

# Jenkins X 文档

书栈(BookStack.CN)

# 目 录

致谢

关于 Jenkins X

概览

特色

加速

概念

License

入门

入门概览

获取 jx

创建新集群

创建自定义 Builder

在 Kubernetes 上安装

安装过程中发生了什么

通过 GitOps 管理

下一步？

配置

研发

研发

创建 Spring Boot

快速开始

导入

浏览

Kubernetes 上下文

升级

预览

IDE

问题

Git 服务器

示例

创建 GKE 集群

创建 Spring

创建集群

架构

架构

组件

[源码](#)

[自定义资源](#)

[构建打包](#)

[Pod 模板](#)

[Docker Registry](#)

[扩展](#)

[API](#)

[命令行](#)

[常见问题](#)

[常见问题概览](#)

[FAQ](#)

[Jenkins 相关问题](#)

[安装问题](#)

[Issues](#)

[技术](#)

[开发问题](#)

[贡献](#)

[给 Jenkins X 项目做贡献](#)

[开发](#)

[API 文档](#)

[文档](#)

[计划的特性](#)

[分类问题](#)

# 致谢

当前文档《Jenkins X 文档》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-07-11。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：Jenkins X <https://jenkins-x.io/zh/about/>

文档地址：<http://www.bookstack.cn/books/jenkins-x>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

- [概览](#)
- [特色](#)
- [加速](#)
- [概念](#)
- [License](#)

# 关于 Jenkins X

---

Jenkins X 是基于 Kubernetes 的持续集成、持续部署平台。

该项目是 [Jenkins](#) 的子项目。

# 特色

---

Jenkins X 如何帮助你做持续交付

## 命令行

---

Jenkins X 带来了一个方便使用的命令行工具 `jx`：

- 安装 `Jenkins X` 到你已经存在的 `Kubernetes` 集群
- 创建一个新的 `kubernetes` 集群 并把 `Jenkins X` 安装进去
- 导入项目 到 `Jenkins X` 中以及他们的持续部署流水线设置
- 创建新的 `Spring Boot` 应用 并导入 `Jenkins X` 中，以及他们的持续部署流水线设置

## 流水线

---

不必深入了解 `Jenkins` 流水线的内部，`Jenkins X` 会默认给你的项目提供一些很好的流水线——基于 `DevOps` 最佳实践实现了所有的持续集成和持续部署

## 环境

---

环境指的是应用部署的地方。开发人员通常使用缩写来描述环境，例如：“测试中（Testing）、Staging/UAT或者生产（Production）”。

在 `Jenkins X` 中每个团队都有一套自己的环境。默认情况下，`Jenkins X` 会给每个团队创建一个 `Staging` 和 `生产` 环境，但你可以通过命令 `jx create environment` 创建一个新的环境。

我们使用 `GitOps` 来管理要部署到每个环境中的 `Kubernetes` 资源的配置和版本。因此，每个环境都有自己的 `git` 仓库，应用在这个环境中运行需要的 `Helm Charts`、版本以及配置都在库中。

在 `Kubernetes` 集群中一个环境对应一个命名空间。当 `Pull Requests` 被合并到环境所在的 `git` 库后，该环境的流水线就会把 `git` 库中的 `Helm Charts` 应用到环境命名空间中。

这意味着开发和运维都可以在同一个 `git` 库中，管理应用和资源在某个环境中的所有配置和版本，并且对环境的所有改变都可以在 `git` 中获取到。因此，这样很容易看到是谁作出的改变，而且，更重要的是当发生问题后很容易回滚改变。

## 部署升级

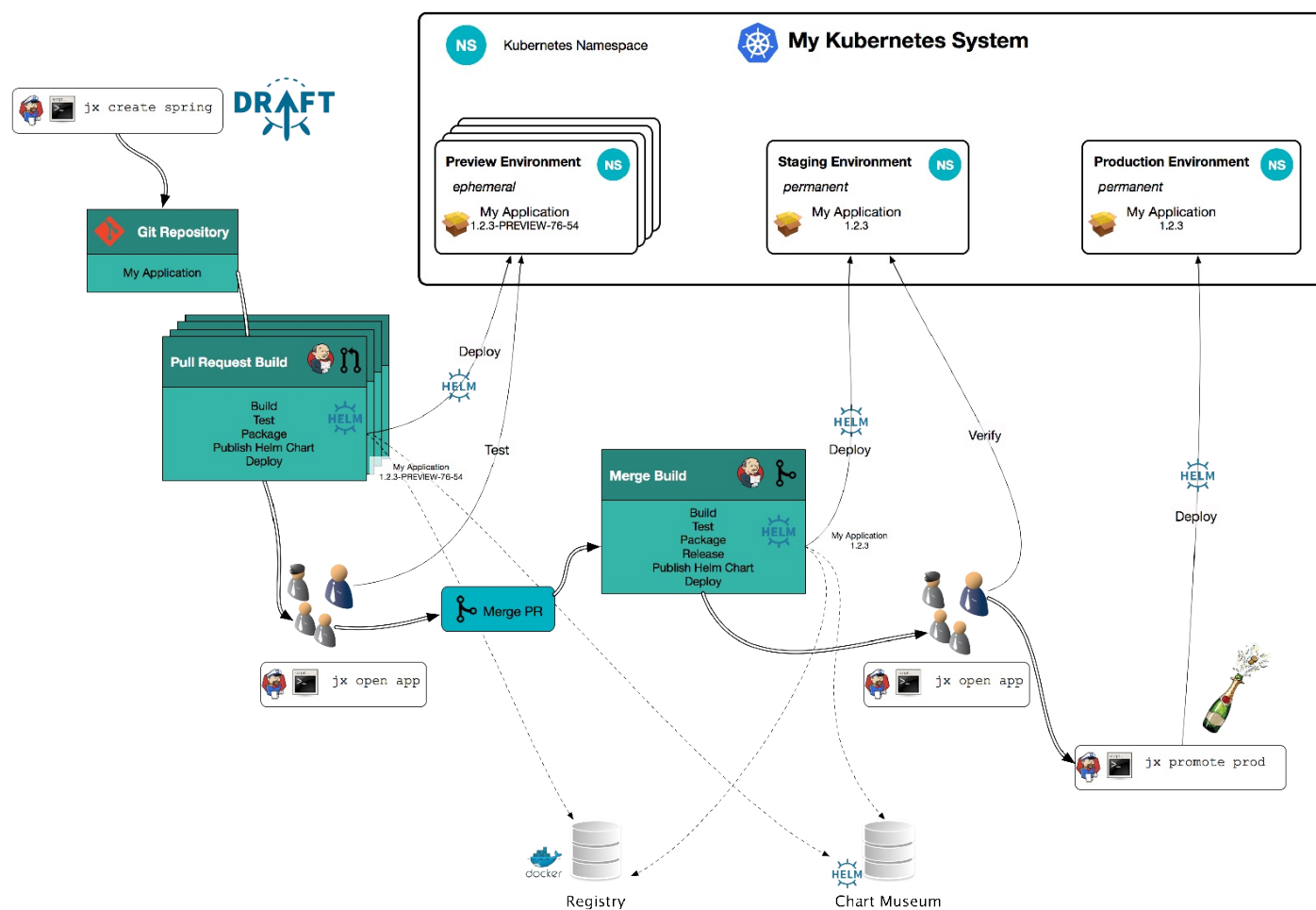
---

部署升级是通过 `GitOps` 在环境关联的 `git` 库上发起一个 `Pull Requests` 来实现的，这样所有的改变都通过 `git` 来审查、批准，因此所有的改变的都很容易回滚。

当环境所关联的 `git` 库上有新的变化合并到 `master` 后，环境的流水线就会触发，`helm` 就会把任何改变应用到资源上。

Jenkins X 的持续部署流水线把改变了的版本自动做部署升级，这是需要把配置中的“部署升级策略”设置为“自动”。默认情况下，“Staging”环境使用自动部署升级，而“生产”环境使用“手动”部署升级。

要手动把某个版本的应用部署升级到一个环境中的话，你可以使用 `jx promote` 命令。



## 预发环境

Jenkins X 允许你给 Pull Requests 设置一个预发环境，这样就可以在变更后并到 master 之前得到更多的反馈。这使你的变更在被合并以及发布之前更快得到反馈，并允许你避免在你的发版流水线中有人为的批准，加速变更在合并后的部署。

当预发环境启动并运行后，Jenkins X 将会在你的 Pull Requests 中添加一个带链接的评论，这样你们团队的成员就可以点击来尝试它！



rawlingsj commented 17 hours ago

Owner



★ PR built and available in a preview environment [rawlingsj-node3-pr-2 here](#)

## 反馈

正如在上面看到的，当你使用预发环境时，Jenkins X 会在你的 Pull Requests 上自动添加评论。



如果你在提交日志中引用了 issues (例如: 通过文本 `fixes #123`) , 那么, Jenkins X 流水线将会生成发版记录, 例如: [the jx releases](#).

同样地, 在升级到 `Staging` 或者 `生产` 环境时, 这些版本上也会在已修复的问题上自动添加对应环境可用的评论。例如:



jstrachan commented 4 days ago

Owner



✓ the fix for this issue is now deployed to **Staging** in version **0.0.2** and available at <http://jx-staging-my-spring-boot10.jx-staging.35.189.107.104.nip.io>

## 应用

一些最好的软件工具已经被打包为 helm charts, 部分预先集成在了 Jenkins X 中, 例如: Nexus、ChartMuseum、Monocular、Prometheus、Grafana等等。

## 插件

部分应用是内置的; 例如: Nexus、ChartMuseum、Monocular。其他的则是作为“插件”提供的。

要安装插件的话, 使用命令 `jx create addon`。例如:

```
1. jx create addon grafana
```

## 蓝图

如果想要看一下有什么特色很快会出现, 请查看[Jenkins X 蓝图](#)

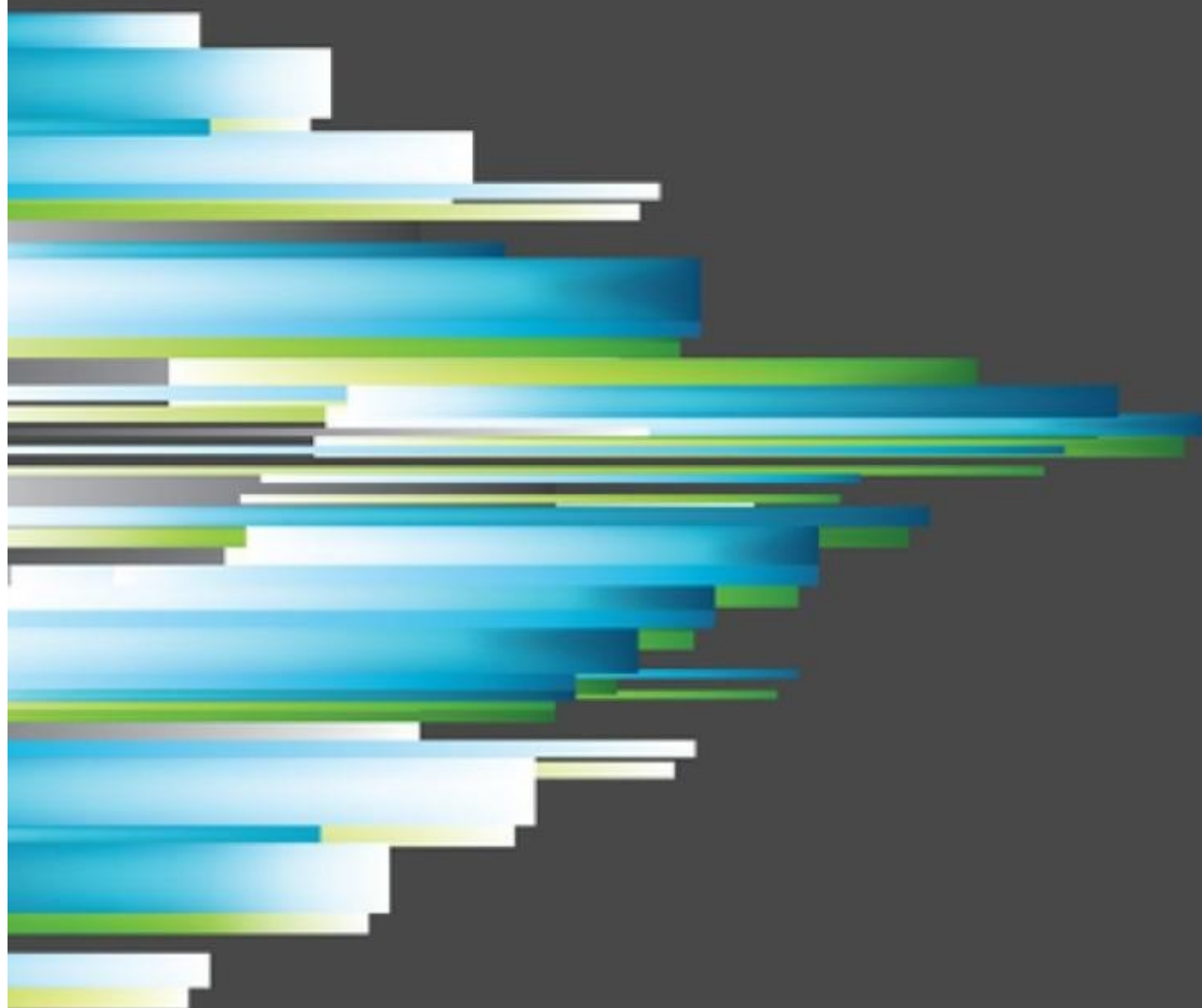
# 加速

---

Jenkins X 用到了哪些在《Accelerate》一书中提及的能力

THE SCIENCE OF DEVOPS  
**ACCELERATE**

Building and Scaling High Performing  
Technology Organizations



Nicole Forsgren, PhD  
Jez Humble *and* Gene Kim

Jenkins X 重新构思了云原生时代下的 CI/CD 实现，这些想法受到了 DevOps 状态报告和近来大热的《Accelerate》一书的深刻影响，这本书的三位合著者分别是：[Nicole Forsgren](#)、[Jez Humble](#)以及[Gene Kim](#)。

经年累月基于真实世界中的团队和组织收集上来的数据被 DevOps 领域的思想领袖和数据科学家们进行了深入的分析。《Accelerate》一书总结了一组有助于实施 DevOps 的能力，这些能力被 Jenkins X 实现以帮助用户以开箱即用的方式获取到科学证明过的收益。我们会从已经实现的能力项入手，并不断整合更多的能力进来。



## Jenkins X 的能力

Jenkins X 用到的《加速》一书中的 DevOps 能力



对所有制品进行版本控制



自动化部署过程



使用主干开发模式



实施持续集成



实施持续交付



使用松耦合的架构



赋能团队的架构设计

<https://jenkins-x.io/about/accelerate>

## 对所有构件进行版本控制

来自 Weaveworks 的天才们创造了 GitOps 的概念，这一点我们非常认同。对环境的任何变更，无论是一个新的

应用，版本升级，资源约束变更，还是简单的应用配置，都应该在 Git 上提交一个 Pull Request，并且采用类似环境的持续集成对这些变更进行验证，并且经过团队的审核，这个团队负责所有相关环境的变更控制。于是针对一个环境的任何变更都可以被追溯并且达到受控状态。

关联的加速能力项：对所有生产构件进行版本控制

## 自动化部署过程

### 环境

Jenkins X 在安装过程中会自动创建基于 Git 的环境，并且使用 `jx create environment` 命令来轻松地创建新的环境。此外，当通过 quickstart ( `jx create quickstart` ) 创建一个新的基于 Java 中 SpringBoot ( `jx create spring` ) 应用，或者导入已有应用 ( `jx import` ) 时，Jenkins X 都会自动帮你添加 CI/CD 流水线，并配置相关任务、git 代码仓库、webhook 来启用自动化部署流程。

Jenkins X 开箱即用地创建了永久的预发布和生产环境（这个是可配置的）以及一个 Pull Request 阶段临时使用的应用预览环境。

### 预览环境

在一个变更被合入主干之前，我们希望尽可能的进行测试、安全、验证和试验工作。使用临时动态创建的预览环境，任何 Pull Request 都会生成有一个预览版本被构建和部署，包括引用了公共库的下游应用。这就意味着我们可以同任何关联团队进行代码评审，测试和更好的协作，来确认这次变更可以部署到生产环境。

Jenkins X 的终极目标是提供一种方式，帮助开发人员、测试人员、设计人员和产品经理来验证将要合入主干的变更完全符合预期。我们希望确信这次变更没有对任何服务或特性带来负面影响，并且按照预想的那样来交付价值。

让预览环境变得真正有趣的是，当我们能够在不同阶段和成熟度的情况下进行 PR，也就是我们可以导入一定比例的真实生产环境流量，比如 beta 用户。那么我们可以分析此次变更的价值，并且使用假设驱动开发的方式运行多种自动化试验。这会帮助我们更好的理解当变更推送给所有用户时的效果。

关联的加速能力项：培养和支持团队试验

使用预览环境是导入自动化测试的绝佳方式。虽然 Jenkins X 支持这种方式，但是我们尚没有针对预览环境进行自动化测试的例子。一个最简测试集合应该可以确保应用正常启动，并且通过一段时间的 Kubernetes 的有效性 (liveness) 检查。相关内容包括：

关联的加速能力项：实施自动化测试

关联的加速能力项：自动化部署过程

### 永久环境

在软件开发中，我们习惯于在变更部署到生产环境之前在多套环境中验证。尽管这看起来没什么问题，但是如果在真正合并到主干之前，某些流程证明它并不合适，这就有可能导致其他变更的严重延迟。后续提交都会阻塞，并且紧急生产环境变更也同样会被推迟。

Jenkins X 希望所有变更和试验在合并主干之前都经过验证。变更在预发布环境中经过一段时间的验证后在推送到生产环境，理想情况下使用自动化的方式。

Jenkins X 的默认流水线提供了环境间自动化部署的能力。它可以被定制以适配你自己的 CI/CD 流水线要求。

Jenkins X 认为预发布环境应该尽可能的模拟生产环境，理想情况下使用服务网格技术导入真实生产数据来验证真实行为。着同样有助于预览环境的变更部署，我们可以将其链接到预发布中的非生产服务。

关联的加速能力项：自动化部署过程

## 使用主干开发分支策略

---

《Accelerate》一书的研究发现那些使用短分支生命周期并基于主干开发的团队拥有更好的效能。这对于 Jenkins X 核心团队成员而言再熟悉不过，所以 Jenkins X 通过配置 Git 仓库和 CI/CD 任务即可轻松实现这个能力。

## 实施持续集成

---

Jenkins X 将 CI 视为一个变更经过 Pull Request 合入主干前的验证活动。自动化配置代码仓库，Jenkins 和 Kubernetes 来提供开箱即用的持续集成功能。

## 实施持续交付

---

Jenkins X 将 CD 视为一个变更合入主干后到线上环境运行的活动。Jenkins X 将发布流水线中的大部分环境自动化：

Jenkins X 建议使用语义化版本号。采用 git 标签来计算下一次发布版本意味着无需在主干分支中保存最新的版本号。当发布系统将最新的和下一次版本保存在 git 仓库中，这会让 CD 变得困难，因为发布流水线中的变更会触发一次新的发布，这会导致递归的发布触发器。使用 git 标签可以避免这种情况来实现 Jenkins X 的完整自动化流程。

Jenkins X 会基于每一次针对主干的变更自动创建一个发布版本，这个版本就是潜在部署到生产环境的版本。

## 使用松耦合的架构

---

Jenkins X 面向 Kubernetes 用户，这让它可以受益于多种云的特性来设计和开发松耦合的解决方案。服务发现、容错性、扩展性、健康检查、滚动升级、容器编排和调度等仅仅是 Kubernetes 所带来的部分能力。

## 赋能团队的架构

---

Jenkins X 旨在帮助多语言的应用开发者。目前 Jenkins X 具备自动语言检测能力的 quickstart 和自动化 CI/CD 配置，比如 Golang, Java, NodeJS, .Net, React, Angular, Rust, Swift 以及更多语言支持。这样做也提供了一个持续性的工作方式来让开发者更加专注于开发活动。

Jenkins X 同样提供了很多插件，比如自动化度量数据收集和可视化工具：Grafana 和 Prometheus。集中化的

度量可以帮助我们查看构建和部署在 Kubernetes 上的应用指标。

**DevPods** 是一个全新的特性，可以帮助开发人员在本地 IDE 中编辑代码，并自动化同步到云环境上进行构建和重新部署。

Jenkins X 相信自动化可以帮助开发者在云环境下进行试验，使用不同的技术，并通过反馈让他们更快的做出最佳决策。



# 概念

---

Jenkins X 中的概念

Jenkins X 旨在使得 DevOps 原则和最佳实践对于研发人员来说简单。

## 原则

---

“DevOps 是一套旨在缩短从提交变更到生产发布的时间的实践，同时保证高质量”

DevOps项目的目标：

- 市场需求
- 提高部署频率
- 缩短修复时间
- 更低的市场错误率
- 更快的平均恢复时间相对于行业平均水平——每周一次和每月一次来说，高效的团队应该有能力每天部署多次。

代码从已提交到上生产应该少于一个小时，变更失败率也应该小于15%，而平均值在31-45%之间。

从失败中的平均恢复时间应该少于一个小时。

Jenkins X 设计了第一原则，允许团队采用 DevOps 的最佳实践，来达到行业的最高目标。

## 实践

---

下列最佳实践被认为是DevOps成功的关键：

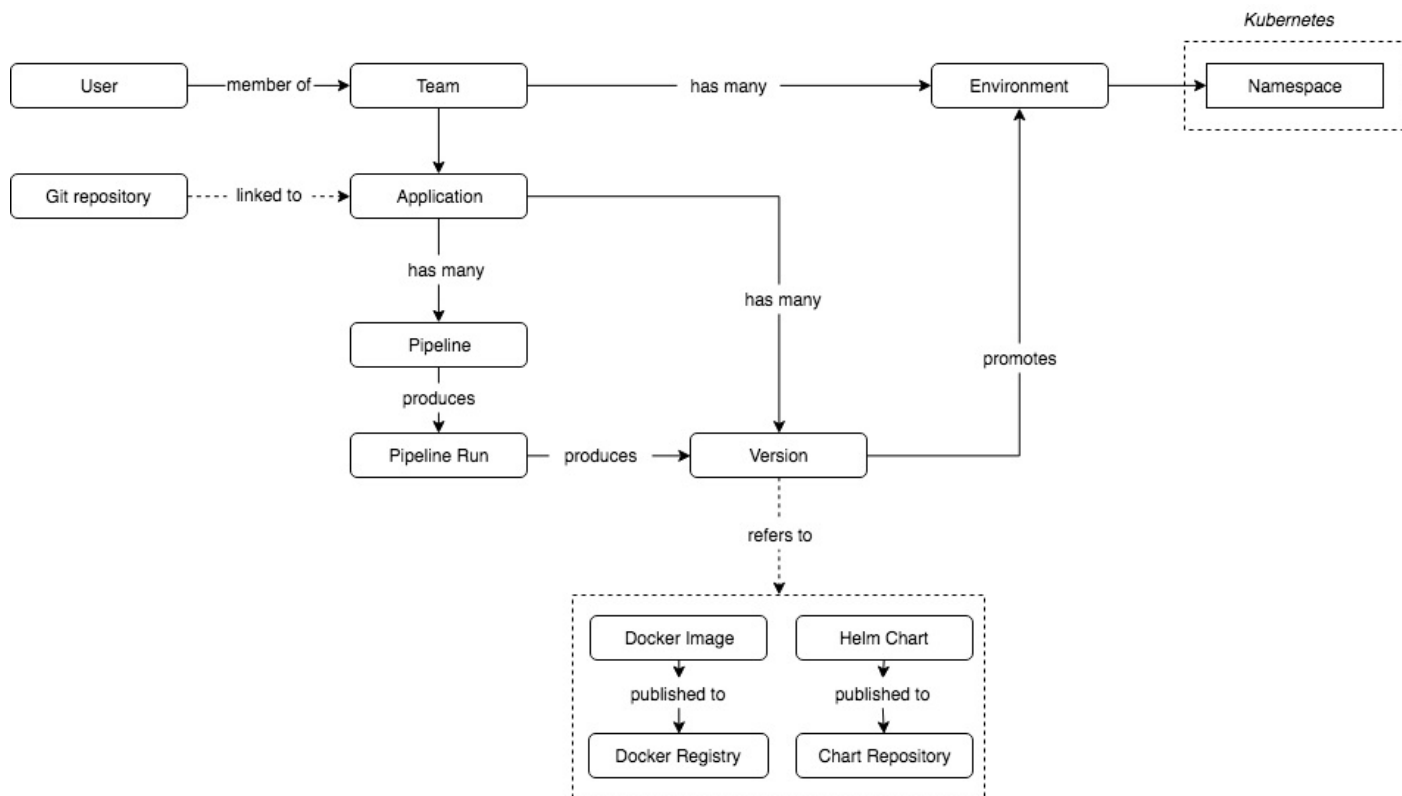
- 架构松耦合
- 自服务配置
- 自动化管理
- 持续构建 / 集成和部署
- 自动发布管理
- 增量测试
- 配置即代码Jenkins X 把大量常见的方法论和组件集成为复杂度最小的方法。

## 架构

---

Jenkins X 基于松耦合架构的 DevOps 模型，被设计用来支持在多个团队之间，部署大量可重复、可管理的分布式微服务。

## 概念模型



## 构成

Jenkins X 基于以下的核心组件：

## Kubernetes 和 Docker

Kubernetes 是系统的核心，它已经成为了 DevOps 事实上的虚拟基础设施平台。每个主要的云服务商现在都已经提供了 Kubernetes 基础设施，并且可能已经在很多私有基础设施中也被安装了。测试环境可能也在使用 Minikube 安装器创建本地开发环境。

从功能上，该Kubernetes 平台扩展了 Docker 提供的基本容器化原则，用于跨多个物理节点。

简单来说，Kubernetes 提供了同构的虚拟基础设施，可以通过动态添加或者移除节点实现伸缩。每个节点都在一个大型专用虚拟网络空间里。

Kubernetes 中的部署单元是 Pod，它包含一个或多个 Docker 容器以及一些元数据。Pod 中所有的容器分享同样的虚拟 IP 地址和端口范围。Deployments在 Kubernetes 中是申明式的，因此，用户指定特定版本的 Pod 中部署的实例数量，Kubernetes 据此来计算跨节点中的 Pod 是应该删除或部署。根据标签匹配来决定资源的实例数量。一旦被部署，Kubernetes 就会通过定期的健康检查来保证 Pod 的数量，终止或则替换没响应的 Pod。

为了增加某些结构，Kubernetes 允许创建虚拟命名空间用于对 Pod 做逻辑分离，隐含地把一组 Pod 和特定资源关联起来。例如：在同一个命名空间里的资源共用安全策略。同一个命名空间里的资源名称必须是唯一的，但在不同的命名空间里不受该约束。

在 Jenkins X 模型中，一个 Pod 相当于部署好的一个微服务（大多数情况下）。当微服务需要横向扩容时，Kubernetes 允许在一个 Pod 中有多个相同的实例被部署，每个实例都有自己的虚拟 IP 地址。它们可以聚合为

一个虚拟终端，也就是服务，它有唯一的静态 IP 地址，并且本地的 DNS 纪录会匹配服务的名称。对服务的调用会动态地随机映射到健康的 POD 实例上。服务还可以重新映射端口。在 Kubernetes 虚拟网络中，服务可以根据域名全称来引用的，格式为 `<service-name>.<namespace-name>.svc.cluster.local`，它也可以缩短为 `<service-name>.<namespace-name>`，或者服务的命名空间都一样的话可以是 `<service-name>`。因此，如果一个 RESTful 服务 'payments' 部署在命名空间 'finance' 中时，就可以通过 `http://payments.finance.svc.cluster.local` 或者 `http://payments.finance` 或只是 `http://payments` 来引用，这要取决于代码的调用位置。

为了在本地网络以外的地方访问服务，Kubernetes 需要给每个服务创建一个入口。最常见的形式是利用一个或者多个负载均衡绑定静态 IP 地址，在 Kubernetes 外面的虚拟基础设施和路由网络请求映射到内部服务。给负载均衡的静态 IP 地址创建一个通配符的外部 DNS 纪录，就划一把服务映射到外部的全称域名上。例如：如果我们的负载均衡映射到 `*.jenkins-x.io`，那么我们负载的服务就可能暴露为 `http://payments.finance.jenkins-x.io`。

Kubernetes 为强大的不断提高的平台在部署服务上提供巨大的伸缩能力，但它同时也是复杂不容易理解的，而且要正确进行配置也有一定的困难。Jenkins X 给 Kubernetes 引入了一系列默认约定以及简单的工具，为了优化 DevOps 并管理松耦合的服务。

命令行工具 `jx` 提供了执行基于 Kubernetes 实例的简单方法，例如：查看日志、连接容器实例。另外，Jenkins X 扩展了 Kubernetes 的命名空间约定来创建环境，这样就把发布流水线串联起来了。

一个 Jenkins X 环境代表一个虚拟基础设施环境，例如：Dev、Staging、Production 等等。两个环境之间的流转规则是可以定义的，因此可以通过流水线实现自动或者人工发版。每个环境的管理都遵循 GitOps 的方法——环境状态的维护依赖于 Git 仓库，提交或者回滚都会关联 Kubernetes 中的环境状态的变化。

Kubernetes 集群可以直接通过命令 `jx create cluster` 来创建，这使得当发生错误时可以很容易地复制一个集群。相同地，Jenkins X 平台通过 `jx upgrade platform` 来升级已有的集群。Jenkins X 通过 `jx context` 支持多 Kubernetes 集群，在一个集群中可以通过 `jx environment` 来切换环境。

研发人员应该知道 Kubernetes 提供的分布式配置数据以及夸集群的安全身份。ConfigMaps 可以用来创建一系列键值对形式的非敏感配置元素句，Secrets 与之类似，但保存的是加密的身份信息。Kubernetes 还提供为 Pod 指定资源配额的机制，这在优化跨节点之间的部署上是有必要的。这一点，我们先简短地讨论下。

默认情况下，Pod 的状态是临时的。当 Pod 被删除后，该 Pod 下载本地文件系统中的任何数据都会丢失。研发人员应该明白，Kubernetes 为了节点的负载均衡，可能随时会删除或者重建 Pod，因此本地数据可能会在任何时间丢失。当用到有状态的数据时，应该申明持久化的卷 (Volumes)，并挂载到指定 Pod 的文件系统中。

## Helms 和 Draft

要直接和 Kubernetes 交互的话可以使用命令 `kubectl`，或者传递各种格式的 YAML 数据给 API。这一点会比较困难，而且错误信息的可读性差。为了沿用 DevOps “配置即代码” 的原则，Jenkins X 借助 Helm 和 Draft 来创建应用的原子配置。

Helm 通过 Chart 的概念简化了 Kubernetes 的配置，它把一套需要部署到 Kubernetes 中的应用或者服务的原数据文件组织起来。Helm 不是维护一套基于 Kubernetes API 的样板化 YAML 文件，而是使用模板语言来通过需要的值来创建 YAML 文件。这使得在部署期间可以重用 Kubernetes 应用的配置文件。

# Apache License

---

Jenkins X is released under the Apache 2.0 license.

Jenkins X is released under the Apache 2.0 license.

Version 2.0, January 2004 <http://www.apache.org/licenses/LICENSE-2.0>

*Terms and Conditions for use, reproduction, and distribution*

## 1. Definitions

---

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the

Work and any modifications or additions to that Work or Derivative Worksthereof, that is intentionally submitted to Licensor for inclusion in the Workby the copyright owner or by an individual or Legal Entity authorized to submiton behalf of the copyright owner. For the purposes of this definition,“submitted” means any form of electronic, verbal, or written communication sentto the Licensor or its representatives, including but not limited tocommunication on electronic mailing lists, source code control systems, andissue tracking systems that are managed by, or on behalf of, the Licensor forthe purpose of discussing and improving the Work, but excluding communicationthat is conspicuously marked or otherwise designated in writing by the copyrightowner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalfof whom a Contribution has been received by Licensor and subsequentlyincorporated within the Work.

## 2. Grant of Copyright License

---

Subject to the terms and conditions of this License, each Contributor herebygrants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free,irrevocable copyright license to reproduce, prepare Derivative Works of,publicly display, publicly perform, sublicense, and distribute the Work and suchDerivative Works in Source or Object form.

## 3. Grant of Patent License

---

Subject to the terms and conditions of this License, each Contributor herebygrants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free,irrevocable (except as stated in this section) patent license to make, havemade, use, offer to sell, sell, import, and otherwise transfer the Work, wheresuch license applies only to those patent claims licensable by such Contributorthat are necessarily infringed by their Contribution(s) alone or by combinationof their Contribution(s) with the Work to which such Contribution(s) wassubmitted. If You institute patent litigation against any entity (including across-claim or counterclaim in a lawsuit) alleging that the Work or aContribution incorporated within the Work constitutes direct or contributorypatent infringement, then any patent licenses granted to You under this Licensefor that Work shall terminate as of the date such litigation is filed.

## 4. Redistribution

---

You may reproduce and distribute copies of the Work or Derivative Works thereofin any medium, with or without modifications, and in Source or Object form,provided that You meet the following conditions:

- **(a)** You must give any other recipients of the Work or Derivative Works a copy ofthis License; and

- **(b)** You must cause any modified files to carry prominent notices stating that You changed the files; and
- **(c)** You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- **(d)** If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

## 5. Submission of Contributions

---

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

## 6. Trademarks

---

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

## 7. Disclaimer of Warranty

---

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without

limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

## 8. Limitation of Liability

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

## 9. Accepting Warranty or Additional Liability

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

*END OF TERMS AND CONDITIONS*

## APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets `[]` replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

apache-notice.txt

```
[ ]
```

- 1.
2. **Copyright** [yyyy] [name of copyright owner]

```
3.
4. Licensed under the Apache License, Version 2.0 (the "License");
5. you may not use this file except in compliance with the License.
6. You may obtain a copy of the License at
7.
8.    http://www.apache.org/licenses/LICENSE-2.0
9.
10. Unless required by applicable law or agreed to in writing, software
11. distributed under the License is distributed on an "AS IS" BASIS,
12. WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13. See the License for the specific language governing permissions and
14. limitations under the License.
```



- [入门概览](#)
- [获取 jx](#)
- [创建新集群](#)
- [创建自定义 Builder](#)
- [在 Kubernetes 上安装](#)
- [安装过程中发生了什么](#)
- [通过 GitOps 管理](#)
- [下一步？](#)
- [配置](#)

# 入门

---

首先，需要在你本地的机器上[安装 jx 命令行工具](#)。

你可以使用 [jx 命令](#) 来[创建一个新的 kubernetes 集群](#)，然后 Jenkins X 就会自动安装。

或者，如果你已经有了一个 kubernetes 集群，那么可以[在你的 kubernetes 集群上安装 Jenkins X](#)。

# 获取 jx

如何在你的机器上安装jx二进制包

根据你的操作系统选择最适合的指令：

## macOS

在 Mac 上你可以使用 [brew](#)：

```
1. brew tap jenkins-x/jx
2. brew install jx
```

或者，如果您尚未安装 [brew](#) ，并且喜欢手动安装的话，请执行如下指令安装：

```
1. curl -L https://github.com/jenkins-x/jx/releases/download/v2.0.413/jx-darwin-amd64.tar.gz | tar xzv
2. sudo mv jx /usr/local/bin
```

## Linux

```
1. mkdir -p ~/.jx/bin
   curl -L https://github.com/jenkins-x/jx/releases/download/v2.0.413/jx-linux-amd64.tar.gz | tar xzv -C
2. ~/.jx/bin
3. export PATH=$PATH:~/.jx/bin
4. echo 'export PATH=$PATH:~/.jx/bin' >> ~/.bashrc
```

## Windows

- 如果你使用 [Chocolatey](#)，那么这里有一个 [可用的包](#)。要安装 `jx` 二进制请运行：

```
1. choco install jenkins-x
```

要升级 `jx` 二进制请运行：

```
1. choco upgrade jenkins-x
```

- 如果你使用 [scoop](#)，那么这里有一个 [可用的清单](#)。要安装 `jx` 二进制请运行：

```
1. scoop install jx
```

要升级 `jx` 二进制请运行：

```
1. scoop update jx
```

获取 jx

## 其他平台

下载二进制包 `jx` 然后加到环境变量 `$PATH` 中

或者，你可以尝试 [自行构建](#)。然而，如果你要自行构建的话，请注意移除所有旧版本的 `jx` 二进制文件，这样你的本地构建才会出现在环境变量 `$PATH` 的第一位 :)

## 获得帮助

查找可用的命令类型：

```
1. jx
```

或者，获取指定命令的帮助，例如： `create` 命令，可以输入：

```
1. jx help create
```

你也可以浏览 [jx 命令参考文档](#)

## 参照

- [创建自定义 Builder](#)
- [配置](#)
- [安装过程中发生了什么](#)
- [在 Kubernetes 上安装](#)
- [创建新集群](#)

# 创建新集群

如何通过 Jenkins X 创建新的 Kubernetes 集群

通过已经安装的 Jenkins X 创建一个新的集群，使用命令 `jx create cluster` 。

如下所示，支持很多不同的公有云提供商。

为了最好的入门体验，我们目前推荐使用 **Google Container Engine (GKE)**。如果你没有谷歌云账号的话，谷歌云平台提供三百美元的额度。查看 <https://console.cloud.google.com/freetrial>

这有一个小的演示，同时展示 GKE、AKS 和 Minikube。在不同的设备（云）上启动需要花点时间，请耐心等待！

## 使用谷歌云（GKE）

使用命令 `jx create cluster gke`：

```
1. jx create cluster gke
```

该命令假设你有一个谷歌账户，并且已经设置了一个默认项目，可以再里面创建 Kubernetes 集群。

现在 [使用 Jenkins X 更快速地开发应用](#)。

## 使用亚马逊（AWS）

使用命令 `jx create cluster aws`：

```
1. jx create cluster aws
```

这会通过你的亚马逊账户，使用命令 `kops` 创建一个新的 Kubernetes 集群并安装 Jenkins X。

来试试这个，我们建议你参照 [AWS Workshop for Kubernetes](#) 设置 AWS Cloud9 IDE。

然后，在 Cloud9 中打开一个新的终端，试试这些命令：

```
1. curl -L https://github.com/jenkins-x/jx/releases/download/v2.0.413/jx-linux-amd64.tar.gz | tar xzv
2. sudo mv jx /usr/local/bin
3. jx create cluster aws
```

现在 [使用 Jenkins X 更快速地开发应用](#)。

## 使用 Azure（AKS）

使用命令 `jx create cluster aks`：

```
1. jx create cluster aks
```

现在 [使用 Jenkins X 更快速地开发应用](#)。

## 使用 Minikube (local)

有些人在开始使用 minikube 时遇到问题，可能有几个原因：

- minikube 需要更新你的机器以及虚拟化软件
- 你可能已经安装了旧版本的 Docker 或者 minikube、kubectl、helm等。因此，我们强烈建议使用上面的公有云来尝试 Jenkins X。他们都有免费体验，所以应该不会花费你的任何现金，而且还给了你体验云的机会。

如果你还是想尝试 minikube，那么，我们建议从头开始，并让 jx 帮你创建

```
1. jx create cluster minikube
```

现在 [使用 Jenkins X 更快速地开发应用](#)。

## 故障排除

如果你在安装 Jenkins X 时遇到任何问题，请检查我们的 [故障排除](#) 或者 [让我们知道](#)，我们会尽力给予帮助。

## 参照

- [创建自定义 Builder](#)
- [配置](#)
- [安装过程中发生了什么](#)
- [在 Kubernetes 上安装](#)
- [获取 jx](#)

# 创建自定义 Builder

如何为 Jenkins X 创建一个自定义 Builder

在 Jenkins X 中，可以创建字段自定义的 Builder（也就是 [POD templates](#)）或覆盖已有的。你只需要基于 [builder-base](#) 或它的 [slim](#) 版本的镜像。

## 从零创建一个自定义 Builder

### Builder 镜像

首先，您需要为 Builder 创建一个 docker 镜像。从 `Dockerfile` 开始的一个实例可能类似于：

```
1. FROM jenkinsxio/builder-base:latest
2.
3. # Install your tools and libraries
4. RUN yum install -y gcc openssl-devel
5.
6. CMD ["gcc"]
```

现在，您可以构建并发布这个镜像到您的 registry：

```
1. export BUILDER_IMAGE=<YOUR_REGISTRY>/<YOUR_BUILDER_IMAGE>:<VERSION>
2. docker build -t ${BUILDER_IMAGE} .
3. docker push ${BUILDER_IMAGE}
```

别担心，当新的镜像需要构建时，您无需每次手动执行这些步骤。Jenkins X 可以为您管理这些。您只需要把

`Dockerfile` 推送到类似于[这个](#)代码仓库中。然后，根据您的组织名称来调整 `Jenkinsfile`，并使用下面的命令导入 Jenkins X 平台：

```
1. jx import --url <REPOSITORY_URL>
```

之后，您每次推送一个变更，Jenkins X 将会自动地构建和发布镜像。

### 安装 Builder

当您安装或者升级 Jenkins X 时就可以安装您的 Builder 了。

在您的 `~/.jx/` 目录下创建文件 `myvalues.yaml` 并写入以下内容：

```
1. jenkins:
2.   Agent:
3.     PodTemplates:
4.       MyBuilder:
5.         Name: mybuilder
```

```

6.      Label: jenkins-mybuilder
7.      volumes:
8.      - type: Secret
9.        secretName: jenkins-docker-cfg
10.       mountPath: /home/jenkins/.docker
11.      EnvVars:
12.        JENKINS_URL: http://jenkins:8080
13.        GIT_COMMITTER_EMAIL: jenkins-x@googlegroups.com
14.        GIT_AUTHOR_EMAIL: jenkins-x@googlegroups.com
15.        GIT_AUTHOR_NAME: jenkins-x-bot
16.        GIT_COMMITTER_NAME: jenkins-x-bot
17.        XDG_CONFIG_HOME: /home/jenkins
18.        DOCKER_CONFIG: /home/jenkins/.docker/
19.      ServiceAccount: jenkins
20.      Containers:
21.        Jnlp:
22.          Image: jenkinsci/jnlp-slave:3.14-1
23.          RequestCpu: "100m"
24.          RequestMemory: "128Mi"
25.          Args: '${computer.jnlpmac} ${computer.name}'
26.        Dlang:
27.          Image: <YOUR_BUILDER_IMAGE>
28.          Privileged: true
29.          RequestCpu: "400m"
30.          RequestMemory: "512Mi"
31.          LimitCpu: "1"
32.          LimitMemory: "1024Mi"
33.          Command: "/bin/sh -c"
34.          Args: "cat"
35.          Tty: true

```

根据需要替换 Builder 名称和镜像。

您可以继续安装 Jenkins X，然后 Builder 将会自动添加到平台。

## 使用 Builder

现在，您的 Builder 已经在 Jenkins 中安装了，您可以在 `Jenkinsfile` 中轻松地引用：

```

1. pipeline {
2.   agent {
3.     label "jenkins-mybuilder"
4.   }
5.   stages {
6.     stage('Build') {
7.       when {
8.         branch 'master'
9.       }
10.      steps {
11.        container('mybuilder') {
12.          // your steps
13.        }

```



```
14.     }
15.   }
16. }
17. post {
18.     always {
19.         cleanWs()
20.     }
21. }
22. }
```

## 覆盖已有的 Builder

Jenkins X 自带了很多[预安装的 Builder](#)，在安装或升级过程中可以根据需要覆盖。

您只需要基于[基础 Builder](#) 镜像或者[Builder 镜像](#) 自定义。在上面查看细节。

然后，您可以在目录 `~/.jx/` 中创建文件 `myvalues.yaml`，并写入一下内容：

```
1. jenkins:
2.   Agent:
3.     PodTemplates:
4.       Maven:
5.         Containers:
6.           Maven:
7.             Image: <YOUR_REGISTRY>/<YOUR_MAVEN_BUILDER_IMAGE>:<VERSION>
8.       Nodejs:
9.         Containers:
10.          Nodejs:
11.            Image: <YOUR_REGISTRY>/<YOUR_NODEJS_BUILDER_IMAGE>:<VERSION>
12.       Go:
13.         Containers:
14.           Go:
15.             Image: <YOUR_REGISTRY>/<YOUR_GO_BUILDER_IMAGE>:<VERSION>
```

您可以继续安装 Jenkins X，这些 Builder 将会自动地添加到平台。

## 参照

- [配置](#)
- [安装过程中发生了什么](#)
- [在 Kubernetes 上安装](#)
- [创建新集群](#)
- [获取 jx](#)

# 在 Kubernetes 上安装

如何在已有的 Kubernetes 集群上安装 Jenkins X

Jenkins X 可以在 Kubernetes 1.8 以及更高版本上安装。需要的依赖有：

- RBAC 是可用的
- 启用 docker 私有仓库。这样的话，流水线可以在 Kubernetes 集群中使用 docker 仓库（通常不是公共的因此不支持 https）。后续，你可以修改你的流水线来使用其他仓库。

## 通过 kops 启用私有仓库

注意，如果你是在 AWS 环境中，你可能会想使用 `jx create aws` 命令来帮你自动化完成所有步骤！

如果你是通过 `kops` 创建的 kubernetes 集群，那么你可以这么做：

```
1. kops edit cluster
```

然后，确保在 YAML 文件的章节 `spec` 中有 `docker` 配置：

```
1. ...
2. spec:
3.   docker:
4.     insecureRegistry: 100.64.0.0/10
5.     logDriver: ""
```

上面的 IP 范围 `100.64.0.0/10` 是 AWS 上的，但你需要修改为其他 Kubernetes 集群的；它依赖于 Kubernetes 服务的 IP 范围。

保存后，你可以参考下面的命令进行验证：

```
1. kops get cluster -oyaml
```

然后查找 `insecureRegistry` 章节。

现在，确保这些修改在你的集群类型上是激活的：

```
1. kops update cluster --yes
2. kops rolling-update cluster --yes
```

你现在可以继续了！

## 安装 Jenkins X

为了在已有的 kubernetes 集群上安装 Jenkins X 你可以使用命令 `jx install`：

```
1. jx install
```

如果你知道提供商的话，可以通过命令行来指定。例如：

```
1. jx install --provider=aws
```

## 参照

---

- [配置](#)
- [安装过程中发生了什么](#)
- [通过 GitOps 管理](#)
- [下一步？](#)
- [创建自定义 Builder](#)

# 安装过程中发生了什么

---

安装 Jenkins X 时究竟做了什么

Jenkins X 命令行在安装 Jenkins X 平台时会做如下事情：

## 安装二进制客户端来管理你的集群

---

如果您运行在 Mac OS X 上，Jenkins X 会使用 `Homebrew` 来安装不同的命令行。不存在就会安装。

### 安装 `kubectl`

`kubectl` 是 Kubernetes 的命令行。它允许您和您的 Kubernetes 集群通过 API server 交互。

### 安装 Helm

Jenkins X 将会安装 `helm` 客户端 - (可能是 `helm 2.x` 或 `helm 3`)，如果它不在您的环境变量里的话。  
Helm 是用于打包 Kubernetes 中的应用或资源（也叫做 `charts`），并迅速地成为标准。

## 安装云提供商的命令行

如果您在使用公有云，会有相关的命令行来与之交互。当通过 `jx create cluster` 命令来安装时，您的云提供商相关的二进制如果不在环境变量里的话，也会被安装。

- AKS 集群 (Azure) 的 `az`
- GKE 集群 (Google Cloud) 的 `gcloud` for GKE cluster (Google Cloud)
- AWS 集群 (Amazon Web Services) 的 `kops`
- AWS EKS 集群的 `eksctl`
- OKS 集群 (Oracle Cloud) 的 `oci` 如果您想要在本机通过 `minikube` 或 `minishift` 运行 Jenkins X 的话，下面的二进制会被添加：
- 本地 `minishift` (OpenShift) 集群 的 `oc` (OpenShift CLI) 和 `minishift`
- 本地 `minikube` 集群的 `minikube` 最后，Jenkins X 将会根据需要安装 VM 驱动，通常 Mac OS X 上是 `xhyve` 或 Windows 上是 `hyperv`。其他驱动则需要手动安装。

## 创建 Kubernetes 集群

---

之后，集群会通过云提供商的命令来创建（例如：Azure 的 `az aks create` 命令）。

## 设置 Jenkins X 平台

---

## 创建 Jenkins X 命名空间

然后，会为 Jenkins X 平台创建一个命名空间，用于存放 Jenkins X 基础组件。默认为：`jx`。

## 安装 Tiller (可选，只有 Helm 2 需要)

Tiller，也就是 Helm 的服务器端，会部署到命名空间 `kube-system` 中。[Helm](#) 是 Kubernetes 的包管理器，也用于部署 Jenkins X 的其他组件。

## 设置 Ingress 控制器

在 Kubernetes 集群中，Service 和 Pod 的 IP 只能在集群网络中访问。为了能够访问集群，必须要创建一个 Ingress。Ingress 是路由到集群内的 Service 的规则集。Ingress 规则是由 Kubernetes API 配置在 Ingress 资源中，而 Ingress Controller 是必要的。所有的这些 Jenkins X 都会替您做——为下面的 Service 设置一个 Ingress Controller 和相关联的后端 Ingress 规则（一旦部署完后）：

- `chartmuseum`
- `docker-registry`
- `jenkins`
- `monocular`
- `nexus`

默认，Jenkins X 将会通过域名 `nip.io` 暴露 Ingress，并生成自签名的证书。当按照完后，您可以通过命令 `jx upgrade ingress --cluster` 轻松地修改为您自己的域名和签名。

## 配置 git 仓库

Jenkins X 需要一个 Git 仓库提供商，以便能够插件环境仓库。如果您没有提供参数 `git-provider-url` 的计划，默认使用 GitHub。您需要提供用户名和 Token 来和 Git 交互，尤其是 Jenkins。

## 创建管理员凭据

Jenkins X 为 Monocular/Nexus/Jenkins 生成管理员密码并保存到 Secret 中。当在 helm 安装的时候就会取出来使用（因此，密码可以用在流水线中）。

## 检出云环境仓库

[云环境仓库](#)保存所有特定的配置以及加密的 Secret，这些将会应用在您的 Kubernetes 集群中的 Jenkins 平台。这些 Secret 将会由 Helm 包管理器来加解密。

## 安装 Jenkins X 平台

[Jenkins X 平台](#)保存安装了的组件的 Helm Chart，用于提供 Jenkins X 真正的 CD 解决方案。这包括：

- [Jenkins](#) 一个 CI/CD 流水线方案

- [Nexus](#) 一个制品仓库
- [ChartMuseum](#) 一个 Helm Chart 仓库
- [Monocular](#) 提供了一个 Web UI 用于搜索和发现通过 Jenkins X 部署到您的集群中的 Chart。

## 参照

---

- [配置](#)
- [在 Kubernetes 上安装](#)
- [下一步？](#)
- [通过 GitOps 管理](#)
- [创建自定义 Builder](#)

# 通过 GitOps 管理

使用 GitOps 配置和升级你的 Jenkins X 设施

我们推荐你使用 GitOps 管理你的 Jenkins X 设施，升级它、配置它、以及添加或移除扩展[应用](#)，这样容易审计谁在你的设施上做了什么变更并且容易恢复坏的变更。

当前这仅在 AWS 和 Google 云可用，因为它要求我们的 vault 操作员（需要云存储和 KMS）存储凭据，而所有其他配置都存储在开发环境 git 仓库中。

## 使用 GitOps 管理 Jenkins X

如果你正在创建一个集群或者在已经存在的集群安装，这里有一种快速简便的方法来使用 GitOps 来管理 Jenkins X 本身。它是 `-ng`，为下一代 Jenkins X 而来。在我们今年晚些时候发布 Jenkins X 2.x 时，我们会将此功能标记设为默认选项。

`-ng` 标记是这些标记的一个别名：`-gitops -vault -no-tiller -tekton`。所以它还附带了对 [Jenkins X 流水线](#) - 基于 Tekton 的新式云原生流水线引擎的支持。

如果你仍然想要使用 Jenkins 服务器作为 Jenkins X 中自动化 CI/CD 流水线的执行引擎，那么你可以使用 `-gitops -vault` 代替。仍要注意是的即使使用了 `-ng` 以及使用了由 Tekton 驱动的 [Jenkins X 流水线](#)，你仍然需要创建你自己的[自定义 Jenkins 服务器](#)来运行传统的 Jenkins 任务和流水线。

一旦你使用 GitOps 安装了 Jenkins X 来管理开发环境，那么表明安装了 Jenkins X 和它的附加应用程序，你将为 Dev, Staging, Production 环境获得一个额外的 git 仓库。它也意味着如果你用一个更新命令如 `jx upgrade platform` 或通过 `jx add app` 添加、更新、删除应用，那么那些命令将在开发环境的 git 存储库生成 Pull Request，就像当你发布新版本的微服务时，`promotion` 是如何工作的。

## 如果出现问题

一般来说，当使用 Tekton 时，Jenkins X 可以很容易地自我升级。但是，如果升级让 Jenkins X 无法实施 CI/CD，那么使用 GitOps 回退更改将不起作用；)

如果你在升级 Jenkins X 过程中遇到任何问题，这里有一种手动方法可以应用开发环境的 git 存储库的内容：

```
1. git clone $MY_DEV_GIT_CLONE_URL jenkins-x-dev-env
2. cd jenkins-x-dev-env/env
3. jx step env apply
```

## 参照

- [下一步？](#)
- [配置](#)
- [安装过程中发生了什么](#)

通过 GitOps 管理

- [在 Kubernetes 上安装](#)
- [创建自定义 Builder](#)



下一步？

## 下一步？

---

在 Kubernetes 集群上安装完 Jenkins X 后该怎么做

好的，现在你已经 [安装了 jx 命令](#)，并完成了下面的步骤：

- [通过 Jenkins X 创建 Kubernetes 集群](#)
- [在已有的 kubernetes 集群上安装 Jenkins X](#)那么，下一步呢？

在 [研发](#) 章节中可能会有你想要尝试的内容，例如：

- [创建一个新的 Spring Boot 应用并导入 Jenkins X](#)
- [创建一个新的快速开始并导入 Jenkins X](#)
- [把已有的源码导入 Jenkins X](#)
- [浏览](#) 可以浏览流水线，构建，应用和活动你也可以看看 [你可以用 Jenkins X 做什么的各种示例](#)

## 参照

---

- [通过 GitOps 管理](#)
- [配置](#)
- [安装过程中发生了什么](#)
- [在 Kubernetes 上安装](#)
- [创建自定义 Builder](#)

# 配置

## 自定义你的 Jenkins X 安装

Jenkins X 应该为你的云服务商提供默认可用的配置。例如：如果你使用 AWS 或 EKS，Jenkins X 自动地使用 ECR。

然而，你可以修改 Jenkins X 使用的 helm charts 的配置。

要做到这一点，你需要在运行命令 `jx create cluster` 或 `jx install` 的目录下创建一个文件

```
myvalues.yaml
```

。

然后，这个 YAML 文件可以覆盖 Jenkins X 中的任何 charts 中的 `values.yaml` 文件。

## Nexus

例如：如果你希望在安装过程中禁用 Nexus，而使用不同主机上的一个独立的 Nexus，那么，你可以使用

```
myvalues.yaml
```

 中的服务链接来替代：

```
1. nexus:
2.   enabled: false
3. nexusServiceLink:
4.   enabled: true
5.   externalName: "nexus.jx.svc.cluster.local"
```

要禁用并使用 chart museum 的服务链接的话添加：

```
1. chartmuseum:
2.   enabled: false
3. chartmuseumServiceLink:
4.   enabled: true
5.   externalName: "jenkins-x-chartmuseum.jx.svc.cluster.local"
```

## Jenkins 镜像

Jenkins X 中我们提供了一个默认的 Jenkins docker 镜像 `jenkinsxio/jenkinsx`，把我们所需要的所有插件包含在里面。

如果你想添加自己的插件，你可以使用我们的基础镜像创建一个你自己的 Dockerfile 和镜像，如下所示：

```
1. # Dockerfile for adding plugins to Jenkins X
2. FROM jenkinsxio/jenkinsx:latest
3.
4. COPY plugins.txt /usr/share/jenkins/ref/openshift-plugins.txt
5. RUN /usr/local/bin/install-plugins.sh < /usr/share/jenkins/ref/openshift-plugins.txt
```

然后以下面的形式将你所有自定义插件放到 `plugins.txt` :

```
1. myplugin:1.2.3
2. anotherplugin:4.5.6
```

一旦你通过 CI/CD 构建和发布了你的镜像，你就可以在安装 Jenkins X 时使用它：

为了用你自定义的镜像配置 Jenkins X ，你可以在 `myvalues.yaml` 文件中指定你的 Jenkins 镜像：

```
1. jenkins:
2.   Master:
3.     Image: "acme/my-jenkinsx"
4.     ImageTag: "1.2.3"
```

这里有一个开源项目的例子 [jenkins-x/jenkins-x-openshift-image](#)，你可以以它为模板创建一个新的 Jenkins 镜像用来在 OpenShift 上使用 Jenkins X 时增加 OpenShift 特定的插件和配置。

## Docker Registry

We try and use the best defaults for each platform for the Docker Registry; e.g. using ECR on AWS.

然而，你也可以在执行命令 `jx create cluster` 或 `jx install` 时，通过选项 `--docker-registry` 来指定。

例如：

```
1. jx create cluster gke --docker-registry eu.gcr.io
```

但是，如果你使用了不同的 Docker Registry 的话，你可能需要修改 secret 才能连接到 docker。

## 参照

- [安装过程中发生了什么](#)
- [在 Kubernetes 上安装](#)
- [下一步？](#)
- [通过 GitOps 管理](#)
- [创建自定义 Builder](#)

- [研发](#)
- [创建 Spring Boot](#)
- [快速开始](#)
- [导入](#)
- [浏览](#)
- [Kubernetes 上下文](#)
- [升级](#)
- [预览](#)
- [IDE](#)
- [问题](#)
- [Git 服务器](#)

# 研发

---

如何使用 Jenkins X 产生持续交付价值。

# 创建 Spring Boot

如何创建Spring Boot应用并导入Jenkins X

如果你在开发基于Java的微服务，那么，你可能正在用流行的[Spring Boot](#)。

你可以利用[Spring Boot Initializr](#)创建Spring Boot应用，然后通过执行命令 `jx import` 来导入Jenkins X。

然而，另外一个快速自动化的方式，是通过执行 `jx create spring` 命令实现：

```
1. $ jx create spring -d web -d actuator
```

参数 `-d` 允许你指定希望添加到 Spring Boot 应用中的依赖。

我们强烈建议你总是包括依赖 **actuator** 到你的 Spring Boot 应用中，它可以为 [Liveness and Readiness probes](#) 提供健康检查。

命令 `jx create spring` 的步骤如下：

- 在子目录中创建一个新的 Spring Boot 应用
- 把你的源码加入到git库中
- 在 git 服务，例如 [GitHub](#), 添加 git 远程库
- 推送代码到 git 远程库
- 添加默认的文件：
  - `Dockerfile` 把你的应用构建为 docker 镜像
  - `Jenkinsfile` 实现 CI / CD 流水线
  - 在 Kubernetes 中通过 helm chart 运行你的应用
- 为你的 Jenkins 在 git 远程库上注册 webhook
- 为你的 Jenkins 添加 git 库
- 首次触发流水线

# 快速开始

如何创建快速开始应用并导入 Jenkins X

你可以由预制的应用开始一个项目，而不是从头开始。

你可以通过命令 `jx create quickstart`，从我们预制的快速应用列表中创建一个新的应用。

```
1. $ jx create quickstart
```

然后，根据列表选择一个。

如果你清楚列表中你所需要的语言，可以进行如下过滤：

```
1. $ jx create quickstart -l go
```

或者使用文本过滤器对项目名称做过滤：

```
1. $ jx create quickstart -f http
```

## 当你选择快速开始时的细节

一旦你选择项目并命名后，下面的步骤会自动完成：

- 在子目录中创建应用
- 把你的代码添加到 git 库中
- 在 git 服务上添加远程库，例如：[GitHub](#)
- 推送代码到远程库
- 添加默认文件：
  - `Dockerfile` to build your application as a docker image
  - `Dockerfile` 把你的应用构建为 docker 镜像
  - `Jenkinsfile` to implement the CI / CD pipeline
  - `Jenkinsfile` 实现 CI / CD 流水线
  - 在 Kubernetes 中通过 helm chart 运行你的应用
- 为你的 Jenkins 在 git 远程库上注册 webhook
- 为你的 Jenkins 添加 git 库
- 首次触发流水线

## 快速开始的原理？

快速开始的源码托管在 [the jenkins-quickstarts GitHub organisation](#)。

当你创建完成后，我们根据工程源码的语言，使用 [Jenkins X build packs](#) 来匹配最合适的构建。

当你使用 `jx create`，`jx install` 或者 `jx init` 时，[Jenkins X build packs](#) 会克隆到目录

`~/jx/draft/packs` 中。

例如：你可以通过下面命令查看支持的所有语言：

```
1. ls -al ~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs
```

你可以使用 `jx create spring` 或 `jx import` 来快速创建，这时 `Jenkins X build packs` 会进行下面的步骤：

- 找到对应的语言包。当前包括 [list of language packs](#)。
- 当文件不存在时，语言包会实现默认的：
  - `Dockerfile` 将程序打包为 docker 镜像
  - `Jenkinsfile` 使用申明式流水线 (pipeline) 实现持续构建、持续部署
  - Helm Charts 在 Kubernetes 上部署程序，并且实现 [预发环境](#)

## 添加你自己的快速开始

如果你想要提交一个新的快速开始给 Jenkins X，请把你 GitHub 中的链接[提交问题](#) 到[快速开始组织](#)，然后它就会出现在菜单 `jx create quickstart` 中。

或者，你是开源项目的一份子，希望管理一套你们项目的快速开始；你可以[提交问题](#)，把你们的GitHub组织详细信息给我们，然后我们会它作为默认的组织添加到命令 `jx create quickstart` 中。如果你把快速开始作为一个单独的 GitHub 组织来维护的话，对于 `jx create quickstart` 会更容易些。

在我们完成这些事情之前，你还是可以在命令 `jx create quickstart` 中通过参数 `-g` 或 `-organisations` 来实现。

```
1. $ jx create quickstart -l go --organisations my-github-org
```

在 `my-github-org` 中可以找到所有 Jenkins X 需要的快速开始。



# 导入

如何把已经存在的项目导入 Jenkins X

如果你已经有一些源码，希望导入 Jenkins X，你可以使用 `jx import` 命令。

```
1. $ cd my-cool-app
2. $ jx import
```

导入将会执行下面的动作（提示你按照这个方法来）：

- 如果你的源码还不在 git 库中，添加进去
- 在给定的 git 服务上创建一个远程库，例如 [GitHub](#)
- 把你的代码推送到远程 git 服务
- 添加任何需要的文件到你的工程中，如果不存在的话：
  - `Dockerfile` 把你的应用作为 docker 镜像进行构建
  - `Jenkinsfile` 实现持续集成、持续构建流水线
  - helm chart 让你的应用在 Kubernetes 中运行
- 为你们团队的 Jenkins 注册一个 webhook 到远程 git 仓库
- 为你们团队的 Jenkins 添加这个 git 仓库
- 首次触发流水线

## 避免 docker + helm

如果你正在导入的仓库而不需要创建 docker 镜像，你可以使用命令参数 `-no-draft`，就不会使用 Draft 默认的 Dockerfile 和 helm chart。

## 通过 URL 导入

如果你希望导入的工程已经在 git 远程库中，那么，你可以使用参数 `-url`：

```
1. jx import --url https://github.com/jenkins-x/spring-boot-web-example.git
```

## 导入 GitHub 项目

如果你希望从 GitHub 组织中导入，可以使用：

```
1. jx import --github --org myname
```

将会提示你需要导入的库。使用光标和空格键来选择（取消）要导入的库。

如果你希望默认导入所有的库（那么反选你不想要的）添加 `-all`：

```
1. jx import --github --org myname --all
```

导入

为了过滤列表，你可以添加参数 `-filter`

```
1. jx import --github --org myname --all --filter foo
```

# 浏览

浏览 Jenkins X 中的资源

如果你之前用过 Kubernetes，你可能使用过 `kubectl` 命令查看 Kubernetes 资源：

```
1. kubectl get pods
```

Jenkins X 的命令行工具，`jx`，和 `kubectl` 看起来相似，并且可以让你看到所有的 Jenkins X 资源。

## 查看 Jenkins 控制台

如果你熟悉 Jenkins 控制台，那么你可以使用 `jx console`：

```
1. jx console
```

就会打开一个浏览器。

## 流水线

要查看当前流水线使用 `jx get pipelines`：

```
1. jx get pipelines
```

## 流水线构建日志

通过 `jx get build logs` 查看当前流水线构建日志：

```
1. jx get build logs
```

你当前看到的是所有能看到的流水线。

你可以通过下面快速过滤

```
1. jx get build logs -f myapp
```

或者，你希望指定

```
1. jx get build logs myorg/myapp/master
```

## 流水线活动

为了查看当前流水线的活动 `jx get activities`：

```
1. jx get activities
```

如果你想要观察你的应用 `myapp`，你可以使用：

```
1. jx get activities -f myapp -w
```

这样将会观察流水线的活动，并无论任何重要的改变发生（例如：发版完成，一个 PR 被创建开始[升级](#) 等等）都会更新屏幕。

## 应用程序

为了查看你的团队所有环境的所有应用的URL和 pod 数量，使用 `jx get applications`：

```
1. jx get applications
```

如果你想要隐藏 URL 或者 pod 数量，你可以使用 `u` 或 `-p`。例如：为了隐藏 URL：

```
1. jx get applications -u
```

或者隐藏 pod 数量：

```
1. jx get applications -p
```

你还可以根据环境来过滤应用：

```
1. jx get applications -e staging
```

## 环境

为了查看你们团队中的 [环境](#)，使用 `jx get environments`：

```
1. jx get environments
```

你还可以

- 通过 `jx create environment` 创建一个新的环境
- 通过 `jx edit environment` 编辑环境
- 通过 `jx delete environment` 删除环境

# Kubernetes 上下文

处理 Kubernetes 上下文

Kubernetes 命令行工具 `kubectl` 通过本地文件 `~/.kube/config`（会在 `$KUBECONFIG` 的文件）记录你使用的 Kubernetes 集群和命名空间。

如果你想要改变命名空间，你可以使用 `kubectl` 命令行：

```
1. kubectl config set-context `kubectl config current-context` --namespace=foo
```

然而 `jx` 还提供了很多有用的命令，用来改变集群、命名空间或环境：

## 切换环境

使用 `jx environment` 来切换 环境

```
1. jx environment
```

你将会看到当前团队的环境列表。使用方向键和回车来选择你想要切换的环境。或者按下 `Ctrl+C` 终止，不切换环境。

或者，如果你知道想要切换的环境，可以直接把它作为参数：

```
1. jx env staging
```

## 切换命名空间

使用 `jx namespace` 在 Kubernetes 不同的命名空间之间进行切换。

```
1. jx namespace
```

你会看到 Kubernetes 集群中所有命名空间的列表。使用方向键和回车选择你想要切换的。或者，按下 `Ctrl+C` 中断，不切换命名空间。

或者，如果你知道想要切换的 Kubernetes 命名空间，可以直接把它作为参数：

```
1. jx ns jx-production
```

## 切换集群

使用 `jx context` 在不同的 Kubernetes 集群（或者上下文）之间切换。

```
1. jx context
```

你会得到当前机器上所有上下文的列表。使用方向键或者回车选择你想要切换的。或者，按下 `Ctrl+C` 中断，不切换集群。

或者，如果你知道想要切换的 Kubernetes 集群，可以直接把它作为参数：

```
1. jx ctx gke_jenkinsx-dev_europe-west2-a_myuserid-foo
2. jx ctx minikube
```

## 本地变化

当前你通过 `kubectl` 切换 Kubernetes 的命名空间或上下文，或上面提到的命令，那么 Kubernetes 会把 你所有的终端 都进行切换，因为它更新的是共享文件（`~/.kube/config` 或 `$KUBECONFIG`）。

这样很方便——但有时候会有危险。例如：如果你想要在生产集群上做一些事情；但是，忘记了，然后在另外一个终端上执行命令要删除你的开发命名空间上所有的 pod——但是你忘记来刚刚切换到来生产命名空间上！

因此，如果通过一个 shell 命令来切换 Kubernetes 上下文或命名空间，有时候是很有帮助的。例如：如果你总是想要看一下集群中的生产环境，就只在那个 shell 中使用那个集群，这样可以减少事故。

你可以使用命令 `jx shell` 提示你选择不同的 Kubernetes 上下文，例如：`jx context` 命令。然而，这样切换命名空间或集群就只能在当前 shell 中有效！

还有 `jx shell` 通过 `jx prompt` 自动更新你的命令提示符，这样使得你的 shell 很清楚上下文或命名空间的修改。

## 定制你的 shell

你可以使用 `jx prompt` 把当前 Kubernetes 集群和命名空间添加到你的终端提示符中。

要为 `jx` 命令 添加命令自动补充，尝试 `jx 自动补充` 。

# 升级

升级你的应用新版本到环境

Jenkins X 的升级策略配置为 `Auto` 时，持续部署流水线通过配置好的环境来自动化升级版本。默认情况下，`Staging` 环境使用自动升级，`生产` 环境使用 `手动` 升级。

要手动升级应用的一个版本到特定环境上，可以使用命令 `jx promote`。

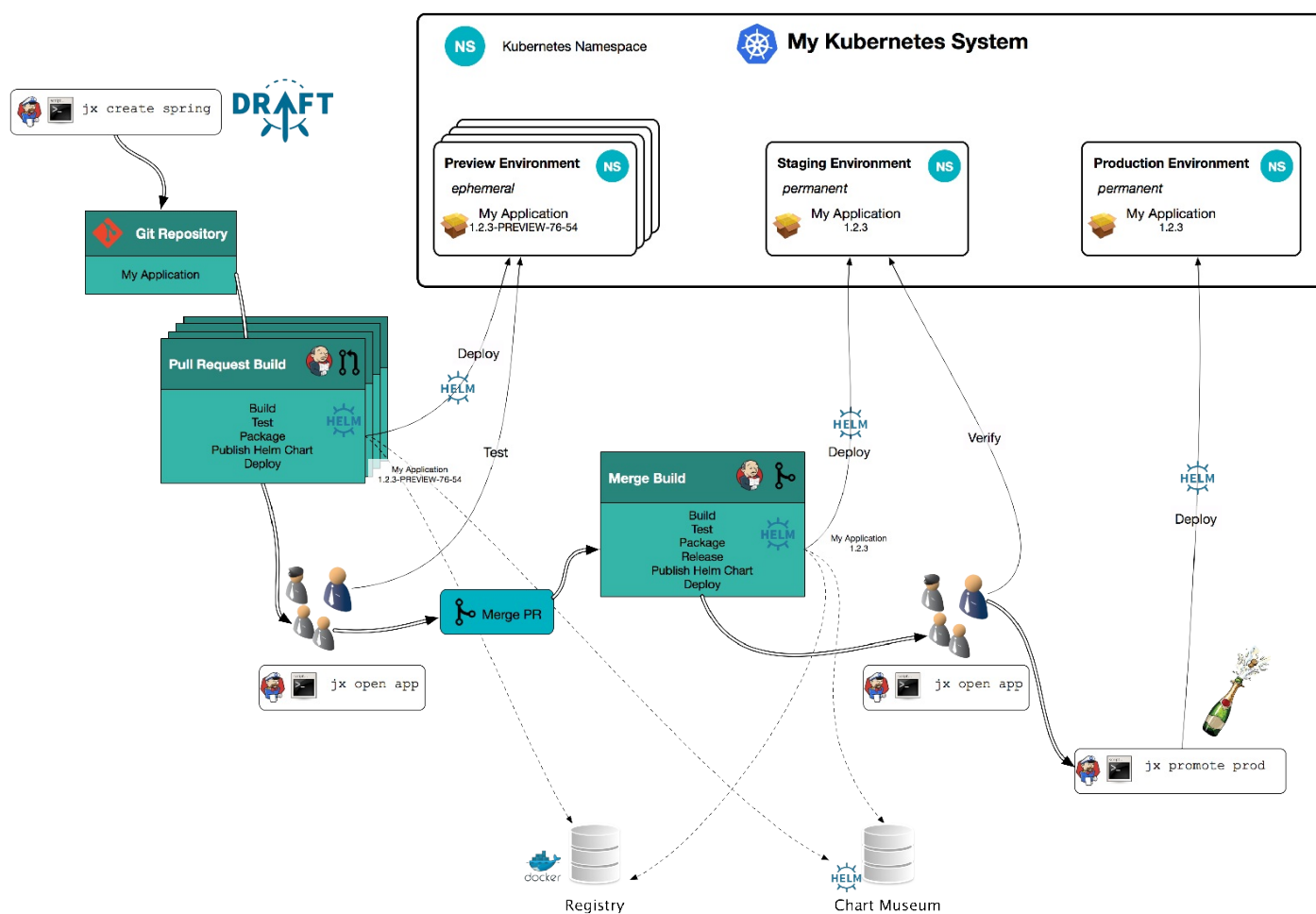
```
1. jx promote myapp --version 1.2.3 --env production
```

该命令会等待升级完成，并记录过程的详细信息。你可以通过参数 `--timeout` 为升级等待设置超时时间。

例如：等待5小时

```
1. jx promote myapp --version 1.2.3 --env production --timeout 5h
```

你可以使用类似 `20m` 或 `10h30m` 这样的时间表达式。



## 反馈

如果提交注释中引用了问题（例如：通过文本 `fixes #123`），那么，Jenkins X 流水线会自动生成类似 `jx 发`

升级

布 的发布记录。

同样的，升级到 **Staging** 或 **生产** 环境中的提交日志中也会自动关联每个修复的问题，包括有发布日志和应用所运行环境的链接。



jstrachan commented 4 days ago

Owner



the fix for this issue is now deployed to **Staging** in version **0.0.2** and available at <http://jx-staging-my-spring-boot10.jx-staging.35.189.107.104.nip.io>



# 预览

在变更合并到 master 之前预览 Pull Requests

我们强烈建议使用 [预览环境](#)，使得在变更合并到 master 之前尽快地得到反馈。

通常，预览环境是由 Jenkins X 的流水线中自动创建的。

然而，你可以使用 `jx` 通过命令 `jx preview` 手动创建一个[预览环境](#)。

```
1. jx preview
```

## 创建预览环境时都做了什么

- 一个新的 [环境](#)，例如 [预览](#) 被创建时，一个 [kubernetes 命名空间](#) 会在 `jx get environments` 出现，使用 `jx 环境`和 `jx 命名空间命令` 你可以看到那个预览环境是活跃的，并可以进入查看。
- Pull Request 会作为预览 Docker 镜像和 chart 构建，并被部署到预览环境中
- 添加一条注释到 Pull Request 中，让你们团队知道该预览应用已经准备好可以测试了，并带有打开应用的链接。因此，只要点击一下就可以让你们团队成员体验预览环境！



rawlingsj commented 17 hours ago

Owner



PR built and available in a preview environment [rawlingsj-node3-pr-2 here](#)

# IDE

---

在你的 IDE 中使用 Jenkins X

作为开发人员，我们经常在 IDE 上花大量的时间来编码。Jenkins X 完全是为了帮助开发人员快速交付商业价值的，因此，我们希望使得 Jenkins X 在你的 IDE 中更加容易使用。

因此，我们有 IDE 插件来方便使用 Jenkins X。

## VS Code

---

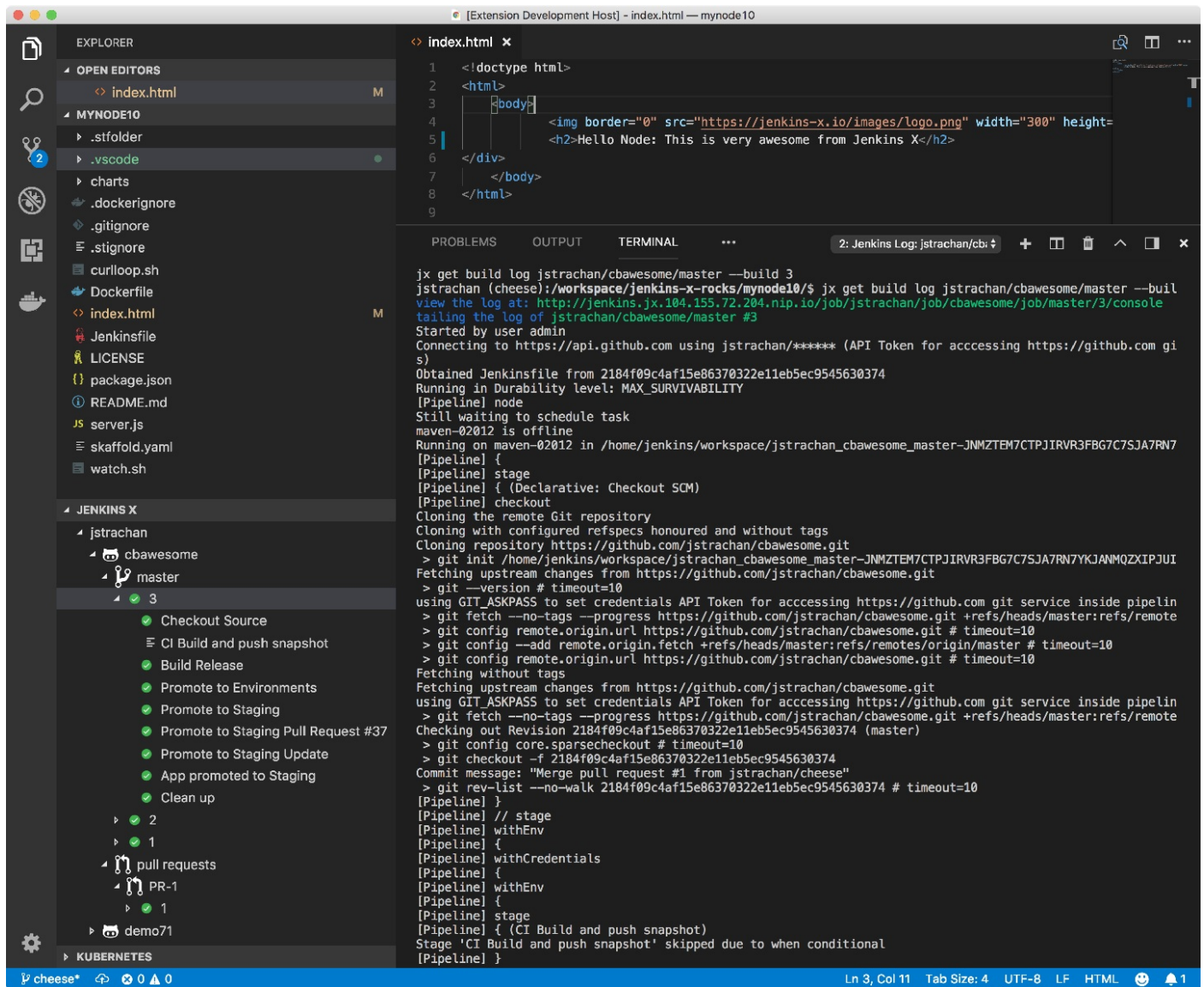
VS Code 是一个流行的来自微软的开源 IDE。

我们已经为 VS Code 研发了插件 `vscode-jx-tools`。

你可以在 `扩展` 窗口把插件安装到 VS Code，搜索 `jx` 应该能查到这个扩展。

安装完后点击 `重新加载`，你应该就能使用了。

如果你展开 `JENKINS X` 导航窗口，应该能看到你创建工程的实时更新界面，还有 Pull Request 被创建或者代码被合并到了 master。



## 特色

- 浏览你所在团队的所有流水线的实时更新，包括发布或者 Pull Request 流水线的开始和结束
- 在 VS Code 终端内打开流水线构建日志
- 轻松地浏览 Jenkins 流水线页面、git 仓库、构建日志或者应用
  - Jenkins X 浏览器的右键点击
  - 还有启动（停止）流水线！
- 通过一个命令打开 DevPods，保持源码与云上的相同容器镜像和 pod 模板同步

# 问题

## 问题处理

Jenkins X 默认使用你的 git 提供商中的问题跟踪系统来创建和浏览问题。

例如：如果你在 GitHub 项目中的源码中，那么你可以输入 `jx create issue`：

```
1. jx create issue -t "lets make things more awesome"
```

一个新的问题就会在 GitHub 上被创建。

你可以在你的项目上通过 `jx get issues` 列出打开的问题：

```
1. jx get issues
```

## 使用不同的问题跟踪

如果你希望在项目中使用 JIRA，你首先需要添加一个 JIRA 服务。

你可以通过 `jx create tracker server` 注册你的 JIRA服务：

```
1. jx create tracker server jira https://mycompany.atlassian.net/
```

然后，你可以通过 `jx get tracker` 来查看你的问题追踪了：

```
1. jx get tracker
```

然后，通过下面添加一个用户和 token：

```
1. jx create tracker token -n jira myEmailAddress
```

## 配置项目的问题跟踪

在你项目的源码中使用 `jx edit config`：

```
1. jx edit config -k issues
```

然后

- 如果你有多问题跟踪系统，选择一个用于当前项目
- 在问题跟踪系统中输入项目名称（例如：大写的 JIRA 项目名称）在你的项目中一个叫做 `jenkins-x.xml` 的文件会被修改，这个文件应该被加到你的 git 库中。

# Git 服务器

使用不同的 Git 服务器

Jenkins X 默认使用 [GitHub](#)，用于开源项目的免费公共 git 托管方案。

然而，在企业中工作时，你可能希望使用不同的 git 服务器。

你可以通过 `jx get git` 列出配置好的 git 服务器。

```
1. jx get git
```

## 添加一个新的 git 服务商

如果你在某个地方已经有了一个 git 服务，你可以通过 `jx create git server` 把它添加到 Jenkins X中：

```
1. jx create git server gitKind someURL
```

这里 `gitKind` 是某个 git 服务商，像 `github, gitea, gitlab, bitbucket`

## 企业 GitHub

要添加一个企业 GitHub 服务，尝试：

```
1. jx create git server github https://github.foo.com -n GHE
```

这里 `-n` 是 git 服务的名称。

## BitBucket

要添加 BitBucket ，尝试：

```
1. jx create git server bitbucket -n BitBucket https://bitbucket.org
```

## 添加用户 tokens

为了添加一个 git 服务，你需要通过 `jx create git token` 添加一个用户名和 API token：

```
1. jx create git token -n myProviderName myUserName
```

然后，就会提示你输入 API token

## Kubernetes 托管的 git 服务

你可以安装 git 服务到运行 Jenkins X 的 Kubernetes 集群中。

例如：有一个 [gitea](#) 的插件，可以让你把 gitea 作为 Jenkins X 安装的一部分。

要在 Jenkins X 中使用 [gitea](#)，你需要在安装 Jenkins X 之前启用 [gitea](#) 插件：

```
1. jx edit addon gitea -e true
```

你可以通过 `jx get addons` 查看启用的插件：

```
1. jx get addons
```

现在，当你 [安装 Jenkins X](#) 时，也会安装 [gitea](#) 插件。

无论什么时候，Jenkins X 需要为一个环境或者新项目创建一个 git 库时，gitea 服务都会出现在选择列表中。

### gitea 已知的限制

在写本文时，[gitea plugin for Jenkins](#) 不能够正确地更新 Pull Request 和 git 提交构建状态，这会打断 GitOps 升级流水线。可以手工审核来升级；但是，流水线会报告失败。

另一个问题是，由 [jx](#) 在 [gitea](#) 创建的新项目，无法使得 [在 Pull Requests 中合并按钮可用](#)。要使得可用的话，当一个项目在 GitHub 中创建后，你到仓库的 [Settings](#) 页面，在 [gitea](#) 的 web 控制台中，启用合并按钮。

示例

- [创建 GKE 集群](#)
- [创建 Spring](#)
- [创建集群](#)

# 创建 GKE 集群

---

如何在 GKE 上创建 Kubernetes 集群并安装 Jenkins X

该 [示例](#) 使用命令 `jx create cluster gke` 来 [创建一个新的 Kubernetes 集群](#):



# 创建 Spring

---

如何创建一个新的带有持续集成、持续部署的 Spring Boot 应用以及 GitOps 提升

该 [示例](#) 使用命令 `jx create spring` 来 [创建带有持续集成、部署流水线的 Spring Boot 应用](#) 并使用 [GitOps 提升](#):

# 创建集群

---

如何创建 Kubernetes 集群并安装 Jenkins X

该 [示例](#) 使用命令并行地 [创建了一个新的 Kubernetes 集群](#)：

- `jx create cluster gke`
- `jx create cluster aks`
- `jx create cluster minikube`

- [架构](#)
- [组件](#)
- [源码](#)
- [自定义资源](#)
- [构建打包](#)
- [Pod 模板](#)
- [Docker Registry](#)

# 架构

---

# 组件

典型 Jenkins X 安装中的组件概览

Jenkins X 安装的包括：

- 每个团队一个开发环境，也就是 **kubernetes 命名空间**
- 零或多个其它 **永久环境**
  - 为每个团队获取各自开箱即用的 **Staging** 和 **生产** 环境
  - 每个团队可以按照需要有很多环境，并依据习惯命名
- 可选的 **预览环境**通常，每个环境会关联对应不同的 **kubernetes 命名空间**，以确保环境之间干净隔离。

尽管从技术上讲，两个团队可以共享同样的基础命名空间，比如 **Staging**，尽管我们建议分开以保持简单——不然的话，在同一个 git 库中配置相同的命名空间可能会发生冲突；例如：服务资源名称或者 DNS 会冲突。如果你希望两个团队共享基础微服务，使用 **服务链接** 在一个命名空间中连接另外一个会比较简单，这样它们可以通过本地 DNS 以本地服务的形式出现。

## 开发环境

在开发环境中，我们安装了很多必要的最小核心应用，才能启动基于 Kubernetes 的 CI/CD。

我们还支持 **addons** 扩展核心套件。

Jenkins X 的配置把这些服务连接起来，就可以直接工作了。这样就神奇地把 Kubernetes 的密码、环境遍历和配置文件全部配置好并可以工作了。

- **Jenkins** —提供 CI 和 CD 自动化。这样对 Jenkins 的解耦变得更加云原生，并利用 Kubernetes 的概念，例如：CRDs（自定义资源定义）、存储和伸缩。
- **Nexus**—作为 Nodejs 和 Java 程序的依赖缓存，可以减少构建时间。一个 SpringBoot 应用的初始化构建后，构建时间能从12分钟减少到4分钟。我们还没有实现，但是已经计划很快用 Artifactory 来替换。
- **Docker registry** —一个集群中的 docker 注册表（registry），用于我们的流水线推送应用的镜像，我们将很快使用原生的云提供商，例如：Google Container Registry, Azure Container Registry 或 Amazon Elastic Container Registry (ECR)。
- **ChartMuseum**—一个发布 Helm charts 的 registry
- **Monocular** —一个用于发现和运行 Helm charts 的 UI

## 永久环境

这些**环境**，像 **Staging** 和 **Production** 使用 GitOps 来管理他们，因此，每个都有一个包含配置所有应用和服务以及要部署的位置信息的源码的 git 仓库。

通常，我们使用 git 仓库中的 Helm charts 来定义哪些 charts 要被安装，它们的版本，环境的具体配置，以及附加资源（例如：Secrets 或 像 Prometheus 可运行的应用等）

## 预览环境

[预览环境](#) 和[永久环境](#) 类似，都在源码中使用 Helm charts 定义。

主要的不同之处，是预览环境配置在应用的源码的 `./chart/preview` 目录中。

而且，它们不是固定的，而是由一个应用的 git 仓库的 Pull Request 创建，而且后面会被删除（手动或者通过垃圾回收自动）。

# 源码

---

多个源码仓库的位置

Jenkins X 建立在巨人的肩膀上，并且拥有许多不同的源码仓库，从 CLI 工具、Docker 镜像、Helm 图表到[插件应用](#)来做各种各样的事情。

这个页面列出了主要的组织和仓库。

## 组织

---

- [jenkins-x](#) 源码的主要组织
- [jenkins-x-apps](#) 包括 Jenkins X 的标准[插件应用](#)
- [jenkins-x-buildpacks](#) 包括可用的[构建打包](#)
- [jenkins-x-charts](#) 我们分发的主要 helm 图表
- [jenkins-x-images](#) 包括一些自定义的 docker 镜像构建
- [jenkins-x-quickstarts](#) 通过[创建快速开始](#)使用的快速开始项目
- [jenkins-x-test-projects](#) 我们在测试用例中使用的项目

## 仓库

---

在这里我们列出上面组织的一些主要仓库

- [jenkins-x/jx](#) 创建 [jx](#) CLI 和可重用的流水线步骤的主要仓库
- [jenkins-x/jx-docs](#) 基于 Hugo 的文档，用来生成网站
- [jenkins-x/bdd-jx](#) 我们用来验证平台变更以及用来验证 [jenkins-x/jx](#) 上 PR 的 BDD 测试
- [jenkins-x/jenkins-x-platform](#) Jenkins X 平台主要合成物的 helm 图表
- [jenkins-x/jenkins-x-versions](#) 包括[版本流](#) - 所有 图表 和 CLI 包 的稳定版本
- [jenkins-x/cloud-environments](#) 不同 cloud providers 的 helm 配置

## 构建 pods 和 镜像

- [jenkins-x/jenkins-x-builders](#) 生成静态 jenkins 服务的构建 pod 和 docker 镜像
- [jenkins-x/jenkins-x-image](#) 为我们默认使用的静态 jenkins 服务器生成 docker 镜像
- [jenkins-x/jenkins-x-serverless](#) 当使用 [prow](#) 时生成 [serverless jenkins](#) docker 镜像

## 工具

- [jenkins-x/exposecontroller](#) 用来生成或更新 [Ingress](#) 资源（或 OpenShift 中的 [Route](#)）的 [Deployment](#) 或 [Job](#)。如果你修改了你的 DNS 域或开启了 TLS，它可以通过 [ConfigMap](#) 注入用来注入外部 URLs 到你的应用中。
- [jenkins-x/updatebot](#) 一个我们用来为库、可执行文件、图表和镜像执行持续交付的命令行机器人。例如：当一个新的上游发布完成后，我们在下游依赖的 git 仓库中生成 Pull Requests。

# 自定义资源

由 Jenkins X 自定的资源

Kubernetes 提供了一个叫做[自定义资源](#)的扩展机制，它允许微服务扩展 Kubernetes 平台来解决更高级的问题。

因此，在 Jenkins X 中定义了若干个自定义资源来扩展 Kubernetes 支持 CI/CD：

## 环境

Jenkins X 原生地支持[环境](#)，允许为你们团队定义环境，并通过 `jx get environments` 查询：

```
1. jx get environments
```

以下的命令都使用 Kubernetes 自定义资源 [环境](#)。

因此，你还可以通过 `kubectl` 查询环境：

```
1. kubectl get environments
```

或者你想要通过 [YAML](#) 直接编辑它们的话：

```
1. kubectl edit env staging
```

尽管，你使用命令 `jx edit environment` 会更容易。

## 发版

Jenkins X 流水线生成了一个自定义资源 [发版](#)，我们可以用来跟踪：

- 版本、git 标签、git 地址映射到 Kubernetes/Helm 中的发版
- Jenkins 流水线地址和执行日志用于执行发布
- 提交日志、问题和 Pull Requests 是每次发版的一部分，因此我们可以实现在 [Staging/生产环境中修复的问题反馈](#)

## 流水线活动

该资源保存了基于 Jenkins 流水线阶段以及 [升级活动](#) 的流水线状态

该资源还会被命令 `jx get activities` 用到



# 构建打包

打包源码为 `kubernetes` 应用

我们使用 `draft` 风格为不同的语言构建打包，我们通过[导入](#)或者[创建他们](#)，运行时和构建工具添加必要的配置文件，因此我们可以在 `Kubernetes` 中构建和部署他们。

如果由于工程没有被创建或导入而不存在的话，构建包会默认使用下面的文件：

- `Dockerfile` 把代码构建为不可变的 `docker` 镜像，准备在 `Kubernetes` 中运行
- `Jenkinsfile` 为应用使用申明式 `Jenkins` 流水线定义 `CI/CD` 步骤
- `helm chart` 在文件夹 `charts` 中生成可以在 `Kubernetes` 中运行的 `Kubernetes` 资源
- 在 `charts/preview` 文件夹中的 `preview chart` 定义了基于 `Pull Request` 部署一个[预览环境](#)的所有依赖默认的构建包在 <https://github.com/jenkins-x-buildpacks/jenkins-x-kubernetes>，每个语言或者构建工具在一个文件夹中。

`jx` 命令行克隆构建包到你的文件夹 `./~/.jx/draft/packs/`，并在你每次尝试创建或者到一个工程时通过 `git pull` 来更新他们。

## 创建新的构建

我们欢迎[贡献](#)，因此，请考虑增加新的构建包和 `pod` 模板。

这里是如何创建一个新的构建包的指导 — 如果有任何不清楚的请[加入社区并提问](#)，我们很乐意帮助！

最好的开始就是 `快速开始` 应用。你可以当作一个测试的样例工程。因此，创建或查找一个合适的例子工程，然后[导入](#)。

然后，如果不存在的话，手动添加 `Dockerfile` 和 `Jenkinsfile`。你可以从[当前构建包文件夹](#)开始 — 使用相似的语言或框架。

如果你的构建包使用了 `pod` 模板中不存在的构建工具，你需要[提交一个新的 pod 模板](#)，还可能需要一个新的构建容器景象。

一旦你有了 `pod` 模板可以使用，例如你的 `Jenkinsfile` 中引用到的 `jenkins-foo`：

```
1. // my declarative Jenkinsfile
2.
3. pipeline {
4.   agent {
5.     label "jenkins-foo"
6.   }
7.   environment {
8.     ...
9.   }
10.  stages {
11.    stage('CI Build and push snapshot') {
12.      steps {
```

```
13.         container('foo') {  
14.             sh "foo deploy"  
15.         }
```

一旦你的 `Jenkinsfile` 可以在你的示例工程为你的语言实现 CI/CD 的话，我们因该把 `Dockerfile` , `Jenkinsfile` 和 `charts` 文件夹拷贝到你的派生 `jenkins-x/draft-packs` 仓库 中。

你可以通过把他们添加到构建包的本地库 `~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs` 中来尝试。

例如：

```
1. export PACK="foo"  
2. mkdir ~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs/$PACK  
3. cp Dockerfile Jenkinsfile ~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs/$PACK  
4.  
5. # the charts will be in some folder charts/somefoo  
6. cp -r charts/somefoo ~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs/$PACK/charts
```

当你的构建包在 `~/.jx/draft/packs/github.com/jenkins-x/draft-packs/packs/` 文件夹中，就可以通过命令 `jx import` 来导入工程，使用编程语言来检测并查找最合适的构建包。如果你的构建包自定义检测逻辑的话，请让我们指导，我们可以帮助改进 `jx import` 使得在你的构建包上做的更好。例如：我们有一些自定义逻辑更好地处理 `maven` 和 `gradle`。

如果你需要任何帮助 [请加入社区](#) 。

# Pod 模板

用于实现 Jenkins 流水线的 Pods

我们使用申明式 (declarative) Jenkins 流水线实现 CI/CD，每个应用或者环境的 git 库源码中有 `Jenkinsfile`。

我们使用 Jenkins 的 `kubernetes` 插件，使得在 Kubernetes 中为每次构建启动一个新的 pod — 感谢 Kubernetes 给了我们一个用于运行流水线的伸缩的代理池。

Kubernetes 插件使用 *pod templates* 定义用于运行 CI/CD 流水线的 pod，包括：

- 一个或多个构建容器，用于运行命令（例如：你的构建工具，像 `mvn` 或 `npm`，还有流水线的其它部分的工具，像 `git`, `jx`, `helm`, `kubect1` 等等
- 永久存储卷
- 环境变量
- 可以写到 git 仓库、docker 注册表、maven/npm/helm 仓库等等的 secret

## 参考 Pod Templates

Jenkins X 带有一套给支持的语言和运行时的默认 pod 模板，在你的 `build packs`中，命名类似于：`jenkins-$PACKNAME`。

例如 `maven build pack` 使用的 pod 模板是 `jenkins-maven`。

然后，我们就可以在 `Jenkinsfile` 中引用 pod 模板名称，在申明式流水线中使用这样的语法 `agent { label "jenkins-$PACKNAME" }`，例如：

```
1. // my declarative Jenkinsfile
2.
3. pipeline {
4.     agent {
5.         label "jenkins-maven"
6.     }
7.     environment {
8.         ...
9.     }
10.    stages {
11.        stage('CI Build and push snapshot') {
12.            steps {
13.                container('maven') {
14.                    sh "mvn deploy"
15.                }
16.                ...
```

## 提交新的 Pod 模板

如果你正在使用一个新的 `build pack`，那么，我们欢迎你 [提交](#) 一个新的 pod 模板，而且我们可以把它包含在 Jenkins X 的发行版中！

现在遵循如何这个的指示 — 如果有任何不清楚的话请[加入社区并提问](#)，我们很高兴帮助你！

为了提交一个新的 `build pack`：

- 派生 `jenkins-x-platform` 库
- 增加你的 `build pack` 到 `jenkins-x-platform` 库中的 `values.yaml` 文件里 在 YAML 文件的 `jenkins.Agent.PodTemplates` 这个区域
- 你甬根想要从复制、粘贴开始大多数相似已经存在 pod 模板（例如：拷贝 `Maven`，如果你使用基于 Java 的构建pod），并且，只是配置名称、标签和 `Image` 等等。
- 现在到 `jenkins-x-platform` 库为你的 pod 模板提交一个 Pull Request

## 构建容器

当使用 pod 模板和 Jenkins 流水线时，每个工具你可以用很多不同的容器。例如：`maven` 容器和 `git` 等。

我们发现，在一个构建容器里有所有通用的工具会比较简单。这也意味着你可以使用 `kubect1 exec` 或 `jx rsh` 打开一个构建 pod 的 shell，当你调试、诊断有问题的流水线时里面有所有需要的工具。

因此，我们有一个 `builder-base` 的 docker 镜像，[包含所有不同的工具](#)，我们倾向于在 CI/CD 流水线中使用像 `jx`, `skaffold`, `helm`, `git`, `updatebot` 的工具。

如果想要在你新的 pod 模板中使用单一的构建惊喜那个，那么，你可以使用 `builder base` 作为基础增加你自定义的工具。

例如：`builder-maven` 使用一个 `Dockerfile` 引用基础构建。

因此，最简单的就是拷贝一个简单的 builder — 像 `builder-maven`，然后编辑 `Dockerfile` 增加你需要的构建工具。

我们欢迎 Pull Requests 和[贡献](#)，因此，请把你新的构建容器和 Pod 模板提交，我们很乐意[帮助](#)！

## 增加你自己的 Pod 模板

为了保持简单，我们倾向于在 Jenkins 配置中定义 pod 模板，然后在 `Jenkinsfile` 中通过名称来引用。

尽管一个 pod 模板倾向于有很多开发环境定义在里面，像 `secrets`；如果需要的话，你可以尝试在 `Jenkinsfile` 中用内联的形式定义 pod 模板，使变得简单。但我们更喜欢把大多数 pod 模板保留在你的开发环境源码中，而不是在每个应用中拷贝、粘贴。

现在，添加新的 Pod 模板最简单的方式就是通过 Jenkins 控制台。例如：

```
1. jx console
```

这样就会打开 Jenkins 控制台。然后，导航到 `管理 Jenkins`（在左侧菜单），然后 `系统配置`。

你将会面临大量的页面配置选项，Pod 模板通常在底部；你应该看到了当前所有的 pod 模板，像 maven、nodejs 等等。

你可以在那个页面编辑、增加、移除 pod 模板并点击保存。

注意，长期来说，尽管我们希望通过 [GitOps](#) 维护你的开发环境，就像是我们做的 [Staging](#) 和 [Production](#) — 也就意味着当你[升级你的开发环境](#)通过 Jenkins 界面做的修改可能会丢失。

因此，我们希望把 Pod 模板添加到你的开发环境 git 库的 `values.yaml` 文件中，就像我们在 [jenkins-x-platform chart](#) 做的一样。

如果你正在使用开源工具创建 pod 模板，那么在 [Pull Request](#) 中提交你的 pod 模板会比较简单，我们可以把它添加到 Jenkins X 未来的发行版中？

# Docker Registry

## 配置你的 docker registry

为了能够创建和发布 docker 镜像，我们需要使用 Docker Registry。

默认，Jenkins X 会在系统命名空间中带一个 Docker Registry，以及 Jenkins 和 Nexus。当 Docker Registry 运行在你的 Kubernetes 集群中，它会在集群内部使用，它很难通过带有自签名证书的 HTTPS 暴露——因此，在你的 Kubernetes 集群服务中，我们默认使用 insecure 的 Docker Registry。

## 使用不同的 Docker Registry

如果你使用公有云的话，可能希望利用你的云服务商的 Docker Registry；或者复用你已有的 Docker Registry。

为了指定 Docker Registry 的主机、端口，你可以使用 Jenkins 控制台：

```
1. jx console
```

然后，定位到 `管理 Jenkins -> 系统配置`，并修改环境变量 `DOCKER_REGISTRY` 指向你选择的 Docker Registry。

另一种方法是，把下面的内容添加到你的自定义 Jenkins X 平台 helm charts 的 `values.yaml` 文件中：

```
1. jenkins:
2.   Servers:
3.     Global:
4.       EnvVars:
5.         DOCKER_REGISTRY: "gcr.io"
```

## 更新 config.json

下一步，你需要为 docker 更新 `config.json` 中的认证。

如果为你的 Docker Registry 创建一个 `config.json` 文件，例如：Google 云的 GCR，它可能看起来像：

```
1. {
2.   "credHelpers": {
3.     "gcr.io": "gcloud",
4.     "us.gcr.io": "gcloud",
5.     "eu.gcr.io": "gcloud",
6.     "asia.gcr.io": "gcloud",
7.     "staging-k8s.gcr.io": "gcloud"
8.   }
9. }
```

对于 AWS 则像：

```
1. {  
2.     "credsStore": "ecr-login"  
3. }
```

然后需要更新凭据 `jenkins-docker-cfg` ，你可以执行以下操作：

```
1. kubectl delete secret jenkins-docker-cfg  
2. kubectl create secret generic jenkins-docker-cfg --from-file=./config.json
```

## 使用 Docker Hub

如果你想要发布你的镜像到 Docker Hub 当中 ，则需要修改你的 `config.json` 像下面那样：

```
1. {  
2.     "auths": {  
3.         "https://index.docker.io/v1/": {  
4.             "auth": "MyDockerHubToken",  
5.             "email": "myemail@acme.com"  
6.         }  
7.     }  
8. }
```

## 为你的 registry 挂载凭证

你的 docker registry 需要将凭证挂载到 Pod 模板当中。

看这里 ==> <https://jenkins-x.io/extending/>



看这里 ==> <https://jenkins-x.io/apidocs/>

# jx

## jx

jx is a command line tool for working with Jenkins X

## Synopsis

jx is a command line tool for working with Jenkins X

```
1. jx [flags]
```

## Options

1. -b, --batch-mode	Runs in batch mode without prompting for user input (default true)
2. --config-file string	Configuration file used for installation
3. -h, --help	help for jx
4. --install-dependencies	Enables automatic dependencies installation when required
5. --no-brew	Disables brew package manager on MacOS when installing binary dependencies
--skip-auth-secrets-merge	Skips merging the secrets from local files with the secrets from Kubernetes
6. cluster	
7. --verbose	Enables verbose output

## SEE ALSO

- [jx add](#) - Adds a new resource
- [jx boot](#) - Boots up Jenkins X in a Kubernetes cluster using GitOps and a Jenkins X Pipeline
- [jx cloudbees](#) - Opens the CloudBees app for Kubernetes for visualising CI/CD and your environments
- [jx completion](#) - Output shell completion code for the given shell (bash or zsh)
- [jx compliance](#) - Run compliance tests against Kubernetes cluster
- [jx console](#) - Opens the Jenkins console
- [jx context](#) - View or change the current Kubernetes context (Kubernetes cluster)
- [jx controller](#) - Runs a controller
- [jx create](#) - Create a new resource
- [jx delete](#) - Deletes one or more resources
- [jx diagnose](#) - Print diagnostic information about the Jenkins X installation
- [jx docs](#) - Open the documentation in a browser
- [jx edit](#) - Edit a resource
- [jx environment](#) - View or change the current environment in the current Kubernetes cluster
- [jx gc](#) - Garbage collects Jenkins X resources

- `jx get` - Display one or more resources
- `jx import` - Imports a local project or Git repository into Jenkins
- `jx init` - Init Jenkins X
- `jx install` - Install Jenkins X in the current Kubernetes cluster
- `jx login` - Onboard an user into the CloudBees application
- `jx logs` - Tails the log of the latest pod for a deployment
- `jx namespace` - View or change the current namespace context in the current Kubernetes cluster
- `jx open` - Open a service in a browser
- `jx options` -
- `jx preview` - Creates or updates a Preview Environment for the current version of an application
- `jx promote` - Promotes a version of an application to an Environment
- `jx prompt` - Generate the command line prompt for the current team and environment
- `jx repository` - Opens the web page for the current Git repository in a browser
- `jx rsh` - Opens a terminal in a pod or runs a command in the pod
- `jx scan` - Perform a scan action
- `jx shell` - Create a sub shell so that changes to the Kubernetes context, namespace or environment remain local to the shell
- `jx start` - Starts a process such as a pipeline
- `jx status` - status of the Kubernetes cluster or named node
- `jx step` - pipeline steps
- `jx stop` - Stops a process such as a pipeline
- `jx sync` - Synchronises your local files to a DevPod
- `jx team` - View or change the current team in the current Kubernetes cluster
- `jx uninstall` - Uninstall the Jenkins X platform
- `jx update` - Updates an existing resource
- `jx upgrade` - Upgrades a resource
- `jx version` - Print the version information

Auto generated by spf13/cobra on 5-Jul-2019

- [常见问题概览](#)
- [FAQ](#)
- [Jenkins 相关问题](#)
- [安装问题](#)
- [Issues](#)
- [技术](#)
- [开发问题](#)

# 常见问题概览

---

## 常见问题解答

---

Jenkins X 常见问题的解决方案。

我们已经试图把一些常见的问题整理到这里。如果你遇到的问题没有在这里列出来，请[让我们知道](#)。

## Jenkins X 是开源的吗？

---

是的！Jenkins X 的所有源码和成品都是开源的；Apache 或 MIT 能保证这一点！

## Jenkins X 和 Jenkins 相比如何呢？

---

Jenkins X 通过[跨环境的 GitOps 部署升级](#)和 [Pull Requests 预发环境](#)为 Kubernetes 中的应用提供了[自动化 CI + CD](#)。更多信息请参考[特性](#)。

Jenkins 是一个通用的 CI/CD 服务器，可以通过添加插件、更改配置和编写自己的流水线来配置它来做你喜欢的任何事情。

对于 Jenkins X 仅仅[安装 Jenkins X](#)，它将自动配置所有各种不同的工具（helm, docker registry, nexus 等等），然后[创建/导入](#)(/zh/developing/import/)，你将获得全面的自动化 CI/CD 和预发环境。这使得当你委托 Jenkins X 管理您的 CI + CD 时，开发人员可以集中精力构建应用程序。

Jenkins X 支持不同的执行引擎；因此它可以通过在 Docker 容器中重用 Jenkins 来为每个团队编排 Jenkins 服务器。然而当使用[无服务 Jenkins X 流水线](#)时，我们使用 Tekton 而不是 Jenkins 作为底层的 CI/CD 引擎来提供一个新式的、高可用的云原生架构。

## Jenkins X 是 Jenkins 的分支吗？

---

不！Jenkins X 可以通过在容器中重用 Jenkins 来编排 Jenkins，并尽可能地用 kubernetes 原生方式配置它。

然而当使用[无服务 Jenkins X 流水线](#)时，我们使用 Tekton 而不是 Jenkins 作为底层的 CI/CD 引擎来提供一个新式的、高可用的云原生架构。

## 为什么要创建一个子项目？

---

我们是 [Kubernetes](#) 和云的超级粉丝，并认为是软件运行的未来趋势。

然而，很多分支仍然想要通过：`java -jar jenkins.war` 以常规的方式来运行 Jenkins。

因此，Jenkins X 子项目的想法，是为了100%关注在 Kubernetes 和云原生使用场景，并让 Jenkins 核心项目关注经典的 Java 方式。

Jenkins 最强大的是它的灵活性和巨大的插件生态。分离 Jenkins X 子项目帮助社区并行地迭代并快速改进云原生和 Jenkins 经典的发行。

## 参照

---

- [技术性问题](#)
- [常见问题解答](#)
- [安装问题](#)
- [开发问题](#)

# Jenkins 相关问题

---

Jenkins 相关问题。

## 密码

---

安装在 Jenkins X 中的 Jenkins 没有提供修改管理员密码的页面，但是你可以通过配置文件 `~/.jx/jenkinsAuth.yaml` 来获取密码。



# 安装问题

---

Jenkins X 的安装和配置问题

## 如何在 Jenkins X 的安装当中添加用户？

---

Jenkins X 假设每个用户都可以访问运行 Jenkins X 的 kubernetes 开发集群。

如果您的用户无权访问 kubernetes 集群，我们需要设置他们的 `~/.kube/config` 文件，以便他们可以访问它。

如果您正在使用 Google 的 GKE，那么您可以浏览 [GKE Console](#) 以查看所有集群，然后单击开发集群旁边的 `Connect` 按钮，然后可以运行复制/粘贴命令以连接到集群。

对于其他集群，我们计划编写一些 [CLI 命令来导出和导入 kube 配置](#)。

## 当用户拥有了 kubernetes 集群的访问权限

当用户拥有了 kubernetes 集群的访问权限：

- [安装 jx 二进制文件](#)如果 Jenkins X 安装在命名空间 `jx` 中，那么应该 [切换你的上下文](#) 到命名空间 `jx` 当中：

```
1. jx ns jx
```

测试安装成功可以输入下列命令：

```
1. jx get env
2. jx open
```

查看环境和任何开发工具，如 Jenkins 或 Nexus 控制台。

## 参照

---

- [技术性问题](#)
- [常见问题解答](#)
- [常见问题解答](#)
- [开发问题](#)

# 常见问题解答

Issues using Jenkins X 常见问题的解决方案。

我们已经试图把一些常见的问题整理到这里。如果你遇到的问题没有在这里列出来，请[让我们知道](#)。

## 无法创建 minikube 集群

如果你使用的是 Mac，那么，`hyperkit` 是最好的虚拟机驱动——但首先需要你安装最新的[Docker for Mac](#)。之后，尝试 `jx create cluster minikube`。

如果，你的 minikube 启动失败，那么你可以尝试：

```
1. minikube delete
2. rm -rf ~/.minikube
```

如果运行 `rm` 失败，你可能需要：

```
1. sudo rm -rf ~/.minikube
```

现在，再试一次 `jx create cluster minikube`，这样有帮助吗？有时候，从安装的旧版本中一些过时的证书或者文件会导致 minikube 失败。

有时候，当虚拟机出错时，重启可能会有帮助。

另外，你还可以尝试下面的 minikube 指令

- [安装 minikube](#)
- [运行 minikube start](#)

## Minikube 和 hyperkit：无法找到 IP 地址

如果你在 Mac 上通过 hyperkit 使用 minikube，并发现 minikube 启动失败的日志如下：

```
1. Temporary Error: Could not find an IP address for 46:0:41:86:41:6e
2. Temporary Error: Could not find an IP address for 46:0:41:86:41:6e
3. Temporary Error: Could not find an IP address for 46:0:41:86:41:6e
4. Temporary Error: Could not find an IP address for 46:0:41:86:41:6e
```

这里可能会给你提示，[minikube](#) 和 [hyperkit](#) 相关问题。

解决的办法是请尝试下面的操作：

```
1. rm ~/.minikube/machines/minikube/hyperkit.pid
```

然后，再试一次。希望这次能够成功！

## 无法访问 minikube 上的服务

当运行 minikube，本地 `jx` 默认使用 `nip.io` 作为服务的域名解析，并解决了大多数笔记本无法使用通配的 DNS。然而，有时候，`nip.io` 会出问题而无法工作。

为了避免使用 `nip.io` 你可以进行以下操作：

编辑文件 `~/.jx/cloud-environments/env-minikube/myvalues.yaml`，并添加下面的内容：

```
1. expose:
2.   Args:
3.     - --exposer
4.     - NodePort
5.     - --http
6.     - "true"
```

然后，再次运行 `jx install`，这将会把把服务暴露在 `node ports`，不再使用 ingress 和 DNS。

因此，如果你输入：

```
1. jx open
```

你将会看到所有的 UR 格式 `http://$(minikube ip):somePortNumber`，不再通过 `nip.io`。这就意味着 URL 使用难记忆的数字格式而不是简单的主机名。

## 其他问题

请[让我们知道](#)，看我们是否可以提供帮助？祝你好运！

## 参照

- [技术性问题](#)
- [常见问题解答](#)
- [安装问题](#)
- [开发问题](#)

# 技术性问题

---

Kubernetes 以及关联的开源项目的技术问题

## 什么是 Helm?

---

`helm` 是 Kubernetes 的开源包管理器。

它和其他的包管理工具（`brew`，`yum`，`npm`等）类似，有一个或者更多的包仓库可以安装（在 `helm` 中叫做 `charts` 和 `kubernetes` 的主题保持一致），可以搜索、安装和升级。

一个 `helm chart` 基本上是带版本的 `kubernetes yaml` 压缩包，可以轻松地安装在任何 `kubernetes` 集群上。

Helm 通过文件 `requirements.yaml` 支持组合（一个 `chart` 可以包含其他 `charts`）。

## 什么是 Skaffold?

---

`skaffold` 是一个开源的工具，用于在 Kubernetes 集群中构建 `docker` 镜像，并通过 `kubect1` 或 `helm` 部署、升级。

在 `kubernetes` 集群中构建 `docker` 镜像的挑战有几种方法来实现：

- 使用本地 `docker daemon` 和 `kubernetes` 集群的 `socket`
- 使用一个云服务，例如：Google Cloud Builder
- 使用无 `docker-daemon`，例如：`kaniko` 不需要访问权限Skaffold 的好处是把你的代码或 CLI 从细节中抽象出来；你可以在文件 `skaffold.yaml` 中配置构建 `docker` 镜像的策略，切换 `docker daemon`、GCB 或 `kaniko`等。

Skaffold 在 `DevPods` 中也很有用，当你改变代码后可以执行快速增量构建。

## Helm 和 Skaffold 比较？

---

`helm` 允许你安装、升级叫做 `charts` 的包，使用一个或者多个在一些 `docker registry` 中的 `docker` 镜像以及一些 `kubernetes YAML` 文件来安装、升级 `kubernetes` 集群中的应用。

`skaffold` 是一个用于执行 `docker` 构建的工具，也可以通过 `kubect1` 或 `helm` 重启部署应用—或者在一个 `CI/CD` 流水线中以及本地开发中使用。

Jenkins X 使用在它的 `CI/CD` 流水线中使用 `skaffold` 创建 `docker` 镜像。我们在每次合并到 `master` 时发布版本化的 `docker` 镜像和 `helm charts`。然后，我们通过 `helm` 升级环境。

## 什么是 exposecontroller?

---

事实证明，在 `Kubernetes` 集群中暴露服务比较复杂。例如：

- 使用什么域名？
- 你是否应该使用 TLS 和生成的证书，并把它们关联到域名上？
- 你是否在使用 OpenShift，如果是的话，可能使用 `Route` 会比 `Ingress` 更好？因此，我们在 Jenkins X 中通过把微服务代理到一个叫做 `exposecontroller` 的服务上实现简化，它的职责就是处理上面的事情——把所有带有表明希望暴露到当前集群的 `Service` 资源暴露，类似域名的命名空间的暴露规则，是否使用 TLS 以及 `Route` 或 `Ingress` 等。

如果你看一下你的环境 git 仓库，可能会注意到两个 `exposecontroller` 默认是 `charts`

默认有两个任务用来自动化生成或者清理 `Ingress` 资源，以实现暴露标记了你希望从集群外部访问的 `Services` 资源。例如：web 应用或者 rest 接口。

你也可以不使用 `exposecontroller` —— 只要不在你的服务中使用 `exposecontroller` 标签即可。你也可以从环境中移除 `exposecontroller` 任务 —— 这么做的话，我们所有的快速开始（QuickStarts）都无法从集群外部访问！

## 参照

---

- [常见问题解答](#)
- [常见问题解答](#)
- [开发问题](#)
- [安装问题](#)

# 开发问题

有关如何使用 Kubernetes, Helm 和 Jenkins X 构建云原生应用

## 如何注入特定的环境配置

Jenkins X 中的每个环境都在 git 存储库中定义；我们使用 GitOps 来管理每个环境中的所有更改，例如：

- 添加/删除应用
- 更改应用程序的版本（更新或回滚）
- 使用环境特定值配置任何应用程序前两个项在您环境的 git 存储库的 `env/requirements.yaml` 文件中定义。后者在 `env/values.yaml` 文件中定义。

Helm charts 使用 `values.yaml`文件，以便您可以覆盖 charts 中的任何配置以修改设置，例如任何资源或资源配置上的标签或注释（例如 `replicaCount` ）或将环境变量等内容传递给 `Deployment` 。

所以，如果你想改变 `staging` 环境中应用 `foo` 的 `replicaCount` ，那么通过 `jx get env` 查找 `staging` 环境的 git 存储库，找到 git URL 。

导航到 `env/values.yaml` 文件并添加/编辑一些 YAML ，如下所示：

```
1. foo:
2.   replicaCount: 5
```

将该更改作为 Pull Request 提交，以便它可以通过 CI 测试并且进行任何同行评审/批准；然后当它合并到它的 master 分支它将修改 `foo` 应用程序的 `replicaCount` （假设在 `env/requirements.yaml` 文件中有一个名为 `foo` 的 chart ）

如果需要，可以使用 vanilla helm 来执行注入当前命名空间之类的操作。

要查看如何使用 `values.yaml` 文件注入 chart 的更复杂示例，请参阅我们如何使用这些文件[配置 Jenkins X 本身](#)

## 如何管理每个环境中的 Secret ？

我们自己使用封闭 Secrets 来管理我们所有 CI/CD 的 Jenkins X 安装 - 所以 Secret 被加密并检出到每个环境的 git 仓库。 我们使用 `helm-secrets` 插件来执行此操作。

虽然更好的方法是使用我们正在调研的 Vault Operator - 它可以通过 Vault 获取和填充密码（并回收它们等）。

## 参照

- [技术性问题](#)
- [常见问题解答](#)

- [常见问题解答](#)
- [安装问题](#)

- [给 Jenkins X 项目做贡献](#)
- [开发](#)
- [API 文档](#)
- [文档](#)
- [计划的特性](#)
- [分类问题](#)



# 给 Jenkins X 项目做贡献

---

Jenkins X 在很大程度上依赖于开源社区的热情参与。我们需要你，所以请加入我们！这里有很多方式提供帮助：

- [给我们反馈](#)。 我们还可以改进什么？你不喜欢什么或者你认为缺少什么？
- 帮助 [改进文档](#)，以便更清楚地了解如何开始使用Jenkins X。
- [添加您自己的快速入门](#) 以便 Jenkins X 社区可以使用您的快速入门轻松引导新项目。如果你在为一个开源项目工作，那么这个项目是否可以作为一个好的快速入门项目添加到 Jenkins X 当中。
- 创建一个 [插件](#)。要添加自己的插件，只需创建一个 Helm Chart ，以获得扩展 Jenkins X，然后提交一个 Pull Request 在 [the pkg/kube/constants.go file](#) 来添加你的 chart 的名称匹配到 `AddonCharts` 。
- 如果你想要 [贡献代码](#) 那么尝试浏览 [当前问题](#)。
  - 我们已经标记了问题 [需要帮助](#) 或者 [好的首个问题](#) 为你节省寻找问题的时间。
  - 我们特别乐意帮助您 [在 Windows 上运行 Jenkins X](#) 或者 [和云服务, git 提供程序以及问题跟踪器集成](#)。
  - 为了更长远的目标，我们制定了 [长期路线图](#)。
  - 我们总是接收更多的测试用例并提高测试覆盖率。

# 贡献代码

如何为 Jenkins X 的发展做贡献

## 介绍

Jenkins X 是由众多[开发者](#)开发的开源项目。还有很多 [open issues](#)，我们需要你的帮助来使 Jenkins X 变得更棒。即使你不是一个 Go 语言的专家，也可以对项目的开发贡献力量。

## 假设

本篇指导文档将帮助新接触 Jenkins X 的读者逐步熟悉它，因此我们假定：

- 你是刚刚接触 Git 或者开源项目
- 你是 Jenkins X 的爱好者并乐于对项目的发展贡献力量

如果在阅读此指导文档过程中有任何问题，请向 Jenkins X 社区的[讨论组](#)寻求帮助。

## 安装 Go

Go 语言环境的安装仅需要几分钟。并且多种方式可供选择。

如果在安装过程当中遇到问题，请查阅 [Go Bootcamp, which contains setups for every platform](#) 或者向 Jenkins X [论坛](#)中寻求帮助。

## 从源码安装 Go

[下载最新版 Go 源码](#)并通过官方[安装文档](#)进行安装。

安装完成后，确认是否一切工作正常。打开一个新的终端或者在 Windows 上的命令行并输入：

```
1. go version
```

在终端的窗口上可以看到类似如下的信息。注意 `version` 表示的是在更新此文档时最新的 Go 的版本信息：

```
1. go version go1.8 darwin/amd64
```

下一步，确保[根据安装文档](#) 设置了 `GOPATH` 环境变量。通过 `echo $GOPATH` 输出 `GOPATH`。应该是指向了你的合法的 Go 的工作目录的非空字符串，如：

```
1. /Users/<yourusername>/Code/go
```

## 使用 Homebrew 安装 Go

如果你是 MacOS 用户并且安装了 [Homebrew](#)，安装过程将会很简单，在终端中执行以下命令：

```
install-go.sh
```



- 1.
2. `brew install go`

## 通过 GVM 安装 GO

更有经验的用户可以使用 [Go Version Manager](#) (GVM)。GVM 允许你在 同一台机器上 安装并切换使用多种版本的 Go 语言环境。如果你是初学者，可能不太需要这个功能。然而， GVM 通过几条命令可以很简单的更新到新发布版本的 Go 语言。

在开发 Jenkins X 很长一段时间后，GVM 使用起来将会特别的方便。Jenkins X 之后的版本将会用最新版版的 Go 语言进行编译，因此如果想与社区开发同步的话，将会需要更新 Go 环境。

## 创建一个 GitHub 账号

如果你想要贡献代码的话，需要创建一个 Github 账号。登录 [www.github.com/join](https://www.github.com/join) 注册个人账号。

## 在你的系统上安装 Git

Jenkins X 开发过程当中需要在本机安装 Git 客户端。Git 的使用学习不包含在 Jenkins X 的文档中，如果你不确定从哪里开始的话，我们推荐通过 [Git book](#) 学习使用 Git 的基本知识。使用的词汇将会通过注解进行解释。

Git 是一个[版本控制系统](#)，用于跟踪源代码的变化。为了不重复造轮子，Jenkins X 使用了第三方的软件包来扩展功能。

Go 提供了 `get` 的子命令来帮助下载软件包以配置工作环境。这些软件包的源码信息在 Git 中记录。`get` 会与承载这些软件包的 Git 服务器端进行交互来下载所有的依赖。

回到终端中，输入 `git version` 并按回车，检验是否安装 Git。如果返回的是一个版本号信息，那么可以跳过下面的配置。否则的话[下载](#)最新版的 Git 并根据[安装文档](#)进行安装。

最后，再一次输入 `git version` 检验 Git 是否已经安装。

## Git 图形化前端

有一些[图形界面客户端](#)可以帮助操作 Git。并不是所有的客户端在所有的操作系统上都有相应的版本，而且不同的客户端的使用方法也可能不同。因此，在以下的操作中，我们会以使用命令行的方式为基准。

## 在你的系统上安装 Hub（可选）

在与 GitHub 协同开发时，Hub 是个很好的工具。请访问 [hub.github.com](https://hub.github.com)，来安装体验这个封装了 Git 的小工具。

在 Mac 系统上，可以通过 [Homebrew](#) 来安装 Hub：

```
1. brew install hub
```

安装之后，在 Bash 中创建[快捷键](#)，以方便我们在执行 `git` 的时候，实际上执行的是 `hub`：

```
1. echo "alias git='hub'" >> ~/.bash_profile
```

确认安装配置是否正确：

```
1. git version 2.6.3
2. hub version 2.2.2
```

## 设置你的工作副本

工作副本是在你的电脑中进行本地设置的。你将会对它进行编辑，编译以及最终推送回到 GitHub。主要的步骤是在远端对源 Git 代码库创建你的分支仓库并之后在本地进行克隆。

## 克隆仓库

我们假定你已经设置了 `GOPATH`（如果不确定的话查阅上面的相应部分）环境变量，现在可以下载 Jenkins X 的代码库到本地电脑中。这一过程就是被称作“克隆仓库”。GitHub 的[帮助文档](#)对其进行了简短的解释：

```
When you create a repository on GitHub, it exists as a remote repository. You can create a local clone of your repository on your computer and sync between the two locations.
```

我们会克隆 Jenkins X 代码库的[主版本](#)。由于你还没有对代码库的提交代码的权限，这看上去有些违反常理。但是这一步骤在 Go 的工作流当中是必须的一项。你将会在主版本的副本中进行工作，将修改的部分提交到你在 GitHub 上的仓库当中。

首先，我们克隆主版本库：

```
1. go get -v -u github.com/jenkins-x/jx
```

Jenkins X 使用 [Testify](#) 进行 Go 代码的测试。如果还没有安装的话，使用下面的方式获得 Testify 测试工具：

```
1. go get github.com/stretchr/testify
```

## 派生仓库

如果对这个术语感到陌生的话，GitHub 的[帮助文档](#) 提供了简单的说明：

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

## 手工派生

打开 [Jenkins X 仓库](#)，点击右上角的“Fork”按钮。



现在打开你在 GitHub 中创建出的派生仓库，拷贝远端连接。你可以在 HTTPS 和 SSH 协议间进行选择。HTTPS 适用于任何情况。如果你不确定的话，请[查阅](#)。



切换到命令窗口中，进入到刚才所克隆的主版本库的工作目录当中。

```
1. cd $GOPATH/src/github.com/jenkins-x/jx
```

现在 Git 需要知道我们刚刚创建出来的分之仓库的地址信息

```
1. git remote add <YOUR-GITHUB-USERNAME> <COPIED REMOTE-URL>
```

## 使用 Hub 派生

相类似的，可以使用 Git 的封装工具 Hub 进行操作。Hub 使得创建分之仓库变得容易：

```
1. git fork
```

这一命令会使用你的账号登录到 GitHub 中，并对当前所在的工作目录的主仓库创建派生仓库，在之后会将新创建的连接信息添加到你的工作副本当中。

## 验证

让我们通过列出所有已有的 remote 来检查是否一切就绪：

```
1. git remote -v
```

输出应该类似如下内容：

```
1. digitalcraftsman  git@github.com:digitalcraftsman/hugo.git (fetch)
2. digitalcraftsman  git@github.com:digitalcraftsman/hugo.git (push)
```

```
3. origin https://github.com/jenkins-x/jx (fetch)
4. origin https://github.com/jenkins-x/jx (push)
```

## Jenkins X Git 贡献流程

### 创建新的分支

永远不要在 “master” 分支上进行代码的开发。开发团队也不会接受在此之上的 pull request。相反， 应该创建一个有描述信息的分支并在其之上进行开发。

首先， 你需要获取在主版本上进行的最新的内容：

```
1. git checkout master
2. git pull
```

现在，为你的附加功能创建一个新的版本：

```
1. git checkout -b <BRANCH-NAME>
```

可以通过 `git branch` 来检查你当前所在的分支。你可以看到一个包含所有本地分支的列表。在当前所操作的版本之前会有 “\*” 标识。

### 贡献文档

也许你想先从 Jenkins X 的文档开始贡献。如果这样的话，你可以省略下面大部分的步骤，仅关注在刚克隆的代码库的 `/docs` 目录中的文件即可。 通过执行 `cd docs` 进入文档目录中。

可以通过 `hugo server` 启动 Jenxins X 内置的服务。 通过浏览器访问 <http://localhost:1313> 进行浏览。Hugo 会监测所有文件内容的修改，并将其在浏览器中进行显示。

想了解更多的信息，包括 Jenkins X 文档是如何构建、组织以及由众多像你一样无私的人如何对其进行改进的，请[参阅](#)。

### 构建 Jenkins X

在代码库上进行更改的同时，创建相应的二进制文件来进行测试是很好的方法：

```
1. go build -o hugo main.go
```

### 测试

有时对代码的修改可能会带来没有注意到的负面影响。或者是并不像预期的那样工作。大部分的功能都有其相对应的测试用例。这些测试文件都以 `_test.go` 结尾。

请确保 `go test ./...` 命令通过没有异常以及 `go build` 执行完毕。

## 格式

Go 语言的代码格式也许根据人的意识会有所不同，但是不论是由谁编写的代码，Go 本身会确保代码看上去一致。Go 提供了格式化工具，使我们的修改风格统一：

```
1. go fmt ./...
```

如果进行了修改，请确保遵循我们的[代码贡献指导说明](#)。

```
1. # Add all changed files
2. git add --all
3. git commit --message "YOUR COMMIT MESSAGE"
```

代码的提交记录信息应该描述提交做了那些工作（如，添加功能 XYZ）而不是描述如何完成的。

## 修改提交

你也许注意到了一些提交记录信息并不遵守贡献指导说明或者你是在某些文件中忘记了什么。没关系，Git 提供了相应的工具来解决类似这样的问题。下面的两种方法将会覆盖所有的常见问题。

如果你不确定如何使用这些命令的话也可以保留不行改正，在之后提交 Pull Request 的时候，我们会对提交信息进行修改。

### 修改最后一次提交

让我们以你想要修改最后的一次提交信息为例。执行下面的命令以替换之前的提交信息：

```
1. git commit --amend -m"新的提交信息"
```

检查历史提交记录，查询修改信息：

```
1. git log
2. # 输入 q 退出
```

在做了最后的修改后，你也许忘记了什么。没有必要创建新的提交。只需要将最新的修改添加到 Git 记录当中并在之后将其合并到之前的修改中：

```
1. git add --all
2. git commit --amend
```

### 修改多次提交

对此章节中介绍的修改，可能会造成不可意料的后果。如果不确定的如何使用的话，跳过下面的部分！

这一部分的操作需要更高的技能。Git 允许你对多次提交进行[修改](#)。换句话说：它允许你对历史的提交进行修改。

```
1. git rebase --interactive @~6
```

在命令结尾处的 `6` 表示的是想要进行修改的提交的编号。它会打开一个编辑器，其内容是之前6次的历史提交信息列表：

```
1. pick 80d02a1 tpl: Add hasPrefix to the template funcs' "smoke test"
2. pick aaee038 tpl: Sort the smoke tests
3. pick f0dbf2c tpl: Add the other test case for hasPrefix
4. pick 911c35b Add "How to contribute to Jenkins X" tutorial
5. pick 33c8973 Begin workflow
6. pick 3502f2e Refactoring and typo fixes
```

在上面的例子中，我们应该将最后的提交到本文档之间的提交（`Add "How to contribute to Jenkins X" tutorial`）历史提交进行合并。你可以“压缩”提交，如，将两个及以上的提交合并为一个。在提交信息之前，所有的操作都将会执行。替换 `pick` 为想要进行的操作。在这个例子当中我们使用 `squash` 或者其省略版 `s`。

```
1. pick 80d02a1 tpl: Add hasPrefix to the template funcs' "smoke test"
2. pick aaee038 tpl: Sort the smoke tests
3. pick f0dbf2c tpl: Add the other test case for hasPrefix
4. pick 911c35b Add "How to contribute to Jenkins X" tutorial
5. squash 33c8973 Begin workflow
6. squash 3502f2e Refactoring and typo fixes
```

根据代码贡献指导文档，在历史提交中的第三个提交忘记了添加前缀 `"docs:"`，因此想要对其进行修改。修改一个提交的操作是 `reword` 或者其省略版 `r`。

修改后，应该是类似如下的内容：

```
1. pick 80d02a1 tpl: Add hasPrefix to the template funcs' "smoke test"
2. pick aaee038 tpl: Sort the smoke tests
3. pick f0dbf2c tpl: Add the other test case for hasPrefix
4. reword 911c35b Add "How to contribute to Jenkins X" tutorial
5. squash 33c8973 Begin workflow
6. squash 3502f2e Refactoring and typo fixes
```

此时关闭编辑器。它会打开新的窗口，将会有文本指导你对之前的两次提交进行的合并（即，“压缩”）设置新的提交信息。输入 `CTRL + S` 保存文件关闭编辑器。

再一次，将会打开新的窗口。输入新的提交信息并且保存。你的终端将会显示如下类似的状态信息：

```
1. Successfully rebased and updated refs/heads/<BRANCHNAME>.
```

检查提交记录以确保修改成功。如果发生了错误的话，可以通过执行 `git rebase --abort` 来撤销操作。

## 推送提交

我们需要指定目标地址以使得将我们的提交推送回到在Github中的分支版本库。目标地址由 `remote` 和



`branch` 名称所构成。在之前的操作中，`remote` 地址与我们的GitHub账号所对应，以我为例是 `digitalcraftsman`。分支（branch）应该和我们本地的一样。这就使得识别相应的分支变得简单。

```
1. git push --set-upstream <YOUR-GITHUB-USERNAME> <BRANCHNAME>
```

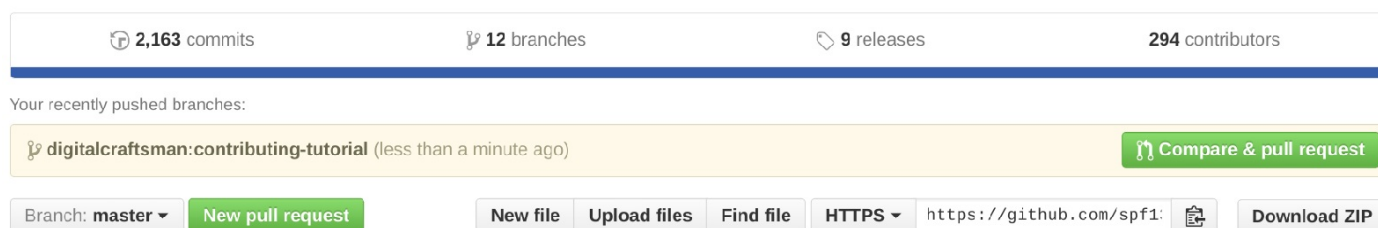
现在Git知道了目标地址。在此之后，想要进行提交的时候，只需要输入 `git push`。

如果你在上一步骤对历史提交记录进行了修改，GitHub 会拒绝你的推送。这是一个保护功能，因为历史提交记录不一致以及新的提交不能像往常一样进行追加。你可以通过 `git push -force` 强制的进行提交。

## 打开一个 Pull Request

做的很好，我们有了很大的进展。在这一步，我们将会提出合并请求来提交我们的附加功能。在浏览器中打开 [Jenkins X 主代码库](#)。

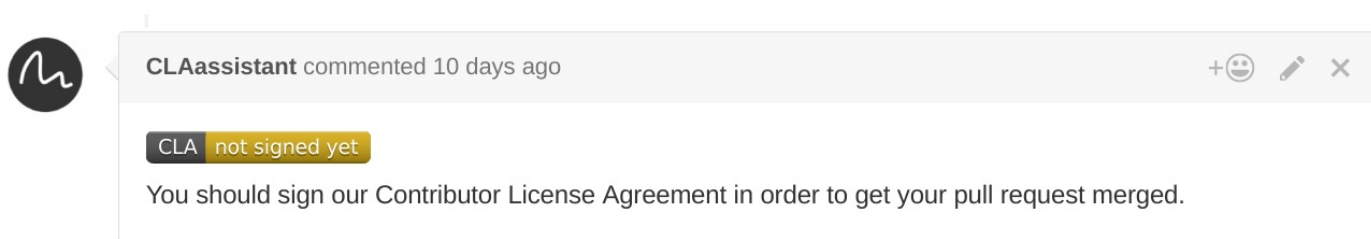
你会发现一个绿色按钮，上面标识 “New pull request”。GitHub 很智能，很有可能像如下图所示的那样，在一个米黄色窗口中建议你开 pull request：



在新的页面当中，将会包含你的 pull request 中的重要信息。滚动鼠标你会发现所有的提交信息。确保所有的一切与构想的一致并点击按钮 “Create pull request”。


## 同意贡献者授权协议

最后也同样重要的是，你应该同意贡献者授权协议（CLA）。一个新的评论信息应该会自动的添加到你的 pull request 当中。点击黄色的徽章，同意协议并用你自己的 GitHub 账号进行认证。它仅需要几步点击之后即可完成。










## 自动化构建

我们使用 [Travis CI loop](#)（Linux 和 OS X）以及 [AppVeyor](#)（Windows）来对包含有你的提交的 Jenkins X 进行编译。这可以确保在合并你的 pull request 之前，所有的都与所设想的工作一致。大部分情况下，如果你对 Jenkins X 的代码库进行了修改的话，这将很有意义。


**Some checks were not successful**
[Hide all checks](#)

1 failing and 5 successful checks

	<b>✗ continuous-integration/travis-ci/pr</b> — The Travis CI build failed	<a href="#">Details</a>
	<b>✓ ci/circleci</b> — Your tests passed on CircleCI!	<a href="#">Details</a>
	<b>✓ code-review/gitcolony</b> — 0/0 qa 0/0 dev, 0 issues	<a href="#">Details</a>
	<b>✓ continuous-integration/appveyor/pr</b> — AppVeyor build succeeded	<a href="#">Details</a>
	<b>✓ continuous-integration/wercker</b> — Wercker build passed	<a href="#">Details</a>
	<b>✓ licence/cla</b> — Contributor License Agreement is signed	<a href="#">Details</a>


**This branch has conflicts that must be resolved**  
 Use the [command line](#) to resolve conflicts before continuing.

[Merge pull request](#) or view [command line instructions](#).

在上图中，你可以看到 Travis 不能够对这个 pull request 进行编译。点击 “Details” 来查看失败的原因。但是这个错误并不一定是由你的提交所导致的。大部分情况下，我们使用 `master` 分支来作为基础来验证你的 pull request 是没有问题的。

如果你遇到问题的话，在 pull request 当中进行评论。我们愿意对你进行帮助。

## 从哪里开始？

感谢你阅读了本篇贡献指导文档。希望我们可以在 GitHub 中再次看到你。有很多 [open issues](#) 需要你的帮助。

如果你认为发现了 bug 或者有新的想法可以改进 Jenkins X，请随时的 [open an issue](#)，我们很乐于听取你的声音。

## 学习 Git 和 Golang 的参考

- [Codecademy's Free "Learn Git" Course](#) (免费)
- [Code School and GitHub's "Try Git" Tutorial](#) (免费)
- [The Git Book](#) (免费)
- [Go Bootcamp](#)
- [GitHub Pull Request Tutorial, Thinkful](#)

## 参照

- [计划的特性](#)

# 给 API 文档做贡献

如何帮助改善 Jenkins X 的 API 文档

Jenkins X 有两种类型的 API 文档：[Kubernetes Custom Resource Documentation](#) 和 [Godoc](#)。这两种类型都是由 [jx](#) 的代码生成。

## 设置你的开发环境

最好在你的本地电脑上对 Jenkins X 的代码进行修改。按照 [开发](#) 指导文档进行配置。

## 编写自定义资源文档

自定义资源的文档大部分是由 [go structs that define the customresources](#) 上的注释以及融合了 [introductory content](#) 和 [structure](#) 而生成的。

## 工具链

自定义资源文档是由与 Kubernetes [同样的工具链](#)而生成的，但是一系列的 `jx` 的命令将其包装了起来，因此你不需要下载以及配置这些不同的工具。

HTML 文档是由 [OpenAPI 说明](#) 生成的，依次由 [Go 结构体](#) 生成，而这些结构体是由代码的注释生成的。想要生成结构体和 OpenAPI 说明执行命令：

```
1. $ make generate-openapi
```

`make generate-openapi` 仅仅是对 `jx create client openapi` 进行了包装，通过传入参数：从哪个包来生成、生成的目标包的名称和组（`jenkins.io`）以及版本（`v1`）来生成最终的文件。如果你愿意的话，也可以直接运行这个命令。

生成 HTML 运行：

```
1. make generate-docs
```

`make generate-docs` 仅仅是对 `jx create client docs` 进行了包装。如果你愿意的话，也可以直接运行这个命令。

当你对自定义资源进行了修改的话，应该运行 `make generate-openapi`，并确认将所生成的修改添加到版本控制当中。这意味着对于其他人，在任意时刻都会有标记了版本的 OpenAPI 说明可供使用。

你也可以运行 `make generate` 它会进行所有 Jenkins X (mocks、client 以及 OpenAPI) 所需要的代码生成工作。

构建版本时会运行 `make generate-docs`，并且对于每一个版本，更新都会自动的上传到 Jenkins X 的网站上。在版本构建结束的几分钟后会生效。

## 对文档进行修改

对于每一个你想要生成文档的文件必须要在 `jenkins.io/v1` 目录中，并且在文件的头部必须有下面的注释：

```
1. // +k8s:openapi-gen=true
```

想要移除一个类型或者成员的话，添加：

```
1. // +k8s:openapi-gen=false
```

对类型的注释会被忽略。结构体中的字段的注释会被作为其描述信息。左侧的菜单栏是由 `config.yaml` 中的 `resource_categories` 而生成。对于每一个种类的介绍文本信息由 `html` 编写。

样式风格也可以 [定制化](#)。

## OpenAPI

OpenAPI 说明是由代码生成的。其结构由结构体以及字段生成。`json` `tags` 被用于提供额外的信息包括：

- `name` 由 `key` 生成
- 如果没有设置 `omitempty` 的话，那么这个属性将是 `必需的`
- 如果 `key` 是 `-` 的话，那么将会跳过这个字段
- 如果设置了 `inline` 的话，这些属性将会嵌入到父对象当中此外，注释可以用于阻止某一属性被设置为 `必需项`

```
1. // +optional
```

例如：

```
1. metav1.TypeMeta `json:",inline"
2.     // +optional
3.     metav1.ObjectMeta `json:"metadata,omitempty" protobuf:"bytes,1,opt,name=metadata"`
4.     Spec BuildPackSpec `json:"spec,omitempty" protobuf:"bytes,2,opt,name=spec"`
```

## OpenAPI 扩展

在类型上，OpenAPI 说明也可以有扩展。想要在一个类型或者成员上添加一个或多个扩展的话，在类型/成员的注释行上添加 `+k8s:openapi-gen=x-kubernetes-$NAME:$VALUE`。一个类型/成员可以有多个扩展。在注释中的其它的行会被作为 `$VALUE` 因此不需要躲避或者字符串加上引号。扩展可以用于向客户端或者文档生成器传入更多的信息。例如，一个类型在文档中可以有友好的名称用于展示或者用于客户端流畅的接口。

## 自定义 OpenAPI 类型定义

自定义类型不会直接的映射到 OpenAPI 当中，而是会通过实现名为 “OpenAPIDefinition” 的方法如下面所示，来覆盖他们的 OpenAPI 说明：

```
1. import openapi "k8s.io/kube-openapi/pkg/common"
2.
3. // ...
4.
5. type Time struct {
6.     time.Time
7. }
8.
9. func (_ Time) OpenAPIDefinition() openapi.OpenAPIDefinition {
10.     return openapi.OpenAPIDefinition{
11.         Schema: spec.Schema{
12.             SchemaProps: spec.SchemaProps{
13.                 Type: []string{"string"},
14.                 Format: "date-time",
15.             },
16.         },
17.     }
18. }
```

此外，类型可以通过定义下面的方法来避免引用 “openapi”。下面的例子会和上面的例子产生相同的 OpenAPI 说明：

```
1. func (_ Time) OpenAPISchemaType() []string { return []string{"string"} }
2. func (_ Time) OpenAPISchemaFormat() string { return "date-time" }
```

## 编写 Godoc

Jenkins X 使用标准的方法来生成 Godoc，而且会由 [godoc.org](https://godoc.org) 自动生成。这一[博客](#) 为编写 Godoc 提供了很好的介绍。

## 参照

- [文档贡献](#)

# 文档贡献

如何完善 Jenkins X 文档

## 创建派生库

最好在你本地的机器上修改 Jenkins X 文档，检查视觉风格一致。确保你已经在 GitHub 上派生了 `jx-docs`，并在你的机器上克隆了这个库。更多信息，你可以查看 [GitHub 的“派生”文档](#) 或者按照 [Jenkins X 开发贡献指导](#)。

然后，你可以创建一个独立的分支。一定要选择符合内容类型的描述性分支名称。下面的一个示例分支的名称，你可以用于添加一个新的网站用于展示：

```
1. git checkout -b jon-doe-showcase-addition
```

## 添加新的内容

Jenkins X 文档重用 Jenkins X 的[骨架](#)特点。在 Jenkins X 文档中所有章节都分配了骨架。

向 Jenkins X 中添加新的内容遵循下面相似的模式，不用考虑内容章节：

```
1. hugo new <DOCS-SECTION>/<new-content-lowercase>.md
```

## 语法标准

Jenkins X 文档中所有的页面，使用典型的三个反引号这样的语法。如果你不想花额外的时间来遵循下面的代码块简码，请使用标准的 GitHub 风格的 markdown。Jenkins X 使用 `highlight.js` 的一组语言。

你可选的语言是 `xml` / `html` , `go` / `golang` , `md` / `markdown` / `mkd` , `handlebars` , `apache` , `toml` , `yaml` , `json` , `css` , `asciidoc` , `ruby` , `powershell` / `ps` , `scss` , `sh` / `zsh` / `bash` / `git` , `http` / `https` , 和 `javascript` / `js` .

```
1. <h1>Hello world!</h1>
```

## 代码块简码

Jenkins X 文档带有强大的简码，用于增加交互式的代码块。

通过 `code` 这个简码，你必须包括三个反引号和语言声明。简码包裹这样的设计，可以轻松地添加到遗留文档，如果有必要在 Jenkins X 未来的版本中移除的话也是很容易的。

## code

`code` 这个简码你将会在 Jenkins X 中经常使用。 `code` 只能接收一个命名参数： `file` 。模式是：

```
1. {{% code file="smart/file/name/with/path.html" download="download.html" copy="true" %}}
```

一大堆的编码会出现在这里！

```
1. {{% /code %}}
```

下面是传递给 `code` 的参数：

- `file`
- 这是唯一的 必需 参数。 `file` 是用于风格需要，但同样也扮演了一个重要的角色，它帮助用户建立一个 Jenkins X 目录结构的思维模式。视觉上，这会作为文本显示在代码块的左上角。
- `download`
- 如果忽略，那么在渲染简码时没有任何效果。当添加一个值到 `download` ，它会被当作文件名来作为这个代码块来下载。
- `copy`
- 拷贝按钮会自动添加到所有 `code` 简码。如果你想保持文件名和 `code` 的风格，但不想要渲染来拷贝代码（例如：在教程中的“不得做”的片段），使用 `copy="false"` 。

## 示例 code 输入

这个 HTML 示例代码块告诉 Jenkins X 用户如下信息：

- 这个文件 会 在 `layouts/_default` 中， `layouts/_default/single.html` 也就是 `file` 的值。
- 这个片段是完全可以下载，并是在 Jenkins X 工程里实现的，也就是 `download="single.html"` 。

```
1. {{< code file="layouts/_default/single.html" download="single.html" >}}
2. {{ define "main" }}
3. <main>
4.     <article>
5.         <header>
6.             <h1>{{.Title}}</h1>
7.             {{with .Params.subtitle}}
8.                 <span>{{.}}</span>
9.             </header>
10.            <div>
11.                {{.Content}}
12.            </div>
13.            <aside>
14.                {{.TableOfContents}}
15.            </aside>
16.        </article>
17.    </main>
18. {{ end }}
19. {{< /code >}}
```

示例 ‘code’ 显示

这个示例的输出将会如下展示到 Jenkins X 文档中：

layouts/\_default/single.html



```
1.
2. {{ define "main" }}
3. <main>
4.   <article>
5.     <header>
6.       <h1>{{.Title}}</h1>
7.       {{with .Params.subtitle}}
8.         <span>{{.}}</span>
9.       </header>
10.    <div>
11.      {{.Content}}
12.    </div>
13.    <aside>
14.      {{.TableOfContents}}
15.    </aside>
16.  </article>
17. </main>
18. {{ end }}
```

## 块引用

块引用可以通过 [典型的 Markdown 块引用语法](#) 添加到 Jenkins X 文档中：

```
1. > Without the threat of punishment, there is no joy in flight.
```

上面的块引用会在 Jenkins X 文档中渲染为：

Without the threat of punishment, there is no joy in flight.

然而，你可以简单快速地添加一个 `<cite>` 元素（通过 JavaScript 在客户端添加），通过在连字符两边添加空格来区分你的块引用和参考。

```
> Without the threat of punishment, there is no joy in flight. - [Kobo Abe]
1. (https://en.wikipedia.org/wiki/Kobo_Abe)
```

这样会在 Jenkins X 文档中渲染为：

Without the threat of punishment, there is no joy in flight. - Kobo Abe

Previous versions of Jenkins X documentation used blockquotes to draw attention to text. This is *not* the [intended semantic use of](#) `<blockquote>`. Use blockquotes when quoting. To note or warn your user of specific information, use the admonition shortcodes that follow.



# 警告

警告 在技术性文档中是常见的。最常见的是在 [reStructuredText Directives](#)。从 SourceForge 的文档中摘录：

Admonitions are specially marked “topics” that can appear anywhere an ordinary body element can. They contain arbitrary body elements. Typically, an admonition is rendered as an offset block in a document, sometimes outlined or shaded, with a title matching the admonition type. - [SourceForge](#)

Jenkins X 文档包含三种警告： `note` ， `tip` ， and `warning` 。

## note 警告

当你想要巧妙地提示信息是，可以使用简码 `note` 。 `note` 不像 `warning` 那样会打断内容。

### 示例 note 输入

note-with-heading.md



```
1.
2. {% note %}
3. Here is a piece of information I would like to draw your attention to.
4. {% /note %}
```

### 示例 note 输出

note-with-heading.html

```
1.
2. <aside class="admonition note">
3.   <div class="note-icon">
4.
5.   </div>
6.
7.   <div class="admonition-content"><p>Here is a piece of information I would like to draw your
8. <strong>attention</strong> to.</p>
9. </div>
10. </aside>
11.
```

### 示例 note 显示

Here is a piece of information I would like to draw your **attention** to.

## tip 警告

当你想要给读者建议时，使用简码 `tip` 。 `tip` ， 有点像 `note` ， 不像 `warning` 那样会打断内容。

## 示例 tip 输入

using-tip.md



```
1.
2. {{% tip %}}
3. Here's a bit of advice to improve your productivity with Jenkins X.
4. {{% /tip %}}
```

## 示例 tip 输出

tip-output.html

```
1.
2. <aside class="admonition tip">
3.   <div class="tip-icon">
4.
5.   </div>
6.
7.
8.   <div class="admonition-content"><p>Here's a bit of advice to improve your productivity with Jenkins
9.   X.</p>
10. </div>
11. </aside>
```

## 示例 tip 显示

Here's a bit of advice to improve your productivity with Jenkins X.

## warning 警告

当你想要使用户引起注意时，使用 `warning` 简码。一个好的例子就是，当在 Jenkins X 版本中会引起阻断变更时，已知问题，或者模板“陷阱”。

## 示例 warning 输入

warning-admonition-input.md



```
1.
2. {{% warning %}}
3. This is a warning, which should be reserved for *important* information like breaking changes.
4. {{% /warning %}}
```

## 示例 warning 输出

warning-admonition-output.html

```
1.
2. <aside class="admonition warning">
3.   <div class="admonition-icon">
4.
5.   </div>
6.
7.   <div class="admonition-content"><p>This is a warning, which should be reserved for <em>important</em>
8. information like breaking changes.</p>
9. </div>
10. </aside>
11.
```

## 示例 warning 显示

这是一个警告，用于 重要的 信息，例如破坏性改变。

和 [给 Jenkins X 贡献开发](#) 相似，当你想要给 Jenkins X 文档贡献时 Jenkins X 团队期望你创建一个独立的分支（派生）。

## 参照

- [给 API 文档做贡献](#)

# 计划的特性

我们计划添加的特性和连接器.....

特性	未计划	已计划	进行中	已发布
操作系统				
macOS				1.1.x
Linux				1.1.x
Windows			进行中	
<b>Cloud Providers</b>				
AWS kops				1.1.x
AWS EKS				1.3.x
Azure				1.1.x
Digital Ocean		已计划		
GKE				1.1.x
IBM Cloud				1.3.x
Minikube				1.1.x
Minishift				1.2.11
OKE		已计划		
OpenShift				1.2.11
PKS				1.3.x
工具				
Helm				1.1.x
Skaifold				1.2.11
Prow				1.3.x (via <code>-prow</code> on <code>jx install</code> )
LetsEncrypt				1.3.x
<b>Git Providers</b>				
GitHub				1.1.x
GitHub Enterprise				1.1.68
Gitea			进行中	
BitBucket Server			进行中	
BitBucket Cloud				1.2.70

GitLab			进行中	
问题跟踪系统				
GitHub				1.1.x
GitHub Enterprise				1.1.68
Gitea			进行中	
JIRA				1.1.68
仓库				
ChartMuseum				1.1.x
Docker Registry				1.1.x
Nexus				1.1.x
Artifactory			进行中	
特性				
Multi Cluster			进行中	
Istio Production Canaries			进行中	
构建打包				
Go				1.1.43
Java + Maven				1.1.x
Java + Gradle				1.1.43
Kubeless		已计划		
Node				1.1.43
Python			进行中	
Rust				1.1.43
Serverless		已计划		
Swift		已计划		
.Net		已计划		
插件				
Anchore				1.2.x
Cloud9		已计划		
Gitea				1.1.x
Prometheus				1.1.x
Theia IDE				1.3.x in <a href="#">DevPods</a>

SonarQube		已计划		
-----------	--	-----	--	--

我们使用 [这个文档](#) 来记录头脑风暴上的想法和工作主题。

如果你有兴趣帮忙，请查看上面的问题，或者 [加入我们的社区](#)。

## 参照

---

- [贡献代码](#)

# 分类问题

如何对 Jenkins X 项目中的问题进行分类

Jenkins X 项目主要的问题跟踪系统是 <https://github.com/jenkins-x/jx/issues>。这旨在捕捉问题、想法和开发工作。如有疑问请提交一个问题，一名 Jenkins X 团队成员将考虑尽快给它分类。

由于 Jenkins X 使用来自 Kubernetes 生态的 [prow](#)，我们认为，我们应该带领他们参与处理分类大量问题，以帮助和鼓励贡献者。我们正在重用标签的样式，包括颜色，以尝试在跨开源项目时创建熟悉度，并减少贡献的障碍。

# 分类问题

所有可用标签列表请参考：<https://github.com/jenkins-x/jx/labels>

当对问题进行分类时，来自 Jenkins X 团队的某个成员将分配标签用来描述问题的 **area** 和 **kind**。有可能，他们还将增加一个 **priority**，但是，在进一步分析或更广泛的可见性之后，这些 **priority** 可能会发生变化。

标签通过 [prow label](#) 插件使用 GitHub 评论被添加。例如：

1. `/kind bug`
2. `/area prow`
3. `/priority important-soon`

**Open** rawlingsj opened this issue 11 days ago · 0 comments



rawlingsj commented 11 days ago

Member



## Summary

When trying to create an environment with `jx create env --prow` the wizard asks for the Name as the first question, we then generate the prow config, update the configmap and then try to create the gitrepo. If a repo already exists with the same name the wizard asks for a new name which is then used. Problem is we have prowconfig for the old name instead of the new one so no builds will trigger.

We should probably move the repo name validation to the start of the wizard.

## Steps to reproduce the behavior

```
jx create env --prow
```

and use a name that you already have in the target git org

```
/kind bug  
/area prow  
/priority important-soon
```



jenkins-x-bot added **kind/bug** **area/prow** **priority/important-soon** labels 11 days ago

## Assignees

No one—assign yourself

## Labels

**area/prow**

**kind/bug**

**priority/important-soon**

## Projects

None yet

## Milestone

No milestone

## Notifications

Unsubscribe

You're receiving notifications because you authored the thread.

2 participants

# 分配问题

当进行分类时我们尝试将问题分配给某个人。这可能会随着调查或人员的可用性而改变。

## 调查问题

---

当任何人在处理一个问题时，我们的目的是通过添加注释来捕获任何分析。这有助于人们学习如何调查类似问题的技巧，帮助人们理解思考过程，并通过 `pull request` 为任何链接修复提供上下文。

## 新建标签

---

如果你想要请求创建一个新的标签，那么请提交一个问题并附带尽可能多的内容。

## 陈旧的问题

---

当我们鼓励广泛的问题类型，如一般的想法和想法，问题跟踪器可能增长得相当高。我们将启用 `prow lifecycle` 插件来帮助管理陈旧的问题。这并不意味着具有侵入性，而是允许我们不断地重新思考问题，并保持跨问题的势头。