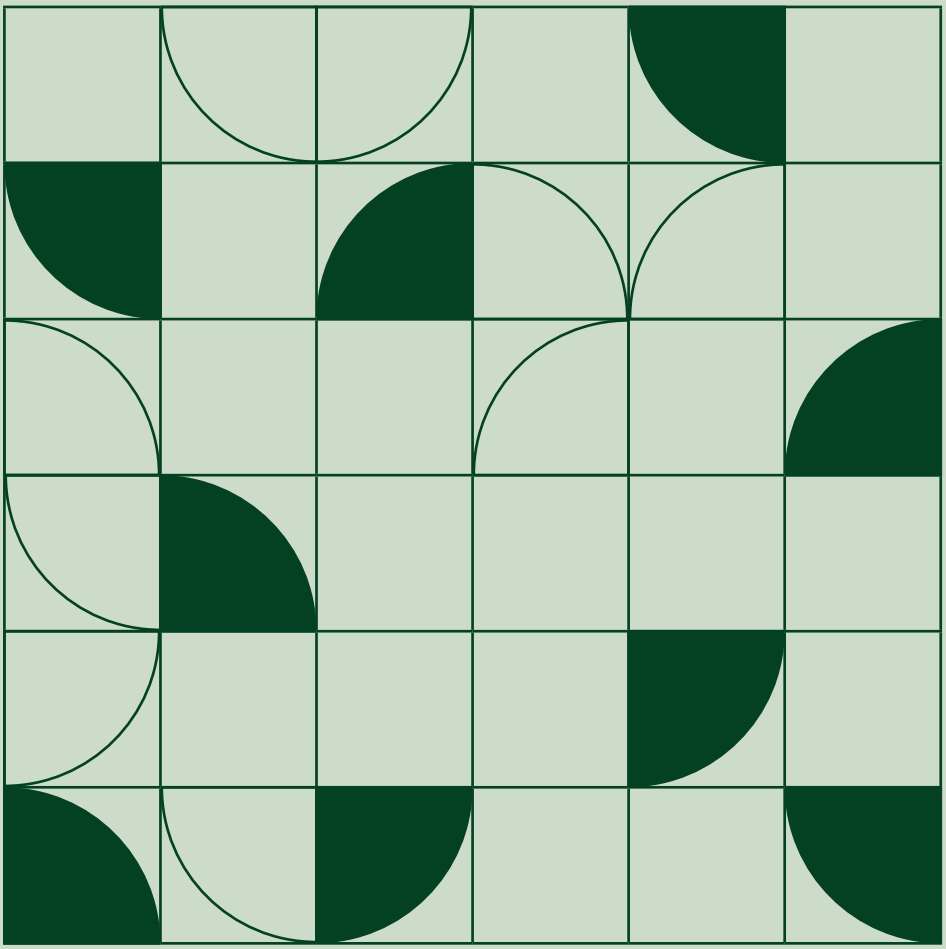
软件部署



简介

由于设备日益复杂化，测试工程师需要创建更复杂、更高级的混合测试系统，而且通常面临着时间期限紧迫和预算缩减等挑战。构建测试系统的一个重要步骤是将测试系统软件部署到目标机器上。这也是最繁琐、最令人沮丧的一个步骤。对于那些只是单纯寻求最便宜和最快速解决方案的工程师，市场上琳琅满目的部署方法往往会增加他们的选择难度。此外，测试系统开发人员也面临着许多其系统特有的考量因素和敏感问题。

在本指南中，部署定义为编译一系列软件组件并将这些组件从开发计算机导入到目标机器上执行的过程。测试工程师之所以采用部署方法，而不是直接在开发环境中运行测试系统软件，主要是出于成本、性能、可移植性和保护方面的考虑。

以下是测试工程师需要从开发环境执行转换到定制的二进制部署的一些常见拐点：

- 每个测试系统的应用软件开发许可证成本开始超过预算限制。为每个系统使用部署许可证是更具吸引力、更高效的解决方案。
- 由于内存限制或依赖关系等原因，测试系统的源代码变得难以移植。
- 测试系统开发人员并不希望最终用户能够编辑或者查看系统的源代码。
- 测试系统在开发环境中运行时，会遇到执行速度缓慢或内存管理问题。编译执行代码可提高性能并减少内存消耗。

本指南提供并比较了不同的考量因素和工具，以解决测试系统部署过程中的疑难和困惑。尽管本指南可以介绍有关测试系统部署的许多不同主题，比如源代码控制最佳实践或安装程序的创建，但所选主题应涵盖大多数通用部署问题。

每部分的末尾均提供了基本用例和高级用例的最佳实践建议：

- 基本用例是由一个可执行文件构成的简单测试系统，会按顺序运行测试步骤并调用少量硬件驱动程序。此类系统包含的测试功能通常少于 200 种。
- 在每个基本用例最佳实践的末尾，都会说明出现何种迹象或指示时应考虑高级用例。

高级用例表示一个大规模生产测试系统，该系统将可执行文件、模块、驱动程序、Web服务或第三方应用相结合，用于执行混有大量不同测试序列的测试。此类系统通常包含数百甚至数千种测试功能。

管理和确定系统组件

定义组件

在软件开发中，组件是系统中使用的任何信息实体，比如二进制可执行文件、数据库表、文档、库或驱动程序。要成功完成部署，必须先确定与测试系统相关的组件并确保每个组件都采用适当的部署方法。此步骤在不同场景下的复杂性可能有很大差异。例如，简单测试系统的组件可能是单个可执行文件和必要的硬件驱动程序。

复杂系统组件

但在复杂的测试系统中，这些组件通常是XML配置文件、数据库表、自述文本文件或Web服务。随着系统复杂性的提高，需要引入更高级的部署选项。例如，可能需要频繁更新配置文件，以便根据天气的季节性变化校准获取的数据，而主要的可执行文件几乎无需更新。没有必要在每次出现更新需求时都将可执行文件与配置文件一起重新部署，因此，配置文件可单独采用与可执行文件不同的部署方法。

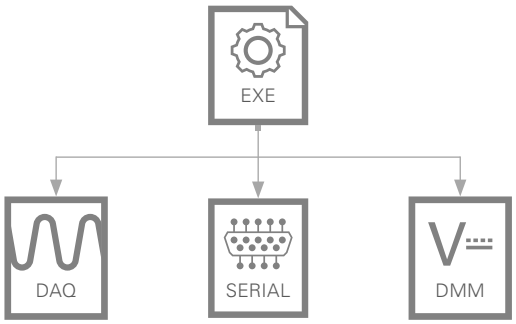


图1 基于DAQ、串口和DMM驱动程序的简单测试系统可执行文件示例

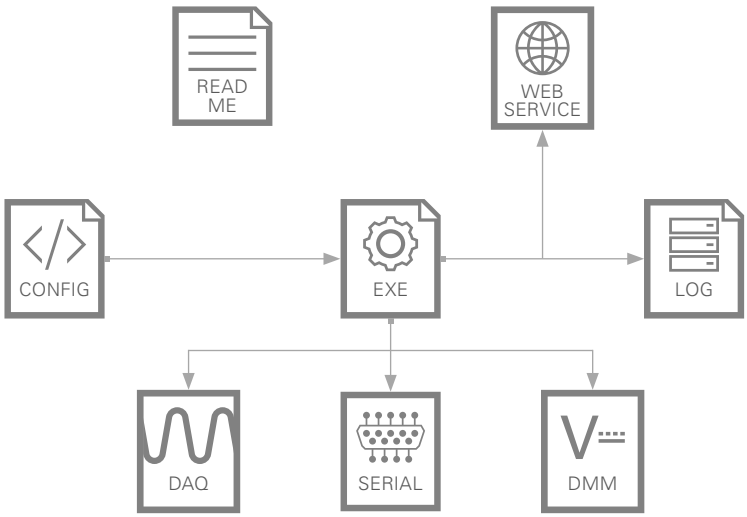


图2 具有复杂依赖关系的测试系统示例

除了确定每个系统组件并规划其部署方法外，确定系统组件之间的关系并确保部署方法不会中断这些关系也十分重要。在需要频繁更新配置文件的示例中，工程师可能需要将配置文件安装到每个部署系统上的相同位置，以便可执行文件可以在运行时找到配置文件。

依赖关系跟踪

维护各个依赖项之间的关系涉及建立依赖关系跟踪方案，此举可确保每个组件的依赖组件均得以部署。尽管在手动确定每个系统组件后依赖关系看似明显，但依赖关系通常会深度嵌套，并需要随着系统扩展而自动识别。例如，系统B中的可执行文件可能依赖于.dll才能正确执行，但创建部署计划的工程师却忘记将.dll文件标识为必要组件或没有意识到该依赖关系。在这些情况下，可使用编译工具自动识别已生成应用的大部分或全部依赖关系。

下面举例介绍了一些编译软件应用：

- **LabVIEW Application Builder**—识别特定上层VI集的依赖项(子VI)，并将这些子VI包含在编译的应用程序中
- **TestStand Deployment Utility (TSDU)**—以TestStand工作区文件或路径作为输入，确定系统的依赖项代码模块；自动编译这些模块并将其包含在编译的安装程序中
- **ClickOnce**—Microsoft的一项技术，可帮助开发人员为其.NET应用轻松创建安装程序、应用程序甚至是Web服务；可配置为在安装程序中包含依赖项，或提示用户在部署后安装依赖项
- **JarAnalyzer**—用于管理Java应用程序间依赖关系的实用程序，可遍历整个目录、解析目录中的每个jar文件并确定各文件之间的依赖关系

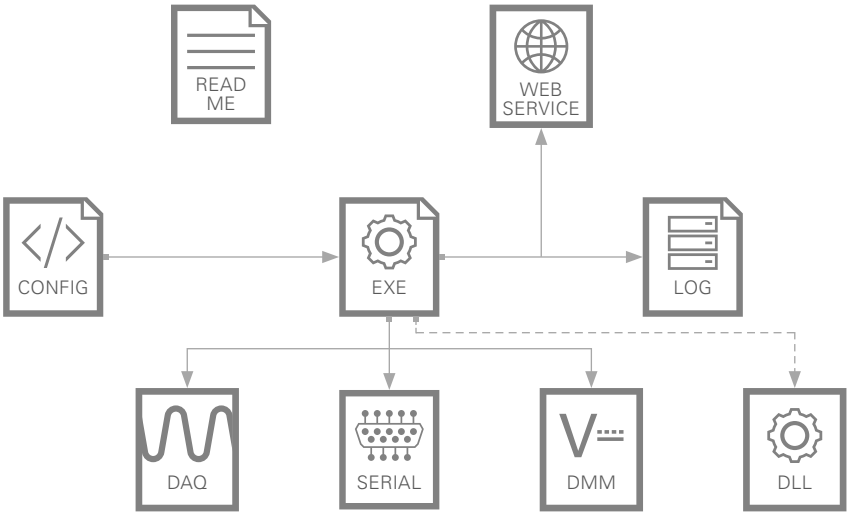


图 3 复杂测试系统中出现意外的依赖关系

关系管理

通常来说，不仅主测试程序可执行文件与其相关组件之间存在关系，各个单独的组件之间也存在关系。因此，需要考虑不同组件或软件模块之间关系的本质。随着系统不断扩展，解析不同库、驱动程序或文件之间的依赖关系可能变得非常复杂。例如，一个测试系统可能使用三个不同的代码库，三者之间的关系如下图所示。

对于这些复杂的系统，通常需要使用依赖关系解析器来识别依赖项冲突并管理无法解决的问题。尽管可以在内部编写依赖关系解析器，但工程师经常使用程序包管理器来管理依赖关系。例如，NuGet就是专为.NET框架程序包设计的免费开源程序包管理器。另一个例子是用于LabVIEW软件的VI Package Manager，通过该管理器，用户能够分发代码库并通过API提供自定义代码库管理工具。

最佳实践

基本用例：对于基本或简单的系统，通常可以手动跟踪所有必要的组件。使用软件应用程序或程序包管理器来管理依赖关系可能并无必要，且这种做法的前期安装成本过高。然而，如果出现警告提示，比如不断遇到依赖项缺失问题或依赖项不断增多，通常表示需要采用更高级的依赖关系管理方案。

高级用例：采用可扩展的依赖关系管理系统时，复杂的系统更容易维护和升级。这可能意味着需要使用程序包管理器来诊断程序包之间的关系，或使用软件应用程序来了解和识别各组件的依赖关系，无论是何种情况，维护此类系统对于系统的长期正常运行都至关重要。

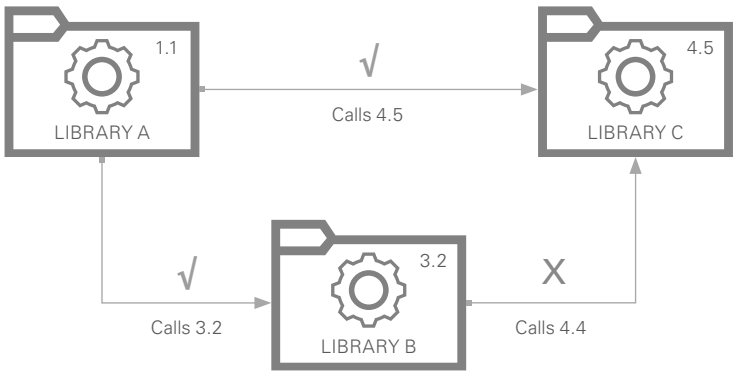


图4 由于库A依赖于库C版本4.5，因此库B对库C版本4.4的依赖会导致无法解决的依赖关系问题

硬件检测

硬件断言

有特定硬件设置需求的测试系统需要确定该硬件是否存在于系统中，并在该硬件不存在或不兼容部署计划的情况下执行应急计划。尽管开发人员经常通过检查测试机外观并将硬件组件与原始开发系统匹配来完成硬件断言，但更好的做法是假设测试系统由第三方所创建。测试系统的客户如何确定是否有不兼容的硬件？对于将正确模块插入不正确的插槽或端口的情况，系统能否做出应对？系统能否解决或调整硬件缺失的问题？尽早解决这些问题有助于简化测试系统的扩展和分发。

硬件标准化

硬件断言的最终目标是确定预期系统与实际物理硬件系统之间并无差异。

为此，最高效的做法通常是先在每个测试系统上对将要使用的硬件组件集合进行标准化：

- **文档记录**—标准硬件集合中的组件列表应该可供所有新系统访问。文档中必须包含供应商、产品编号、订单号、组件数量、可更换组件、保修、支持政策、产品生命周期等信息。
- **可维护**—硬件标准化最困难的问题之一是确保每个测试系统中使用的硬件组件将来仍然可用。通常，较旧的硬件会被制造商标记为生命周期结束(EOL)，需要刷新测试系统的标准硬件组件。从硬件升级和测试系统停机的角度来看，这种刷新会产生高昂的成本。与硬件制造商共同探讨硬件组件的生命周期策略有助于缓解日后的挑战。大多数硬件制造商(如NI)都提供生命周期咨询服务，并支持各硬件组件在生命周期内的逐步下线。
- **可复制**—应考虑是否需要全球性甚至区域性地部署硬件。必须确保有成熟的硬件部署方案，以便在偏远位置快速搭建新系统。对于很多系统，保证备用硬件组件的供货渠道以便进行维护或紧急更换也至关重要。

开机自检(POST)

即使测试系统选用合适的硬件并已正确连接，也必须对硬件进行简单测试，以确保系统运行后硬件能够按预期工作。幸运的是，大多数硬件组件都包含制造商预置的自检功能，通过该功能，可对设备的通道、端口和内部电路板进行简单检查。在为每个测试系统供电后，应为所有连接的设备执行自检程序，作为对硬件故障的早期检查。例如，每个NI设备都具有自检功能，可通过设备的驱动程序API以编程方式调用。因此，测试系统供电后的第一步可能是在每台设备上调用自检功能，并警告操作员任何硬件故障。

别名配置

遗憾的是，对硬件集合进行标准化并不能完全确保配置完全相同。通常，需要使用Measurement & Automation Explorer (MAX)等硬件配置软件将硬件设备重新映射到别名。例如，在安装所有硬件组件并为系统供电后，工程师可以使用MAX检测系统中存在的NI硬件，并使用Windows设备管理器查找非NI硬件。随后即可编辑.ini配置文件，将硬件设备正确映射到别名。该过程可能的输出如下图所示。

别名	设备名称
PXI NI-4139	PXI 1插槽1
PXI NI-3245	PXI 1插槽2
PXI NI-2239	PXI 2插槽1

表1 使用hw_config.ini文件将物理硬件映射到测试系统别名

编程配置

LabVIEW软件中用于NI硬件的System Configuration API等库可通过编程方式生成所有可用在线硬件的列表并配置别名映射。例如，测试系统可执行文件可以调用System Configuration API的Find Hardware函数来生成可用NI硬件列表。在该列表中，每个设备的别名属性均可通过硬件节点设置为预定义的名称。此方法可能导致系统出现问题，如导致硬件设备映射到不当的别名。因此，工程师应将此方法与其他保护措施结合使用，比如手动确认映射列表或标准化硬件集合。

最佳实践

基本用例：对于基本或简单的系统，务必要确保系统中存在预期的硬件。硬件标准化是适用于所有系统的最佳实践，随着硬件系统数量不断增加，这种做法尤为重要。应记录正确执行测试系统所需的机箱、模块和外设，并定期对文档进行修订更新。但通常只需要借助MAX等工具进行手动检查即可验证系统上运行的设备是否正确，无需采用编程或重新配置的方法。而随着硬件系统中的模块和设备数量不断增加，可能就需要换用更高级的解决方案，以防出现硬件缺失的问题。

高级用例：在复杂系统中，应综合使用不同的解决方案来跟踪系统中必需的硬件或系统中存在的硬件。与基本用例的做法一样，应跨系统进行硬件标准化和文档记录。如需检测故障硬件，应执行开机自检(POST)以确保连接的硬件按预期运行。此外，当硬件标准化失败时，应使用编程方式或手动操作别名映射系统，将预期设备自动重新映射到系统的别名。

依赖关系解析

依赖关系断言

一种切实可行的做法是制定计划来处理已部署系统上现有的依赖项和缺失的依赖项。通常，所部署的测试机器上已经安装了测试系统镜像的部分依赖项。对于规模较小的系统，只需重新安装所有依赖项便可确保万无一失。但是，对于规模较大的系统，重新安装所有依赖项十分耗时，建议先检查系统上是否存在相应依赖项。这种做法被称为依赖关系断言，有助于缩短部署时间，但需要针对依赖项差异进行规划。“组件化”部分将进一步讨论如何通过组件化加快部署。

例如，测试系统可能与14.0和15.0版本的NI-DAQmx驱动程序均兼容。测试系统可能会要求安装NI-DAQmx 15.0，但允许14.0版本充当依赖项。然而，用14.0版本代替15.0版本后，虽然不会影响兼容性，却可能会改变测试系统的行为。系统可能会跳过某些测试步骤或调用不同的函数。需要记录所有这些变化并进行测试。

依赖关系断言的第二个要素是确定如何处理缺失的依赖项。如前文所述，较好的做法是假定将测试系统部署到客户的机器上。是否应通知负责部署的工程师缺少依赖项？缺少的依赖项应该在后台静默安装，还是需要用户手动查找并安装？尽早解决这些问题有助于加快部署速度并正确处理缺失的依赖项。

最佳实践：

基本用例：对于基本的测试系统，通常没有必要进行依赖关系解析和断言。安装所有测试系统的依赖项，而不考虑其是否存在于系统中，这种做法通常比尝试确定缺失的依赖项并仅安装缺失的元素更加简单。但随着测试系统的扩展，系统总体安装时间可能会增加，达到一定规模时，开发依赖关系断言和解析工具将成为更有效的解决方案。

高级用例：即使网络连接稳定或镜像已进行压缩处理，部署时间也可能快速增加到不合理的程度。对于大部分高级测试系统，为了避免重新安装所有组件，有必要进行一定程度的依赖关系断言。在部署过程中，可以加入用于查找系统中安装的NI软件的System Configuration API或用于生成Windows机器上所有程序列表的WMIC命令集等工具。这样，安装程序便可跳过特定组件或允许存在版本差异。

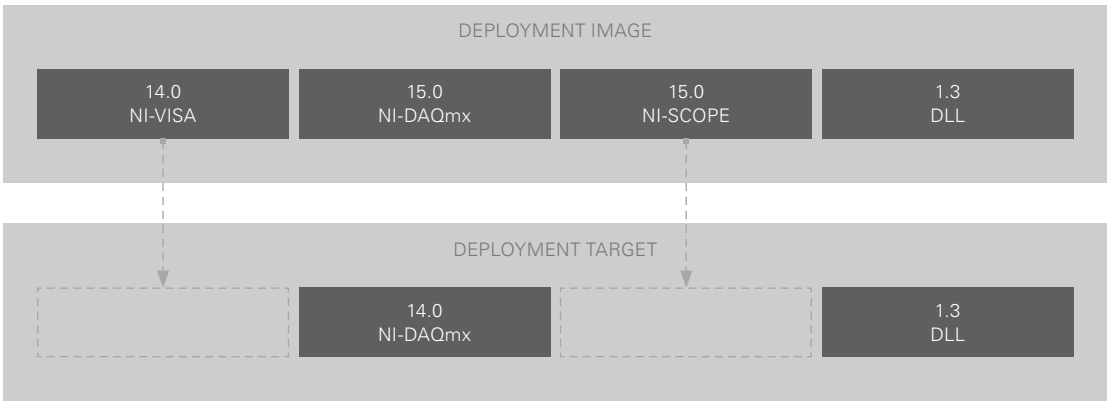


图 5 依赖关系断言

版本管理

通常，工程师需要了解当前部署在测试系统中的软件镜像版本，或者能够提供版本部署历史记录。

如有此类必要需求，则应配备版本管理系统来解决以下各项问题：

- 当前部署到系统A的是哪一版本？
- 系统B的最新部署状态是什么？
- 版本1、2和3部署到哪一系统？
- 系统A的版本历史是什么？

在大多数测试环境中，工程师需要使用铅笔和剪贴板系统来手动记录这些信息，但是，也有一些工具可以自动记录版本信息并提供特定系统的版本历史记录。这些版本管理工具可以集成到集成开发环境(IDE)中，也可以作为独立的版本管理工具存在。

示例工具：

- **Visual Studio Release Management**—Visual Studio IDE附带了这款支持自动化部署、版本历史跟踪和版本安全性管理的工具。
- **Jenkins Release Plugin**—利用这款用于Jenkins持续集成(CI)服务的插件，开发人员可以指定编译前和编译后的操作，以管理集成了Jenkins服务的开发版本。
- **XL Deploy**—此应用版本自动化(ARA)软件可以扩展到企业级别，并提供可视化状态仪表板、安全功能和分析功能来管理版本。

尽管上述示例工具可作为IDE和独立部署解决方案，但更常见的做法是将版本管理工具与CI服务器和端到端部署过程相结合。这种方法非常直观，因为对于部署过程来说，更常见的问题是某个机器上存在哪些特定代码，而不是使用哪个版本。对于已编译的系统镜像，这一点可能很难通过手动观察来确定。要实现版本管理最佳实践，有必要跟踪从开发到部署所用的代码。

端到端系统自动化

要为测试系统部署开发更为复杂的端到端流程，高效的版本管理必不可少。从开发到部署，这一过程中的每一步都与先前的步骤紧密相关；如果源代码管理良好，测试和编译也可以得到有效管理。有了良好的测试和编译流程，版本管理就只是原系统的简单扩展。下图所示为典型的端到端系统。

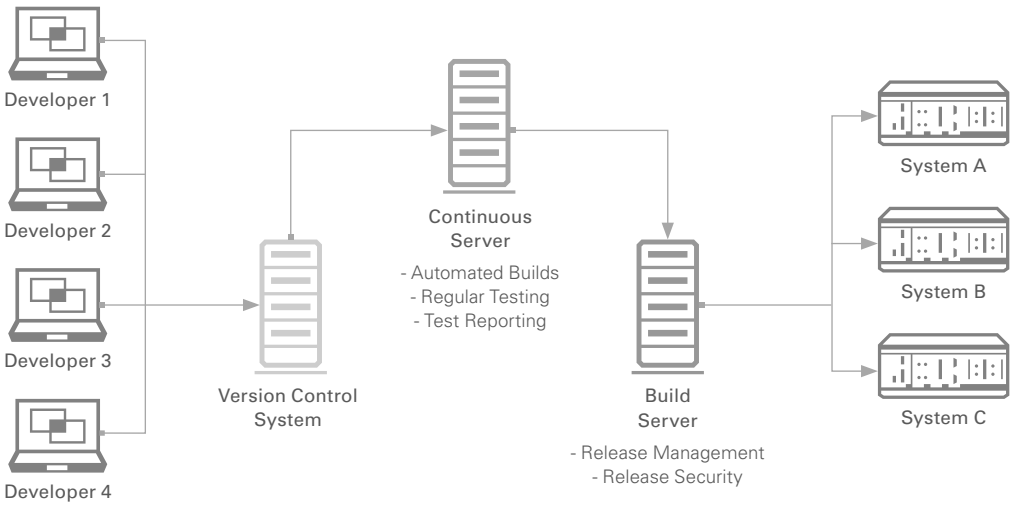


图 6 开发人员将代码提交到版本控制存储库，然后可以在CI服务器中进行编译和测试。在版本控制存储库中，生成版本可存储在编译服务器中并进行版本管理

在此架构中，测试系统开发人员会定期开发源代码并将其提交到版本控制存储库。随后，CI服务可将源代码从版本控制存储库提取到自己的存储库，并适当地编译和测试代码。此时，开发人员可通过自动或手动方式将通过CI测试的代码移动和存储到编译服务器或存储库。编译服务器通过报告和跟踪功能将每个软件版本链接到特定的测试机器，实现版本管理。通常，测试机器通过请求安装测试系统的特定版本来启动部署过程；但是，开发人员也可以配置编译服务器以将镜像推送到选定的机器上。

如果基本系统也需要采用一定级别的版本管理，则应根据版本需求的内在复杂性选择最实用的解决方案。如果需求是跟踪部署到系统的版本，则通过配置文件或作为编译可执行文件的组件进行手动版本管理即可。如果需求范围扩大、测试系统数量增加或应用程序版本数量增加，则有必要使用定义的版本管理系统。

最佳实践：

高级用例：对于需要进行版本管理的复杂测试系统，采用特定形式的端到端自动化方案通常效果最佳。最简单的方式是选择Jenkins或Bamboo等CI服务，这些服务可将版本管理与版本测试和源代码控制相结合。

版本测试

回归测试

在软件工程中，回归测试是指在先前开发的系统发生更改后所进行的测试。回归测试的目的是保持每个版本的完整性，并跟踪系统在特定更新或补丁处的错误。对于组件化系统，回归测试对于确定模块A的升级是否会导致模块B中出现意外行为尤为重要。例如，升级系统中的NI-DAQmx硬件驱动程序可能会导致硬件抽象库出现问题，该库调用的较旧NI-DAQmx版本中的函数现已在较新版本中弃用。回归测试有两种类型：功能测试和单元测试。

功能测试

在测试系统中，关于软件更新的最重要的问题是，这些更改是否会破坏系统的功能？系统是否仍然按照预期的方式运行？功能测试会验证系统在采用一组已知输入时是否会生成预期输出，可以帮助用户解答有关整套系统的这些问题。这种类型的测试通常采用“黑盒”方法；不分析系统的内部机制，只分析系统的输出是否符合预期。对于测试系统，此测试可验证更新硬件配置、更改驱动程序或添加测试步骤是否不会改变原有的测试功能。工程师可在使用可模拟待测设备(DUT)的测试系统上执行功能测试，验证这些系统经校准后是否通过特定测试。例如，用于测试对象是否为圆形的系统由四个组件组成：相机控制器、周长传感器、直径传感器和体积传感器。如果系统从版本1.0更新为版本1.1并对直径传感器进行了更改，则下图中的第二个待测圆形在更新前可通过圆形测试仪的测试，更新后则无法通过测试。

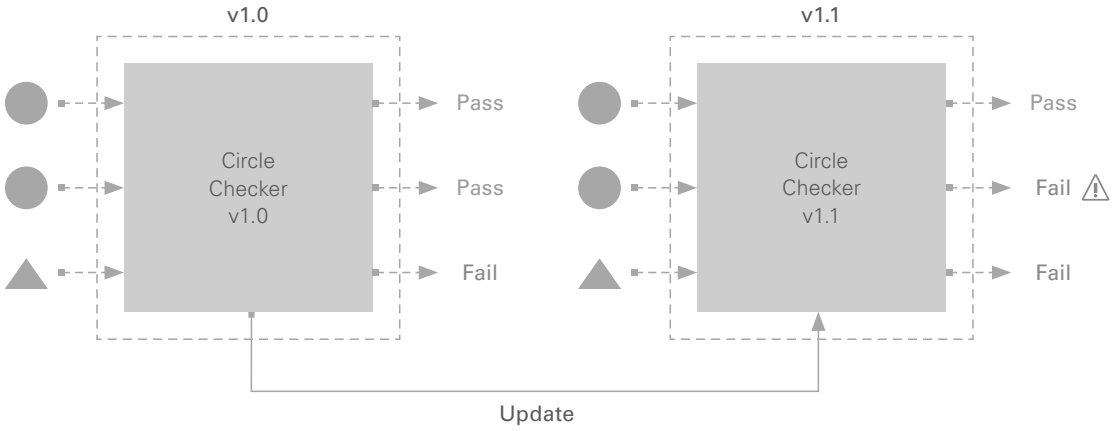


图 7 对测试系统中的模块进行少量更新可能会导致功能测试出现故障，这是一种伪故障

单元测试

功能测试针对的是整个系统，而单元测试则是针对特定模块、组件或功能。此类测试旨在跟踪测试系统特定部分的质量，而不仅仅是正确性。例如，如果测试结果记录到数据库中，则可以在数据库控制器上进行单元测试以测量数据吞吐量。通过这种方式，不仅可以分析对数据库控制器进行的更改以确定日志记录功能是否可正常运行，还可以了解软件更改是加快还是减慢了系统的日志记录能力。除了有助于查找错误之外，单元测试还可以将发现的性能升降情况与特定更改相关联。之前提到的圆形测试仪示例可以解释单元测试和功能测试之间的区别。假设圆形测试仪的直径传感器软件组件像之前一样升级，则可以只对直径传感器进行单元测试，而不是对整个系统进行功能测试。对于单元测试，可以为特定组件提供表示具有特定直径的圆形的二进制图像数据，并测试输出是否与该已知直径匹配。这种方法可验证模块的正确性并进行定量测量，例如测量模块的执行时间。

在这种特定情况下，升级明显降低了模块的速度。还可以推断，由于升级后系统未通过功能测试但通过单元测试，相机控制器与直径传感器之间的通信很可能存在软件错误。这种验证系统正确性和个体模块功能的能力可确保仅将高质量的版本部署到测试机器上。

测试过程

在大多数测试系统中，为节省开发时间，回归测试与源代码控制、编译或版本管理同时进行。这样便可重复使用更新相对不频繁的测试代码。但是，针对测试代码的编译开发时间制定计划和预算也很重要。通常，回归测试是CI服务或IDE的组成部分，其中源代码控制、编译和测试都会按顺序进行。

最佳实践：

基本用例：对于所有测试系统，在将系统部署到新机器之前，都应该进行一定程度的功能测试。这种功能测试既包括使用模拟硬件在开发环境中手动运行应用程序的简单情况，也涉及基于配置文件运行一系列功能测试等稍显复杂的场景。对于比较简单、相对单一的应用，单元测试可能并无必要，但随着测试系统复杂性的提高，就需要考虑进行单元测试。随着添加的模块越来越多，有必要通过特定的自定义测试来跟踪漏洞或确保系统满足某些规范。

高级用例：对于复杂的测试系统，不应仅对测试系统每个新版本的各种输入进行功能测试，还应针对系统的每个单独模块开发单元测试。两种回归测试方法都应该在部署过程中最有效的时间点进行。例如，可以在编译每个版本后执行功能测试，在每个源代码控制提交点执行单元测试，这样便很好地混用了这两种回归测试。通常，这些测试对于系统来说是强制或默认执行的，尤其是对于航空航天和国防工业领域的系统。

图 8 对v1.0和v1.1执行单元测试后，处理时间被识别为更新后的一项问题，导致过程功能测试出现伪故障

组件化

由于部署时间是大型测试系统的常见考量因素，所以最佳做法是只更新测试系统中需要更改的单个组件，而不是重新编译整个系统。本指南的“依赖关系解析”部分介绍了该主题的部分内容，但仍需要单独探讨如何开发模块化架构或基于插件的架构以提高部署效率。无论工程师选择何种架构，最佳做法都是定期更新外设模块，而更核心的模块在开发时保持相对恒定，无需重新编译。这种做法自然会导致有关更新频率的问题，本节稍后将对此进行探讨。

部署插件架构

软件部署中使用的插件是一个代码模块，在安装上独立于主应用程序，在功能上独立于其他插件，遵守全局插件接口规定，用于编译的应用程序中时可避免名称冲突。主应用程序随后应能够动态加载每个插件，通过标准接口调用每个插件，并将每个插件用作扩展，而不需要重新编译。成功开发后，插件框架可对部署进行组件化处理，即仅更新或安装特定或缺失的插件，而无需重新编译主应用程序或任何未受影响的插件。

例如，为简单应用程序开发的插件框架可能包含一个主要的可执行文件，该可执行文件会在加载时搜索插件目录或在运行时定期搜索插件目录，并通过标准接口执行该插件。这样，无需编辑主应用程序即可将插件持续部署到系统的插件目录中。

硬盘驱动器复制

通常，代码库、硬件驱动程序或特定文件是测试系统核心的组成部分，不需要像其他模块化外设组件一样频繁更新。在这类情况下，可通过硬盘驱动器复制对环境进行标准化处理，以作为进一步开发的基准。工程师可以将开发机器或归零测试机器的硬件驱动器复制并克隆到其他测试机器上。驱动器复制完成后，测试机器将拥有共同的起点，通常包括主测试应用程序或程序、必要的硬件驱动程序、系统驱动程序集和关键的外设应用程序，例如用于硬件配置的MAX。但必须意识到，硬盘驱动器复制有其自身的限制，比如需要在各测试机器之间使用相同的计算机硬件或分发占用大量内存的镜像，因此硬盘复制不适用于软件频繁更新的情况。

使用硬盘驱动器复制为进一步的测试开发奠定基础的一个示例是将常用的硬盘驱动器复制工具Symantec Ghost与TestStand Deployment Utility (TSDU)结合使用。在下图(A)中的第一个框中，开发机器将其核心软件堆栈(红色)复制到目标机器上。该核心软件堆栈是Windows操作系统、硬件设备驱动程序、运行引擎和MAX的组合。目标机器生成镜像后，便可在开发机器上进行开发(B)，使用TestStand和LabVIEW(绿色)创建测试序列。开发人员随后可以使用TSDU将测试序列移动到目标机器。如果需要频繁对测试序列进行更新，开发人员可以继续使用TSDU以节省开发时间，因为核心软件堆栈不需要更改。有时，开发可能是在未部署到目标机器(C)的开发机器上进行。这种系统不匹配可能会导致依赖项缺失问题。在这种情况下，开发人员可以不使用TSDU来更新目标机器，而是选择重新为开发机器生成镜像并将其复制到目标机器上，以同步这两台机器(D)。之后，开发人员可以继续使用TSDU进行频繁更新，如果将来出现系统不匹配问题，可以使用Ghost重新为目标机器的硬盘驱动器创建镜像。

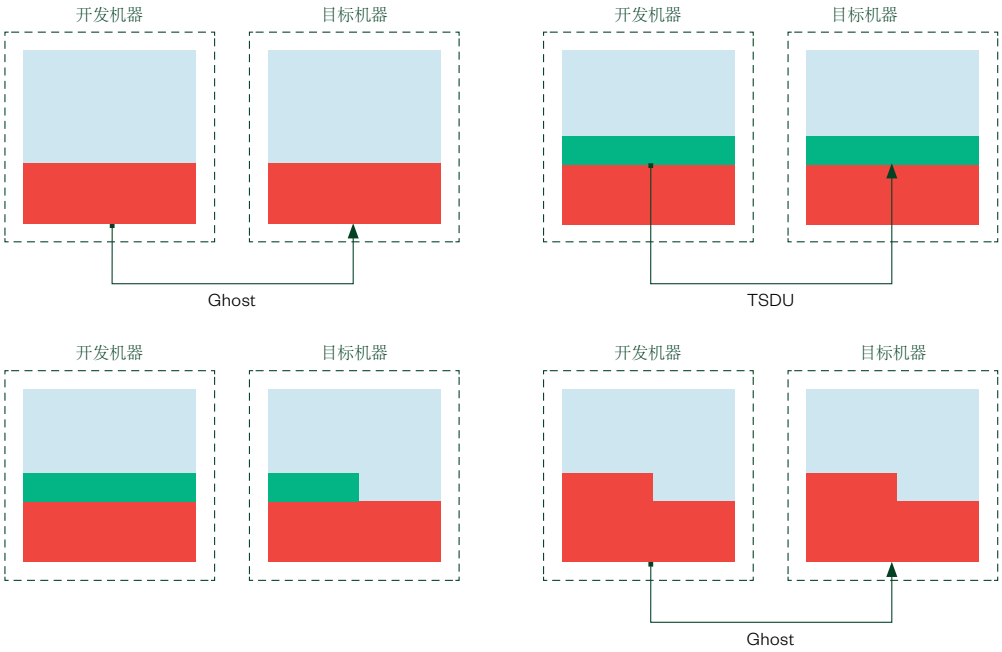


图 9 TSDU和硬盘驱动器复制示例

持续集成和持续部署

持续集成(CI)是指通常在单独的CI服务器上持续提交、编译和测试代码的做法。大多数测试系统会使用CI服务提供编译、测试和部署系统软件所必需的框架。这些服务在CI服务器上定期自动运行，具有多种配置选项，可用于创建编译计划、自动化测试规则和版本部署等。使用CI服务器最明显的优势之一是能够跟踪和管理不同的编译和部署。

编译

版本	状态	上次编译时间
1.2	未通过	2016年5月24日
1.1	通过	2016年4月2日
1.0	通过	2015年12月7日

部署

版本	状态	上次编译时间	机器
1.0	通过	2015年6月7日	A
1.1	未通过	2015年6月9日	B
1.1	通过	2016年3月16日	C

表 2 CI服务提供仪表板来跟踪应用程序的编译和部署

CI工具涵盖多种功能，开源开发人员和软件公司都积极开发CI工具。后者还具有为系统设置提供支持的额外优势。

- **Jenkins**—Jenkins被誉为“领先的开源自动化服务器”，因其安装和配置的便捷性而成为目前最受欢迎的CI服务之一。Jenkins几乎可与所有编程语言一起使用，因为它可以通过命令行接口或丰富的Jenkins插件与程序连接。
- **Bamboo**—Bamboo是软件公司Atlassian开发的一项领先的专有CI服务。除了Bamboo的测试、编译和集成功能之外，Atlassian还可提供优于Jenkins的“一流的部署支持”。
- **Travis CI和Circle CI**—这两项开源CI服务可提供强大的扩展能力，但只与位于GitHub存储库中的项目集成。

总体而言，CI的目标是提供自动化且可配置的工具，使开发人员能够在编译和测试软件的同时继续编码。

最佳实践：

基本用例：简单的系统通常无需过度关注组件化。尽管系统使用很少的代码模块或不采用插件架构，但每个测试系统通常可以部署为独立的应用。但是，如果安装时间变得非常长，并开始导致部署速度变慢，则可能需要转为使用更加组件化的方法，避免重新安装所有组件。

高级用例：当测试系统变得庞大、复杂或使用插件架构时，就有必要从单一部署镜像转为可单独更新每个组件的模块化部署。利用插件架构可快速实现这种模块化设置，但配置CI服务也是一种可行方法。

实际案例

高级部署框架的一个案例是一家音频设备生产公司使用TestStand和LabVIEW对其产品进行功能性电气测试。这家音频设备制造商的测试部门在全球部署了超过50个测试系统。每个系统都使用混有多种模块的PXI机箱，包括数据采集模块、数字I/O模块、数字信号采集模块、数字万用表模块和频率计数器卡模块。

负责部署的测试工程师为每个即将上线的新测试系统执行下述程序。

01

创建基本系统镜像

每个新测试系统都有一个确保系统安全所必需的软件列表，其中包括公司开发的软件和第三方软件。公司的IT部门需要使用这些杀毒软件、VPN安全应用程序和Windows组策略配置规范。其次，每个系统都需要一套基础软件来执行其必要的测试序列。该软件的主要组件是一组对照已发布NI系统驱动程序集进行交叉检查的驱动程序。也就是说，一个版本的测试系统可能包含NI-DMM 14.0、NI-Switch 15.1、NI-FGEN 14.0.1和NI-DAQmx 14.5驱动程序。此外，还需要LabVIEW 2014和TestStand 2014的运行引擎来运行主要的测试系统可执行文件。下表列出了所有必要软件的概况。

要部署到新测试系统，需要先在开发机器上创建此镜像，并使用硬盘驱动器镜像软件复制此镜像。该软件镜像可能是以前创建的，因此可以在多台机器上重复使用。这有助于降低部署成本，因为每批相同的测试机器只需安装一次。

通过安装所有必要的软件生成基本系统镜像后，使用Symantec Ghost复制硬盘驱动器并将新镜像上传到编译服务器。编译服务器位于总部，该服务器只需要维护一个大容量存储器，以便将多个系统镜像保存在服务器上。

02

部署基本镜像

将基本镜像上传到编译服务器后，测试工程师会将新的测试系统连接到公司网络，然后使用Web界面连接镜像服务器并浏览各种可供安装的基本系统镜像。选择适当的版本后，Symantec Ghost会使用复制镜像对新系统硬盘驱动器进行镜像。至此，测试系统便已具备执行测试序列必备的基本软件。

03

验证硬件

在将必要的硬件模块物理安装到PXI机箱并启动系统后，测试工程师需要在软件中将系统别名映射到实际设备。尽管已提供模块列表及相关插槽编号，但测试工程师必须使用配置系统设置来映射别名，以便模块位置可在系统之间切换。对于该公司，每个测试系统都使用工程师编辑的.ini文件以提供实际系统硬件到测试系统别名的映射。此过程通过在MAX中识别设备并手动编辑.ini文件以创建适当的映射来实现。

软件	版本
NI-DAQmx驱动程序	14.5.0
NI-DMM驱动程序	14.0.0
NI-Switch驱动程序	15.1
NI-FGEN驱动程序	14.0.0
LabVIEW运行引擎	2014
TestStand运行引擎	2014
内部杀毒软件	3.2

表3 创建基本系统镜像时，必须明确列出将包含的驱动程序和运行引擎版本

04

安装应用程序和组件

至此，测试系统已安装基本系统镜像并验证了实际硬件。现在，工程师的任务是安装最新版本的测试应用程序。本例中，应用程序是由TSDU生成的TestStand安装程序，其中包含所有必要的代码模块、序列文件和支持文件。为明确该安装程序的生成方式，请务必查看生产公司采用的开发系统。每个开发人员会在LabVIEW中创建特定的测试步骤，或在TestStand中创建测试序列，并将测试步骤或测试序列提交到Apache Subversion源代码控制存储库。该存储库位于运行CI服务(Jenkins)的服务器上。Jenkins服务会对提交的代码模块运行测试，使用TestStand序列分析器通过命令行验证序列，然后使用TSDU命令行接口将必要的测试序列编译到安装程序中。每个安装程序在编译完成后，都会连同其必要的支持文件一起自动通过Jenkins Deploy Plugin部署到编译服务器。

05

执行

将TestStand安装程序部署到编译服务器后，测试工程师即可将安装程序下载到新的测试系统。然后，工程师随后运行该安装程序，找到主要的测试可执行文件，并开始运行基本测试系统。

利用此部署系统，测试工程师可以快速轻松地对每个测试系统进行更改。现有的硬盘镜像系统可用于进行大规模代码修订或驱动程序集升级，而较为轻量级的编译服务器可用于部署对主测试应用程序或单个组件和插件的细微更改。

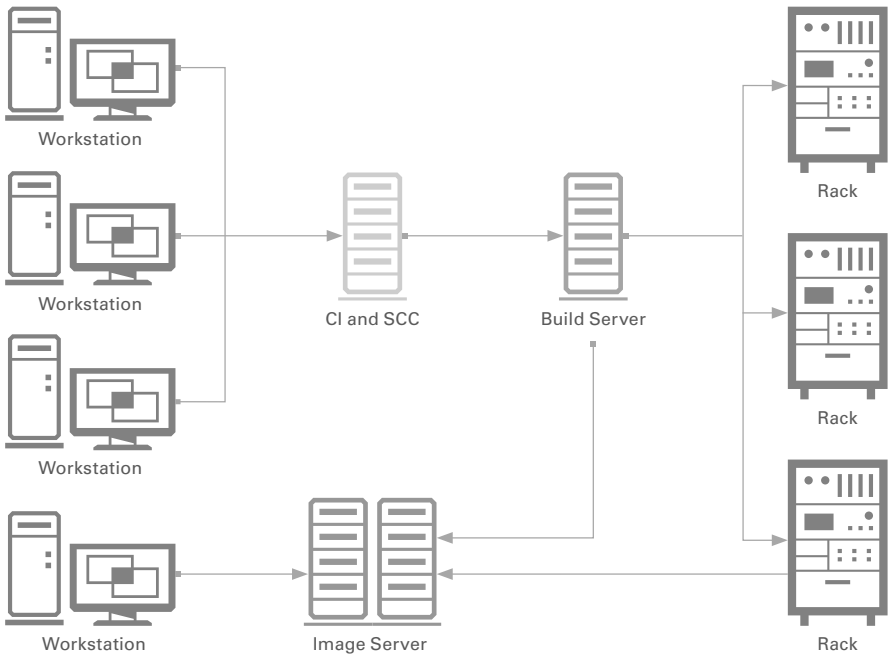


图10 此测试部署系统使用镜像服务器来存储和部署基本系统镜像，从而使各个测试站彼此保持同步。随后，开发人员会定期将源代码上传到持续集成和源代码控制服务器，以定期编译和测试提交的代码。提交的代码通过了所有必要的测试之后，系统便会将编译的镜像添加到编译服务器，由该服务器负责处理测试软件系统镜像的大规模分配。

总结

测试系统部署通常是一个复杂的过程，随着测试系统的升级和数量的扩展，复杂性还将持续提升。在测试系统开发的早期敲定部署过程是实现可扩展的成功部署的关键。要打造成功的部署过程，首先应确保已确认并定义所有必要的测试系统工件，并且采用适当的部署方法。动态硬件配置选项也是很多部署系统的重要考量因素。对于规模更大、更高级的系统，动态解析部署镜像与目标机器之间的依赖关系有助于降低部署过程的复杂性，并可缩短升级或重新创建系统镜像所需的时间。管理和测试部署镜像的各个版本是测试系统开发人员的另一个重要考量因素。无论是采用持续集成服务还是配置文件，都有必要维护一个可扩展的版本管理系统以实现分布式部署。测试系统的部署方法必须根据测试系统的功能和性质进行高度自定义。本指南中提供的构建可扩展解决方案的建议为通用建议，与使用的工具或系统功能无关。

TestStand Deployment Utility

TestStand Deployment Utility会自动执行部署中涉及的众多步骤，包括收集测试系统的序列文件、代码模块和支持文件，以及为这些文件创建安装程序，能够降低TestStand系统部署的复杂性。

了解TestStand Deployment Utility的更多信息

LabVIEW Application Builder最佳实践

LabVIEW Application Builder最佳实践可简化LabVIEW应用程序的管理和组织。这些建议可帮助工程师在开发之前确定指导原则和操作步骤，以确保其应用程序适用于大量VI和多个开发人员，从而节省开发时间和资源。

开始使用Application Builder最佳实践开展LabVIEW项目

©2022 NATIONAL INSTRUMENTS. 版权所有NATIONAL INSTRUMENTS、NI和NI.COM均为NATIONAL INSTRUMENTS的商标。此处提及的其他产品和公司名称均为其各自公司的商标或商业名称。 124400