Packt>

微服务运维实战（第二卷）

The DevOps 2.1 ToolKit: **Docker Swarm**

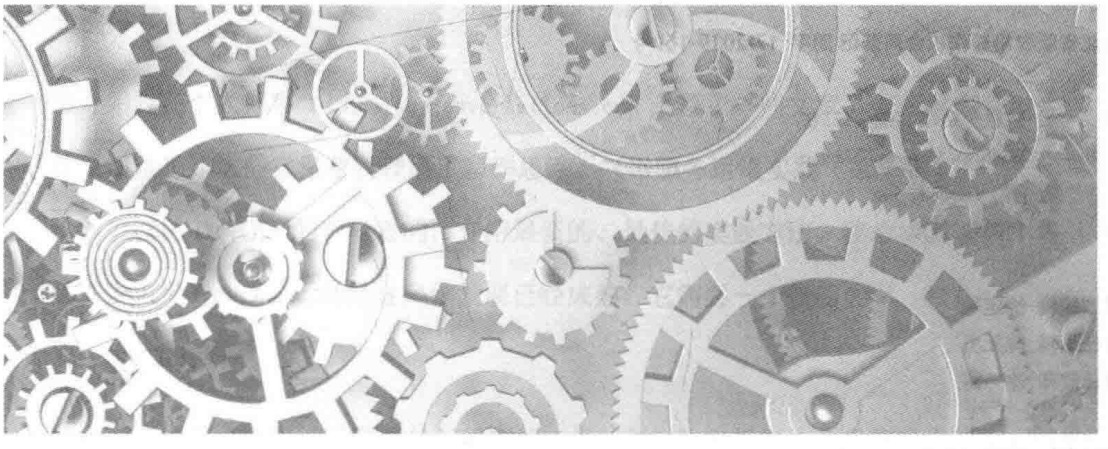
在Docker Swarm上构建、测试、部署、监控微服务

[西] Viktor Farcic 著
何腾欢 汪欣 译

華中科技大學出版社
http://www.hustp.com

微服务运维实战（第二卷）

The DevOps 2.1 ToolKit: **Docker Swarm**



[西] Viktor Farcic 著
何腾欢 汪欣 译

华中科技大学出版社
中国 · 武汉

图书在版编目(CIP)数据

微服务运维实战. 第二卷 / (西) 维克托·法西克(Viktor Farcic)著；何腾欢, 汪欣译 . -- 武汉 :华中科技大学出版社, 2019.8

ISBN 978-7-5680-5353-2

Ⅰ. ①微… Ⅱ. ①维… ②何… ③汪… Ⅲ. ①互联网络－网络服务器－程序设计 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字(2019)第154721号

湖北省版权局著作权合同登记 图字: 17-2019-169号

书　　名 微服务运维实战(第二卷)
Weifuwu Yun-wei Shizhan
作　　者 [西] Viktor Farcic
译　　者 何腾欢 汪欣

策划编辑 徐定翔
责任编辑 陈元玉
责任监印 徐　露

出版发行 华中科技大学出版社（中国·武汉）
武汉市东湖新技术开发区华工科技园（邮编 430223 电话027-81321913）
录　　排 武汉东橙品牌策划设计有限公司
印　　刷 武汉华工鑫宏印务有限公司
开　　本 787mm × 960mm 1/16
印　　张 24.25
字　　数 584千字
版　　次 2019年8月第1版第1次印刷
定　　价 110.80元

本书若有印装质量问题, 请向出版社营销中心调换
全国免费服务热线400-6679-118竭诚为您服务

前言

Preface

2016 年初，我出版了《微服务运维实战（第一卷）》。写这本书花的时间比预想的要长得多。

最开始我在个人博客 TechnologyConversations.com 上写了一些文章，收到很多读者的反馈意见。这些意见帮我理清了写作思路。《微服务运维实战（第一卷）》是写给希望运用 DevOps 的读者看的，介绍最新的、最佳的实践方法。我希望通过第一卷告诉大家 DevOps 运动经过多年发展已经成熟，它需要一个新的名字，我称之为 DevOps 2.0。

技术变革的速度太快，《微服务运维实战（第一卷）》出版仅 6 个月就发生了很多变化：新版本的 Swarm 发布了，它现在是 Docker Engine v1.12 的一部分，其中还集成了服务发现（service discovery）。在我看来，1.12 版是 Docker Engine 发布以来最重要的版本。

我还记得 2016 年与 Docker 的工程师们在西雅图共度的时光。我花了 4 天时间和他们一起研究即将在 1.12 版中发布的新特性，以及 1.12 版之后的路线图。当时我以为已经理解了他们提出的所有概念和特性。然而，当我一周后开始“把玩”新的 Docker Swarm 模式时，我才意识到自己的大脑仍然束缚于以往的工作方式。新版本的改动太多，随之也出现很多新的可能性。我花了几周的时间才把大脑清

零，直到那时，我才真正明白 Docker 的工程师们在这个版本中做了多大的改动。

在探索 Swarm 模式的同时，我也陆续收到来自《微服务运维实战（第一卷）》读者的电子邮件。他们提了很多要求，希望我在深入探讨已有话题的同时也能涵盖新的主题，其中反复提到的一个要求是“希望你更深入地介绍集群”。读者希望更详细地了解如何操作集群以及如何将其与持续部署结合。他们还希望我研究零停机时间部署的其他方法，希望更有效地监控系统，更接近自愈系统，等等。

因此，结合 Docker 1.12 的新特性与读者的请求，我决定写一本新书《微服务运维实战（第二卷）》，同时“微服务运维实战”系列也应运而生了。

概述
Overview

本书探讨了运行 Swarm 集群所需的方法和工具。我们要做的不是简单的部署，而是要研究如何创建持续部署。我们将搭建多个集群，一个集群用于测试，其他的用于生产。我们将看到如何实现零停机时间部署，如何处理故障转移情况，如何大规模地运行服务，如何监控系统以及如何让系统自愈。我们要探索的流程不仅允许在不同的供应商设备上运行集群，也能在笔记本电脑上运行集群（用于演示目的）。我将尽可能地涵盖各种托管解决方案。书中的流程、工具、实践可以很容易地应用到任何解决方案中去。本书依然注重实用性，但目标不是掌握一套特定的工具，而是学习这些工具背后的逻辑。这样，即使你选择不一样的工具，也能顺利开展工作。

当然，《微服务运维实战（第一卷）》并未过时，我认为书中的逻辑在很长时间内依然有效。第二卷以第一卷为基础，更深入地探讨了一些主题。我强烈建议读者从《微服务运维实战（第一卷）》开始阅读。

本书前面几章的部分内容曾在《微服务运维实战（第一卷）》中出现过，但这些内容是必不可少的，因为它们解决了我们在集群内部工作时会遇到的一些问题，也为后面的章节奠定了基础。

这是一本注重实践的书，只在通勤车上读是不够的，你必须上机实践。如果你需要帮助，或者想发表评论，可以访问本书在 Disqus 上的频道，或者第一卷的 Slack 频道。我很重视自己写的书，也希望能带给你良好的阅读体验。与我的沟通也是这体验的一部分，别害羞。

这两卷书都是我自行出版的。我认为作者和读者之间最好没有中介，这样我可以写得更快，更新得更快。理论上，这本书永远不可能真正写完。随着时间的推移，书的内容也需要更新。当技术的变化太多时，就需要写一本新书了。只要有你们的支持，我就会继续写下去。

目录

Contents

第 1 章

利用 Docker 容器持续集成

Continuous Integration with Docker Containers

我们知道得越多，从绝对意义上来说我们就变得越无知，因为只有通过启蒙，我们才能意识到自己的局限性。智力进化最令人欣慰的结果之一，就是不断开拓新的、更广阔的前景。

——尼古拉·特斯拉

为了充分理解 Docker Swarm 带来的挑战和好处，我们需要从头开始讲。我们得重回代码库并决定如何构建、测试、运行、更新和监视正在开发的服务。尽管目标是在 Swarm 集群上实现持续部署，但我们得先退一步去研究持续集成（Continuous Integration，CI）。为 CI 流程定义的步骤将决定我们如何继续实施持续交付（Continuous Delivery，CD），继而实施持续部署（Continuous Deployment，CDP），并最终决定如何确保服务被监控并能够自愈。本章探讨的持续集成是更高级过程的先决条件。

给《微服务运维实战（第一卷）》读者的提示

这一页的内容与已经出版的《微服务运维实战（第一卷）》的内容大致相同。如果你对这部分内容印象依旧深刻，请直接跳到第 1.1 节。写第二卷时，我发现了一些更好的方法来实现 CI 过程。希望你能从本章的内容中获益。

要了解持续部署，首先应该定义它的前身，即持续集成和持续交付。

项目开发的集成阶段往往是软件开发生命周期（Software Development Life Cycle，SDLC）中最痛苦的阶段之一。各个团队要花费数周、数月甚至数年独立开发不同的应用程序和服务。每个团队都有自己的任务。尽管定期单独验证这些应用程序和服务并不困难，但是当领导决定把它们集成到一起时，我们还是会忧心忡忡。凭借以往的经验，我们知道集成会出问题（未满足的依赖关系、不能正确通信的接口等）。这个阶段往往要花费几周甚至几个月的时间。更糟糕的是，在集成阶段发现的错误意味着要回过头来重做几天或几周的工作。如果有人问我对当时集成的感觉如何，我会说糟透了。

后来，极限编程（eXtreme Programming，XP）的各种实践应运而生，自动化测试变得频繁，持续集成开始起步。今天我们知道以前的开发方式是错误的。软件行业已经发生了很大的变化。

持续集成是指在开发环境中集成、构建和测试代码。它要求开发人员频繁将代码集成到共享代码库中。集成的频率取决于团队的规模、项目的规模以及开发时间。大多数情况下，程序员要么直接推送到共享代码库，要么将代码合并。无论是推送还是合并，这些操作一天至少应该进行几次。让代码进入共享代码库中还不够，我们还需要一条流水线，它能检出代码并直接或间接地运行测试。如果流水线执行失败了，应该通知提交代码的人。

持续集成流水线应该在每次提交或推送时运行。与持续交付不同，持续集成没有明确定义该流水线的目标。我们不知道还需要做多少工作才能把代码交付到生产环境，只能尽量做到让提交的代码通过现有测试。尽管很难实施，但 CI 仍是一个巨大的进步。一旦大家适应了这个流程，结果往往出奇的好。

集成测试需要提前提交，或者与代码一起提交。为了获得最好效果，应该采用测试驱动开发（TDD）方式编写测试代码[1]。

最重要的 CI 原则是，当流水线失败时，解决问题比其他任务具有更高的优先级。如果不及时解决问题，那么流水线的下一次执行也将失败。长此以往，人们会慢慢忽视失败通知，CI 也就失去了意义。越早解决问题越好。

---

[1] http//technologyconversations.com/category/test-driven-development

1.1 完全 Docker 化的手动持续集成流程
Defining a Fully Dockerized Manual Continuous Integration Flow

每个持续集成过程都以从代码库检出的代码开始。本书将使用 GitHub 代码库 vfarcic/go-demo（https://github.com/vfarcic/go-demo），它包含本书使用的服务代码。该服务是用 Go 编写的（https://golang.org/）。别担心！你不需要学习 Go。我们仅使用 go-demo 服务演示和解释流程。所有例子都是与编程语言无关的。

本章中的所有命令都可以在 01-continuous-integration.sh 中找到（https://gist.github.com/vfarcic/886ae97fe7a98864239e9c61929a3c7c）。

给 Windows 用户的说明

请确保你的 Git 客户端配置为原样（AS-IS）检出代码。否则，Windows 可能会将回车更改为 Windows 格式。

下面开始吧，看看 go-demo 代码：

```
git clone https://github.com/vfarcic/go-demo.git

cd go-demo
```

有些文件将在主机文件系统和我们即将创建的 Docker Machine 之间共享。Docker Machine 使得属于当前用户的整个目录在虚拟机内可用。因此，请确保代码在用户的某个子文件夹中检出。

现在已经从代码库中检出了代码，还需要一个服务器来构建和运行测试。我们将使用 Docker Machine，因为它能在笔记本电脑上简单地创建 Docker ready 虚拟机。

Docker Machine（https://docs.docker.com/machine/overview/）可让你在虚拟主机上安装 Docker Engine，并使用 docker-machine 命令管理主机。Docker Machine 可以在本地 Mac 或 Windows 机器、公司网络、数据中心或云供应商（如 AWS 或 DigitalOcean）上创建 Docker 主机。

使用 docker-machine 命令可以启动、检查、停止并重启被管理的主机，升级 Docker 客户端和守护程序，并配置 Docker 客户端与主机通信。

在 Docker v1.12 版之前，Docker Machine 是在 Mac 和 Windows 上运行 Docker

的唯一方法。从 Docker v1.12 版开始，出现了 Docker for Mac 和 Docker for Windows，它们更适合较新的桌面和笔记本电脑。Docker for Mac 和 Docker for Windows 的安装程序包括 Docker Machine 和 Docker Compose。

本书的例子假设你已安装 1.13 以上版本 Docker Engine 的 Docker Machine v0.9（`https://www.docker.com/products/docker-machine`），可以在 Install Docker Machine（`https://docs.docker.com/machine/install-machine/`）页面找到安装命令。

给 Windows 用户的说明

建议从（通过 Docker Toolbox 和 Git 安装的）Git Bash 运行所有示例。这样，使用的命令将与 OSX 或 Linux 系统上执行的相同。

给 Linux 用户的说明

Linux 上的 Docker Machine 可能无法在虚拟机中挂载主机卷。这跟主机操作系统和 Docker Machine 操作系统都使用 `/home` 目录有关。从主机挂载 `/home` 将覆盖一些所需的文件。如果你在安装主机卷时遇到问题，请设置

`VIRTUALBOX_SHARE_FOLDER` 变量：

```
export VIRTUALBOX_SHARE_FOLDER = "$PWD:$PWD"
```

如果已经创建了 Docker 机器，则必须销毁并再次创建。

请注意，此问题在较新的 Docker Machine 版本中已解决，因此，仅当你遇到未加载卷时才使用此解决方法（来自主机的文件在 VM 中不可用）。

使用下面的命令创建第一个名为 `go-demo` 的服务器。

```
docker-machine create  -d virtualbox go-demo
```

给 windows 用户的说明

如果你使用的是 Docker for Windows 而不是 Docker Toolbox，则需要将驱动程序从 virtualbox 更改为 Hyper-V。问题在于 Hyper-V 不允许安装主机卷，因此，建议在使用 Docker Machine 时使用 Docker Toolbox。选择在 Docker Machine 内部运行 Docker 而不是在主机上本地运行的原因在于需要运行一个集群（第 2 章将介绍）。Docker Machine 是模拟多节点集群的最简单的方法。

该命令将 virtualbox 指定为驱动程序（如果你正在运行 Docker for Windows，则指定为 Hyper-V）并将 machine 命名为 `go-demo`：

给 Windows 用户的说明

在某些情况下，Git Bash 可能会认为它仍在以 BAT 批处理形式运行。如果你遇到 `docker-machine env` 命令的问题，请设置 `SHELL` 环境变量：

```
export SHELL=bash
```

现在机器正在运行，我们应该让本地 Docker Engine 使用它，请使用以下命令：

```
docker-machine env go-demo
```

`docker-machine env go-demo` 命令输出本地引擎所需的环境变量以查找我们想要使用的服务器。这种情况下，远程引擎位于我们使用 `docker-machine create` 命令创建的 VM 内部。

输出如下：

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/vfarcic/.docker/machine/machines/go-demo"
export DOCKER_MACHINE_NAME="go-demo"
```

我们可以将 `env` 命令封装到一个执行输出的 `eval` 中，在本例中，使用以下命令创建环境变量：

```
eval $(docker-machine env go-demo)
```

从现在开始，我们在本地执行的所有 Docker 命令将会被引导到 go-demo 机器运行的引擎。

现在我们准备好运行 CI 流程的前两个步骤。下面将运行单元测试并构建服务的二进制文件。

1.2 运行单元测试并构建服务的二进制文件
Running Unit Tests and Building Service Binary

我们将在 CI 流程中使用 Docker Compose。正如你将看到的，Docker Compose 在运行集群时几乎没什么用处。但是，对于要在单台机器上执行的操作，Docker Compose 仍然是最简单和最可靠的方法。

Compose 是定义和运行多容器 Docker 应用的工具。你可以使用 Compose 文件来配置应用的服务。然后，使用单个命令创建并启动配置中的所有服务。Compose 非常适合开发、测试和预备环境以及 CI 工作流程。

之前克隆的代码库已经包含 docker-compose-test-local.yml 中定义的所有服务（https://github.com/vfarcic/go-demo/blob/m aster/docker-compose-test-loc）。

让我们看看文件的内容（https://githu b.com/vfarcic/go-demo/blob/master/docker-compose-test-local.yml）。

```
cat docker-compose-test-local.yml
```

我们把用于单元测试的服务称为 unit，如下所示：

```
unit:
    image: golang:1.6
    volumes:
      - .:/usr/src/myapp
      - /tmp/go:/go
    working_dir: /usr/src/myapp
    command: bash -c "go get -d -v -t && go test --cover -v \
./... && go build -v-o go-demo"
```

这是一个相对简单的定义，由于服务是使用 Go 编写的，因此我们使用的是 golang:1.6 镜像。

接下来，我们公开了几个卷。在这种情况下，卷是挂载在主机上的目录，它们由两个参数定义。第一个参数是主机目录的路径，第二个参数表示容器内的目录。主机目录内的任何文件都将在容器中可用，反之亦然。

第一个卷用于源文件。我们正把当前的主机目录共享到容器的/usr/src/myapp 目录。第二卷用于 Go 依赖库，由于我们希望避免每次运行单元测试时都下载所有依赖项，因此它们将存储在主机目录/tmp/go 中。这样，只有在第一次运行服务时才会下载依赖项。

卷之后是 working_dir 指令。当容器运行时，将使用指定的值作为起始目录。

最后，指定想要在容器中运行的命令。笔者不会详细介绍它们，因为它们是专门针对 Go 的。简言之，我们下载所有依赖项 go get -d -v -t，运行单元测试 go test --cover -v ./ ...，然后构建 go-demo 二进制文件 go build -v -o go-demo。由于加载了有源代码的目录作为卷，因此二进制文件将存储在主机上，供以后使用。