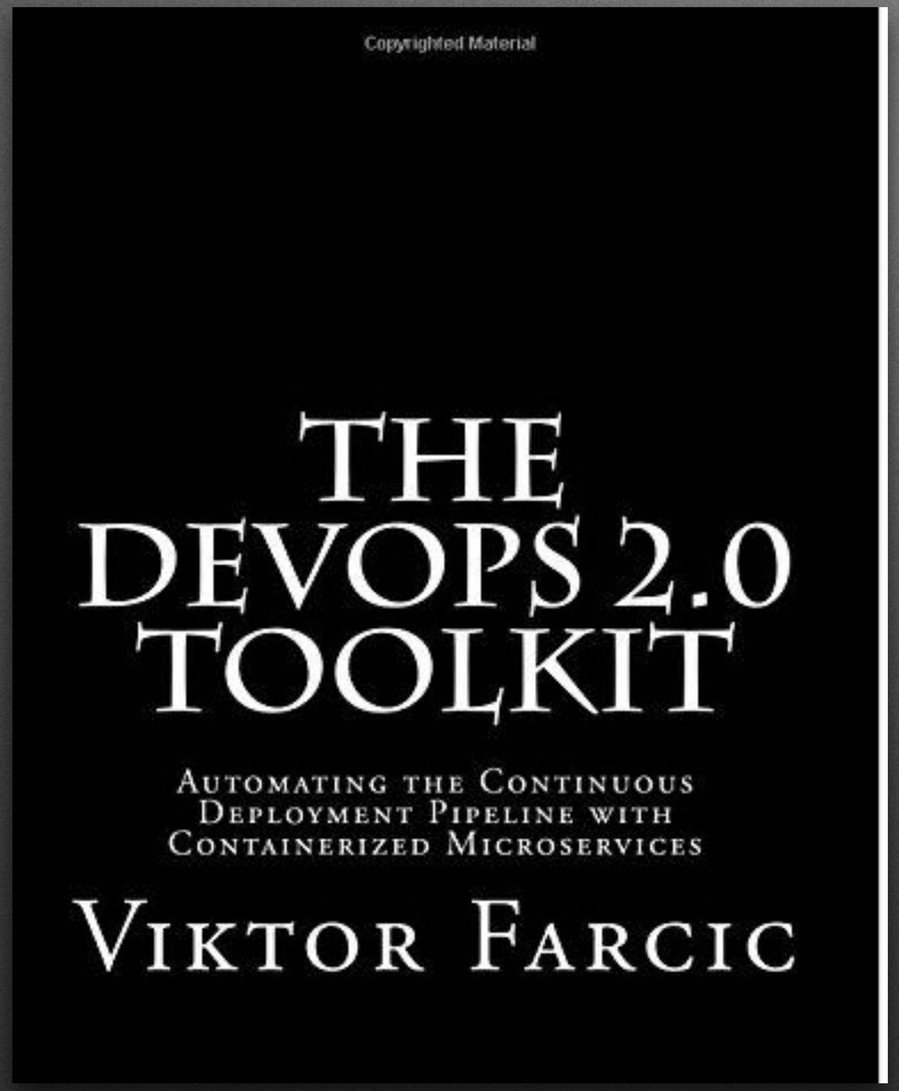
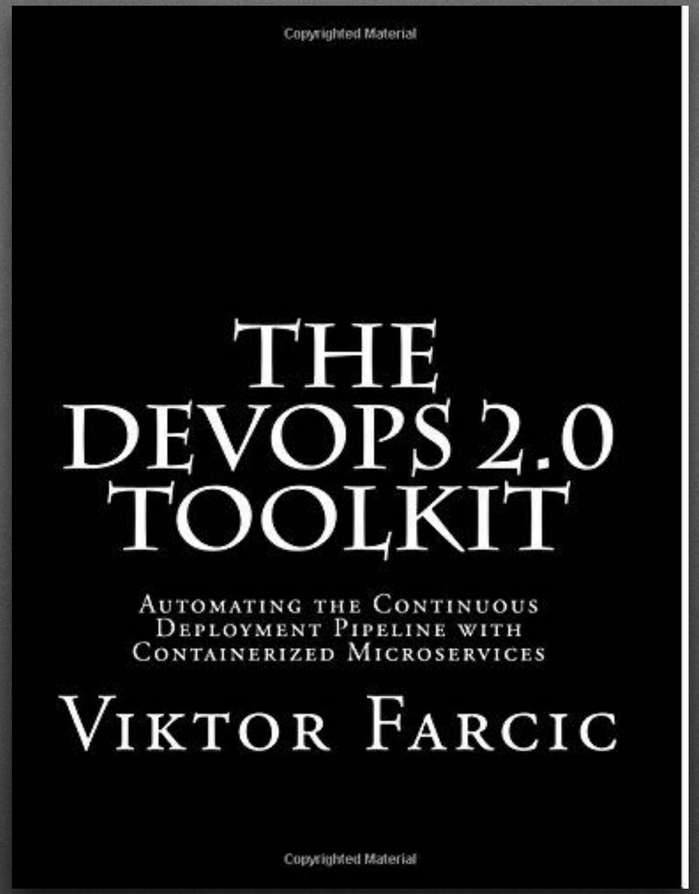


读书笔记

Martin Liu



第一章 DevOps概念

微服务架构的潜在价值和优势

- 现有的两种项目类型：
  - 1）新生的从无到有开发的项目；可以直接受益；上手风险小。
  - 2）需要和已有系统对接的，属于老旧系统延伸部分的，用瀑布式开发模式和运维的项目。则需要做系统重构；难度和风险高。

项目特点

- 新项目优势：
  - 可以使用最新的技术，为系统的不同子服务来选择最佳开发语言和技术栈。
  - 一个错误的设计或者模块可以推倒重来的代价按周计算。
- 传统项目痛点：
  - 必须考虑和已有系统的对接，无法尝试新生技术和事物，技术创新被完全压制
  - 系统设计和发布的失误导致的代价是巨大的。

传统软件系统建设痛点

- 传统软件开发方式下，系统集成测试被放在最后阶段。把不同团队经过数月开发的成果集成在一起的时候，才是开发人员噩梦的开始。各种返工、需求重定义像灾难一样不可避免地频繁发生。

- 从这些血淋漓的经验教训中，作者提出了CI/CD的必要性和价值。但是CI/CD往往被认为是Ops团队的任务，而不关Dev人员什么事。

- 对于Dev人员来说，如果还没有入手和使用CI工具和自动化测试框架，这些欠下的技术债早晚还是要还的。你有没有半夜被Ops从床上捉起来，去生产系统里排bug的经历？有没有留下心理阴影？

CI/CD实践的必要性

- CI/CD的操作流程和实践其实是要把所有可能的坑都提前踩一遍，建立以天、小时、甚至分钟的快速反馈机制。
- 它需要影响和关联到系统开发和交付的所有环节中，从架构设计到测试，研发和运维人员每天都需要关注的是管理和业务需求是否满足。
- 整体思维、系统性思维是很重要的

关于系统架构

- 一个形象的比喻：很大的团队多年开发出的单体应用，不加充分测试，紧耦合，技术选择陈旧而保守。去给这样的系统做瘦身和优化，就像是企图把一个八十岁的老太太整形为小姑凉一样困难。所有类似优化改造和升级都是无力回天的。

- 微服务架构的定义：把系统构造成很多小的独立服务，它们能被多个小团队维护，任何程序猿对每个服务的代码都能秒懂。微服务可以被任意改变开发框架、语言、数据库；而不会影响到其它系统。可以被独立的部署。我想很多人都在探求微服务应该切分多大和多小；我想微服务的服务并不关乎大小，而在乎其功能切分的合理性，微服务的总数与维护团队人数的匹配。

- 程序猿所使用的共享库是紧耦合的陷阱，会拖慢系统实施的速度。标准化服务接口也是创新的隐形杀手。这些单体应用经常会出的错，微服务也可能会有，而且其后果可能是所有错误相乘的影响度。

- 微服务的有其自身的特点、切分、构建原则和潜在问题。只有清晰认识和定位它，才能保障在实践过程中，能够收获它的优势和好处。

软件系统部署

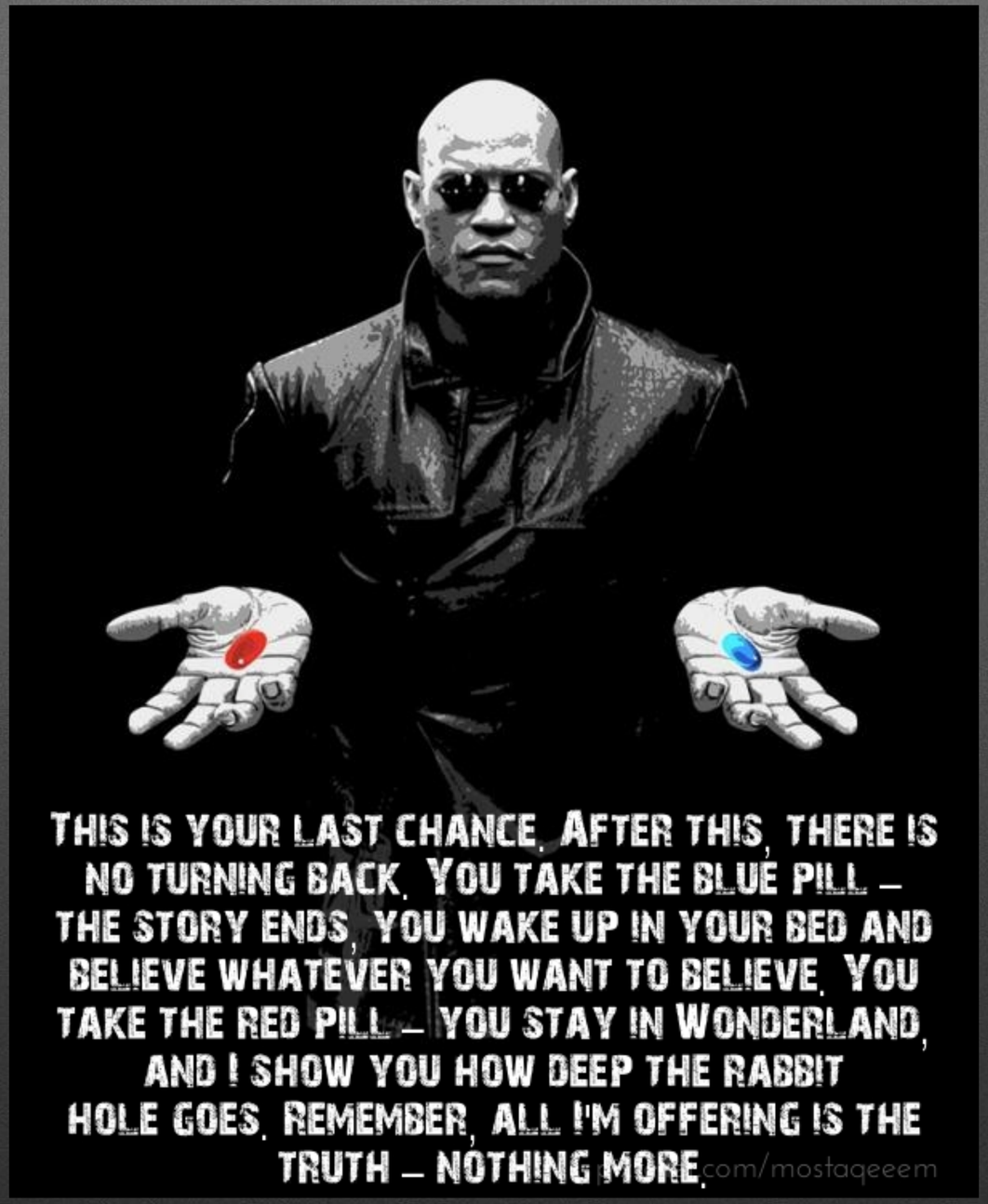
- 服务部署的对象是一堆Jar，WAR，DLL和其它任何程序代码的输出制品。定位那些要部署在旧服务器上，那些要在新服务器上是困难重重。所有服务器有差异性，生产环境和准生产环境和开发环境有天然的差异性。
- 用虚拟机可以保证一定的一致性，可是对于每一个运行了一段时间的虚拟机，谁也无法回溯它完整的手工系统配置和变更过程。
- 系统理想配置状态和现实状态的差异性是由时间积累导致的。除非用自动化工具和流程来代替手工配置操作，才能保证服务部署和更新的成功率。
- 手工很难把每个子服务都部署在对应的VM上，那样做会导致虚拟机的泛滥，VM的管理同样是问题。
- Immutable server is good idea.：不可变服务/服务器是好方法。而目前已用的工具无法支持它的部署。

服务编排

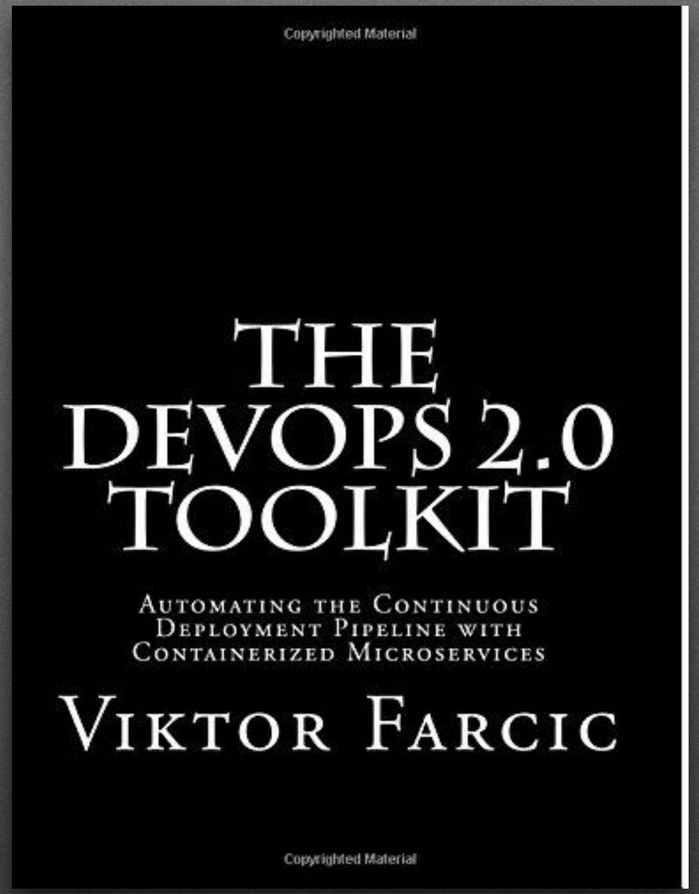
- 传统的服务配置/编排工具，如：Puppet和Chef，如果是手工开发、维护和运作这些脚本，而没有系统性思考的话，基本上也是一个悲剧的噩梦结果。这些编排工具也定性为：比什么工具都不用要好的东西。

DevOps工具分类

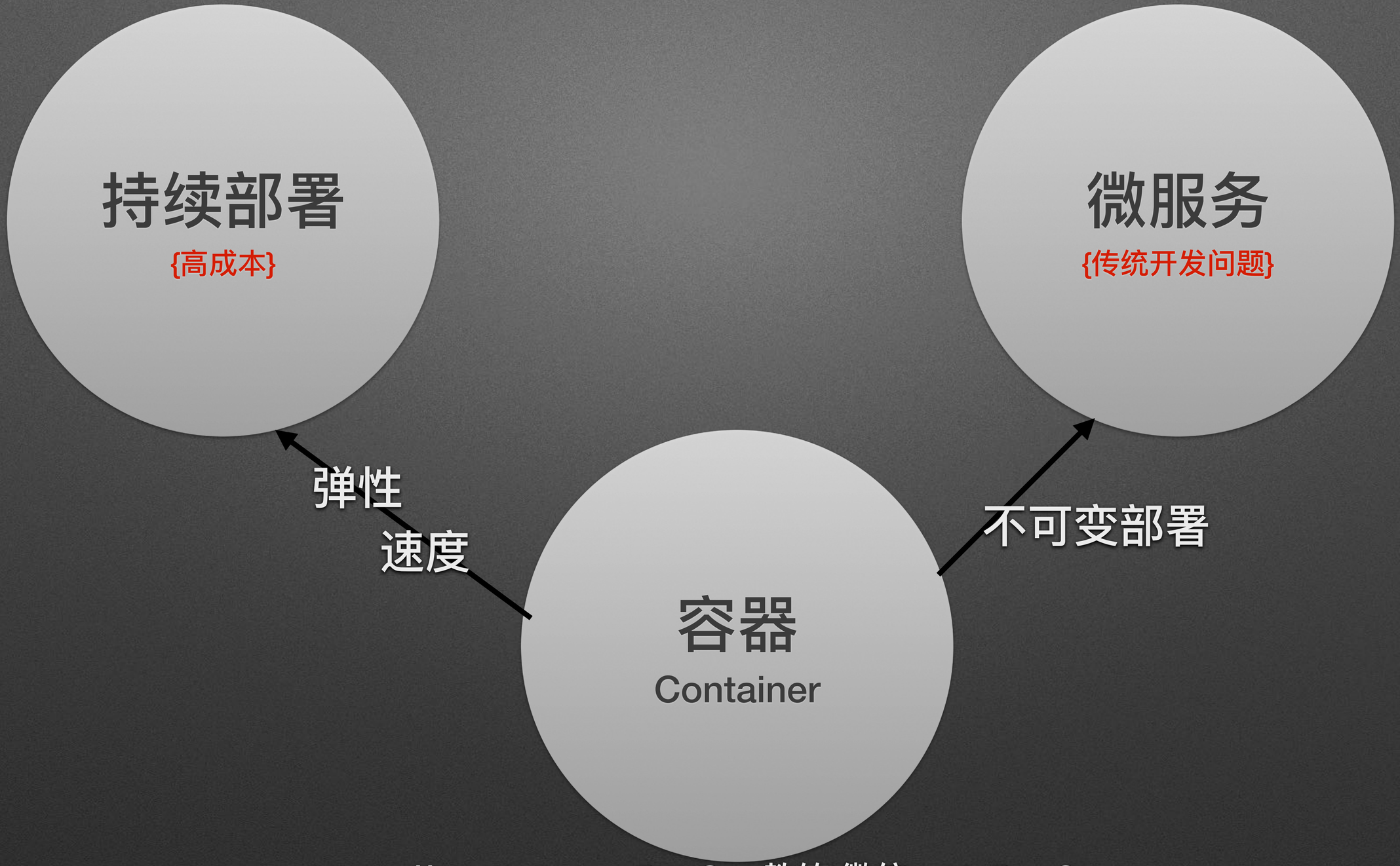
- 配置管理工具Ansible：替代Puppe和Chef之类的工具。
- Docker容器工具：替代缓慢的虚拟机，它是推荐的创建不可变（immutable）部署的工具。
- 服务发现工具：使用Swarm，Kubernets和Mesos/DCOS都可以。
- 微服务正在以缓慢的速度被应用于构架大型系统，还需要涉及的工具包括：CoreOS、etcd，Consul，Fleet，Mesos，Rocket等等



Https://martinliu.cn DevOps教练 微信：MyDevOps



第二章 实施上的突破：持续部署、微服务和容器


持续部署
{高成本}
微服务
{传统开发问题}
弹性
速度
不可变部署
容器
Container

持续集成

传统项目开发— 集成阶段

- 软件开发周期中最痛苦的阶段，重大的问题在最后的这个阶段才开始暴露
- 实际上非常耗时、缺乏统筹、各个团队独自开发
- bug导致的模块返工，数天、数周的工作量的浪费很平常
- 团队背靠背独自开发，各自选择自己有利的，有经验的工具开发
- 每个人对系统集成的感受都是绝望、沮丧、痛苦；但是大家依然认为“这是对的”

单次集成 —> 持续集成 CI

- 极限编程XP、敏捷开发Agile 、自动化测试的出现带来了持续集成的兴起
- 持续集成：开发环境、共享代码仓库、程序员频繁地集成他们的代码
- 集成频度：和团队大小、项目规模、开发时间有关
- 最小化CI流水线：检出代码、执行测试案例、错误通知
- 触发条件：每次 commit / push
- 目标：至少已有测试案例结果没有报错
- 最大化CI价值：应用TDD的方式编写测试案例
- 原则：修复流水线的报错是高优先级任务，应尽早及时修复报错，能最快地修复，完成一次反馈，避免潜在问题

持续集成流程



CI-01：提交代码

1. 程序员提交代码到mainline，feature branches
2. CI工具监控提交动作，并执行CI流水线
3. 流水线运行一组任务校验代码的正常工作
4. 如果流水线没有报错，提升为RC，继续做手工测试
5. 如果流水线报错，通知提交的程序员

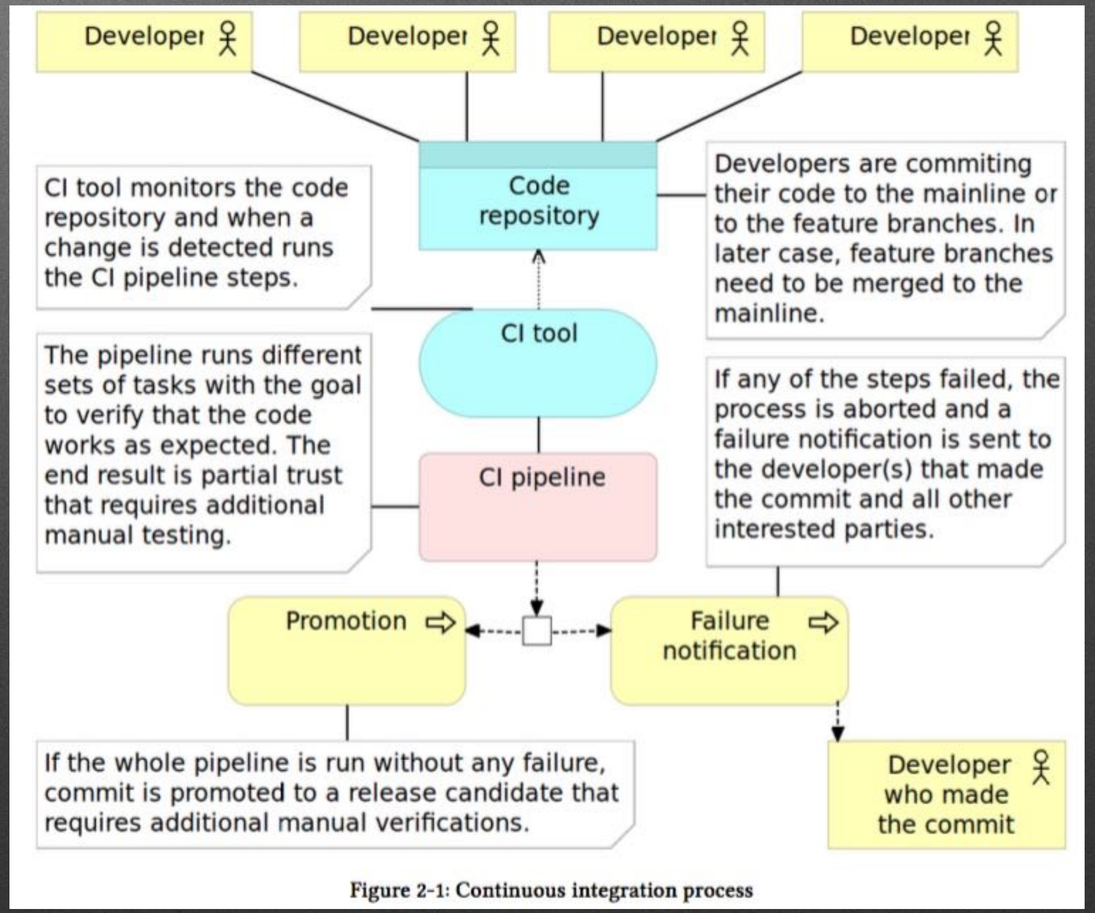


Figure 2-1: Continuous integration process

CI-02：静态检查

- CI工具检查到代码的变化，检出代码
- 静态代码检查工具检查代码
- 如果出错，执行出错通知
- 通知提交代码的程序员
- 代码检查的工具因程序语言不通而各异，常用的包括：CheckStyle、FindBugs、JSLint、JSHint、PMD

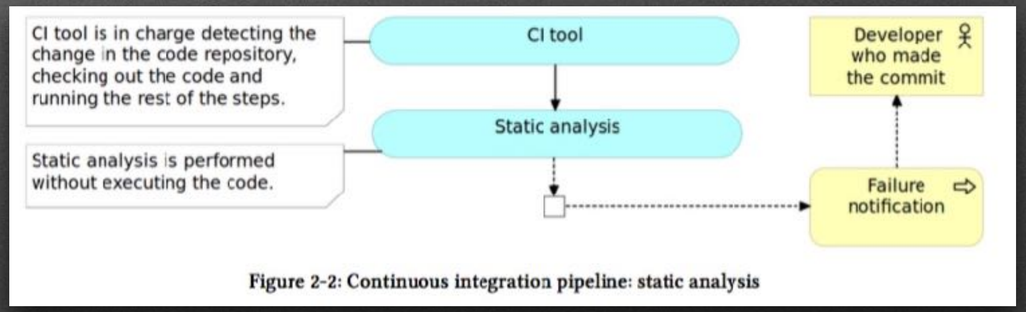


Figure 2-2: Continuous integration pipeline: static analysis

CI-03：部署前测试

- 执行测试->错误通知->通知代码提交程序员
- 是CI步骤中最关键的步骤
- 不需要代码的部署
- 测试类型：单元测试、功能测试
- 测试案例编写原则：逻辑简洁、快速执行、代码覆盖率高
- 不适合测试类型：集成、性能

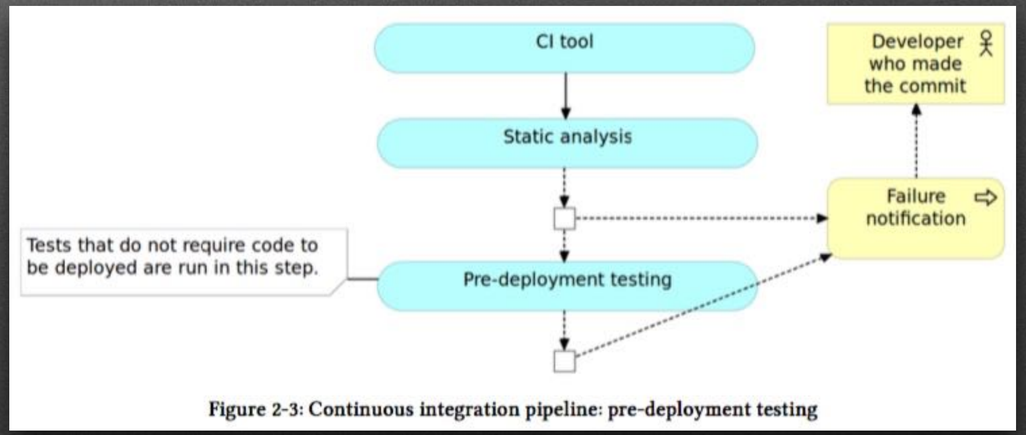


Figure 2-3: Continuous integration pipeline: pre-deployment testing

CI-04：打包部并署到QA环境

- 编译代码->生产构建物->构建容器镜像->部署到一个或者多个QA环境
- 多个QA环境：性能测试和其它所有测试需求的环境
- 可选动作：上传构建物到artifact仓库、构建Docker镜像，上传Docker镜像到镜像仓库
- 部署：可以是非容器的部署，但本书是的建议方法是：此步骤的两个动作都由docker技术实现

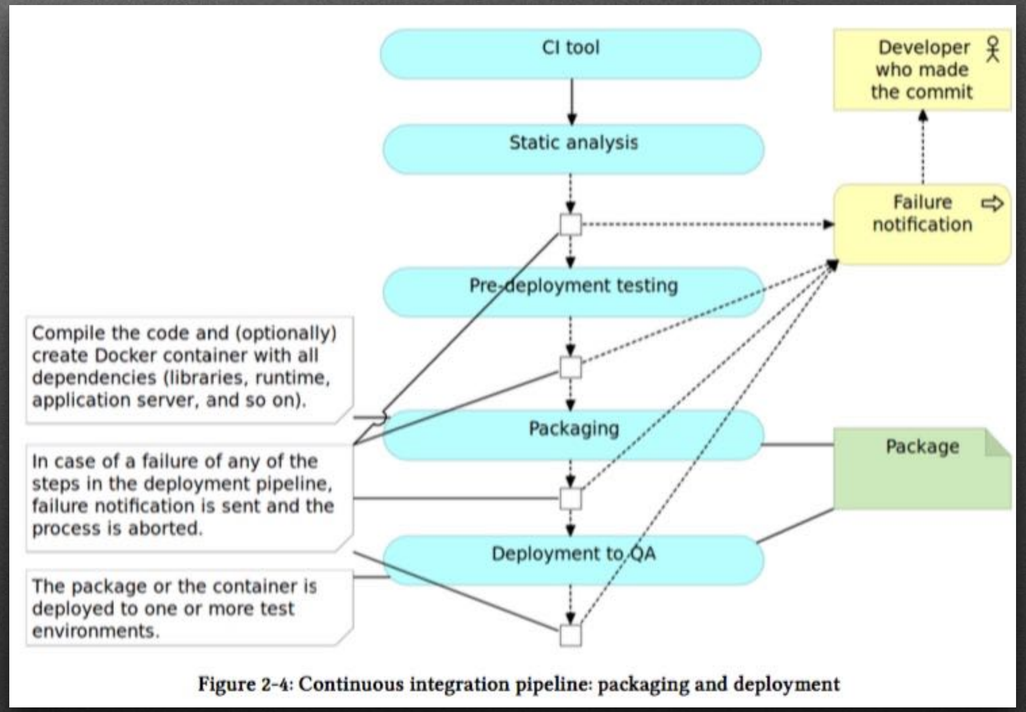


Figure 2-4: Continuous integration pipeline: packaging and deployment

CI-05：部署后测试

- 测试必须是应用代码/服务部署成功后，正常运行状态才能执行
- 测试类型包括：功能测试、集成测试、性能测试
- 功能测试的方法建议：behavior-driven development
- 性能测试工具：Gatling
- 其它细节：跑手工测试案例、选择上生产的版本、本步骤通过的为集成成功的build

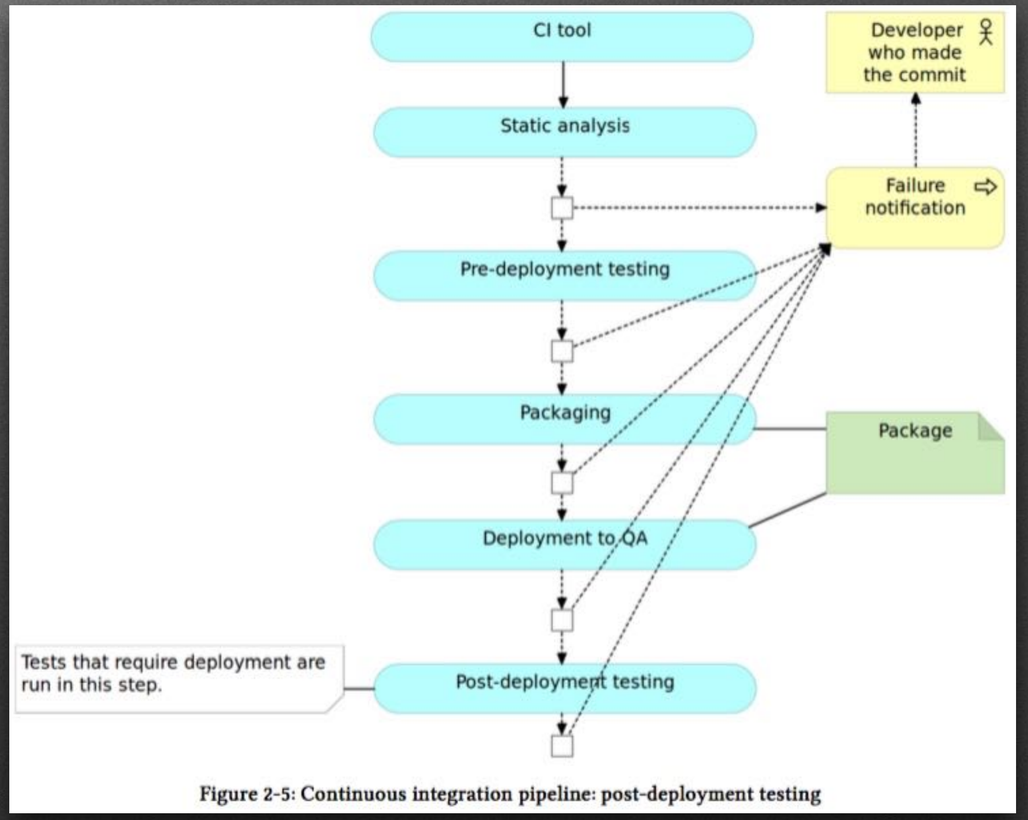


Figure 2-5: Continuous integration pipeline: post-deployment testing

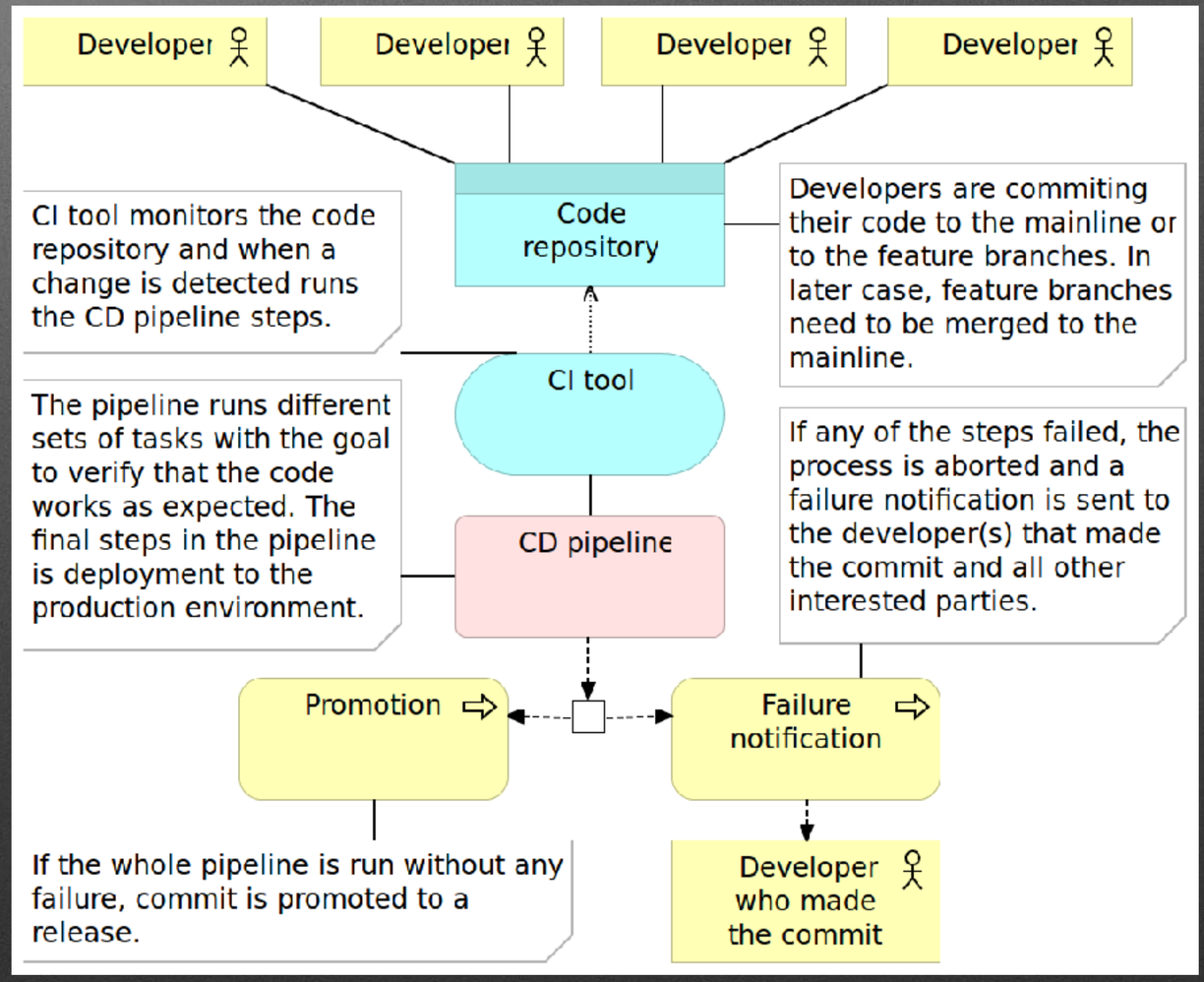
持续交付和部署

CI和CD的异同

- CD的流水线和CI中部署部分几乎的操作相同，只是CD在生产环境上发生，CD/CI的流水线工具是相同的(Jenkins)
- CD过程的需要执行的特定的动作其实很少，CI中有大量动作，CD需要实现100%自动化
- 理论上讲：每一个CI成功而手工校验测试案例亦完全成功的版本，都是可以上生产的版本，这里不存在任何质疑的可能！
- CI和CD唯一的差异：CD是不需要做CI过程中的手工校验的动作
- CD中究竟发那个版：不是技术问题，取决于业务和市场的需求，其实是政治问题

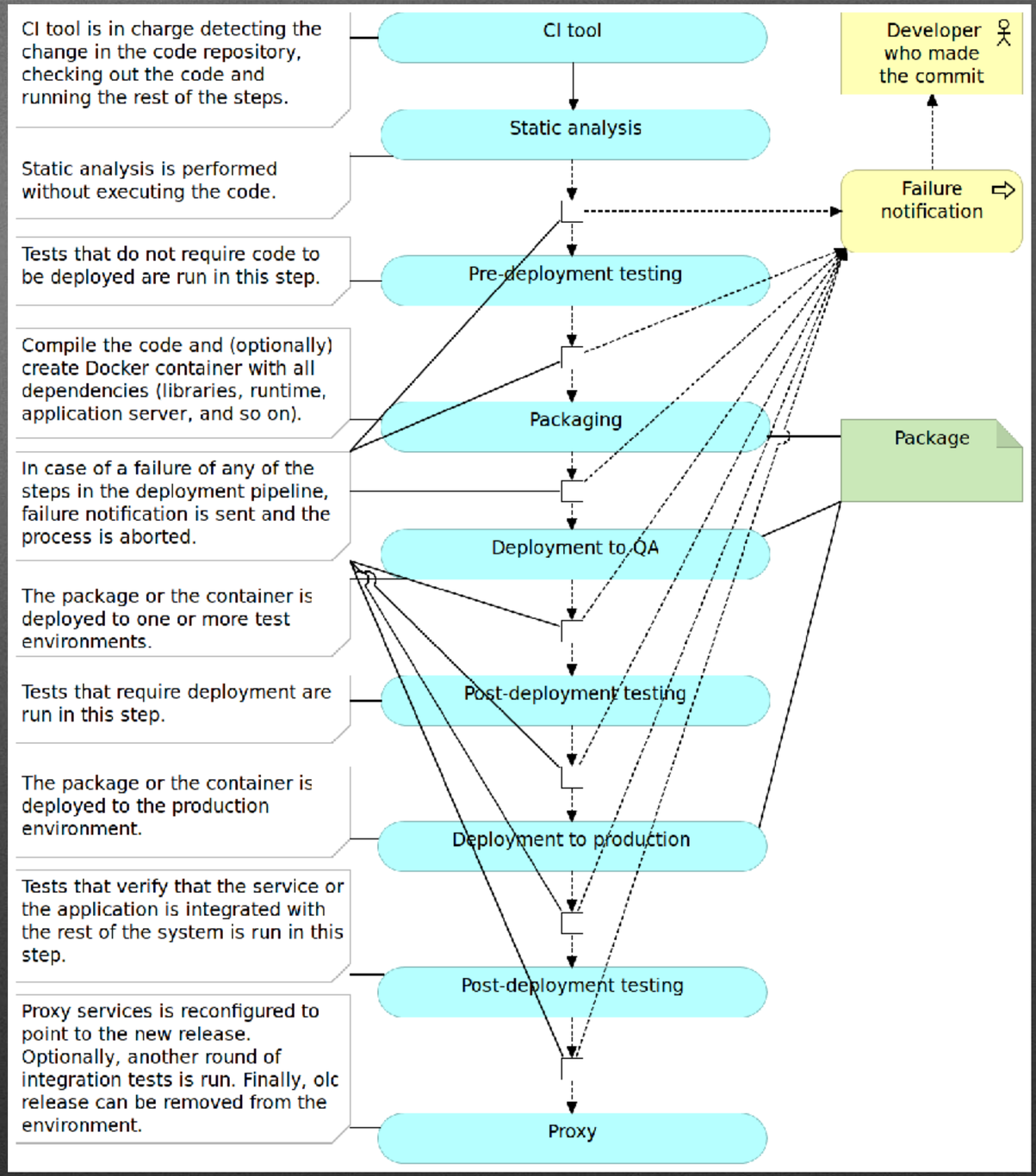
持续部署流程

- CI工具：监控代码库的变更
- CD流水线：运行一些列校验的测试任务；最后一步是部署到生产环境
- 发版/Promotion：如果整个流水线无失败，则提交此版本为当前的生产版本
- 失败通知：程序员收到通知



持续部署流水线

- 和CI的步骤相比，其实它就多了代理服务配置这一个动作
- CI-CD无本质区别，CD无手工QA的阶段
- CD从代码开始，走完所有步骤，最后应用服务部署到生产环境，更强调全自动化部署，无手工干预
- 部署后测试可能做两次或者多次（与部署的服务器数量相同的次数）或者只有集成测试
- 代理服务配置重新加载：让新的版本在生产环境中可见（还不是GA，还不存在回退的问题）



选择新版本GA的考虑

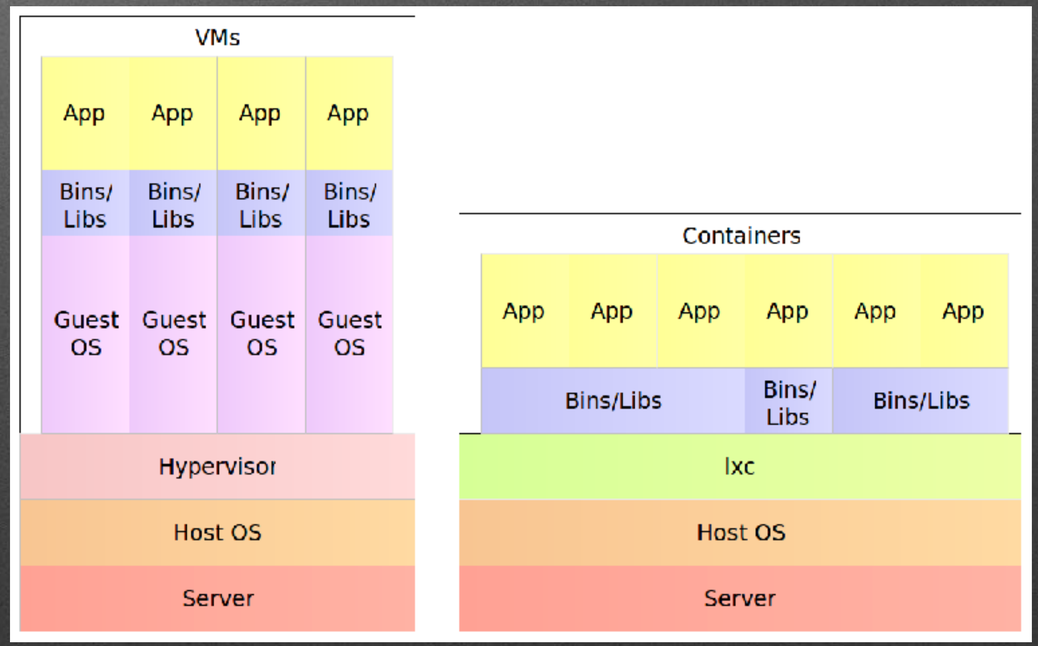
- 后端数据库需要在一段时间内对两个（多个）版本兼容，向后兼容老版本
- CD是绝对需要完整的集成测试，集成测试案例和CI阶段相同，需要照顾到生产环境中的其它系统
- 功能切换开关 feature toggles：例如临时禁用登录屏幕上的注册功能。为正在部署和更新的其它服务提供时间窗口，同时并不影响已注册用户的操作
- CD流水线的步骤，不仅表示了逻辑顺序，而且还是时间顺序，在获得了每个步骤的反馈之后，才能进入下一个步骤，否则停止流水线

微服务

- IT业务驱动是微服务火热的原因，新创新产出价值的周期需要越短越好；微服务正好能在时效上有帮助，它快的原因就是足够的小。巨大的单体应用，变更是在跟不上形势。
- 对于今天摆在每个人面前的压力而言，微服务可能是去践行的最好的架构类型
- 可以获得“用最好的工具解决特定的问题”的自由
- 但是它比处理单体架构系统的难度高

容器

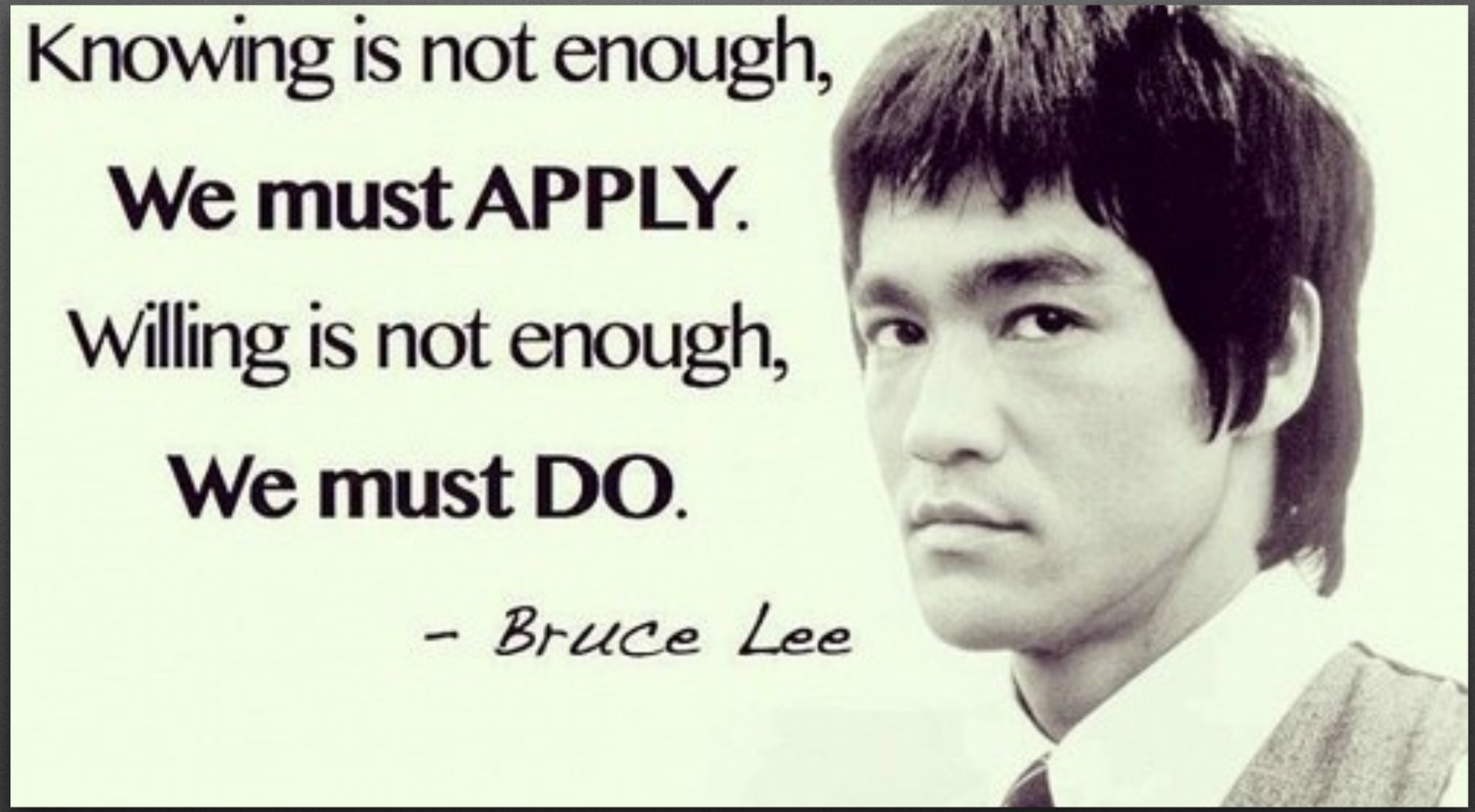
- 用来消除微服务部署的各种不爽；相对而言单体应用部署的逻辑比较简单，只需要做好流程标准化即可。但是标准化可是创新的克星
- 与其强制推行标准化部署流程，不如交付标准化的交付对象【容器】
- 容器的特性：服务的隔离性、不可变形、自给自足性、更高的计算密度



持续部署、微服务和容器的协同

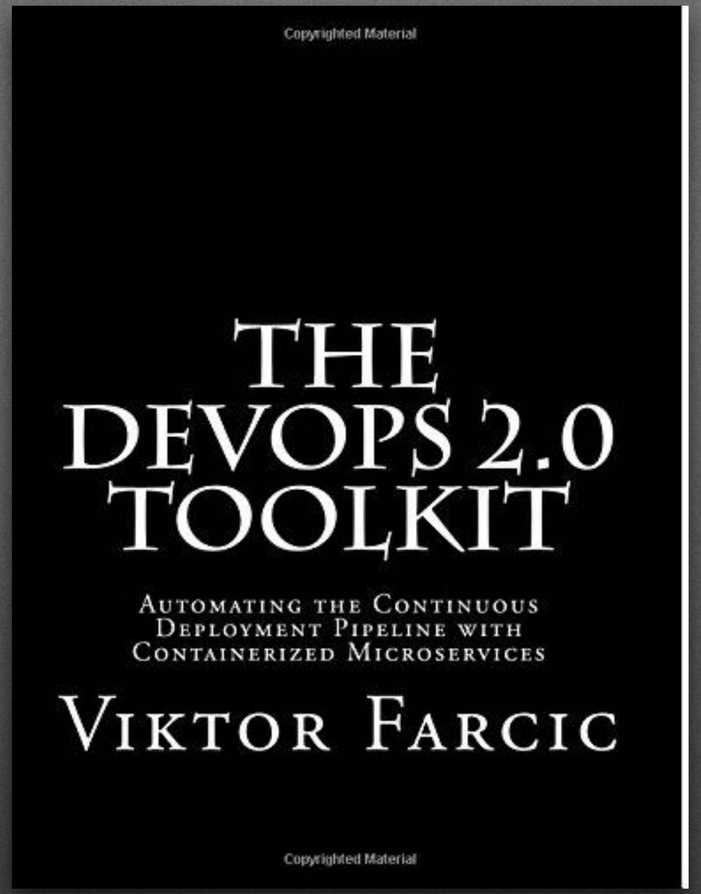
- 三个火枪手：这三个人简直是天作之合
- 持续部署：提供快速持续的反馈
- 微服务：自由决策、快速开发、更容易扩展
- 容器：解决了部署问题、帮助微服务落地、不可变性增加了服务可靠性





仅限于知道是不够的，我们必须去实践；单纯的希望是不够的，我们必须去行动。

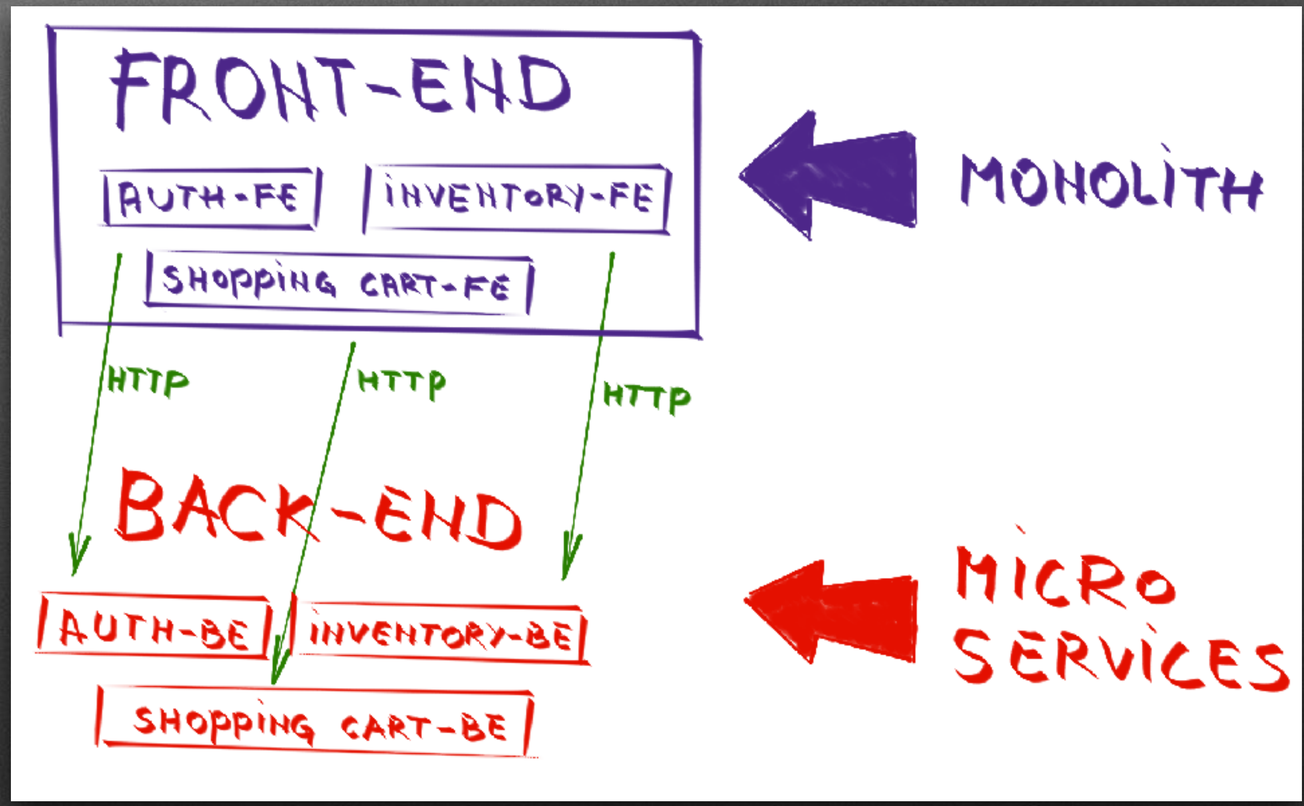
– Johann Wolfgang von Goethe 歌德


Copyrighted Material
THE DEVOPS 2.0 TOOLKIT
Automating the Continuous Deployment Pipeline with Containerized Microservices
Viktor Farcic
Copyrighted Material

第三章 系统架构

本书示例应用：线上书店假象的需求

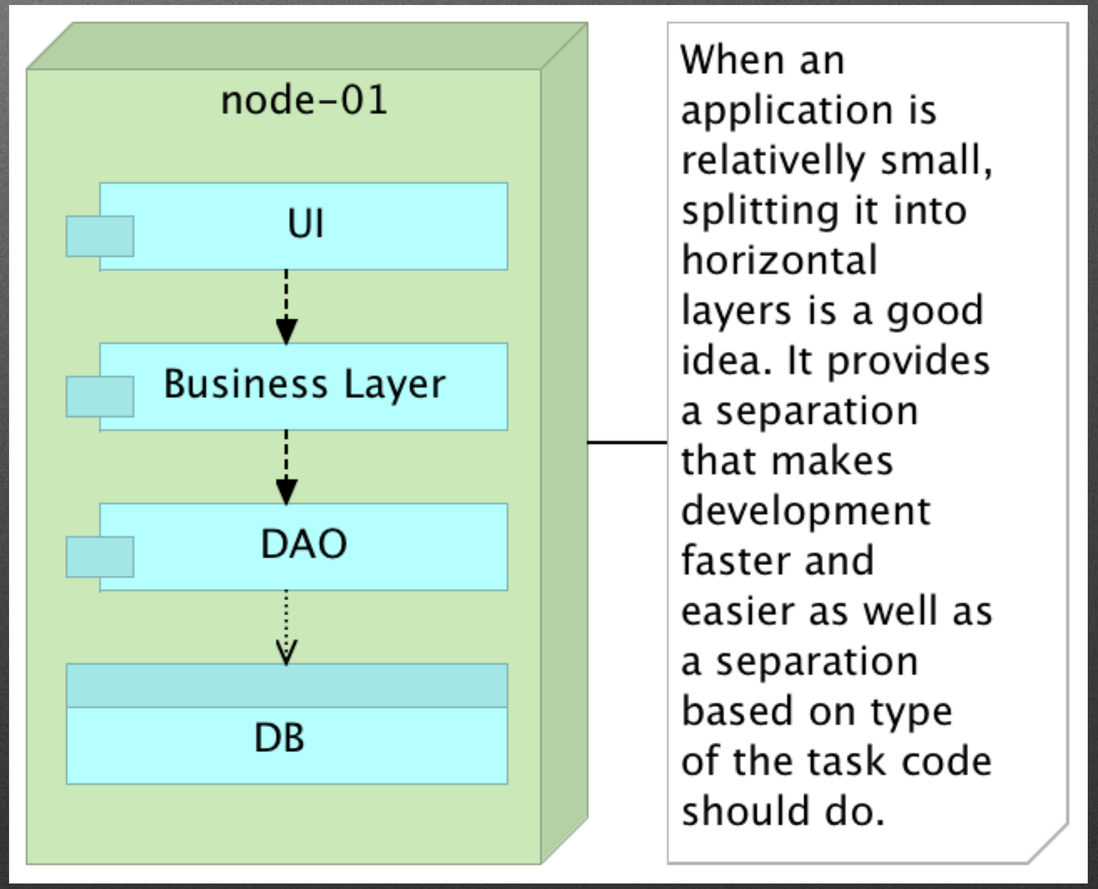
- 目标卖书，具体怎么卖？不知道！
- 设计成容易扩展的应用服务
- 没有完整的需求、但对未知需求要做好准备
- 要能快速地响应需求、易于扩展的系统
- 即使是新版本发布也无服务中断窗口
- 面对这些需求：请从单体和微服务架构之间做出选择



单体应用

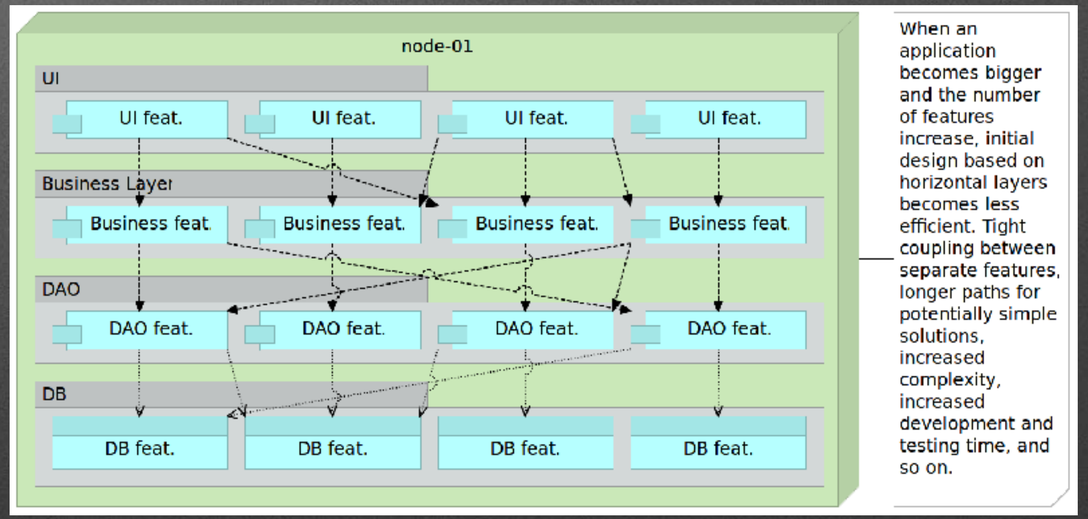
简单初级的单体应用

- 由一种语言编写，一个项目，一个WAR/JAR
- 软件的大小和复杂度和开发时间成正比
- 开发速度和时间成反比



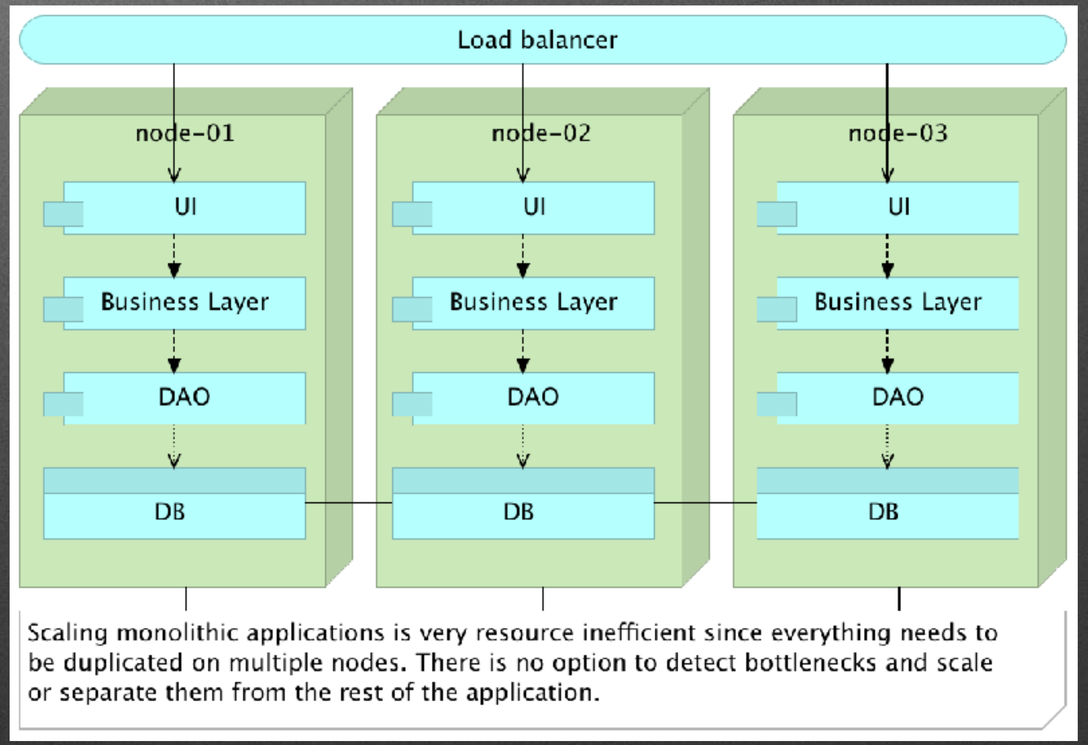
逐渐复杂的单体应用

- 逻辑上垂直分为四层，初期工作效率会提高、入市时间会缩短、代码整体会更清晰
- 随后：功能增加导致了复杂度增加；UI的一个功能变更可能导致下层所有层的多点变更；实际上任何层的变更会导致系统范围内未知风险性的提升
- 任何层程序变更之后，都必须做涉及所有层的整体测试，任何层都不太可能做替换
- 组件和模块之间紧耦合



单体应用的系统扩展

- 单体应用的每一层的计算/网络资源利用率是不平衡的
- 任何层的某模块性能瓶可能会导致整个节点的故障
- 扩展新的节点需要部署全套应用的所有模块



水平切分服务

SOA 架构

- SOA的提出是为了解决紧耦合单体应用的问题，它的实施概念如下：
  - 边界是显性的
  - 服务是自治的
  - 服务共享的是数据定义和通讯方式，而不是类
  - 服务之间不基于策略而相互兼容
- SOA催生了ESB企业服务总线

- 通常SOA的实施都是基于已有应用系统，对之前的开发模式并没有改变
- 可能的前期收益：每一层可以物理上彼此分布、开发和部署相对独立进行、扩展性提升
- 往往需要采购某种ESB产品，ESB负责服务之间的转换和重定向。
- 残酷的现实：开发出了一个更大的单体应用，它和所有服务紧耦合在一起，切分的目的实现了么？最终可能整个数据中心都会成为一个最大的单体应用

微服务

微服务架构

- 微服务的目标：
  - 通过上下文边界切割服务
  - 从物理上分割服务
  - 每个子服务独立运行自己的进程
  - 清晰地定义子服务之间的通讯
- 子服务之间的不依赖性是关键，独立开发、独立测试、独立运行
- 子服务间的关系：通过彼此暴露API的端口做数据交换
- 微服务使用和继承着SOA所定义的概念
- 微服务和传统SOA的差异：微服务最后会是自给自足和独立部署的；而SOA倾向于实施出一个更大的单体应用

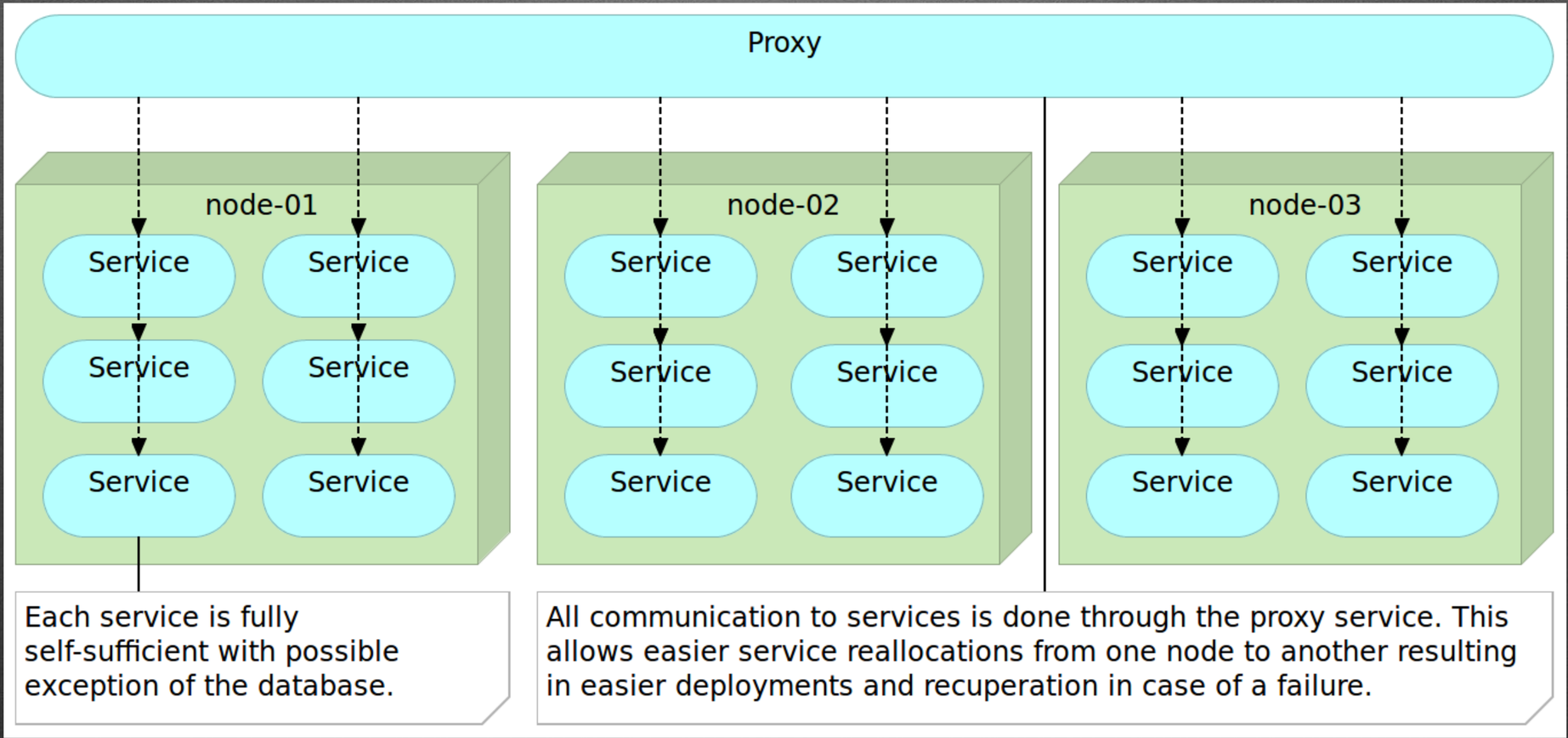
“Microservice architectures promise to deliver flexibility and scalability to the development and deployment of service-based applications. But how is that promise delivered? In short, by adopting an architecture that allows individual services to be built and deployed independently and dynamically; an architecture that embraces DevOps practices. Microservices are simpler, developers get more productive and systems can be scaled quickly and precisely, rather than in large monolithic globs. And I have not even mentioned the potential for polyglot coding and data persistence.”

–Gartner Gary Olliffe

微服务关键点

- 只做一件事，或者只对一个功能负责
- 由于彼此的不依赖性，每个微服务可以用任何一组工具或者语言来构建
- 由于彼此物理的离散性，它们是真正的松耦合
- 开发不同微服务的团队之间相对比较独立，他们暴露的API是提前定义好的
- 更容易地测试和持续交付或者部署

微服务的建议架构



Https://martinliu.cn DevOps教练 微信：MyDevOps

单体应用和微服务的对比

对比示意图

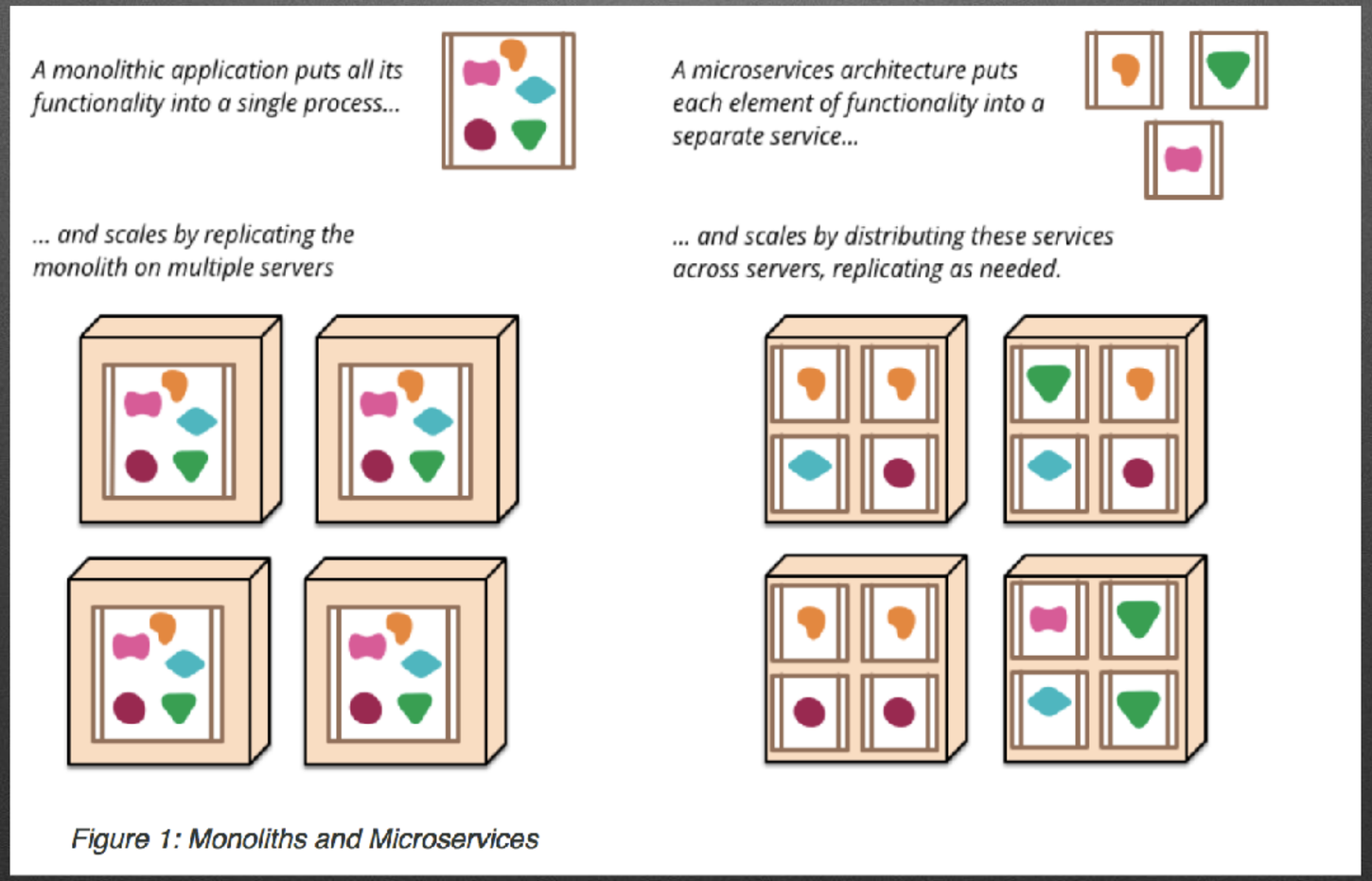


Figure 1: Monoliths and Microservices

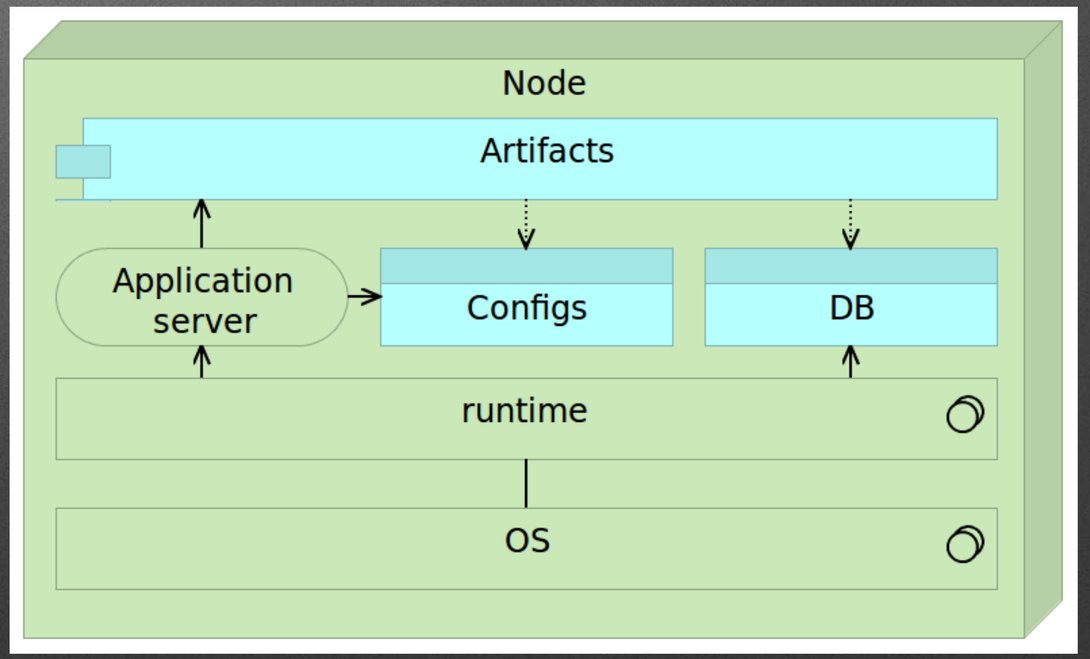
Https://martinliu.cn DevOps教练 微信：MyDevOps

	运维部署的复杂度	远程进程调用	扩展	创新	大小	部署、回滚故障隔离	期限承诺
微服务	更高，必须借助配置管理工具和Docker	可能性能较低	更快、按需扩展、更好的资源利用率	创新的乐土	小、容易理解、更快速	更快、更容易、很好地隔离故障	周期短、船小好调头
单体应用	正常水平	类和方法之间可能更快	复制整机堆栈、慢速	创新杀手	大、笨重	慢、困难、成本高、风险高	周期长、不靠谱、没有回头路、无后悔药

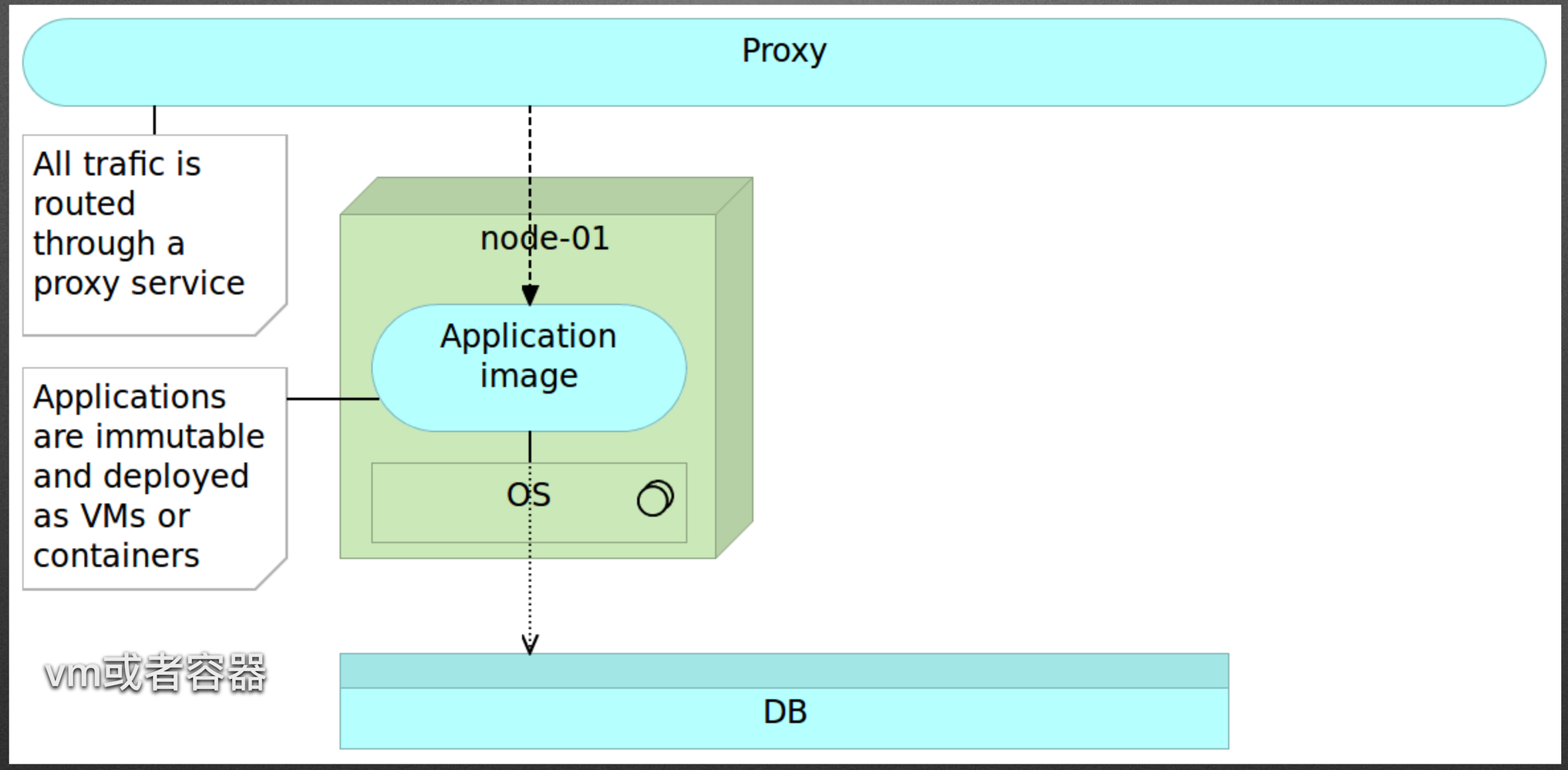
部署策略

可变更的巨兽型服务器

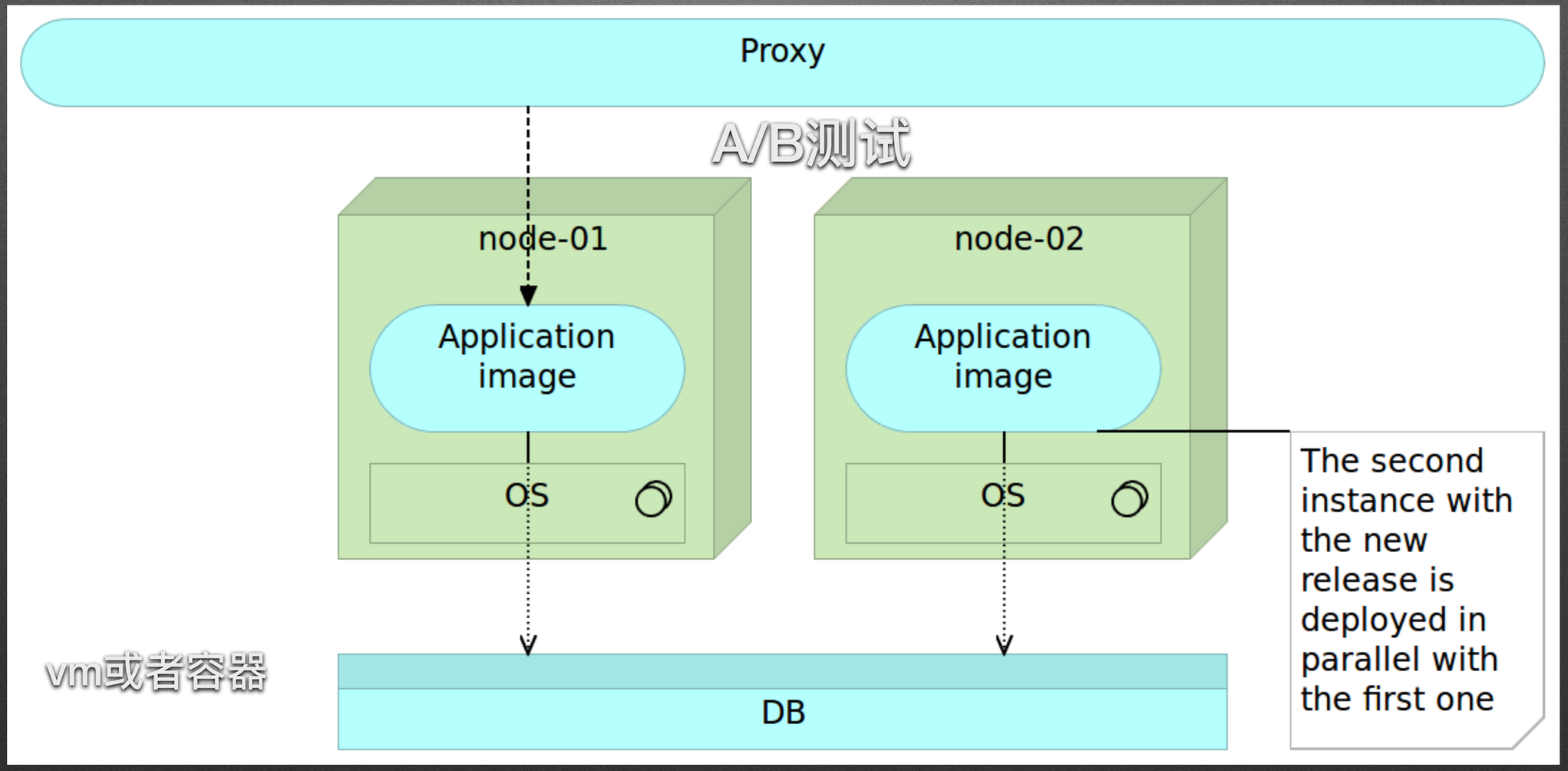
- 持续更新已有的旧应用代码制品物、数据库的定义
- 通常由手工部署，节点之间配置难于一致，VM对一致性改进甚微
- 每个节点的当前状态都是经年累月变更操作积累的结果，无法回溯到之前的任何时间点
- 当年小而美的项目，最终变成面目全非的怪兽，最终在测试和部署各个阶段都很难驯服
- 零宕机时间变得越来越遥不可及



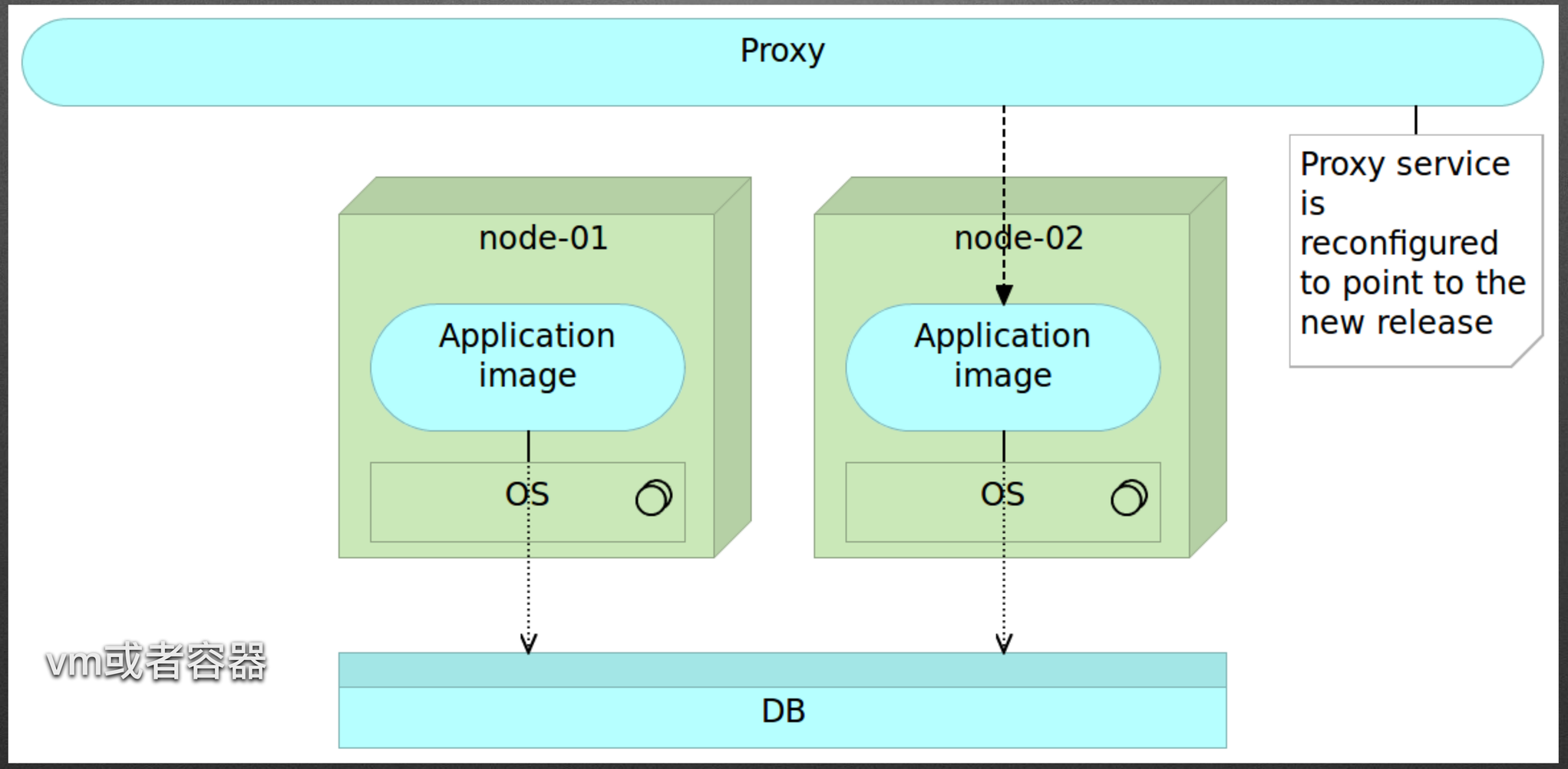
不可变更服务器升级过程 — 版本1投产中



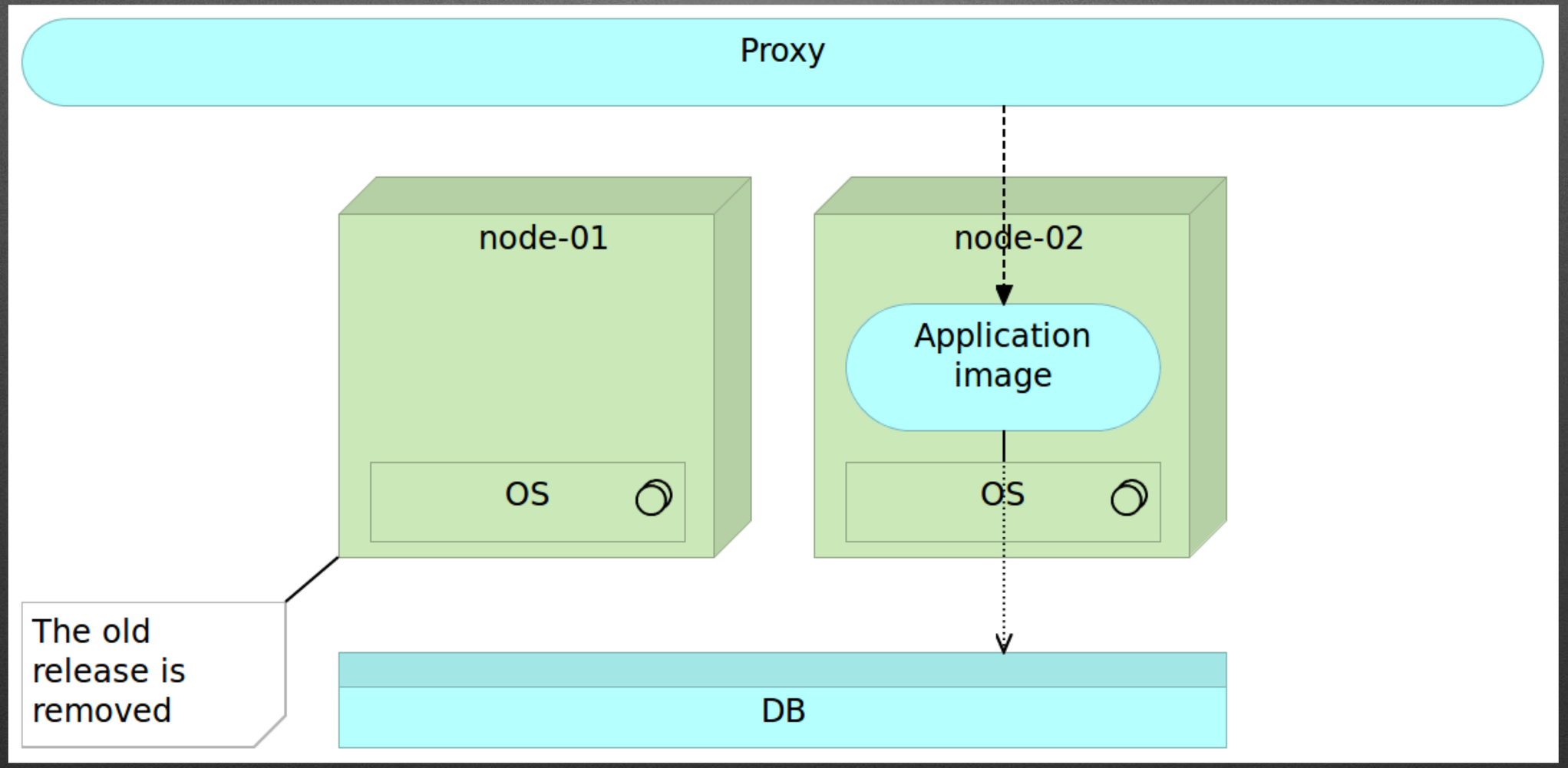
不可变更服务器和反向代理 — 版本1.5预发布



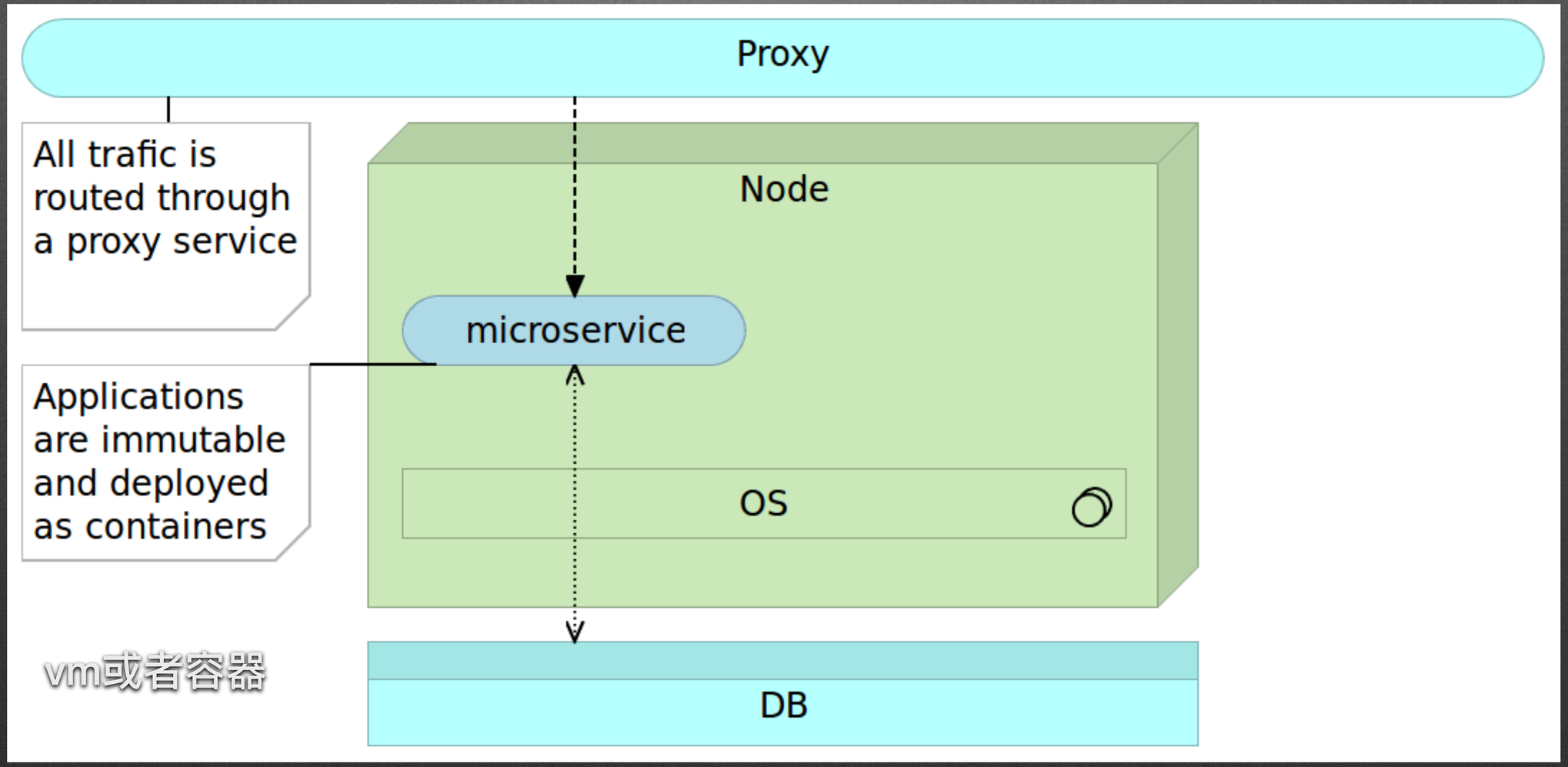
不可变更服务器和反向代理 — 版本1.5上线



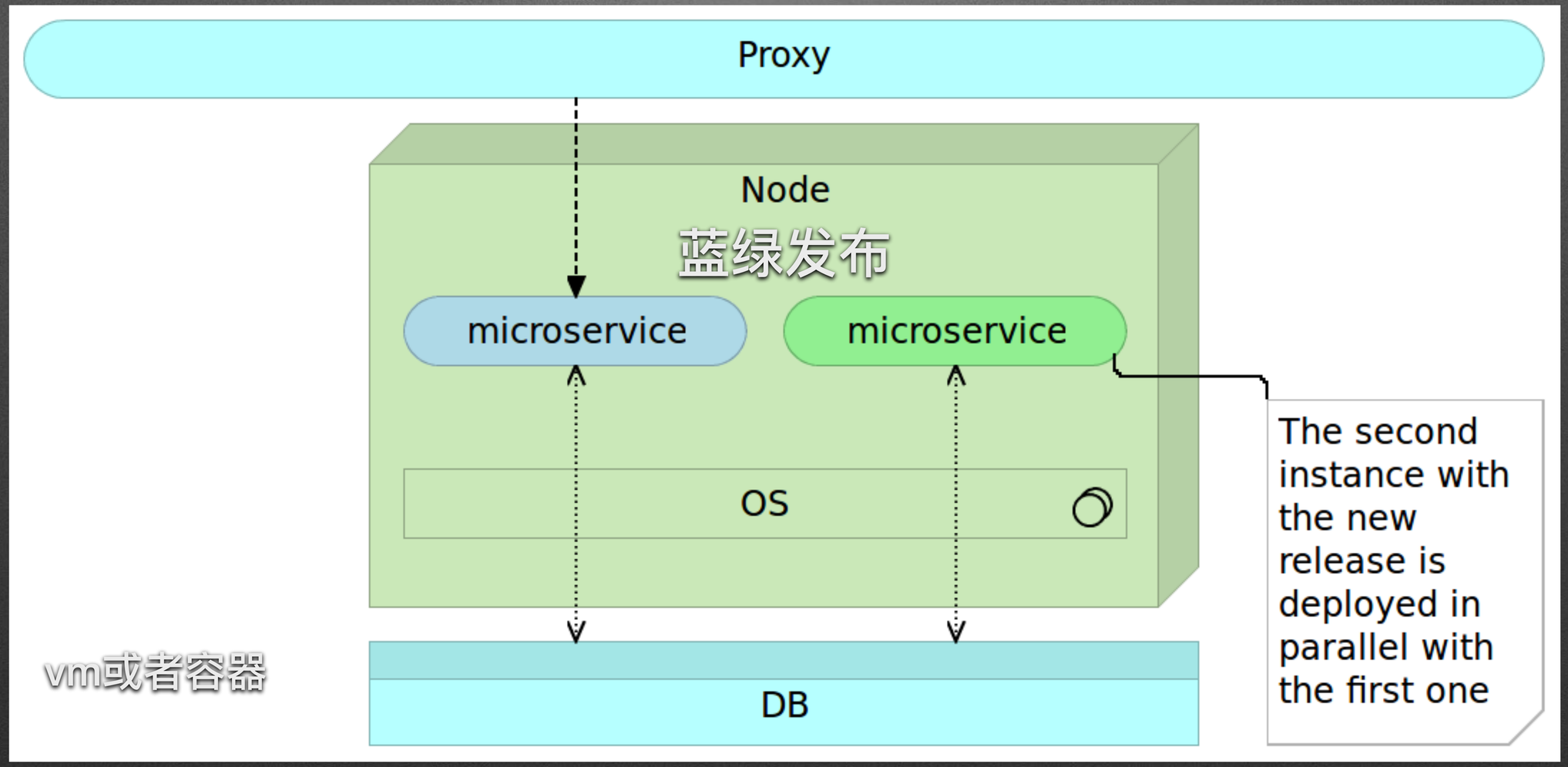
不可变更服务器和反向代理 – 版本1下线



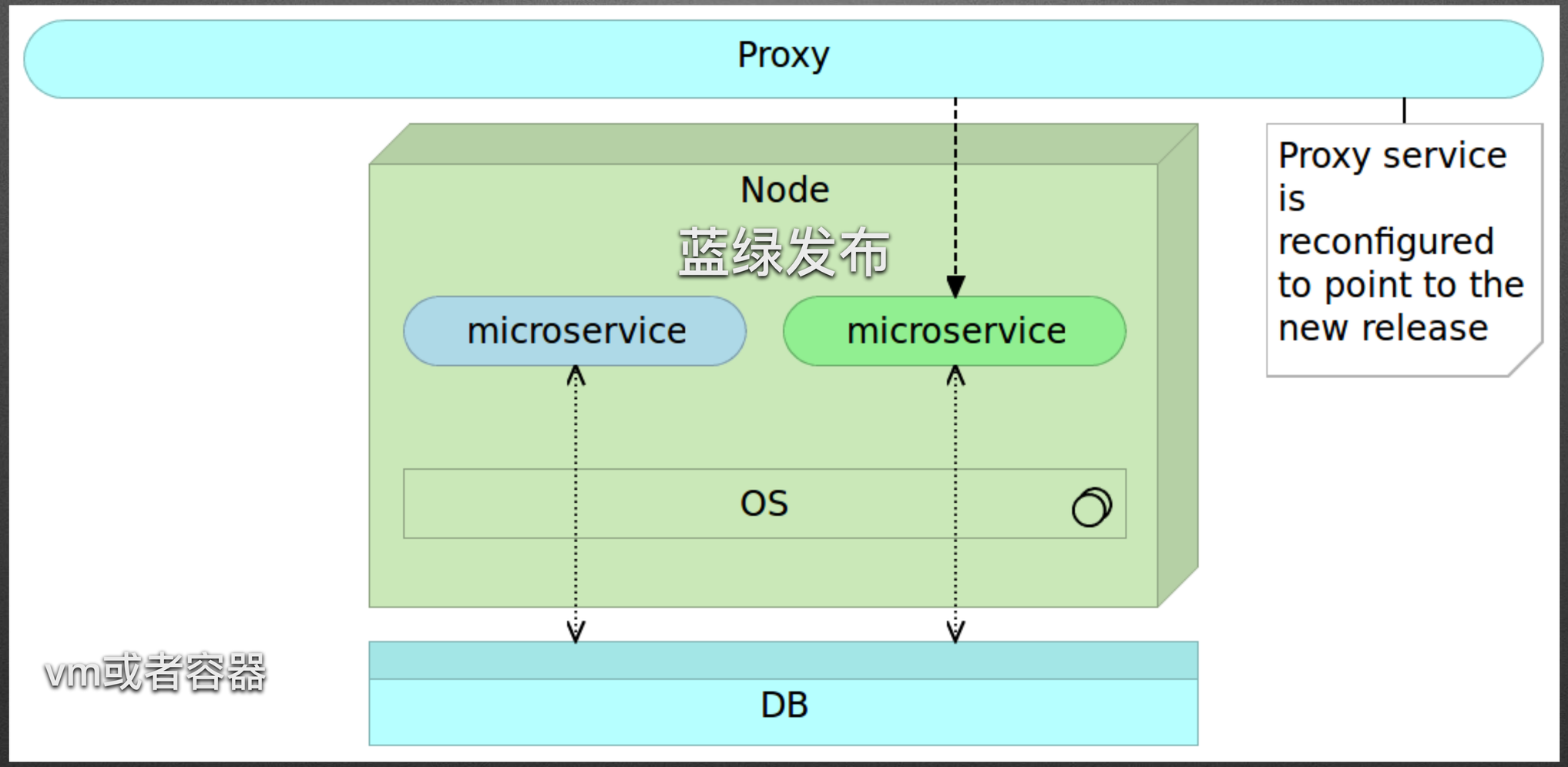
不可变更微服务升级过程 — 版本1投产中



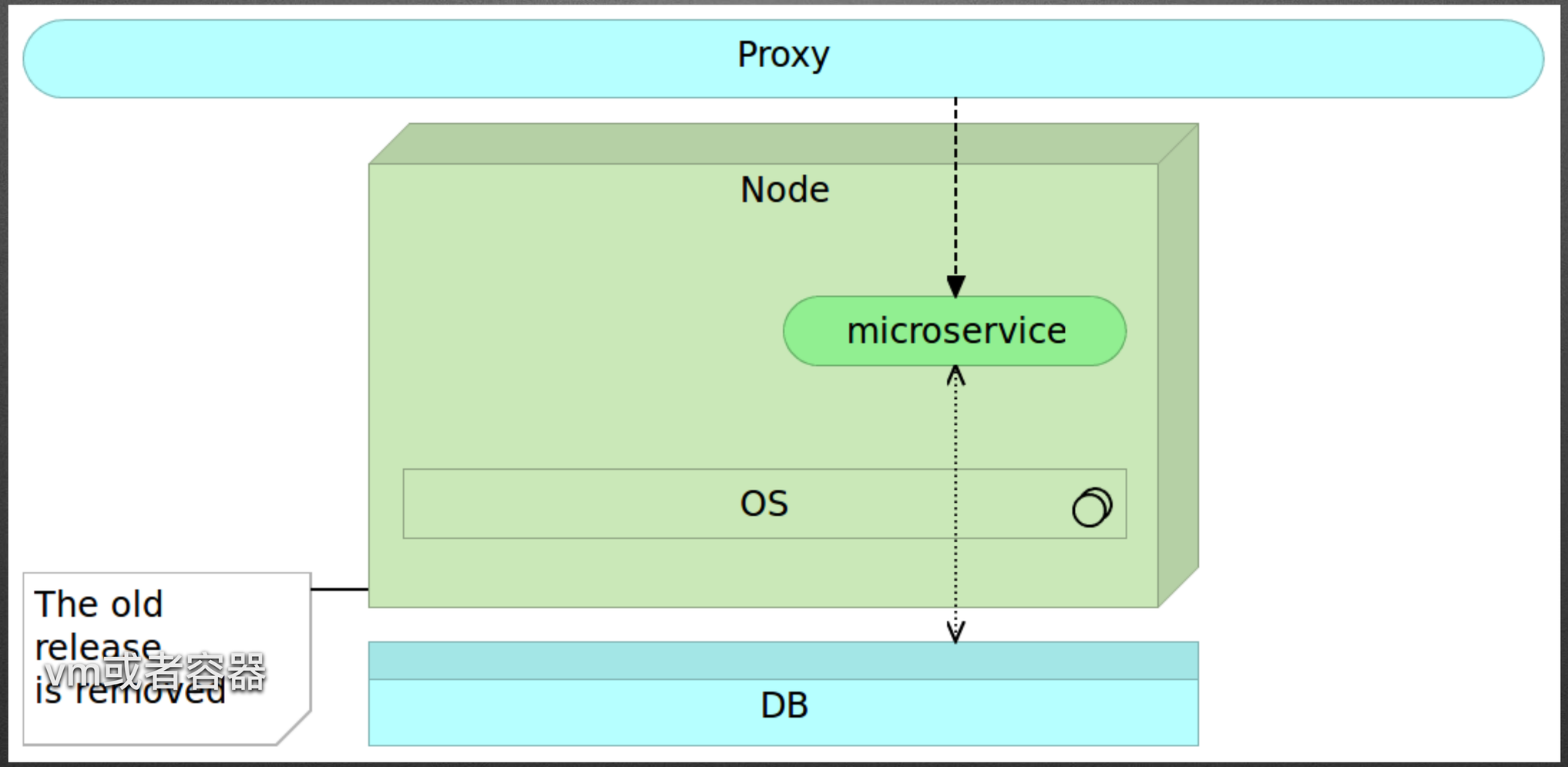
不可变更微服务 – 版本1.5预发布



不可变更微服务 — 版本1.5发布

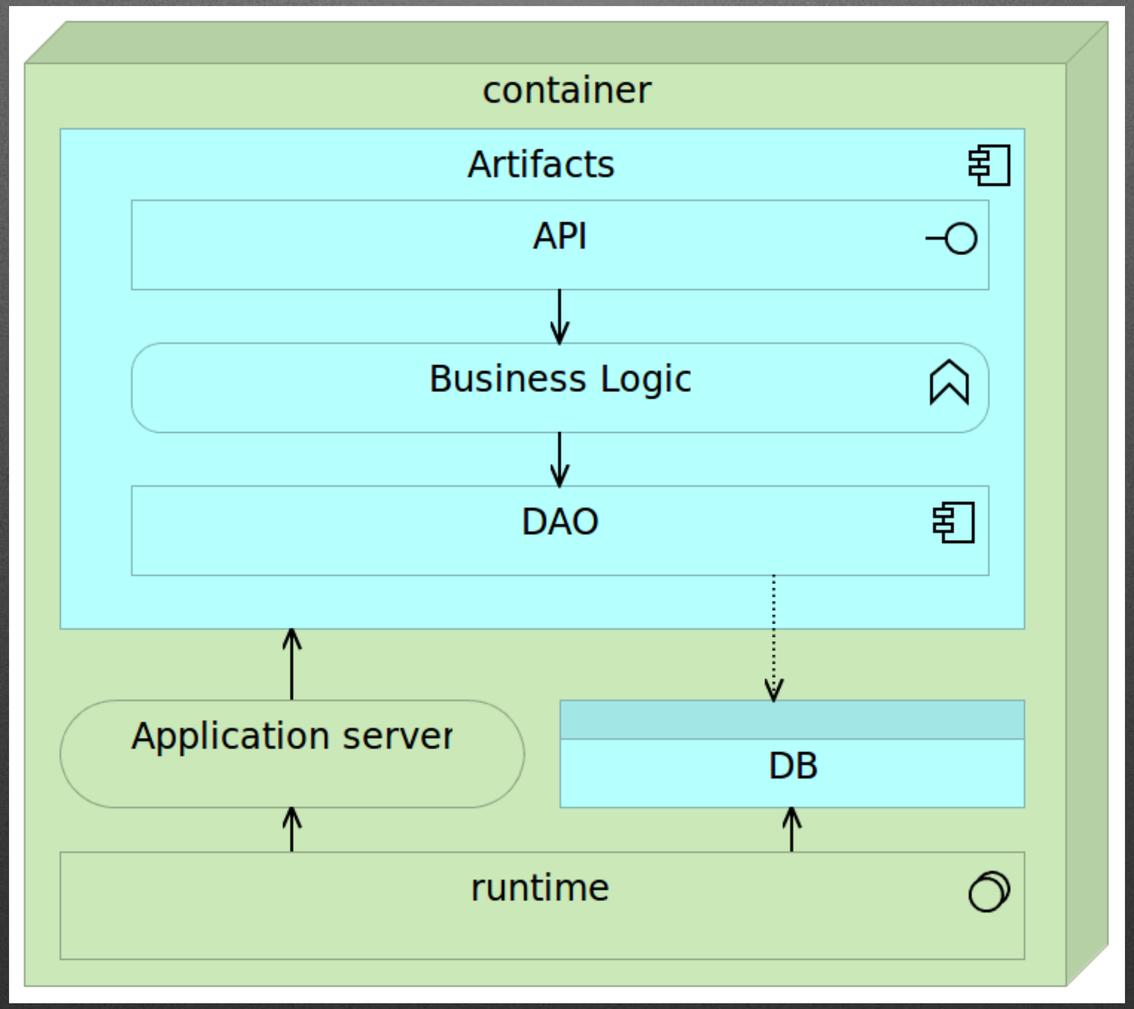


不可变更微服务 – 版本1下线

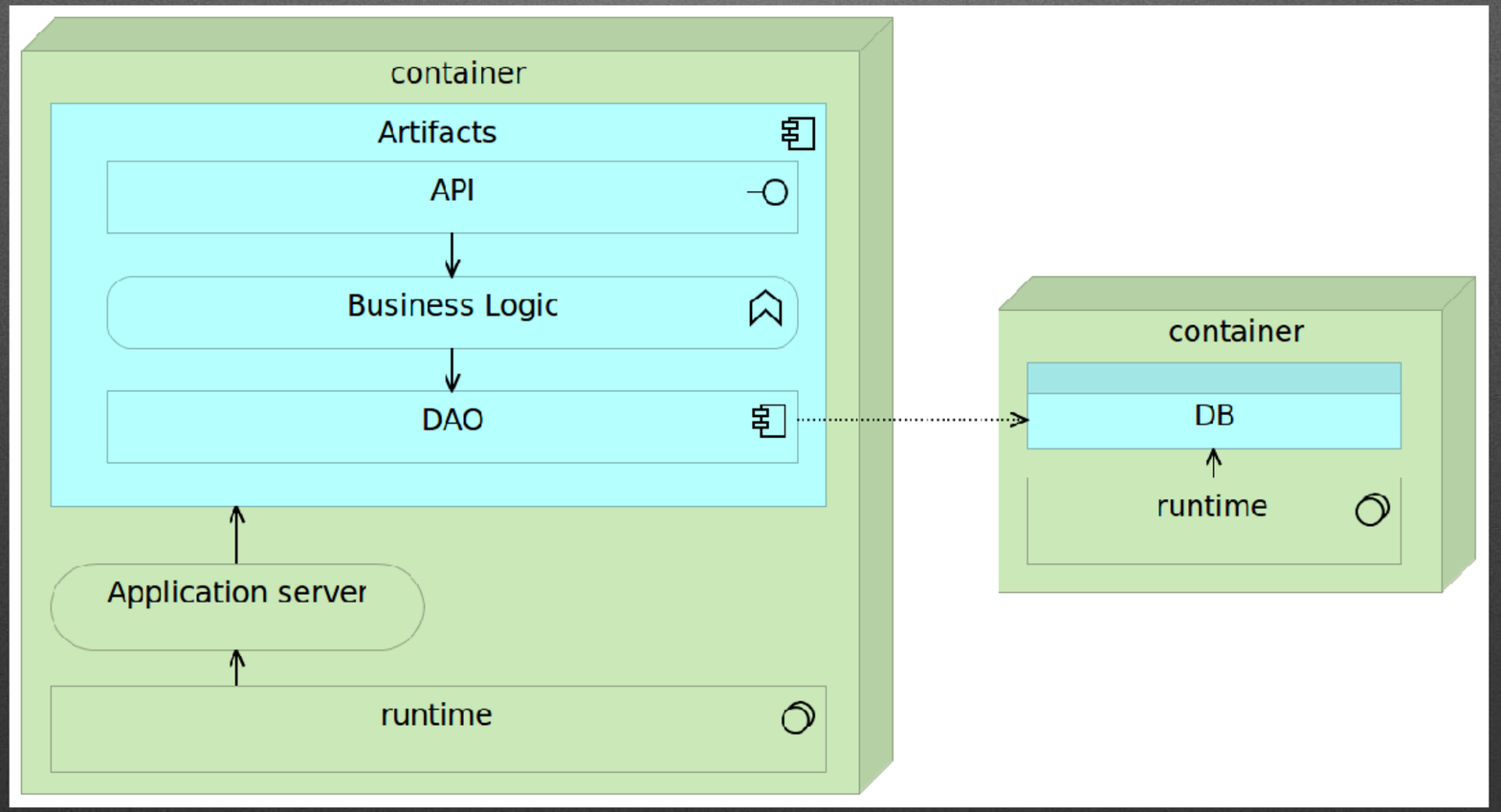


微服务最佳实践

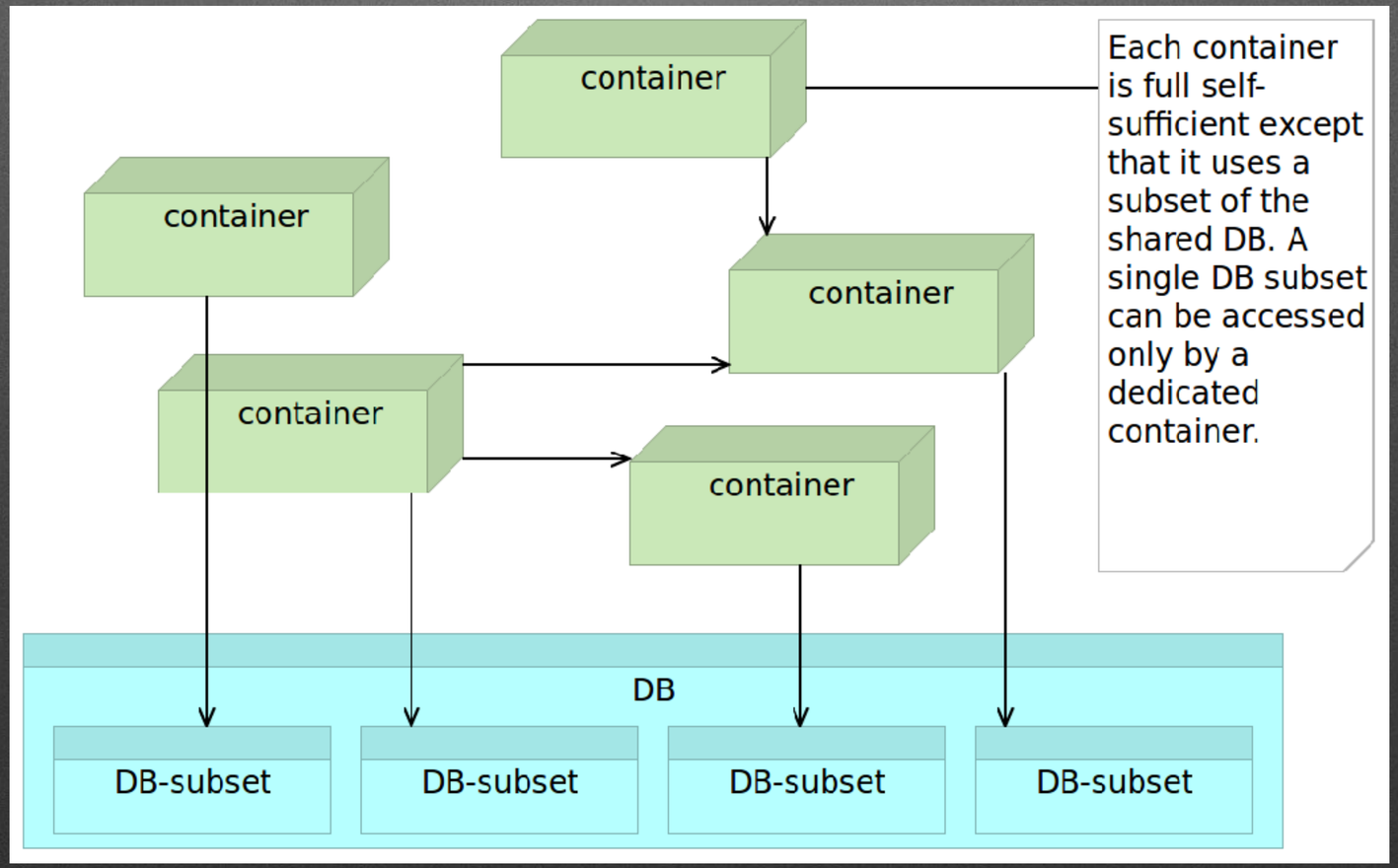
容器自给自足：内置数据库服务模型



容器自给自足：使用专有数据库服务模型



容器自给自足： 访问共享数据库模型



其它建议

- 使用Proxy微服务或者API网关
- 使用反向代理：Nginx，Apache Tomcat， HAProxy
- 应用极简架构风格：JBoss，Tomcat，Spray，CoreOS
- 应用配置自动化管理工具：Puppet，Chef，Ansible
- 重新做开发团队的配置：跨功能团队的建立
- 时间API Versioning功能：保证服务API的向后兼容和多版本共存

总结

微服务的最经典类比 — Unix/Linux

```
ps aux | grep jav[a] | awk '{print $2}' | xargs kill
```

项目目标：杀死系统中所有正在运行的java进程

设计方案：使用命令行工具加管道符功能

项目效果：一指禅！

You Got It ?

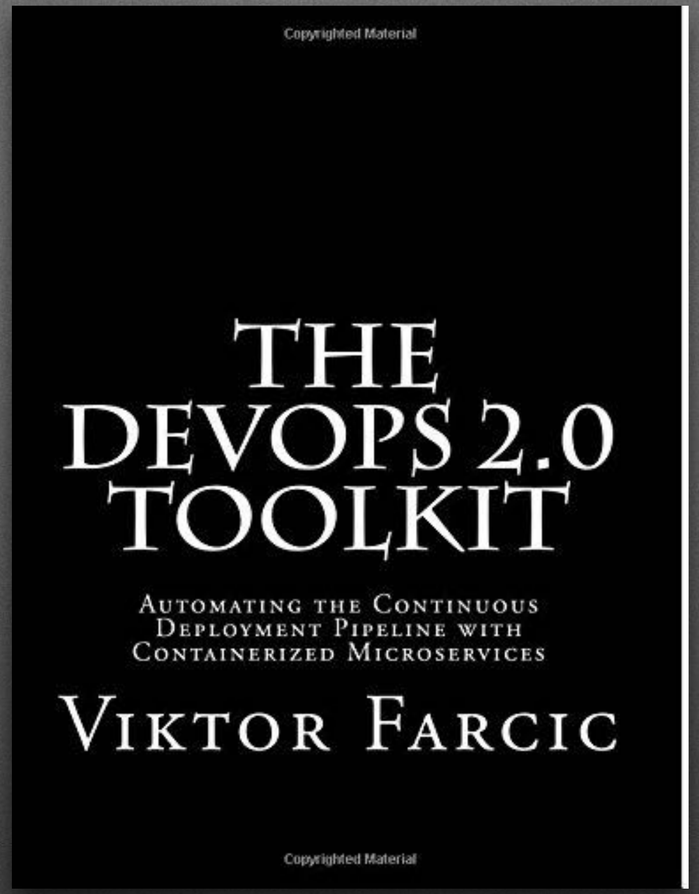
Https://martinliu.cn DevOps教练 微信：MyDevOps



“Gather together those things that change for the same reason, and separate those things that change for different reasons”

–Robert C. Martin

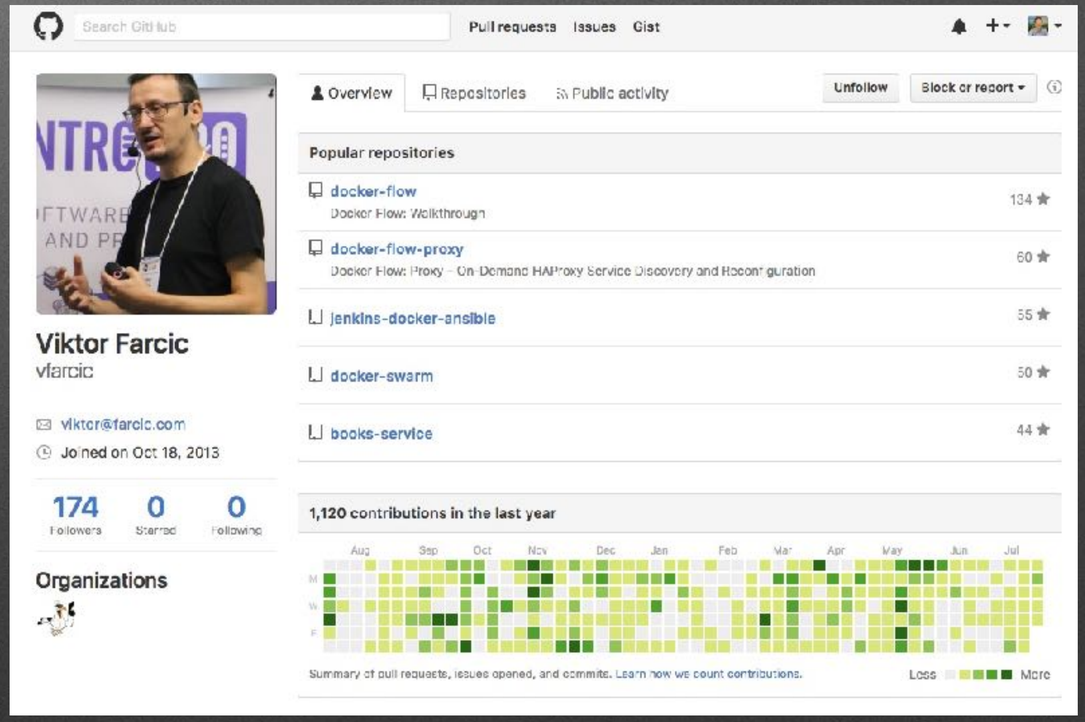
Https://martinliu.cn DevOps教练 微信：MyDevOps



第四章 用Vagrant和Docker配置开发环境

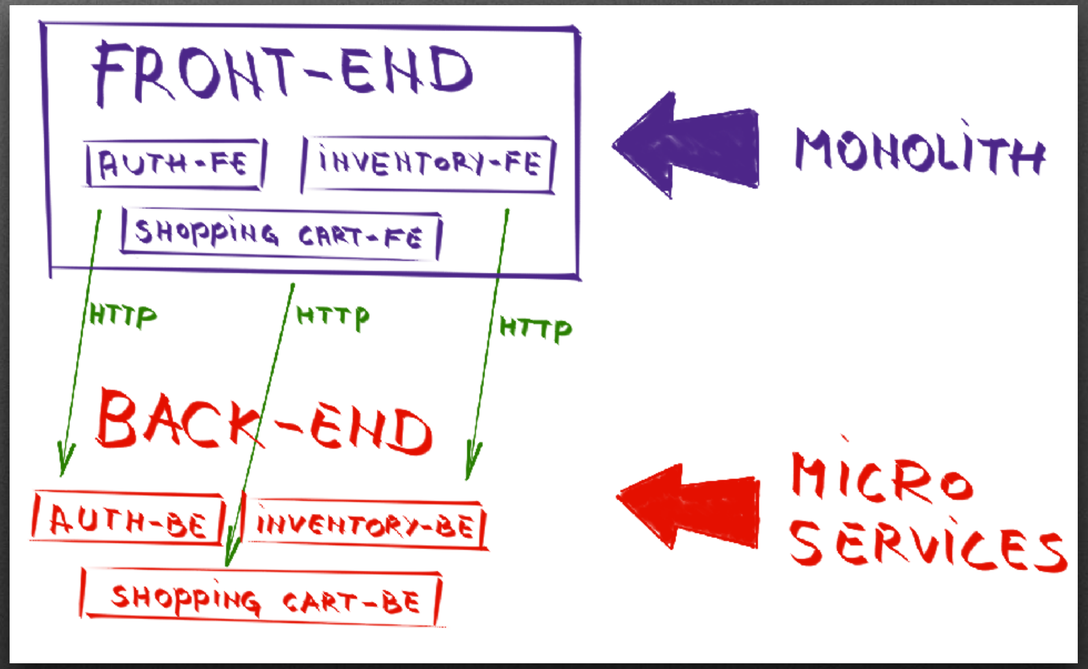
开发测试项目信息

- 本书所有代码都在作者的Github仓库中
- 左下角的图标是Docker Caption 刀客船长的标志，是Dcoker公司认可的Docker大牛
- 本章测试的代码：https://github.com/vfarcic/books-ms.git



项目使用了微服务架构和容器技术

- 项目有两部分：前端和后端
- 容器和微服务对开发环境配置的好处：
  - ✦ 微服务代码比单体应用更容易理解
  - ✦ 配合Vagrant可以使开发环境配置的标准化而快速
  - ✦ 开发的结果和生产环境趋于完全一致
- 前端：单体结构；后端微服务结构
- 此架构是本书示例代码，并不是参考架构
- 后端使用的技术：Scala、Spary、Polymer



Vagrant 和 Docker



- 作者的开发环境是Ubuntu Linux + Vagrant + Docker + VitualBox
- 我的测试环境：Mac OS X + Vagrant + Docker for Mac + VirtualBox
- 我的硬件：11年款 MacBook Pro（8GB RAM + SSD），我推荐的配置
- Windows桌面系统：Vagrant + Docker for Windows + VirtualBox + Git + SSH；
  以上工具的命令都在当前的PATH中
- 本机已经下载了 ubuntu/trusty64 Vagrant 镜像，用 vagrant box list 命令可以看到

开发环境配置和启动

```
1    git clone https://github.com/vfarcic/books-ms.git
2    cd books-ms
3    vagrant plugin install vagrant-cachier
4    vagrant up dev
```

1.下载测试项目代码到本地

2.进入代码目录

3.安装vagrant 缓存插件

4.启动名字为dev的虚拟机，2GB内存的Ubuntu虚拟

dev虚拟机启动做了什么？

- 读配置文件Vagrantfile，配置dev虚拟机私网IP地址为：10.100.199.200
- 挂载当前目录到虚拟机的 /vagrant ；这样所有代码和脚本就都在虚机里面可见了
- 操作系统启动后，执行 bootstrap.sh 脚本，读这个脚本：安装 ansible
- 执行ansible的playbook，读playbook的task：添加docer软件仓库、安装docker服务、安装python-pip/python-py、安装docker-compose、设置docker-compose执行权限

上一步可能遇到的问题

- ubuntu更新的源默认不是国内的源服务器，可能比较慢，可以修改bootstrap.sh脚本配置国内ubuntu更新源
- 安装ansible的过程可能由于国外的源的关系有点慢
- 安装docker-compose可能失败，建议用高速宽带，使用国外vpn

登录dev开发虚拟机的操作系统

- vagrant ssh dev // 从桌面操作系统登录到dev Ubuntu 开发机操作系统中
- ansible --version //检查Ansible是否安装成功
- docker —version //检查docker是否安装成功
- docker-compose --version //检查docker-compose是否安装成功
- cd /vagrant //进入开发的项目目录，就是book-ms的所有代码目录
- 可能的问题：ansible安装失败，docker-Compose安装失败，修复以后才能继续

启动前端程序并测试

```
sudo docker run -it --rm \

-v $PWD/client/components:/source/client/components \

-v $PWD/client/test:/source/client/test \

-v $PWD/src:/source/src \

-v $PWD/target:/source/target \

-p 8080:8080 \

--env TEST_TYPE=watch-front \

vfarcic/books-ms-tests
```

上一步可能遇到的问题

- vfarcic/books-ms-tests 这个镜像有1.2+GB，建议翻墙下载好，放到 book-ms 目录下，执行本测试前先docker load它到Ubuntu dev 虚拟机
- 这个过程中有很多东西需要下载，他们都是前端程序所依赖的库
- 期望的结果 在桌面机的浏览器中访问 http://10.100.199.200:8080/components/tc-books/demo/index.html 可以看到正常的页面显示
- 到命令行，按 CTRL+c ，终止这个容器，第一个测试结束

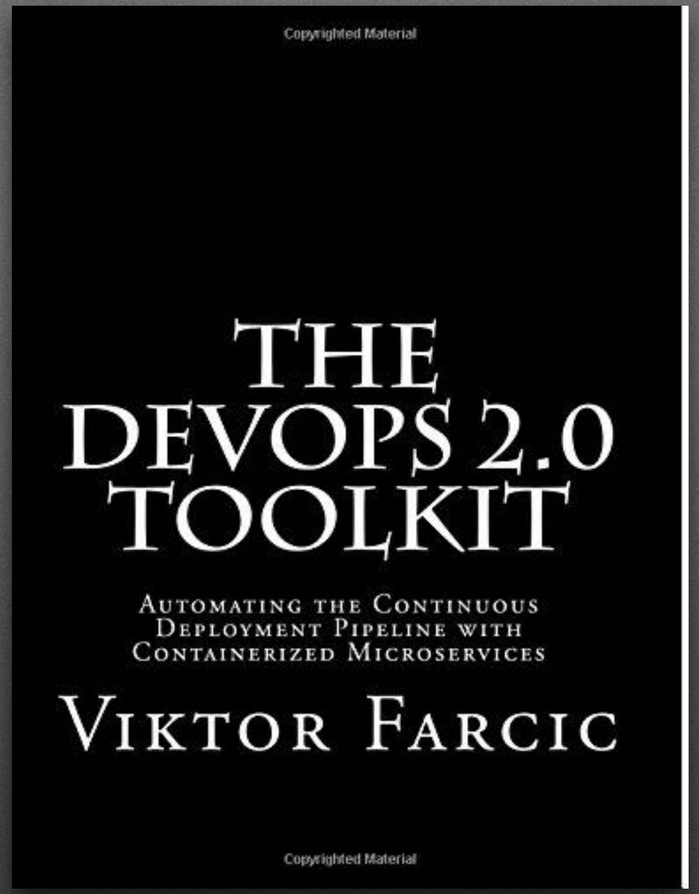
执行其它操作

- sudo docker-compose -f docker-compose-dev.yml run feTestsLocal
- CTRL+c
- sudo docker-compose -f docker-compose-dev.yml run testsLocal
- CTRL+c
- 执行另外两个测试，必须看的文件：Dockerfile、 docker-compose-dev.yml



“The big idea is ‘messaging’. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.”

–Alan Kay



第五章 实施部署流水线：初始化阶段

启动持续部署虚机

测试环境介绍

10.100.198.200



roles:
- common
- docker
- docker-compose
- docker-flow
- consul-template
- registry

- 启动虚拟机前，确保你的机器有墙外的高速网络连接，个人或者是公司的VPN；或者选择墙外的云主机
- 读一下这几个Ansible的role文件，最主要的是Registry功能，本章最后一步是push两个docker镜像到镜像库中
- 本章测试过程为纯手工，还没有涉及到Jenkins的部分

启动 cd 虚拟机的步骤

```
cd ..  //进入当前用户根目录

git clone https://github.com/vfarcic/ms-lifecycle.git  //下载cd启动和配置代码

cd ms-lifecycle   //进如配置代码目录

vagrant up cd   //启动cd虚拟机

vagrant ssh cd    // ssh 进入cd虚拟机

sudo ps    //应该能看到三个运行的容器
```

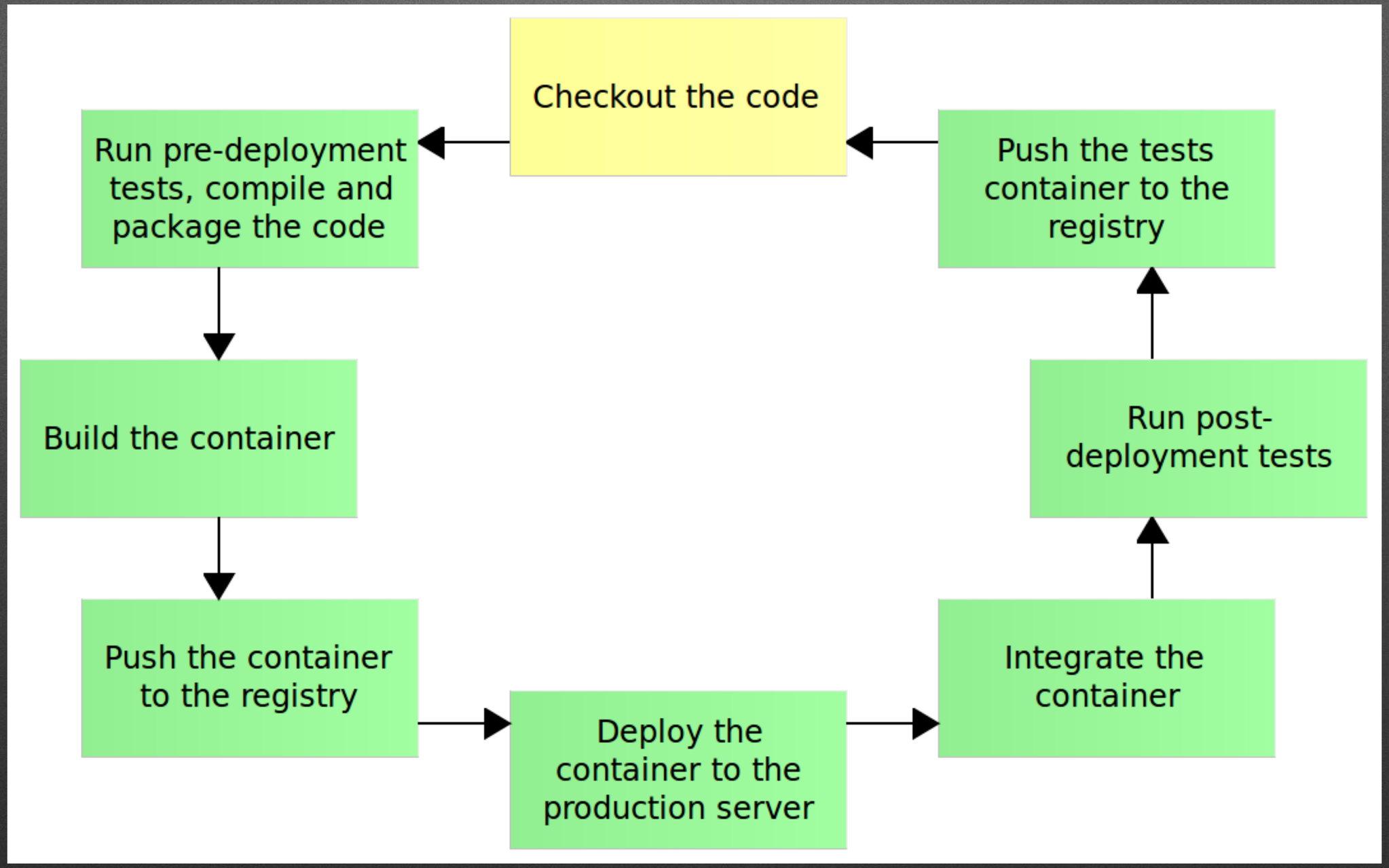
上一步正常的输出结果如下

```
ubuntu@ip-172-31-30-243:/vagrant$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
fe52c23977db        registry:2          "/bin/registry serve "   About a minute ago
Up About a minute   0.0.0.0:5001->5000/tcp   registry-mirror

a002375071f7        registry:2          "/bin/registry serve "   About a minute ago
Up About a minute   0.0.0.0:5000->5000/tcp   registry

2e11897bdc3b        uifd/ui-for-docker  "/dockerui"              About a minute ago
Up About a minute   0.0.0.0:9000->9000/tcp   dockerui
```

部署流水线的步骤



Https://martinliu.cn DevOps教练 微信：MyDevOps

步骤

1. 检出代码
2. 运行预部署测试
3. 编译和打包代码
4. 构建容器
5. 推送测试镜像到镜像仓库
6. 部署容器到生产服务器
7. 集成容器
8. 运行集成后测试
9. 推送测试镜像到镜像仓库

检出代码

- git clone https://github.com/vfarcic/books-ms.git //下载书店项目的程序代码
- cd books-ms //进入代码目录

运行预部署测试、编译和打包程序代码-1

- sudo docker build -f Dockerfile.test -t 10.100.198.200:5000/books-ms-tests .
- 根据 Dockerfile.test 文件来构建测试容器镜像
- 此条命令是本章中最耗时和最考验网络的命令，下载量2.5GB左右，是否能正常完成这个命令，完全取决于网络速度和稳定性，建议使用墙外云主机
- 这个镜像中包含了所有执行所有测试案例所需要的工具：Mongo, NodeJS, Git, SBT, xvfb, FireFox, Chrome

Dcokerfile.test 工作原理概述

- 使用Debin:jessie作为基础镜像
- 安装测试工具，应用服务器所需的所有依赖库，并安装测试工具
- 部署源代码、测试案例脚本和相关支持数据文件到容器中
- 设置基础环境变量，设置基础工作路径/卷
- 设置启动命令为 /source/run_tests.sh
- 书中对此文件做了详细的解释，很多是和此书店项目代码相关，建议阅读书中原文

上一步正常的输出结果如下

```
ubuntu@ip-172-31-30-243:~/books-ms$ sudo docker images
REPOSITORY                                TAG                 IMAGE ID            CREATED             SIZE
10.100.198.200:5000/books-ms-tests        latest              fc19ea46623b        About a minute ago  2.229 GB
registry                                  2                   8ff6a4aae657        6 weeks ago         171.5 MB
debian                                    jessie              1b088884749b        6 weeks ago         125.1 MB
uifd/ui-for-docker                        latest              c2b1b90a9e75        12 weeks ago        7.391 MB
```

运行预部署测试、编译和打包程序代码-2

- sudo docker-compose -f docker-compose-dev.yml run --rm tests
- 这个步骤使用上条命令构建出来的镜像和MongoDB镜像为基础启动测试环境
- 编译书店项目代码：Scala —> JAR ； 加载构建物到应用服务器
- 执行13个测试案例

上一步正常的输出结果如下

屏幕输出结果中应该包含这一段

[info] Total for specification ServiceSpec

[info] Finished in 906 ms

[info] 13 examples, 0 failure, 0 error

[info] Passed: Total 13, Failed 0, Errors 0, Passed 13

[success] Total time: 36 s, completed Jul 23, 2016 2:46:07 PM

运行预部署测试、编译和打包程序代码-3

```
ubuntu@ip-172-31-30-243:~/books-ms$ ll target/scala-2.10/     //查看书店应用代码最终的编译后构建物
total 38296
drwxr-xr-x 4 root root     4096 Jul 23 14:46 ./
drwxr-xr-x 3 root root     4096 Jul 23 14:45 ../
-rw-r--r-- 1 root root 39198710 Jul 23 14:46 books-ms-assembly-1.0.jar
drwxr-xr-x 3 root root     4096 Jul 23 14:45 classes/
drwxr-xr-x 3 root root     4096 Jul 23 14:46 test-classes/
ubuntu@ip-172-31-30-243:~/books-ms$
```

构建和测试容器镜像（生产镜像）- 1

- sudo docker build -t 10.100.198.200:5000/books-ms . //按books-ms目录中的Dockerfile构建生产镜像，把上一步的构建物打包到镜像中
- 详细阅读~/books-ms/Dockerfile 文件

上一步正常的输出结果如下

ubuntu@ip-172-31-30-243:~/books-ms$ sudo docker images //用此命令确认目标镜像是否生成

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
10.100.198.200:5000/books-ms	latest	c6543443c183	About a minute ago	708.4 MB
10.100.198.200:5000/books-ms-tests	latest	fc19ea46623b	30 minutes ago	2.229 GB
registry	2	8ff6a4aae657	6 weeks ago	171.5 MB
java	latest	264282a59a95	6 weeks ago	669.2 MB
debian	jessie	1b088884749b	6 weeks ago	125.1 MB
uifd/ui-for-docker	latest	c2b1b90a9e75	12 weeks ago	7.391 MB

构建和测试容器镜像（生产镜像）- 2

- 手工把书店服务跑起来，手工测试功能是否正常

sudo docker run -d --name books-db mongo

sudo docker run -d --name books-ms -p 8080:8080 --link books-db:db 10.100.198.200:5000/books-ms

sudo docker exec -it books-ms bash //进入容器内部

env | grep DB //查看容器内环境变量

exit //退出容器

sudo docker ps -a //查看服务器端和数据库容器

sudo docker logs books-ms //查看书店应用服务器端的输出日志

sudo docker rm -f books-ms books-db //删除手工测试容器

sudo docker-compose -f docker-compose-dev.yml up -d app //用docker-compose把书店项目启动起来

手工测试书店后台api功能

```
curl -H 'Content-Type: application/json' -X PUT -d \     //执行3次插入书籍的动作
'{"_id": 1,
"title": "My First Book",
"author": "John Doe",
"description": "Not a very good book"}' \
http://localhost:8080/api/v1/books | jq '.'
```

```
curl http://localhost:8080/api/v1/books | jq '.'   //查询所有书籍
```

```
curl http://localhost:8080/api/v1/books/_id/1 | jq '.'   //查询ID为1的书籍
```

推送测试镜像到镜像仓库

sudo docker push 10.100.198.200:5000/books-ms //只含服务器后台的jar

sudo docker push 10.100.198.200:5000/books-ms-tests //含所有测试工具

- 如果使用云主机，需要给eth0配置附加IP地址 10.100.198.200 ； 否则推送会失败

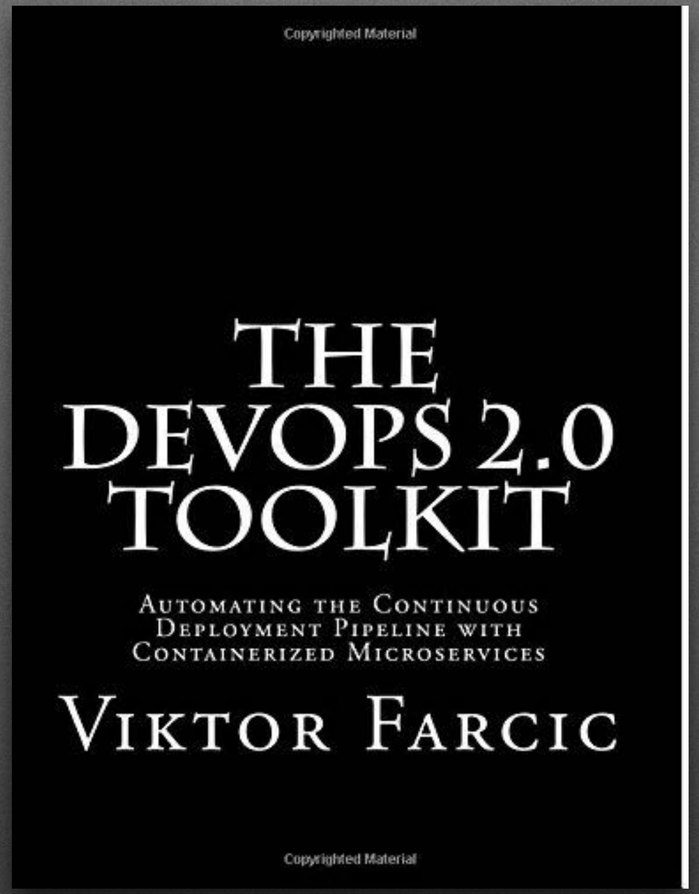
步骤检查清单

1. 检出代码 - Done
2. 运行预部署测试 - Done
3. 编译和打包代码 - Done
4. 构建容器 - Done
5. 推送测试镜像到镜像仓库 - Done
6. 部署容器到生产服务器
7. 集成容器
8. 运行集成后测试
9. 推送测试镜像到镜像仓库



“First they ignore you,
then they laugh at you,
then they fight you,
then you win.”

–Mahatma Gandhi



第五章 Docker世界里的配置管理









CFENGINE	PUPPET	CHEF	ANSIBLE
1993年创立，2008年第一个企业版产品发布的时候开源了	2005年发布第一个版本，同时有开源和企业版	2009年发布第一个版本，开源和商业版本并存	2012年发布第一个版本，开源和商业版本并存
由C语言编写，执行速度非常快，client-server架构	由Ruby语言编写，client-server架构	由Ruby和Erlang编写，client-server架构	由Python编写，无代理架构（用SSH直连）
需要有C语言技能才能掌握	需要掌握Puppet的配置语言语法或者Ruby DSL才能使用	需要掌握pure-Ruby和DSL来写系统配置的recipes	需要遵循YAML语法编写Playbook配置脚本
依赖程序员C语言开发能力强	操作员友好、Ops/SysAdmin能容易上手	程序员友好、对于Ops/SysAdin稍微有点难	学习门槛低、上手快

https://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software

总结

- 对于容器和不可变部署来说，Ansible是作者所推荐的配置管理工具
- 配置管理将会很快成为整个团队的责任，这个工具能更便于协作，从团队的角度看比其他配置工具的学习成本都低

配置生产环境

测试环境介绍

10.100.198.200



roles:
- common
- docker
- docker-compose
- docker-flow
- consul-template
- registry

10.100.198.201



roles:
- common
- docker

- 启动虚拟机前，确保你的机器有墙外的高速网络连接，个人或者是公司的VPN；或者选择墙外的云主机
- 读一下这几个Ansible的role文件
- 本章主要介绍Ansible的工作方式和基本功能，ansible帮我们完成了prod节点的初始化

启动测试环境

```
git clone https://github.com/vfarcic/ms-lifecycle.git

cd ms-lifecycle

vagrant up cd prod --provision //自动 cd 和 prod 两个虚拟机

vagrant ssh cd  //进入cd 虚拟机

ansible-playbook /vagrant/ansible/prod.yml \

    -i /vagrant/ansible/hosts/prod   //执行 prod 虚拟机初始化的操作
```

设置 Ansible Playbook

理解 prod.yml 文件 — 详情请读代码

```
- hosts: prod   //配置任务运行在 prod 的host上

  remote_user: vagrant  //使用vagrant用户登录prod节点，所有vm都有这个账号

  serial: 1

  sudo: yes

  roles:  //把prod节点配置上这两个role的功能，每个role都是一个子目录，里面放了需要执行的task

    - common

    - docker
```

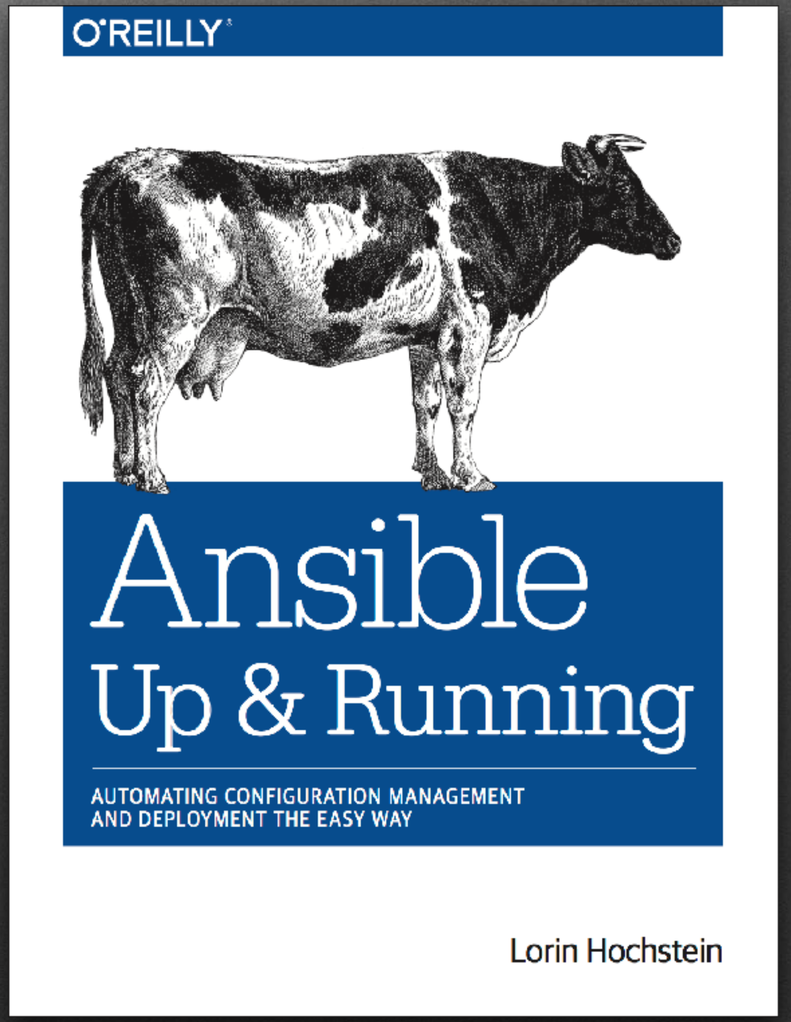
最后收尾操作

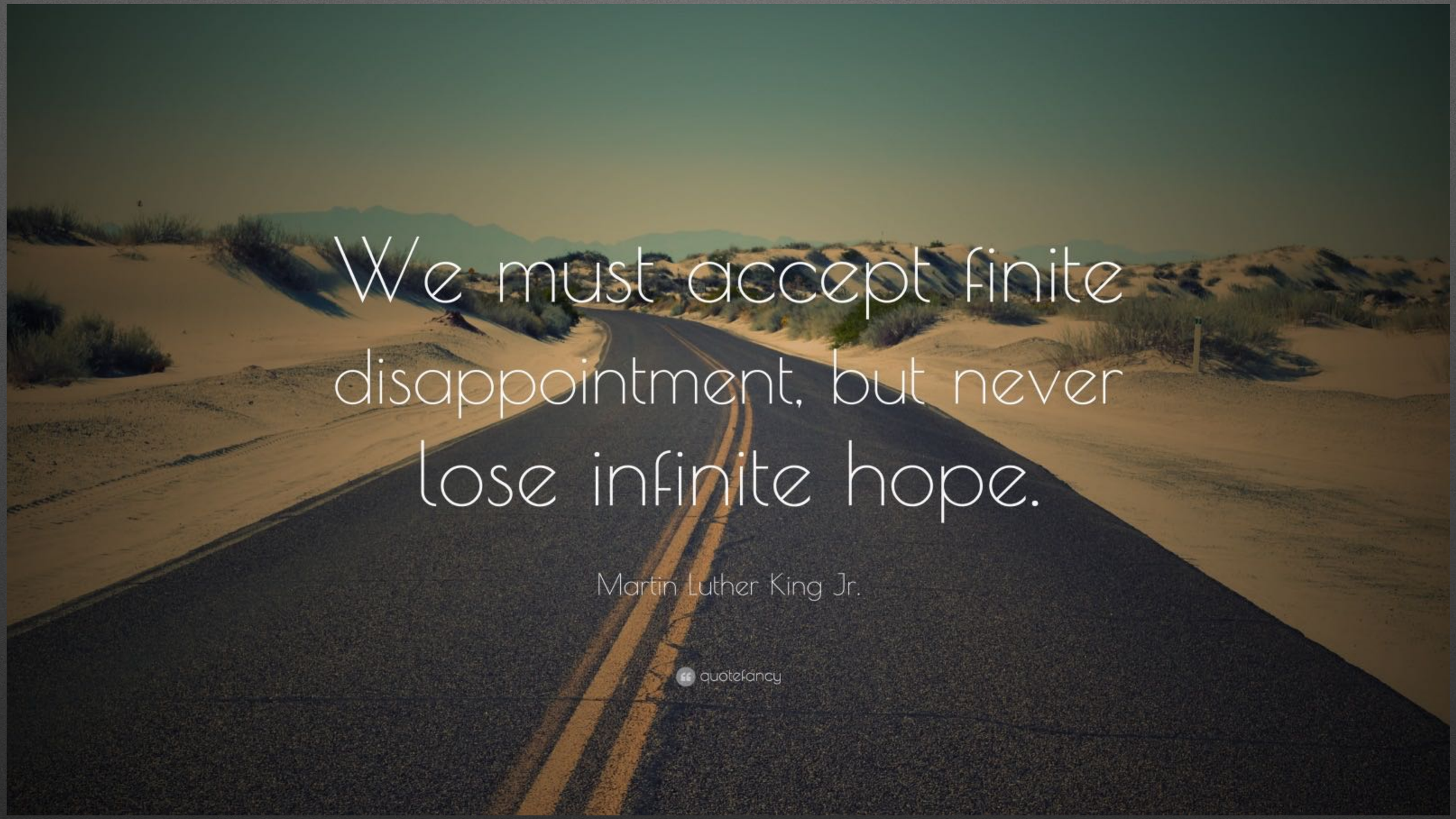
```
exit   //退出cd虚拟机

vagrant destroy -f   //删除所有刚刚创建的虚拟机
```

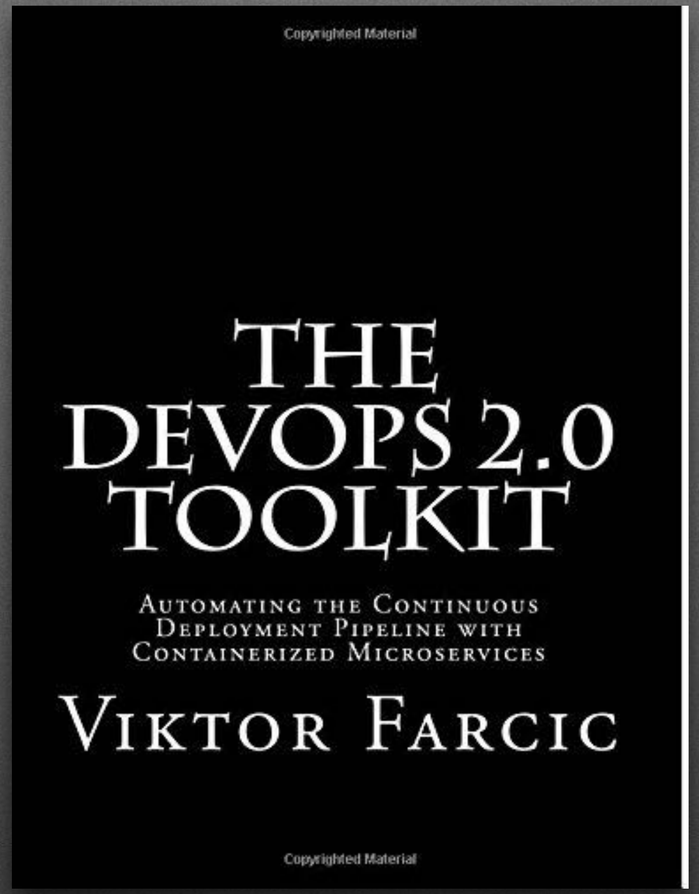
推荐 Ansible 书籍

- 这本书中文版已经出版《奔跑吧Ansible》
- 我正在学习这边书的英文版，电子版请进QQ群下载
- 感觉这是本不错的入门书籍
- 学习和重用他人的代码也是非常重要的，推荐去 https://galaxy.ansible.com/ 学习它的各种功能模块





Https://martinliu.cn DevOps教练 微信：MyDevOps



第七章 实施部署流水线：中间阶段

部署容器到生产服务器

测试环境介绍

10.100.198.200

cd

roles:
- common
- docker
- docker-compose
- docker-flow
- consul-template
- registry

10.100.198.201

prod
生产

roles:
- common
- docker

- 启动虚拟机前，确保你的机器有墙外的高速网络连接，个人或者是公司的VPN；或者选择墙外的云主机
- 读一下这几个Ansible的role文件
- 本章主要介绍在prod服务器上部署书店生产应用系统环境

启动测试环境

如果上一章您的实验场景已经全部删除了的话，您需要重新执行以下步骤，创建并初始化cd和prod虚拟机

git clone https://github.com/vfarcic/ms-lifecycle.git

cd ms-lifecycle

vagrant up cd prod --provision //自动 cd 和 prod 两个虚拟机

vagrant ssh cd //进入cd 虚拟机

ansible-playbook /vagrant/ansible/prod.yml \

    -i /vagrant/ansible/hosts/prod   //执行 prod 虚拟机初始化的操作

最后：执行 sh /vagrant/scripts/preload_cd.sh ； 这会构建出 ms-book镜像用于部署，并把它保存在cd的镜像仓库中

书店应用服务的部署compose 文件

下载位置：

wget https://raw.githubusercontent.com/vfarcic/books-ms/master/docker-compose.yml

docker-compse.yml 部分注释

```
base:
  image: 10.100.198.200:5000/books-ms  //这个镜像已经在cd的镜像库中
  ports:
    - 8080   //对外暴露8080端口
  environment:
    - SERVICE_NAME=books-ms
app: extends:
    service: base
  links:
  - db:db db:
    image: mongo  //会从docker hub上下载默认的MongoDB镜像
```

docker-compse.yml 说明

- 和之前使用的 docker-compose-dev.yml 文件不同，这里服务没有定义 8080:8080 这样的固定端口
- 生产环境中容器数量可能巨大，服务的部署需要考虑到潜在的端口冲突
- 使用随机端口，这里为下一章的服务发现，埋下伏笔

prod 服务器部署说明

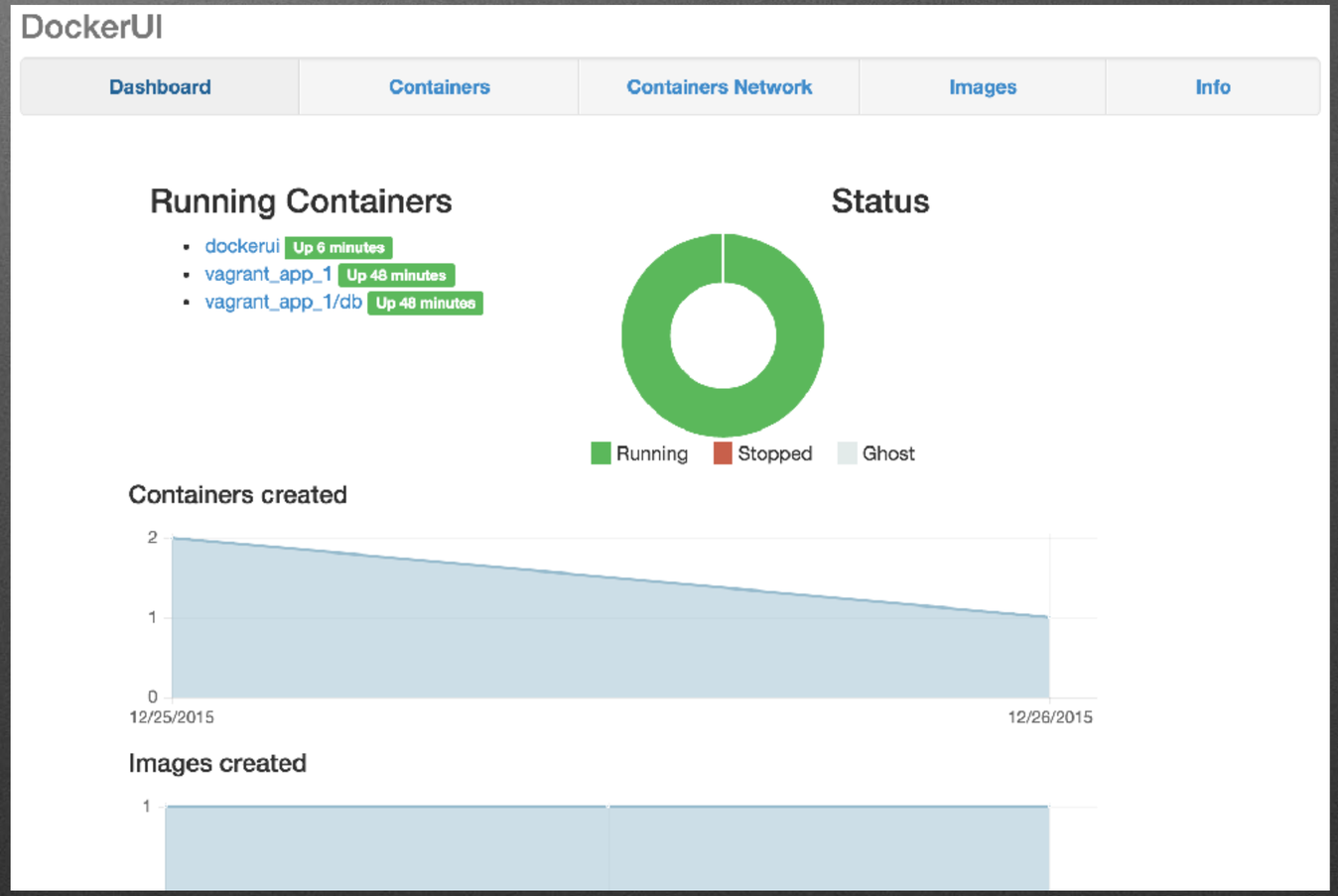
- /etc/default/docker 中的参数配置
- DOCKER_OPTS="$DOCKER_OPTS --insecure-registry 10.100.198.200:5000 -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock”
- 非安全镜像库地址，指向了cd 服务器的镜像库
- 接受来自所有服务器的docker api 指令调用

在 cd 服务器上执行的操作命令

- export DOCKER_HOST=tcp://prod:2375 //把docker命令调用指向prod服务器
- docker-compose up -d app //启动app应用服务
- docker-compose ps //查看服务的状态
- docker inspect vagrant_app_1 //找出书店服务的对外随机端口
- 用和curl命令测试书店服务是否正常工作

容器节点的图形化管理工具

- docker pull uifd/ui-for-docker
- 只有 7.391 MB
- 图形界面非常友好易懂



DockerUI

Dashboard | Containers | Containers Network | Images | Networks | Volumes | Info

Refresh

Running Containers

- ubuntu_app_1 Up About an hour
- ubuntu_db_1 Up About an hour
- registry-mirror Up 13 hours
- registry Up 13 hours
- dockerui Up 13 hours

Status



Containers created



Images created



Docker API Version: **1.23** UI Version: **v0.10.1-beta**

dockerui

Https://martinliu.cn DevOps教练 微信：MyDevOps

DockerUI

Dashboard | Containers | Containers Network | Images | Networks | Volumes | Info

Refresh

Containers:

Actions

Display All | Filter

Select	Name	Image	Command	Created	Status
☐	ubuntu_db_1	mongo	/entrypoint.sh mongod	Thu Jul 28 2016	Up About an hour
☐	ubuntu_app_1	10.100.198.200:5000/books-ms	/run.sh	Thu Jul 28 2016	Up About an hour
☐	registry	registry:2	/bin/registry serve /etc/docker/regis...	Wed Jul 27 2016	Up 13 hours
☐	registry-mirror	registry:2	/bin/registry serve /conf/config.yml	Wed Jul 27 2016	Up 13 hours
☐	dockerui	uifd/ui-for-docker	/dockerui	Wed Jul 27 2016	Up 13 hours

Docker API Version: **1.23** UI Version: **v0.10.1-beta**

dockerui

Https://martinliu.cn DevOps教练 微信：MyDevOps

DockerUI

Dashboard | Containers | Containers Network | Images | Networks | Volumes | Info

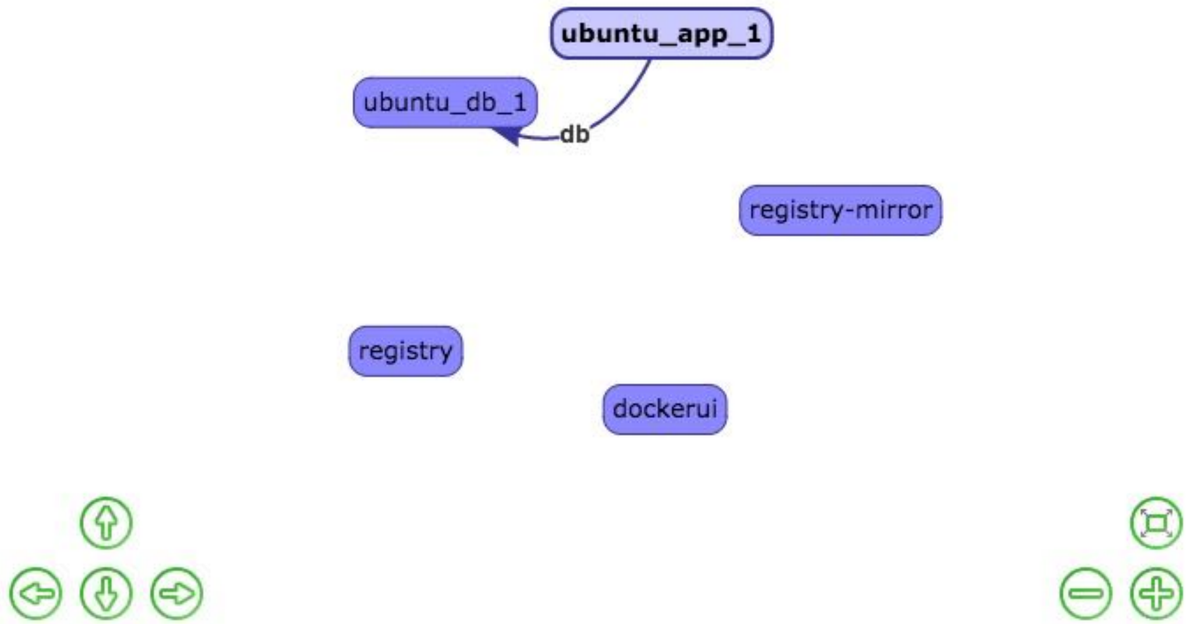
Refresh

Containers Network

Search

Hide Selected | Show Selected Downstream | Show Selected Upstream | Show All

Include stopped containers



Https://martinliu.cn DevOps教练 微信：MyDevOps

步骤检查清单

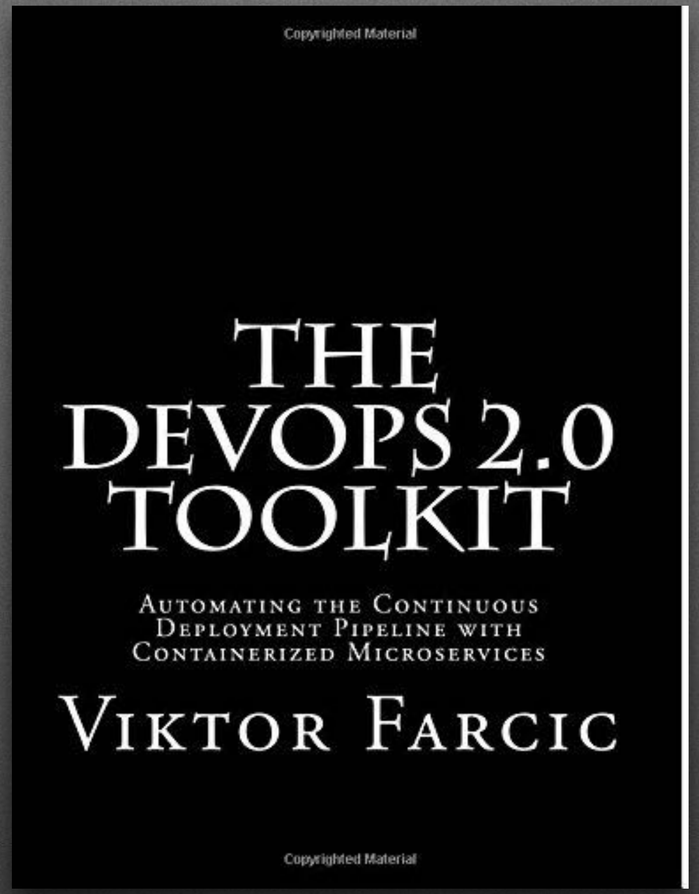
1. 检出代码 - Done
2. 运行预部署测试 - Done
3. 编译和打包代码 - Done
4. 构建容器 - Done
5. 推送测试镜像到镜像仓库 - Done
6. 部署容器到生产服务器 - Done
7. 集成容器
8. 运行集成后测试
9. 推送测试镜像到镜像仓库

总结

- 通过 ansible 远程地从 cd 服务器上初始化了 prod 服务器
- 使用 docker api 远程地，在 cd 服务器上执行 docker-compose up 命令在 prod 服务器上运行起了书店应用
- 用目前这种方式部署，应用是有downtime的
- 后面的章节会讲解使用蓝绿发布的方式消除downtime



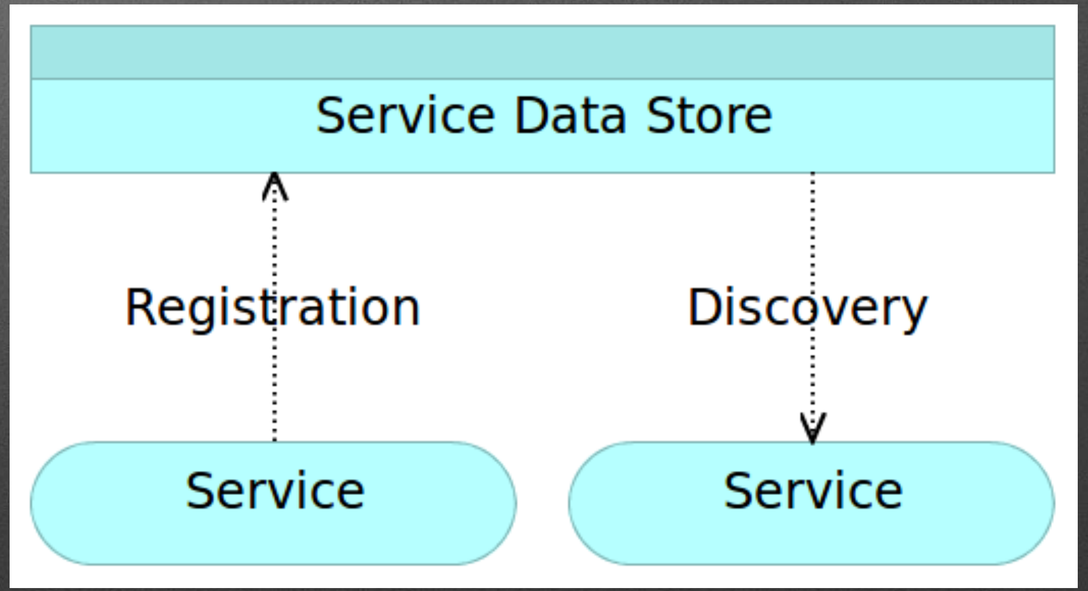
Https://martinliu.cn DevOps教练 微信：MyDevOps



第八章 服务发现：分布式服务的关键

服务注册和服务发现

- Service Data Store 服务数据存储
- 服务注册 Registration：至少存储正在运行的【服务端口+主机】信息的过程
- 服务发现 Discovery：让其它人/服务发现所存储的服务信息

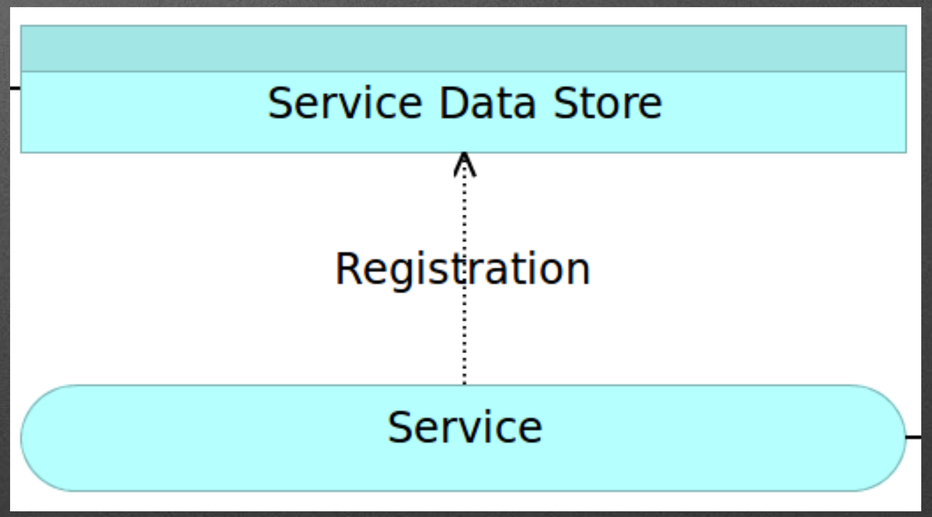


服务注册表-Service Registry

- 存储的服务信息（服务端口+主机+其它）是很轻量的数据
- 需要：快速、持久、容错等等
- 是一个简单够用的数据库
- 对外开放API接口，用于简单快速地数据访问

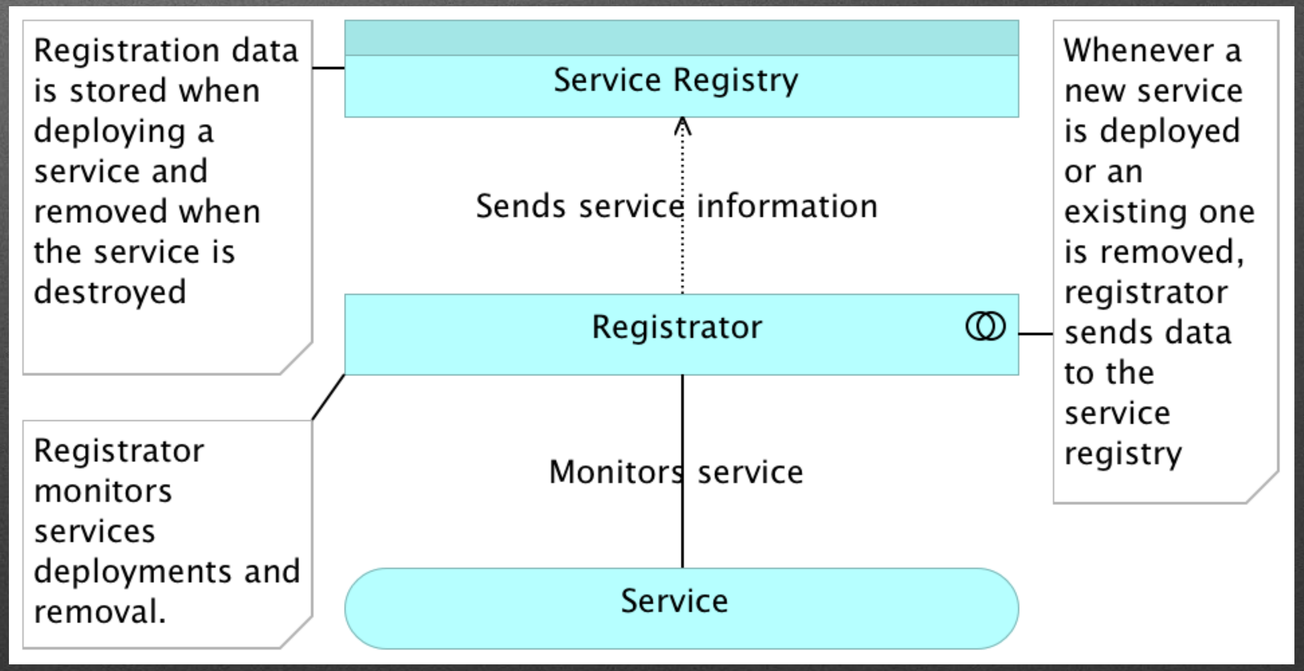
服务注册：自注册

- 在服务注册表中：当部署一个服务时存储一个信息；当服务销毁时服务信息数据被删除
- 由服务程序代码自己来管理注册和取消注册的过程。
- 特点：简单常见，但不是优化和有效率的方式



服务注册：服务注册器

- 在服务注册表和服务之间引入注册服务：注册器程序（容器）
- 在服务数据存储中：还是当部署一个服务时存储一个信息；服务销毁时数据删除。
- 注册器程序监视服务的部署和删除
- 当一个新的服务部署、旧服务删除时，注册器更就会新注册表中的数据
- 特点：优于自注册、更加可靠、不需要在服务代码中包含服务注册的代码(解耦)



服务发现

- 客户端的服务自发现
  - 每一个客户端/服务在需要访问其它服务的时候，就去注册表中寻找。
  - 客户端需要一对多的发送请求到不同的服务地址/ip端口
  - 仅仅用于内部服务的互动还可以
- 用代理服务
  - 服务的访问路径：http://[域名]/api/v1/books
  - 每个服务通过代理服务的一个弹性的地址被访问
  - 把服务消费者和注册表解耦

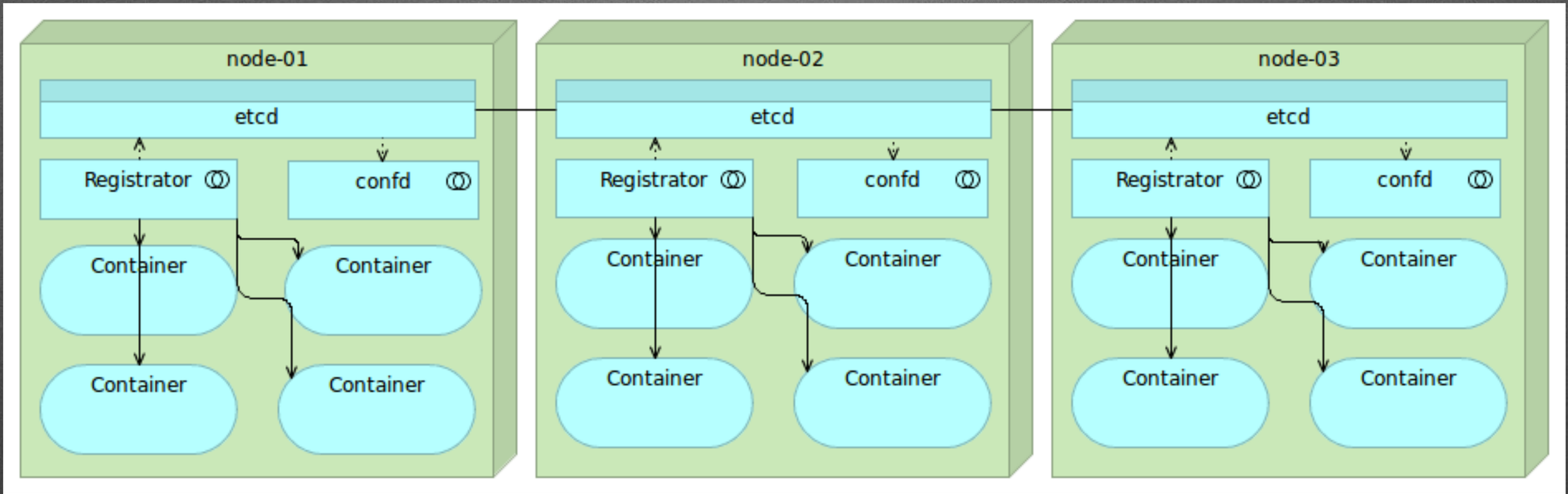
手工配置的服务发现

- 服务数量少的情况下可以使用手工的服务配置
- 不容易扩展，每次服务发生变化了，都需要重复执行一遍手工配置操作流程
- 无法处理工作负载不均匀、硬件故障等情形
- 都是静态的配置，可视性比较低；服务配置分散在各处，没有统一的配置管理
- 在服务动态变化的情况下，管理的复杂度和难度大，不推荐使用

Https://martinliu.cn DevOps教练 微信：MyDevOps

Zookeeper

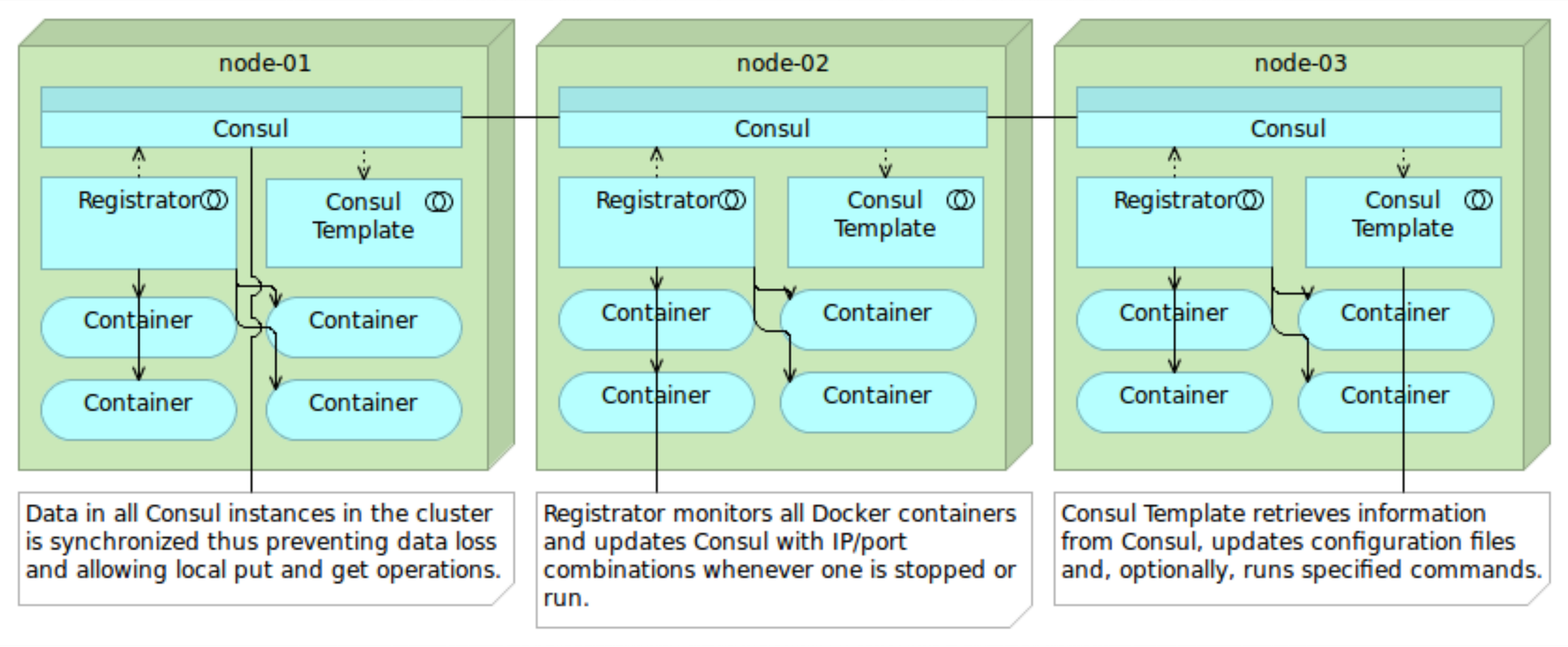
- 服务配置管理工具中最老的一种，起源于Hadoop的群集管理
- 对配置信息管理的形式类似于文件系统的形式
- 优势：成熟度高、强壮性好、功能丰富
- 劣势：复杂、维护成本、资源占用多，太老了



etcd + Registrator + confd 方案

Registrator通过/var/run/docker.sock监控所有容器运行的信息，把信息存储到etcd中，confd自动化地把etcd中的服务信息转换为配置文件，这些配置文件可以被代理服务器所加载使用

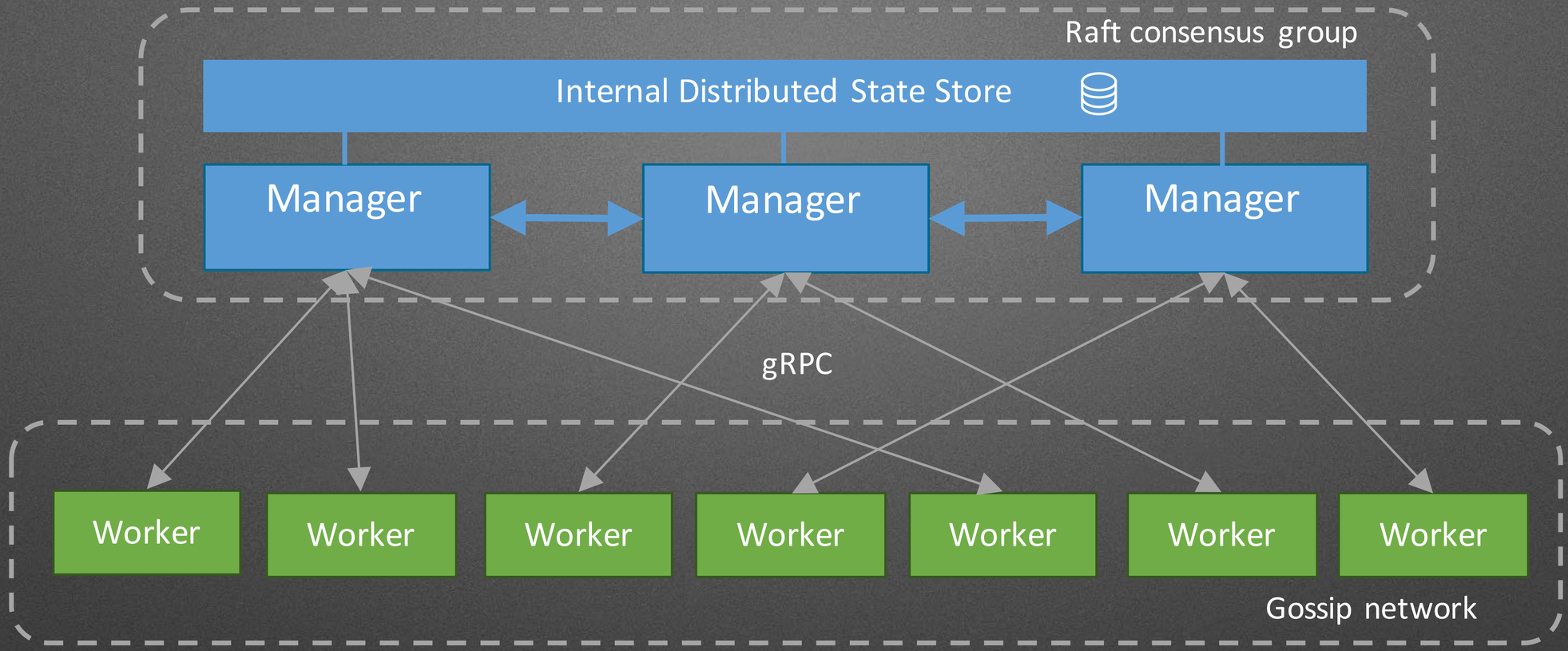
Https://martinliu.cn DevOps教练 微信：MyDevOps



Consul + Registrator 方案

Registrator还是同样地采集每个host上的容器的运行情况，把服务注册信息自动地发送到Consul中，Consul使用自有的配置模板服务，把服务注册信息更新到文件系统中，用于代理服务器的加载

Https://martinliu.cn DevOps教练 微信：MyDevOps



Docker 1.12+ 内置了服务注册、发现和路由功能

用Swarm编排模式工作，所有manager节点用于管理和存储服务的配置信息，用Raft方式存储和同步，能实现高可用，在任何节点上访问服务的端口，都可以被路由到该容器所在的主机上，内置负载均衡功能。

非本书内容，增加参考内容

服务发现工具的比较



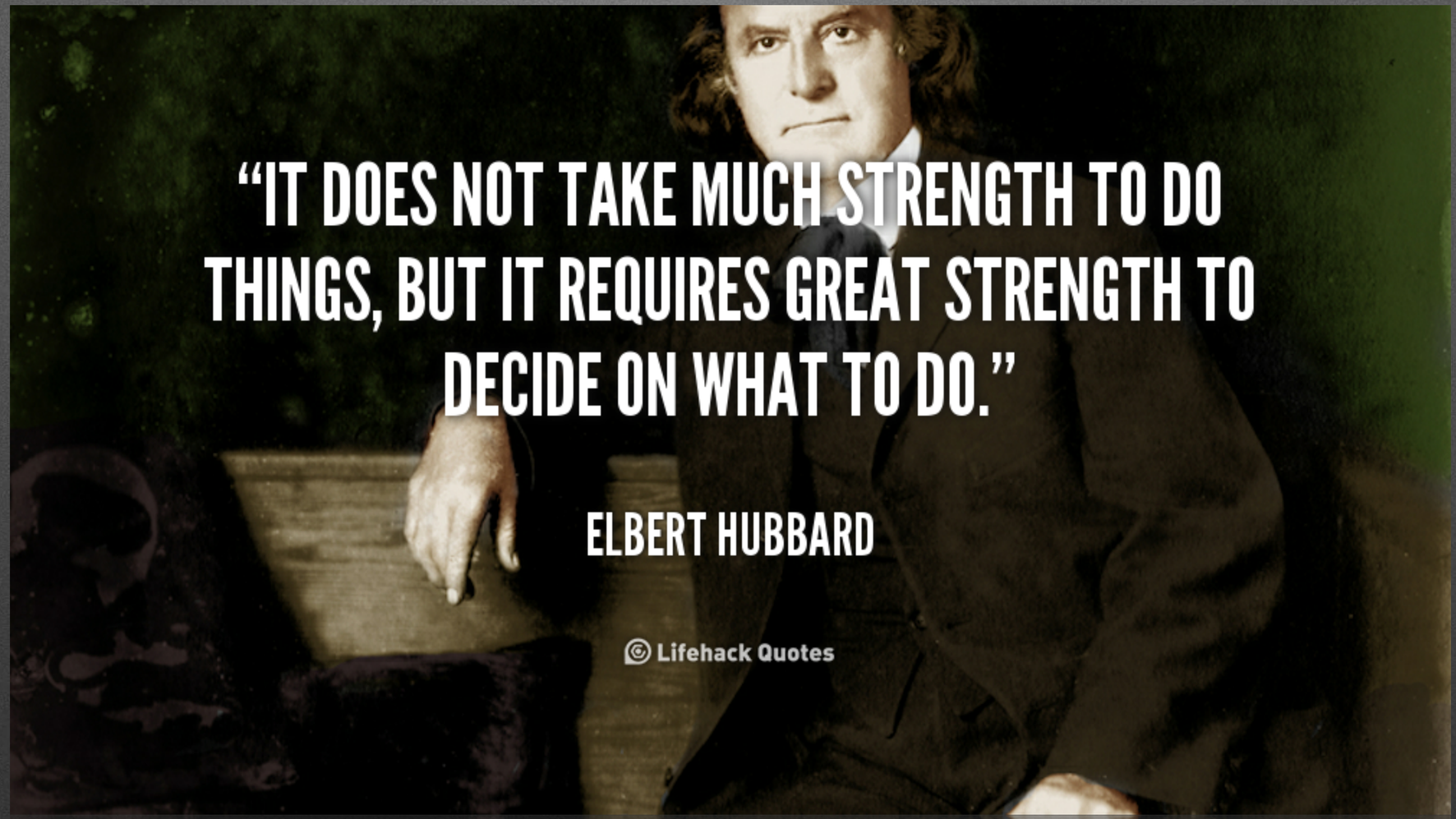
- 三个中最老的
- 复杂度、资源消耗最高
- 为那个年代设计的

etcd

- 简单、轻量、坚强大
- 能解决绝大多数实际需求
- 本身就是微服务设计，能自治

CONSUL

- 支持多数据中心、监控检查
- 功能丰富完整而平衡
- 使用gossip协议通讯，比etcd还容易配置
- 提供WebUI，模板功能



Https://martinliu.cn DevOps教练 微信：MyDevOps