

## 27 | 巨人的肩膀：那些你不能忽视的开源工具

2019-12-19 石雪峰

DevOps实战笔记

[进入课程 >](#)



讲述：石雪峰

时长 20:05 大小 18.41M



你好，我是石雪峰。

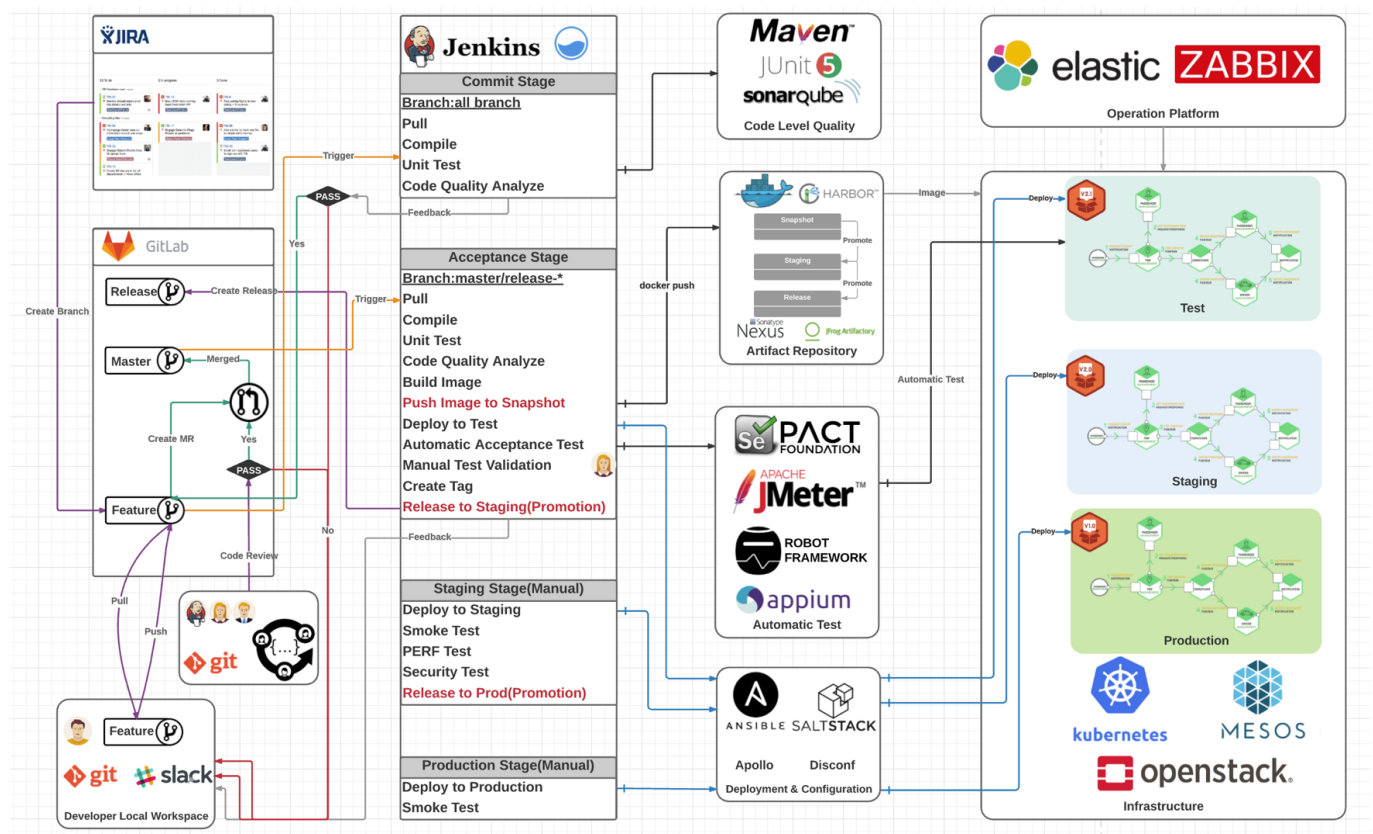
自研工具平台对公司来说是一件高成本和高投入的事情，对于技术人员的要求也非常高。很少有公司能够像 BAT 一样投入近百人的团队来开发内部系统工具，毕竟，如果没有这么大规模的团队，平台产生的收益也比较有限。

另外，也很少有公司像一些行业头部公司一样，会直接投入大量资金购买成熟的商业化工具或者通过乙方合作的方式联合共建。

这些方法的长期投入都比较大，不太适用于中小型企业。那么，有其他可以低成本、快速见效的解决方案吗？

实际上，现在的开源工具已经非常成熟了，只要稍加熟悉，就能快速地基于开源工具搭建一整套研发交付工具链平台。

几年前，我跟几个朋友利用业余时间就搭建了这样一套开源的端到端流水线解决方案。我依稀记得，这个解决方案架构图是在北京开往上海的高铁上完成的。目前，这个方案在行业内广为流传，成为了很多公司搭建自己内部工具链平台的参考资料。这个系统的架构图如下：



今天，我会基于这个解决方案，给你介绍几个主要阶段的工具的使用技巧，希望可以手把手地帮助你快速搭建一套完整的持续交付平台。

**对于持续交付工具链体系来说，工具的连通性是核心要素**，所以我不会花太多时间介绍工具应该如何搭建，毕竟这类资料有很多，或者，你参考一下官网的搭建文档就可以了。尤其是现在很多工具都提供了容器化的部署方式，进一步简化了自身工具的建设成本。

## 需求管理 - Jira

在 Jira 官网上的醒目位置，写着一句话：**敏捷开发工具的第一选择**。在我看来，Atlassian 公司的确有这个底气，因为 **Jira 确实足够优秀**，跟 **Confluence** 的组合几乎已经成为了**很多企业的标配**。这也是为什么我没有选择开源工具 Redmine 或者其他诸如 Teambition 等的 SaaS 化服务。

当然，近些年来，各大厂商也在积极地对外输出研发工具能力，以腾讯的 TAPD 为代表的敏捷协同开发工具，就使用得非常广泛。但是，其实产品的思路都大同小异，搞定了 Jira，其他工具基本也就不在话下了。

作为敏捷协同工具，Jira 新建工程可以选择团队的研发模式是基于 Scrum，还是看板方法，你可以按需选择。在专栏的 [🔗第 8 讲](#)和 [🔗第 9 讲](#)中，我给你介绍了精益看板，你完全可以在 Jira 中定制自己团队的可视化看板。

看板的配置过程并不复杂，我把它整理成了文档，你可以点击网盘 [🔗链接](#)获取，提取码是 mrttd。需要提醒你的一点是：**别忘了添加 WIP 在制品约束，别让你的精益看板变成了可视化看板。**

需求作为一切开发工作的起点，是贯穿整个研发工作的重要抓手。**对于 Jira 来说，重点是要实现跟版本控制系统和开发者工具的打通。**接下来，我们分别来看下应该如何实现。

如果你也在用特性分支开发模式，你应该知道，一个特性就对应到一个 Jira 中的任务。通过任务来创建特性分支，并且将所有分支上的提交绑定到具体任务上，从而建立清晰的特性代码关联。我给你推荐两种实现方式。

第一种方式是基于 Jira 提供的原生插件，比如 [🔗Git Integration for Jira](#)。这个插件配置起来非常简单，你只需要添加版本控制系统的地址和认证方式即可。然后，你就可以在 Jira 上进行查看提交信息、对比差异、创建分支和 MR 等操作。但是这个插件属于收费版本，你可以免费使用 30 天，到期更新即可。

DigMyData / BTHREE-6999

There is an exception in `send-notifications` when integrations does not have data for previous week

Edit Comment Assign Changes required Close Issue Admin

Type: Bug Status: MERGED (View workflow)  
Priority: History Resolution: Unresolved  
Affects Version/s: None Fix Version/s: None  
Component/s: Admin tools  
Labels: None  
Technology: Java  
Severity: Major

Assignee: Adam Wride  
Reporter: Alexander Malyskhin (inactive)  
Votes: 0 Vote for this issue  
Watchers: 4 Start watching this issue  
Created: 02/Jun/14 2:58 PM  
Updated: 6 minutes ago

Description

Steps to Reproduce

1. Log in to the test server
2. Execute the following commands:

```
cd /home/dmd/bis  
./admin.sh send-notifications 20140615-1800 2544 2546
```

Actual result

The user gets `java.lang.IndexOutOfBoundsException` exception  
<http://screencast.com/t/oLxEuntoEtZK>

Expected result

There should be no exception

Attachments

Drop files to attach, or browse.

Issue links

relates to

BTHREE-6755 Standardize all email sends into a single design

Git Commits

9 commits  
Roll Up

Branches

Create branch

Pull Requests

Create pull request

Tags

dmd-20140717  
dmd-20140717  
dmd-20140717

第二种方式，就是使用 Jira 和 GitLab 的 Webhook 进行打通。

首先，你要在 GitLab 项目的“设置 - 集成”中找到 Jira 选项，按下图添加相应配置即可。配置完成之后，你只需要在提交注释中添加一个 Jira 的任务 ID，就可以实现 Jira 任务和代码提交的关联，这些关联体现在 Jira 任务的 Issue links 部分。

另外，你也可以实现 Jira 任务的状态自动流转操作，无需手动移动任务卡片。我给你提供一份 [配置说明](#)，你可以参考一下。



Active	<input checked="" type="checkbox"/>
Trigger	<div><input checked="" type="checkbox"/> <b>Commit</b> JIRA comments will be created when an issue gets referenced in a commit.</div> <div><input checked="" type="checkbox"/> <b>Merge request</b> JIRA comments will be created when an issue gets referenced in a merge request.</div>
Web URL	<input type="text" value="http://dms.m.jd.com"/>
JIRA API URL	<input type="text" value="http://dms.m.jd.com"/>
Username or Email	<input type="text" value="shixuefeng1"/>
Enter new password or api token	<input type="password" value="....."/>
Transition ID(s)	<input type="text" value="21"/>

不过，如果只是这样的话，还不能实现根据 Jira 任务来自动创建分支，所以接下来，还要进行 Jira 的 Webhook 配置。在 Jira 的系统管理界面中，你可以找到“高级设置 - Webhook”选项，添加 Webhook 后，可以绑定各种系统提供的事件，比如创建任务、更新任务等，这基本可以满足绝大多数场景的需求。

假设我们的系统在创建 Jira 任务的时候，要自动在 GitLab 中基于主线创建一条分支，那么你可以将 GitLab 提供的创建分支 API 写在 Jira 触发的 Webhook 地址中。参考样例如下：

```
https://这里替换成你的 GitLab 服务地址/repository/branches?  
branch=${issue.key}&ref=master&private_token=[这里替换成你的账号 Token]
```

Name\*

Status\*

URL\*

You can use the following additional variables in the URL: `${board.id}`, `${comment.id}`, `${issue.id}`, `${issue.key}`, `${mergedVersion.id}`, `${modifiedUser.key}`, `${modifiedUser.name}`, `${project.id}`, `${project.key}`, `${sprint.id}`, `${version.id}`  
[Read more](#)

Description

Events **Issue related events**  
Events for issues and worklogs. You can specify a JQL query to send only events triggered by matching issues.

☒ All issues

[Syntax help](#)

Comment	Worklog	Issue link	Issue
<input type="checkbox"/> created	<input type="checkbox"/> created	<input type="checkbox"/> created	<input checked="" type="checkbox"/> created
<input type="checkbox"/> updated	<input type="checkbox"/> updated	<input type="checkbox"/> deleted	<input type="checkbox"/> updated
<input type="checkbox"/> deleted	<input type="checkbox"/> deleted		<input type="checkbox"/> deleted
			<input type="checkbox"/> worklog changed

到这里，Jira 和 GitLab 的打通就完成了。我们来总结下已经实现的功能：

1. GitLab 每次代码变更状态都会同步到 Jira 任务中，并且实现了 Jira 任务和代码的自动关联 (Issue links) ；
2. 可以在 MR 中增加关键字 Fixes/Resolves/Closes Jira 任务号，实现 Jira 的自动状态流转；
3. 每次在 Jira 中创建任务时，都会自动创建特性分支。

关于 Jira 和开发者工具的打通，我把操作步骤也分享给你。你可以点击 [网盘链接](#) 获取，提取码是 kf3t。现在很多工具平台的建设都是以服务开发者为导向的，所以距离开发者最近的 IDE 工具就成了新的效率提升阵地，包括云 IDE、IDE 插件等，都是为了方便开发者可以在 IDE 里面完成所有的日常任务，对于管理分支和 Jira 任务，自然也不在话下。

## 代码管理 - GitLab

这个示例项目中的开发流程是怎样的呢？我们一起来看下。

第 1 步：在需求管理平台创建任务，这个任务一般都是可以交付的特性。你还记得吗？通过前面的步骤，我们已经实现了自动创建特性分支。

第 2 步：开发者在特性分支上进行开发和本地自测，在开发完成后，再将代码推送到特性分支，并触发提交阶段的流水线。这条流水线主要用于**快速验证提交代码的基本质量**。

第 3 步：当提交阶段流水线通过之后，开发者创建合并请求（Merge Request），申请将特性分支合并到主干代码中。

第 4 步：代码评审者对合并请求进行 Review，发现问题的话，就在合并请求中指出来，最终接受合并请求，并将特性代码合入主干。

第 5 步：代码合入主干后，立即触发集成阶段流水线。这个阶段的检查任务更加丰富，测试人员可以手动完成测试环境部署，并验证新功能。

第 6 步：特性经历了测试环境、预发布环境，并通过部署流水线最终部署到生产环境中。

在专栏的 [🔗第 12 讲](#) 中，我提到过，持续集成的理念是通过尽早和及时的代码集成，从而建立代码质量的快速反馈环。所以，**版本控制系统和持续集成系统也需要双向打通**。

这里的双向打通是指版本控制系统可以触发持续集成系统，持续集成的结果也需要返回给版本控制系统。

接下来，我们看看具体怎么实现。

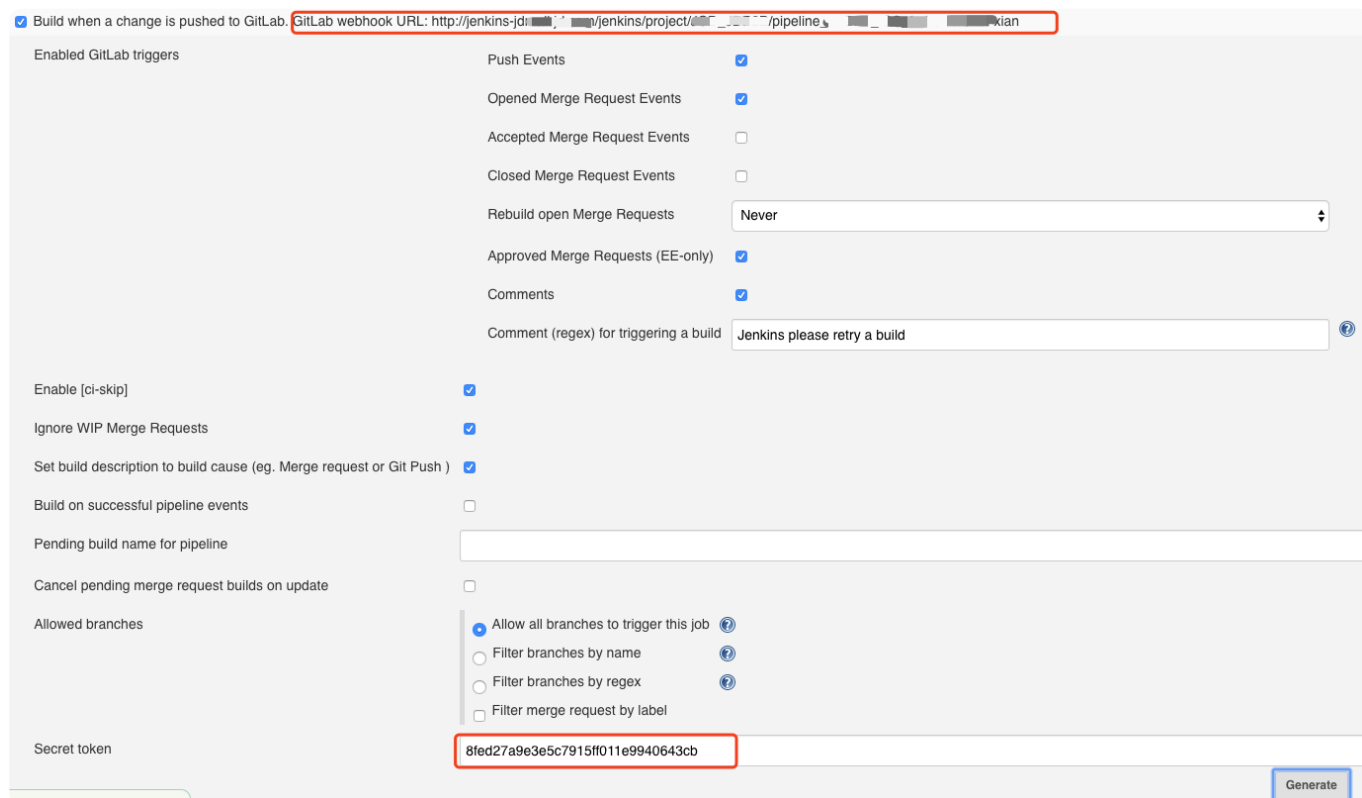
## 代码提交触发持续集成

首先，你需要在 Jenkins 中安装 [🔗GitLab 插件](#)。这个插件提供了很多 [🔗GitLab 环境变量](#)，用于获取 GitLab 的信息，比如，gitlabSourceBranch 这个参数就非常有用，它可以提取本次触发的 Webhook 的分支信息。毕竟，这个信息只有 GitLab 知道。只有同步给 Jenkins，才能拉取正确的分支代码执行持续集成过程。

当 GitLab 监听到代码变更的事件后，会自动调用这个插件提供的 Webhook 地址，并实现解析 Webhook 数据和触发 Jenkins 任务的功能。

其实，我们在自研流水线平台的时候，也可以参考这个思路：**通过提供 Webhook 接口并注册到 GitLab 中，从而实现代码变更事件的监听和自动任务执行。**

当 GitLab 插件安装完成后，你可以在 Jenkins 任务的 Build Triggers 中发现一个新的选项，勾选这个选项，就可以激活 GitLab 自动触发配置。其中比较重要的两个信息，我在下面的图片中用红色方块圈出来了。



Build when a change is pushed to GitLab. GitLab webhook URL: `http://jenkins-jd:8080/jenkins/project/.../pipeline...`

Enabled GitLab triggers

- Push Events ☒
- Opened Merge Request Events ☒
- Accepted Merge Request Events ☐
- Closed Merge Request Events ☐
- Rebuild open Merge Requests
- Approved Merge Requests (EE-only) ☒
- Comments ☒
- Comment (regex) for triggering a build

Enable [ci-skip] ☒

Ignore WIP Merge Requests ☒

Set build description to build cause (eg. Merge request or Git Push) ☒

Build on successful pipeline events ☐

Pending build name for pipeline

Cancel pending merge request builds on update ☐

Allowed branches

- ☒ Allow all branches to trigger this job
- ☐ Filter branches by name
- ☐ Filter branches by regex
- ☐ Filter merge request by label

Secret token

Generate

上面的链接就是 Webhook 地址，每个 Jenkins 任务都不相同；

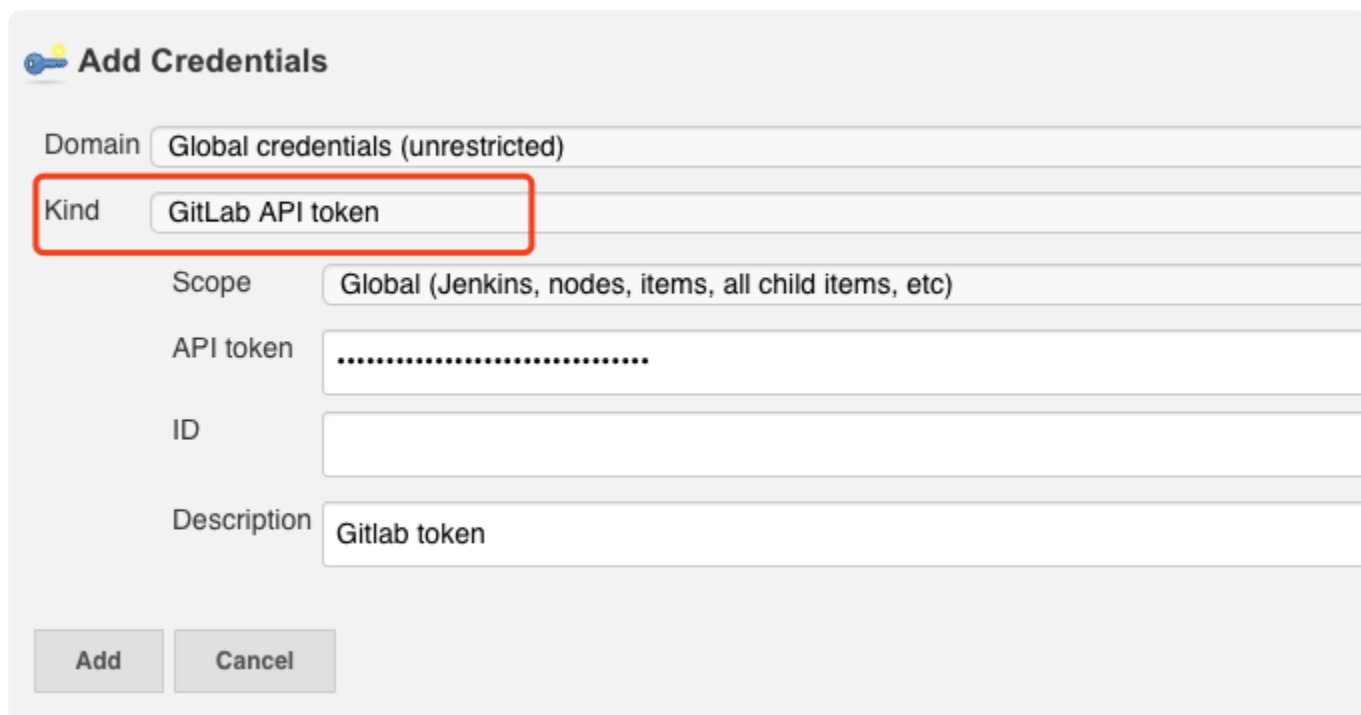
下面的是这个 Webhook 对应的认证 Token。

你需要把这两个信息一起添加到 GitLab 的集成配置中。打开 GitLab 仓库的“设置 - 集成”选项，可以看到 GitLab 的 Webhook 配置页面，将 Jenkins 插件生成的地址和 Token 信息复制到配置选项中，并勾选对应的触发选项。

GitLab 默认提供了多种触发选项，在下面的截图中，只勾选了 Push 事件，也就是只有监听到 Git Push 动作的时候，才会触发 Webhook。当然，你可以配置监听的分支信息，只针对特性分支执行关联的 Jenkins 任务。在 GitLab 中配置完成后，可以看到新添加的 Webhook 信息，点击“测试”验证是否可以正常执行，如果一切正常，则会提示“200-OK”。







**Add Credentials**

Domain: Global credentials (unrestricted)

Kind: **GitLab API token**

Scope: Global (Jenkins, nodes, items, all child items, etc)

API token: .....

ID:


Description: Gitlab token

Add Cancel

那么，配置完成后，要如何更新 GitLab 的提交状态呢？这就需要用到插件提供的 [更新构建结果](#) 命令了。

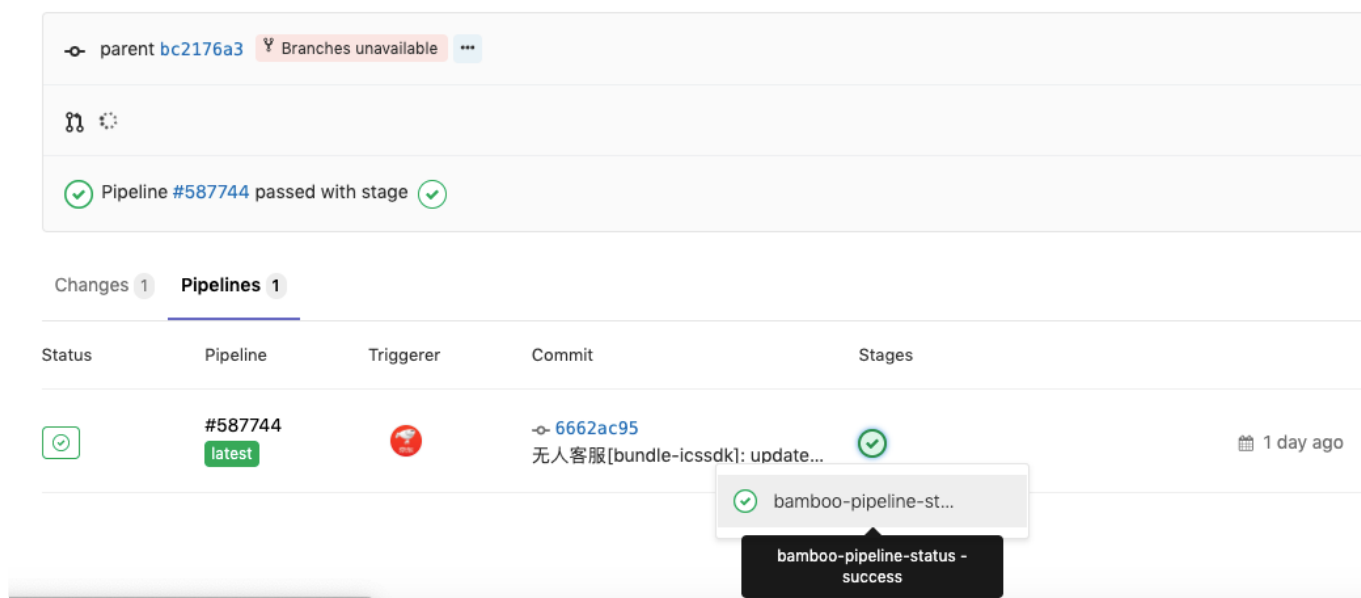
对于自由风格类型的 Jenkins 任务，你可以添加构建后处理步骤 - Publish build status to GitLab，它会自动将排队的任务更新为 “Pending”，运行的任务更新为 “Running”，完成的任务根据结果更新为 “Success” 或者是 “Failed”。

对于使用流水线的任务来说，官方也提供了相应的 [示例代码](#)，你只需要对照着写在 Jenkinsfile 里面就可以了。

 复制代码

```
1 updateGitlabCommitStatus name: 'build', state: 'success'
```

这样一来，每次提交代码触发的流水线结果也会显示在 GitLab 的提交状态中，可以在查看合并请求时作为参考。有的公司更加直接：如果流水线的状态不是成功状态，那么就会自动关闭提交的合并请求。其实无论采用哪种方式，初衷都是**希望开发者在第一时间修复持续集成的问题**。



我们再阶段性地总结一下已经实现的功能：

1. 每次 GitLab 上的代码提交都可以通过 Webhook 触发对应的 Jenkins 任务。具体触发哪个任务，取决于你将哪个 Jenkins 任务的地址添加到了 GitLab 的 Webhook 配置中；
2. 每次 Jenkins 任务执行完毕后，会将执行结果写到 GitLab 的提交记录中。你可以查看执行状态，决定是否接受合并请求。

## 代码质量 - SonarQube

SonarQube 作为一个常见的开源代码质量平台，可以用来实现静态代码扫描，发现代码中的缺陷和漏洞，还提供了比较基础的安全检查能力。除此之外，它还能收集单元测试的覆盖率、代码重复率等。

对于刚开始关注代码质量和技术债务的公司来说，是一个比较容易上手的选择。关于技术债务，在专栏的 [第 15 讲](#) 中有深入讲解，如果你不记得了，别忘记回去复习一下。

那么，代码质量检查这类频繁执行的例行工作，也比较适合自动化完成，**最佳途径就是集成到流水线中，也就是需要跟 Jenkins 进行打通**。我稍微介绍一下执行的逻辑，希望可以帮你更好地理解这个配置的过程。

SonarQube 平台实际包含两个部分：

一个是平台端，用于收集和展示代码质量数据，这也是我们比较常用的功能。

另外一个是客户端，也就是 SonarQube 的 Scanner 工具。这个工具是在客户端本地执行的，也就是跟代码在一个环境中，用于真正地执行分析、收集和上报数据。这个工具之所以不是特别引人注意，是因为在 Jenkins 中，后台配置了这个工具，如果发现节点上没有找到工具，它就会自动下载。你可以在 Jenkins 的全局工具配置中找到它。

## SonarQube Scanner

SonarQube Scanner installations

Add SonarQube Scanner



SonarQube Scanner

Name

sonar-scanner-4.2.0



Install automatically



Install from Maven Central

Version

SonarQube Scanner 4.2.0.1873

了解了代码质量扫描的执行逻辑之后，我们就可以知道，对于 SonarQube 和 Jenkins 的集成，只需要单向进行即可。这也就是说，只要保证 Jenkins 的 Scanner 工具采集到的数据可以正确地上报到 SonarQube 平台端即可。

这个配置也非常简单，你只需要在 Jenkins 的全局设置中添加 SonarQube 的平台地址就行了。注意勾选第一个选项，保证 SonarQube 服务器的配置信息可以自动注入流水线的环境变量中。

## SonarQube servers

Environment variables

☒ Enable injection of SonarQube server configuration as build environment variables

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

SonarQube installations

Name

SonarQube-777C...

Server URL

http://192.168.1.100:9000/

Default is http://localhost:9000

Server authentication token

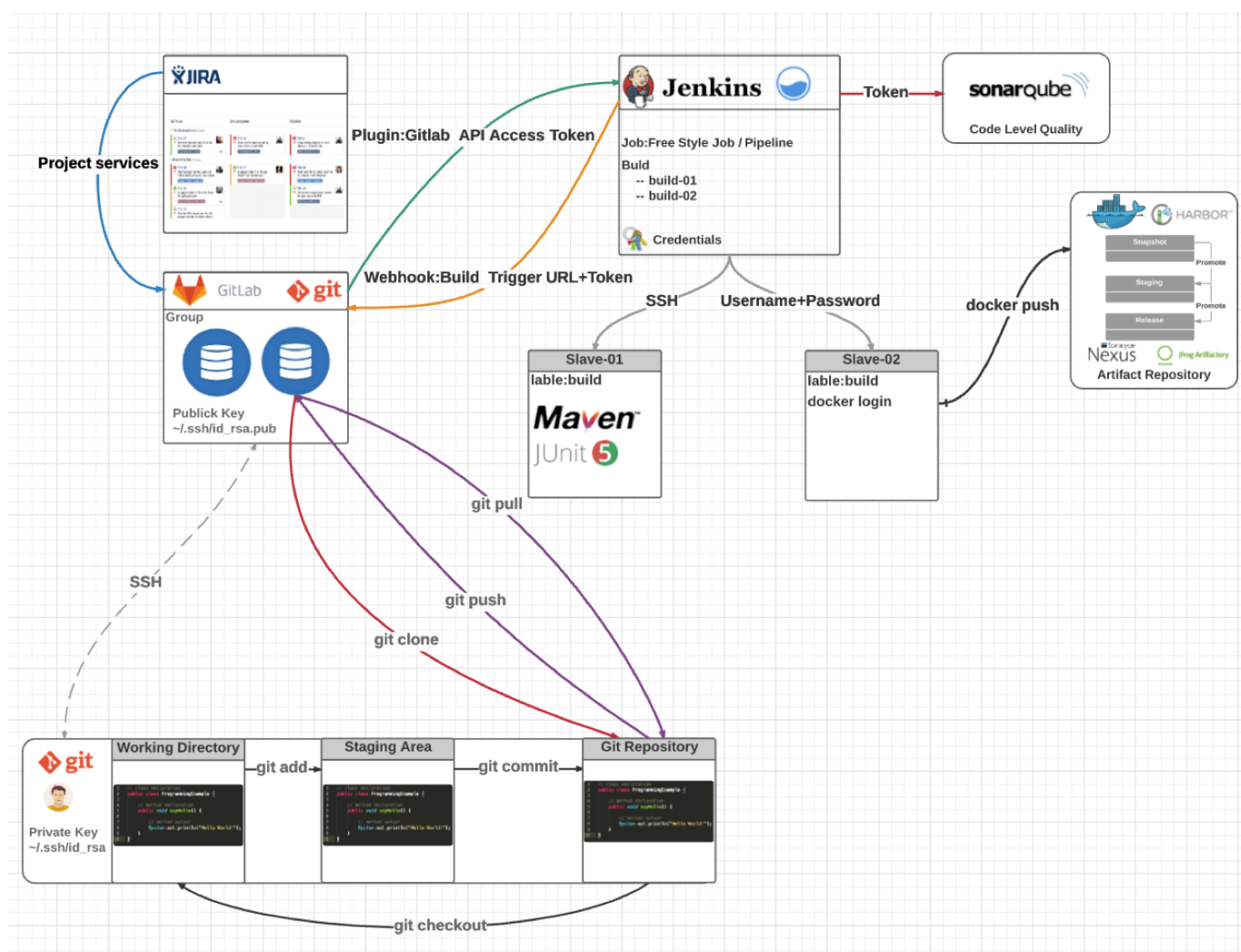
sonar

Add

SonarQube authentication token. Mandatory when anonymous access is disabled.

在执行 Jenkins 任务的时候，同样可以针对自由风格的任务和流水线类型的任务，添加不同的上报方式。关于具体的内容，你可以参考 SonarQube 的 [官方网站](#)，这里就不赘述了。

到此为止，我们已经实现了 GitLab、Jenkins 和 SonarQube 的打通。我给你分享一幅系统关系示意图，希望可以帮助你更好地了解系统打通的含义和实现过程。



## 环境管理 - Kubernetes

最后，我们再来看看环境管理部分。作为云原生时代的操作系统，Kubernetes 已经成为了云时代容器编排的事实标准。对于 DevOps 工程师来说，Kubernetes 属于必学必会的技能，这个趋势已经非常明显了。

在示例项目中，我们同样用到了 Kubernetes 作为基础环境，所有 Jenkins 任务的环境都通过 Kubernetes 来动态初始化生成。

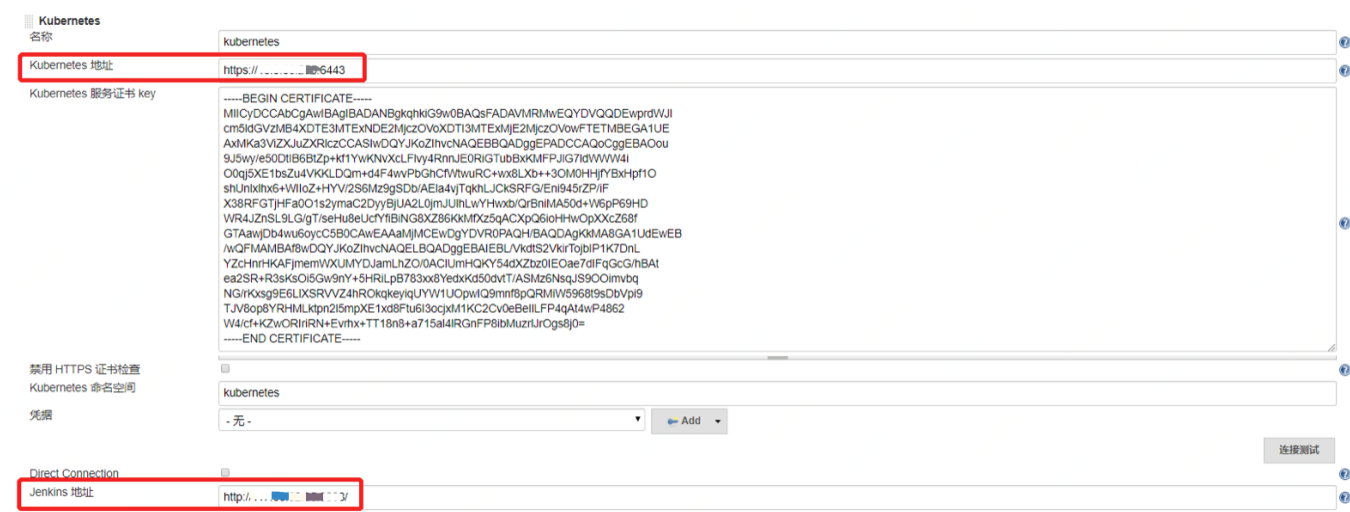
这样做的好处非常多。一方面，可以实现环境的标准化。所有环境配置都是以代码的形式写在 Dockerfile 中的，实现了环境的统一可控。另一方面，环境的资源利用率大大提升，不再依托于宿主机自身的环境配置和资源大小，你只需要告诉 Kubernetes 需要多少资源，它就会帮助你找到合适的物理节点运行容器。资源的调度和分配统一通过 Kubernetes 完



成，这就进一步提升了资源的有效利用率。想要初始化一套完整的环境，对于中小系统来说，是分分钟就可以完成的事情。关于这一点，我会在讲“云原生时代应用的平台建设”时跟你探讨。

那么，想要实现动态初始化环境，需要打通 Jenkins 和 Kubernetes。好在 Jenkins 已经提供了官方的 [Kubernetes 插件](#)来完成这个功能。你可以在 Jenkins 系统配置中添加云 - Kubernetes，然后再参考下图进行配置。

需要注意的是，必须正确配置 Jenkins 的地址（系统配置 - Jenkins Location），否则会导致新建容器无法连接 Jenkins。



生成动态节点时，需要使用到 JNLP 协议，我推荐你使用 Jenkins 官方提供的镜像。

JNLP 协议的全称是 Java Network Launch Protocol，是一种通用的远程连接 Java 应用的协议方式。典型的使用场景就是在构建节点（也就是习惯上的 Slave 节点）上发起向 Master 节点的连接请求，将构建节点主动挂载到 Jenkins Master 上，供 Master 调度使用。区别于使用 SSH 长连接的方式，这种动态连接的协议特别适合于 Kubernetes 这类的动态节点。镜像配置如下图所示：

Pod Template

名称

命名空间

标签列表

用法

父级的 Pod 模板名称

容器列表

环境变量

卷

jnlp-slave

jnlp-slave

Use this node as much as possible

Container Template

名称

jnlp

Docker 镜像

jenkins/jnlp-slave:alpine

总是拉取镜像

工作目录

/home/jenkins/agent

运行的命令

命令参数

\$(computer.jnlpmac) \$(computer.name)

分配伪终端

EnvVars

添加环境变量

设置到 Pod 节点中的环境变量列表

添加容器

Pod 代理中的容器列表

添加环境变量

该 Pod 中所有容器的环境变量

添加卷

挂载到 Pod 代理中的卷列表

Advanced...

删除容器

在配置动态节点的时候，有几个要点你需要特别关注下。

- 静态目录挂载。**由于每次生成一个全新的容器环境，所以就需要将代码缓存（比如.git 目录）、依赖缓存（.m2, .gradle, .npm）以及外部工具等静态数据通过 volume 的方式挂载到容器中，以免每次重新下载时影响执行时间。
- 如果你的 Jenkins 也是在 Kubernetes 中运行的，注意**配置 Jenkins 的 JNLP 端口号**（使用环境变量：JENKINS\_SLAVE\_AGENT\_PORT）。否则，在系统中配置的端口号是不会生效的。
- 由于每次初始化容器有一定的时间损耗，所以你可以**配置一个等待时长**。这样一来，在任务运行结束后，环境还会保存一段时间。如果这个时候有新任务运行，就可以直接复用已有的容器环境，而无需重新生成。
- 如果网络条件不好，可以适当地加大创建容器的超时时间**，默认是 100 秒。如果在这个时间内无法完成容器创建，那么 Jenkins 就会自动杀掉创建过程并重新尝试。

如果一切顺利，动态 Kubernetes 环境就也可以使用了。这时，我们就可以完整地运行一条流水线了。在设计流水线的时候，你需要注意的是**流水线的分层**。具体的流水线步骤，我已经写在了系统架构图中。比如，提交阶段流水线需要完成拉取代码、编译打包、单元测试和代码质量分析四个步骤，对应的代码如下：

复制代码

```
1 // pipeline 2.0 - Commit stage - front-end
```

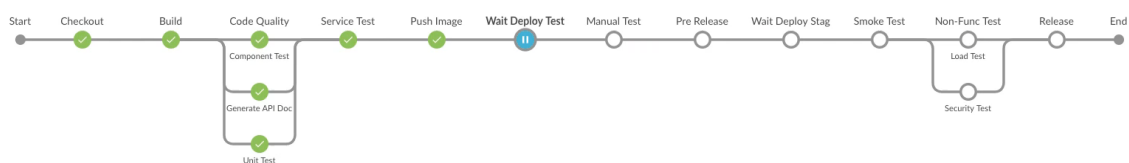


```

54         sh "${scannerHome}/bin/sonar-scanner"
55         updateGitlabCommitStatus name: 'build', state:
56     }
57 }
58 }
59 }
60 }
61 }
62 }
63 }

```

完整的流水线演示效果如下：



结合 Jenkins 自身的人工审批环节，可以实现多环境的自动和手动部署，构建一个真正的端到端持续交付流水线。

## 总结

在今天的课程中，我通过一个开源流水线的解决方案，给你介绍了如何建立一个开源工具为主的持续交付流水线平台。你应该也有感觉，对于 DevOps 来说，真正的难点不在于工具本身，而在于如何基于整个研发流程将工具串联打通，把它们结合在一起，发挥出最大的优势。这些理念对于自建平台来说也同样适用，你需要在实践中多加尝试，才能在应用过程中游刃有余。

## 思考题

关于这套开源流水线解决方案，你对整体的工具链、配置、设计思路还有什么疑问吗？在实施过程中，你遇到了哪些绕不过去的问题呢？

欢迎在留言区写下你的思考和答案，我们一起讨论，共同学习进步。如果你觉得这篇文章对你有所帮助，也欢迎你把文章分享给你的朋友。

# DevOps 实战笔记

精要 30 计，让 DevOps 快速落地

石雪峰

京东商城工程效率专家



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | 平台产品研发：三个月完成千人规模的产品要怎么做？

下一篇 28 | 迈向云端：云原生应用时代的平台思考

## 精选留言 (2)

写留言



leslie

2019-12-19

今天的内容就是一个精简版的可以直接上手使用的DevOps：不过篇章划分感觉错了-应当是《案例总结/分析》篇之类的，一个不错的经典小型实战类项目，正找案例呢；刚好后面可以拿来可以用。

不知不觉就33讲了，好快啊；谢谢今天的分享，期待后续的内容。

展开



1



swordman

2019-12-19

终于等来了工具串联环节，正是我想要的，写得非常清楚，值得收藏。还有一个问题，就是Jenkins和Jira连通的场景，能否能再分享一下。附上我们现在实现：（1）代码提交流水



线失败，自动往Jira上添加一个故障，然后开发人员在开发IDE中选择这个故障，展开开发和提交工作（2）Jira任务变更为完成时，自动触发Jenkins流水线，流水线完成拉取分支代码，编译打包，自动化测试等工作，如果运行失败，则修改Jira任务状态为进行中。不知...  
展开 ∨

