基于 Docker 部署服务器的基础框架设计与实现

摘要 本项目完成了一个以 Docker 为基础的基本服务器集群框架，解决了服务访问控制、数据存储、应对弹性计算需求、服务配置冗杂、调用方式不统一、构建流程手动化等问题。基于 Docker 封装的程序、使用社区工具构建 Docker 小集群，整合多个开源项目构建 CI/CD 功能，构建一个基本的、可扩展的服务器集群框架，实现了管理与实例抽离、结构与服务抽离，取得了短时间创建实例并部署服务、代码从提交到测试的自动化及部署滚动更新的效果。

关键词 Docker；集群；持续集成；架构；DevOps

ABSTRACT The project completed a basic server cluster framework based on Docker, which solved the problems of service access control, data storage, response to elastic computing requirements, redundant service configuration, inconsistent calling methods, and manual construction process. Based on Docker-encapsulated programs, build Docker small clusters using community tools, integrate multiple open source projects to build CI/CD functions, build a basic, scalable server cluster framework, complete management and instance extraction, structure and service abstraction , achieved the effect of creating instances and deploying services for a short time, automating code from submission to testing, and deploying rolling updates.

KEYWORDS Docker; cluster; continuous integration; architecture; DevOps

目 录

基于 Docker 部署服务器的基础框架设计与实现

1. 前 言

1.1 背景

1. 容器的发展与目前情况

容器是对应用程序进行封装的一种技术，Docker 是一种当下流行的、由社区长期维护的容器技术。

Docker 是基于 Linux 内核机制 cGroups、NameSpace 和 SElinux 等实现资源隔离、边界隔离，使得程序如同运行在一个可控制的“容器”内，使用读写分离与镜像层的概念，在提供一致性开发环境与保证读写边界安全的前提下，为服务器程序调度提供了新的单位，与其编排工具 Docker-compose、调度工具 Docker-Swarm 组建 PaaS 平台[1]，近几年来发展迅速，成为运维管理的重要基础工具。

2. 微服务、运维自动化与 DevOps 的发展[2][3]

随着服务器规模的庞大、计算场景的复杂，传统的单个进程实现多个服务出现瓶颈，将多个应用拆分为微服务进行调度已成为优化必要。而容器技术在微服务浪潮中不免为优秀实践者，以进程为单位、隔离边界的原子性使得微服务调度更为简洁明朗。

也随着服务的增多，复杂性剧增的同时也引发了人工成本的增多，重构服务器结构、宕机人工干涉等情况，服务器运营维护，监控、报警、策略处理、升级优化、滚动更新等自动化流程被提上议事日程。

同时也随着开发与运维部门之间需要有越来越多的线上线下配合，运维开发交界处也发起了 DevOps（Development 和 Operations 的组合词）运动，致力于构建开发运维工作的桥梁，推动交流，开发参与运维，运维同时参与开发。

3. 开源项目与社区的壮大

开源社区的健壮性、活跃性推动计算机编码行业飞速发展，开源社群的发达，越

来越多的社会团体也用上了自由软件（此处为“自由”，非“免费”、“开源”），通过引擎自建网站、部署服务成为可能。开放源代码保证代码安全可审查的情况下，较为大型的项目同时被多个社会团体所使用产生利益支撑，在基金会的保护下，项目也维持着较高效的维护频率，使得有足够的力量支撑着开源项目的可靠性。

1.2 研究现状

1. 超大规模部署框架的研究现状

Kubernetes（简称 K8s）源自谷歌内部的超大规模服务调度框架 Borg，在 2014 年对外开放源代码，支持包括不限于 Docker 在内的容器的超大规模编排、调度，解决运行环境一致性、编排调度监控、滚动更新等问题。目前国内大型企业对于超大规模部署框架的优秀实践与研究多数来源于 K8s[4]，在谷歌的带头下，K8s 以其本身高可用性与成功实践，推动着超大规模部署框架理论的快速发展。

2. CI/CD 的研究现状[5]

持续集成与部署（Continuous Integration and Continuous Delivery，以下简称 CI/CD）是软件工程领域概念，旨在推动软件在短时间内（甚至是一小时）承受多次更新并部署，以应对“持续性交付”，与其带来的持续部署需求，配合单元测试，可以构建出一套完整的从提交、构建、测试、上线的流程。CI/CD 的理论研究已经相当成熟，随着运维自动化的提出，基于此的相关工具在近几年也都如雨后春笋般出现。

3. 小规模服务框架的研究现状

基于部署框架、CI/CD 工具出现，大型企业取而付诸实践，以财力支撑足以降低部署产生的边际成本，而针对仅有小规模服务部署需求，所提供的项目还不是非常多。一方面是相关项目的高度模组化，使得中小规模服务部署无需遵循某一套方案，有一定的自由度；另一方面是不少小规模服务部署不选择直接部署服务器在 IaaS 上，而使用 PaaS（平台即服务，platform as a service）或 SaaS（软件即服务，Software as a Service）。

1.3 论文结构

1. 一般问题与需求分析：针对传统模式存在与即将面临的问题进行分析，归纳需求与解决思路，明确需求边界，对应到业务场景，并对方案进行技术可行性与经济可行性上的分析。

2. 架构设计阐述：阐述了整个框架的组合结构、连通关系以及业务逻辑。

3. 技术选型：阐述了在架构中所使用的具体技术及类似技术的比较。

4. 方案示例：为该架构提供了四套方案。

5. 迁移示例：提供了一些从裸机服务打包、迁移到框架的网络服务的示例，帮助用户切换、接入到该框架中。

6. 测试与比较：进行了一系列的比较与测试。

7. 框架扩展与相关问题讨论：提供框架的后续扩展可能性，并讨论集群部署一些可预见的问题及解决方案。

8. 总结。

2. 一般问题与需求分析

2.1 一般问题

1. 服务的访问控制、数据存储

服务程序人为设计或被动地访问文件是系统的潜在威胁因素。传统方法选用了用户分组与系统审计软件，但也引起了用户组管理、审计权限管理上更多的复杂因素。

Linux 系统在设计之初也规定了目录的基本工作（如服务程序文件通常在/var/目录存放）现实使用上很难规范，导致了管理员在排查、迁移、清理服务程序时的许多麻烦。

2. 应对弹性计算需求的调整

单一的机器在面对流量冲击、计算任务分发等场景，往往需要短时间内聚合大规模的计算力以应对需求。传统服务模式的扩展性不够强大，除了要解决系统安装、环境配置的问题，还要解决服务测试到上线等一系列问题。遇见异构平台的情况下，还需要提前做好应急方案与测试，在扩容上有大量的工作需要。服务架构的弹性计算能力与环境统一化成为应对场景的重要问题。

3. 服务配置文件冗杂、调用方式不统一

关于服务数据存储位置，配置文件是启动服务的关键文件，配置文件存放位置不统一，存储管理不统一，版本管理不到位，给检查、同步、服务迁移都带来了麻烦。

不同的服务程序之间，配置文件的管理与程序调用方式的不统一，使得服务程序的调度有比较大的困难。

4. 代码测试、构建、部署的手动化

服务程序内容的更新，通常需要经过一系列的测试方才可更新上线，“测试驱动开发”使得自动化的测试流程成为开发程序的一个标准。程序规模一旦变大，测试与

开发的分离，同步不统一、测试工作量大等问题出现，因开发、测试、部署的环境不统一引发的问题也出现，配置更新大部分时候需要重启服务程序（或其他步骤）而手动化重启或脚本重启很多时候也需要人工干涉排查各种不可预见问题。

2.2 需求分析与解决思路

1. 程序边界实现可观可控，确保数据安全性[6]

应对服务程序的访问控制与数据存储问题，需要可观可控的边界管理，来确保数据安全与程序管理。一种思路是做到文件访问位置可控，对程序本身进行的访问进行全部监控，另一种思路是将程序在虚拟环境中运行，根据管理员需要只选择映射可以访问的地址，例如只映射一个数据库的文件地址或配置文件的地址给虚拟环境。

2. 环境一致的多云结构，支持性能的弹性伸缩[7]

应对异构机器、多种云的计算力集中，需要提供统一的计算环境以方便服务迁移部署，同时集群的部署也需要可以弹性调整。目前较为完善的思路是使用统一的虚拟机环境，在不同平台机器上构建相同的基础环境，但折损的性能无法忽略不计。

应对弹性调整，通过物理层面进行配置调整以应对性能伸缩过于繁琐且不安全，而基于虚拟机层面调整性能上也不尽人意（如从关机到扩容到重启服务，部分虚拟机需要 5 到 15 分钟不等）。需要快速接入使用，且可以在上面提供统一的当前运行环境，并快速部署服务以达到扩容的目的。

3. 源码与配置文件集中化、服务调用统一化

面对配置文件管理问题，需要配置文件集中化管理，接入版本管理工具。而应对服务调用方式的不统一，需要有一套非系统层面的方式来统一规定服务调用的行为，描述服务之间的关系（比如前置、择优策略等）。同时可以暴露接口使用更高层的工具进行管理，以确保其调度的扩展性。

4. 构建集成流程、部署流程

面对手动化流程，需要有工具可以实现环境统一测试、自动化构建并测试与分析报告，在通过测试的情况下触发部署。传统是提供虚拟机实例，并使用脚本等工具来

构建统一环境，消耗性能较大，测试方面也是使用脚本进行管理。脚本规范化因服务不同很难实现统一，另一种思路便是接入现在愈发成为主流的“持续集成与持续部署”。

2.3 需求边界

1. 方案应当解决代码与数据的统一管理问题，不解决在使用过程中用户的版本管理、转移、备份、私密数据存储。

2. 方案应当解决多台机器之间连通、管理的问题，不解决在使用过程中跨机房、跨可用区及跨服务提供商之间通讯与延迟产生的问题。

3. 方案应当解决环境一致性与滚动更新问题，不解决在使用过程中因用户编写错误导致的无法运行。

2.4 应用场景

1. 个人用户：包括个人网站非托管运营、文件代理下载、数据托管等。

2. 中小企业：内网组建、内部代码与测试流程自动化、访问控制与审计。

3. 异构集群：统一环境与计算力整合、大量任务分发、弹性伸缩。

2.5 技术可行性

1. 容器技术规定程序边界、数据存储[8]

Docker 使用 cgroups 与 namespace 等技术完成虚拟环境的架设，将程序封装成静态的镜像层，在调用的时候指定访问控制。数据卷可以将原本分散的数据存储，根据自己的需要定义集中存储。

而程序在虚拟环境中的行为，不会影响到外部的状态，有效地提高了数据安全性。Docker 灵活的文件映射机制为集中化管理配置提供了便利。

2. Dockerfile 与 Docker-compose

Dockerfile 可以将程序打包成 Docker 镜像，完成对服务程序行为的封装，从而

实现程序调用的统一化。而 Docker-compose 可以记录 Docker 运行配置的参数以及彼此之间的连接关系，在 Docker-Swarm 中可以使用其作为统一的微服务调度单位，从而实现两个层级的服务调度统一化。

3. Docker-machine 构建统一执行环境

Docker-machine 是提供统一 Docker 环境的底层服务程序，在非 Linux 的环境下，Docker-machine 将会安装基于 KVM 的一台 Docker 机器以构建 Docker 环境，并可以将多台装有 Docker-machine 的机器统一连接到某一台核心机器上进行管理。

4. Docker-Swarm 实现伸缩、自动部署[9]

应对单一机器扩展的需求，Docker-Swarm 技术支持基于 Docker 环境的集群构建，以 Docker 为单位构建微服务，将任务分发到多台机器上实现弹性伸缩。而基于相同 Docker 的统一的应急方案，远比应对不同服务准备多个不同的应急方案要方便得多。

5. 版本管理与 CI/CD 工具

git 的分支管理与版本管理，为统一的文件管理提供了必要的功能保障。“持续集成与持续部署”，狭义上它提供了针对服务从源代码到上线的自动化解决方案，在 Docker 提供统一环境的基础上，预先设计好测试条件。同时用统一的环境进行构建，排除了由于多端环境引发的不统一问题。

2.6. 经济可行性

1. 自由软件

经济上自由软件开放免费使用，低门槛确保使用人群，以驱动开发者进行维护。通常大型企业会参与投资到相关的软件基金会，以确保这些自由软件项目得以充分维护，对于中小企业与个人来说，相对于购买传统软件公司提供的解决方案，自由软件的经济成本较低。

2. 人工成本

一方面，由于自由软件需要团队有更专业的人参与到相关领域，专业人才的人工

成本会相对提高一些。

另一方面，框架本身有不低的入门门槛，提供一定的自动化环境，在迁移、构建新环境方面要花费较多的人工，但在后期的维护上（即长远的花费上）可以减少不必要的人工干涉，进而降低维护、扩展方面的成本，提升团队效率。

3. 计算资源

框架支持异构计算意味着可以互为连通的机器便可加入计算集群提供计算资源，使得平时可能无法参与的计算资源也可以加入到其中，降低了计算成本。

另一方面，IaaS 平台从部署速度、底层网络管理、内网管理等，性价比远比自建裸机运行要高。相同配置的物理服务器与其相关托管费用、人工维护成本，与等配 IaaS 平台提供的服务器，成本高接近 50%或以上，性能损失却几乎可以忽略不计，IaaS 也帮用户完成了网络配置、申请公网 IP 等繁琐工作。

IaaS 价格情况：

1. 境内外平台价格，（2019 年 4 月 27 日份数据，同架构，同 CPU）同样的价格，低配服务器（如 1c2g），境外比境内便宜 20%~40%不等，高配服务器（如 16c32g）则价格接近。

2. 而境内不少机器带宽单独收费，成本较大，（2019 年 4 月 27 日份数据），境外大部分机房提供免费无限速带宽（流量配额限制），境内则按流量计费（0.5~1 元每 G 不等）或按带宽计费，费用较为昂贵。

2.7 总结

1. IaaS 是小规模网络服务的首选，经济可行且门槛低。

2. 对程序进行访问控制、数据文件管理，配置文件实现集中管理，能够使整个服务架构更为安全简洁可控。

3. 提供统一的运行环境，是集群弹性伸缩、集成与部署流程的必要基础。

4. 重新调整服务调度单位的颗粒化，使得整个服务器的调度灵活可控。

5. Docker 化的趋势可以应对越来越多的场景，CI/CD 的趋势也将对服务部署有所革新。

3. 架构设计阐述

3.1 基础设施与管理单位

1. 阐述

基础设施包括公有云、私有云、裸机服务器、虚拟机等提供操作系统支持的设施，确保可以互相访问。

框架使用 Docker 镜像作为基础的管理单位，将每一个单独的服务组件打包成 Docker 镜像（如 nginx、redis-master），再将不同需要的 Docker 镜像打包成服务（services）。在服务管理上以单个 Docker 作为管理单位，控制其启动、停止、查看日志等操作。在服务部署上，使用 services 对一套单独的结构的服务进行逻辑上的同时部署。

2. 具体部署

确保基础设施可长时间提供服务，且可以互相访问，构建基础的计算设施。

优先使用公有 registry（如 Dockerhub）上的镜像，套用自身配置文件并编排成 services。同时也可以为自己构建的服务撰写 Dockerfile 进行 Docker 封装，上传到公有或自建的 registry 上。

3.2 主机系统管理

1. 阐述

构建一台独立于业务集群的单独机器，完成对于即将部署的业务集群所有机器的管理。将底层的系统管理与实际的业务部署分开，有利于管理上的分层，细化管理边界。

需要为集群构建统一 Docker 环境，进而使得集群可以根据需要快速部署、伸缩。同时需要有集群的管理节点。

2.具体部署

构建一台专门用于内部管理与访问的堡垒服务器，接管所有即将部署的机器。

使用既定工具完成对所有主机的 Docker 环境统一安装，并将主机注册到管理节点上。

3.3 CI/CD 部分

1. 阐述

持续集成（CI）在源代码变更后自动检测、拉取、构建和（在大多数情况下）进行单元测试的过程，根据所定义的自动化过程，将结果执行信息反馈、构建输出或系统调用等多项功能操作。

持续交付（CD）通常是指整个流程，自动监测源代码变更并通过构建、测试、打包和相关操作运行它们以生成可部署的版本，基本上没有任何人为干预。

构建一台提供代码托管服务、CI/CD 服务、可部署文件托管服务的机器，用于管理团队代码、内部测试、程序打包等流程，完成将 Docker 镜像作为基础管理单位的工作。

2. 具体部署

git 代码托管服务。

用于链接到 git 的 CI 服务。

用于存储 CI 所生成的可部署文件的服务。

CI/CD 部分应当与集群部分分开，构建独立的可运行环境。

3.4 文件系统

1. 阐述

集群需要一套独立于管理节点的文件系统以确保管理节点的文件安全性，文件系统需要有能够应对集群的特性，诸如 NFS（Network File System）单节点方案、HDFS（Hadoop Distributed File System）分布式方案[10]。

基于文件系统，需要通过一定机制授权给集群内的所有机器，包括可以有应对读写权限的规范、读写的审计等需求的特性。

2. 具体部署

在小规模的节点文件系统上，使用一个专门的节点上构建具有访问控制功能的文件系统，将所有集群只读授权访问到该文件系统，并授权给集群管理节点写权限。

在较中等规模构建多个文件系统，合理分配使用读写分离控制来完成负载。

3.5 集群

1. 阐述

以 Docker 为服务基本单位，以多个 Docker 编排为 Services 为服务调度单位，需要底层提供统一的 Docker 运行时环境

manager 节点需要有管理的职能，包括发布任务、管理任务、完成写操作等。

worker 节点需要能根据 manager 节点分发的任务，完成任务并反馈，或持续保持服务状态。

2.具体部署

使用统一的环境构建工具构建 Docker 环境。

申请 manager 节点并纳入管理节点，由 manager 节点创建 Docker 逻辑集群。

申请 worker 节点并加入到 manager 节点所提供的 Docker 逻辑集群之中，开始工作。

4. 技术选型

4.1 基础设施

1. 计算资源

泛指公有云、私有云、裸机服务器、虚拟机等提供计算机操作系统支持的硬件、软件设施,此处选用 IaaS 基础服务,如境内大厂阿里云、青云,境外大厂如 aws、linode、digitalocean。受限于一些历史原因与社会环境因素，国内的部分线路网速的性价比不是很理想，会因不同的方案配置，影响整个集群效率，举下文“默认配置”为例：

表 4.1 默认配置方案中部署区域及可能出现影响情况

管理与数据节点	文件系统与集群	可能出现的影响情况
境内	境内	同可用区内网问题较少，不同可用区容易因为用网高峰导致连通不及时、网速较慢等
境内	境外	集群拉取镜像时候可能不够及时
境外	境外	管理节点有略微延迟
境外	境内	集群拉取镜像时候可能不够及时

同时所考虑的因素较多，诸如可执行文件的大小、用网时段、机房状况、具体的业务对网络状况的需求等等。

基础设施技术选型上，需要参考各方面的口碑、使用习惯等，根据业务需要选择稳定、有网速保障的服务商，大部分 IaaS 平台提供类似的操作接口，学习成本较低。

2. 操作系统：Ubuntu[11]

由于 Docker 技术深度依赖于 Linux 内核，优先选择 Linux 内核的发行版，有利于避免产生额外运算成本(在Windows平台上部署需要基于KVM虚拟机)。在多个Linux

发行版系统中，较为推荐 Ubuntu 但并不排斥其他发行版，理由如下：

1. 以“人性化”作为系统设计理念，在系统管理上体现。

2. 完善的包维护频度与维护机制，为系统稳定性提供保障。

3. 相对于 RHEL 系列的独家支持，Ubuntu 依赖社群，问题反馈与解决及时。

同时，因有 Docker-machine 的加持，需要权衡好操作系统的熟悉程度、性能损耗等多方因素做出选择。此处默认使用 Ubuntu。

4.2 主机管理

1. 统一环境与管理工具：Docker-machine

工具需要提供统一的集中管理注册功能，即可以在单一一台机器上操作，便对其余机器进行操作，可以使用 ssh-copy 等工具完成同步与访问认证。Docker-machine 支持多种操作系统构建统一的 Docker 环境，同时支持 ssh 访问机制，提供较为稳定的同一环境与管理服务。

4.3 CI/CD 系统

CI/CD 系统具有较大的扩展选择，可较为灵活使用。[12]

1. 代码托管服务：Github、Gitea

代码版本管理使用 git 以获取最佳 CI 服务体验。基于 git 的代码托管服务可选择自建社群维护的 gitea 项目。或者使用 Github 作为代码托管。具体的选择使用需要根据对代码的隐私性要求、项目大小、通讯速度等多方面进行选择。

2. 持续集成服务：Jenkins、Drone CI、wercker

持续集成服务将完成集群自动化中重要的一环。功能较为综合的 CI 有 Jenkins，功能齐全包括不限于 Docker 为底层形式的 CI，自定义程度高，上手略有难度；Drone CI 属于社群维护的一套针对 Docker 工作流的 CI 服务方案，简单上手，可以应对大部分项目的 CI 工作。同时亦可使用第三方服务 wercker 等，上手难度适中。

3. 镜像存储：Docker-registry、Dockerhub

因为Docker部署依赖于镜像，镜像来自项目的Dockerfile生成。通常会将生成好的 Docker 镜像保存在一个固定站点，任由服务需要进行拉取。默认会使用Dockerhub官方提供的服务，或自建Docker-registry。

4.4 文件系统

1. 网络文件系统：包括不限于NFS

为确保框架的可扩展性，选择了允许网络调用的网络文件系统，诸如NFS、HDFS等。根据业务的不同，文件系统的选择、部署方案也会有较大差异，例如单点单写多读、单点分离读写等。这里使用基础的NFS单点单写多读作为默认方案。

4.5 集群与部署

1. 容器编排：Docker-compose

Docker-compose是一项用于编排Docker服务逻辑的技术，使用yml格式定义编排内容批量部署Docker镜像，以减少手动干涉的工作量。同时Docker-compose不仅适用于Docker基础环境的服务部署，也适用于Docker-Swarm集群中的服务部署。

2. 多节点部署：Docker-Swarm、Kubernetes[13]

Docker-Swarm用于将多台提供Docker环境的服务器联合组成一个Docker工作集群，以多个 Docker 一组，组成 services 基于文件系统完成不同的调度编排。Docker-Swarm分为manager节点与worker节点，由manager节点分发任务，由worker执行。这里使用受Docker官方支持Docker-Swarm。

5. 方案示例

5.1 最小配置方案

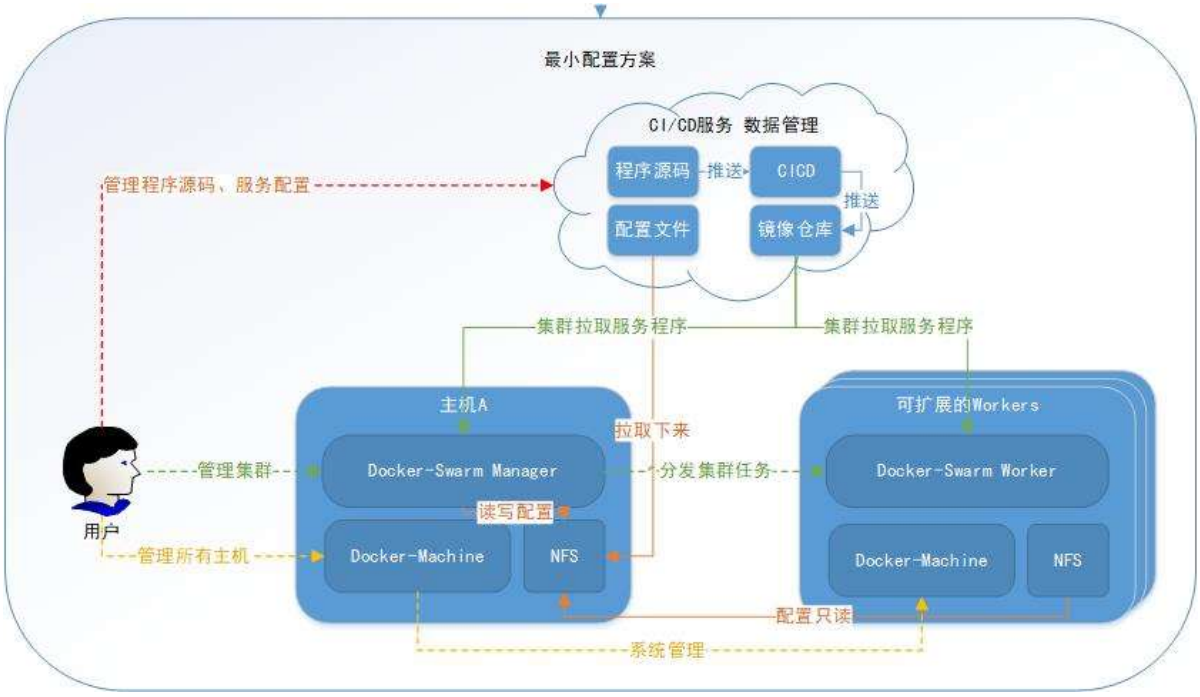

图 5.1 最小配置方案示意图

1. 基础设施

一台主机 A、workers。

2. 系统管理

使用 Docker-machine 提供统一 Docker 环境，主机 A 作为 Master 节点。

3. CI/CD 服务与数据管理

CI 与数据管理方面全部交给了第三方应用服务。较为经典可使用 Github+TravisCI+Dockerhub。

4. 文件系统

在主机 A 上构建 NFS，设置为对外访问只读，专门用于集群部署跟状态服务所需要的文件空间，与主机 A 原本空间分隔开。主机 A 本身对其具有写权限，对外访问为读权限。这一部分亦可使用第三方如 aws、aliyun 提供的文件系统。

5. 集群管理与扩展

在主机 A 上部署 Docker-Swarm 集群管理（Manager）节点，与 Master 统一为同一个节点，在单独的节点上实现机器与集群的统一管理。

扩展一台新的内网 worker 节点，Master（主机 A）主动访问将其并入 Docker-machine 管理，登录 worker 将其加入到 Docker-Swarm 集群的 Manager（主机 A）中，挂载自身（主机 A）NFS，在 Manager（主机 A）调整集群任务，完成扩展。

5.2 默认配置方案

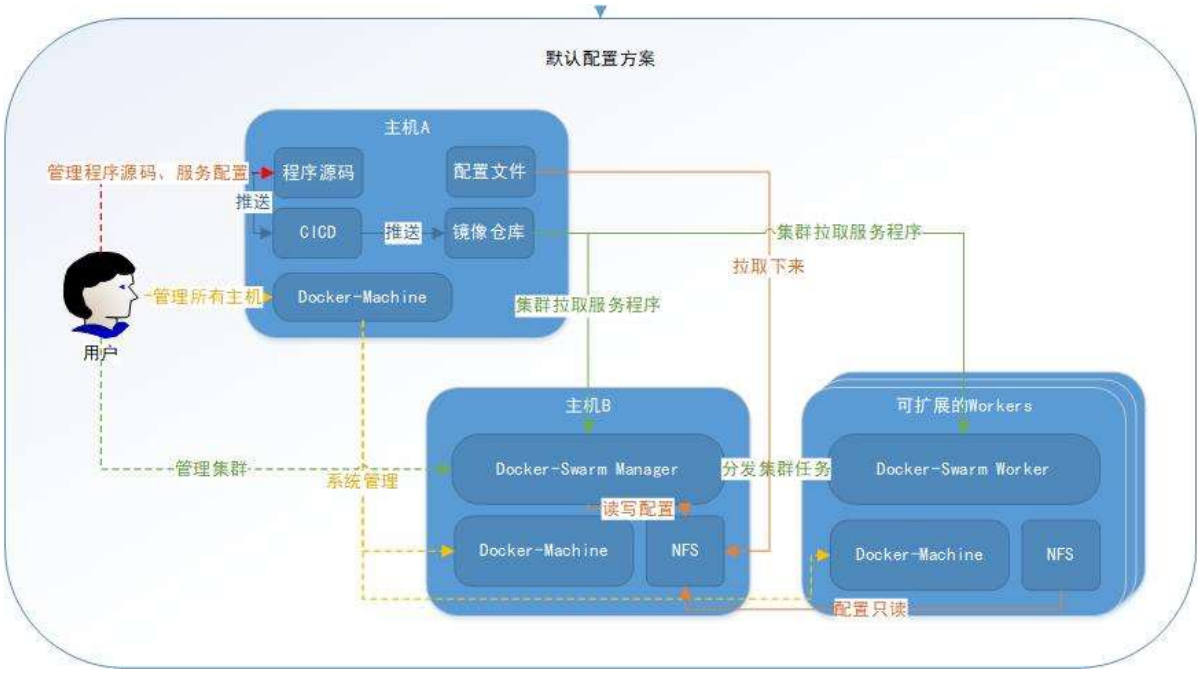

图 5.2 默认配置方案示意图

1. 基础设施

一台主机 A、一台主机 B、workers。

2. 系统管理

使用 Docker-machine 为两台机器提供统一 Docker 环境，主机 A 将作为 Master 节点管理其他主机，将主机 B 收纳管理。

3. CI/CD 服务与数据管理

在主机 A（Master）上使用自建的 gitea、drone、Docker-registry 作为 CI/CD 系统。CI 种 drone 的计算力由集群与自身提供。

4. 文件系统

在主机 B 上构建 NFS，设置对外访问为只读，专门用于集群部署跟状态服务所需要的文件空间，与主机 B 原本空间分隔开。主机 B 本身对其具有写权限，对外访问为读权限。

5. 集群管理与扩展

在主机 B 上部署 Docker-Swarm 集群管理（Manager）节点，与 Master 分离为两个不同节点，通过拉取主机 A 上的镜像进行部署。

扩展一台新的内网（主机 B 所在内网）worker 节点，Master（主机 A）主动访问将其并入 Docker-machine 管理，登录 worker 将其加入到 Docker-Swarm 集群的 Manager（主机 B）中，集群挂载自身（主机 B）NFS 只读文件系统，在 Manager（主机 B）调整集群任务，完成扩展。

5.3 小规模集群配置方案

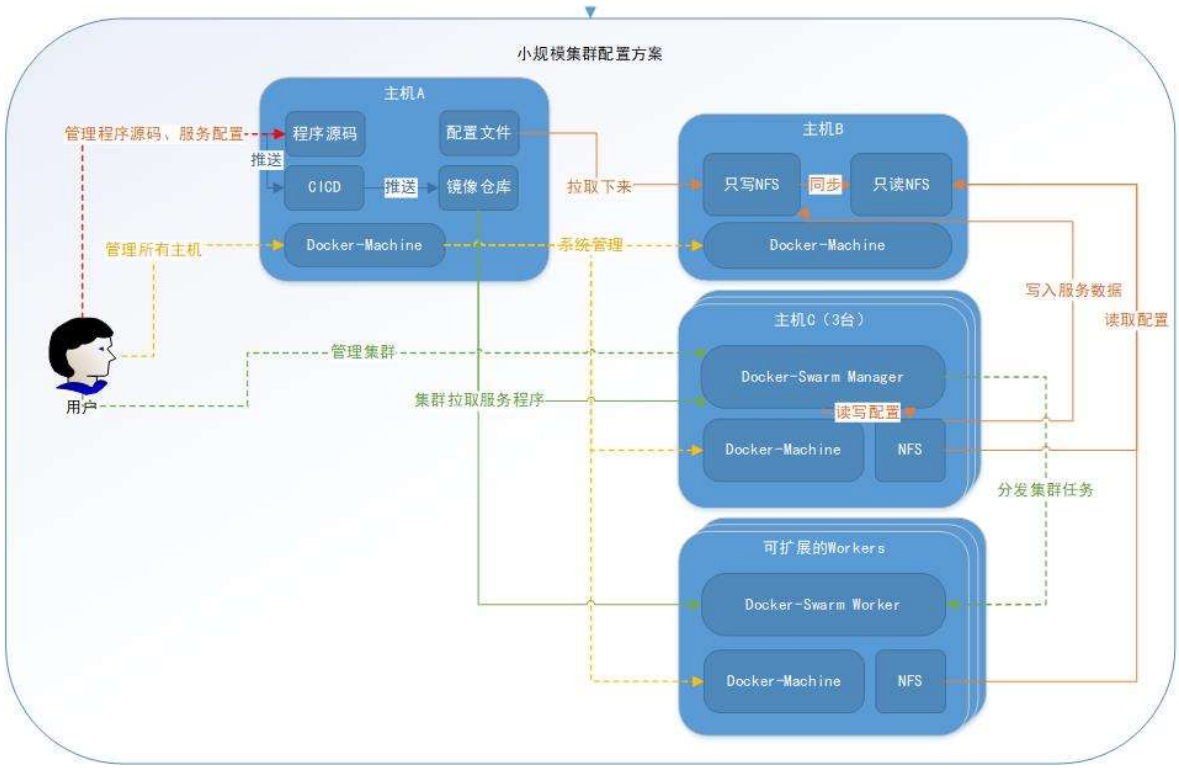

图 5.3 小规模集群配置方案示意图

1. 基础设施

一台主机 A、一台主机 B、三台主机 C、workers。

2. 系统管理

使用 Docker-machine 为五台机器提供统一 Docker 环境，主机 A 将作为 Master 节点管理其他主机，将其他主机收纳管理。

3. CI/CD 服务与数据管理

在主机 A（Master）上使用自建的 gitea、drone、Docker-registry 作为 CI/CD 系统。

4. 文件系统

在主机 B 上构建两个 NFS，构建读写分离，该主机专门完成文件读写操作。

5. 集群管理与扩展

在主机 C 上部署 Docker-Swarm 集群管理（Manager）节点，与 Master、NFS 分别为三种不同节点，挂载主机 B 上的可写文件系统、可读文件系统，通过拉取主机 A 上的镜像进行部署。构建另外两台主机 C，加入集群管理并升级为 Manager（集群有三个 manager），亦挂载主机 B 上的可写文件系统、可读文件系统。

扩展任意一台机器（建议内网优先）worker 节点，Master（主机 A）主动访问将其并入 Docker-machine 管理，登录 worker 将其加入到 Docker-Swarm 集群的 Manager（主机 C）中，集群 NFS（主机 B）只读文件系统，在 Manager（主机 C）调整集群任务，完成扩展。

5.4 中等规模集群配置方案

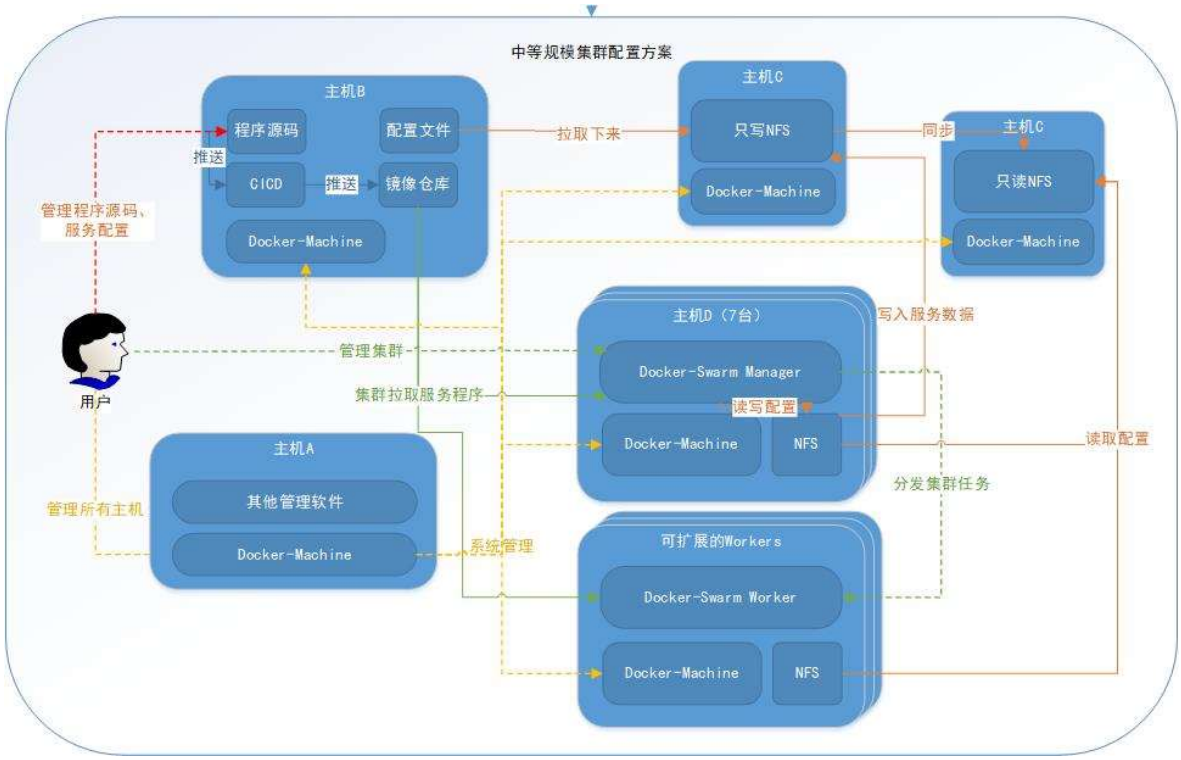

图 5.4 中等规模集群配置方案示意图

1. 基础设施

一台主机 A、一台主机 B、两台主机 C、七台主机 D、workers。

2. 系统管理

使用 Docker-machine 为所有台机器提供统一 Docker 环境，其中一台主机 A 将作为 Master 节点管理其他主机，将其他主机收纳管理。同时根据需要，在该台主机上构建堡垒服务（Jumpserver 等）、数据审计服务（Vault 等），将机器置于公网。

3. CI/CD 服务与数据管理

在主机 B 上，单独完成 CI 职能，可以使用功能较为丰富的 Jenkins。

4. 文件系统

在两台主机 C 上构建读写分离结构等一致性机制，置于内网。

5. 集群管理与扩展

相对于小规模集群配置方案，将所有集群机器都收纳内网，为应对宕机情况，除了 Manager 节点，部分 worker 节点可挂载读写系统，以便实现集群的高可用（Worker 亦有可能升为 Manager）。

5.5 方案之间的比较、演进

表 5.1 各个示例方案的比较与演进

	最小	默认	小规模	中等规模
基础设施	1+N	2+N	5+N	11+N
系统管理	A	A	A	A
CICD/数据管理				B
文件系统		B	B	C
集群管理			C	D
向上扩展	拆分文件系统与集群管理	拆分文件系统、集群管理	拆分系统管理、CI/CD/数据管理	
向下收缩		合并所有职能	合并文件系统、集群管理	合并系统管理、CI/CD/数据管理

6. 迁移示例

6.1 有状态与无状态服务

1. 无状态服务（Stateless Service）：

服务实例不会在本地存储需要持久化的数据，多个实例对于同一个请求响应的结果是完全一致的。这类服务的实例可能会因为一些原因停止或者重新创建，停止的实例里的所有信息都将丢失。

2. 有状态服务（Stateful Service）：

服务实例可以将一部分数据随时进行备份，在创建一个新的有状态服务时，可以通过备份恢复这些数据，以达到数据持久化的目的。有状态服务只能有一个实例，因此不支持“自动服务容量调节”。

这样的架构区分较为适合读写分离的集群系统,例如 redis 提供了 master、slave 端，master 端完成数据写操作，提供访问接口给 slave 端，slave 可以根据需要动态扩展，因而将 redis-master、redis-slave 分别封装为有状态服务、无状态服务。

通常来说，迁移有状态服务，会将可能变动的配置文件、实际的数据持久化存储进行声明。而无状态服务，会将默认与对应的主服务配置文件，硬编码进 Docker 镜像，多个配置则构建多个硬编码 Docker 镜像。

同时变量可动态声明，通过 Docker 的变量进而在单个镜像配置多个配置。选择硬编码与变量动态声明，是为了减少数据卷的使用，进而使得无状态服务可以更为灵活的扩展。

6.2 使用 Docker 打包服务应用

1. 整理数据与代码

整理整个服务所涉及的配置文件、程序文件（包括底层依赖库）、数据文件（如数据库文件、log 目录）以及一切所可能产生的、需要管理的数据目录与数据文件。

从配置文件中抽取出变量，在 Docker 运行前声明为环境变量。

选择稳定的程序文件来源。这里的来源包括不限于来自于官方发布版、来自系统软件库、平台编译，同时需要折中选择新旧版本与稳定性。

明确数据文件位置，以便映射所需要的目录地址。

2. 撰写 Dockerfile

Dockerfile 有自成一套的脚本语法，描述了从基础镜像（如系统镜像 Ubuntu、Alpine 或可执行程序镜像 Nginx），一步一步构建程序、声明变量与运行方式的过程，其中对于镜像操作使用基本的 shell 脚本。

3. 本地测试

使用 Docker build 可以将编写 Dockerfile 内容运行，拉取镜像，根据所描述的操作拉取服务程序，进而封装打包。

如若 build 没有出错，将在本地生成层结构化的 Docker 镜像。直接使用 Docker 启动该镜像，会按照既定的运行方式运行，如若运行目录、配置、端口等没有配置错误，新生成的 Docker 文件可以正常运行。

4. 上线测试

由于通常微服务会拆分为多个镜像完成单一职能，再组合多个镜像来完成一个大的功能服务（例如前台服务、中间件逻辑、文件服务、数据服务拆分后进行逻辑组合，而不是全部封装在同一个 Docker 内）。通常来说会在进行独立于生产环境的镜像测试，构建一个合理的有状态、无状态服务分离的组合服务，有利于服务框架的扩展。

6.3. 渐进式服务迁移

通常来说，会有在生产环境稳定旧版服务，另外创建新的服务集群构建环境，而通过流量分流（负载均衡、DNS 分流等），渐进地将服务迁移到新的框架中。这部分将在后文的“框架扩展”提及。

7. 测试与比较

7.1 部署方案的比较

表 7.1 部署方案的比较

	裸机部署方案	容器部署方案
环境依赖	需要集群环境统一化的构建	环境单独存储
服务更新	手动或脚本切换、热重载	自动、渐进地基于稳定环境切换
访问安全	程序访问无边界	严格控制，根据配置开放权限
部署速度	频繁发布场景依赖环境与脚本	频繁发布场景可以更快部署
方案瓶颈	接入新的同一环境集群机器	镜像分发效率与程序更新频率

7.2 跨机房部署

通常来说，来自同一服务商的服务，按同机柜之间、同可用区之间、同地区之间、同国家之间网络状况依次递减。部署的情况较为复杂，较难根据具体的跨机房部署结构分析每一部分的问题，但可以根据职能不同，推算跨机房部署对性能最低影响的需求。

表 7.2 职能及其所需的保障

职能	保障
系统管理	确保与所有主机的低延迟、确保 ssh 等通信
CI/CD 与数据管理	确保与所有主机访问可达，确保与所有主机之间的网速
文件系统	根据业务需要，高读写需要网速保障
集群管理	确保与所有集群 workers 的低延迟访问、确保 ssh 等通信

通常来说集群的节点会在确保连通性与网速的情况下，部署在不同服务提供商的不同可用区，以确保地区上的及时访问，以及集群的冗余。

7.3 压测

以默认配置为例，配置了两套集群方案进行压力测试：

表 7.3 压力测试部署方案

方案	裸机部署	Swarm 部署
Master 节点 1	Nginx 负载、Web 服务	Docker 网关作负载、Web 服务
Worker 节点 1	Web 服务	Web 服务
访问 Master	经 Nginx 分流，交付 Web 页面	经 Docker 网关，交付 Web 页面
访问 Worker	无法访问	经 Docker 网关，交付 Web 页面

以及两套方案嵌套 CDN 服务后的方案，共四套方案

服务部署选择了 Vultr 西雅图机房，压测使用了阿里云性能测试 PTS，并发数从 0 至 100，来自全球各地压力源，检测其每秒相应次数，获取 Web 服务页面，结果如下：

表 7.4 压力测试结果

方案		裸机部署	Swarm 部署	裸机+CDN	Swarm+CDN
Master 应答时间(ms)	平均访问	541.11	589.54	415.15	453.51
	低频访问	436.98	570.12	406.68	443.62
	高频访问	645.24	608.96	423.62	463.4
Master 请求成功率		99.98%	99.99%	99.99%	99.99%
平均每秒响应(次)		89	104	110	115
Worker 应答时间(ms)	平均访问	无	686.11	无	471.23
	低频访问	无	616.82	无	451.51
	高频访问	无	755.4	无	490.95
Worker 请求成功率		无	99.98%	无	99.99%
平均每秒响应(次)		无	97	无	110

在同等规模集群的应用部署上取得了较为接近的效果。裸机部署随访问频度应答时间变化较大，Swarm 部署则响应稳定。同时嵌套 CDN 后两套方案各方面都有了显著增加。另外，Swarm 要比裸机部署多消耗一些时间（约 40ms），原因在于 Docker-Swarm

使用的逻辑内网，从服务监听转发与内网 DNS 查询所花费的时间。

另外一方面好处在于可以使用 worker 分担流量（即除了 Swarm 本身的负载，可额外再做一层负载），以应对增长的访问频度。

7.4 滚动部署测试

因部署项目的多样性，无法做出统一比较。取同一项目在云服务提供商中等配置（16c32g）的机器上，从 CI 中用于 Docker 封装的工作流，到具体的部署，排除网速因素制约，基本可以秒级到分钟级别完成。

7.5 集群从零部署速度测试

默认配置可以在 15～20 分钟完成，因方案不同所需要应对业务的设计、测试部署、整理所花费的时间不等。在明确的方案指导下，可以在小时以内完成从零部署。

8. 框架扩展与相关问题讨论

8.1 系统与故障策略

更为完善的集群框架还应当有日志与审计系统、监控与警报处理，甚至是决策自动化等去手动化工具的整合，进而组合成一个功能完善的运维系统。

而节点故障有较多的情况出现，以节点宕机的恢复、基础设施故障的迁移为例。两者区别在于是否需要迁移整个系统。

8.1.1 节点宕机问题

节点宕机需要做的工作是重启、恢复服务。

1. 系统管理节点

会导致整套系统的管理途径受阻，通常来说是需要尽快恢复。可以通过服务提供商提供的原本接口进行单个节点或多节点的管理。

2. CI/CD 与数据管理节点

会导致代码托管、CI/CD 系统受阻，属于内部开发职能的缺失，而因为会影响到 registry 进而影响 Swarm 集群的滚动更新，灾难程度视发布频率而定。

3. 文件系统

会导致集群系统出现异常，通常来说会将文件系统置于内网，且仅与管理节点、集群相关联，需要最快恢复。可以选用多热备的、专门用于集群访问的文件系统结构，如 NAS 群组。

4. Swarm 集群 Manager 节点宕机

Manager 节点需要时刻维护、保存当前 Swarm 集群中各个节点的一致性状态（分

布式一致性算法 Raft[14]）。为了确保集群的 Manager 节点们的高可用，通常会选择部署奇数个 Manager 节点，以便于在一定数量 Manager 宕机的时候，不用关机维护乃至迁移整个集群,进而确保服务的高可用、热扩展特性。部署N个节点最多可容纳(N-1)/2个节点宕机。

8.1.2 基础设施故障

基础设施故障需要做的工作是迁移文件、重新构建服务

1. 系统管理节点迁移

因为连接着所有的服务器的管理接口，由包含着管理的 log 等数据。在急需迁移的情况下，该节点应当在新环境最先部署，在旧环境最后撤离。如果没办法快速定位管理节点中一些重要数据文件位置，通常还需要在新环境稳定后进行清理。

2. CI/CD 与数据管理节点迁移

需要从开始就明确项目的位置，同时进行备份或版本管理，以防出现该问题。在急需迁移的情况下，该节点需要在集群部署之前迁移完数据，在获取数据后快速使用 CI 集成可执行文件，进而使得集群可以运转。

3. 文件系统迁移

在整个集群服务起重要作用，需要在管理节点迁移之后、Swarm 集群建立之前进行。除了依赖网速的、从旧文件系统上将文件迁移过来所耗费的时间，配置新的、与之前较为一致的文件系统结构也需要一定时间。尽可能选用具有备份的，或在内网构建多个文件系统。

4. Swarm 集群 Manager 节点迁移

如果将 Swarm 的 Manager 部署在不同服务提供商与可用区，出现全部 Manager 宕机或者逼近最大宕机容忍数量的可能性较低，可以通过 Swarm 本身的高可用来完成迁移。如果出现全部 Manager 宕机，则需要在另外新环境里快速构建集群，同时对服务进行迁移（快速部署需要有稳定的 config 文件支撑）。

8.2 负载均衡与滚动更新

通常来说，服务程序的切换不应在一开始便全部投入使用，使用蓝绿部署、通过访问分流，渐进式地完成服务更新。

而访问分流的方式较多，如 DNS 分流、使用负载均衡服务器进行分流，亦或服务版本重定向等。在集群使用滚动更新是使用等间隔进行服务版本的替代，以至全部更新。

关于部署全自动化，Swarm 本没有提供直接支持，可配合其他一些工具完成。同时关于滚动上线的业务，建议手动监督部署，以便出现错误时候可以及时回滚。

8.3 文件服务器与文件系统

文件系统根据业务会有所不同，如 HDFS、NFS 或 NFS 的扩展 NFSP[15]等任何适用于集群的文件系统，小规模可单台机器或读写分离，规模增大需要引入分布式文件系统。Swarm 也提供了 global 部署，会在集群的每一个节点上部署同样的服务，以便构建文件系统。

8.4 往 Kubernetes 的切换

与 Swarm 相同，Kubernetes 也使用 yaml 作为描述文件，也有工具支持将 Docker-compose 文件转为 Kuberntes 对象。就产品定位而言，如果是极大扩展的需要而将 Swarm 转向 Kubernetes，关于服务的描述应当重新整理为后者所需。

8.5 公网访问安全

1. 使用内网地址进行通信以确保集群之间的安全。
2. 若所有节点均需要公网访问，则做好审计、访问控制、身份认证的工作。
3. 为整套系统使用 CDN 保护，一些 DNS 服务提供商也有在 CDN 服务里增加了 DDoS protection 功能，如 Cloudflare、阿里云等。

9. 结论

本项目完成了一个以 Docker 为基础的基本服务器集群框架，解决了服务访问控制、数据存储、应对弹性计算需求、服务配置冗杂、调用方式不统一、构建流程手动化等问题。基于 Docker 封装的程序、使用社区工具构建 Docker 小集群，整合多个开源项目构建 CI/CD 功能，构建一个基本的、可扩展的服务器集群框架，实现完成管理与实例抽离、结构与服务抽离，取得了短时间创建实例并部署服务、代码从提交到测试的自动化及部署滚动更新的效果。

DevOps 与运维自动化在日益复杂的服务环境中是必然趋势，容器技术的特性很好地符合了趋势，目前发展仍方兴未艾，但在可见的未来会成为基础设施的标准之一。个人与小型企业根据业务需要，架设可定制的服务框架，云基础设施的普及与互联网的不可替代性，带来了较大的市场潜力。在可见的未来，容器技术将成为服务器部署的中流砥柱，推动 DevOps 快速发展。

参考文献

[1] 王亚玲，李春阳，崔蔚，等. 基于 Docker 的 PaaS 平台建设[J]. 计算机系统应用，2016，25(3)：72-77.

[2] 牛晓玲，吴蕾. DevOps 发展现状研究[J]. 电信网技术，2017，10：48-51.

[3] 蒋勇. 基于微服务架构的基础设施设计[J]. 软件，2016，37(5)：93-97.

[4] 王骏翔,郭磊.基于Kubernetes和Docker技术的企业级容器云平台解决方案[J].上海船舶运输科学研究所学报,2018,41(03):51-57.

[5] 姜文，刘立康. 基于云环境的持续集成[J]. 计算机技术与发展，2018，28(1)：11-16.

[6] 蔡志强. 基于 Docker 技术的容器隔离性分析[J]. 电子世界，2017，17：195.

[7] Naik N. Building a virtual system of systems using Docker Swarm in multiple clouds[C]//2016 IEEE International Symposium on Systems Engineering (ISSE). IEEE, 2016: 1-3.

[8] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.

[9] 吴杰楚. 基于 Docker-Swarm 的微服务管理技术研究与实现[D]. 华南理工大学，2018.

[10] 刘胜强，王晶. Docker 的 Hadoop 平台架构分析[J]. 自动化与仪器仪表，2018 (10)：55.

[11] Petersen R. Ubuntu 18.04 LTS Server: Administration and Reference[M]. surfing turtle press, 2018.

[12] Freas C, Jones J. Continuous integration and continuous deployment/delivery for software systems[J]. 2018.

[13] 刘梅. 基于 docker 的持续集成及发布平台的设计与实现[D]. 中国科学院大学 (中国科学院沈阳计算技术研究所)，2018.

[14] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014: 305-319.

[15] Lombard P, Denneulin Y. nfsp: a distributed NFS server for clusters of workstations[C]//Proceedings 16th International Parallel and Distributed Processing Symposium. IEEE, 2001: 6 pp.