

## Rancher 企业容器云平台

### 基于 Docker 的构建流程（第一部分）- 持续集成及测试

Usman Ismail, Bilal Sheikh (Rancher labs) 于 2016.1.6

译者：雷伟（有容云 Yourun Cloud）



了解最新云计算资讯，关注有容云官方微信

## 前言

在过去的一年里，我们写了很多文章，关于如何在 Docker 上运行不同 Stack(译者注：在 Rancher 中，一个 Stack 通常对应一个应用，它包含了一系列容器，及容器之间的关系描述)，如：Magento, Jenkins, Prometheus 等。然而，[容器化](#)部署不仅仅用于定义应用 Stack。在本系列文章中，我们将讨论端到端的开发流程，包括在流程的各个阶段如何平衡 Docker 和 Rancher。具体涉及到：代码构建，测试，打包，持续集成及部署，以及在生产环境中管理应用 Stack。您也可以同时[下载](#)本系列的电子书。

首先，我们从代码构建开始。一般来说，代码的构建/编译不是很大的问题，因为大部分语言和编译代码的工具已很清楚，且有很完善的文档说明。然后，随着项目和团队规模增长，模块之间依赖关系变得复杂，如何确保代码质量的同时，保证代码构建的一致性和稳定性，将成为更大的挑战。在本文中，我们将讨论如何用 Docker 去实现 CI（持续集成）和测试的最佳实践。

## 构建系统扩展带来的挑战

首先，我们来看下维护构建系统所面临的一些挑战：

第一是依赖管理：当开发人员将库集成到源代码中时，需要注意的是，如何保证项目中所有模块使用的同一个库版本，且当库版本升级时，如何能及时将新版本提交到项目中的所有模块。

第二是环境依赖管理。包括 IDE 及 IDE 配置，工具版本（如 Maven, Python 版本）及相关配置，如代码静态分析规则，代码格式化模版。环境依赖管理的麻烦在于，项目中的不同模块会有依赖冲突，与代码级冲突不同的是，模块依赖冲突更难或是不能解决。比如，最近的一个项目里，我们用 Fabric 进行自动化部署，用 s3cmd 上传 Artifacts 到 Amazon S3。不幸的是，Fabric 的最新版本需要 Python 2.7，但 S3cmd 需要 Python2.6。如要兼顾两者，我们要么切换到 s3cmd Beta 版本，要么用 Fabric 的老版本。

最后，每个大型项目需要面临的一个主要问题是构建时间。随着项目规模扩大、复杂性增加，需要的语言也越来越多（我当前的项目使用 Java, Groovy, Python 和 Protocol Buffers IDL）。测试中不同组件也会相互依赖，比如，测试时不能在同一时间运行共享数据库中相同的数据（译者注：同样的系统环境，同一共享数据库，不同人员同时测试时可能会引起数据冲突，后面实例会提到如添加同一用户时，会导致用户冲突）。这样，我们需要在测试执行前确保初始状态，测试完后再清理状态。这将减慢开发进度。

## 解决方案和最佳实现

为解决上述问题，一个好的构建系统需要达到如下要求：

- **可重复性**

能在不同的开发机器和自动构建服务器中，生成/创建有一致依赖关系的构建环境。

- **集中化管理**

所有开发机器和构建服务器的构建环境，是来自于同一个代码仓库中心或服务器，且环境设置能及时更新。

- **隔离性**

项目的各个子模块相互隔离，而不是相互依赖。

- **并行性**

并行构建子模块，提高构建效率。

### 可重复性

大多数语言和开发框架支持自动化依赖管理。如 Maven 用于 Java，Python 用 PIP，Ruby 用 Bundler。这些工具比较类似，当你提交一个索引文件（`pop.xml`, `requirements.txt` 或 `Gemfile`）到源代码控制器中，工具会自动下载相关依赖到构建机器中。我们测试后需集中管理这些文件，并提交到源代码控制服务器中。然而，仍需管理环境依赖，如是否安装了 Maven, Python, Ruby 的正确版本。Maven 自动检测依赖的更新，但对 PIP 和 Bundler，我们必须通过脚本来触发更新。

## 集中化管理

为了安装依赖管理工具和脚本，大部分小团队通过文档来描述。当团队扩张时，如何保持实时依赖更新将会是关键。另外，构建环境的 OS 和平台不同，也会引起工具的安装差异。当然，你可以用配置管理工具，如 Puppet 或 Chef（译者注：Puppet 和 Chef 均为跨平台配置管理工具）去管理安装包的依赖和配置文件的设置；提前测试后再提交并推送到所有开发者。同时 Puppet 和 Chef 也有一些不足：

1. 安装配置不简单，且完整功能的版本都是收费的；
2. 有自己单独的语言进行配置（译者注：如 Chef 基于 Ruby 语言）；
3. 配置管理工具不具备隔离功能，工具版本冲突和并行测试仍是一个问题；

## 隔离性

为保证组建隔离，减少构建时间，可以用虚拟机自动化系统，如 Vagrant（译者注：Vagrant 是一个基于 Ruby 的工具，用于创建和部署虚拟化开发环境）。Vagrant 能创建并运行不同组建环境的虚拟机（盒子），这样可以保证隔离和并行测试。Vagrant 的配置文件提交到源代码控制器中，且推送到所有开发人员。另外，虚拟机（盒子）能用于测试，部署到 Atlas，便于开发人员下载。不足在于：

1. 将需要更高层次的配置描述去设置 Vagrant；

2. 虚拟机也是一个非常重量级的隔离解决方案：虽然只需要一个测试运行环境或编译环境，但每个虚拟机运行了一个完整的 OS 和网络栈，还需提前分配内存和磁盘资源；

尽管有各种问题，用依赖管理工具（Maven, PIP, Rake），配置管理工具(Puppet, Chef) 和虚拟化工具（Vagrant），也能建立一个稳定、集中管理的构建系统，不是所有的项目都需要所有这些工具，但是任何长时间运行的大项目都需要自动化到这个程度。

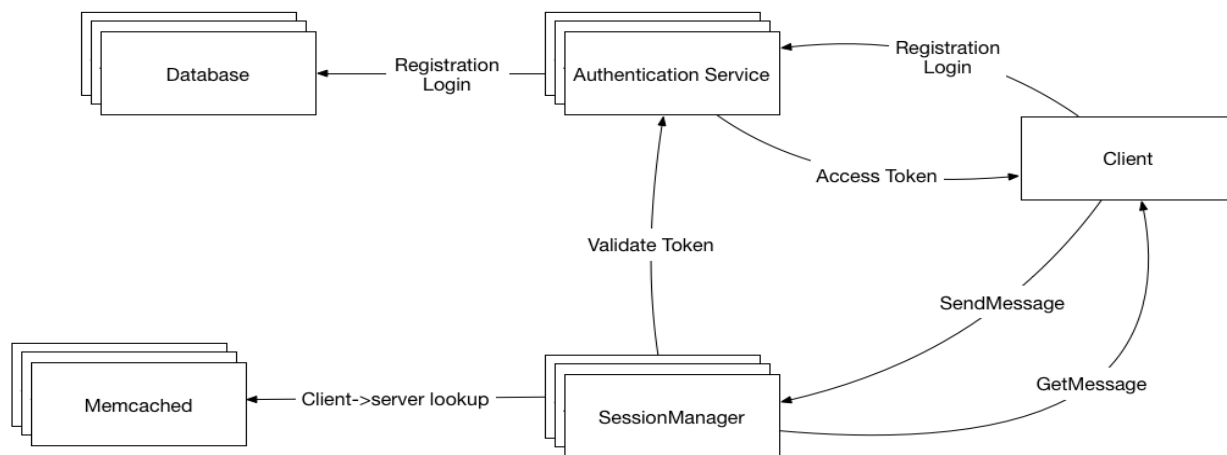
## 利用 Docker 进行系统构建

无需投入大量时间和资源，Docker 和其生态系统能帮助我们支持上述工具。在本节中，我们将通过如下步骤来为应用创建集中化构建环境。

- 1. 集中化构建环境
- 2. 用 Docker 打包应用
- 3. 用 Docker Compose 生成构建环境

在本文及后续的文章中，为方便描述，我们将使用 Go-Messenger 作为示例应用。

在 GitHub 上[下载](#)此应用。此系统的主要数据流如下图所示，由两个组件构成：RESTful 认证服务器，用 Golang 所写；会话管理，用于接收来自客户端的 TCP 长连接和客户端之间的路由信息。本文将重点关注



RESTful 认证服务（[Go-Auth](#)）。它包含了多个无状态的 Web 服务器和数据库集群，数据库集群用于存储用户信息。

## 1. 集中化构建系统

首先，需要创建一个包含了构建系统所需工具的容器镜像，此镜像的 Dockerfile 如下所示，也可在此处[下载](#)。因为应用是用 Go 所写，所以我们是基于官方的 Golang 镜像，安装了 Godep 依赖管理工具。如果你的项目是用 Java 语言，同样你可以基于 Java 基本镜像，安装 Maven 来代替 Godep。

```

from golang:1.4

# Install godep

RUN go get github.com/tools/godep

Add compile.sh /tmp/compile.sh
  
```

```
CMD /tmp/compile.sh
```

然后添加一个编译脚本，包含了构建和测试代码的过程。脚本如下所示：

1. 使用 Godep Restore 下载依赖包；
2. 用 Go Fmt 格式化源码；
3. 用 Go Test 运行测试；
4. 用 Go Build 编译项目；

```
#!/bin/bash

set -e

# Set directory to where we expect code to be
cd /go/src/${SOURCE_PATH}

echo "Downloading dependencies"

godep restore

echo "Fix formatting"

go fmt ./...

echo "Running Tests"

go test ./...

echo "Building source"

go build

echo "Build Successful"
```

为确保可重复性，我们用 Docker 构建一个有版本的容器镜像，可以从 Dockerhub 上下载此镜像，或通过 Dockerfile 构建。至此，所有的开发人员（和构建机器）都能通过此容器，用以下命令来构建任何 Go 工程。

```
docker run --rm -it \  
  
-v $PWD:/go/src/github.com/[USERNAME]/[PROJECT]/[SUB-CDIRECTORY]/ \  
  
-e SOURCE_PATH=github.com/[USERNAME]/[PROJECT]/[SUB-CDIRECTORY]/ \  
  
usman/go-builder:1.4
```

上面这条命令，我们运行了一个 Docker，镜像为 usman/go-builder，版本 1.4。且通过 -v 将源代码 Mount 到容器中，通过 -e 配置了环境变量 SOURCE\_PATH。在此示例工程中，为测试 Go-Builder，你可用以下命令来产生一个名为 Go-Auth 的可执行文件，存放于 Go-Auth 工程的 Root 目录下：

```
git clone git@github.com:usmanismail/go-messenger.git  
  
cd go-messenger/go-auth  
  
docker run --rm -it \  
  
-v $PWD:/go/src/github.com/usmanismail/go-messenger/go-auth/ \  
  
-e SOURCE_PATH=github.com/usmanismail/go-messenger/go-auth/ \  
  
usman/go-builder:1.4
```



隔离构建工具带来的另一个好处是，可以很容易更换构建工具和其配置。如在上面的例子里，我们用的是 Golang1.4，用以上命令将 Go-Build:1.4 更改为 Go-Build:1.5，可以很快测试本工程中是否能使用 Golang1.5。

为集中管理镜像，我们可以将此构建容器的最新版本设置为固定版本，这样所有开发者可直接使用 `go-builder:latest` 来构建源代码。如果工程中用到构建工具的不同版本，使用不同的容器构建即可，不用担心在一个构建环境管理多个版本的问题。比如，用支持多版本的官方 Python 镜像，可以解决前面的 Python 问题（译者注：上文所描述的 Fabric 和 s3cmd 对 Python 不同版本的依赖问题：“Fabric 的最新版本需要 Python 2.7，但 s3cmd 需要 Python 2.6”）。

## 2. 用 Docker 打包应用

如果你想将二进制打包到容器，添加如下内容的 Dockerfile，运行 “`docker build -t go-auth`” 即可。在 Dockerfile 中，将二进制输出到一个新的容器；暴露 9000 端口来接入连接；配置运行二进制的入口参数。因为 Go 二进制是自包含的，我们用现有 Ubuntu 镜像即可。如你的项目需要一些依赖包，也可一同打包进容器。如生产一个 war 文件时就用 Tomcat 容器。

```
FROM ubuntu
ADD ./go-auth /bin/go-auth
EXPOSE 9000
ENTRYPOINT ["/bin/go-auth","-l","debug","run","-p","9000"]
```

### 3. 用 Docker Compose 创建 Build 环境

到现在为止，我们已经完成项目构建、实现可重复性、集中管理且隔离各种组件。我们还可以将构建流程扩展到集成测试中，这也突出了 Docker 并行化加速构建的能力。

测试不能并行的一个主要原因在于共享数据库。在本示例项目中，用 MySQL 存储用户信息，也存在着类似的问题。测试新用户注册时，第一次测试注册新用户，当运行第二次测试时，由于注册相同的用户而导致用户冲突错误。这就只能在完成一次测试后，清空注册用户再开始新一轮测试。

为设置隔离的并行构建，我们可以定义 Docker Compose 模版（docker-compose.yml），如下所示。其中定义了一个数据库服务，使用 MySQL 官方镜像并配置了一些环境变量。然后产生了一个 GoAuth 服务，用已将应用打包的容器，并将数据库容器连接到此容器中。

```
Database:

image: mysql

environment:

MYSQL_ROOT_PASSWORD: rootpass

MYSQL_DATABASE: messenger

MYSQL_USER: messenger

MYSQL_PASSWORD: messenger

expose:

- "3306"

stdin_open: true

tty: true
```

Goauth:

image: go-auth

ports:

- "9000:9000"

stdin\_open: true

links:

- Database:db

command:

- "--db-host"

- "db"

tty: true

通过运行 `docker-compose up`，先将应用环境跑起来，然后通过运行如下 `curl` 命令来模拟集成测试，第一次运行会返回 200 表示成功，第二次将返回 409 表示冲突。最后，运行 `docker-compose rm` 来清理应用环境。

```
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
```

为了运行应用多个相互独立的版本，需要更新 Docker Compose 模版，添加相同配置的服务 Database1 和 Goauth1，Goauth1 唯一需要更改的是端口从 9000:9000 到 9001:9000，保证应用暴露的端口不相冲突。完整的模版可在此[下载](#)。当再运行 `docker-compose up` 时，就能并行的运行两个集成测试了。同样的，当工程有多个独立的子模块时，此并行化的方式也可用于加速构建系统中，如多模块的 Maven 工程。

```
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
... 200 OK

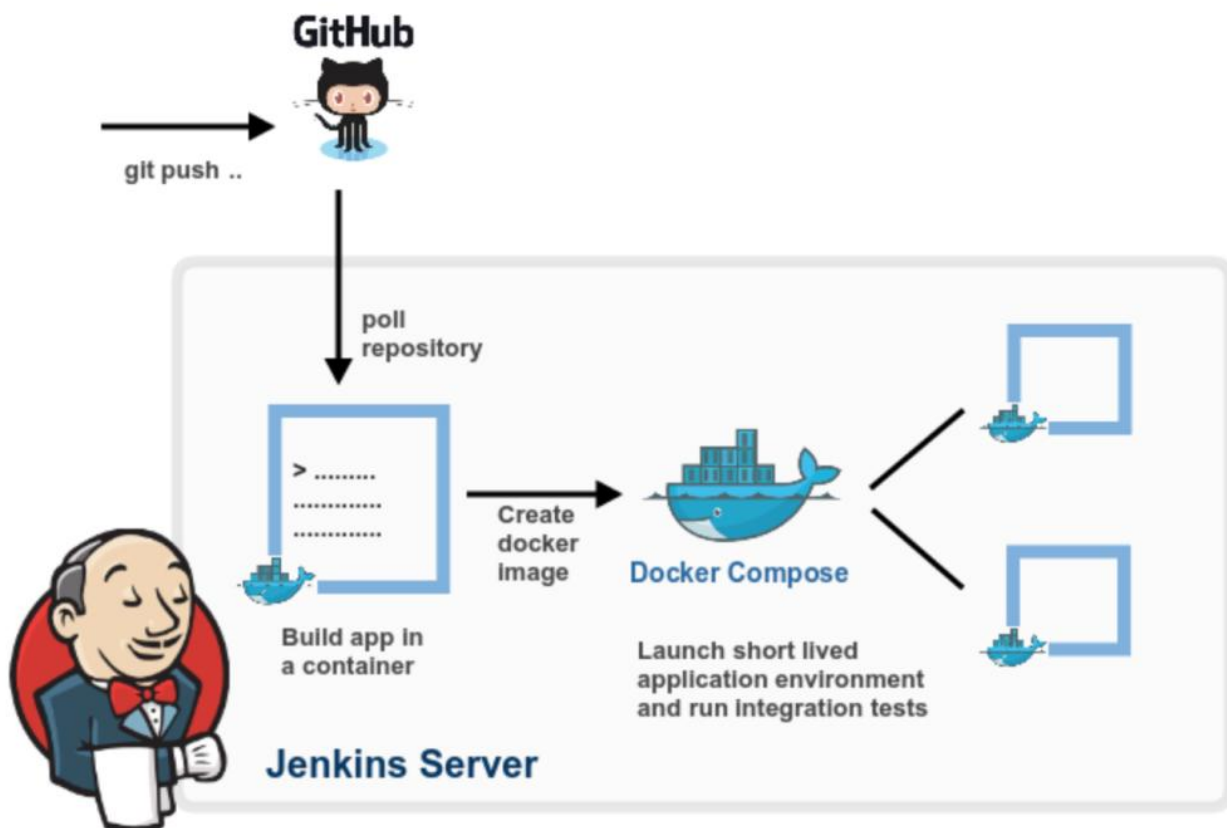
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9001/user
... 200 OK

curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9001/user
... 409 Conflict

curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
... 409 Conflict
```

## 通过 Docker 和 Jenkins 进行持续集成 ( CI )

现在我们将为示例应用创建持续集成 ( CI ) 流程。首先，我们先花一部分时间讨论如何进行代码分支。



## 1. 分支模型

在自动化持续集成中，需要重点考虑的是开发模型和团队。开发模型通常取决于团队如何使用版本控制系统。因为我们的应用托管在 Git 仓库中，所以使用 Git-Flow 模型，这种方式也很常用。

Git-Flow 模型中维护两种分支：开发（Develop）分支和主（master）分支。当加入一个新功能时，从开发分支中创建一个新的分支，当开发完成，将合并回开发分支。所有的功能分支由开发人员单独管理，一旦代码被提交到开发分支，CI（持续集成）服务器将负责编译，通过自动化测试，并提供一个服务器用于 QA 测试及评审。一旦需要发布版本，将开发分支合并到主分支中，本次合并提交会有一个版本号并打标签（tag），被标签的发布版本可用于 Beta 版本、模拟环境、或生产环境中。

以下，我们将用 GitFlow 工具管理 Git 分支。安装 Git-Flow，按此[说明](#)即可。安装好后，如下所示，运行 Git Flow init 来配置 Git 仓库，过程中提示问题选择默认即可。当执行 Git-Flow 命令时，它将创建一个开发分支（如果不存在的话），并将此作为当前工作分支。

```
$ git flow init

Which branch should be used for bringing forth production releases?

- master

Branch name for production releases: [master]

Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? []
```

现在通过命令 `Git flow feature start [feature-name]` 来创建一个新的功能，通常以 ticket/issue id 作为功能名。比如，如果你在用 Jira，且已有一个 Ticket，Ticket ID(如 MSP-123)可以作为功能名。当通过 Git-Flow 创建一个新的功能时，将会自动切换到此功能分支。

```
git flow feature start MSP-123

Switched to a new branch 'feature/MSP-123'
```

Summary of actions:

- A new branch 'feature/MSP-123' was created, based on 'develop'
- You are now on branch 'feature/MSP-123'

Now, start committing on your feature. When done, use:

```
git flow feature finish MSP-123
```

此时，你可以开发此新功能，然后运行自动化测试保证功能完成。更新 README 文件，通过运行 命令 `Git flow feature finish MSP-123`，即可完成新功能开发过程。

Switched to branch 'develop'

Updating 403d507..7ae8ca4

Fast-forward

README.md | 1 +

1 file changed, 1 insertion(+)

Deleted branch feature/MSP-123 (was 7ae8ca4).

Summary of actions:

- The feature branch 'feature/MSP-123' was merged into 'develop'
- Feature branch 'feature/MSP-123' has been removed
- You are now on branch 'develop'

注：Git Flow 会将此功能合并到开发分支中，并删除此功能分支，将当前工作环境切换到开发分支。

此时你可将开发分支提交到远程仓库中（命令 `Git push origin develop:develop`）。当提交时，CI 持续集成服务器将启动持续集成流程。

注：对大型项目而言，步骤会有所不同，一般是先评审代码，将远程代码合入开发分支后，再将开发分支提交到远程仓库。

## 2. 用 Jenkins 创建 CI 流程

本节我们假设你已将 Jenkins 集群运行起来了。如果没有，可以阅读[此文章](#)完成设置。除此之外，还需要安装如下插件和依赖：

- Jenkins Plugins
  - [Build Pipeline Plugin](#)
  - [Copy Artifact Plugin](#)
  - [Parameterized Trigger Plugin](#)
  - [Git Parameter Plugin](#)
  - [Mask Password Plugin](#)
- Docker 1.7+
- Docker Compose

安装好后，我们将进行构建流程的前三个主要任务：编译，打包，集成测试。这也是持续集成，持续部署的首要工作。

### 任务 1：构建 Go-Auth 服务

首先，确保代码是代码控制库上最新的。



为示例工程进行配置过程：选择 “New Item” （新建）-> “Freestyle project” （构建一个自由风格的软件项目），选中 “This build is parameterized” （参数化构建过程）来添加 “Git Parameter”，如下图所示：

名字设为 “GO\_AUTH\_VERSION”，“tag filter” 标签过滤设置为 “v\*”（如 v2.0），“Default

☒ This build is parameterized

**Git Parameter**

Name:

Description:

Parameter Type:

Branch:

Tag filter:

Tag sort mode:

Default Value:

[Delete](#)

value” 可设置为开发分支（develop）。

本页面其他设置见以下描述：

源码管理部分：在 “Source Code Management” 中添加 <https://github.com/usmanismail/go-messenger.git> 作为仓库链接，配置 “Branches to build” 为 “\*/develop”，设置触发器 poll 间隔，如 5

**Build**

**Execute shell**

Command: 

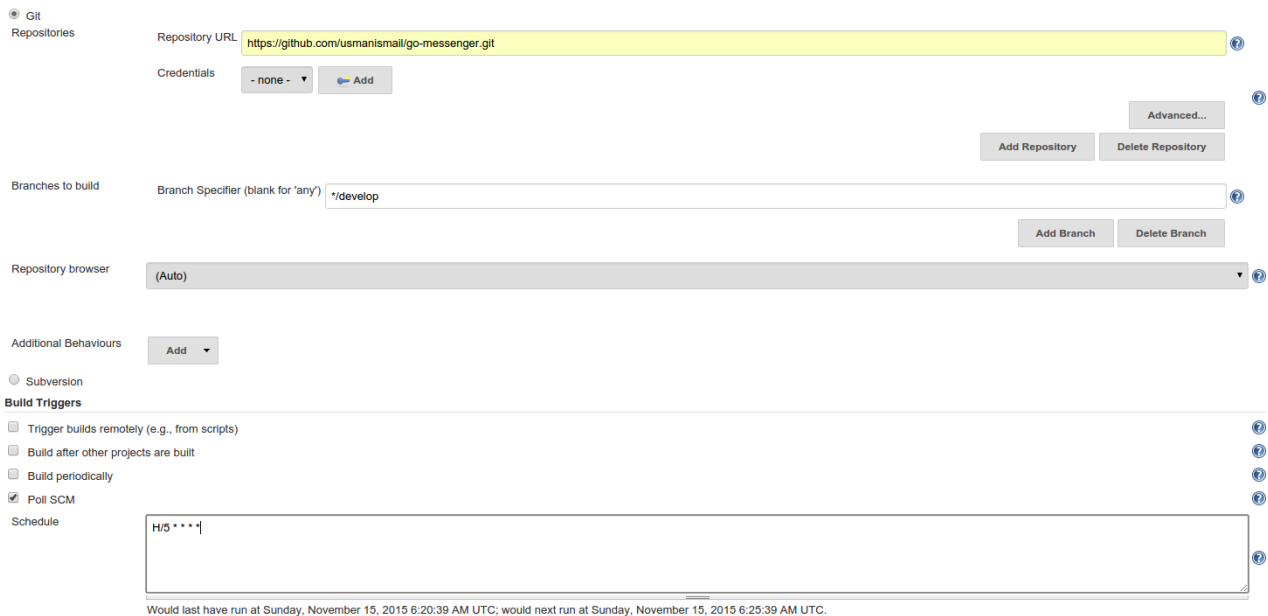
```
cd go-auth
docker run --rm \
-v $PWD:/go/src/github.com/usmanismail/go-messenger/go-auth/ \
-e SOURCE_PATH=github.com/usmanismail/go-messenger/go-auth/ \
usman/go-builder:1.4
```

[See the list of available environment variables](#)

[Delete](#)

分钟。Jenkins 将跟踪开发分支的任务改变，并触发持续集成 CI(和持续部署 CD) 流程的第一个任务。

构建部分：在“Build（构建）”配置中，选择“Add Build Step”>“Execute Shell”，并拷贝前面所提到的 docker run 命令。此步骤将从 GitHub 上获取最新代码，并将代码编译成 Go-Auth 可执行文件。



Git  
Repositories

Repository URL

Credentials

Advanced...

Add Repository Delete Repository

Branches to build

Branch Specifier (blank for 'any')

Add Branch Delete Branch

Repository browser (Auto)

Additional Behaviours

Subversion

Build Triggers

☐ Trigger builds remotely (e.g., from scripts)

☐ Build after other projects are built

☐ Build periodically

☒ Poll SCM

Schedule

Would last have run at Sunday, November 15, 2015 6:20:39 AM UTC; would next run at Sunday, November 15, 2015 6:25:39 AM UTC.

构建后操作：还需要添加两个“Post-Build”步骤，选择“Archive the Artifacts”归档 Go-Auth 二进制， “Trigger parameterized builds” 启动此流程的下一步工作，如下所示。当添加 “Trigger parameterized builds” 时，确保从 “Add Parameters” 中选中了 “Current build parameters”，此设置将使本阶段的所有设置（如 GO\_AUTH\_VERSION）同样应用于下一阶段。

Post-build Actions

Archive the artifacts

Files to archive

Advanced... Delete

Trigger parameterized build on other projects

Build Triggers

Projects to build

Trigger when build is

Trigger build without parameters ☐

Current build parameters

Add Parameters Add trigger...

构建任务的日志输出如下所示。我们用了个 Docker 化的容器在运行本次构建：先用 Go Fmt 格式标准化代码，后运行单元测试，如有编译错误或者测试错误，Jenkins 将检测到此错误。此外，你还可以配置 Email 或集成即时通讯工具（如 HipChat, Slack）来通知团队人员，以便及时修复问题。

Started by an SCM change

Building in workspace /var/jenkins/jobs/build-go-auth/workspace

```
> git rev-parse --is-inside-work-tree # timeout=10
```

Fetching changes from the remote Git repository

```
> git config remote.origin.url https://github.com/usmanismail/go-messenger.git # timeout=10
```

Fetching upstream changes from <https://github.com/usmanismail/go-messenger.git>

```
> git --version # timeout=10
```

```
> git -c core.askpass=true fetch --tags --progress https://github.com/usmanismail/go-messenger.git +refs/heads/*:refs/remotes/origin/*
```

```
> git rev-parse refs/remotes/origin/develop^{commit} # timeout=10
```

```
> git rev-parse refs/remotes/origin/origin/develop^{commit} # timeout=10
```

Checking out Revision 89919f0b6cd089342b1c5b7429bca9bcda994131 (refs/remotes/origin/develop)

```
> git config core.sparsecheckout # timeout=10
```

```
> git checkout -f 89919f0b6cd089342b1c5b7429bca9bcda994131
```

```
> git rev-list 7ae8ca4e8bed00cf57a2c1b63966e208773361b4 # timeout=10
```

[workspace] \$ /bin/sh -xe /tmp/hudson1112600899558419690.sh

+ echo develop

develop

+ cd go-auth

+ docker run --rm -v /var/jenkins/jobs/build-go-auth/workspace/go-auth:/go/src/github.com/usmanismail/go-messenger/go-auth/ -e SOURCE\_PATH=github.com/usmanismail/go-messenger/go-auth/ usman/go-builder:1.4

Downloading dependencies

Fix formatting

Running Tests

? github.com/usmanismail/go-messenger/go-auth [no test files]

? github.com/usmanismail/go-messenger/go-auth/app [no test files]

? github.com/usmanismail/go-messenger/go-auth/database [no test files]

? github.com/usmanismail/go-messenger/go-auth/logger [no test files]

ok github.com/usmanismail/go-messenger/go-auth/user 0.328s

Building source

Build Successful

Archiving artifacts

Warning: you have no plugins providing access control for builds, so falling back to legacy behavior of permitting any downstream builds to be triggered

Triggering a new build of package-go-auth

Finished: SUCCESS

## 任务 2：打包 Go Auth

当编译好代码后，需要将其打包到 Docker 容器中。选择 “New Item > Freestyle Project ”，此任务命名与上一任务匹配。如上所述，本任务也将选中 “The build is parameterized” ，并设置参数 “GO\_AUTH\_VERSION ”，如下所示。

☒ This build is parameterized

String Parameter

Name	GO_AUTH_VERSION
Default Value	develop
Description	

[Plain text] [Preview](#)

[Delete](#)

如之前在源码部分配置 GitHub 工程一样，本任务中添加一个构建步骤来执行 shell。

```
echo ${GO_AUTH_VERSION}

cd go-auth

chmod +x go-auth

chmod +x run-go-auth.sh

chmod +x integration-test.sh

docker build -t usman/go-auth:${GO_AUTH_VERSION} .
```

因为需要上一步骤编译的二进制来构建 Docker 容器，我们添加了一个构建步骤，用于从上一步构建中拷贝数据。注意：我们用 “GO\_AUTH\_VERSION” 参数来标签(tag)此镜像。开发分支的任何改变，默认将会构建 usman/go-auth:develop，并更新已有镜像。

Build

Copy artifacts from another project

Project name

Which build

☐ Use "Last successful build" as fallback

Advanced...

Artifacts to copy

## 任务 3 运行集成测试

本节我们用上面的 Docker Compose 模版来生成多容器的测试环境，并进行集成测试。最后，我们用 shell 脚本运行 HTTP 查询，如下：更改目录到 go-auth，再运行 integration-test.sh。

```
echo ${GO_AUTH_VERSION}

cd go-auth

chmod +x integration-test.sh

./integration-test.sh
```

integration-test.sh 脚本内容[在此](#)。此工作的日志输出类似于如下打印，过程如下：启动数据库容器，并将其连接到 GoAuth 容器中；如连接正常，将会看到“Pass : ...”之类的输出；测试运行完成，将清理环境并删除数据库及 GoAuth 容器。

```
Creating goauth_Database_1...

Creating goauth_Goauth_1...

[36m04:02:52.122 app.go:34 NewApplication DEBUG [0m Connecting to database db:3306

[36m04:02:53.131 app.go:37 NewApplication DEBUG [0m Unable to connec to to database: dial tcp 10.0.0.28:3306: connection refused. Retrying...
```

```
[36m04:02:58.131 app.go:34 NewApplication DEBUG [0m Connecting to database db:3306

[36m04:02:58.132 app.go:37 NewApplication DEBUG [0m Unable to connec to to database: dial tcp 10.0.0.28:3306: connection refused. Retrying...

[36m04:03:03.132 app.go:34 NewApplication DEBUG [0m Connecting to database db:3306

[36m04:03:03.133 common.go:21 Connect DEBUG [0m Connected to DB db:3306/messenger

[36m04:03:03.159 user.go:29 Init DEBUG [0m Created User Table

[36m04:03:03.175 token.go:33 Init DEBUG [0m Created Token Table

[36m04:03:03.175 app.go:42 NewApplication DEBUG [0m Connected to database

[36m04:03:03.175 app.go:53 Run DEBUG [0m Listening on port 9000

Using Service IP 10.0.0.29

Pass: Register User

Pass: Register User Conflict

Stopping goauth_Goauth_1...

Stopping goauth_Database_1...

Finished: SUCCESS
```

以上三个任务，在 Jenkins 视图页面中，选择标签 “+” ，你将会新建一个构建流程视图，在弹出的配置



页面中，选择编译/构建任务作为启示任务，并选择确定。将会很直观的看到每个提交经过构建/部署的完整工作流程。

当你在开发分支上有所更改时，Jenkins 将会被自动触发。

回顾以上步骤：

1. 用 Git-Flow 添加新功能，并合并到开发分支中；
2. 跟踪开发分支的变化，并在集中管理环境中构建系统；
3. 将上一步骤所生成的应用打包成一个 Docker 容器；
4. 用 Docker Compose 部署临时测试环境；
5. 进行集成测试，并清理环境；

以上 CI 流程，每当一个新功能（或是 Bug 修复）合入开发分支时，以上过程便会自动执行，并生成 “usman/go-auth:develop” 的 Docker 镜像。下一步我们会有一系列文章描述集成部署。



## 总结

本文中我们讲述了如何将 Docker 应用到 CI 中，以达到集中管理、可测试、可重复性、隔离性（不同组件的环境依赖）等要求。基于 Docker 进行构建和部署流程的后续部分，下周我们将展示如何用 rancher 来部署一个完整的服务环境，对大规模项目而言，我们也将提供如何配置长期测试环境的最佳实践。

如需了解 Rancher，请注册 [Rancher beta 版](#)，同时，你也能[下载本文的电子书](#)。

### 温馨提示：

专业玩 Docker，大道至简，趋势跟我走！

加微信群方法：

1.关注【有容云】公众号

2.留言“加群”

QQ 群号：454565480



