数据的使用场景，其版本管理能力只需要能够对单一数据对象不同版本的标识和应用能力即可，不需要太多其他版本管理系统的高级特性；

▶ **测试数据使用场景集成能力。**无论是测试用例、测试脚本，还是服务虚拟化都有可能使用到测试数据。测试数据集中管理平台需要能和相关使用场景形成无缝的集成，让整个测试数据的使用过程变得简单直接。例如，一个接口测试场景需要一组输入数据，那么它应该可以在接口测试用例里面指定这组数据在集中测试平台中的数据对象地址和版本，然后在测试执行过程中能够通过标准协议自动获取相应的数据并用于测试。

对于自动化测试团队，当自动化测试用例到达一定规模后，并且测试数据已经成为日常自动化测试运行不稳定的主要原因之一时，就应该重视测试数据集中管理平台。团队可以基于公司内部的数据管理平台自我研发，也可以采纳外部专业测试平台提供的类似工具。

3.2.4 提供服务虚拟化能力

服务虚拟化（即日常说的 Mock 服务）是另外一个被经常提到的持续测试最佳实践。之所以服务虚拟化对于持续测试如此重要，是因为它是解决自动化测试中外部依赖不稳定或者不可得的关键手段。用户被测系统通常并不孤立存在，它依赖各种外部系统才能正常运行。但是，在测试环节，这些外部系统依赖可能会因为各种原因不具备支持持续测试正常运行的问题。常见原因有如下列表：

▶ 外部依赖系统还未开发完成，不具备对外提供服务的能力。这常见于多系统并行开发的场景；

▶ 外部依赖系统无法正常稳定访问。比如，外部依赖系统在测试环境未部署，或者在第三方不可控地方部署，无法在时间或者空间上满足自动化测试稳定运行的要求；

▶ 外部依赖系统在容量或者性能上满足不了当前持续测试的需求，导致部分测试用例无法正常完成；

▶ 外部依赖系统无法或者不适合模拟部分特别的场景或者数据，而部分测试用例又依赖于这些场景和数据；

▶ 外部依赖系统用于持续测试的成本太高，公司无法在持续测试环节承受这个成本。

需要强调的一点是，持续测试期待自动化测试能够持续、长期和高频率的运行，所以其对外部依赖系统的要求也是如此。这一点也更加重了持续测试对于服务虚拟化的依赖。团队在建设服务虚拟化能力时，需要注意以下几个方面的要点：

▶ 服务虚拟化需要能够快速、准确地模拟外部依赖系统。服务虚拟化是为了满足测试的需要，其实现过程是典型的“测试驱动开发”模式。在明确完测试需求后，整个实现过程不需要关注太多真实的业务逻辑，而是关注接收正确的输入，并快速产生预期输出即可；

▶ 服务虚拟化要尤其关注自身的稳定性。构建服务虚拟化的一个初衷就是解决外部系统的依赖的稳定性和可获得性。如果服务虚拟化能力自身不能稳定运行，就无法支撑持续测试高效、高频地持续运行。

市面上广泛存在面向不同阶段需求的服务虚拟化框架，持续测试团队可以按照服务虚拟化使用的场景进行合理选择，以加速相应能力的建设。另外，服务虚拟化自身也需要合理的集中管理，并与测试数据、测试脚本等进行合适的联动。

3.2.5 管理好测试环境与测试基础设施

测试环境准备是消耗测试团队很多时间的重要环节之一，也经常成为企业持续测试落地过程中的关键障碍。因为部署方式和使用目的的不同，测试环境分成如表 6 所示的不同类型：

表6 不同类型测试环境的分类

环境类型	部署方式	使用目的
本地测试环境	研发人员本机	研发人员用于单元测试或者组件测试
集成测试环境	测试环境独立部署	研发团队进行系统功能集成测试
验收测试环境	测试环境独立部署	产品团队进行系统功能验收和验证测试
兼容性测试环境	测试环境独立部署或外部租用部署	产品运行于多种运行环境的适配测试
性能测试环境	测试环境独立部署或外部租用部署	产品性能压测环境，通常包括大量压测集群
模拟仿真环境	类生产环境独立部署	研发团队进行真实流量验证和预发布测试

当然，企业还可能因为某些特殊测试需求而建设未在表 6 上包括的测试环境。由此可见，测试环境种类繁多，其维护成本也必然不低。尽管如前所述，通过集中的测试数据管理和服务虚拟化建设，可以有效解决测试环境对于外部数据和服务的依赖。但是，被测系统自身测试环境的部署和维护仍然充满挑战。这些挑战具体表现在以下几个方面：

▶ **不具备构建不同测试环境的条件。**这个挑战的原因可能来自于成本的考虑，也有可能来自于公司内基础设施建设和管理的原因。相关测试环境的缺少会明显影响测试活动的展开和持续运行；

▶ **不具备快速构建测试环境的能力。**即使企业有条件构建不同的测试环境，但是缺少快速构建的能力。不同测试环境从申请到搭建投产时间仍然会是一个巨大的瓶颈。在持续测试实践过程中，用户业务系统迭代速度快，其对环境要求的变化也会频繁，所以快速构建全新的测试环境能力是保障整个持续测试活动进行的重要因素；

▶ **缺乏环境一致性管理的能力和手段。**研发团队在测试活动遇到的一个常见问题是测试环境的不一致性。包括系统自身版本一致性问题、

依赖包一致性问题、基础设施环境一致性问题等。这些问题很多时候导致测试运行不稳定，对这类问题的排查也经常耗费研发团队大量的时间。

测试行业一直致力于解决以上这些问题。在过去不断实践过程中，虚拟化与容器技术、云计算技术成为解决以上问题的关键所在。

▶ 云计算技术，尤其是公有云的出现让大量企业可以用非常低的成本高效获得测试环境。无论是最基本的计算、存储和网络能力，还是各种高级的 PaaS 类服务能力，乃至各种兼容性测试需要用到的测试设备都能提供；

▶ 虚拟化技术和容器技术给多环境一致性管理带来了新的突破。虚拟化技术通过机器镜像解决整个运行机器的环境管理，而容器技术则采用更为轻量的运行时封装模式进行构建，从而屏蔽了业务系统依赖的底层基础设施的差异。研发团队只需要在整个测试周期中管理好虚拟机及容器的镜像版本，就基本能够保障多个测试环境的一致性；

▶ 自动化技术大幅度提升环境搭建的效率。无论是云平台提倡的基础设施即代码（IaC），还是容器领域的容器编排技术（以 Kubernetes 为代表）都让环境的自动化部署难题得到了极大的缓解。

持续测试团队在建设整个持续测试能力时，以上这些技术应该需要被重点考量并进入到整个实践规划过程中。当然，这些能力的建设和采纳可能需要涉及到跨团队的协作，但这仍然是非常值得投入的事情，毕竟测试环境是一切持续测试的基础保障。

3.2.6 基于业务风险覆盖度的用例设计

当讨论用例有效性问题时，首先我们需要明确一个现实：对于现代复杂软件系统进行全覆盖测试是一个几乎不可能（或者说在经济上不允许）的选择，这就意味着必须有所选择。在做这个选择时，大家需要回到测试活动的初衷，即保障产品质量风险，从而控制产品业务风险。因此，对系统测试场景选择遵循的第一原则就是场景的业务风险。而考核测试对产品质量风险的保障力度也就转换成为测试对于系统业务风险的覆盖度。为了度量这个覆盖度，我们一般需要进行以下两步操作：

第一步：对系统需求进行基于业务风险的分析

首先，团队需要理解什么是业务风险。对于任何一个软件系统需求，其背后都会对应业务场景，以及这个场景出现缺陷导致的业务影响。而业务风险可用如下公式进行定义：

业务风险 = 业务发生概率 x 业务影响程度

依照以上公式，可以看出高频发生的场景和带来严重业务影响的业务需求经常有较高的业务风险。团队在做系统需求的业务风险主要就是识别这两种情况，并最终结合上面的公式进行业务风险的判定和排序。

要分析业务发生的概率或者业务影响程度，都需要有一个参考基准。通常建议团队选择一个典型且简单的系统需求来做基准判定。例如，对于一个资产管理系统，我们可以把“查看资产详情”定义为基准场景。为量化分析业务风险，可以将其“业务发生概率”和“业务影响程度”都定义为 100。以此标准依次分析出其他业务场景的业务风险。具体数值可能如表 7 所示：

表7 资产管理系统业务风险分析表（示例）

需求简述	发生概率	影响程度	业务风险值
查看资产详情（基准）	100	100	10000
增加全新资产	10	500	5000
删除存量资产	5	50	250
更新存量资产	50	100	5000
查询相关资产	200	200	40000
打印资产清单	10	10	100
生成资产报表	20	10	200

当然，团队还可以对以上每个场景内的业务风险再做进一步的细化分析。比如，团队可以对查看资产详情内的各种子场景再展开分析，并确保子场景的业务风险累加数值和主场景风险值一致即可。在完成基于业务风险的需求分析后，研发团队基本清楚了业务系统真实的业务风险分布情况。如你所期，这个业务系统风险的评估过程需要经常回顾，尤其是在不断积累生产运营经验后，需要团队来重新确认以前的判断是否合理。

第二步：按照业务风险覆盖度进行用例设计

完成业务系统需求风险评估后，团队需要以业务风险覆盖度为最终追求目标进行测试用例设计。整个用例设计过程的一个基本指导原则就是：用尽可能少的用例覆盖尽可能多的业务风险。具体来说，这个过程可以细分成如下两个过程：

▶ **首先，按场景的业务风险度进行场景选择。**这基本上都是从最高业务风险场景开始选择并进行测试用例设计的。至于需要达到多高的业务风险覆盖度，这会因不同系统和不同团队而异，但无论如何都需要有一个可以明确量化的指标来作为参考；

▶ **其次，对于每个选择场景进行详细用例设计。**对于每个场景，都可能有多维度输入，并且每个维度输入还会有多个可能选项。为此，软件测试领域有一系列的用例设计方法，例如组合覆盖、正交法、Pair-wise 等。在这其中，线性膨胀法（Linear Expansion）被认为是一个在用例数量和业务风险覆盖度之间很好平衡的方法。下面我们用一个具体例子说明线性膨胀法的工作方式。

如前面的资产管理系统，需要为“添加全新资产”进行测试用例设计。已知添加每个新资产需要提供“资产编号”、“资产类型”、“所属部门”三个必选参数。其中，按等价划分法把资产编号分成三类输入，即“非法资产编号”、“合法有效新编号”、“存量冲突编号”；资产类型有二类：即固定资产、流动资产；所属部门有三类：即业务部门、研发部门和后勤部门。

在线性膨胀法里，首先需要在多种可能组合中寻找到一个最主要组合，也是业务风险最大的一个组合。例如，团队一致认为“合法有效新编号”、“固定资产”、“业务部门”这样的组合为最主要组合。线性膨胀法就会先按这个主组合设计第一条用例。然后，在每条新加用例保持和主组合只有一个维度的数值不同。以此类推，直到所有维度选型覆盖完成。这一场景的用例列表如表 8 所示：

表8 基于线性膨胀法设计的测试用例（示例）

用例编号	资产编号	资产类型	所属部门	备注
01	合法有效新编号	固定资产	业务部门	主组合
02	非法资产编号	固定资产	业务部门	资产编号发生变化
03	存量冲突编号	固定资产	业务部门	
04	合法有效新编号	流动资产	业务部门	资产类型发生变化
05	合法有效新编号	固定资产	研发部门	所属部门发生变化
06	合法有效新编号	固定资产	后勤部门	

通过表 8 可以看出，线性膨胀方法的完整测试用例数量为 6 个。相比较来说，组合覆盖需要设计 18 个测试用例（18=3x2x3），而正交法虽然只需要设计 3 个测试用例，但是其对业务风险的覆盖度又明显低于线性膨胀法。

线性膨胀法还会有另外一个优势。就是当用例运行失败后，可以非常方便地定位问题。实际工作中主组合是被反复测试的场景，一般容易保障其成功率。一旦其他组合出现问题，由于和主组合只有一个维度的输入值不同，非常容易定义到哪个维度带来的错误。甚至，通过友好设计的用例命名就可以帮助我们知道问题的精确方向。

经过基于业务风险覆盖率的测试用例设计，团队测试用例的有效性会得到大幅度提升，可以有效避免大量无明显业务价值的测试用例浪费团队日常实现和维护时间。而且，由于其能有效帮助定位问题，测试用例运行报告会对开发团队带来明显的问题定位价值，更容易体现测试活动的业务价值。

3.2.7 建立精准测试能力

测试有效性的另外一个重要实践就是精准测试。顾名思义，精准测试就是让测试活动能够有针对性地精确发生。它强调整个测试活动应该在明确、精细的度量指标指导下进行，并且需要对整个测试活动进行持续有效监控，及时采集到度量指标的反馈来促进测试过程的不断完善。从以上定义可以看出，精准测试需要包括如下构成要件：

▶ **度量标准：**一组精确监控测试活动质量的度量标准和数据。由于软件产品本质是由软件代码构建而来，所以软件系统精准测试最常用的度量指标自然就和代码关联起来。所以，大家经常采用的软件精准测试度量标准有代码覆盖率、分支路径覆盖率、函数方法覆盖率等；

▶ **测试活动：**对目标系统的测试活动。在精准测试下的测试活动与其他测试活动大体相同，但是精准测试下的测试活动中需要持续收集和监控度量标准下的度量数据；

▶ **度量分析：**根据测试活动中产生的度量数据进行集中分析，发现测试活动中的盲点和潜在问题；

▶ **过程改进：**根据度量分析的结果，指导测试活动的改进。例如补充部分测试用例，修改测试用例的运行顺序等。

按照以上构成要件，推行精准测试的团队需要建立以下能力：

▶ **建立度量数据的采集能力。**精准测试依赖度量数据，所以团队需要能够在测试活动进行过程中全程监控并采集到需要的度量数据。由于软件精准测试的度量标准基本是和代码相关，这个采集能力通常需要给被测系统嵌入“插桩”组件（通常是在构建过程中集成到被测系统中），收集测试活动时候的代码具体运行情况；

▶ **建立度量指标的分析评价能力。**分析采集到的度量数据，结合对于系统源代码的静态分析，从而可以将原始度量数据转换成为度量指标维度的数据，并按照系统的属性和级别指定相应的评价模型；

▶ **建立代码与用例的双向关联机制。**精准测试的最终目标仍然是改进测试过程。所以需要建立测试用例和系统代码之间的双向关联机制。让基于代码的度量指标能够关联到相关测试用例，从而达到改进测试过程的目标。

在具备以上精准测试能力后，团队进入精准测试落地运行的环节。和其他持续测试最佳实践类似，整个持续测试的运作过程仍然需要和持续交付流水线的深度融合。融合后的整体运作模式见图 7。

图7 精准测试落地环

1. 将精准测试的度量数据采集过程融入到持续构建过程中。可以通过构建过程加入“插桩”组件或者在代码内框架加入“染色”日志等方法，让构建出来的系统可以在测试过程中持续对外输出需要的度量信息；

2. 将构建过程产生的制品包通过持续部署过程部署到测试环境，以便接下来测试活动的展开；

3. 基于用例库进行持续测试活动。需要注意的是，这里的持续测试活动可以是各种类型的测试活动，包括单元测试、集成测试、功能测试乃至性能测试等。所以，精准测试实践是可以应用到持续测试中的各种测试类型上；

4. 基于测试活动产生的度量数据，结合对代码库的静态分析，将度量数据输入到度量库中。团队再基于代码和用例之间的关联关系将度量数据用于指导改进测试活动。

在实践过程中，精准测试的效果非常明显，尤其是在智能筛选回归用例集，快速定位缺陷具体位置等方面有良好的表现。而且，由于精准测试对于测试改进有精确指导方向，能够帮助团队发现传统方式不容易覆盖的业务逻辑和代码片段，从而将整个测试用例的代码覆盖率提

升到传统用例设计很难达到的新高度。

3.2.8 最佳方式融入持续交付流水线

持续测试一个重要的实践是要将测试活动融入到持续交付流水线，从而实现测试活动在交付过程中的内建。考虑到持续交付最核心的载体就是交付流水线，实践过程中普遍会将测试活动集成到持续交付流水线中。但是在这个集成过程中，需要确保以下两个方面的影响可控：

▶ **对于持续交付执行效率的影响。**持续交付执行的频率非常高（如每日构建或者提交时构建），且开发团队高度依赖这些活动的及时反馈来开展日常工作。所以，这些活动的执行时间不能太长。例如，提交时构建需要在 30 分钟级别完成。而每日构建也需要在 3~5 个小时内完成。加入测试工作势必影响它的执行时间，团队需要控制被集成的测试活动对整体执行时间的影响；

▶ **对于持续交付执行稳定性的影响。**持续交付过程稳定运行是其提供反馈的基础保障。被集成的测试需要确保其自身的稳定运行，从而不会影响整个持续交付过程的稳定性。

一方面，需要确保被集成到持续交付流水线中的测试活动能够稳定高效运行，另外一个方面还需要追求测试对质量保障的有效性。这就要求能够对测试用例进行有效的影响分析，确定不同阶段应该执行的测试范围。整个测试范围选择的基本策略可以从当前软件变化 / 生产缺陷和业务风险覆盖度这两个维度考虑，推荐如表 9 进行交付流水线的融入策略选择：

表9 测试范围选择的基本策略

	高业务风险覆盖用例	中业务风险覆盖用例	低业务风险覆盖用例
有明显变化或生产缺陷场景	高频融入流水线 （如每日构建）	高频融入流水线 （如每日构建）	中频融入流水线 （如版本回归）
无明显变化或生产缺陷场景	中频融入流水线 （如版本回归）	中频融入流水线 （如版本回归）	低频融入流水线 （择机执行）

如前所述，以上策略在应用时还需要考虑前面提到的测试用例运行效率和稳定性。对于越需要高频融入流水线的用例，越需要花费时间保障其中执行效率和稳定性。需要注意的是，测试范围的选择是一个持续的过程，团队需要在每个研发周期内进行相应的回顾，并判断当前测试范围的选择是否仍然适合。所以，这也给自动化测试工具提出了要求，需要能够非常方便地调整融入到流水线上的测试范围。

3.2.9 探索式测试

以上描述的各种持续测试实践对于判定规范明确的测试非常有效。但是，在实际测试实践中，测试团队仍然需要进行一些非明确规范的测

试工作。业内把这种无明确判定规范，但体现设计、执行和学习同时进行的测试工作统称为探索式测试。由于其无明确判定规范，探索式测试一般无法有效自动化，需要大量的手工工作。尽管持续测试非常强调自动化测试，并以此为基础融入到企业软件持续交付流水线中，但这并不意味持续测试就否定探索式测试的必要性。相反，持续测试仍然认可探索式测试是整个敏捷测试中的重要一环。其主要原因在于：

▶ **探索式测试可以帮助测试团队发现很多其他测试方法发现不了的问题。**软件系统的质量不仅仅只有“是否按预期运行”这一个评价标准，系统是否“好用”很多时候是另一个关键的评价标准。探索式测试可以帮助团队发现系统的设计逻辑是否合理，有无让用户困惑，某个功能和其他功能是否协调等诸多这样的问题；

▶ **探索式测试是一个增强整个软件研发团队对于软件产品质量理解的有效手段。**在实践过程中，测试团队经常召集包括开发、运维、设计、产品等多个不同角色参与产品的探索式测试（例如“bug bash”活动）。这种测试是一个非常好的契机，让软件生产过程中各个不同角色的人重新理解产品质量，并进行跨角色的沟通，有效地发现潜在缺陷和现有测试的局限性；

▶ **探索式测试还是对新功能进行常规自动化测试前的有效测试手段。**不难想象，自动化测试对于回归测试非常有效。但当新功能开发完成后，进行以主流程为目标的局部探索式测试可以非常快速地发现常见缺陷，并快速修复。在实践过程中，这种测试很多时候是由测试团队提供用例和工具，开发团队在完成开发后快速执行的。这种场景下，探索式测试表现出来的执行效率和灵活性通常会非常好。

正因为以上多种原因，探索式测试仍然应该是持续测试中非常重要的组成部分。但是，由于以手工执行方式为主、测试质量高度依赖参与人员的特征，团队在执行探索式测试需要注意以下几个方面：

▶ **严格明确探索式测试的范围。**避免将很多可以明确自动化的功能扩散到探索式测试范畴，从而降低了整个测试的效率和质量。在探索式测试过程发现了部分有明确功能规范的缺陷，应该更新到自动化测试用例进行持续的自动化回归验证；

▶ **探索式测试设计和安排要确保“探索式”特征。**在探索式测试过程（尤其是如“bug bash”这样的活动）中，测试团队给出的测试指引应比较宽泛性，要避免给出那种明确测试路径和预期结果的指引；

▶ **要寻找更广泛的人群参与探索式测试。**除了上面提到的软件研发团队中的多个角色，还可以寻找公司内其他团队，乃至部分种子客户参与进行测试。越广泛的测试群体意味着更多的测试视角和更广泛的测试覆盖度；

▶ **对探索式测试进行认真进行总结和回顾。**测试团队需要牵头完成对每次探索式测试的总结，形成明确的测试报告，并召集参与人员对测试执行情况回顾分析，形成可执行的改进意见。

3.2.10 高度自动化的压力测试

压力测试是经常需要面对的一个测试场景，在持续测试实践过程中也同样如此。不同于其他测试类型关注系统功能运行情况，压力测试主要关注系统在受压情况下的性能表现，并以此来优化系统性能，准备合理的系统容量。由于国内互联网的巨大用户规模和各种层出不穷的市场活动，压力测试在国内软件测试的重要性变得非常关键。过去几年也确实频繁出现因为过大线上压力而导致的系统奔溃事故。在持续测试的理念下，压力测试也是一个需要持续高效运行的测试类型。为此，持续测试团队在推动压力测试过程中需要注意以下几个方面：

▶ **结合业务实际需要，明确压力测试的关键指标。**大家熟悉的各种国内电商大促活动，尽管其背后涉及的系统和模块极其复杂，但是在系统压力测试上一般都会定义几个比较非常清晰的测试指标，比如“每秒订单数”和“每秒支付数”。有了非常清晰的技术指标后，所有压测流量模型和压测用例设计就非常清晰。所以，研发团队需要和业务团队进行深入沟通，定义出系统压力的关键测试指标；

▶ **坚持以自动化地方式从局部到全链路逐步推进。**对于绝大部分测试团队来说，全链路压测的落地要求都显得过高，而且其中还会涉及到大量的跨团队协作。所以，压力测试实践落地更适合在局部先展开。这个局部可以是局部的组件和接口，也可以是局部的场景。通过从局部进行实践，找到关键的性能瓶颈并指导开发团队进行改进，或者指导运维团队进行合理容量规划。与此同时，压力测试的自动化执行是一个更为重要的目标。只有能够自动化执行压力测试，才能够实现在系统迭代过程中持续稳定地运行，从而及时发现系统的性能滑坡问题；

▶ **适当引入外部专家服务。**性能测试是一个非常依赖于执行人员经验和素质的测试类型，而这些经验和素质需要大量的实践积累。很多测试团队缺少压力测试经验，适当引入外部专家服务是解决这一问题的有效手段。外部专家可以在两个方面帮助到测试团队：一是在测试用例设计和流量模型规划上。由于外部专家长期执行压力测试，更有经验发现系统的性能瓶颈，并针对性地设计测试用例和流量模型；二是对于测试结果的解读和解决方案的指导。外部专家可以更深入地读懂测试结果并提出针对性的解决方案，包括对于性能瓶颈的定位和系统容量的评估。

3.3 拥抱新技术

和其他任何领域类似，测试领域也在不断涌现出新的技术。持续测试团队需要对此保持必要的敏感度和积极的拥抱态度。目前，在测试领域有以下几个方面的新技术值得测试团队高度关注：

▶ **利用线上历史流量指导测试工作。**在测试领域，如何真实模拟用户的线上行为永远是一个关键的问题，而线上历史流量则是最接近这个模拟需求的。这个领域涉及到非常多的技术环节，包括流量的抓取、存储和回放等，有非常多可以研究和探讨的地方。最近各个测试领域技术峰会可以看出这个领域保持着极高的热点，也是大型互联网公司测试团队重点尝试的方向。建议测试团队保持高度关注，有可能的情况下可以在局部项目进行尝试，并积累技术能力和操作经验；

▶ **应用人工智能（AI）技术到测试领域。**人工智能是一个存在已久的话题，尤其是进入 2010 年后，随着深度学习（Deep Learning）技术的突破，机器已经在某些特有领域具备了类似（甚至超过人类）的技能。例如，利于深度学习技术，经过多次迭代学习来模拟人使用软件的行为，最终构建虚拟测试机器人协作测试团队工作。另外，精准测试也是从人工智能受益比较大的领域之一，结合人工智能、代码静态分析和存量用例库进行综合分析，可以用于自动推荐新的可能测试用例。总体来说，这个领域值得关注，但是离实践落地还有一定距离，建议测试团队保持关注，及时了解行业进展。

当然，测试领域还有很多其他新技术的发展，这里不再一一列举。作为测试领域的从业人员，保持开放态度，并能积极思考新技术和实际工作之间的关联性是一个好的思维实践。

4. 持续测试成熟度能力模型

作为一项复杂且长期的企业实践，合适的度量模型是持续测试执行过程中的重要一环。根据以上章节的阐述和行业内的实践经验，本章构建了以下持续测试成熟度能力模型。期待这个模型框架可以帮助企业更好地定位自身当前状态，并能建立相对清晰的努力方向和路径。

4.1 成熟度模型级别说明

企业研发团队的持续测试成熟度级别定义如下：

▶ **Level 1**：团队处于持续测试实践的原始阶段。用例管理分散且用例设计质量高度依赖测试团队个人的能力，测试执行以手动测试为主，无任何高级的持续测试实践被常规化采纳；

▶ **Level 2**：团队处于持续测试的初步尝试阶段。开始积极引入各种自动化测试能力，融入到持续测试实践体系。但是整个努力仍然在局部和初级阶段，还未形成稳定的常规实践；

▶ **Level 3**：团队处于持续测试的高级实践阶段。已经在各个方面形成较为稳定的持续测试实践落地，并且在局部已经达到较高的实践能力。团队日常工作高度依赖这些持续测试实践运营；

▶ **Level 4**：团队处于持续测试的专家实践阶段。团队已经熟练掌握持续测试各个方面的关键实践，团队日常运营效率高，对软件测试的其他环节还能形成了非常有力的支持。与此同时，团队积极探索超出现有持续测试实践的其他方面工作。

4.2 成熟度模型的构成

本着紧贴用户实践的原则，本成熟度模型选择从具体的持续测试实践维度来定义。具体的维度包括测试用例设计与管理、测试用例执行、测试环境管理、交付流水线集成和高级测试实践采纳。整个成熟度模型定义如表 10 所示：

表10 持续测试成熟度模型的级别定义

测试维度		Level 1	Level 2	Level 3	Level 4
测试用例设计与管理	测试用例管理模式	无	低	中	高
	测试用例设计	无	低	中	高
测试用例执行	接口测试与集成测试自动化	无	中	高	高
	UI 测试自动化	中	中	中	高
	基于脚本的手工测试	高	中	低	低
测试环境管理	测试数据管理	无	低	中	高
	服务虚拟化	无	低	中	高
	测试环境能力	无	低	中	高
交付流水线集成	CI/CD 流水线集成	无	低	中	高
高级测试实践采纳	探索式测试	无	低	中	高
	非功能性测试	无	低	中	高

上表中各个维度的“无、低、中、高”评判标准的描述如表 11 所示：

表11 持续测试成熟度模型分维度评价标准

评定维度\评定级别	无	低
测试用例管理模式	用例管理以线下个人管理为主，通过简单的文件方式进行共享和协作。	建立了用例线下用例协作方式，有较为规范的线下用例评审机制。
测试用例设计	测试用例设计无团队层面统一指导思路，以个人设计为主。	初步具备测试用例设计的逻辑覆盖度思维，用例设计会进行日常评审。
接口测试与集成测试自动化	无常规性接口与集成测试自动化实践。	初步引入脚本或者工具进行接口与集成自动化测试，但以单接口为主，且接口与集成测试自动化占比较低（低于 50%）。
UI 测试自动化	无 UI 自动化测试，以手工测试为主。	引入脚本或者相关工具进行局部的 UI 测试自动化，并未变成测试团队常规测试内容。
基于脚本的手工测试	无基于脚本的手动测试。	在局部有相应基于脚本的手动测试。
测试数据管理	测试数据以个人管理为主，生成机制随机，质量无法保障。	通过文件或者数据库形成初步的测试数据集中管理，但是无合理的版本管理机制，无数据生成和质量保障机制。
服务虚拟化	未建设任何服务虚拟化。	通过脚本或者工具构建了基本的服务虚拟化，并在测试自动化中使用。
测试环境能力	测试环境不完备，部署实践长，无有效的一致性管理能力。	初步具备分工明确的不同测试环境，部署过程仍以手工或者脚本为主，环境一致性管理问题仍然比较严重。
CI/CD 流水线集成	未和 CI/CD 流水线形成任何形式集成。	和 CI/CD 流水线形成初步集成，可以在流水线中触发自动化测试。
探索式测试	无任何特意组织和规划的探索式测试。	在软件研发局部阶段有嵌入必要的探索式测试，例如新功能开发后的快速功能测试。
非功能性测试	无任何有意组织和设计的压力测试、安全测试和兼容性测试等非功能性测试工作。	通过脚本或相关工具对于系统局部进行包括压力测试、安全测试和兼容性测试工作，但未形成规律的非功能性测试。

中	高
用例管理以线上集中平台协作为主，用例评审可以线上展开，但用例管理与用例执行未形成关联和互动。	用例管理以线上集中平台协作为主，支线上用例评审，用例管理与用例执行可联动。
有初步业务风险覆盖率意识，能对项目业务风险进行初步评判并以此指导用例设计。	熟练业务风险覆盖率评估思路，能够持续以此为指导优化测试用例设计。
逐步形成相对常规的接口与集成测试自动化实践，支持单接口和场景化接口与集成测试自动化。	接口测试自动化占比超过 50%，且保持持续的优化。接口自身的设计、实现和变更与测试自动化形成良好的互动。
UI 测试成为版本发布日常测试工作中的一部分，且以自动化执行为主。	UI 测试常规化运行，且以自动化为主，执行稳定性有较好的保障机制。
测试团队日常工作大幅度依赖基于脚本的手动测试。	测试团队几乎全部依靠基于脚本的手动测试，无常规自动化测试实践。
有集中的测试数据管理平台，平台数据进行合理的版本管理，有较为丰富的数据生成和生产数据脱敏机制，数据质量稳定。	测试数据集中管理，数据版本和质量可靠，测试数据与测试用例和自动化测试场景形成良好关联和互动。
有集中的服务虚拟化管理平台，能够支持常见的服务虚拟化需求，能在测试自动化中使用。	有集中服务虚拟化管理管理，支持丰富的服务虚拟化需求，服务虚拟化可以与测试数据、测试用例及自动化测试形成良好关联和互动。
具备相对完整的不同测试环境，部署过程基本实现自动化，测试环境一致性能力基本形成。	有完备的测试环境管理机制且能高效一致地部署，不同测试环境的部署可以融入到不同测试需求中按需实时建设。
自动化测试已经嵌入到研发团队的每日构建或者提交构建，测试运行时间合理，且运行结果文档，能得到研发团队认可。	无论是测试左移还是测试右移实践都已经嵌入到 CI/CD 流水线。在保障整个流水线平滑运行的同时，能够有很好地测试自动化运行质量，研发及运维团队高度依赖测试团队的质量反馈。
在软件研发各个阶段都有意设计相应的探索式测试环境，并有明确的测试计划和流程。	探索式测试成为软件研发各个阶段中的必要阶段，并且形成广泛的参与度和高效的测试反馈机制。
有完整的规划、设计和执行，测试自动化执行程度高。针对非功能性测试的结果能形成有效反馈和改进计划。	非功能性测试成为系统性能、容量、安全和兼容性反馈的主要手段。且为此建立了广泛地工具和平台，测试自动化、常规化运行，能够支持生产流量乃至生产环境的测试。

4.3 如何提升成熟度

对于不同企业和团队来说，都可以参照以上成熟模型来做自我评估。但是不同企业和团队在提升成熟度拓展的路径可能会因为实际情况不同而有所不同。这里推荐一种典型的提升成熟度的路径（以企业和团队当前处于 Level 1 为例）供大家参考。具体如图 8 所示：

图8 持续测试成熟度模型的提升路径

图 8 整体描述了一种持续测试程度模型的提升路径，每个阶段的工作内容以及评判标准。关于每一步骤的详细描述如下：

1. 发展起步：重点提升测试执行的自动化程度。尤其是要将自动化测试的主要努力转向接口自动化领域。这一提升过程可以选择部分项目或者部分团队开始尝试。但是，这个尝试不同于之前的大部分自动化测试尝试，它一定要求自动化测试用例占比到一个较高的比例，让选择的项目或团队能形成真正依赖自动化测试的运营习惯。在达成这个运营方式过程中会让企业对自动化测试有更加深入的理解，各种原来未曾遇到的问题和挑战会逐步暴露出来，并最终形成合适的解决方案。与此同时，团队可以初步尝试将部分测试活动合适地嵌入到持续交付流水线中，并在局部尝试测试环境管理的改善，以及采纳高级测试实践；

2. 关键提升：重点在于落实测试用例设计与管理，以及改进自动化测试稳定性上。一方面，企业和团队需要回顾分析大量已经存在的测试用例，判断是不是有大量无效的测试用例在浪费维护和运行的时间。这种有效性分析可以从已经实施自动化测试的项目和团队开始，逐步扩散到其他项目。通过这个阶段的工作来形成测试团队对于有效测试用例的认知，并逐步构建出测试用例有效性的评审机制。测试团队还需要基于有效性分析的结果对存量测试用例进行改造，而这一过程也是企业实现测试用例自动化改造的契机。另一方面，企业和团队需要构建测试用例的线上管理和协作能力，让测试团队的日常工作主要在线上完成。与此同时，保障自动化测试稳定运行也是这一方面的工作

关键点。尤其是在测试数据管理、服务虚拟化和测试环境部署保障上。这一阶段努力成功的重要标识是，自动化测试结果“误报率”大幅度下降，集成和回归测试通过率大幅度提升，且能够给产品提供有效的质量反馈；

3. 成熟落地：全面提升整个持续测试运行效率，建立测试及时反馈机制。将自动化测试集成到项目日常的 CI/CD 流水线中，并根据不同运行时间点选择合适的测试用例集去运行。这一阶段的工作让研发团队能够及时获得测试反馈，改进产品质量。同时，持续高频率执行的自动化测试结果可以让整个团队清晰地判断软件产品的业务风险走势，增强业务部门对于软件发布的信心。另外，建立稳定的高级测试实践运行体系也是这一阶段的重要职责。引入如探索式测试、压力测试、安全测试、兼容性测试等高级测试手段，将测试覆盖的风险从功能层面向更深入层面推进。

5. 参考资料

[1] Wolfgang Platz, Cynthia Dunlop,《 Enterprise Continuous Testing：Transforming Testing for Agile and DevOps》, https://www.tricentis.com/enterprise-continuous-testing-book/, 2019

[2] 朱少民，《软件测试方法和技术（第三版）》，清华大学出版社，2014

[3] 朱少民，《全程软件测试（第三版）》，人民邮电出版社，2019

[4] 乔梁，《持续交付 2.0 业务引领的 DevOps 精要》, 人民邮电出版社，2019

[5] 陈冬严等，《精通自动化测试框架设计》，人民邮电出版社，2016

[6] Gartner,《Take a‘Shift Left’Approach to Testing to Accelerate and Improve Application Development》, https://www.gartner.com/en/documents/3953675/take-a-shift-left-approach-to-testing-to-accelerate-and- , 2019

[7] Gartner,《China Summary Translation: 'Magic Quadrant for Software Test Automation'》, https://www.gartner.com/en/documents/3880027/china-summary-translation-magic-quadrant-for-software-te, 2019

MeterSphere

MeterSphere

一站式开源持续测试平台

www.metersphere.io

关于“软件质量报道”公众号：

“软件质量报道”公众号致力于创建健康、安全、绿色的软件生态，分享软件质量管理、软件测试的思想、方法、技术与优秀实践，追踪软件质量领域的热点，及时报道软件质量管理的成功案例或质量事故等。

软件质量报道公众号

关于 MeterSphere 开源项目：

MeterSphere 是 FIT2CLOUD 飞致云旗下品牌。作为国内知名的一站式开源持续测试平台（github.com/metersphere），MeterSphere 致力于帮助企业的开发和测试团队充分利用云的弹性进行高度可扩展的自动化测试，加速高质量软件的交付进程，最终提升国内软件测试领域的整体运营效率。自 2020 年 6 月发布至今，MeterSphere 开源项目得到了开源社区的广泛认可和积极反馈，并已经在众多企业内落地使用。截至 2020 年底，MeterSphere 项目在代码托管平台 GitHub 上获得了超过 3000 个 Star 和 20000 次以上的独立下载。

MeterSphere 公众号