Kubernetes在企业信息化建设中的实践

目录

[Kubernetes在企业信息化建设中的实践 1](#_Toc20321227)

[前言 11](#_Toc20321228)

[第一章 容器云的技术基础 13](#_Toc20321229)

[1 Kubernetes的整体概述(2019-04-22 Y) 13](#_Toc20321230)

[1.1 Kubernetes是什么 13](#_Toc20321231)

[1.2 Kubernetes的整体架构 14](#_Toc20321232)

[1.3 Master Node（主节点） 15](#_Toc20321233)

[1.3.1 API Server（API服务器） 15](#_Toc20321234)

[1.3.2 Cluster state store（集群状态存储） 15](#_Toc20321235)

[1.3.3 Controller-Manager Server（控制管理服务器） 16](#_Toc20321236)

[1.3.4 Scheduler（调度器） 16](#_Toc20321237)

[1.4 Worker Node（工作节点） 16](#_Toc20321238)

[1.4.1 Kubelet（命令行工具） 17](#_Toc20321239)

[1.4.2 Container Runtime（容器运行时） 17](#_Toc20321240)

[1.4.3 kube proxy（代理） 18](#_Toc20321241)

[1.5 kubectl（命令行工具） 18](#_Toc20321242)

[1.6 Add-On（附加项和其他依赖） 19](#_Toc20321243)

[2 Kubernetes功能逻辑架构(2019-06-18 Y) 20](#_Toc20321244)

[2.1 整体介绍 20](#_Toc20321245)

[2.2 逻辑层级 21](#_Toc20321246)

[2.2.1 核心层：API和执行 21](#_Toc20321247)

[2.2.2 应用层：部署和路由 24](#_Toc20321248)

[2.2.3 治理层：自动化和策略执行 25](#_Toc20321249)

[2.2.4 接口层：类库和工具 27](#_Toc20321250)

[2.2.5 生态层：外部系统提供的功能 27](#_Toc20321251)

[3 Kubernetes离线环境搭建(2019-05-16 Y) 30](#_Toc20321252)

[3.1 环境准备 30](#_Toc20321253)

[3.1.1 操作系统 30](#_Toc20321254)

[3.1.2 设置防火墙策略 32](#_Toc20321255)

[3.1.3 配置主机时间、时区、系统语言 33](#_Toc20321256)

[3.1.4 环境清理（可选） 33](#_Toc20321257)

[3.2 安装介质下载和准备 34](#_Toc20321258)

[3.3 Docker安装部署 35](#_Toc20321259)

[3.3.1 Docker安装 35](#_Toc20321260)

[3.3.2 设置根目录(可选) 36](#_Toc20321261)

[3.3.3 配置远程访问 37](#_Toc20321262)

[3.3.4 设置docker开机启用 37](#_Toc20321263)

[3.3.5 docker卸载 37](#_Toc20321264)

[3.4 提供网络存储(可选) 38](#_Toc20321265)

[3.5 私有镜像仓库安装部署 39](#_Toc20321266)

[3.6 Kubernetes安装部署 40](#_Toc20321267)

[3.6.1 安装rancher服务 40](#_Toc20321268)

[3.6.2 添加节点 41](#_Toc20321269)

[3.7 安装kubectl 43](#_Toc20321270)

[3.8 安装helm 44](#_Toc20321271)

[3.8.1 安装Helm客户端 44](#_Toc20321272)

[3.8.2 安装Tiller服务器 45](#_Toc20321273)

[3.8.3 验证安装 46](#_Toc20321274)

[4 Kubernetes快速入门(2019-06-19 Y) 48](#_Toc20321275)

[4.1 入门概述 48](#_Toc20321276)

[4.2 步骤1：部署应用 48](#_Toc20321277)

[4.3 步骤2：获取应用的Pod信息 50](#_Toc20321278)

[4.4 步骤3：对外暴露应用 52](#_Toc20321279)

[4.5 步骤4：扩缩容应用 54](#_Toc20321280)

[4.6 步骤5：升级应用 58](#_Toc20321281)

[5 Kubernetes核心对象 63](#_Toc20321282)

[5.1 工作负载(2019-04-11 Y) 63](#_Toc20321283)

[5.1.1 Pod（容器组）Y 63](#_Toc20321284)

[5.1.2 Deploymnet（部署） 86](#_Toc20321285)

[5.1.3 StatefulSet（有状态副本集） 99](#_Toc20321286)

[5.1.4 DaemonSet（守护进程）(Y) 104](#_Toc20321287)

[5.1.5 CronJob（定时任务）（Y） 110](#_Toc20321288)

[5.2 服务发现（2019-04-12 Y） 113](#_Toc20321289)

[5.2.1 Service（服务）整体介绍 113](#_Toc20321290)

[5.2.2 发现服务 118](#_Toc20321291)

[5.2.3 发布服务和服务类型 120](#_Toc20321292)

[5.2.4 服务的相关kubectl命令 122](#_Toc20321293)

[5.2.5 Ingress（反向代理） 123](#_Toc20321294)

[5.3 配置与存储(2019-04-02) 129](#_Toc20321295)

[5.3.1 Volume（存储卷） 129](#_Toc20321296)

[5.3.2 PersistentVolume（持久化数据卷） 134](#_Toc20321297)

[5.3.3 PersistentVolumeClaim（持久化卷声明） 143](#_Toc20321298)

[5.3.4 StorageClass（持久化类） 145](#_Toc20321299)

[5.3.5 Secret（保密文件） 158](#_Toc20321300)

[5.3.6 ConfigMap（配置 2019-04-09 Y） 158](#_Toc20321301)

[6 Kubernetes安全 Y 169](#_Toc20321302)

[6.1 身份认证(2019-04-12 ) 169](#_Toc20321303)

[6.1.1 Kubernetes中的用户 169](#_Toc20321304)

[6.1.2 认证策略（Authentication strategies） 170](#_Toc20321305)

[6.1.3 X509客户端证书 170](#_Toc20321306)

[6.1.4 Service Account令牌 173](#_Toc20321307)

[6.1.5 静态Tokent文件 175](#_Toc20321308)

[6.1.6 Bootstrap令牌 176](#_Toc20321309)

[6.1.7 静态密码文件 176](#_Toc20321310)

[6.1.8 Keystone密码 177](#_Toc20321311)

[6.1.9 匿名请求 177](#_Toc20321312)

[6.2 访问授权(2019-04-15 Y) 178](#_Toc20321313)

[6.2.1 RBAC介绍 178](#_Toc20321314)

[6.2.2 命令行工具 182](#_Toc20321315)

[6.2.3 服务帐户权限 183](#_Toc20321316)

[6.3 日志管理 186](#_Toc20321317)

[6.3.1 统一日志管理的整体方案 186](#_Toc20321318)

[6.3.2 安装统一日志管理的组件 189](#_Toc20321319)

[6.3.3 日志数据展示 199](#_Toc20321320)

[7 Kubernetes客户端和工具(2019-04-10 Y) 202](#_Toc20321321)

[7.1 Kubectl命令行工具 202](#_Toc20321322)

[7.1.1 kubectl概述 202](#_Toc20321323)

[7.1.2 kubectl安装部署 207](#_Toc20321324)

[7.1.3 kubectl的常用命令示例 208](#_Toc20321325)

[7.2 Dashboard界面 212](#_Toc20321326)

[7.2.1 部署Dashboard用户界面 212](#_Toc20321327)

[7.2.2 访问Dashboard用户界面 212](#_Toc20321328)

[7.2.3 使用Dashboard 214](#_Toc20321329)

[7.3 Helm部署工具 218](#_Toc20321330)

[7.3.1 Helm介绍 218](#_Toc20321331)

[7.3.2 Helm安装部署 221](#_Toc20321332)

[7.3.3 使用Helm 223](#_Toc20321333)

[7.3.4 Chart应用仓库 226](#_Toc20321334)

[第二章 生态环境 227](#_Toc20321335)

[8 镜像仓库(2019-04-16 Y) 227](#_Toc20321336)

[8.1 安装Nexus 227](#_Toc20321337)

[8.2 构建私有镜像仓库 228](#_Toc20321338)

[8.3 基本操作 231](#_Toc20321339)

[8.3.1 登录认证 231](#_Toc20321340)

[8.3.2 推送镜像 231](#_Toc20321341)

[8.3.3 拉取镜像 232](#_Toc20321342)

[8.3.4 从私有镜像拉取镜像 233](#_Toc20321343)

[9 网络(2019-05-05 Y) 235](#_Toc20321344)

[9.1 Docker网络模式 235](#_Toc20321345)

[9.1.1 bridge网络的构建过程 236](#_Toc20321346)

[9.1.2 外部访问 237](#_Toc20321347)

[9.2 Kubernetes网络模式 238](#_Toc20321348)

[9.2.1 同一个Pod中容器之间的通信 238](#_Toc20321349)

[9.2.2 不同Pod中容器之间的通信 239](#_Toc20321350)

[9.2.3 数据传递过程 239](#_Toc20321351)

[10 文件存储(2019-04-17 Y) 241](#_Toc20321352)

[10.1 NFS介绍 241](#_Toc20321353)

[10.1.1 nfs原理 241](#_Toc20321354)

[10.1.2 共享配置 242](#_Toc20321355)

[10.2 nfs服务端配置 244](#_Toc20321356)

[10.2.1 安装nfs服务 244](#_Toc20321357)

[10.2.2 创建用户和共享目录 245](#_Toc20321358)

[10.2.3 配置共享目录 245](#_Toc20321359)

[10.2.4 启动服务 246](#_Toc20321360)

[10.2.5 检查nfs服务是否正常启动 246](#_Toc20321361)

[10.3 nfs作为volume 246](#_Toc20321362)

[10.4 nfs作为PersistentVolum 247](#_Toc20321363)

[10.5 nfs作为动态存储提供 248](#_Toc20321364)

[10.5.1 部署nfs-provisioner 248](#_Toc20321365)

[10.5.2 创建StorageClass 250](#_Toc20321366)

[10.5.3 创建PersistenetVolumeClaim 250](#_Toc20321367)

[10.5.4 创建使用PersistenVolumeClaim的部署 251](#_Toc20321368)

[第三章 应用实践 253](#_Toc20321369)

[11 基于Dockfile构建docker镜像实践(2019-04-17 Y) 253](#_Toc20321370)

[11.1 Dockerfile文件和核心指令 253](#_Toc20321371)

[11.1.1 FROM：设置基础镜像 254](#_Toc20321372)

[11.1.2 RUN：设置构建镜像时执行的命令 254](#_Toc20321373)

[11.1.3 CMD：设置容器的默认执行命令 254](#_Toc20321374)

[11.1.4 ENTRYPOINT：设置容器为可执行文件 255](#_Toc20321375)

[11.1.5 ENV：设置环境变量 257](#_Toc20321376)

[11.1.6 ADD：添加内容到容器中 258](#_Toc20321377)

[11.1.7 COPY：拷贝内容到镜像中 259](#_Toc20321378)

[11.1.8 WORKDIR：设置当前工作目录 259](#_Toc20321379)

[11.1.9 EXPOSE：设置暴露的端口 260](#_Toc20321380)

[11.1.10 LABEL：设置镜像的元数据信息 260](#_Toc20321381)

[11.1.11 VOLUME：设置存储卷 261](#_Toc20321382)

[11.2 构建镜像 261](#_Toc20321383)

[11.2.1 命令选项 261](#_Toc20321384)

[11.2.2 URL参数 263](#_Toc20321385)

[11.2.3 构建示例 263](#_Toc20321386)

[11.3 最佳实践 264](#_Toc20321387)

[12 基于Kubernetes的Devops平台(2019-5-23 Y) 267](#_Toc20321388)

[12.1 基于kubernetes devops的整体方案 267](#_Toc20321389)

[12.2 组件安装部署 268](#_Toc20321390)

[12.2.1 代码托管工具-Gitlab 269](#_Toc20321391)

[12.2.2 镜像仓库-Nexus 271](#_Toc20321392)

[12.2.3 流水线工具-Jenkins 272](#_Toc20321393)

[12.3 devops平台搭建 274](#_Toc20321394)

[12.3.1 nexus设置 274](#_Toc20321395)

[12.3.2 maven设置 275](#_Toc20321396)

[12.3.3 docker设置 277](#_Toc20321397)

[12.3.4 jenkins设置 277](#_Toc20321398)

[12.4 devops持续集成示例 278](#_Toc20321399)

[13 部署高可用的mysql（2019-5-22 Y） 284](#_Toc20321400)

[13.1 MySQL简介 284](#_Toc20321401)

[13.2 MySQL的高可用方案 284](#_Toc20321402)

[13.3 安装部署 286](#_Toc20321403)

[13.3.1 环境要求 286](#_Toc20321404)

[13.3.2 部署MySql 286](#_Toc20321405)

[13.4 理解有状态Pod的初始化 291](#_Toc20321406)

[13.4.1 创建配置文件 291](#_Toc20321407)

[13.4.2 克隆已存在的数据 291](#_Toc20321408)

[13.4.3 启动副本 292](#_Toc20321409)

[13.5 MySQL部署环境验证 292](#_Toc20321410)

[13.6 扩容slave的数量 293](#_Toc20321411)

[14 部署高可用的redis（2019-05-14 Y） 294](#_Toc20321412)

[14.1 Redis简介 294](#_Toc20321413)

[14.2 基于Sentinel模式的高可用方案 294](#_Toc20321414)

[14.3 安装部署 296](#_Toc20321415)

[14.3.1 环境要求 296](#_Toc20321416)

[14.3.2 Helm chart配置 296](#_Toc20321417)

[14.3.3 持久化 298](#_Toc20321418)

[14.3.4 通过Chart安装Redis 298](#_Toc20321419)

[14.4 Helm Chart分析 299](#_Toc20321420)

[14.4.1 values.yaml 299](#_Toc20321421)

[14.4.2 redis-server-deployment.yaml 300](#_Toc20321422)

[14.4.3 redis-master-service.yaml 301](#_Toc20321423)

[14.4.4 redis-slave-service.yaml 302](#_Toc20321424)

[14.4.5 redis-sentinel-deployment.yaml 302](#_Toc20321425)

[14.4.6 redis-sentinel-service.yaml 303](#_Toc20321426)

[14.4.7 redis-serviceaccount.yaml 304](#_Toc20321427)

[14.4.8 redis-role.yaml 304](#_Toc20321428)

[14.4.9 redis-rolebinding.yaml 304](#_Toc20321429)

[14.4.10 redis-auth-secret.yaml 305](#_Toc20321430)

[14.5 Redis部署环境验证 305](#_Toc20321431)

[15 基于Promethues和Grafana的系统监控(2019-05-13 Y) 308](#_Toc20321432)

[15.1 Prometheus介绍 308](#_Toc20321433)

[15.2 Prometheus关键概念 310](#_Toc20321434)

[15.2.1 数据模型 310](#_Toc20321435)

[15.2.2 指标类型 311](#_Toc20321436)

[15.2.3 工作和实例 312](#_Toc20321437)

[15.3 使用Helm在Kubernetes中部署Prometheus 312](#_Toc20321438)

[15.3.1 环境要求 313](#_Toc20321439)

[15.3.2 通过Chart安装Prometheus 313](#_Toc20321440)

[15.4 Grafana介绍和部署 313](#_Toc20321441)

[15.4.1 Grafana介绍 313](#_Toc20321442)

[15.4.2 基于Helm在Kubernetes中部署Grafana 315](#_Toc20321443)

[15.5 监控Kubernetes实践 315](#_Toc20321444)

[15.5.1 监控Kubernetes内容分析 315](#_Toc20321445)

[15.5.2 在Grafana中配置数据源 316](#_Toc20321446)

[15.5.3 监控实践 316](#_Toc20321447)

[15.6 Rancher中使用Prometheus和Grafana进行性能监控 318](#_Toc20321448)

[15.6.1 集群性能监控 318](#_Toc20321449)

[15.6.2 项目/命名空间性能监控 320](#_Toc20321450)

[16 基于client-go的Kubernetes定制开发指南 322](#_Toc20321451)

[16.1 安装和配置开发环境 322](#_Toc20321452)

[16.1.1 安装kubectl 322](#_Toc20321453)

[16.1.2 安装go-client 323](#_Toc20321454)

[16.2 开发示例 327](#_Toc20321455)

[16.2.1 示例介绍 327](#_Toc20321456)

[16.2.2 运行示例 330](#_Toc20321457)

[17 管理控制台备份和恢复 334](#_Toc20321458)

[17.1 备份 334](#_Toc20321459)

[17.2 恢复 335](#_Toc20321460)

[18 问题定位和处理 336](#_Toc20321461)

[18.1 应用部署问题定位和处理(2019-05-22 Y) 336](#_Toc20321462)

[18.1.1 应用部署问题处理的整体思路 336](#_Toc20321463)

[18.1.2 调试Pods 337](#_Toc20321464)

[18.1.3 调试代理服务 340](#_Toc20321465)

[18.2 docker守护程序问题处理 343](#_Toc20321466)

[18.2.1 检查Docker是否正在运行 343](#_Toc20321467)

[18.2.2 解决*daemon.json*与启动脚本之间的冲突 343](#_Toc20321468)

[18.2.3 查看日志 344](#_Toc20321469)

[18.3 处理由于服务器时间不一致带来的问题 346](#_Toc20321470)

[19 企业网盘基于Kubernetes的研发和运行 350](#_Toc20321471)

[19.1 企业网盘介绍 350](#_Toc20321472)

[19.2 基于DevOps的持续集成 351](#_Toc20321473)

[19.2.1 企业网盘的产品研发团队 351](#_Toc20321474)

[19.2.2 基于DevOps的研发过程 351](#_Toc20321475)

[参考文献 357](#_Toc20321476)

# 前言

编写本书的意图是为了积淀多年在Kubernetes方面的掌握的知识、技术和实践经验，以希望能够在航天等企业中进行Kubernetes理念和技术的推广，并基于Kubernetes帮助航天等企业提升信息化的运维能力和效率。

本书主要由Kubernetes的基础知识、生态环境和应用实践这三部分所组成。基础知识部分是整本书的基础，此部分会向读者介绍Kubernetes的整体架构和发展路线图；以及如何在联网和离线的情况下进行安装和部署；并通过快速入门的内容快速引导读者使用Kubernetes；接下来会详细阐述工作负载、服务发现和配置与存储这些核心的资源对象；在安全方面，阐述了如何进行身份认证、访问控制、性能监控和日志管理；在工具部分，介绍了kubectl、Dashboard界面和Helm部署工具。

Kubernetes作为容器的运行和管理平台，首先要解决的就是容器镜像的来源问题。因此，生态环境部分首先读者介绍什么是镜像仓库，如何搭建自己的私有镜像仓库；接着，以flannel为例，介绍Kubernetes中的网络模型；由于容器本身不进行数据的持久化，因此需要通过网络文件存储进行数据的持久化，网络存储以NFS为例，介绍如何将容器的数据保存到持久化设备中。

在应用实践部分，本书将以部署java应用和php应用为实例。从如何构建镜像开始，结合基于Kubernetes的Devops平台，系统性的介绍如何在Kubernetes上进行应用系统生产环境的部署和实践。同时，为了帮助读者处理在使用容器云的过程中遇到的问题，提供了有针对性的问题定位和处理方法。最后，对Kubernetes的关键特性进行分析和梳理。

本书结合了传统企业在信息化建设的特色和经验，体系化的介绍了Kubernetes的技术能力，并提供了可供企业借鉴的在Kubernetes上进行应用部署和运维的实例。

本书主要由季向远编写，中国空间技术研究院的纪炜和北京神舟软件技术有限公司的王林参与了本书的编写工作。本书在编写过程中得到了杨涛、李巧红、王善鹏、冯少敏、王翼冲等领导和同事的帮助，在此一并表示感谢。

由于Kubernetes和相关技术一直处于发展状态，加之作者水平有限，编写时间仓促，因而本书难免存在错漏之处，恳请读者见谅。

本书的主要目标读者为企业信息化运维和管理人员,以及软件工程师、架构师和咨询顾问等其他人员。

季向远

# 容器云的技术基础

# Kubernetes的整体概述(2019-04-22 Y)

此章节作为本书的第一部分，首要的任务是要阐述清楚Kubernetes是什么，以及能够给用户带来什么益处。另外，在本章节会着重说明Kubernetes的整体架构，以及相应的Master Node、Worker Node、kubectl和Add-On组成部分，以便读者对于Kubernetes有一个整体的认识。

## Kubernetes是什么

Kubernetes是一个轻便的和可扩展的开源容器云平台，用于管理容器化的应用和服务，能够很好的实现传统云计算体系框架中的PaaS层。通过Kubernetes，应用能够进行自动化的部署和扩缩容。在Kubernetes中，将组成应用的所有容器组合成一个自包含的逻辑单元，以更易进行管理和发现。作为Google生产环境运行工作负载，Kubernetes已经积累了15多年的实践经验，并吸收和融合了来自于社区的最佳想法和实践。经过多年的快速发展，Kubernetes形成了完善的生态体系。在2014年，Google将Kubernetes进行了开源，Kubernetes的关键特性如下：

* **自动化部署**：在不牺牲可用性的条件下，基于容器对于资源的要求和约束自动的部署容器。同时，为了提高资源利用率，Kubernetes会对关键和最佳工作量进行综合考虑。
* **应用自愈：**当容器运行失败时，Kubernetes会在后台自动对容器进行重启；当容器所部署的主机节点存在问题时，Kubernetes会对容器进行重新调度和部署；当容器未通过监控检查时，会关闭此容器；直到容器正常运行时，才会对外提供服务。
* **水平扩容**：通过简单的命令、用户界面或基于CPU的使用情况，能够对应用进行扩容和缩容。
* **服务发现和负载均衡**：开发者不需要使用额外的服务发现机制，就能够基于Kubernetes进行服务发现和负载均衡。
* **自动发布和回滚**：Kubernetes能够程序化的发布应用和相关的配置。如果发布有问题，Kubernetes将能够回归发生的变更。
* **保密和配置管理**：在不需要重新构建镜像的情况下，可以部署和更新保密和应用配置。
* **分布式存储**：自动挂接存储系统，这些存储系统可以来自于本地、公共云提供商（例如：GCP和AWS）、网络存储(例如：NFS、iSCSI、Gluster、Ceph、Cinder和Floker等)。

## Kubernetes的整体架构

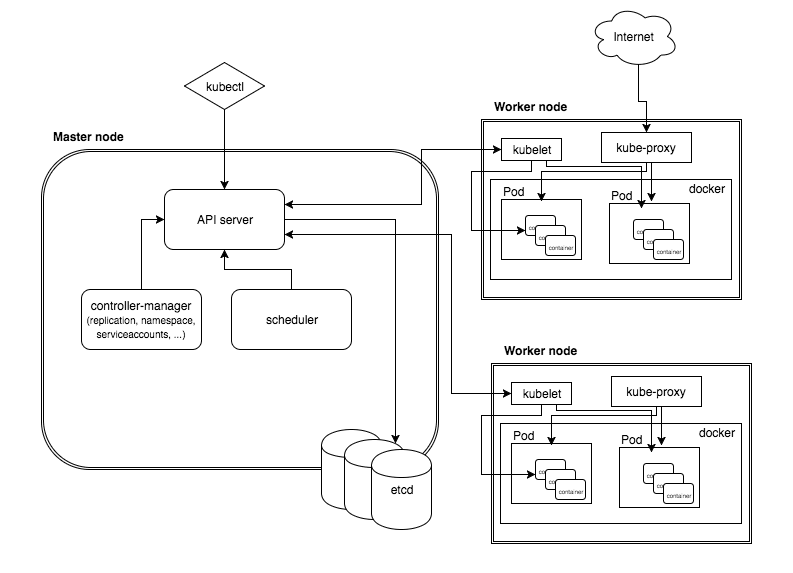


图 1-1 Kubernetes整体架构

Kubernetes属于主从分布式架构，主要由Master Node和Worker Node组成，以及包括客户端命令行工具kubectl和其它附加项。

* **主节点（Master Node）：**作为控制节点，对集群进行调度管理；Master Node由API Server、Scheduler、Cluster State Store和Controller-Manger Server所组成；
* **工作节点（Worker Node）：**作为真正的工作节点，运行业务应用的容器；Worker Node包含kubelet、kube proxy和Container Runtime；
* **命令行工具（kubectl）：**用于通过命令行与API Server进行交互，而对Kubernetes进行操作，实现在集群中进行各种资源的增删改查等操作；
* **附加功能（Add-on）：**是对Kubernetes核心功能的扩展，例如增加网络和网络策略等能力。

## Master Node（主节点）

### API Server（API服务器）

API Server主要用来处理REST的操作，确保它们生效，并执行相关业务逻辑，以及更新etcd（或者其他存储）中的相关对象。API Server是所有REST命令的入口，它的相关结果状态将被保存在etcd（或其他存储）中。API Server的基本功能包括：

* REST语义，监控，持久化和一致性保证，API 版本控制，放弃和生效；
* 内置准入控制语义，同步准入控制钩子，以及异步资源初始化；
* API注册和发现。

另外，API Server也作为集群的网关。默认情况，客户端通过API Server对集群进行访问，客户端需要通过认证，并使用API Server作为访问Node和Pod（以及service）的堡垒和代理/通道。

### Cluster state store（集群状态存储）

Kubernetes默认使用etcd作为集群整体存储，当然也可以使用其它的技术。etcd是一个简单的、分布式的、一致的key-value存储，主要被用来共享配置和服务发现。etcd提供了一个CRUD操作的REST API，以及提供了作为注册的接口，以监控指定的Node。集群的所有状态都存储在etcd实例中，并具有监控的能力，因此当etcd中的信息发生变化时，就能够快速的通知集群中相关的组件。

### Controller-Manager Server（控制管理服务器）

Controller-Manager Serve用于执行大部分的集群层次的功能，它既执行生命周期功能(例如：命名空间创建和生命周期、事件垃圾收集、已终止垃圾收集、级联删除垃圾收集、node垃圾收集)，也执行API业务逻辑（例如：pod的弹性扩容）。控制管理提供自愈能力、扩容、应用生命周期管理、服务发现、路由、服务绑定和提供。Kubernetes默认提供Replication Controller、Node Controller、Namespace Controller、Service Controller、Endpoints Controller、Persistent Controller、DaemonSet Controller等控制器。

### Scheduler（调度器）

Scheduler组件为容器自动选择运行的主机。依据请求资源的可用性，服务请求的质量等约束条件，scheduler监控未绑定的pod，并将其绑定至特定的node节点。Kubernetes也支持用户自己提供的调度器，Scheduler负责根据调度策略自动将Pod部署到合适Node中，调度策略分为预选策略和优选策略，Pod的整个调度过程分为两步：

1）预选Node：遍历集群中所有的Node，按照具体的预选策略筛选出符合要求的Node列表。如没有Node符合预选策略规则，该Pod就会被挂起，直到集群中出现符合要求的Node。

2）优选Node：预选Node列表的基础上，按照优选策略为待选的Node进行打分和排序，从中获取最优Node。

## Worker Node（工作节点）

### Kubelet（命令行工具）

Kubelet是Kubernetes中最主要的控制器，它是Pod和Node API的主要实现者，Kubelet负责驱动容器执行层。在Kubernetes中，应用容器彼此是隔离的，并且与运行其的主机也是隔离的，这是对应用进行独立解耦管理的关键点。

在Kubernets中，Pod作为基本的执行单元，它可以拥有多个容器和存储数据卷，能够方便在每个容器中打包一个单一的应用，从而解耦了应用构建时和部署时的所关心的事项，已经能够方便在物理机/虚拟机之间进行迁移。API准入控制可以拒绝或者Pod，或者为Pod添加额外的调度约束，但是Kubelet才是Pod是否能够运行在特定Node上的最终裁决者，而不是scheduler或者DaemonSet。kubelet默认情况使用cAdvisor进行资源监控。负责管理Pod、容器、镜像、数据卷等，实现集群对节点的管理，并将容器的运行状态汇报给Kubernetes API Server。

### Container Runtime（容器运行时）

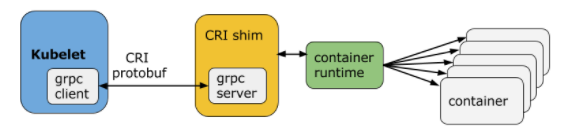
每一个Node都会运行一个Container Runtime，其负责下载镜像和运行容器。Kubernetes本身并不停容器运行时环境，但提供了接口，可以插入所选择的容器运行时环境。kubelet使用Unix socket之上的gRPC框架与容器运行时进行通信，kubelet作为客户端，而CRI shim作为服务器。

图1-2 容器运行时工作逻辑

protocol buffers API提供两个gRPC服务，ImageService和RuntimeService。ImageService提供拉取、查看、和移除镜像的RPC。RuntimeSerivce则提供管理Pods和容器生命周期管理的RPC，以及与容器进行交互(exec/attach/port-forward)。容器运行时能够同时管理镜像和容器（例如：Docker和Rkt），并且可以通过同一个套接字提供这两种服务。在Kubelet中，这个套接字通过–container-runtime-endpoint和–image-service-endpoint字段进行设置。Kubernetes CRI支持的容器运行时包括docker、rkt、cri-o、frankti、kata-containers和clear-containers等。

### kube proxy（代理）

基于一种公共访问策略（例如：负载均衡），服务提供了一种访问一群pod的途径。此方式通过创建一个虚拟的IP来实现，客户端能够访问此IP，并能够将服务透明的代理至Pod。每一个Node都会运行一个kube-proxy，kube proxy通过iptables规则引导访问至服务IP，并将重定向至正确的后端应用，通过这种方式kube-proxy提供了一个高可用的负载均衡解决方案。服务发现主要通过DNS实现。

在Kubernetes中，kube proxy负责为Pod创建代理服务；引到访问至服务；并实现服务到Pod的路由和转发，以及通过应用的负载均衡。

## kubectl（命令行工具）

kubectl是Kubernetes集群的命令行接口。运行kubectl命令的语法如下所示：

$ kubectl [command] [TYPE] [NAME] [flags]

这里的command，TYPE、NAME和flags为：

* **comand**：指定要对资源执行的操作，例如create、get、describe和delete
* **TYPE**：指定资源类型，资源类型是大小学敏感的，开发者能够以单数、复数和缩略的形式。例如：

$ kubectl get pod pod1

$ kubectl get pods pod1

$ kubectl get po pod1

* **NAME：指定资源的名称，名称也大小写敏感的。如果省略名称，则会显示所有的资源，例如:**

 $kubectl get pods

* **flags**：指定可选的参数。例如，可以使用-s或者–server参数指定Kubernetes API server的地址和端口。

另外，可以通过运行kubectl help命令获取更多的信息。

## Add-On（附加项和其他依赖）

在Kunbernetes中可以以附加项的方式扩展Kubernetes的功能，目前主要有网络、服务发现和可视化这三大类的附加项，下面是可用的一些附加项：

**1）网络和网络策略**

* ACI 通过与Cisco ACI集成的容器网络和网络安全。
* Calico 是一个安全的3层网络和网络策略提供者。
* Canal 联合Fannel和Calico，通过网络和网络侧。
* Cilium 是一个3层网络和网络侧插件，它能够透明的加强HTTP/API/L7 策略。其即支持路由，也支持overlay/encapsultion模式。
* Flannel 是一个overlay的网络提供者。

**2）服务发现**

* CoreDNS 是一个灵活的，可扩展的DNS服务器，它能够作为Pod集群内的DNS进行安装。
* Ingress 提供基于Http协议的路由转发机制。

**3）可视化&控制**

* Dashboard 是Kubernetes的web用户界面。

# Kubernetes功能逻辑架构(2019-06-18 Y)

在了解了Kubernetes整体的架构后，在此章节将会以逻辑分层的形式对Kubernetes的核心功能进行阐述。首先从整体上介绍Kubernetes的逻辑层级，从底向上包括核心层、应用层、治理层、接口层和生态层；在整体介绍后，就就每一层所提供的功能和能力进行讲解。

## **整体介绍**

作为一个平台，Kubernetes与其它平台一样，都会遇到一个不可回避的问题，即这个平台应该包含什么以及不包含什么。作为一个部署和管理容器的平台，Kubernetes不能也不应该试图解决用户的所有问题。Kubernetes必须提供一些基本功能，用户可以在这些基本功能的基础上运行容器化的应用程序或构建它们的扩展。从逻辑上来看，kubernetes的架构分为如下几个层次：

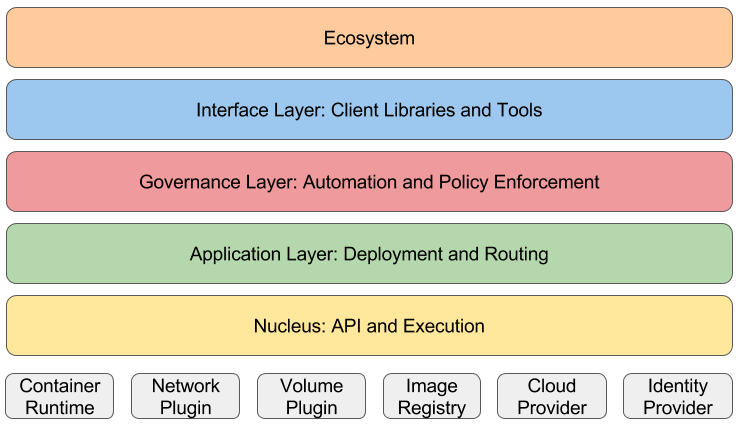


图2-1 Kubernetes功能逻辑架构

* **核心层（Nucleus）： 提供标准的API和执行机，包括基本的REST机制、安全、Pod、容器、网络接口和存储卷管理，通过接口能够对这些API和执行机进行扩展。核心层是必需的，它是系统最核心的一部分。**
* **应用管理层（Application Management Layer )：提供基本的部署和路由，包括自愈能力、弹性扩容、服务发现、负载均衡和流量路由。此层即为通常所说的服务编排，这些功能都提供了默认的实现，但是允许进行一致性的替换。**
* **治理层（The Governance Layer）：提供高层次的自动化和策略执行，包括单一和多租户、度量、智能扩容和供应、授权方案、网络方案、配额方案、存储策略表达和执行。这些都是可选的，也可以通过其它解决方案实现。**
* **接口层（The Interface Layer）：提供公共的类库、工具、用户界面和与Kubernetes API交互的系统。**
* **生态层（The Ecosystem）：包括与Kubernetes相关的所有内容，严格上来说这些并不是Kubernetes的组成部分。包括CI/CD、中间件、日志、监控、数据处理、PaaS、serverless/FaaS系统、工作流、容器运行时、镜像仓库、Node和云提供商管理等。**

## **逻辑层级**

就像Linux拥有内核（kernel）、核心系统类库、和可选的用户级工具，Kubernetes也拥有自己的功能和工具层次。对于开发者来说，理解这些层次是非常重要的。

### ****核心层：API和执行****

**在核心层中，包含了Kubernetes最核心的API和执行机。**这些API和功能由Kubernetes的代码库所实现，是由最小的特性集和概念集所组成，这些特征和概念是系统上层所必需的。这些由Kubnetes代码库所实现的API和函数，包括了建立高阶层功能所需的最小特征集和概念集，每个容器化的应用都会使用它们。

#### ****API和集群控制面板****

在Kubernetes集群中提供了类似REST的API集合，并通过Kubernetes API Server对外进行暴露，以支持持久化资源的增删改查操作。作为系统的最底层，**核心层**支持必要的扩展机制，以支持在上层中定制功能。另外，也需要支持单租户和多租户的应用场景。在核心层需要提供足够的弹性，以支持上层能够扩展新的范围，以保证在安全模式方面不需要进行妥协。如果没有下面这些基础的API机和语义，Kubernetes将不能够正常工作。

* **Authentication API: 认证机制是非常关键的一项工作，在Kubernetes中需要通过服务器和客户端双方的认证。API server 支持基本认证模式 (用户命名/密码) (注意，在将来会被放弃)， X.509客户端证书模式，OpenID连接令牌模式，和不记名令牌模式。通过kubeconfig支持，客户端能够使用上述各种认证模式。另外，第三方认证系统也可以实现TokenReview API，并通过配置认证webhook来进行调用，通过非标准的认证机制可以限制可用客户端的数量。**
* [**Authorization**](https://kubernetes.io/docs/admin/authorization/) **API：负责进行资源和功能的授权管理，第三方授权系统可以实现SubjectAccessReview API，并通过配置授权webhook进行调用。**
* **REST 语义、监控、持久化和一致性保证、API版本控制、违约、验证。**
* **Endpoints API：用于管理端点的接口，API服务器端点的自我发布，高可用和应用层目标发行。**
* **Namespace API：用于管理用户资源的范围的接口，命名空间生命周期(例如：大量删除)。**
* **Event API：用于管理对重大事件的发生进行报告的接口，例如状态改变和错误，以及事件垃圾收集。**
* **API server：作为集群的网关，根据定义API server必需能够被集群外的客户端访问，而Node和Pod是不被集群外的客户端访问的。客户端认证API server，并使用API server作为堡垒和代理。**

理想情况下，核心层API server仅仅支持最小的必需的API，额外的功能通过聚合、钩子、初始化器、和其它扩展机制来提供。注意，中心化异步控制器以名为Controller Manager的独立进程运行，例如垃圾收集。

#### ****执行****

在Kubernetes中最重要的控制器是kubelet，它是Pod和Node API的主要实现者。在Kubernetes中，提供了用于管理多个容器和存储卷的Pod，Pod作为最基本的执行单元。对于容器来说，从逻辑上来看，Pod就类似于宿主机。Kubelet API包括：

* **Pod API**：作为Kubernetes的执行单元，包括：
  + Pod可行性准入控制基于Pod API中的策略(资源请求、Node选择器、亲和性和反亲和性等)。API准入控制可以拒绝Pod或添加额外的调度约束，但Kubelet才是决定Pod最终被运行在哪个Node上的决定者，而不是Schedulers或DaemonSets；
  + 容器和存储卷语义和生命周期；
  + Pod IP地址分配(每个Pod要求一个可路由的IP地址)；
  + 将Pod连接至一个特定安全范围的机制(i.e., ServiceAccount)；
  + 存储卷来源：
    - emptyDir
    - hostPath
    - secret
    - configMap
    - downwardAPI
  + 子资源：绑定、状态、执行、日志、attach、端口转发、代理；
  + 容器镜像和日志生命周期。
* **Secret API**：用于启用第三方加密管理。
* **ConfigMap API**：用于管理组件配置和Pod引用的接口。
* **Node API**：用于管理Pod宿主机的接口，涉及的内容包括：
  + Node和pod网络，业绩它们的控制(路由控制器)；
  + Node库存、健康、和可达性(node控制器)；
  + pod终止垃圾收集；
  + 存储卷控制器。

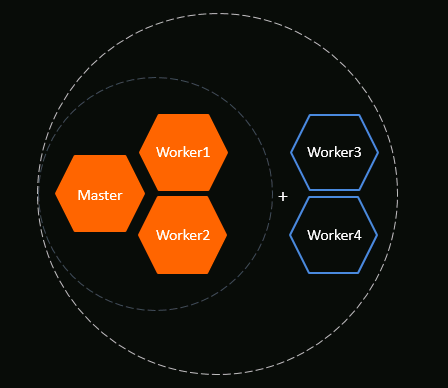


图2-2 主机节点接入

* **PersistentVolume API**：用于管理持久化存储卷的接口
* **PersistentVolumeClaim API**：用于管理持久化存储卷声明的接口

### ****应用层：部署和路由****

在应用层，Kubernetes提供了**自愈、扩容、应用生命周期管理、服务发现、负载均衡和路由的能力**。应用层的API和功能是Kubernetes分发应用所需要的，Kubernetes提供了这些API的默认实现，当然用户也可以使用替代的实现方案。如果没有应用层的API，大部分的容器化应用将不能在Kubernetes上运行。

Kubernetes API提供了类似于IaaS的以容器为中心的基础单元，以及生命周期控制器，以支持所有工作负载的编排(自愈、扩容、更新和终止)。这些应用管理、组合、发现、和路由API和功能包括：

* **容器调度**：在Pod API中实现的调度策略有资源请求、主机选择器、Pod亲和容忍度等。调度可以作为一个独立的进度在集群内或外运行。
* **持续运行**：容器化应用能够通过声明式的更新、级联删除、以及孤儿/领养的方式支持发布(回滚)。除了DaemonSet，其它类型应用都能够支持水平扩容。
* **发现、负载均衡和路由**：
  + **Service API：**用于管理服务的接口，包括集群IP地址分配，修复服务分配映射，通过kube-proxy或者对等的功能实现服务的负载均衡，自动化创建端点，维护和删除。
  + **Ingress API**：用于管理负载均衡的接口。
  + **服务DNS**：DNS使用[官方Kubernetes的模式](https://git.k8s.io/dns/docs/specification.md)。

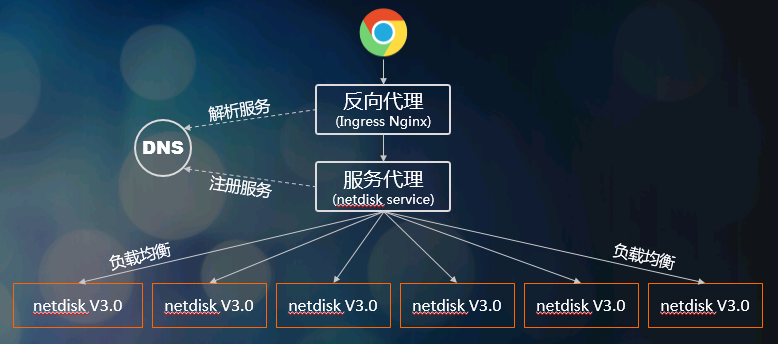


图2-3 服务注册和发现

### ****治理层：自动化和策略执行****

**治理层提供高层次的自动化和策略执行，包括单一和多租户、度量、智能扩容和供应、授权方案、网络方案、配额方案、存储策略表达和执行。这些特性都是可选的，因此可以通过其它解决方案进行实现。在Kubernetes中，治理层主要提供的功能包括**：

* **租户和集群**：
  + 单一租户/单一用户集群；
  + 多租户集群；
  + 生产和开发集群；
  + 用于将计算/应用服务转售给他人的分段集群。
* **资源使用**：
  + CPU的使用情况；
  + 内存的使用情况；
  + 磁盘的使用情况；
  + 网络的使用情况。
* **用户管理**：
  + 最终用户；
  + 管理员。
* **服务质量**（DoS）
* **自动化APIs和功能**：
  + 度量APIs (水平/垂直自动扩容的调度任务表)；
  + 水平Pod自动扩容API；
  + 集群自动化扩容和Node供应；
  + 动态存储卷供应；
  + 动态负载均衡供应。

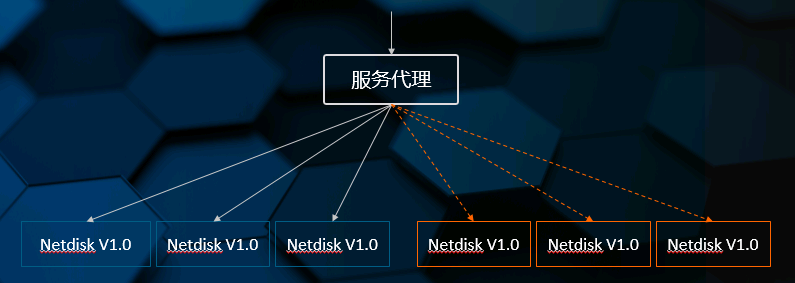


图2-4 集群和负载均衡

* **策略APIs和功能**：
  + 授权：ABAC和RBAC授权策略方案；
  + LimitRange API：用于管理资源的接口；
  + ResourceQuota API：用于管理资源配额的接口；
  + PodSecurityPolicy API：用于管理Pod安全策略的接口；
  + ImageReview API：用于管理镜像审核的接口；
  + NetworkPolicy API：用于管理网络策略的接口。

### ****接口层：类库和工具****

接口层包括Kubernetes官方项目所开发的通用的类库、工具、系统和界面。通过接口层的类库和工具，用户能够调用Kubernetes提供的功能，对Kubernetes进行定制开发；与Kubernetes进行通讯，对Kubernetes集群进行管理和部署容器化应用。接口层提供的核心工具或功能包括：

* **kubectl：kubectl作为Kubernetes众多客户端工具中的一种，**是Kubernetes集群的命令行接口工具，通过kubectl基本上能够进行Kubernetes的所有管理能力**。**
* **Dashboard：即Kubernetes的界面控制台，即通过图形化的界面对Kubernetes进行管理。**
* **Helm：Helm是**装部署容器化应用的一个客户端工具。通过helm，能够帮助开发者定义、安装和升级Kubernetes中的容器化应用。同时，也可以通过helm进行容器化应用的分享。
* **客户端类库：包括client-go和client-python等语言的类库，通过这些类库，可以通过程序与Kubernetes进行通讯，或对Kubernetes进行管理和二次开发。**

### ****生态层：**外部系统提供的功能**

Kubernetes本身有明确的功能界限定义，所包含的功能范畴为提供部署和管理容器化应用所必需通用功能的集合。作为一般规则，在对Kubernetes通用编排功能进行补足的领域，用户可以自行进行选择，这些补足的功能即作为Kubernetes生态体系。Kubernetes提供插件API，并公开多个后端实现的通用API，用于支持用户进行补足功能的实现和接入。

此外，如果开发的功能要考虑成为Kubernetes的一部分，这些开发的组件就需要遵循Kubernetes设计约定。例如，如果所设计用来支持多平台的解决方案没有遵循Kubernetes API协议，就可能不会被认为是Kubernetes的一部分。根据当前的情况来看，常见的Kubernetes生态内容包括：

* **持久化集成和部署(CI/CD)：Kubernetes不提供从源代码到镜像的应用研发生命周期的管理能力。Kubernetes 本身即不部署源代码也不构建应用，用户和项目可以根据自身的需要选择持久化集成和持久化部署的工作流，Kubernetes的目标是方便CI/CD的使用，而不是命令它们如何工作。**
* **应用中间件：应用中间件不作为Kubernetes内置的基础设施，例如：消息队列和SQL数据库等。然而，会结合用户的实际的需要，在Kubernetes中会提供这些通用的应用以功能上层的容器化应用使用。**
* **日志和监控：Kubernetes本身并不提供日志聚合和综合应用监控的能力，也没有监控分析和警报系统，虽然日志和监控的机制是Kubernetes集群必不可少的部分。**
* **数据处理平台：在数据处理平台方面，Spark和Hadoop是还有名的两个例子，但市场中还存在很多其它的系统。**
* [**Kompose**](https://github.com/kubernetes-incubator/kompose)：Kompose是一个适配器工具，它有助于将容器云应用从Docker Compose迁移到Kubernetes ，并提供简单的用例。Kompose不遵循Kubernetes约定，而是基于手动维护的DSL。
* **容器运行时**：Kubernetes本身不提供容器运行时环境，但是其提供了容器运行时接口（CRI），可以来插入所选择的容器运行时。
* **镜像仓库**：Kubernetes本身不提供容器的镜像，可通过Harbor、Nexus和docker registry等搭建镜像仓库，以为集群拉取需要的容器镜像。
* **集群状态存储**：用于存储集群运行状态，例如默认使用Etcd，但也可以使用其它存储系统。
* **网络**：与容器运行时一样，Kubernetes提供了接入各种网络插件的容器网络接口（CNI）。
* **文件存储**：对于容器云来说，其自身不提供数据的持久化存储能力，存储需要依赖于外部的本地文件系统和网络存储，在Kubernetes中也提供了存储的接口（CSI）。
* **Node管理**：Kubernetes既不提供也不采用任何综合的机器配置、维护、管理或自愈系统。通常针对不同的公有/私有云，针对不同的操作系统，针对可变的和不可变的基础设施。
* **集群创建和管理**：Kubernetes社区已经开发了很多的工具，利润minikube、kubeadm、bootkube、kube-aws、kops、kargo, kubernetes-anywhere等待。 从工具的多样性可以看出，集群部署和管理（例如，升级）没有一成不变的解决方案。也就是说，常见的构建块（例如，安全的Kubelet注册）和方法（特别是自托管）将减少此类工具中所需的自定义编排的数量。

当然，除了上述的内容以外，根据用户和项目的需要会存在其它的生态功能。

# Kubernetes离线环境搭建(2019-05-16 Y)

由于各种原因和现实约束，像航天等企业的网络环境是与外部的互联网隔离的。因此，为了能够针对这些企业提供容器云服务，就需要提供容器云的离线部署能力。相对于能够连接互联网的环境，在局域网的环境下安装会相对复杂一些。

## 环境准备

在进行Kubernetes部署之前，需要提供符合要求软硬件的环境。首先，Kubernetes对于每一个主机节点的操作系统、操作系统版本、CPU核数、内存大小和磁盘空间都相应的要求。另外，本书所描述的Kubernetes集群由1个Master节点和3个Worker节点所组成，并基于Rancher进行Kubernetes集群的安装部署。

### 操作系统

对于操作系统和对应Docker版本要求如下：

* Ubuntu 16.04.x (64-bit)
  + Docker 17.03.x, 18.06.x, 18.09.x
* Ubuntu 18.04.x (64-bit)
  + Docker 18.06.x, 18.09.x
* Red Hat Enterprise Linux (RHEL)/CentOS 7.5+ (64-bit)
  + RHEL Docker 1.13
  + Docker 17.03.x, 18.06.x, 18.09.x

下表是在本文中部署Kubernetes所需要的硬件资源：

表3-1 硬件资源

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 序号 | 主机 | 角色 | 软硬件环境 | 硬件配置 | 备注 |
| 1 | k8s-master | 主节点 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：500GB * 所有磁盘：1TB | 部署Kunbernetes集群主节点和etcd，用于管理和监控Kubernetes其它的工作节点和存在状态信息。 |
| 2 | k8s-worker01 | 工作节点 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：500GB * 所有磁盘：1TB | 部署Kubernetes集群的工作节点，用于运行容器化的应用。 |
| 3 | k8s-worker02 | 工作节点 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：500GB * 所有磁盘：1TB | 部署Kubernetes集群的工作节点，用于运行容器化的应用。 |
| 4 | k8s-worker03 | 工作节点 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：500GB * 所有磁盘：1TB | 部署Kubernetes集群的工作节点，用于运行容器化的应用。 |
| 5 | nfs-server | NFS服务器 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * NFS：nfs v4 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：1TB * 所有磁盘：2TB | 部署nfs服务，用于为上层所有的应用提供持久化存储。 |
| 6 | registry-server | 私有镜像仓库 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 * 镜像仓库：sonatype/nexus3 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：1TB * 所有磁盘：2TB | 部署镜像仓库，用于提供镜像的存储和拉取。 |
| 7 | kubectl/helm | 工具节点 | * 操作系统：Windows * 命令行工具：kubectl * 包安装工具：helm | * CPU：4核 * 内存：8GB * 磁盘：500GB |  |
| 8 | public | 下载资源 | * 操作系统：CentOS Linux release 7.2.1511 (Core) * 容器环境：Docker version 1.13.1 | * CPU：4核 * 内存：8GB * 磁盘： * 根磁盘：500GB * 所有磁盘：1TB |  |

### ****设置防火墙策略****

**对Kubernetes的部署环境来说，需要开放一些端口，具体端口信息如下：**

**表3-2 端口**

|  |  |  |
| --- | --- | --- |
| 协议 | 端口 | 描述 |
| TCP | 80 | Rancher UI/API when external SSL termination is used |
| TCP | 443 | Rancher agent, Rancher UI/API, kubectl |
| TCP | 6443 | Kubernetes apiserver端口 |
| TCP | 22 | SSH provisioning of nodes using Node Driver |
| TCP | 2379 | Etcd客户端请求端口 |
| TCP | 2380 | Etcd点对点通信端口 |
| UDP | 8472 | Canal/Flannel VXLAN overlay网络端口 |
| TCP | 10250 | Kubelet端口 |
| TCP/UDP | 30000-32767 | NodePort端口 |
| TCP | 8081 | Nexus端口 |
| TCP | 5001 | Registry端口 |

**如果是刚开始试用，可以通过执行下面的命令关闭防火墙：**

$ systemctl stop firewalld

Ubuntu默认未启用UFW防火墙，无需设置。也可手工关闭：

$ sudo ufw disable

### 配置主机时间、时区、系统语言

这部分需要为部署Kubernetes集群的各集群配置正确的主机时间、时区和系统语言。

1. 查看时区

查看系统当前的时区信息：

date –R

或者

timedatectl

1. 修改时区

将时区修改为亚洲上海时区：

ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

1. 修改系统语言环境

将系统语言设置成en\_US.UTF-8:

sudo echo 'LANG="en\_US.UTF-8"' >> /etc/profile

source /etc/profile

### 环境清理（可选）

如果在宿主机上已经安装部署过Kubernetes，则需要先对环境进行清理，清理的步骤如下：

1）请检查有没有/var/lib/rancher/state/这个文件夹，如果有则删除；

2）如果以前已经通过Rancher安装过Kubernetes，请执行命令:

# 删除所有的容器

$ docker rm -f -v $(docker ps -aq)

# 删除已有的存储卷

$ docker volume rm $(docker volume ls)

# 删除遗留的目录

$ rm -rf /etc/kubernetes/ssl

$ rm -rf /var/lib/etcd

$ rm -rf /etc/cni

$ rm -rf /opt/cni

$ rm -rf /var/run/calico

## 安装介质下载和准备

由于是离线安装，银座在开始安装部署前，需要先准备好安装介质。这些安装介质包括docker、私有镜像仓库、Rancher、kubectl工具和helm工具。

**1）docker**

下载docker的安装文件：

$ wget <https://download.docker.com/linux/static/stable/x86_64/docker-17.03.2-ce.tgz>

**2）私有镜像仓库**

下载nexus3的镜像，后续将以nexus3作为私有镜像仓库：

$ docker pull sonatype/nexus3:latest

$ docker save sonatype/nexus3:latest > nexus3.tar

**3）rancher镜像**

下载拉取镜像的脚本：rancher-save-images.sh和上传镜像至镜像仓库的脚本：rancher-load-images.sh。

$ wget https://github.com/rancher/rancher/releases/tag/v2.0.0/rancher-save-images.sh

$ wget https://github.com/rancher/rancher/releases/tag/v2.0.0/rancher-load-images.sh

通过执行rancher-save-images.sh拉取镜像：

$ . rancher-save-images.sh

此脚本用以下载部署时所需的所有镜像，并将这些镜像压缩到rancher-images.tar.gz中。

**4）kubectl**

下载在windows下使用的[kubectl](http://docs.kubernetes.org.cn/61.html)工具：

$ wget https://storage.googleapis.com/kubernetes-release/release/v1.9.0/bin/windows/amd64/kubectl.exe

**5）helm**

下载helm客户端，此处下载的是Windows下的2.8.0版本：

$ wget https://storage.googleapis.com/kubernetes-helm/helm-v2.8.0-windows-amd64.tar.gz

下载helm服务端tiller，此处下载的是Windows下的2.8.0版本：

$ docker pull rancher/tiller:v2.8.2

$ docker save rancher/tiller:v2.8.2 > tiller.tar

## Docker安装部署

Kubernetes运行时需要一个容器引擎，默认情况下容器引擎采用Docker。

### Docker安装

由于是离线安装，在本书中下载的是Docker的静态二进制文件。使用Docker为17.03.2版本，静态二进制文件的压缩包是docker-17.03.2-ce.tgz。此文件的下载地址是<https://download.docker.com/linux/static/stable/x86_64/>，安装过程如下所示：

1）通过wget下载安装文件

$ wget <https://download.docker.com/linux/static/stable/x86_64/docker-17.03.2-ce.tgz>

2）解压缩文件

$ tar xzvf /{path}/docker-17.03.2-ce.tgz

3）将二进制文件移动到可执行文件路径上的目录中，例如/usr/bin/。

$ cp docker/\* /usr/bin/

4）启动Docker守护程序

通过执行dockerd命令来启动Docker守护程序，当然如果需要的话，可以添加其他启动参数。

$ dockerd &

5）验证安装结果

通过执行docker info命令来验证安装是否正确，如果返回信息没有问题，则表示安装没有问题。

$ docker info

### 设置根目录(可选)

在docker安装成功后，通过执行如下的命令可以查看docker的信息：

$ docker info

默认情况，docker的根目录为/var/lib/docker，它将会占据大量的磁盘空间。因此需要预先为其提供足够的磁盘空间，此处为其挂接一块专用的磁盘。假设这里存在一个新增的/dev/vdc磁盘。

1）创建新的专用的根目录

$ mkdir /docker-root

2）将磁盘挂接至新的根目录

$ mount /dev/vdc /docker-root

3）设置挂接永久有效

$ echo “/dev/vdc  /docker-root  ext4   defaults  0 0”  >  /etc/fstab

4）将docker设置为使用新的根目录

$ vi /etc/docker/daemon.json

添加：”graph": "/docker-root"

"graph":"/docker-root"

5）重启docker

$ systemctl daemon-reload

$ systemctl restart docker

### 配置远程访问

由于docker本身的架构就是服务器端/客户端模式，因此可以通过远程访问docker的服务。但为了安全考虑，默认情况下是不允许远程访问的。

1. 在/etc/docker/daemon.json文件中添加如下的信息：

{

"hosts": ["unix:///var/run/docker.sock", "tcp://127.0.0.1:2375"]

}

1. 通过执行下面命令重启docker：

$ systemctl daemon-reload

$ systemctl restart docker

1. 通过执行执行下面的命令，检查所配置的远程访问是否生效：

$ sudo netstat -lntp | grep dockerd

tcp 0 0 127.0.0.1:2375 0.0.0.0:\* LISTEN 3758/dockerd

1. 远程的机器可以通过执行下面的命令来访问docker：

$ sudo docker –H {ip}:2375 info

### 设置docker开机启用

如果需要设置docker开机就启用的话，可以通过执行如下的命令进行设置。

$ sudo systemctl enable docker

### docker卸载

对于老版本的Docker，通过执行下面的命令进行卸载：

$ sudo yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

对于新版本的Docker，通过执行下面的命令进行卸载：

$ sudo yum remove docker-ce

在执行完卸载操作后，可以通过执行下面的命令检查卸载是否成功：

$ sudo yum list installed | grep docker

卸载完成后，通过执行下面的命令移除相关数据：

$ sudo rm -rf /var/lib/docker

/var/lib/docker/目录下包含镜像、数据卷、容器、网络等相关内容。

## 提供网络存储(可选)

由于容器本身只承载应用，并不提供进行数据持久化的能力。因此，需要为容器云中的应用提供存储技术和手段，在本书中采用nfs网络存储。

#### ****配置共享目录****

在使用nfs时，首先需要在nfs服务器中为客户端配置可使用共享目录。在此处，为容器创建了名称为nfs-share的根目录，并在根目录下创建了一个名称为docker-registr的目录。另外，需要为共享目录设置访问权限。

# 创建共享根目录

$ mkdir /nfs-share

# 创建私有镜像仓库目录

$ mkdir /nfs-share/docker-registry

$ echo "/nfs-share \*(rw,async,no\_root\_squash)" >> /etc/exports

设置完成后，通过执行如下命令使共享目录的配置生效：

$exportfs -r

#### ****启动服务****

1）启动rpcbind服务：这里须先启动rpcbind服务，再启动nfs服务，才能让nfs服务在rpcbind服务上注册成功。

$ systemctl start rpcbind

2）启动nfs服务：在启动rpcbind服务，通过执行如下的命令启动nfs-server。

$ systemctl start nfs-server

3）设置rpcbind和nfs-server开机启动：为了便于允许和维护，通过执行下面的命令，使rpcbind和nfs-server能够随着机器开机自动启动。

$ systemctl enable rpcbind

$ systemctl enable nfs-server

#### ****检查nfs服务状态****

通过上述的工作完成nfs服务的部署后，在被容器云正常使用前，我们需要验证一下服务的运行状态。

$ showmount -e localhost

$ mount -t nfs 127.0.0.1:/data /mnt

## 私有镜像仓库安装部署

对于局域网的容器云来说，就需要一个私有的镜像仓库，此镜像仓库用于管理企业的镜像。在此文中，采用nexus作为容器云的私有镜像仓库。

**1）导入镜像**

复制nexus.tar文件到需要安装镜像仓库的机器，并通过docker load命令导入镜像：

$ docker load < nexus.tar

**2）设置存储目录**

为nexus创建持久化目录，并挂接nfs的共享目录：

$ mkdir /mnt/nexus-data && chmod 777 /mnt/nexus-data

$ mount -t nfs {nfs-server}:/nfs-share/docker-registry /mnt/nexus-data

**3）运行私有镜像仓库**

运行nexus3容器，并8081端口和5001端口，5001端口为docker私有镜像仓库的对外端口：

$ docker run -d -p 8081:8081 -p 5001:5001 -v /mnt/nexus-data:/nexus-data --name nexus sonatype/nexus3

**4）创建docker镜像仓库**

在nexus3中创建一个名称为docker的镜像仓库，端口为5001。

## Kubernetes安装部署

拷贝tiller.tar、rancher-images.tar.gz和rancher-load-images.sh到安装rancher服务的机器上。

1）上传rancher相关镜像至私有镜像仓库

执行rancher-load-images.sh：

$ . rancher-load-images.sh

系统会导入所有的镜像，并将其打上私有镜像仓库的标签，并上传至私有镜像仓库。

2）上传tiller镜像至私有镜像仓库

# 导入tiller镜像

$ docker load < tiller.tar

# 将tiller打上私有镜像的标签

$ docker tag rancher/tiller:v2.8.2 {registry-ip}/rancher/tiller:v2.8.2

# 上传至私有镜像仓库

$ docker push 10.10.30.190:5001/rancher/tiller:v2.8.2

### 安装rancher服务

通过执行docker run的命令，进行rancher服务的安装：

$ sudo docker run -d --restart=unless-stopped -p 80:80 -p 443:443 {registry-ip}/rancher/rancher:2.0.0

在rancher部署完成后，可以通过下面的步骤登录Kubernetes，并进行管理员密码、访问地址和私有镜像仓库地址的设置。

**1）登录rancher**

在rancher服务正常启动后，通过浏览器访问rancher。

**2）设置管理员密码**

在此次登录时，根据提示设置管理员的密码。

**3）设置访问地址**

在设置好管理员的密码后，设置rancher的对外提供访问的地址。

**4）设置私有镜像仓库**

在全局下，在系统设置界面中，将system-default-registry设置为本文前面创建的私有镜像仓库，此处为10.10.30.190:5001。



图3-1 设置私有镜像仓库

5）创建集群

进入rancher后，创建名称为demo的custom类型集群。

### 添加节点

在上述工作完成后，接下来进入创建Kubernetes集群的工作中，在本文中为Kubernetes添加一个Master和etcd节点，以及两个Worker节点。

#### 添加Master和etcd

在添加创建集群的页面，选择node角色为“etcd”和“control”，即添加的为Master和etcd节点：

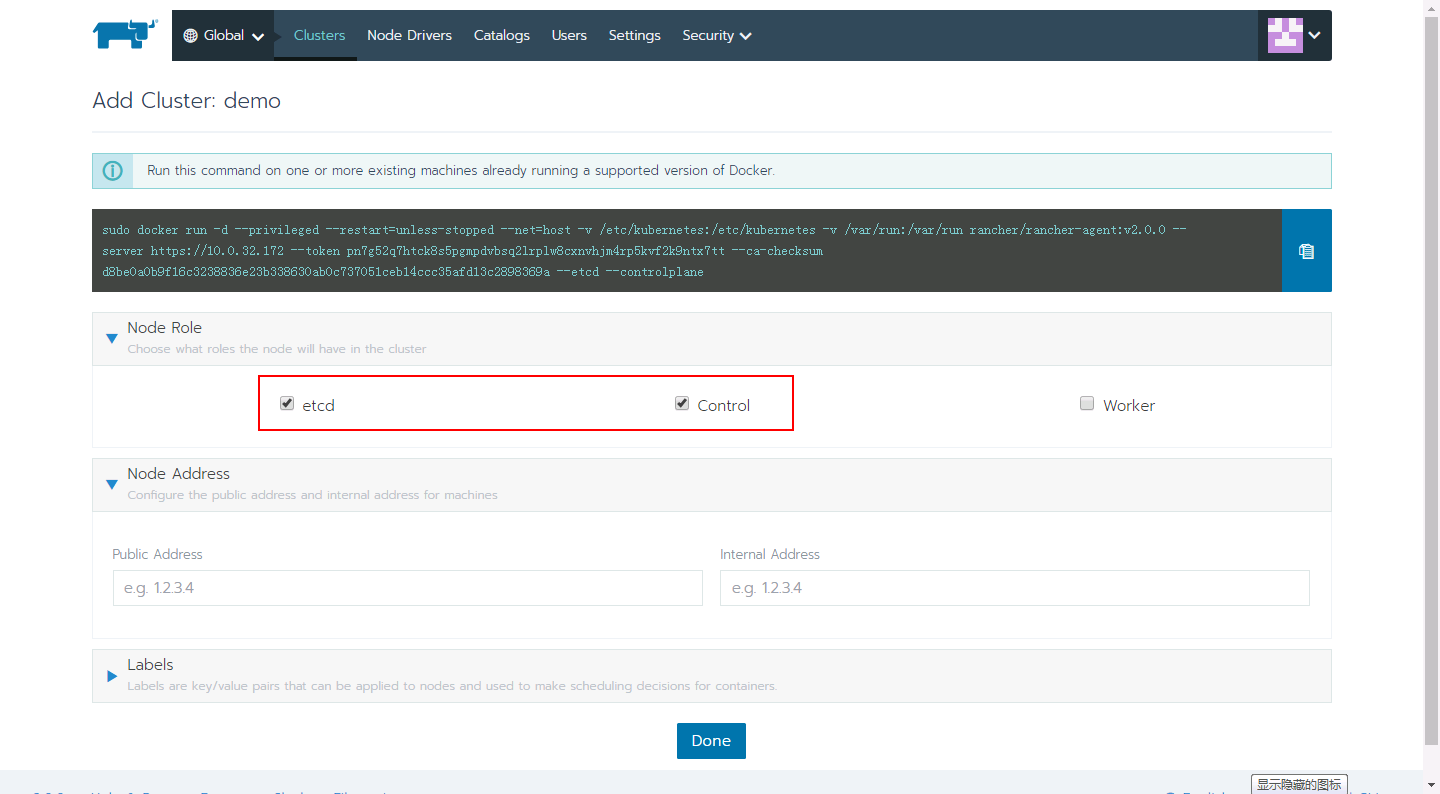


图3-2 添加Master节点

并在集群上执行如下的命令，以将机器添加到集群中作为Master和etcd节点。

$ sudo docker run -d --privileged --restart=unless-stopped --net=host -v /etc/kubernetes:/etc/kubernetes \

-v /var/run:/var/run {registry-ip}/rancher/rancher-agent:v2.0.0 --server https://10.0.32.172 \

--token pn7g52q7htck8s5pgmpdvbsq2lrplw8cxnvhjm4rp5kvf2k9ntx7tt \

--ca-checksum d8be0a0b9f16c3238836e23b338630ab0c737051ceb14ccc35afd13c2898369a --etcd --controlplane

#### 添加worker

在添加创建集群的页面，选择node角色为“worker”，即添加的为Worker节点：

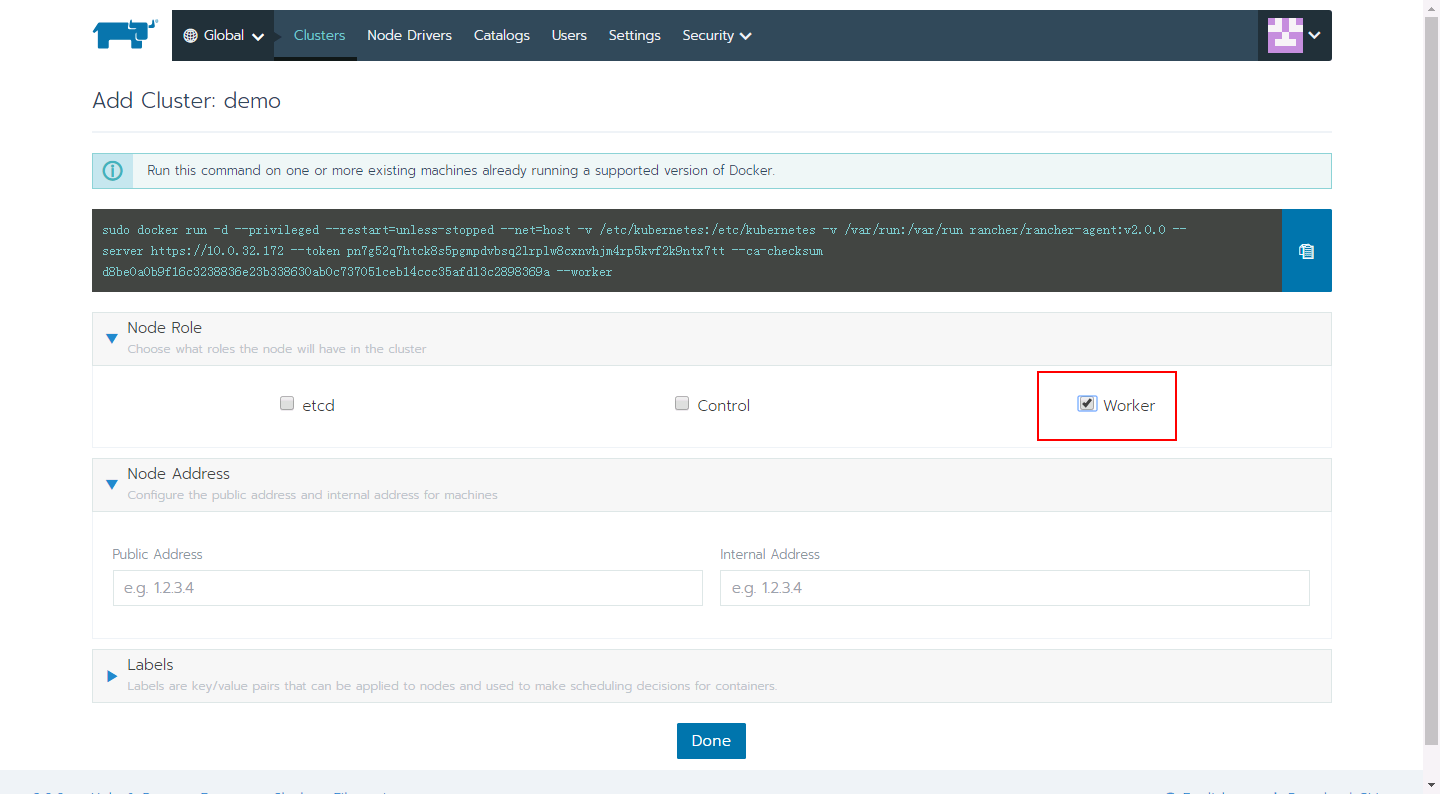


图3-3 添加Worker节点

并在集群上执行如下的命令，以将机器添加到集群中作为Worker节点。

$ sudo docker run -d --privileged --restart=unless-stopped --net=host -v /etc/kubernetes:/etc/kubernetes \

-v /var/run:/var/run {registry-ip}/rancher/rancher-agent:v2.0.0 --server https://10.0.32.172 \

--token pn7g52q7htck8s5pgmpdvbsq2lrplw8cxnvhjm4rp5kvf2k9ntx7tt \

--ca-checksum d8be0a0b9f16c3238836e23b338630ab0c737051ceb14ccc35afd13c2898369a --worker

## 安装kubectl

Kubectl是用于与容器云进行交互的命令行客户端工具，本文将kubectl安装在Windows操作系统中。

1）安装部署kubectl

拷贝kubectl.exec可执行文件到特定的文件夹目录下，并将kubectl.exe所在的文件夹目录地址添加至Windows的环境变量的Path中。

2）配置kubeconfig文件

在当前用户的文件夹目录下，创建./kube文件夹，并创建config文件。rancher系统中，进入所创建的集群的主页，通过点击“Kube config File”进入kubeconfig信息页面。并将kubeconfig文件的内容拷贝到~/.kube/config中。

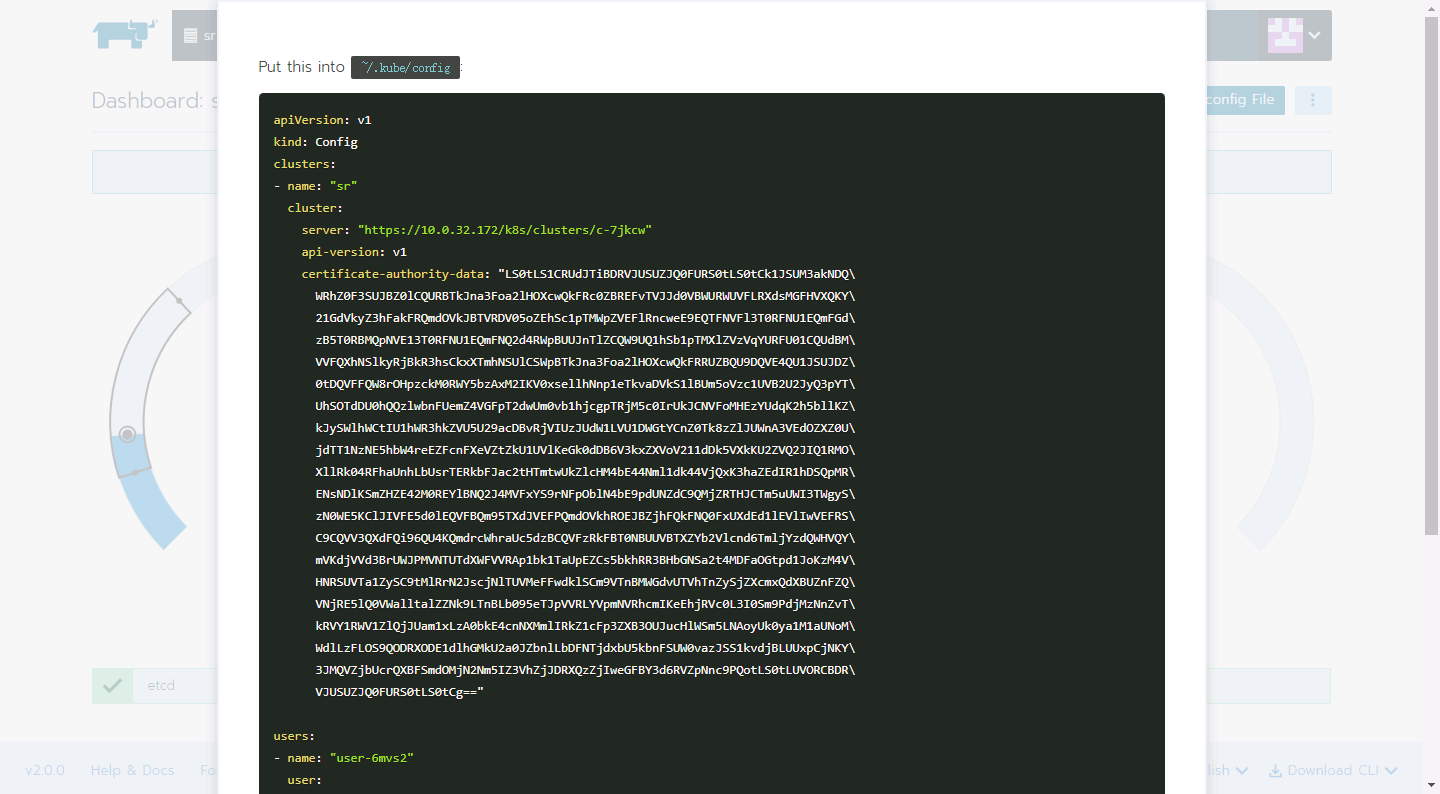


图3-4 kubeconfig文件信息

**3）验证**

通过执行如下的命令，验证kubectl配置是否成功：

$ kubectl get nodes

## 安装helm

由于helm客户端需要通过Kubectl的kubeconfig配置文件与Kubernetes进行交互，为了便于操作和使用，因此与kubectl安装在同一台机器中。

### ****安装Helm客户端****

拷贝helm-v2.8.0-windows-amd64.tar.gz，并将其解压缩到操作系统的本地特定目录地址下，并将helm.exe所在的地址添加至Windows的环境变量的Path中。

在本书中，helm.exe放置的目录地址为：d:/helm。

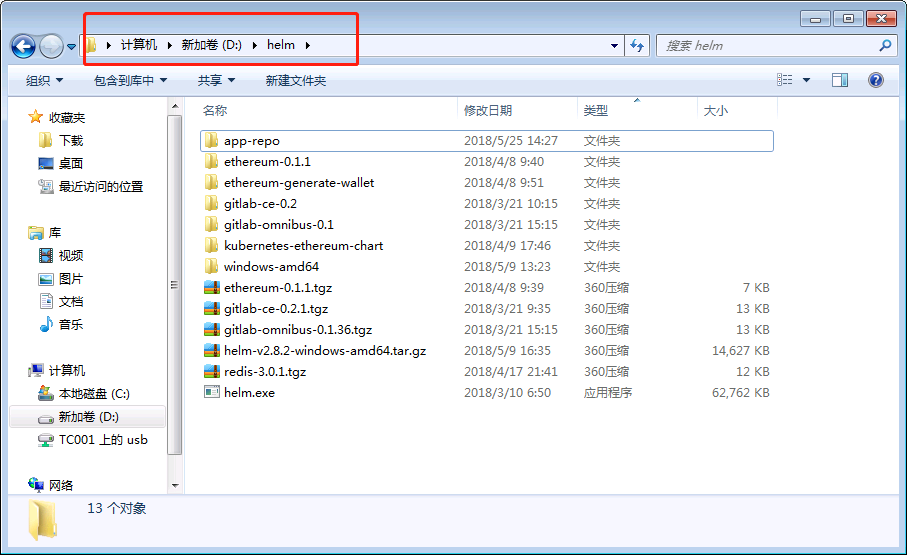


图3-5 helm客户端目录

在Windows的环境变量，将其添加到Path变量的值中，如下图所示：

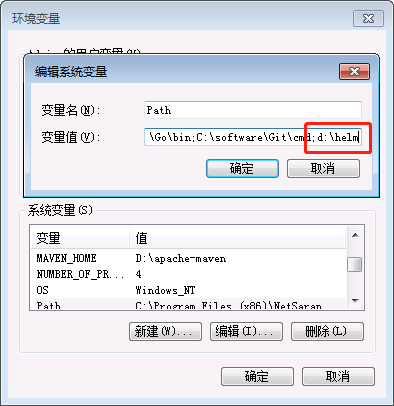


图3-6 设置helm的环境变量

### ****安装Tiller服务器****

Tiller是Helm的服务器端，此处将其部署在Kubernetes中，同时也作为helm安装在Kubernetes中的插件。

1）创建一个名为tiller的Service Account

$ kubectl create serviceaccount tiller --namespace kube-system

2）授予名为tiller的Service Account集群管理员角色cluster-admin：

将tiller绑定至集群管理员角色的的yaml文件如下所示：

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRoleBinding

metadata:

name: tiller

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: tiller

namespace: kube-system

通过执行kubectl create -f将授予tiller集群管理员角色：

$ kubectl create -f rbac-config.yaml

3）安装Tiller服务器

由于是离线安装，先启动本地的chart仓库：

$ helm serve

通过helm init命令在Kubernetes集群中安装Tiller服务器端，在这里通过–service-account字段设置访问用户为tiller，通过–stable-repo-url字段设置所使用的仓库为本地仓库，通过–tiller-image字段指定使用私有镜像仓库中的tiller:v2.8.2镜像。

$ helm init --service-account=tiller --stable-repo-url=http://127.0.0.1:8879 \

-–tiller-image={registry-ip}/rancher/tiller:v2.8.2

### ****验证安装****

在安装完成后，可以通过执行如下命令来检查是安装成功：

$ helm version

如果正确显示Helm客户端和Tiller服务器的版本，这表示安装成功。

或者通过执行kubectl的如下命令来查看是否已正常按照Tiller服务器：

$ kubectl get pods -n kube-system

# Kubernetes快速入门(2019-06-19 Y)

在对Kubernetes有了整体了解和部署了环境之后，本章节将通过demo的方式带领读者快速的入门，这里提供了应用部署、Pod信息获取、对外暴露应用、扩容应用和升级应用的示例。

## 入门概述

本文以在容器云上部署一个nexus3应用为例，通过通过一步步的操作，帮助用户快速的对Kubernetes有一个快速和整体的认识。通过快速入门，可以提供如下知识内容：

* 在集群中部署一个容器的应用
* 对部署的应用进行弹性伸缩
* 使用新版本的软件更新容器化应用
* 对容器话应用进行Debug

在开始之前，应该具备如下的必要条件：

* 具备kubernetes的运行环境；
* 已安装kubectl。

## 步骤1：部署应用

在Kubernetes集群中，底层的计算能力由各个Node节点提供，这些Node节点即可以是物理机、也可以是虚拟机和云主机。节点分为两类，即主节点(Master Node)和从节点(Woker Node)，通过部署可以将容器化的应用部署在Kubernetes集群中。

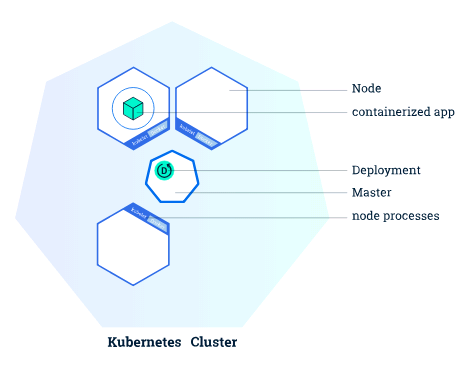


图4-1 在Kubernetes集群中的应用

在Kubernetes中，通过Kubectl使用Kubernetes API和集群进行交互，开发者使用Kubectl（Kubernetes命令行界面）创建和管理部署。在创建部署的步骤，将会学习一些创建Deployment的Kubectl命令，通过执行这些命令，将能在Kubernets集群中部署和运行应用。创建部署，需要为应用指定所使用的容器镜像，以及应用的副本数量。后续，可以通过更新部署来修改相关的信息；在后续的步骤中，也将会讨论如何进行伸缩和升级部署。当然，这里的应用需要提前被打包成支持的容器格式，这样才能够在Kubernetes中进行部署运行，这里以部署nexus3应用作为例子。

1）部署应用

使用sonatype/nexues3:3.9.0镜像创建一个名称为my-nexus3的部署：

$ kubectl create deployment my-nexus3 --image=sonatype/nexus3:3.9.0

2）查看集群中存在的部署

在部署完成后，可以通过如下命令获取在kubernetes中的部署，通过AVAILABLE字段可以可以确认部署是否已经准备就绪：

$ kubectl get deployments

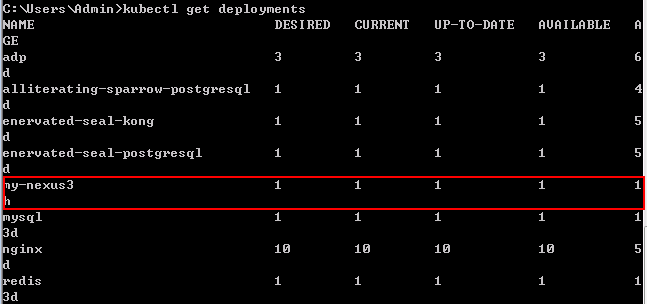


图4-1 集群中应用部署列表

## 步骤2：获取应用的Pod信息

在Kubernetes中，容器通过Pod进行组织管理，在一个Pod中可以管理多个容器和存储卷，每个Pod都有自己在集群中唯一的IP地址。

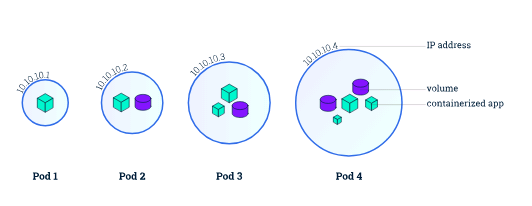


图4-3 在Kubernetes中Pod管理模式

通过kubectl get pods获取集群中的Pods。

1）获取集群中的pod

通过执行kubect get pods命令获取集群中的Pod：

$ kubectl get pods

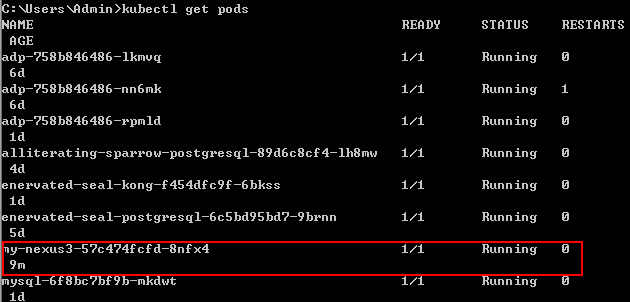


图4-4 集群中的Pod信息

2）获取Pod的详细信息

通过kubectl describe pod命令可以获取Pod的详细信息，但Pod没有正常启动时，可以通过查看Pod的详细信息获取初步的解决方案：

$ kubectl describe pods/my-nexus3-57c474fcfd-8nfx4

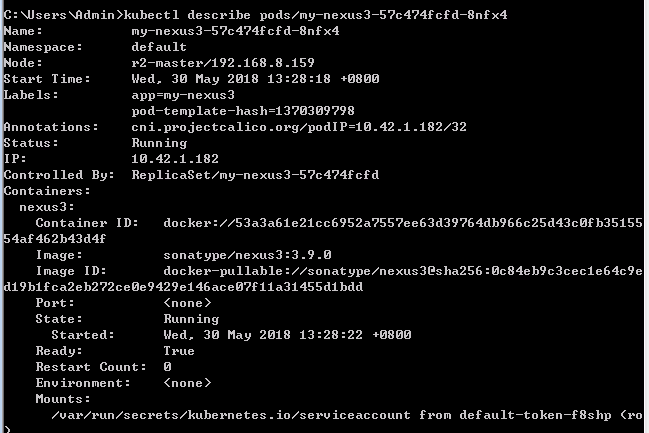


图4-5 Pod的详细信息

3）获取容器的日志信息

另外，开发或运维人员可能需要通过日志获取应用的运行情况，以处理可能存在问题。获取应用运行日志的信息，通过执行kubectl logs命令：

$ kubectl logs my-nexus3-57c474fcfd-8nfx4

## 步骤3：对外暴露应用

在Kubernetes中，服务是Pod的逻辑集合，以及访问这些Pod的策略。与其它Kubernetes对象一样，服务使用YAML文件或JSON进行定义。服务是通过标签选择器来确定使用哪些Pod的，虽然每个Pod都有一个唯一的IP地址，但这些IP地址只在集群内部可用，并不对外暴露。Pod需要通过服务对外暴露，服务支持以下四种对外暴露的类型：

* ClusterIP (default) – 将服务暴露在集群内部的IP，此类型仅支持在集群内访问服务。
* NodePort – 将服务暴露在所选定的每一个Node的同一个端口，集群外可以通过<NodeIP>:<NodePort>方式访问服务。
* LoadBalancer – 在当前的集群中创建一个外部的负载均衡，并为服务（service）指派一个固定的、外部的IP地址。
* ExternalName – 使用一个随意的名称（在规格中指定）来暴露服务，并会返回一个带有名称的CNAME记录。此类型不使用代理，这种类型只在kube-dns v1.7上才支持。

另外，这里的示例没有在规格中定义选择器，对于这种没有定义选择器的服务，也就没有相应的端口对象。这就允许通过手工方式将服务映射到指定的端口。另外，如果使用的暴露类型为ExternalName的话，也不需要选择器。

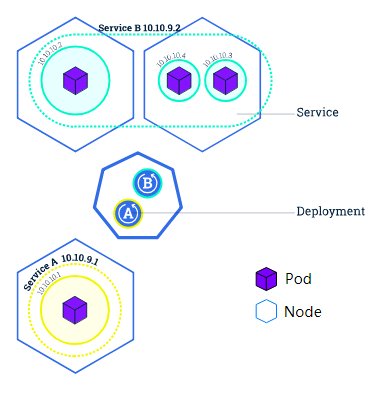


图4-6 通过服务对外暴露应用

此处通过NodePort类型将my-nexus3部署对外进行暴露，暴露部署服务的命令如下：

$ kubectl expose deployments/my-nexus3 --name=nexus3 --type="NodePort" --port=8081

通过执行上述命令，Kubernetes将以nexus3的名称对外暴露服务。然后，可以通过kubectl describe services的命令查询服务的相关信息。

$ kubectl describe services/nexus3

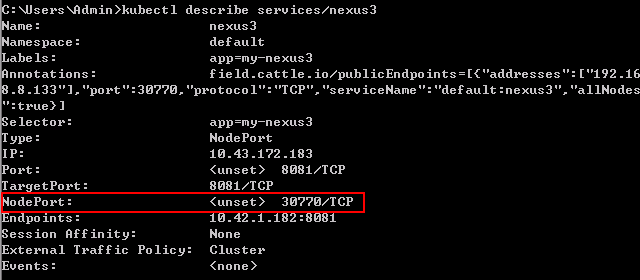


图4-7 服务的详细信息

从执行命令的输出结果可以看出，在每个node上暴露了一个30770端口。在浏览器的地址中访问：http://<nodeip>:30770，将进入nexus的页面：

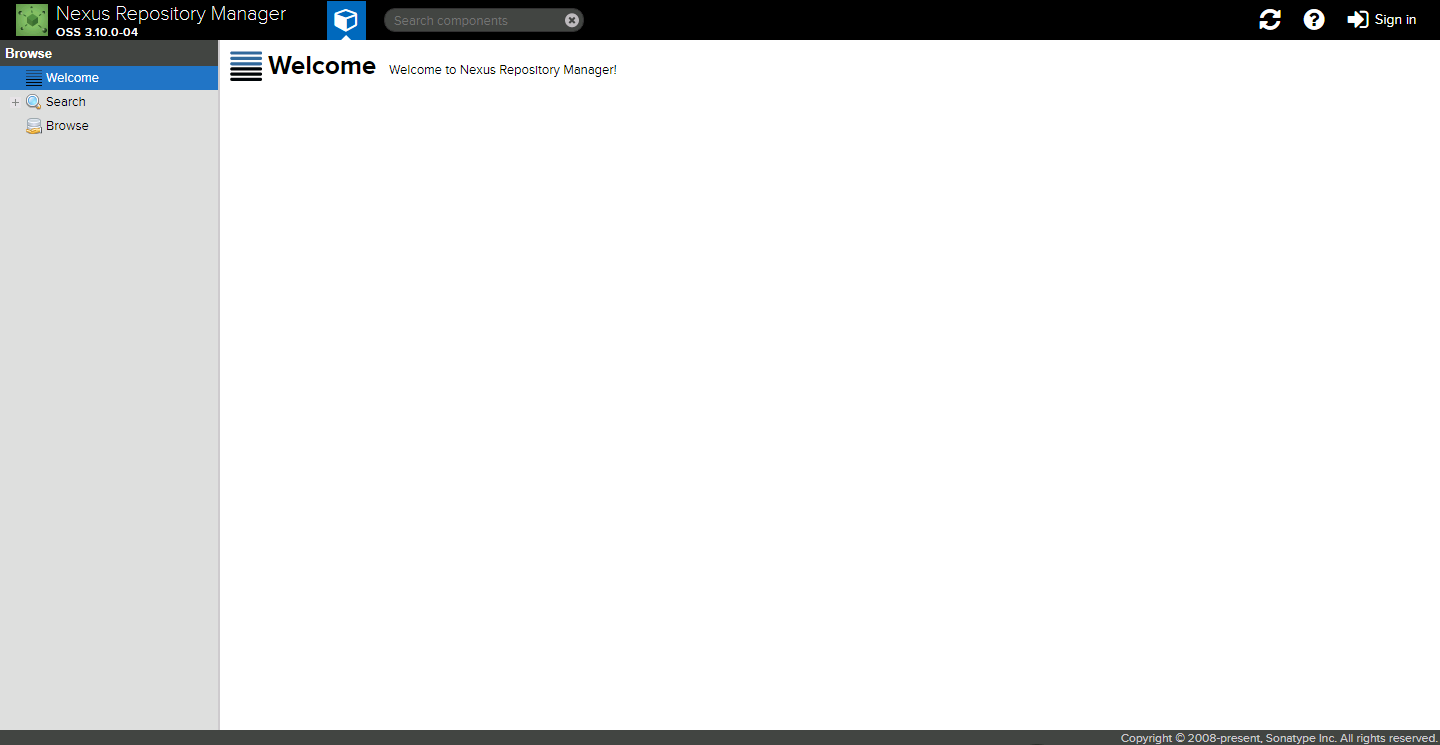


图4-8 nexus主页

## 步骤4：扩缩容应用

根据应用的访问情况，可以对部署进行扩缩容，以提升用户体验和有效利用资源。通过扩容部署，将能够创建新的Pod，以及根据可用的资源情况，将新的Pod调度到合适Node中。通过缩容，将可以需要减少Pod的数量，释放资源供其他应用使用。同时，Kubernetes也支持Pod的自动伸缩。如果运行应用的有多个实例，则需要提供一个进行负载分流的途径。服务集成了负载均衡，能够将网络的流量分流到所部署的各个Pod中。服务将使用端口持续的监控正在运行的Pod，以确保流量会被送到可用的Pod。扩缩容通过修改部署的副本来实现。

下图是扩容前部署、服务和Pod之间的关系，在集群中有一个部署，部署包含一个Pod，并通过服务进行了对外暴露。

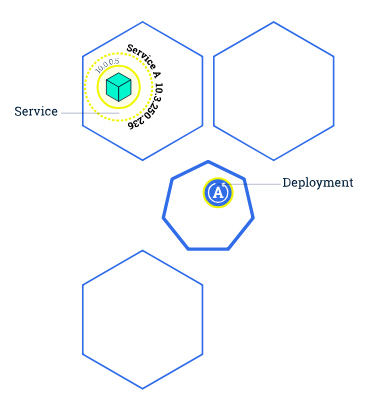


图4-8 扩容前部署、服务和Pod的关系

下图是扩容后的部署、服务和Pod的关系。在原来的基础上，在两个新的Node中扩容了3个Pod，并重新通过服务进行了对外暴露。

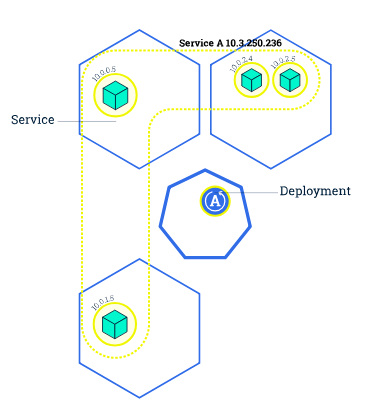


图4-9 扩容后部署、服务和Pod的关系

1）进行扩容

根据场景需要，通过kubectl scale deployment命令将Pod的扩容到4个。

$ kubectl scale deployments my-nexus3 --replicas=4

2）查看扩容后的Pod

在扩容后，通过kubectl get pods能够查看扩容后的Pod数量。

$ kubectl get pods -o wide

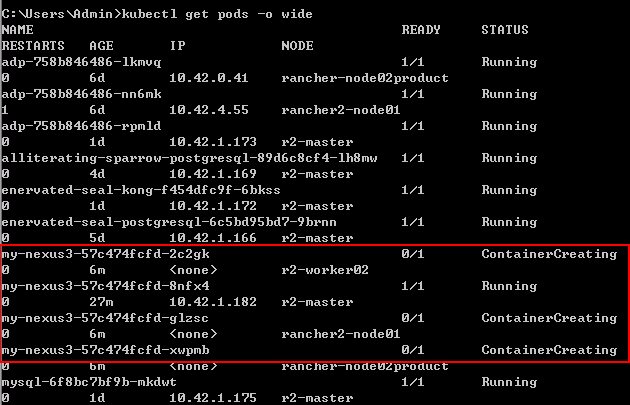


图4-10 集群中Pod的列表信息

3）将扩容后的Pod通过服务进行对外暴露

接下来，使用kubectl expose来创建名为nexus-lb一个服务来暴露扩展后的部署。

$ kubectl expose deployments/my-nexus3 --name=nexus-lb --type="NodePort" --port=8081

4）查看暴露的nodeport

$ kubectl get services/nexus-lb

https://www.kubernetes.org.cn/img/2018/05/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20180530135905.png

图4-11 集群服务的列表信息

其中：NodePort：31400

5）访问应用

在应用通过服务暴露后，可以在浏览器中通过http://<nodeip>:<nodeport>的方式访问应用。

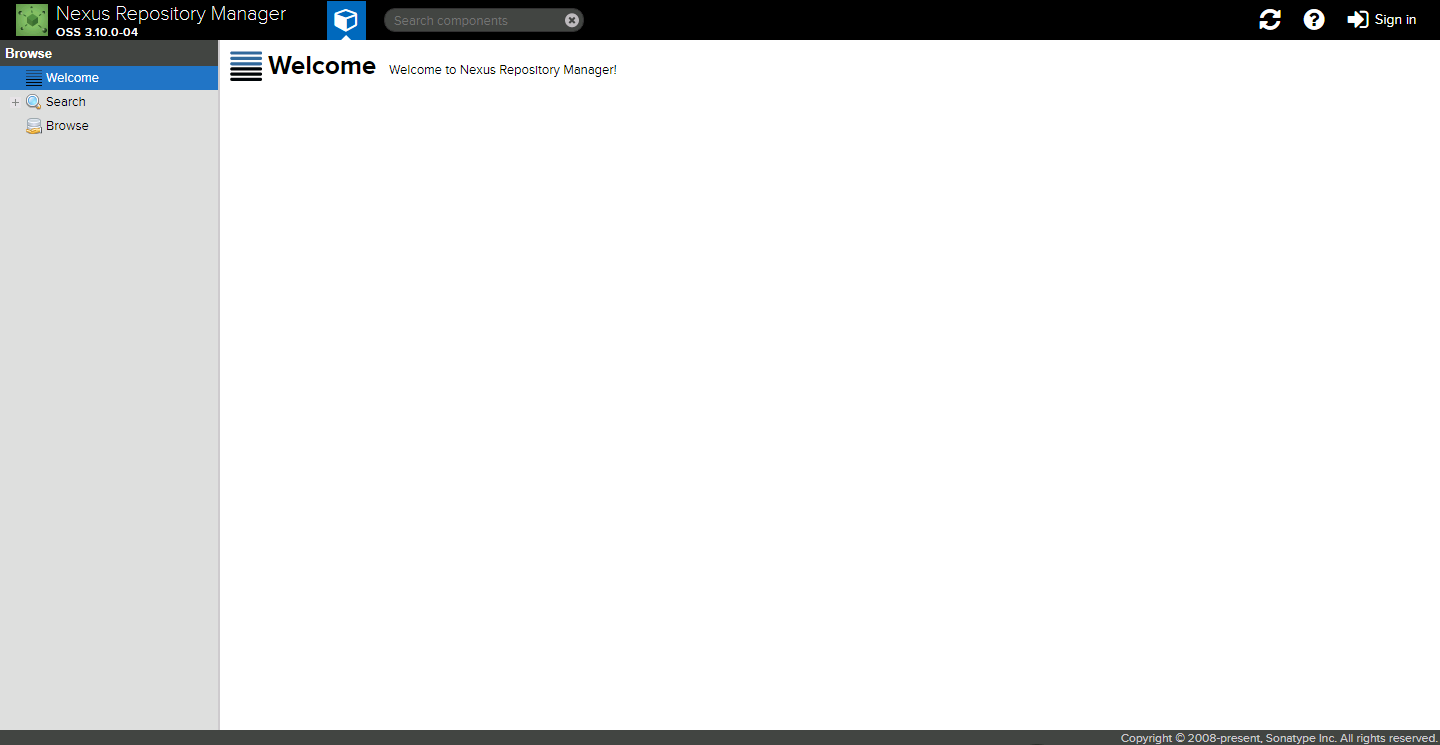


图4-12 nexus主页

## 步骤5：升级应用

从用户的角度，希望应用在任何时间都高可用的。从开发者的角度，需要在部署新版本的应用时，不影响用户的体验。在Kubernetes中，通过滚动升级来实现前述两类用户的期望，滚动升级通过增量式的升级Pod实例，从而实现在不影响用户体验的情况下，对应用进行升级应用。根据资源的可用性，新的Pod会被调度到合适的Node上。

步骤4通过扩容应用运行多个实例，这是不影响应用可用性升级模式的前提。升级时，通过设置数量和百分比来控制可用的Pod数量。在Kubernetes中，升级是有版本的，因此部署升级后，能回滚到之前的版本。与应用扩容相似，如果部署通过服务被暴露，在升级过程中，服务将会通过负载均衡会将流量分流到各个可以的Pod中。一个可用的Pod就是一个可用的应用实例。

滚动升级实现如下的行为：

* 通过容器镜像的升级，逐步升级环境中的应用；
* 回滚至之前的版本；
* 零宕机的持续集成和持续交付。

在升级前，集群中存在4个Pod。

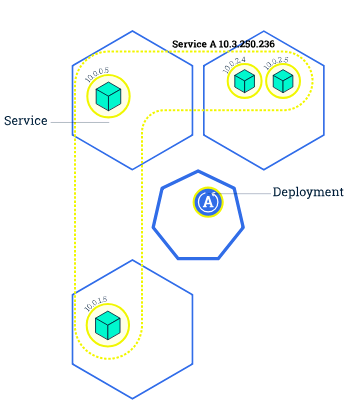


图4-13 升级前

升级开始后，Kubernetes先升级其中一个Pod，其它Pod继续对外提供服务。

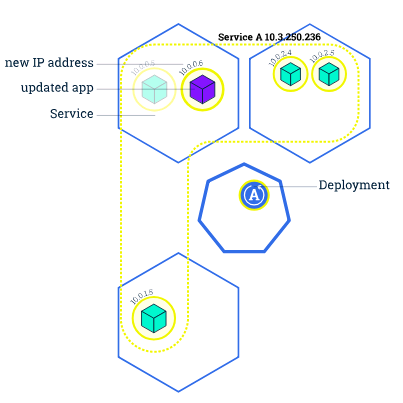


图4-14 升级第1个Node中的Pod

在第一个Pod升级后，Kubernetes将进行第二个Pod的升级，已升级的和未升级的应用会正常对外提供服务。

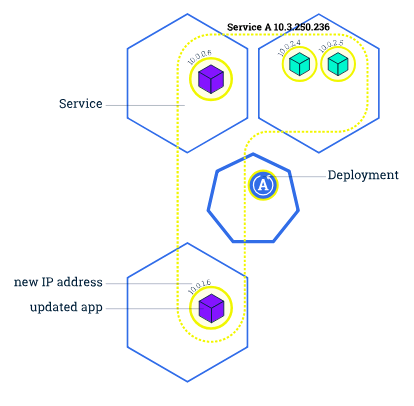


图4-15 升级第2个容器中的Pod

然后，Kubernetes升级接下来的应用，在在所有的Pod都完成升级后，整个升级过程才正式完成。

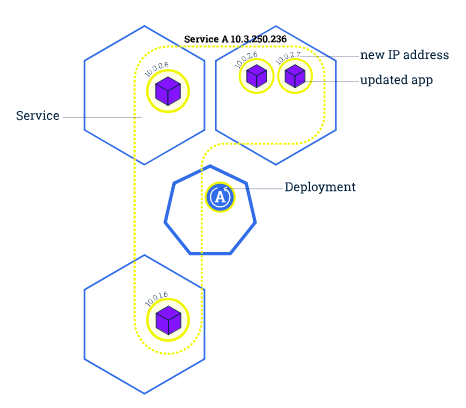


图4-16 升级第3个容器中的Pod

1）使用kubectl set image命令更新应用镜像版本

当应用发布新版本后，可以

$ kubectl set image deployments/my-nexus3 \*=sonatype/nexus3:latest

2）回滚升级到之前的版本

当部署的版本存在问题时，可以通过执行kubectl rollout unduo回滚至之前的版本：

$ kubectl rollout undo deployments/my-nexus3

3）查看部署的回滚状态

回滚状态，可以通过rollout status确认。通过执行kubectl rollout stauts可以查看升级和回归的状态信息：

$ kubectl rollout status deployments/my-nexus3

# Kubernetes核心对象

在这一章，本文对Kubernetes的核心对象进行了一个简单分类，分为工作负载、服务发现、配置与存储这三类，并对Pod、Deployment、DaemonSet、Service和Volume等核心的对象进行讲解。

## 工作负载(2019-04-11 Y)

工作负载可以理解为容器化应用在容器云中运行的承载者，负责对容器化应用的封装和运行管理。在当前版本中，Kubernetes主要有Pod、Deployment、DaemonSet、StatefulSet等工作负载对象。

### Pod（容器组）Y

Pod是Kubernetes最核心的执行单元，从逻辑上来看，对于容器来说Pod类似于宿主机。

#### ****Pod概述****

在Kubernetes集群中，Pod是所有业务类型的基础，它是一个或多个容器的组合。这些容器共享存储、网络和命名空间，以及如何运行的规范。在Pod中，所有容器都被统一安排和调度，并运行在共享的上下文中。对于具体应用而言，Pod是它们的逻辑主机，Pod包含业务相关的多个应用容器。Kubernetes不只是支持Docker容器，它也支持其他容器。Pod 的上下文可以理解成多个linux命名空间的联合：

* PID 命名空间（同一个Pod中应用可以看到其它进程）；
* 网络命名空间（同一个Pod的中的应用对相同的IP地址和端口有权限）；
* IPC命名空间（同一个Pod中的应用可以通过VPC或者POSIX进行通信）；
* UTS命名空间（同一个Pod中的应用共享一个主机名称）。

一个Pod的共享上下文是Linux命名空间、cgroups和其它潜在隔离内容的集合。 在Pod中，容器共享一个IP地址和端口空间，它们可以通过localhost发现彼此。在同一个Pod中的容器，可以使用System V 或POSIX信号进行标准的进程间通信和共享内存。在不同Pod中的容器，拥有不同的IP地址，因此不能够直接在进程间进行通信。容器间通常使用Pod IP地址进行通信。在一个Pod中的应用于口访问共享的存储卷，它被定为为Pod的一部分，可以被挂接至每一个应用文件系统。

与独立的应用容器一样，Pod是一个临时的实体，它有着自己的生命周期。在Pod被创建时，会被指派一个唯一的ID，并被调度到Node中，直到Pod被终止或删除。如果Pod所在的Node宕机，给定的Pod（即通过UID定义）不会被重新调度。相反，它将被完全相同的Pod所替代。这所说的具有和Pod相关生命周期的情况，例如存储卷，是说和Pod存在的时间一样长。如果Pod被删除，即使完全相同的副本被创建，则相关存储卷等也会被删除，并会Pod创建一个新的存储卷等。Pod本身就没有打算作为持久化的实体，在调度失败、Node失败和获取其它退出（缺少资源或者Node在维护）情况下，Pod都会被删除。一般来说，用户不应该直接创建Pod，即是创建单个的Pod也应该通过控制器创建。在集群范围内，控制器为Pod提供自愈能力，以及副本和部署管理。一个多容器的Pod会包含一个文件拉取器和一个web服务器，此web服务器使用一个持久化存储卷在容器中共享存储。

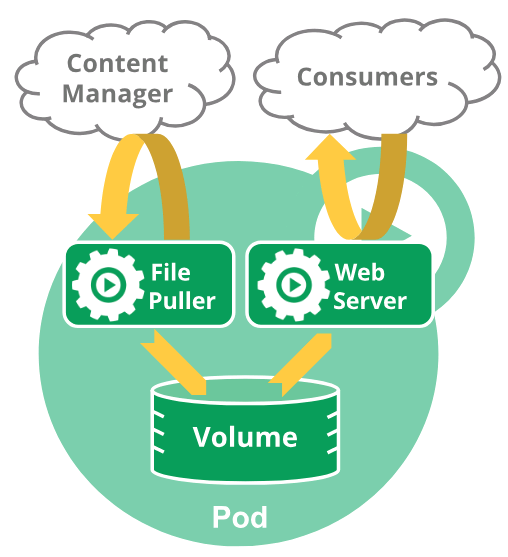


图 5-1 Pod的组成

**网络：**每一个Pod都会被指派一个唯一的Ip地址，在Pod中的每一个容器共享网络命名空间，包括Ip地址和网络端口。在同一个Pod中的容器可以同locahost进行互相通信。当Pod中的容器需要与Pod外的实体进行通信时，则需要通过端口等共享的网络资源。

**存储：**Pod能够被指定共享存储卷的集合，在Pod中所有的容器能够访问共享存储卷，允许这些容器共享数据。存储卷也允许在一个Pod持久化数据，以防止其中的容器需要被重启。

#### ****Pod的工作方式****

在Kubernetes中一般不会直接创建一个独立的Pod，这是因为Pod是临时存在的一个实体。当直接创建一个独立的Pod时，如果缺少资源或者所被调度到的Node失败，则Pod会直接被删除。这里需要注意的是，重起Pod和重起Pod中的容器不是一个概念，Pod自身不会运行，它只是容器所运行的一个环境。Pod本身没有自愈能力，如果Pod所在的Node失败，或者如果调度操作本身失败，则Pod将会被删除；同样的，如果缺少资源，Pod也会失败。Kubernetes使用高层次的抽象，即控制器来管理临时的Pod。通过控制器能够创建和管理多个Pod，并在集群范围内处理副本、部署和提供自愈能力。例如，如果一个Node失败，控制器可以自动的在另外一个节点上部署一个完全一样的副本。控制器是Pod模板来创建Pod，Pod的控制器包括：

* Deployment
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)

Pod模板是一个被包含在其它对象(例如：Deployment、StatefuleSet、DaemonSet等)中的Pod规格。控制使用Pod模板创建实际的Pod，下面是Pod模板的一个示例：



##### ****重启策略****

在Pod中的容器可能会由于异常等原因导致其终止退出，Kubernetes提供了重启策略以重启容器。重启策略对同一个Pod的所有容器起作用，容器的重启由Node上的kubelet执行。Pod支持三种重启策略，在配置文件中通过restartPolicy字段设置重启策略：

* Always：只要退出就会重启。
* OnFailure：只有在失败退出（exit code不等于0）时，才会重启。
* Never：只要退出，就不再重启

需要注意的是，这里的重启是指在Pod的宿主Node上进行本地重启，而不是调度到其它Node上。

##### ****镜像拉取策略****

在Kubernetes中，容器的运行是基于容器镜像的。Pod支持三种镜像拉取策略，在配置文件中通过*imagePullPolicy字体设置镜像的拉取策略*：

* Always：不管本地是否存在镜像都会进行一次拉取。
* Never：不管本地是否存在镜像都不会进行拉取。
* IfNotPresent：仅在本地镜像不存在时，才会进行镜像拉取。

默认情况下，镜像拉取策略的默认值为IfNotPresent，但:latest标签镜像默认为Always。在拉取镜像时，docker会对镜像进行校验，如果镜像中的MD5码没有变，则不会拉取镜像数据。生产环境中应该尽量避免使用:latest标签，而开发环境中可以借助:latest标签自动拉取最新的镜像。

#### ****使用私钥镜像仓库****

在Kubernetes中运行容器时，需要为容器获取镜像。Pod中容器的镜像有三个来源，即Docker公共镜像仓库、私有镜像仓库和本地镜像。在内网使用的Kubernetes场景下，就需要搭建和使用私有镜像仓库。在使用私有镜像拉取镜像时，需要为私有镜像仓库创建一个docker registry secret，并在创建容器中进行引用。

通过kubectl create secret docker-registry命令创建docker registry secret：

$ kubectl create secret docker-registry regsecret --docker-server=<your-registry-server> \

--docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>

在容器中通过imagePullSecrets字段指定该secret：



#### ****环境变量****

在创建Pod时，可以为在Pod中运行的容器设置环境变量。在Kubernetes中，通过env或envFrom字段进行设置。使用env或envFrom字段设置的环境变量，将会覆盖容器镜像中指定的环境变量。在下面的YAML文件中，设置了名称为**DEMO\_GREETING**和**DEMO\_FAREWELL**的两个环境变量。

**apiVersion:** **v1**

**kind: Pod**

**metadata:**

**name: envar-demo**

**labels: purpose: demonstrate-envars**

**spec:**

**containers:**

**- name: envar-demo-container**

**image: gcr.io/google-samples/node-hello:1.0**

**env:**

**- name: DEMO\_GREETING**

**value: "Hello from** **the environment"**

**- name: DEMO\_FAREWELL**

**value: "Such a sweet** **sorrow"**

#### ****启动命令****

在创建Pod时，也能够为Pod中的容器定义命令和参数。在配置文件通过设置command字段来定义命令，通过设置args字段来定义参数。在Pod被创建后，定义的命令和参数将不能被修改。在配置文件中定义的命令和参数会覆盖在容器镜像中定义的命令和参数。下面的YAML配置文件中，设置了printenv命令，以及设置了**HOSTNAME和KUBERNETES\_PORT两个参数。**

**apiVersion**: **v1**

**kind**:Pod

**metadata**:

**name**: **command**-**demo**

**labels**:

**purpose**: **demonstrate**-**command**

**spec**:

**containers**:

- **name**: **command**-**demo**-**container**

**image**: **debian**

**command**:["printenv"]

**args**:["HOSTNAME","KUBERNETES\_PORT"]

**restartPolicy**:OnFailure

1）通过上述YAML配置文件参加Pod：

**$ kubectl create -f**[**https://k8s.io/docs/tasks/inject-data-application/commands.yaml**](https://k8s.io/docs/tasks/inject-data-application/commands.yaml)

2）以列表的形式展示正在运行的Pod：

**$ kubectl get pods**

3）可以通过Pod的日志信息，参看命令的输出结果:

**$ kubectl logs command-demo**

输出结果显示了HOSTNAME和KUBERNETES\_PORT环境变量的值：

**command**-**demo tcp**://10.3.240.1:443

在前面的例子中，通过提供字符串直接定义了参数，在参数中也可以使用环境变量来定义参数：

**env:**

**- name: MESSAGE**

**value: "hello world"**

**command: ["/bin/echo"]args: ["$(MESSAGE)"]**

这意味可以使用所有任意的技术变量(用于定义环境变量的)来定义Pod的参数，包括ConfigMaps和Secrets。在参数中，环境变量以”$(VAR)“的格式出现。

#### ****资源限制****

Kubernetes通过cgroups来限制容器的CPU和内存等计算资源，在创建Pod时，可以为Pod中的每个容器设置资源请求(request)和资源限制(limit)。在Kubernetes中，计算资源是可以被请求、分配和消耗的可测量的数量。资源请求是容器需要的最小资源要求，资源限制为容器所能使用的资源上限。CPU的单位是核(core)，内存(Memory)的单位是字节(byte)。在Pod中，容器的资源限制通过resources.limits进行设置：

* spec.containers[].resources.limits.cpu：容器的CPU资源上限，可以短暂超过，容器也不会被停止；
* spec.containers[].resources.limits.memory：容器的内存资源上限，不可以超过；如果超过，容器可能会被停止或调度到其它资源充足的Node上。

而资源请求则通过resources.requests进行设置：

* spec.containers[].resources.requests.cpu：容器的CPU资源请求，可以超过；
* spec.containers[].resources.requests.memory：容器的内存资源请求，可以超过；但如果超过，容器可能会在Node内存不足时清理。

Kubernetes在进行Pod调度时，Pod的资源请求是最重要的一个指标。Kubernetes Schedule会检查Node是否存在足够的资源，判断是否能够满足Pod的资源请求，从而决定是否可以运行Pod。

在Kubernetes，当配置Pod时，可以为每一个容器设置CPU和内存这些计算资源。当容器被指定资源请求后，调度器将能够更好的决定将Pod部署在那一个Node上。

虽然从根源上来说，requests和limits是在容器上进行设置的，但是在Pod级别上的设置会带来更大的便利。Pod上的request/limit是Pod中各个容器request/limit的总和。

###### CPU资源

在Kubernetes中CPU资源通过cpu数量进行计算。如果容器的spec.containers[].resources.requests.cpu值为0.5，则表示它需要半个cpu。在Kubernetes中，如果spec.containers[].resources.requests.cpu值为0.1，则等价于spec.containers[].resources.requests.cpu值为100m。在kubernetes，设置CPU资源时，最小值为1m，也就是0.001。CPU资源是一个绝对值，而不是相对值，因此在单核，双核或者48核机器上，0.1都表达是同一意思，即0.1个CPU core。

###### 内存资源

在Kubernetes中，内存资源的计算单位为字节数(byte)，可以直接使用整型数字表达，也可以使用整数加国际单位制来表示。国际单位制包括：十进制(E, P, T, G, M, K)和二进制(Ei, Pi, Ti, Gi, Mi, Ki)，其中：1KB(kilobyte)=1000bytes，1KiB(kibibyte)=2^10bytes=1024bytes。例如，以下代表大致相同的值：

**128974848, 129e6, 129M, 123Mi**

###### 示例

下面的例子中拥有两个容器的Pod，每一个容器的request是0.25 cpu和 64MiB 内存，每个容器的limit是0.5 cpu和128MiB 内存。因此，Pod的requst是0.5核cpu和128MiB内存，Pod的limit是1核cpu和256MiB内存。

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: frontend**

**spec:**

**containers:**

**- name: db**

**image: mysql**

**env:**

**- name: MYSQL\_ROOT\_PASSWORD**

**value: "password"**

**resources:**

**requests:**

**memory: "64Mi"**

**cpu: "250m"**

**limits:**

**memory: "128Mi"**

**cpu: "500m"**

**- name: wp**

**image: wordpress**

**resources:**

**requests:**

**memory: "64Mi"**

**cpu: "250m"**

**limits:**

**memory: "128Mi"**

**cpu: "500m"**

#### ****健康检查****

在Pod部署到Kubernetes集群中以后，为了确保Pod处于健康正常的运行状态，Kubernetes提供了两种探针，用于检测容器的状态：

* Liveness Probe ：检查容器是否处于运行状态。如果检测失败，kubelet将会杀掉掉容器，并根据重启策略进行下一步的操作。如果容器没有提供Liveness Probe，则默认状态为Success；
* ReadinessProbe ：检查容器是否已经处于可接受服务请求的状态。如果Readiness Probe失败，端点控制器将会从服务端点（与Pod匹配的）中移除容器的IP地址。Readiness的默认值为Failure，如果一个容器未提供Readiness，则默认是Success。

kubelet在容器上周期性的执行探针以检测容器的健康状态，kubelet通过调用被容器实现的处理器来实现检测，在Kubernetes中有三类处理器：

* ExecAction ：在容器中执行一个指定的命令。如果命令的退出状态为0，则判断认为是成功的；
* TCPSocketAction ：在容器IP地址的特定端口上执行一个TCP检查，如果端口处于打开状态，则视为成功；
* HTTPGetAcction ：在容器IP地址的特定端口和路径上执行一个HTTP Get请求使用container的IP地址和指定的端口以及请求的路径作为url，用户可以通过host参数设置请求的地址，通过scheme参数设置协议类型(HTTP、HTTPS)如果其响应代码在200~400之间，设为成功。

健康检测的结果为下面三种情况：

* Success ：表示容器通过检测
* Failure ：表示容器没有通过检测
* Unknown ：表示容器容器失败

#### ****初始化容器****

在一个Pod中，可以运行多个容器，同时它也可以拥有有一个或多个初始化容器，初始化容器在应用程序容器启动之前运行。初始化容器与普通容器完全一样，只是：

* 它们总是完全执行；
* 每一个初始化容器都必须在下一个初始化开始之前成功完成。

如果Pod中的初始化容器失败，Kubernetes将会重复重启Pod，直到初始化容器成功执行。然而，如果Pod的重启策略为Never，则Pod不会重启。初始化容器支持应用程序容器的所有字段和特性，包括资源限制、存储卷和安全设置等。初始化容器不支持健康检测探针，因为，它们必须在POD准备好之前完成运行。如果为Pod指定了多个初始化容器，则这些初始化容器将会按顺序依次运行。每一个都必须在下一个运行之前成功运行。当所有的初始化容器都运行完成时，Kubernetes完成Pod的初始化，并像通常的方式一样运行应用程序容器。

#### Pod选择器

通过.spec.selector字段定义Pod选择器。从Kubernetes 1.8开始，必须指定一个与.spec.template标签相匹配选择器。一旦创建了DaemonSet，就不能对.spec.selector进行修改，改变Pod选择器可能会导致Pod成为孤儿。

.spec.selector是一个由两个字段组成的对象：

* matchLabels– 与ReplicationController的.spec.selector字段同样的工作机制。
* matchExpressions – 允许通过指定键，值列表以及与键和值相关的运算符来构建更复杂的选择器。

如果同时指定了两者时，结果取两者的和。如果指定了.spec.selector，则必须匹配.spec.template.metadata.labels，如果不匹配将会被API拒绝。

#### ****容器调度****

Kubernetes Scheduler负责根据调度策略自动将Pod部署到合适Node中，调度策略分为预选策略和优选策略，Pod的整个调度过程分为两步：

1）预选Node：遍历集群中所有的Node，按照具体的预选策略筛选出符合要求的Node列表。如没有Node符合预选策略规则，该Pod就会被挂起，直到集群中出现符合要求的Node。

2）优选Node：预选Node列表的基础上，按照优选策略为待选的Node进行打分和排序，从中获取最优Node。

##### ****预选策略****

随着版本的发展，Kunbernetes提供了大量的预选策略，通过预选策略能够筛选出符合条件的Node列表。预选策略是强制性规则，用来检测Node是否匹配Pod所需要的资源。如果没有任何Node能够满足预选策略, 该Pod就会被挂起，直到出现能够能够满足要求的Node。

表5-1 调度预选策略

|  |  |  |
| --- | --- | --- |
| 序号 | 预选策略 | 策略说明 |
| 1 | CheckNodeConditionPredicate | 检查是否可以将Pod调度到磁盘不足、网络不可用和未准备就绪的Node。 |
| 2 | PodFitsHost | 检查集群Node中是否存在与Pod配置文件中指定的Node名称相匹配。 |
| 3 | PodFitsHostPorts | 检查Node是否存在空闲可用的端口。 |
| 4 | PodMatchNodeSelector | 检查Pod上的Node选择器是否匹配Node的标签。 |
| 5 | PodFitsResources | 检查Node上的cpu、内存、gpu等资源是否满足Pod的需求，来决定是否调度Pod到Node上。 |
| 6 | NoDiskConflict | 根据Pod请求的存储卷进行评估，如果在这个Node已经挂载了存储卷，则其它同样请求这个存储卷的Pod将不能调度到这个Nods上。 |
| 7 | PodToleratesNodeTaints | 检查pod的能否容忍Node上的污点。 |
| 8 | PodToleratesNodeNoExecuteTaints | 检查Pod是否能容忍Node上未执行的污染。 |
| 9 | CheckNodeLabelPresence | 检查所有指定的标签是否存在于Node上，而不考虑它们的值。 |
| 10 | checkServiceAffinity | 检查服务的亲和性，确定是否在Node部署Pod。 |
| 11 | MaxPDVolumeCountPredicate | 检查Pod所需要的存储卷的数量，确定在哪个Node上部署Pod。 |
| 12 | VolumeZonePredicate | 根据volumes需求来评估Node是否满足条件。 |
| 13 | CheckNodeMemoryPressurePredicate | 检查Node内存的压力情况 |
| 14 | CheckNodeDiskPressurePredicate | 根据Node磁盘的压力情况，确定是否调度Pod到Node上。 |
| 15 | InterPodAffinityMatches | 根据Pod的亲和和反亲和的配置，检查是否能够将Pod调度到指定的Node上。 |

##### ****优选策略****

通过预选策略对Node过滤后，获得预选的Node列表。在预选Node列表的基础上，对这些预选的Node进行打分，从而为Pod选择一个分值最高的Node。Kubernetes通过一系列的优选策略对预选Node进行打分。每一个优选函数都会为Node给出一个0-10的分数，分数越高表示节点越优；同时，每个优选函数也会有一个对应的权重值。那个Node的最终得分是每个优选函数给出的得分的加权分数之和，因此每个Node的最终主机的得分如以下公式计算：

finalScoreNode = (weight1 \* priorityFunc1) + (weight2 \* priorityFunc2) + … + (weightn \* priorityFuncn)

表5-2 调度优选策略

|  |  |  |
| --- | --- | --- |
| **序号** | **优选策略** | **优选说明** |
| 1 | BalancedResourceAllocation | 根据Node上各项资源(CPU、内存)使用率均衡情况进行打分。 |
| 2 | ImageLocalityPriority | 基于Pod所需镜像的总体大小，根据Node上存在Pod所需镜像的大小从0到10进行打分。 |
| 3 | InterPodAffinityPriority | 基于Pod亲和情况打分。 |
| 4 | LeastRequestedPriority | 计算Pod需要的CPU和内存资源与在Node可用资源的百分比，具有最小百分比的节点就是最优。 |
| 5 | PriorityMetadata | 根据元素进行打分。 |
| 6 | MostRequestedPriority | 根据Node上所提供的资源进行打分。 |
| 7 | NodeAffinityPriority | 根据亲和情况进行打分。 |
| 8 | NodeLabelPriority | 根据Node上是否存在特殊的标签进行打分。 |
| 9 | NodePreferAvoidPodsPriority | 根据Node上的注释进行打分。 |
| 10 | ResourceAllocationPriority | 根据在Node上的分配的资源进行打分。 |
| 11 | ResourceLimitsPriority | 根据Pod的资源限制进行打分。 |
| 12 | SelectorSpreadPriority | 按service,RC,RS or StatefulSet归属计算Node上分布最少的同类Pod数量，得分计算，数量越少得分越高。 |
| 13 | TaintTolerationPriority | 基于Node上不可容忍的污点数进行打分。 |

##### ****将Pod分配给特定的Node****

有些场景下，需要指定Pod只能在特定的节点上运行，或者说更喜欢在特定的节点上运行。Kubernetes有几种方法可以做到这一点，比如使用nodeSelector和nodeName。一般情况下，这样的约束是不必要的，因为调度器会自动地进行合理的调度（例如，Kubernetes将的Pod部署在合适的节点上，而不是将Pod放置在自由资源不足的节点上）。但是在某些情况下，可能希望在Pod的节点上进行更多的控制。例如，确保Pod运行在带有SSD的机器上，或者将两个不同服务的Pod定位到相同的可用性区域中。

###### ****nodeSelector****

使用nodeSelector是指定Pod在特定主机上运行的最简单方式之一，nodeSelector是PodSpec的一个字段。为了使Pod能够在指定Node上运行，Node节点必须具有所指示的键值对作为标签（它也可以有附加的标签）。nodeSeletor的用法如下：

**1）为Node打上标签**

通过下面的命令，为Node节点打上标签，标签的键为disktype，标签的值为ssd：

**$ kubectl label nodes** <**node**-**name**>disktype=ssd

**2）在Pod配置文件中添加nodeSelector字段**

下面是用于创建Pod的YAML文件，名称为nginx，通过设置nodeSelector为disktype:ssd，明确此Pod将需要被部署在对应标签的Node主机节点上。

**apiVersion**: **v1**

**kind**:Pod

**metadata**:

**name**: **nginx**

**labels**:

**env**: **test**

**spec**:

**containers**:

- **name**: **nginx**

**image**: **nginx**

**imagePullPolicy**:IfNotPresent

**nodeSelector**:

**disktype**: **ssd**

**3）创建Pod，并将Pod调度到Node上**

通过执行如下命令，在集群中将会创建Pod，并在后台会将其调度到打上了键值对的Node上。

**$ kubectl create** -**f nginx**.**yaml**

通过下面的命令，可以查看Pod调度的情况

**$ kubectl get pods -o wide**

###### ****nodeName****

另外一种方式是在Pod中设置nodeName来指定所要部署的Node主机节点，但这种方式的灵活性相对差一些。

1）在Pod的配置文件中添加nodeName字段

**apiVersion:** **v1**

**kind: Pod**

**metadata:**

**name: nginx**

**labels:**

**env: test**

**spec:**

**containers:**

**- name: nginx**

**image: nginx**

**imagePullPolicy: IfNotPresent**

**nodeName： <NodeName>**

2）创建Pod，并将Pod调度到Node上

通过执行如下命令，在集群中将会创建Pod，并在后台会将其调度到所指定的Node上。

**$ kubectl create** -**f nginx**.**yaml**

通过下面的命令，可以查看Pod调度的情况

**$ kubectl** get **pods** -**o wide**

##### 污点和容忍

根据相关特性，DaemonSet Pods会自动添加以下的容忍度。

表5-3

| **容忍键** | **影响** | **版本** | **描述** |
| --- | --- | --- | --- |
| node.kubernetes.io/not-ready | NOEXECUTE | 1.13+ | 当存在诸如网络分区之类的节点问题时，不会驱逐DaemonSet pod。 |
| node.kubernetes.io/unreachable | NOEXECUTE | 1.13+ | 当存在诸如网络分区之类的节点问题时，不会驱逐DaemonSet pod。 |
| node.kubernetes.io/disk-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/memory-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/unschedulable | NoSchedule | 1.12+ | DaemonSet pods可以通过默认调度程序容忍不可调度的属性。 |
| node.kubernetes.io/network-unavailable | NoSchedule | 1.12+ | 使用主机网络的DaemonSet pod可以通过默认调度程序容忍网络不可用的属性。 |

##### 基于资源的Pod调度

在创建一个Pod时，Kubernetes调度器将会为Pod选择一个运行的Node。对于每一个Node来说，其都存在一个最大的资源能力(CPU和内存)。调度器在调度时，要确保Node上CPU和内存能够满足所有Pod对于计算资源的要求。

当kubelet启动Pod中的容器时，它会将容器的request和limit作为参数传递给容器运行时。如果容器运行时使用的是docker：

* + - spec.containers[].resources.requests.cpu的值会被转换为core，然后乘以1024，再将结果通过–cpu-shares参数的值传递给docker run命令。
    - spec.containers[].resources.limits.cpu的值会被转化为millicore，然后乘以100。结果值是作为容器在100微秒内能够使用的CPU总量时间。默认的配额周期是100ms，最小的CPU配额是1ms。
    - spec.containers[].resources.limits.memory被转化为整数，在docker run命令中作为–memory字段的值。

如果容器在运行过程中使用的内存超过了内存的limit，它将会被终止。同时如果此容器是可重启的，则kubelet会在后续会重新启动它。如果容器在运行过程中使用的内存超过了内存的request，则当Node内存不足时，它所在的Pod会被删除。与内存不同的是，在容器运行过程中如果使用了超过要求CPU，容器并不会被杀死。

#### ****Pod的基本操作****

##### ****创建Pod****

按照Kubernetes的设计，Pod一般不独立进行创建，这是因为独立创建的Pod没有自愈能力，也就说在Pod异常终止后，无法进行自动重启和重新调度。

1） 通过执行kubectl create -f命令创建名为nginx的部署和Pod：

$ kubectl create -f nginx.yml

2）通过执行kubectl get pods命令，可以看到在Kubernetes中运行了的nginx的Pod：

$ kubectl get pods

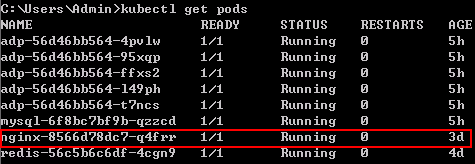


图5-1 集群中的Pod

##### ****查看Pod信息****

在Pod被创建出来以后，可以通过如下的命令查看特定Pod的信息：

$ kubectl describe pods/nginx-8566d78dc7-q4frr

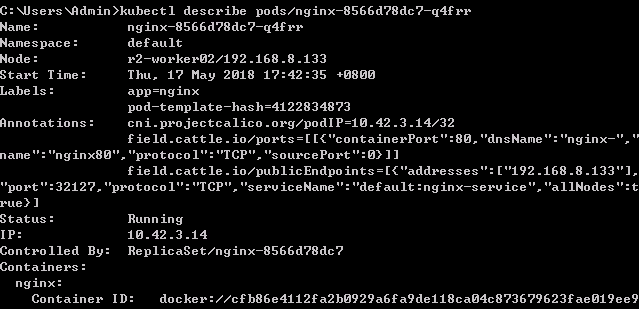


图5-2 Pod的详细信息

##### ****终止Pod****

在集群中，Pod代表着运行的进程，但不再需要这些进程时，如何优雅的终止这些进程是非常重要。以防止在Pod被暴力删除时，没有对Pod相关的信息进行必要的清除。当用户请求删除一个Pod时，Kubernetes将会发送一个终止（TERM）信号给每个容器，一旦过了优雅期，杀掉（KILL）信号将会被发送，并通过API server删除Pod。可以通过kubectl delete pod/{Pod名称} -n {命名空间名称}删除特定的Pod，一个终止Pod的流程如下：

1） 用户可以通过kubectl、dashboard等发送一个删除Pod的命令，默认优雅的退出时间为30秒；

2）更新API server中Pod的优雅时间，超过该时间的Pod会被认为死亡；

3）在客户端命令行中，此Pod的状态显示为”Terminating（退出中）”；

4）（与第3步同时）当Kubelet检查到Pod的状态退出中的时候，它将开始关闭Pod的流程：

* 如果该Pod定义了一个停止前的钩子(preStop hook)，其会在Pod内部被调用。如果超出优雅退出时间，钩子仍然还在运行，就会对第2步的优雅时间进行一个小的延长(一般为2秒)；
* 发送TERM的信号给Pod中的进程。

5）（与第3步同时进行）从服务的端点列表中删除Pod，对于副本控制器来说，此Pod将不再被认为是运行着的Pod的一部分。缓慢关闭的pod可以继续对外服务，直到负载均衡器将其移除。

6.）当超过优雅的退出时间，在Pod中任何正在运行的进程都会被发送被杀死。

7）Kubelet完成Pod的删除，并将优雅的退出时间设置为0。此时会将Pod删除，在客户端将不可见。

在默认情况下，Kubernetes集群所有的删除操作的优雅退出时间都为30秒。kubectl delete命令支持–graceperiod=的选项，以支持用户来设置优雅退出的时间。0表示删除立即执行，即立即从API中删除现有的pod，同时一个新的pod会被创建。实际上，就算是被设置了立即结束的的Pod，Kubernetes仍然会给一个很短的优雅退出时间段，才会开始强制将其杀死。

#### ****Pod的生命周期****

Pod的生命周期包括：从Pod被创建、并调度到Node中、以及Pod成功或失败的终止。Pod的阶段是一个简单的、高层次的Pod所处在生命周期的概述。在Pod的生命周期中，有如下的几个状态：

* Pending: Pod已经被Kubernetes系统接受，但是还有一个或者多个容器镜像未被创建。这包括Pod正在被调度和从网络上下载镜像的时间。
* Running: Pod已经被绑定到了一个Node，所有的容器也已经被创建。至少有一个容器已经在运行，或者在启动或者重新启动的过程中。
* Succeeded: 在Pod中的所有的容器都已经被成功的终止，并且不会再重启。
* Failed: 在Pod中所有容器都已经被终止，并且至少有一个容器是非正常终止的。即，容器以非零状态退出或者被系统强行终止的。
* Unknown: 由于某些原因，Pod不能被获取，典型的情况是在与Pod的主机进行通信中发生了失败。

在Pod的规格中有一个restartPolicy属性，它的值包括：Always, OnFailure和Never。

* Always：当容器终止退出后，总是会重启容器，这是默认值；
* OnFailure：只有在容器非正常退出时，才会重启容器。
* Never：不管容器是否正常退出，都不再重启容器。

#### 水平Pod自动伸缩

在当前的版本中，Pod水平自动伸缩支持基于CPU使用情况对Pod进行自动伸缩。Pod水平自动伸缩通过Kubernetes API资源和控制器进行实现。控制器会根据CPU的使用情况周期性的调整Pod的副本数量。

##### 水平Pod自动伸缩工作机制

水平Pod自动伸缩被实现为控制周期循环，其周期由控制器管理器的–horizontal-pod-autoscaler-sync-period标志进行控制（默认值为30秒）。在每个周期间，控制器管理器会根据每个Horizo​​ntalPodAutoscaler定义中指定的度量标准查询资源利用率。控制器管理器从资源指标的API（针对每个窗格的资源指标）或自定义指标的API（针对所有其他指标）获取度量。

* 对于每个Pod资源指标（如CPU），控制器为HorizontalPodAutoscaler的目标Pod从资源指标API中获取指标。如果设置了目标利用率的值，则控制器将在会计算使用率，作为每个Pod中容器的等效资源请求的百分比。如果设置了目标原始值，则直接使用原始指标标准值。然后，控制器在所有目标Pod中获取利用率的平均值或原始值（取决于指定的目标类型），并产生用于缩放所需副本数量的比率。需要注意的，如果某些Pod的容器没有设置相关的资源请求，则不会定义Pod的CPU利用率，并且autoscaler不会对该指标标准采取任何操作。
* 对于每个Pod自定义的指标，与每个pod资源指标的工作机理类似；不同之处在于它使用初始值，而不是使用率值。
* 对于对象指标和外部指标，将获取单个指标，该指标将描述相关对象。该指标将与目标值进行比较，以产生如上所述的比率。在autoscaling/v2beta2API版本中，可以选择在进行比较之前将该值除以pod的数量。

Horizo​​ntalPodAutoscaler通常从一系列的API聚集中获取指标（metrics.k8s.io， custom.metrics.k8s.io和external.metrics.k8s.io）。该metrics.k8s.io API通常由指标服务器提供，其需要单独启动。自动缩放器使用伸缩子资源访问相应的可伸缩控制器（例如复制控制器，部署和副本集）。Scale是一个接口，允许您动态设置副本数并检查每个当前状态。

##### 水平Pod自动伸缩算法

从最基本的角度来看，Horizo​​ntalPodAutoscaler控制器根据所需指标与当前指标之间的比率进行操作：

desiredReplicas = ceil[currentReplicas \* ( currentMetricValue / desiredMetricValue )]

例如，如果当前的指标值为200m，而期望的指标值为100m，则Pod的副本数量将会翻倍，即200/100=2；如果当前的指标值为50m，则会将Pod的副本数减半。如果比率足够接近1.0（在全局可配置的容差范围内，从*–horizontal-pod-autoscaler-tolerance*标志，默认为0.1），将会忽略缩放。当指定*targetAverageValue*或*targetAverageUtilization*时,*currentMetricValue会*通过所有Pod(Horizo​​*ntalPodAutoscaler伸缩*目标)获取给定指标的平均值来计算。当基于CPU进行伸缩时，如果任意Pod还未准备就绪，则此Pod放弃。*currentMetricValue / desiredMetricValue*伸缩比率基于保留的Pod进行计算，而不是放弃的Pod进行计算。如果在*HorizontalPodAutoscaler*中指定了多个指标，则会计算每一个指标，并选择期望副本数最大的指标。

##### 在kubectl中使用水平Pod自动伸缩

与所有API资源一样，kubectl以标准方式支持水平Pod自动伸缩。可以使用kubectl create命令创建一个新的自动缩放器。通过kubectl get hpa可以列出自动伸缩器，通过kubectl describe hpa能够获得自动伸缩器的详细说明。最后，可以使用kubectl delete hpa删除自动缩放器。此外，还有一个特殊kubectl autoscale命令可以轻松创建Horizo​​ntalPodAutoscaler。

**1）创建autoscale**

$ kubectl autoscale deployment netdisk --min=2 --max=5 --cpu-percent=80

为部署netdisk创建一个自动伸缩器，目标CPU利用率设置为80%，副本数量介于2和5之间。

**2）获取autoscale**

获取default命名空间下的hpa

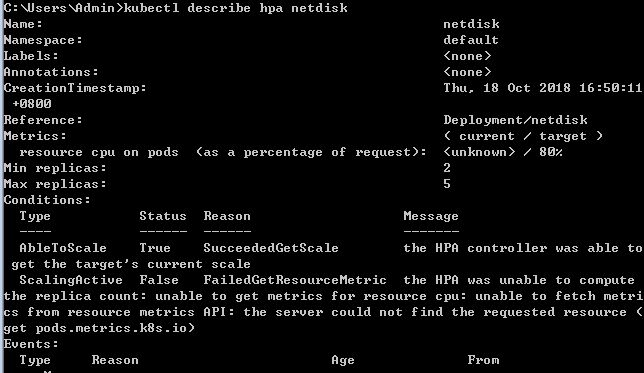
$ kubectl get hpa



**3）查看autoscale**

查看netdisk hpa的详细信息。

$ kubectl describe hpa netdisk



**4）删除autoscale**

**$ kubectl delete hpa netdisk**

### Deploymnet（部署）

#### 部署的用例

在开始学习部署的相关内容之前，先来了解一下部署的典型用例：

* 创建一个带有副本集的部署： RecplicSet在后台创建Pod，并检查Pod的发布状态，以确定其是否发布成功。
* 声明Pod的新状态：通过更新部署的PodTemplateSpec 。创建一个新的ReplicaSet ，并且将Pod从旧的ReplicaSet移动到新的ReplicaSet。每个
* 新的ReplicaSet都都会更新部署的版本。
* 回滚至之前的部署版本：如果当前的部署状态不稳定，就可以通过回滚返回到以前的版本。
* 扩容部署以满足更高负载需求：当对于应用的负载增加，可以通过对部署进行扩容满足需求。
* 暂停部署：将多个补丁应用到它的PodTemplateSpec 中，然后恢复启动。
* 查看部署的状态：通过查看部署的状态，能够查看发布是否成功。
* 清除旧的副本集：对于不再需要的RecplicSet，可以进行清除。

#### 创建应用部署

##### 定义YAML文件

下面是部署的YAML配置文件示例，它将创建一个带有3个nginx Pod的副本集（ReplicaSet）：

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

revisionHistoryLimit: 2

template:

metadata:

labels:

app: nginx

spec:

imagePullSecrets:

- name: dc-hspfd

containers:

# 应用的镜像

- image: nginx:1.7.9

name: nginx

imagePullPolicy: IfNotPresent

# 应用的内部端口

ports:

- containerPort: 80

name: nginx80

# 持久化挂接位置，在docker中

volumeMounts:

- mountPath: /usr/share/nginx/html

name: nginx-data

- mountPath: /etc/nginx

name: nginx-conf

volumes:

# 宿主机上的目录

- name: nginx-data

nfs:

path: /k8s-nfs/nginx

server: 192.168.8.150

- name: nginx-conf

nfs:

path: /k8s-nfs/nginx/conf

server: 192.168.8.150

在上述的YAML配置文件的示例中：

* 创建了一个名称为nginx的部署，名称通过metadata:name进行设置。
* 部署创建带有三个副本的Pod，副本通过replicas来设置。
* selector定义了部署如何发现所管理的Pod。在此例子，部署通过app:nginx标签选择Pod。
* Pod模块规范template:spec指定在Pod中将运行什么容器，此例子中是nginx，使用的是nginx:1.7.9镜像。
* 部署为Pod开放了80端口。

在配置文件的template中包含了下面的命令：

* 将Pod被打上app:nginx标签。
* 创建一个名为nginx的容器。
* 容器使用nginx:1.7.9的镜像。
* 打开80端口，容器能够通过此端口发送和接受流量。

##### 执行部署操作

**1）创建部署**

通过执行下面的命令来创建此部署：

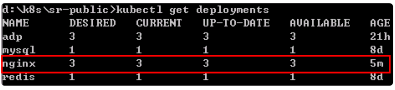
$ kubectl create -f {path}/nginx.yaml

**2）查看部署的信息**

通过执行如下命令，将能够或部署的相关信息：

$ kubectl get deployment

执行上面的命令后，能够看到集群中部署的相关信息：



* 名称（NAME）：列出在集群部署的名称。
* 期望的副本数量：显示期望的副本数量。
* 当前运行的副本数量（CURRENT）：显示当前正在运行的发布集数量。
* 最新副本数量：显示已更新以实现所需状态副本的数量。
* 可使用的副本数量：显示可被用户使用的副本集数量。
* 运行时间：实现应用已经运行的时间。

注意，每个字段中的值如何对应于部署规范中的值：

* 根据spec:replicas字段，所需副本的数量为3。
* 根据.status.replicas字段，当前副本的数量为3个。
* 根据 .status.updatedReplicas字段，最新副本的数量为3个。
* 根据.status.availableReplicas字段，可用副本的数量为3个。

**3）查看部署的发布状态**

要查看部署的发布状态，请运行：

$ kubectl rollout status deployment/nginx

此命令返回以下输出：等待发布完成：3个新副本中的0个已经更新…

等一会儿，再次运行kubectl get deployments命令：

$ kubectl get deployments

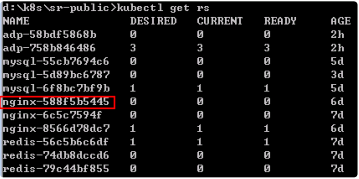
注意，部署已经创建了所有三个副本，并且所有副本都是最新的（它们包含最新的Pod模板）和可用的（Pod状态已经准备好，至少等

于 .spec.minReadySecondsfield的值）。

**4）查看副本集的信息：**

通过执行下面的命令，可以查看副本集的情况：

$ kubectl get rs



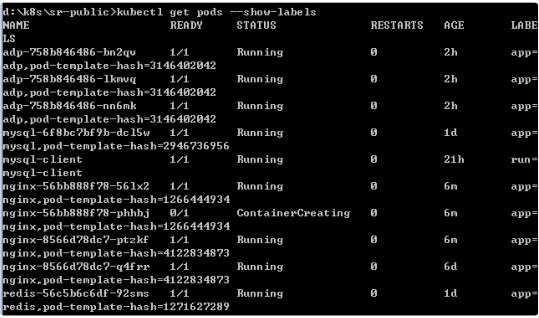
请注意，副本集名称的格式化为[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]。哈希值是在创建部署时自动生成的。

**5）查看标签**

通过运行如下的命令，能够查看为每一个Pod自动创建的标签：

$ kubectl get pods --show-labels

创建的副本集将会确保有一直3个nginx的Pod在运行。



注意：必须在部署中指定合适的选择器和Pod模板标签（此处为App:nginx）。不要与其他控制器（包括其他部署和有状态集）重叠标签或选择器。Kubernetes不会阻止重叠，如果多个控制器有重叠的选择器，这些控制器可能会发生冲突和行为意外。

注意不要改变这个Pod的哈希标签，部署控制器将Pod模板哈希标签添加到所创建或采用的每个副本集中。此标签确保部署的子副本集不重叠。它是通过对ReplicaSet的PodTemplate进行哈希而生成，并将得到的哈希值作为标签值添加到ReplicaSet选择器、Pod模板标签，以及ReplicaSet存在的任何现有的Pod中的。

#### 更新部署

一般情况下，已发布的应用版本都可能会存在一些问题；或者，由于业务的需要，需要在原有版本上增加和改进新功能。当发布了应用新的版本时，则需要对部署进行更新。在前面使用nginx:1.7.9的镜像进行了部署，官方发布了新版本nginx:1.9.1，此版本的功能更加成熟和稳定，因此考虑进行更新：

$ kubectl set image deployment/nginx nginx=nginx:1.9.1

另外，也可以通过编辑部署的yaml文件，将.spec.template.spec.containers[0].image的修改为：nginx:1.9.1来实现更新。

$ kubectl edit deployment/nginx

通过运行下面的命令，可以看到发布的状态：

$ kubectl rollout status deployment/nginx

当被更新时，部署可以确保只有一定数量的Pod被停掉。例如，如果仔细观察上面的部署，会发现首先创建了一个新的Pod，然后删除了一些旧的Pod并创建新的Pod。在足够数量的Pod出现之前，不会杀死旧的Pod；直到有足够数量的Pod被杀死之后，才会创建新的Pod。它会确保可用的Pod至少为2，总Pod最多为4个。

当第一次创建部署时，创建了一个副本集，并将其直接扩容到3个副本。当更新部署时，创建一个新的副本集，并将其缩容到1。然后将旧的副本集缩减为2，以便至少有2个Pod可用，并且最多创建了4个Pod。然后，用相同的滚动更新策略继续扩容和缩容新的和旧的副本集。最后，在新的副本集中有3个可用的副本，旧的副本集被缩减为0个。

##### 滚动更新

每当部署控制器观察到新的部署对象时，如果没有现有的ReplicaSet ，则会创建一个ReplicaSet 来生成所需的Pod。现有的ReplicaSet控制Pod(.spec.selector匹配、.spec.template不匹配）进行缩容。最终，新的ReplicaSet 将被缩小到.spec.replicas给定的值，并且所有的旧ReplicaSets 将被缩小到0。

当已有的发布正在运行，如果要更新部署，则部署将根据更新创建一个新的ReplicaSet ，并开始对其进行扩容。同时将滚动更新以前扩容的ReplicaSet，并将旧的ReplicaSet进行缩容。例如：假设为nginx:1.7.9创建由5个副本集的部署，然而，将部署更新到带5个副本集的nginx:1.9.1，但nginx:1.7.9仅仅有3个副本集被创建时。在这种情况下，部署将马上杀死已被创建的3个nginx:1.7.9 Pod，并开始创建nginx:1.9.1 Pod。

##### 标签选择器更新

一般不鼓励做标签选择器更新，建议您预先规划您的选择器。在任何情况下，如果您需要执行标签选择器更新，请非常小心，并确保您已经掌握了所有的含义。注意: 在 API版本 apps/v1, 部署的标签选择器在创建之后是不可变的。

* 选择器添加：要求在部署spec中的Pod模板标签也使用新标签进行更新，否则会返回验证错误。此更改是非重叠的，这意味着新的选择器不会选择用使用旧的选择器创建的ReplicaSets 和Pod，可能导致旧的ReplicaSets 成为孤儿，并创建新的ReplicaSets。
* 选择器更新：即改变选择器键中的现有值，导致的行为与选择器添加相同。
* 选择器移除：即从部署选择器中移除现有的键，不需要在Pod模板标签中进行任何更改。现有的ReplicaSet 不会被孤立，也不会创建新的ReplicaSet ，但请注意，删除的标签仍然会存在于任何现有的Pod和ReplicaSets中。

##### 回滚部署

如果当前部署不稳定或存在其它问题的情况下，则需要将部署回滚到以前的版本。在Kunbernetes的默认情况下，所有的部署的历史都保存在系统中，这样就可以随时进行部署的回滚。需要注意的是，当一个部署发布被触发，Kubernetes将会创建一个版本。仅仅当部署的Pod模板(.spec.template)的内容发生变化，Kubernetes才会创建新的版本。这就意味着，当回滚到之前的版本是，仅仅进行了Pod模板部分内容的回滚。

假设之前部署的了nginx:1.9.1，在Kubernetes中将其升级到了nginx:1.91

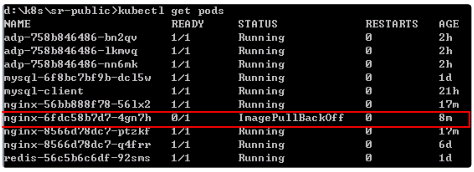
$ kubectl set image deployment/nginx nginx=nginx:1.91

可以通过kubectl rollout status来查看部署的升级状态：

$ kubectl rollout status deployments nginx

通过ctrl-c退出状态信息的查看，通过查看已创建的Pod，将能够看到有1个Pod被卡住了。

$ kubectl get pods



部署控制器会自动停止糟糕的发布，并且会停止新副本集的扩容。当这取决于所设置的滚动更新(rollingUpdate)参数（通过maxUnavailable设置）。默认情况下，Kubernetes将maxUnavailable 值设置为1，并将spec.replicas值也设置为1。因此如果不设置这些参数，则在默认情况下，您的部署可能会100%不可用！这个问题将会在Kubernetes的未来版本中得的处理。

##### 检查部署的发布历史

能够通过下面的命令检查部署的所有版本历史：

$ kubectl rollout history deployment/nginx

通过–revision参数能够查看特定版本的详细信息：

$ kubectl rollout history deployment/nginx --revision=2

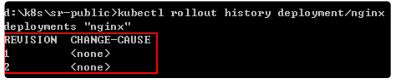
##### 回滚到以前的版本

如果发现新发布的版本有问题，则可以取消当前的发布，并将其回滚到之前的版本：

$ kubectl rollout undo deployment/nginx

另外，也可以通过–to-revision参数指定回滚的之前的哪个版本：

$ kubectl rollout undo deployment/nginx --to-revision=2



#### 扩缩容部署

在实际的应用中，由于应用所面对的用户规模会发生变化。为了在用户量增加时能够保证应用的服务质量，则需要增加部署的数量；但用户量减少时，为

了将资源释放出来给其他有需要的应用使用，则需要减少部署的数量。

在Kubernetes中，可以通过kubectl scale命令控制部署的数量。通过执行如下的命令，可以将部署扩容到10个：

$ kubectl scale deployment nginx --replicas=10

另外，能够根据Pod当前系统的负载进行自动水平扩容，如果系统负载超过预定值，就开始增加Pod的个数，如果低于某个值，就自动减少Pod的个数。

目前Kunbernetes的自动水平扩展只能根据CPU和内存去度量系统的负载，并依赖heapster去收集CPU的使用情况。下面的一个自动水平扩展的例子，

CPU的利用率在80%，Pod的数量维持在10至15个之间。

$ kubectl autoscale deployment nginx --min=10 --max=15 --cpu-percent=8

通过执行下面的命令可以查看新创建的自动水平控制：

$ kubectl get hpa



#### 部署状态

部署在创建后，会经历自身的生命周期，因此存在各种状态。在发布一个新的ReplicaSet时，它即可能是完成状态，也可能是失败的状态。

##### 运行中的部署

当下列的进程被执行时，Kubernetes 标志部署的状态为正在进行：

* 部署创建一个新的ReplicaSet；
* 部署扩容一个最新的ReplicaSet；
* 部署缩容旧的ReplicaSet(s)；
* 新Pods就绪或处于可用的情况。

通过执行如下命令可以监控部署的进程：

$ kubectl rollout status deployments/nginx

##### 已完成的部署

当具有如下特点时，Kubernetes 标识部署为完成状态：

* 与部署相关联的所有副本都已更新到您指定的最新版本，这意味着您所请求的任何更新都已完成。
* 与部署相关联的所有副本都可用。
* 部署的没有旧副本正在运行。

通过kubectl rollout status可以检查部署是否以完成。如果发布成功，kubectl rollout status会返回退出代码为0的返回值。

$ kubectl rollout status deploy/nginx

##### 已失败的部署

部署可能会被卡住，因此会试图部署其最新的ReplicaSet 。这可能是由于以下因素之一：

* 计算资源配额不足；
* 准备就绪探测器失败；
* 镜像拉取失败；
* 没有足够的权限；
* 应用程序运行时错误配置。

##### 清理策略

可以在部署中设置 .spec.revisionHistoryLimit字段，以指定要为部署保留多少个ReplicaSets 。后台的垃圾收集将会处理其余的ReplicaSets 。默认情况下，将保留所有的修订历史。在将来的版本中，它将默认将会设置为2。注意：如果将该字段显式设置为0，将导致清理部署的所有历史，因此部署将无法回滚。

#### 部署规格说明

与所有其他Kubernetes 配置一样，部署需要apiVersion, kind和metadata 字段，部署也需要.spec部分。

##### Pod模板

.spec.template是必需的字段，它是一个Pod模板，因此与Pod的模式是一样的。但是它被嵌套在部署中，同时没有apiVersion和kind。

除了Pod所需的字段之外，部署中的Pod模板还必须指定适当的标签和适当的重新启动策略。对于标签，确保不要与其他控制器重叠。 只有.spec.template.spec.restartPolicy (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/)等于Always是允许的，如果未指定，这是默认值。

##### 选择器

.spec.selector是一个可选的字段，用于为部署指定目标Pod。.spec.selector必需匹配.spec.template.metadata.labels, 否则江会被API拒绝

在API的apps/v1版本中，如果不进行设置时， .spec.selector 和 .metadata.labels 默认并不是.spec.template.metadata.labels。因此，它们必须被明确设置。还要注意的是，在apps/v1版本中，部署被创建后，.spec.selector 不可变的。

如果它们的模板不同于.spec.replicas，或者如果这些Pod的总数超过.spec.template，则部署可以终止其标签与选择器匹配的Pod。如果Pod的数量小于所需数量，它会产生新的Pod。注意：通过创建部署，或者诸如ReplicaSet和ReplicationController控制器来创建Pod，Pod的标签与的选择器。如果这样做，第一次部署就认为它创建了其他的Pod。Kubernetes并没有阻止这样做。如果你有多个控制器有重叠的选择器，控制器会互相争斗。

##### 策略

.spec.strategy被用来指定使用新的Pod代替旧的Pod的策略。.spec.strategy.type的值可以“Recreate” 和“RollingUpdate”。其中，“RollingUpdate” 是默认值。

* 重建部署：当.spec.strategy.type==Recreate时，在新的Pod被创建时，所有已存在的Pod都会被杀死。
* 滚动更新部署：当.spec.strategy.type==RollingUpdate时，部署将会以滚动更新的模式更新Pod，可以通过指定maxUnavailable和maxSurge来控制滚动更新过程。
* 最大不可用数量

.spec.strategy.rollingUpdate.maxUnavailable是一个可选字段，指定在更新过程中不可用的最大数量的Pod。该值可以是绝对数（例如，5）或期望Pod的百分比（例如，10%）。绝对数通过百分比的舍入计算获得。该值不能为0，默认值为25%。例如，当该值设置为30%时，当滚动更新开始时，旧的ReplicaSet 立即缩容到期望Pod数量的70%。一旦新的Pod准备好，旧的ReplicaSet就会被进一步缩小，接着扩容新的ReplicaSet，确保在更新期间随时可用Pod的总数是期望Pod的至少70%。

* 最大期望数量

.spec.strategy.rollingUpdate.maxSurge是一个可选字段，指定可在所需数量Pod的基础上创建的最大数量的Pod。该值可以是绝对数（例如，5）或期望的Pod的百分比（例如，10%）。如值不能为0。绝对数是通过舍入百分比来计算获得的。默认值为25%。

例如，当该值设置为30%时，新的ReplicaSet 可以在滚动更新开始时立即扩容，使得旧的和新的Pod的总数不超过期望的Pod的130%。一旦旧的Pod被杀死，新的ReplicaSet 将会进一步被扩容，确保在更新期间在任何时间运行的Pod的总数最多是所需Ood的130%。

##### 进度截止时间

.spec.progressDeadlineSeconds是一个可选字段，指定在系统报告部署失败之前，希望等待部署的秒数，部署控制器将继续重新部署。在将来，一旦实现了自动回滚的能力，部署控制器在观察到条件(Type=Progressing, Status=False和Reason=ProgressDeadlineExceeded)，就会回滚部署。如果指定，此字段需要大于.spec.minReadySeconds。

##### 最小的准备就绪时间

.spec.minReadySeconds是一个可选字段，指定在没有任何容器崩溃的情况下，新创建Pod应该准备就绪(被认为是可用的)的最小秒数。默认值为0（Pod一旦它准备好了，将被认为是可用的）

##### 版本历史限制

部署的修订历史存储在它控制的副本集中。.spec.revisionHistoryLimit是一个可选字段，指定的允许回滚的旧ReplicaSets 的数量。它的理想值取决于新部署的频率和稳定性。所有旧的ReplicaSets 将默认被保存，但它会消耗etcd的资源。每个部署修订的配置存储在其ReplicaSet 中；因此，一旦删除了旧的ReplicaSet ，就会失去了回滚到部署的修订的能力。

更具体地说，如果将此字段设置为零，意味着将不会清除所有具有0副本的ReplicaSets 。在这种情况下，将无法回滚新部署，因为它的修订历史已被清除。

##### 暂停

.spec.paused是一个可选的布尔字段，用于暂停和恢复部署。暂停部署与未暂停的部署之间的唯一区别在于，暂停的部署的PodTemplateSpec中的任何更改只是暂停，因此就不会触发新的发布。在创建时，默认部署不会暂停。

### StatefulSet（有状态副本集）

在Kubernetes中，StatefulSe被用来管理有状态应用的API对象。StatefulSets在Kubernetes 1.9版本才稳定。StatefulSet管理Pod部署和扩容，并为这些Pod提供顺序和唯一性的保证。与Deployment相似的地方是，StatefulSet基于spec规格管理Pod；与Deployment不同的地方是，StatefulSet需要维护每一个Pod的唯一身份标识。这些Pod基于同样的spec创建，但互相之间不能替换，每一个Pod都保留自己的持久化标识。

#### 使用StatefulSet的场景

在实际应用过程中，有一些应用场景需要记录应用的状态信息，对于这些应用场景，则需要使用StatefulSets，而不是Deployment。这些场景包括：

* 稳定、唯一的网络标识；
* 稳定、持久的存储；
* 按照顺序、优雅的部署和扩容；
* 按照顺序、优雅的删除和终止；
* 按照顺序、自动滚动更新。

上述的稳定是持久的同义词，如果应用不需要稳定的标识或者顺序的部署、删除、扩容，则应该使用无状态的副本集。Deployment或者ReplicaSet的控制器更加适合无状态业务场景。

在使用StatefulSet，需要注意下面的事项：

* Pod存储由PersistentVolume（storage类或者管理员预先创建）提供；
* 删除或者缩容StatefulSet不会删除与StatefulSet关联的数据卷，这样能够保证数据的安全性；
* 当前的StatefulSets需要一个Headless服务来为Pod提供网络标识，此Headless服务需要通过手工创建。

#### StatefulSet示例

在这里展示nginx以StatefulSet类型部署的YAML文件示例，此StatefulSet的组成如下：

* Headless服务：一个名称为nginx的Headless服务，用来控制网络域。
* 副本集：一个名称为nginx的statefulSet，它拥有nginx容器（在唯一的Pod启动）的3个副本集。
* PersistenVolumes存储卷：使用PersistenVolumes（由PersistentVolume Provisioner提供）提供稳定存储的volumeClaimTemplates。

apiVersion:v1

kind:Service

metadata:

  name:nginx

  labels:

     app:nginx

spec:

  ports:

  -port:80

     name:web

 #通过设置clusterIP的值为None，声明此服务为Headless服务

clusterIP:None

  selector:

     app:nginx

---

apiVersion:apps/v1

kind:StatefulSet

metadata:

  name:nginx

spec:

  selector:

     matchLabels:

       # 此处的值需要匹配 .spec.template.metadata.labels的值

app:nginx

  serviceName:"nginx"

  # 设置容器的副本数量，默认值为1

replicas:3

  template:

     metadata:

       labels:

         # 此处的值需要匹配 .spec.selector.matchLabels的值

app:nginx

     spec:

       terminationGracePeriodSeconds:10

       containers:

       -name:nginx

         image:k8s.gcr.io/nginx-slim:0.8

         ports:

         -containerPort:80

           name:web

         # 挂接数据卷

volumeMounts:

         -name:www

           # 容器内的挂接路径

mountPath:/usr/share/nginx/html

 # 持久化数据卷声明模板

volumeClaimTemplates:

  -metadata:

       name:www

     spec:

       accessModes:["ReadWriteOnce"]

       storageClassName:my-storage-class

       resources:

         requests:

           storage:1Gi

* **Pod选择器**

必须设置StatefulSet的sepc.selector，以匹配.spec.template.metadata.labels。在Kubernetes 1.8之前，spec.selector是可以忽略的，它被设置一个默认值。在1.8或者后续的版本，如果不设置sepc.selector，则会导致创建StatefulSet失败。

* **Pod身份标识**

StatfuleSet Pod拥有一个唯一的身份标识，它由顺序、稳定的网络标识和稳定的存储所组成。此身份标识一直跟随着Pod，不过它被调度到那个Node上。

* **序数索引（Ordinal Index）**

对于拥有N个副本集的StatefulSet，在StatefulSet中的每一个Pod都会被指派一个整型的序数，此序数在0和N之间，在整个集合中是唯一的。

* **网络ID(Stable Network ID)**

在StatefulSet中，每一个Pod的主机名称都由StatefulSet的名称和序数所组成。Pod的主机名称的格式：$(statefulset name)-$(ordinal)。如果创建了三个Pod，这他们的主机名称为web-0，web-1，web-2。StatefulSet能够使用Headless服务来控制Pod的域。Service管理的域的格式为：$(service name).$(namespace).svc.cluster.local，cluster.local是集群域。对于每一个被创建的Pod，它将得到一个DNS子域，格式为： $(podname).$(governing service domain)，这里的管理服务在StatefulSet中，通过serviceName设置。

下面是StatefulSet中Pod在DNS中的名称：

| 集群域 | Service (ns/name) | StatefulSet (ns/name) | StatefulSet域 | Pod DNS | Pod主机名 |
| --- | --- | --- | --- | --- | --- |
| cluster.local | default/nginx | default/web | nginx.default.svc.cluster.local | web-{0..N-1}.nginx.default.svc.cluster.local | web-{0..N-1} |
| cluster.local | foo/nginx | foo/web | nginx.foo.svc.cluster.local | web-{0..N-1}.nginx.foo.svc.cluster.local | web-{0..N-1} |
| kube.local | foo/nginx | foo/web | nginx.foo.svc.kube.local | web-{0..N-1}.nginx.foo.svc.kube.local | web-{0..N-1} |

* **稳定的存储**

kubernetes为每一个VolumeClaimTemplate创建一个对应的PersistentVolume。在前面的nginx实例中，每一个Pod将会my-storage-class存储类型的PersistenVolume单一实例和1Gib的存储空间。如果没有指定存储类，则会使用默认的存储。但一个Pod被调度到Node上，它的volumeMounts将会挂接PersistentVolumes，并将其与PersistentVolumeClaims进行关联。需要注意的是，即使在Pod被删除，PersistentVolumes与PersistentVolumeClaims之间的关联关系也不会被删除。

* **Pod名称标签**

当StatefulSet控制器创建了Pod，它将会添加一个标签，此Pod名称的集合。此标签将能够管理服务到指定的Pod。

#### 部署和扩容保证

* 对于一个带有N个副本集的StatefulSet，当Pod被部署，它们将按0到N-1的顺序被创建。
* 当一Pod被删除时，它们将按照N-1到0的倒序被终止。
* 在进行Pod扩容前，所有依赖的Pod应该都已在运行和准备好。
* 在Pod被终止前，所有的依赖它的Pod都必须完全停止。

在前文创建的nginx例子中，将按照顺序部署web-0,web-1和web-2。web-1只能在web-0运行和准备好以后才能够被部署，web-2只能在web-1运行和准备好以后才能够被部署。如果web-0失败，就算web-1正在运行，web-2也是不能正常启动的，除非web-0被重启，并正常运行。

如果缩容上述例子，设置replicas=1，则web-2首先被终止，接着是web-1。如果在web-2被终止后，但在web-1被终止前，web-0失败了，web-1将不能被终止，除非web-0处于正常运行状态。

#### Pod管理策略

在Kubernetes 1.7以后，StatefulSet的唯一性标识可以通过.spec.podManagementPolicy的值进行保证。

* OrderedReady Pod管理

OrderedReady pod管理是StatefulSets默认的管理模式，此模式安装顺序启动或者终止Pod。

* 并行Pod管理

并行Pod管理告诉StatefulSet控制器以并行的方式启动或者终止所有的Pod。

#### 更新策略

在Kubernetes 1.7之后，运行通过配置StatefulSet的.spec.updateStrategy，实现Pod的容器、标签、资源请求/限制和注释自动更新。

* **On Delete策略**

OnDelete更新策略是1.6之前版本的行为。当StatefulSet的.spec.updateStrategy.type被设置为OnDelete，则StatefulSet控制器将不会知道更新Pod。

* **Rolling Updates策略**

RollingUpdate更新策略将实现StatefulSet中Pod的自动滚动更新，这是StatefulSet的默认更新模式。如果.spec.updateStrategy.type设置为 RollingUpdate，则StatefulSet控制器将会删除和重建StatefulSet中的每一Pod。它将会按照从最大到最小的序数终止Pod，并按照从小到大顺序重建Pod。

* **Partitions策略**

RollingUpdate更新策略能够通过指定.spec.updateStrategy.rollingUpdate.partition进行分隔。当分隔被指定，所有序数大于或等于分隔的Pod将会被更新，其它的Pod将被不会进行更新。在大部分的情况下，不会使用分隔；当希望进行金丝雀发布，或者执行阶段发布时，分隔是很有用的。

### DaemonSet（守护进程）(Y)

DaemonSet是Kubernetes工作负载的一种，作为守护进程应用，DaemonSet用于确保在所有（或部分）Node节点上运行Pod的副本。随着新的Node被添加到群集中的同时，DaemonSet控制器会将Pod添加到新加入的Node节点中。当Node节点从群集中删除，垃圾收集器也将会删除这些Pod。DaemonSet的一些典型场景和用法：

* **集群存储：**在每个Node节点上，运行**集群存储**守护程序，例如*glusterd*，*ceph*。
* **日志收集：**在每个Node节点上，运行**日志收集**守护程序，例如*fluentd*或*logstash*。
* **节点监控：**在每个Node节点上，运行**节点监控**守护程序，例如Prometheus Node Exporter，collectd，Dynatrace OneAgent，Datadog agent，New Relic agent，Ganglia gmond或者Instana agent。

#### 创建DaemonSet

在这里创建一个fluented-elasticsearch的守护进程应用，首先，通过YAML文件定义一个DaemonSet。在此daemonset.yaml文件中，使用的镜像为k8s.gcr.io/fluentd-elasticsearch:v2.2.0。

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

namespace: kube-system

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

# 定义DaemonSet Pod信息

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

tolerations:

- key: node-role.kubernetes.io/master

effect: NoSchedule

containers:

- name: fluentd-elasticsearch

#容器所使用的镜像

image: k8s.gcr.io/fluentd-elasticsearch:v2.2.0

# 设置资源配额，

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

# 持久化存储挂接

volumeMounts:

- name: varlog

mountPath: /var/log

- name: varlibdockercontainers

mountPath: /var/lib/docker/containers

readOnly: true

terminationGracePeriodSeconds: 30

volumes:

- name: varlog

hostPath:

path: /var/log

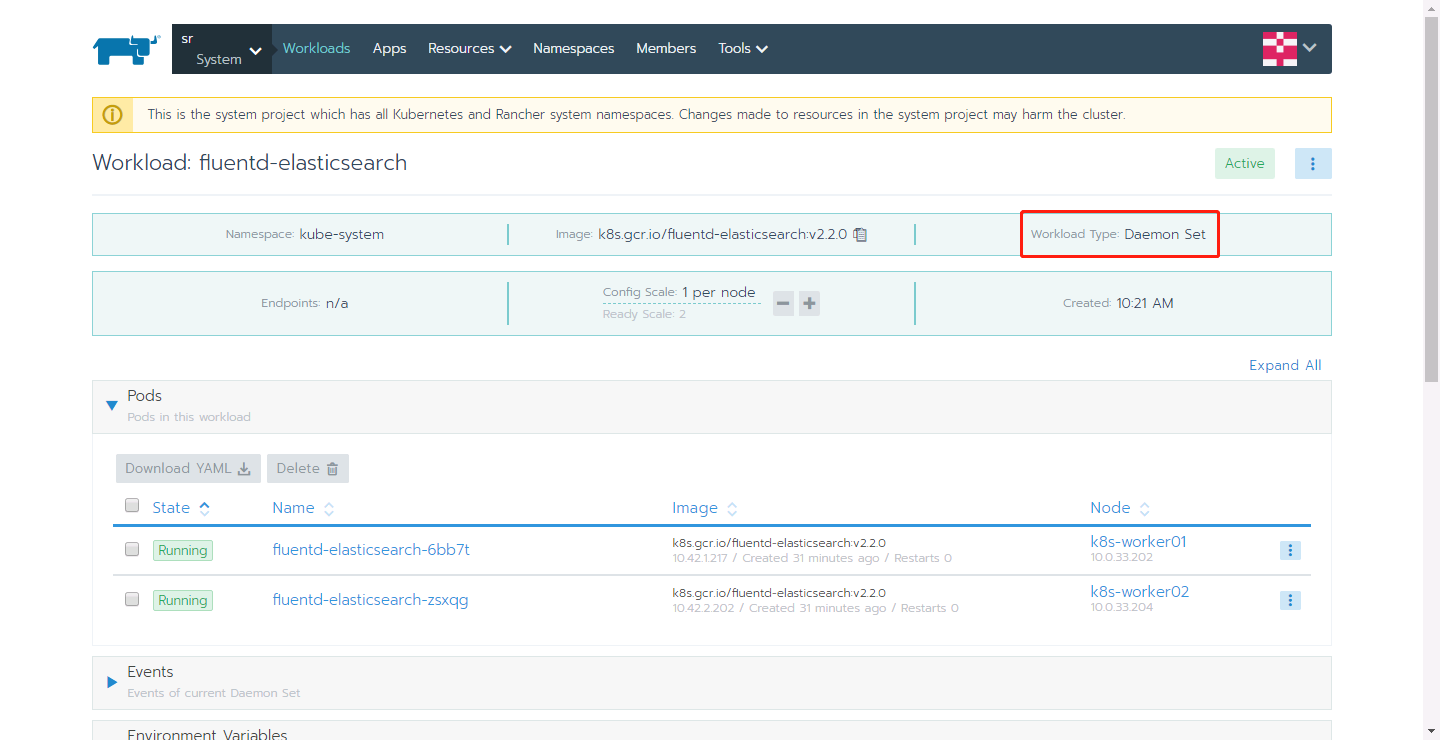
- name: varlibdockercontainers

hostPath:

path: /var/lib/docker/containers

在定义后YAML文件后，通过执行kubectl create命令创建DaemonSet：

# kubectl create -f {path}/daemonset.yaml



图

#### 调度DaemonSet Pod

DaemonSet的Pod和普通Pod一样，都需要进行调度，以在集群中特定的Node节点上部署运行。

##### 仅在某些节点上运行Pod

如果在YAML文件中设置了.spec.template.spec.***nodeSelector***，则DaemonSet控制器将在与该节点选择器匹配的Node节点上部署DaemonSet Pod 。同样，如果设置了.spec.template.spec.***affinity***，则DaemonSet控制器将在与该节点关联相匹配的Node节点上创建Pod 。如果未指定上述两个字段，则DaemonSet控制器将会在所有的Node节点上创建Pod。

##### 由默认调度器调度

DaemonSet确保所有符合条件的节点都运行Pod的副本。通常由Kubernetes调度器选择Pod所运行的Node节点。但是，DaemonSet Pod是由DaemonSet控制器所创建和调度的。这就产生了以下的问题：

* 不一致的Pod行为：正常的Pod在创建是处于Pending状态；但DaemonSet Pod在创建时，并不是处于Pending 状态。
* Pod抢占：启用抢占后，DaemonSet控制器将不考虑Pod优先级和抢占的情况下制定的调度决策。

ScheduleDaemonSetPods允许使用默认调度器而不是DaemonSet控制器来调度DaemonSet，方法是将NodeAffinity术语添加到DaemonSet Pods，而不是使用.spec.nodeName术语。默认调度器将Pod绑定到目标主机。如果DaemonSet Pod的节点关联已存在，则替换它。DaemonSet控制器仅在创建或修改DaemonSet Pod时执行这些操作，并且不对DaemonSet的spec.template进行任何更改。

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchFields:

- key: metadata.name

operator: In

values:

- target-host-name

此外，node.kubernetes.io/unschedulable:NoSchedule会自动添加到DaemonSet Pod。在调度DaemonSet Pod时，默认调度器会忽略unschedulable的Node节点。

#### DaemonSet对象相关参数

在定义DaemonSet对象时，可以在规格域中设置如下的参数，用于进行Pod可用准备时间、回滚记录和Pod模板等方面的控制。

|  |  |
| --- | --- |
| 参数 | 说明 |
| minReadySeconds | 在没有任何容器崩溃的情况下，新创建的DaemonSet Pod应该准备就绪的最小秒数，以使其对外可用。默认值为0，即一旦准备好，Pod将被视为可用。 |
| revisionHistoryLimit | 允许保留回滚的旧历史记录数。默认值为10。 |
| selector | 由守护程序集管理的用于查询Pod的标签，必须匹配才能被控制。它必须与Pod模板的标签匹配。 |
| template | 描述所要创建的Pod的对象，DaemonSet将在与模板的节点选择器（node selector）所匹配的每个主机节点上创建一个Pod副本（如果未指定节点选择器，则会在每个节点上创建）。 |
| updateStrategy | 用新Pod替换现有DaemonSet Pod的更新策略。 |

#### 查看日志

守护程序的日志可以帮助诊断问题，docker守护程序日志的具体存储位置取决于操作系统配置和使用的日志记录子系统：

| **操作系统** | **目录** |
| --- | --- |
| RHEL，Oracle Linux | /var/log/messages |
| Debian | /var/log/daemon.log |
| Ubuntu 16.04+，CentOS | 使用journalctl -u docker.service命令检查日志 |
| Ubuntu 14.10- | /var/log/upstart/docker.log |
| macOS（Docker 18.01+） | ~/Library/Containers/com.docker.docker/Data/vms/0/console-ring |
| macOS（Docker <18.01） | ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/console-ring |
| Windows | AppData\Local |

#### 启用调试

在daemon.json文件中将 debug键的值设置为true就可以启用docker的调试模式，此方法适用于每个Docker平台。

1. daemon.json文件通常位于/etc/docker/目录，如果该文件尚不存在，则需要手动创建此文件。
2. 如果文件为空，请添加以下内容：

{

"debug": true

}

如果文件中已经包含JSON，则只需添加键值对”debug”: true，如果它不是文件中的最后最后一行，请在行尾添加“；”。同时需要确认log-level是否已设置为info或debug。默认值为info，可能的值包括debug，info，warn，error，fatal。

1. 向守护程序发送hup信号以使其重新加载新的配置。在Linux主机上，使用以下命令。

$ sudo kill -SIGHUP $(pidof dockerd)

也可以手动停止Docker守护程序，并使用debug的-D字段标志重新启动它。

#### 强制记录堆栈跟踪

如果守护程序没有响应，则可以通过向SIGUSR1守护程序发送信号来强制记录完整堆栈跟踪。

$ sudo kill -SIGUSR1 $(pidof dockerd)

这会强制记录堆栈跟踪，但不会停止守护程序。守护程序日志显示堆栈跟踪或包含堆栈跟踪的文件的路径（如果已记录到文件）。处理完SIGUSR1信号并将堆栈跟踪转储到日志后，守护程序继续运行。堆栈跟踪可用于确定守护程序中所有goroutine和线程的状态。

可以使用以下方法之一查看Docker守护程序日志：

* 在Linux系统上执行journalctl -u docker.service命令；
* 在较旧的Linux系统上，查看/var/log/messages，/var/log/daemon.log或/var/log/docker.log。

在Docker日志中查找如下消息：

...goroutine stacks written to /var/run/docker/goroutine-stacks-2017-06-02T193336z.log

...daemon datastructure dump written to /var/run/docker/daemon-data-2017-06-02T193336z.log

Docker保存这些堆栈跟踪和转储的位置取决于操作系统和配置，有时可以直接从堆栈跟踪和转储中获取有用的诊断信息。

### CronJob（定时任务）（Y）

CronJob是一种能够定时运行的Job对象，它使用Linux系统上的定时任务Cron的语法来定义Job的固定运行时间，使用Cron Jobs能够以基于时间的计划运行相关的作业。这些自动化作业会在Linux或UNIX系统上像[Cron](https://en.wikipedia.org/wiki/Cron)任务一样运行。Cron Job对于创建定期和重复的任务非常有用，例如进行备份或发送电子邮件等。

#### 创建CronJob对象

在Kubernetes 1.8版本之前，Cron Job使用batch/v2alpha1版本的API，而默认情况下这个版本的API是被禁用的，因此需要通过设置参数--runtime-config=batch/v2alpha1=true进行启用。而在Kubernetes 1.8之后，CronJob使用batch/v1beta1版本的API，并且默认是启用的。下面是CronJob的一个例子：

apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: hellocronJob

spec:

schedule: "\*/1 \* \* \* \*" #表示每各一分钟执行一次

jobTemplate:

spec:

template:

spec:

containers:

- name: hellocronjob

image: busybox

args:

- /bin/sh

- -c

- date; echo hellocronjob from the Kubernetes cluster

restartPolicy: OnFailure

在定义好cronjob的yaml文件后，可以通过kubectl create -f命令创建cronjob：

$ kubectl create -f ./cronjob.yaml

cronjob "hellocronjob" created

或者也可以通过下面的命令方式运行一个CronJob：

$ kubectl run hellocronjob --schedule="\*/1 \* \* \* \*" --restart=OnFailure --image=busybox -- /bin/sh -c "date; echo hellocronjob from the Kubernetes cluster"

cronjob "hellocronjob" created

在创建好cronjob后，可以通过kubectl get cronjob命令能够获取所创建的cronjob：

$ kubectl get cronjob hellocronjob

NAME SCHEDULE SUSPEND ACTIVE LAST-SCHEDULE

hello \*/1 \* \* \* \* False 0 <none>

#### 删除CronJob

如果不再使用所创建CronJob，则可以删除CronJob。通过kubectl delete cronjob命令来删除一个CronJob：

$ kubectl delete cronjob hellocronjob

cronjob "hellocronjob" deleted

删除一个CronJob对象会连带删除其创建的所有Job和Pod。

#### CronJob对象相关参数

在定义CronJob对象时，可以在规格域中设置如下的参数，用于进行Job并行工作策略、执行周期和Job模板等方面的控制。

|  |  |
| --- | --- |
| 领域 | 描述 |
| concurrencyPolicy | 指定如何处理作业的并发执行情况，可选值为：  - “Allow”（默认值）：允许CronJobs同时运行多个作业;  - “Forbid”：禁止并发运行多作业，如果之前的运行尚未完成，则不会运行下一个作业;  - “Replace”：取消当前正在运行的作业，并将其替换为新作业。 |
| failedJobsHistoryLimit | 需要保留的已失败作业的数量，默认值为1。 |
| jobTemplate | 指定执行CronJob时将创建的作业。 |
| schedule | Cron格式的时间表。，用于描述CronJob的执行周期。 |
| startingDeadlineSeconds | 在CronJob没有按时启动（可能规定时间时CronJob controller恰好出现故障）时，用于指定CronJob能够存活的时间。如果超过了这个设定的时间，那么此CronJob就会被标记为Failed。 |
| successfulJobsHistoryLimit | 要保留的成功完成工作的数量。这是一个区分显式零和未指定的指针。默认为3。 |
| suspend | 此参数用于告诉控制器暂停后续认为执行，它不影响于已经执行的任务，默认值为false。 |

对这些参数，其中相当复杂和重用的是schedule，它是一个Cron类型的字符串，用于描述CronJob的执行周期，其格式如下：

Minutes Hours DayofMonth Month DayofWeek Year

格式的说明如下：

* minute：表示分钟，可以是从0到59之间的任何整数。
* hour：表示小时，可以是从0到23之间的任何整数。
* day：表示日期，可以是从1到31之间的任何整数。
* month：表示月份，可以是从1到12之间的任何整数。
* week：表示星期几，可以是从0到7之间的任何整数，这里的0或7代表星期日。
* yea：表示年，此值为可选。

 上述格式的值支持 ", - \* / "四中字符，这四种字符分别表示：

* \* ：表示匹配任意值，如果在Minutes中使用，表示每分钟；
* / ：表示起始时间开始触发，然后每隔固定时间触发一次；
* , ：可以用逗号隔开的值指定一个列表范围，例如，“1,2,5,7,8,9”；
* - ：可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2,3,4,5,6”。

例如在Minutes 设置的是3/20，则表示第一次触发是在第3min时，接下来每20min触发一次，即，第23min，43min等时刻触发

示例：比如每隔1min执行一次任务：则Cron 表达式如下：

\*/1 \* \* \* \*

## 服务发现（2019-04-12 Y）

### Service（服务）整体介绍

在Kubernetes中，Pods是有生命周期的。它们被创建、被终止，但不能被复活。在Kubernetes中通过ReplicationControllers动态的创建和删除Pod。然后，每一个Pod都拥有自己的IP地址，但是这些IP地址随着时间会发生变化。这会导致一个问题：如果在Kubernetes集群中，前端的Pod需要调用后端的Pod的功能，那么这些前端的Pod如何发现和跟踪后端的Pod？

在Kubernetes中，Service是一个抽象的概念，它定义了Pod逻辑集合和访问这些Pod的策略。Service通过Label Selector选择Pod。例如，在后端运行着有3个副本的Pod，这些副本是可互相替换的，前端不需要关注使用那个副本。Service抽象就用来实现此解耦的能力的。对于 Kubernetes-native的应用，当Service中的Pod发生变化时，Kubernetes通过了Endpoints API类进行更新。对于non-native applications, Kubernetes 提供了virtual-IP-based桥，通过它重定向后端的Pod。在本文中，描述如何定义Service、发布Service和发现Serivce的整个过程。

#### 虚拟IP和服务代理

在Kubernetes的每一个Node中，都运行着一个kube-proxy，kube-proxy负责为服务（ExternalName除外）实现虚拟IP的格式。在Kubernetes v1.0中，服务是一个4层（IP之上的TCP/UDP）结构，纯粹在userspace实现代理；在Kubernetes v1.1，增加了Ingress API，它表达了7层（HTTP）服务；也增加了iptables代理，此代理是Kubernetes v1.2后的默认代理模式；在Kubernetes v1.8.0-beta.0, 增加了ipvs代理。

在iptables模式中，kube-proxy通过创建iptables规则，将访问Service虚拟IP的请求重定向到Endpoints上，iptables代码模式方式利用linux的iptables nat转发进行实现。kube-proxy监控Kubernetes master中的Service和Endpoints对象，并进行添加和移除，以更新iptables规则。

* 对于每一个Service，它将会安装iptable规则，此规则获取流量发送给至Service‘s clusterIP和端口，并将这些流量传递给Service后端的集合。
* 对于每一个Endpoints对象，它将安装iptable规则，用于选择后端的Pod。

显然，iptable不需要在userspace和kernelspace之间进行转换，它比userspace更快更可靠。不像userspace代理器，如果一个Pod被提供服务，iptable代理器不能自动的使用另外一个Pod。

#### 定义服务

在Kubernetes中，服务是一个REST对象，类似于Pod。像其它所有的REST对象一样，服务定义能够被传递给apiserver来创建一个新的实例，例如，这里有一组Pod，对外暴露的端口为9367，标签为app：MyApp：

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

**selector:**

**app:MyApp**

**ports:**

**- protocol:TCP**

**port:80  #暴露在cluster Ip上的端口，供集群内部使用**

**targetPort:9376  #pod上的端口**

在此配置文件中，创建了一个名为“my-service”的服务对象，在每一个标签带有“app=MyApp”的Pod上，它的targetPort为9376。此服务将被指派一个IP地址（有时也称为“cluster IP”），服务选择器将被持续的评估，评估的结果将被传递给名称也为“my-service”的Endpoints对象。

需要注意的是，服务能够映射一个输入端口至任意的targetPort。在默认情况下，targetPort将被设置成与port的值一样。targetPort可以是一个字符串，可以引用后端Pod中的port名称。基于后端Pod的不同，实际的port值也会不同。这就为部署服务提供大量的灵活性。Kubernetes服务支持TCP和UDP协议，默认为TCP协议。

#### 无选择器的服务

Service一般被用来代理访问Pod，但也能够代理后端的其他类型，例如：

* 在生产环境中使用外部的数据库，但在测试环境中使用集群内的数据；
* 服务将需要被另外的命名空间或者另外的集群上的服务调用；
* 正在迁移应用至Kubernetes，并且一些后端在Kubernetes外运行。

在上述的这些场景中，可以定义无选择器的Service：

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

**ports:**

**- protocol:TCP**

**port:80 #Cluster IP上的端口为80，此端口仅可以在集群内访问**

**targetPort:9376 #Pod上的端口**

因为此Service没有选择器，则不会创建对应的Endpoints对象，您可以手工将服务映射至指定的endpoints中：

**kind:Endpoints**

**apiVersion:v1**

**metadata:**

**name:my-service**

**subsets:**

**- addresses:**

**- ip:1.2.3.4**

**ports:**

**- port:9376**

注意：Endpoint IP不可以是loopback（127.0.0.0/8）， link-local (169.254.0.0/16), 或者link-local multicast (224.0.0.0/24)。 访问无选择器的Service与访问有选择器的Service是一样。流量将通过用户定义的endpoints进行路由。

#### ExternalName服务

ExternalName Service是Service的一个特例，它没有选择器，也没有定义任何端口或Endpoints。它的作用是返回集群外Service的外部别名。

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**namespace:prod**

**spec:**

**type:ExternalName #服务类型为外部服务**

**externalName:my.database.example.com #外部服务**

当查找my-service.prod.svc.CLUSTER时，集群DNS服务将会返回一条CNAME记录，此记录的值为my.database.example.com。当然后续也可以将此数据库迁移到集群中，这样就可以通过Pod启动，并为其添加合适的选择器或者Endpoints，并修改服务类型。

#### headless服务

在有些场景下，服务可能不需要作为负载均衡代理，而仅仅需要一个单一的cluster ip。这时就可以通过设置“**spec.clusterIP”**的值为**None**，来创建一个”headless“类型的服务。此类型的服务允许开发者减少对Kubernetes系统的依赖，开发者可以通过自己的方式实现对服务的自动发现。应用也能够使用其他的服务发现系统，进行服务的自注册和适配器，实现服务的自动发现。对于这样的服务：

* Kubernetes未指派cluster IP；
* kube-proxy将不处理这些服务；
* 因此也就没有负载均衡和代理；
* 但会依赖服务是否拥有选择器进行DNS的配置。

对于定义了选择器的headless service，Endpoints控制器在API中创建Endpoints记录，并通过修改DNS的配置信息返回一条记录，此记录指向服务后端的Pod。对于没有定义选择器的headless service，Endpoints控制器不会创建Endpoints记录，然而，DNS系统将会进行寻址和配置。

* CNAME记录：ExternalName类型的服务；
* Endpoints记录：任意与service共享一个名称的Endpoints。

#### 多端口服务

在实际的应用场景中，有一些服务需要暴露多个端口。在Kubernetes中，支持在Service对象上定义多个端口。当使用多个端口时，则需要为每个端口设置一个名称。例如，下面名称为my-service的服务YAML配置文件，它暴露了一个http的端口和一个https的端口。

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

**selector:**

**app:MyApp**

**ports:**

**- name:http #名称为http的端口**

**protocol:TCP**

**port:80**

**targetPort:9376**

**- name:https #名称为https的端口**

**protocol:TCP #端口协议为TCP**

**port:443 #ClusterIP端口号为443**

**targetPort:9377 #**Pod上的端口为9377

#### 代理外部的服务

另外，在一些场景下，Kubernetes中的容器化应用需要调用集群外的应用。Service也可以代理任意其它的后端应用，比如运行在Kubernetes集群外部的Oracle、MySQL和Redis等。在Kubernetes中，通过定义同名的Service和EndPoints来实现对于外部应用的代理，以实现其能够被集群内部的应用调用。

下面是接入外部Oralce端点YAML文件：

apiVersion: v1

kind: Endpoints

metadata:

name: oracle-service

subsets:

- addresses:

- ip: 192.168.8.159

ports:

- port: 1521

protocol: TCP

下面是代理外部Oralce的服务YAML文件：

apiVersion: v1

kind: Service

metadata:

name: oracle-service

spec:

ports:

- port: 1521

targetPort: 1521

protocol: TCP

### 发现服务

在服务定义好，接下来要考虑的事情就是这些服务如何被其它服务和用户进行访问和调用。在Kubernetes中，支持两种服务发现的模式：

* 环境变量：通过服务的环境变量发现服务；
* DNS：通过。

#### 环境变量

当一个Pod运行在Node上时，kubelet将为每一个活动的Service添加环境变量，环境变量有两类：

* DockerLink 环境变量：相当于Docker的–link参数实现容器连接时设置的环境变量。
* Kubernetes Service环境变量：Kubernetes为Service设置的环境变量形式，{SVCNAME}\_SERVICE\_HOST 和{SVCNAME}\_SERVICE\_PORT变量，环境变量的名称为大写字母和下划线。

例如：存在一个名称为“redies-master”的Service（它的cluster ip地址为10.0.0.11，端口号为6379，协议为TCP），它的环境变量如下：

#Kubernetes Service环境变量：

REDIS\_MASTER\_SERVICE\_HOST=10.0.0.11

REDIS\_MASTER\_SERVICE\_PORT=6379

#Docker Link环境变量：

REDIS\_MASTER\_PORT=tcp://10.0.0.11:6379

REDIS\_MASTER\_PORT\_6379\_TCP=tcp://10.0.0.11:6379

REDIS\_MASTER\_PORT\_6379\_TCP\_PROTO=tcp

REDIS\_MASTER\_PORT\_6379\_TCP\_PORT=6379

REDIS\_MASTER\_PORT\_6379\_TCP\_ADDR=10.0.0.11

在这里，可以看到环境变量中记录了”redies-master”服务的IP地址和端口，以及协议信息。因此，Pod中的应用就可以通过环境变量来发现此服务。但是，环境变量方式存在如下的限制：

1）环境变量只能在相同的命名空间中使用；

2）另外，Service必须在Pod创建之前被创建，否则Service变量不会被设置到Pod中；

3）DNS服务发现机制则没有这些限制。

#### DNS

DNS服务发现是基于Cluster DNS的，DNS服务器会对新服务进行监控，并为每一个服务创建DNS记录，用于域名解析。在集群中，如果启用DNS，则所有的Pod都可以自动通过名称解析服务。

例如，如果在“my-ns”命名空间下拥有一个名为“my-serivce”的服务，则会有一个名为“my-service.my-ns”的DNS记录被创建。

* 在“my-ns”命名空间下，Pod将能够通过名称“my-service”来发现此服务。
* 在其它命名空间，Pod必须通过“my-serivce.my-ns”来发现此服务，此名称选址的结果即为cluster IP。

Kubernetes也支持端口的DNS SRV（serivce）记录。如果 “my-service.my-ns”服务拥有一个TCP协议名称为”http”的端口，就能够通过”\_http.\_tcp.my-service.my-ns”名称来发现”http”端口的值。Kubernetes DNS服务器是发现ExternalName类型服务的唯一途径。

### 发布服务和服务类型

对于某些应用（例如：前端）的一部分功能，可能需要暴露一个使用外部IP地址的Sevice。通过Kubernetes的ServiceType，能够指定所使用的service类型。在默认的情况下，服务为ClusterIP类型。Kubernetes的服务类型如下：

* *ClusterIP***(default)** – 将服务暴露在集群内部的IP，此类型仅支持在集群内服务。
* *NodePort*– 将服务暴露在所选定每一个Node的同一端口，集群外可以通过**<NodeIP>:<NodePort>**方式访问服务。
* *LoadBalancer* – 在当前的集群中创建一个外部的负载均衡，并为服务（service）指派一个固定的、外部的。
* *ExternalName* – 使用一个随意的名称（在规格中指定）来暴露服务，并会返回一个带有名称的CNAME记录。此类型不使用代理，这种类型只在kube-dns v1.7上才支持。

#### 主机端口（NodePort）类型

如果Service的type为“NodePort”，则Kubernetes master将会在每一个Node为此Service暴露一个对外的端口（默认：30000-32767）。外部网络将能够通过[NodeIP]:[NodePort]对服务进行访问。也可以通过nodePort指定此端口，但此端口的值必须在“30000-32767”范围内，手动指定的话需要注意端口存在冲突的可能性。此类型使开发者能够自由的设置自己的负载均衡，即也可以采用Kubernetes未支持的负载均衡技术。

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

type：NodePort   #指定Service类型为NodePort

**selector:**

**app:MyApp**

**ports:**

**- protocol:TCP**

**port:80**

**targetPort:9376**

#### 负载均衡（LoadBalancer）类型

负载均衡服务是类型为LoadBalance的服务，它建立在NodePord类型服务的基础上。Kubernetes会分配给LoadBalance服务一个内部的虚拟IP，并且暴露NodePort。通过LoadBalanceIP进来的请求，将会被转发给NodePort。

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

**selector:**

**app:MyApp**

**ports:**

**- protocol:TCP**

**port:80**

**targetPort:9376**

**clusterIP:10.0.171.239**

**loadBalancerIP:78.11.24.19**

**type:LoadBalancer #指定Service的类型为LoadBalancer**

**status:**

**loadBalancer:**

**ingress:**

**- ip:146.148.47.155**

来自于外部负载均衡的流量将被直接引到后端的Pod中。

#### 外部IP

如果这里有一些外部IP，通过它们能够路由至一个或者多个集群的Node，Kubernetes服务将可以被暴露在这些externalIPs上。通过外部IP（作为目标IP），ingress导入到集群流量的将被路由到其中的一个服务endpoint上。externalIPs由集群管理员进行管理。所有的服务类型都可以指定externalIPs，在下面的“my-service”服务中，客户端口可以通过“80.11.12.10:80”外部端口来访“my-service”服务。

**kind:Service**

**apiVersion:v1**

**metadata:**

**name:my-service**

**spec:**

**selector:**

**app:MyApp**

**ports:**

**- name:http**

**protocol:TCP**

**port:80**

**targetPort:9376**

**externalIPs:  #定义外部ip地址**

**- 80.11.12.10**

### 服务的相关kubectl命令

在此部分列示了与服务相关的一些kubectl命令，服务即可以通过YAML，也可以直接通过命令创建。

**1）kubectl create service clusterip**

此命令用于创建一个ClusterIP类型的服务，示例如下：

$ kubectl create service clusterip my-cs --tcp=5678:8080

**2）kubectl create service externalname**

此命令用于创建一个ExternalName类型的服务，示例如下：

$ kubectl create service externalname my-ns --external-name bar.com

ExternalName类型的服务引用外部DNS地址而不是仅POD，这将允许应用程序可以引用其他平台上、其他集群和本地的服务。

**3）kubectl create service loadbalancer**

此命令用于创建一个LoadBalancer类型的服务，示例如下：

$ kubectl create service loadbalancer my-lbs --tcp=5678:8080

**4）kubectl create service nodeport**

此命令用于创建一个NodePort类型的服务，示例如下：

$ kubectl create service nodeport my-ns --tcp=5678:8080

**5）kubectl expose**

此命令用于为资源暴露服务，这些资源包括pod (po), service (svc), replicationcontroller (rc), deployment (deploy), replicaset (rs)。下面是为nginx部署暴露服务的示例：

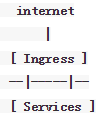
$ kubectl expose deployment nginx --port=80 --target-port=8000

只有当其拥有的选择器可转换为服务支持的选择器时，即当选择器只包含matchLabels组件时，部署或副本集才会作为服务公开。注意，如果没有通过–port指定端口，并且被暴露的资源具有多个端口，则服务将会使用这些端口。此外，如果没有指定标签，新服务将会使用来自资源的标签。

### Ingress（反向代理）

#### ****Ingress简介****

在Kubernetes中，服务和Pod的IP地址仅可以在集群网络内部使用，对于集群外的应用是不可见的。为了使外部的应用能够访问集群内的服务，在Kubernetes中可以通过NodePort和LoadBalancer这两种类型的服务，或者使用Ingress。Ingress本质是通过http代理服务器将外部的http请求转发到集群内部的后端服务。Kubernetes目前支持GCE和nginx控制器；另外，F5网络为Kubernetes提供了F5 Big-IP控制器。通过Ingress，外部应用访问群集内容服务的过程如下所示。



Ingress控制器通常会使用负载均衡器来负责实现Ingress，尽管它也可以通过配置边缘路由器或其它前端以HA方式处理流量。

#### ****Ingress配置文件****

下面是Ingress YAML配置文件的示例：

**apiVersion:** **extensions/v1beta1**

**kind:** **Ingress**

**metadata:**

**name:** **test-ingress**

**annotations:** **nginx.ingress.kubernetes.io/rewrite-target:** **/**

**spec:**

**rules:**

**-** **http:**

**paths:**

**-** **path:** **/testpath**

**backend:**

**serviceName:** **test**

**servicePort:** **80**

* **1-6行**：Ingress YAML文件中的1-6行与其它的Kubernetes配置文件一样，需要**apiVersion**, **kind**和**metadata**字段。此示例定义了名称为test-ingress的Ingress。
* **7-9行**：Ingress规格具有配置负载均衡器或代理服务器所需的所有信息。最重要的是，它包含与所有传入请求相匹配的规则列表。目前，Ingress资源仅支持http规则。
* **10-11行**：每个http规则都包含以下信息：一个**主机**（例如：foo.ba.com，在这个例子中为\*），一个路径列表（例如：/testpath），每个路径都有一个相关的后端（**test：80**）。在负载均衡器将业务引导到后端之前，主机和路径都必须匹配传入请求的内容。
* **12-14行**：后端是服务:端口（**test：80**）的组合。Ingress流量通常被直接发送到与后端相匹配的端点。

#### ****Ingress类型****

##### ****代理单一服务****

Kubernetes可以使用LoadBalancer和NodePort类型的服务暴露服务，也可以通过一个Ingress来实现。下面是名称为test-ingress的Ingress YAML文件，它将对外代理名称为testsvc的服务。

**apiVersion:** **extensions/v1beta1**

**kind:** **Ingress**

**metadata:**

**name:** **single-ingress**

**annotations:** **nginx.ingress.kubernetes.io/rewrite-target:** **/**

**spec:**

**rules:**

**-** **host:** **foo.bar.com**

**http:**

**paths:**

**-** **path:** **/foo**

**backend:**

**serviceName:** **s1**

**servicePort:** **80**

通过kubectl create -f创建上述的Ingress，在创建后可以通过kubectl get ing的命令获取Ingress的列表信息：

**$ kubectl get ing**

**NAME RULE BACKEND ADDRESS**

**single-ingress –s1**:80 107.178.254.228

##### ****代理多个服务****

如前所述，在kubernetes 中Pod的IP地址只能对群集内的其它的应用可见。因此，如果需要接受集群外部的流量，并将其代理到集群中后端服务。在此示例中，通过foo.bar.com(IP地址为：178.91.123.132)主机作为代理服务器。当路径为http://foo.bar.cm:80/foo时，将会访问后端的s1服务；当路径为http://foo.bar.cm:80/bar时，将会访问后端的s2服务。

**foo.bar.com -> 178.91.123.132 -> / foo s1:80 / bar s2:80**

针对上述场景，Ingress的YAML配置文件如下所示：

**apiVersion:** **extensions/v1beta1**

**kind:** **Ingress**

**metadata:**

**name:** **mult-ingress**

**annotations:** **nginx.ingress.kubernetes.io/rewrite-target:** **/**

**spec:**

**rules:**

**-** **host:** **foo.bar.com**

**http:**

**paths:**

**-** **path:** **/foo**

**backend:**

**serviceName:** **s1**

**servicePort:** **80**

**-** **path:** **/bar**

**backend:**

**serviceName:** **s2**

**servicePort:** **80**

通过kubctl create -f命令创建上述的Ingress：

**$ kubectl get ing**

**NAME RULE BACKEND ADDRESS**

**mult-ingress – foo.bar.com /foo s1:80 /bar s2:80**

默认后端：没有规则的入口，就像前一节中所示的那样，会将所有的流量发送到一个默认的后端。通过指定一组规则和默认后端，可以使用相同的技术来告诉负载均衡器，可以在哪里能够找到网站的404页。如果在Ingress中没有与请求头中主机相匹配的主机，并且/或者没有与请求的URL相匹配的路径，那么路由将被路由到默认的后端。

## 配置与存储(2019-04-02)

### Volume（存储卷）

#### 存储卷概述

由于容器本身是非持久化的，因此在容器中运行应用程序会遇到的一些关于持久化的问题。首先，当容器崩溃时，Kubernetes的kubelet将会重新启动容器，因此临时写入容器的文件将会丢失；然后，容器将会以镜像的初始状态实例化，并重新以容器的形式运行。其次，同一个Pod中一起运行的容器，通常需要共享容器之间一些文件。对于上述两个问题，Kubernetes通过存储卷来解决。

在Docker技术中本身就有存储卷的概念，但Docker中存储卷只是磁盘的或另一个容器中的目录，并没有对其生命周期进行有效的管理。Kubernetes的存储卷有自己的生命周期，它的生命周期与所在Pod的生命周期一致。因此，相比于在Pod中运行的容器来说，存储卷的存在时间会比其中任何的容器存在时间都要长，并且在容器重新启动时会保留数据。当然，当Pod不复存在时，存储卷也将不再存在。Kubernetes支持多种类型的存储卷，而Pod可以同时使用各种类型和任意数量的存储卷。在Pod中通过指定下面的字段来使用存储卷：

* spec.volumes：通过此字段提供指定的存储卷；
* spec.containers.volumeMounts：通过此字段将存储卷挂接到容器中。

#### 存储卷类型和示例

当前版本的Kubernetes支持如下所列这些存储卷类型，本书以hostPath、nfs和persistentVolumeClaim类型的存储卷为例，介绍如何定义存储卷，以及如何在Pod中被使用这些存储卷。

* awsElasticBlockStore
* azureDisk
* azureFile
* cephfs
* configMap
* csi
* downwardAPI
* emptyDir
* fc (fibre channel)
* flocker
* gcePersistentDisk
* gitRepo
* glusterfs
* hostPath
* iscsi
* local
* nfs
* persistentVolumeClaim
* projected
* portworxVolume
* quobyte
* rbd
* scaleIO
* secret
* storageos
* vsphereVolume

##### hostPath类型的存储卷

hostPath类型的存储卷用于将容器所在的宿主机的文件系统中文件或目录直接挂接到Pod中，除了需要指定path字段之外，在使用hostPath类型的存储卷时，也可以设置type，type支持的枚举值由下表所示。另外在使用hostPath时，需要注意下面的事项：

* 具有相同配置的Pod（例如：从同一个podTemplate创建的），可能会由于主机节点Node中文件的不同，从而导致行为不同。
* 在容器所在的宿主机上创建的文件或目录，只有root用户具写入的权限。因此，要么在容器中以root身份运行进程，要么在主机上修改的文件或目录的权限，以便具备写入内容到hostPath的存储卷中的权限。

| 值 | 行为 |
| --- | --- |
|  | 空字符串（默认）是用于向后兼容，如果为空字符串，则在挂接主机路径存储卷之前不执行任何检查。 |
| DirectoryOrCreate | 如果type的值为DirectoryOrCreate，在path指定目录不存在的情况下，则会在宿主机上创建一个新的目录，并设置目录权限为0755，此目录与kubelet拥有一样的组和拥有者。 |
| Directory | 如果type的值为Diretory，则在宿主机上，path指定的目录必需已经存在。 |
| FileOrCreate | 如果type的值为FileOrCreate，在path指定的文件不存在的情况下，则会在宿主机上创建一个空的文件，设置权限为0644，此文件与kubelet拥有一样的组和拥有者。 |
| File | 如果type的值为File，则path指定的文件必需存在。 |
| Socket | 如果type的值为Socket，则path指定的UNIX socket必需存在 |
| CharDevice | 如果type的值为CharDevice，这path指定的字符设备必需存在。 |
| BlockDevice | 如果type的值为BlockDevice，在path给定路径上必须存在块设备。 |

下面是使用hostPath作为存储卷的YAML文件示例，此YAML文件定义了一个名称为hostpath-demo的Pod资源。它通过hostPath类型的存储卷，将Pod所在宿主机上的/data目录挂接到容器中的/hostpath-demo目录。

apiVersion: v1

kind: Pod

metadata:

name: hostpath-demo

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: hostpath-demo

# 指定在容器中挂接的目录

volumeMounts:

- mountPath: /hostpath-demo

name: hostpath-volume

# 指定所提供的存储卷

volumes:

#存储卷名称，需要与volumeMounts的名称对应

- name: hostpath-volume

hostPath:

# 宿主机上的目录

path: /data

type: Directory

##### nfs类型的存储

在Kubernetes中，可以通过nfs类型的存储卷将现有的NFS（网络文件系统）到的挂接到Pod中。在移除Pod时，NFS存储卷中的内容并不会被删除，只是将存储卷卸载而已。这意味着在NFS存储卷总可以预先填充数据，并且可以在Pod之间进行数据共享。NFS可以被同时挂接到多个Pod中，并能同时进行写入。需要注意的是，在使用NFS存储卷之前，必须已正确部署和运行NFS服务器，并已经设置了共享目录。

下面是一个redis部署的YAML配置文件，在容器中的持久化数据被保存在/data目录下；存储卷使用nfs，nfs的服务地址为：192.168.8.150，存储路径为：/k8s-nfs/redis/data。容器通过volumeMounts.name的值确定所使用的存储卷。

apiVersion: apps/v1

kind: Deployment

metadata:

  name: redis

spec:

  selector:

    matchLabels:

      app: redis

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: redis

    spec:

      containers:

      # 应用的镜像

      - image: redis

        name: redis

        imagePullPolicy: IfNotPresent

        # 应用的内部端口

        ports:

        - containerPort: 6379

          name: redis6379

        env:

        - name: ALLOW\_EMPTY\_PASSWORD

          value: "yes"

        - name: REDIS\_PASSWORD

          value: "redis"

# 持久化挂接位置，在容器挂接目录为/data

        volumeMounts:

        - name: redis-persistent-storage

          mountPath: /data

      volumes:

      # nfs存储信息

      - name: redis-persistent-storage

**nfs:**

**# 在nfs服务器中的目录**

**path: /k8s-nfs/redis/data**

**# nfs服务器的ip地址**

**server: 192.168.8.150**

##### PersistentVolumeClaim

PersistentVolumeClaim类型存储卷将PersistentVolume挂接到Pod中作为一类存储卷。使用此类型的存储卷，用户并不需要知道存储卷的详细信息。

在此处定义了名为busybox-deployment的YAML部署配置文件，使用的镜像为busybox。基于busybox镜像的容器需要对**/mnt**目录下的数据进行持久化，在YAML文件中指定使用名称为nfs的PersistenVolumeClaim对容器的数据进行持久化。

# This mounts the nfs volume claim into /mnt and continuously

# overwrites /mnt/index.html with the time and hostname of the pod.

apiVersion: v1

kind: Deployment

metadata:

name: busybox-deployment

spec:

replicas: 2

selector:

name: busybox-deployment

template:

metadata:

labels:

name: busybox-deployment

spec:

containers:

- image: busybox

command:

- sh

- -c

- 'while true; do date > /mnt/index.html; hostname >> /mnt/index.html; sleep $(($RANDOM % 5 + 5)); done'

imagePullPolicy: IfNotPresent

name: busybox

volumeMounts:

- name: nfs

mountPath: "/mnt"

volumes:

- name: nfs

persistentVolumeClaim:

claimName: nfs-pvc

### PersistentVolume（持久化数据卷）

#### ****持久化存储卷和声明介绍****

持久化存储卷PersistentVolume（PV）用于为用户和管理员提供如何存储持久化数据的能力，由管理员在集群中提供的持久化存储卷。在集群中，持久化存储卷就像主机节点Node一样是集群中的一种资源。

持久化存储卷PersistentVolume（PV）也是和存储卷一样的一种插件，但其有着自己独立的生命周期。PersistentVolumeClaim (PVC)是用户对存储的请求，类似于Pod消费主机节点Node资源，PersistentVolumeClaim (PVC)消费持久化存储卷PersistentVolume（PV）资源。Pod能够请求特定的资源(CPU和内存)，声明请求特定的存储大小和访问模式。持久化存储卷PersistentVolume（PV）是一个系统的资源，因此没有所属的命名空间。

#### ****持久化存储卷和声明的生命周期****

在Kubernetes集群中，持久化存储卷PersistentVolume（PV）作为存储资源存在。PersistentVolumeClaim (PVC)是对持久化存储卷PersistentVolume（PV）资源的请求和使用，也是对PV存储资源的“提取证明”，而Pod会通过PersistentVolumeClaim (PVC)来使用持久化存储卷PersistentVolume（PV）。持久化存储卷PersistentVolume（PV）和 PersistentVolumeClaim (PVC)之间的交互过程有着自己的生命周期，这个生命周期分为5个阶段：

* + **供应(Provisioning)**：即PersistentVolume的创建，可以直接创建PersistentVolume（静态方式），也可以使用StorageClass动态创建。
  + **绑定（Binding）**：此阶段的目的是将PersistentVolume分配给PersistentVolumeClaim。
  + **使用（Using）**：在此阶段，Pod将通过PersistentVolumeClaim使用该Volume。
  + **释放（Releasing）**：在此阶段，Pod将释放Volume，并删除PersistentVolumeClaim。
  + **回收（Reclaiming）**：回收PersistentVolume，可以保留PersistentVolume以便下次使用，也可以直接从云存储中删除。

根据上述的5个阶段，存储卷的存在下面的4种状态：

* + **Available**：可用状态，表明PersistentVolume已经准备就绪，处于可以被PersistentVolumeClaim使用的状态。
  + **Bound**：绑定状态，表明PersistentVolume已被分配给了PersistentVolumeClaim。
  + **Released**：释放状态，表明PersistentVolumeClaim解绑了PersistentVolume，但还未执行回收策略。
  + **Failed**：错误状态，表明PersistentVolume发生了一些错误。

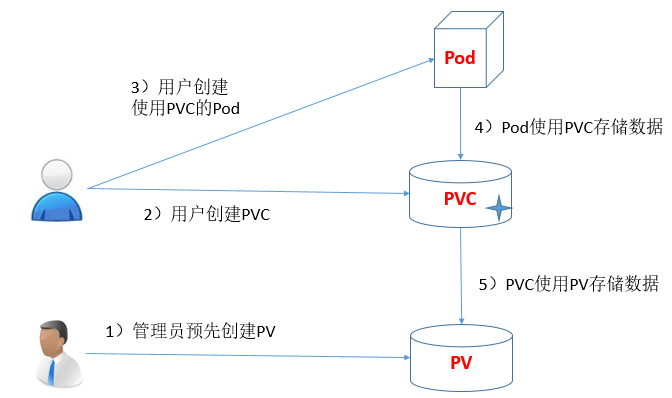
##### ****供应（Provisioning）****

供应是为集群提供可用的存储卷，在Kubernetes中有两种持久化存储卷的提供方式：静态方式或者动态方式。

###### ****静态(Static)****

在集群中，PersistentVolume是由Kubernetes的集群管理员创建的，PersistentVolume代表实际的存储，PersistentVolume提供的这些存储对于集群中所有的用户都是可用的。这些存储存在于Kubernetes API中，可被Pod作为实际的存储使用。在静态供应的情况下，由集群管理员预先创建PersistentVolume，开发者创建PersistentVolumeClaim和Pod，Pod通过PersistentVolumeClaim使用PersistentVolume提供的存储。

静态供应方式的过程如下图所示：



###### ****动态（Dynamic）****

对于动态的提供方式，当管理员创建的静态PersistentVolume都不能够匹配用户的PersistentVolumeClaim时，集群会尝试自动为PersistentVolumeClaim提供一个存储卷，这种提供方式基于StorageClass。在动态提供方式中，PersistentVolumeClaim需要请求一个存储类，但此存储类必须有管理员预先创建和配置。集群管理员需要在API Server中启用DefaultStorageClass的接入控制器。

动态供应过程如下图所示：



##### ****绑定****

在Kubernetes中，会动态的将PersistentVolumeClaim与可用的PersistentVolume的进行绑定。在kubernetes的Master中有一个控制回路，它将监控新的PersistentVolumeClaim，并为其查找匹配的PersistentVolume（如果有），并把PersistentVolumeClaim和PersistentVolume绑定在一起。如果一个PersistentVolume曾经动态供给到了一个新的PersistentVolumeClaim，那么这个回路会一直绑定这个PersistentVolume和PersistentVolumeClaim。另外，用户总是能得到它们所要求的存储，但是volume可能超过它们的请求。一旦绑定了，PersistentVolumeClaim绑定就是专属的，无论它们的绑定模式是什么。

如果没有匹配的PersistentVolume，那么PersistentVolumeClaim会无限期的处于未绑定状态，一旦存在匹配的PersistentVolume，PVC绑定此PersistentVolume。比如，就算集群中存在很多的50G的PersistentVolume，需要100G容量的PersistentVolumeClaim也不会匹配满足需求的PersistentVolume。直到集群中有100G的PersistentVolume时，PersistentVolumeClaim才会被绑定。PersistentVolumeClaim基于下面的条件绑定PersistentVolume，如果下面的条件同时存在，则选择符合所有要求的PersistentVolume进行绑定:

1）如果PersistentVolumeClaim指定了存储类，则只会绑定指定了同样存储类的PV；

2）如果PersistentVolumeClaim设置了选择器，则选择器去匹配符合的PersistentVolume；

3）如果没有指定存储类和设置选取器，PersistentVolumeClaim会根据存储空间容量大小和访问模式匹配符合的PersistentVolume。

##### ****使用****

Pod把PersistentVolumeClaim作为卷来使用，Kubernetes集群会通过PersistentVolumeClaim查找绑定的PersistentVolume，并将其挂接至Pod。对于支持多种访问方式的卷，用户在使用 PersistentVolumeClaim 作为卷时，可以指定需要的访问方式。

一旦用户拥有了一个已经绑定的PersistentVolumeClaim，被绑定的PersistentVolume就归该用户所有。用户能够通过在Pod的存储卷中包含的PersistentVolumeClaim，从而访问所占有的PersistentVolume。

##### ****释放****

当用户完成对卷的使用时，就可以利用API删除PersistentVolumeClaim对象了，而且还可以重新申请。删除PersistentVolumeClaim后，对应的持久化存储卷被视为“被释放”，但这时还不能给其他的PersistentVolumeClaim使用。之前的PVC数据还保存在卷中，要根据策略来进行后续处理。

##### ****回收****

PersistentVolume的回收策略向集群阐述了在PersistentVolumeClaim释放卷时，应如何进行后续工作。目前可以采用三种策略：**保留，回收**或者**删除**。保留策略允许重新申请这一资源。在PersistentVolumeClaim能够支持的情况下，删除策略会同时删除卷以及AWS EBS/GCE PD或者Cinder卷中的存储内容。如果插件能够支持，回收策略会执行基础的擦除操作（rm -rf /thevolume/\*），这一卷就能被重新申请了。

###### ****保留****

保留回收策略允许手工回收资源。当PVC被删除，PV将仍然存储，存储卷被认为处于已释放的状态。但是，它对于其他的PVC是不可用的，因为以前的数据仍然保留在数据中。管理员能够通过下面的步骤手工回收存储卷：

1）删除PersistentVolume：在PersistentVolume被删除后，在外部设施中相关的存储资产仍然还在；

2）手工删除遗留在外部存储中的数据；

3）手工删除存储资产，如果需要重用这些存储资产，则需要创建新的PersistentVolume。

###### 回收

此策略将会废弃。循环回收会在存储卷上执行基本擦除命令：rm -rf /thevolume/\*，使数据对于新的PersistentVolumeClaim可用。

###### ****删除****

对于支持删除回收策略的存储卷插件，删除即会从Kubernetes中移除PersistentVolume，也会从相关的外部设施中删除存储资产，例如AWS EBS, GCE PD, Azure Disk或者Cinder存储卷。

#### ****持久化存储卷定义****

在Kubernetes中，PersistentVolume通过各种插件进行实现。持久化存储卷可以通过YAML配置文件进行定义，并指定使用哪个插件类型，下面是一个持久化存储卷的YAML配置文件。在此配置文件中要求提供5Gi的存储空间，存储模式为Filesystem ，访问模式是ReadWriteOnce，通过Recycle回收策略进行持久化存储卷的回收，指定存储类为slow，使用nfs的插件类型。需要注意的是，nfs服务需要提供存在。

**apiVersion:v1**

**kind:PersistentVolume**

**metadata:**

 **name:pv0003**

**spec:**

 **capacity: #容量**

 **storage:5Gi**

 **volumeMode:Filesystem #存储卷模式**

 **accessModes: #访问模式**

**- ReadWriteOnce**

 **persistentVolumeReclaimPolicy:Recycle #持久化卷回收策略**

 **storageClassName:slow #存储类**

 **mountOptions: #挂接选项**

**- hard**

**- nfsvers=4.1**

 **nfs:**

 **path:/tmp**

 **server:172.17.0.2**

##### ****容量（Capacity）****

一般来说，在PersistentVolume中都会指定存储容量。在Kubernetes中通过使用PersistentVolume的capcity属性进行存储容量的设置。

目前，capcity属性仅有storage（存储大小）这一个唯一的资源需要被设置。

##### ****存储卷模式（Volume Mode）****

在kubernetes v1.9之前的版本，存储卷模式的默认值为filesystem，不需要指定。在v1.9版本，用户可以指定volumeMode的值，除了支持文件系统外（file system），也支持块设备（raw block devices）。volumeMode是一个可选的参数，如果不进行设定，则默认为Filesystem。

##### ****访问模式（Access Modes）****

只要资源提供者支持，持久卷能够通过任何方式加载到主机上。每种存储都会有不同的能力，每个PV的访问模式也会被设置成为该卷所支持的特定模式。例如NFS能够支持多个读写客户端，但某个NFS PV可能会在服务器上以只读方式使用。每个PV都有自己的一系列的访问模式，这些访问模式取决于PV的能力。

访问模式的可选范围如下：

* ReadWriteOnce：该卷能够以**读写模式**被加载到一个节点上。
* ReadOnlyMany：该卷能够以只读模式加载到多个节点上。
* ReadWriteMany：该卷能够以读写模式被多个节点同时加载。

在 CLI 下，访问模式缩写为：

* RWO：ReadWriteOnce
* ROX：ReadOnlyMany
* RWX：ReadWriteMany

一个卷不论支持多少种访问模式，同时只能以一种访问模式加载。例如一个 GCEPersistentDisk既能支持ReadWriteOnce，也能支持ReadOnlyMany。

|  |  |  |  |
| --- | --- | --- | --- |
| 存储卷插件 | ReadWriteOnce | ReadOnlyMany | ReadWriteMany |
| AWSElasticBlockStore | ✓ | – | – |
| AzureFile | ✓ | ✓ | ✓ |
| AzureDisk | ✓ | – | – |
| CephFS | ✓ | ✓ | ✓ |
| Cinder | ✓ | – | – |
| FC | ✓ | ✓ | – |
| FlexVolume | ✓ | ✓ | – |
| Flocker | ✓ | – | – |
| GCEPersistentDisk | ✓ | ✓ | – |
| Glusterfs | ✓ | ✓ | ✓ |
| HostPath | ✓ | – | – |
| iSCSI | ✓ | ✓ | – |
| PhotonPersistentDisk | ✓ | – | – |
| Quobyte | ✓ | ✓ | ✓ |
| NFS | ✓ | ✓ | ✓ |
| RBD | ✓ | ✓ | – |
| VsphereVolume | ✓ | – | – (works when pods are collocated) |
| PortworxVolume | ✓ | – | ✓ |
| ScaleIO | ✓ | ✓ | – |
| StorageOS | ✓ | – | – |

##### ****类（Class）****

在PV中可以指定存储类，通过设置storageClassName字段进行设置。如果设置了存储类，则此PV只能被绑定到也指定了此存储类的PVC。在以前的版本中，使用注释volume.beta.kubernetes.io/storage-class字段来指定存储类，而不是storageClassName字段来指定存储类。 此注释仍然可用，但是，在将来的版本中将会被废弃。

##### ****回收策略****

当前的回收策略可选值包括：

* Retain-持久化卷被释放后，需要手工进行回收操作。
* Recycle-基础擦除（“rm-rf /thevolume/\*”）
* Delete-相关的存储资产，例如AWSEBS或GCE PD卷一并删除。

目前，只有NFS和HostPath支持Recycle策略，AWSEBS、GCE PD支持Delete策略。

##### ****挂接选项（Mount Options）****

当持久化卷被挂接至Pod上时，管理员能够指定额外的挂接选项。但不是所有的持久化卷类型都支持挂接选项，下面的存储卷类型支持挂接选项：

* GCEPersistentDisk
* AWSElasticBlockStore
* AzureFile
* AzureDisk
* NFS
* iSCSI
* RBD (Ceph Block Device)
* CephFS
* Cinder (OpenStack block storage)
* Glusterfs
* VsphereVolume
* Quobyte Volumes
* VMware Photon

挂接选项不会进行验证，因此如果如果设置不正确，则会失败。在以前的版本中，使用volume.beta.kubernetes.io/mount-options注释指定挂接选项，而不是使用mountOptions字段。此注释仍然可用，但是在将来的版本中将会被废弃。

### PersistentVolumeClaim（****持久化卷声明）****

下面YAML文件定义了一个名称为myclaim的PersistentVolumeClaim，它的访问模式为ReadWriteOnce，存储卷模式是Filesystem，需要的存储空间大小为8Gi，指定的存储类为slow，并设置了标签选择器和匹配表达式。

**kind: PersistentVolumeClaim**

**apiVersion: v1**

**metadata:**

**name: myclaim**

**spec:**

**accessModes: #访问模式**

**- ReadWriteOnce**

**volumeMode: Filesystem #存储卷模式**

**resources: #资源**

**requests:**

**storage: 8Gi**

**storageClassName: slow #存储类**

**selector: #选择器**

**matchLabels:**

**release: "stable"**

**matchExpressions: #匹配表达式**

**- {key: environment, operator: In, values: [dev]}**

#### ****选择器****

在PersistentVolumeClaim中，可以通过标签选择器来进一步的过滤PersistentVolume。仅仅与选择器匹配的PersistentVolume才会被绑定到PersistentVolumeClaim中。选择器的组成如下：

* matchLabels：只有存在与此处的标签一样的PersistentVolume才会被PersistentVolumeClaim选中；
* matchExpressions ：匹配表达式由键、值和操作符组成，操作符包括In, NotIn, Exists和DoesNotExist，只有符合表达式的PersistentVolume才能被选择。

如果同时设置了matchLabels和matchExpressions，则会进行求与，即只有同时满足上述匹配要求的PersistentVolume才会被选择。

#### ****存储类****

如果PersistentVolumeClaim使用storageClassName字段指定一个存储类，那么只有指定了同样的存储类的PersistentVolume才能被绑定到PersistentVolumeClaim上。

对于PersistentVolumeClaim来说，存储类并不是必须的。依赖于安装方法，可以在安装过程中使用add-on管理器将默认的StorageClass部署至Kubernetes集群中。当PersistentVolumeClaim指定了选择器，并且指定了StorageClass，则在匹配PersistentVolume时，取两者之间的与。即仅仅同时满足存储类和带有要求标签值的PersistentVolume才能被匹配上。

#### PersistentVolumeClaim****作为存储卷****

Pod通过使用PersistentVolumeClaim来访问存储，而PersistentVolumeClaim必须和使用它的Pod在同一个命名空间中。Pod会在同一个命名空间中选择一个合适的PersistentVolumeClaim，并使用PersistentVolumeClaim为其获取存储卷，并将PersistentVolume挂接到主机和Pod上。

**kind:Pod**

**apiVersion:v1**

**metadata:**

**name:mypod**

**spec:**

**containers:**

**- name:myfrontend**

**image:dockerfile/nginx**

**volumeMounts: #挂接存储卷**

**- mountPath:"/var/www/html" #挂接的路径**

**name:mypd #所要挂接的存储卷的名称**

**volumes: #定义存储卷**

**- name:mypd**

**persistentVolumeClaim: #所使用的持久化存储卷声明**

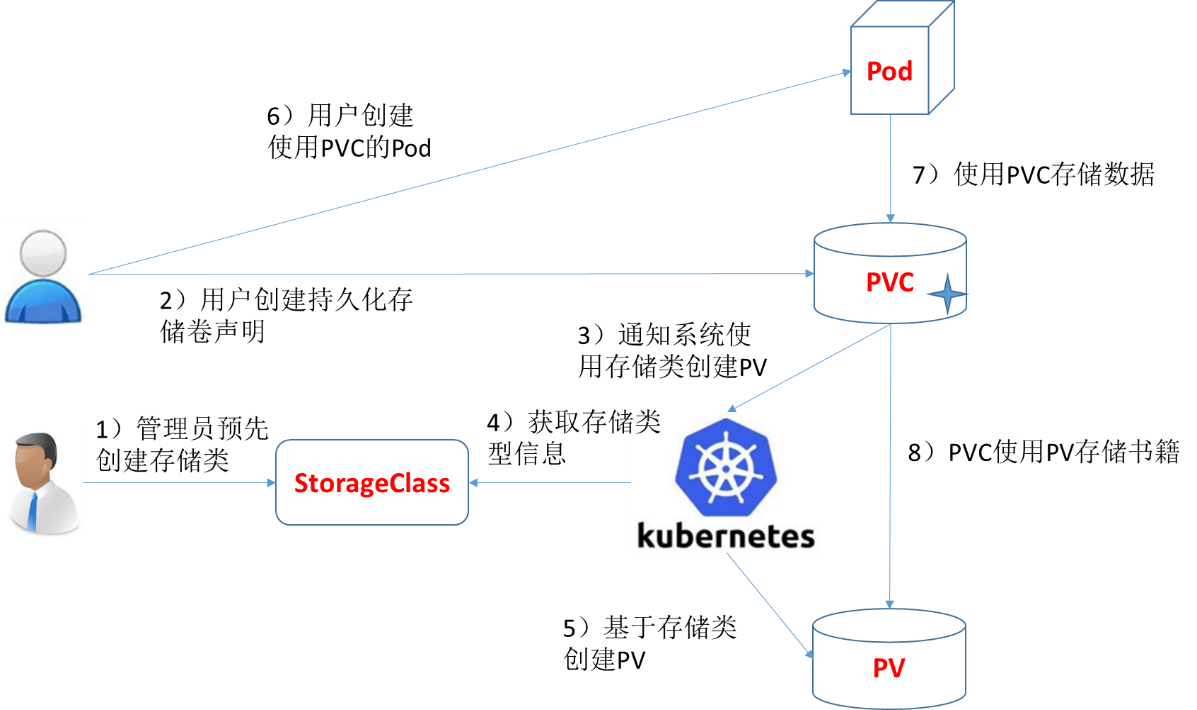
**claimName:myclaim**

### StorageClass（持久化类）

#### ****存储类介绍****

Kubernetes集群管理员通过提供不同的存储类，可以满足用户不同的服务质量级别、备份策略和任意策略要求的存储需求。动态存储卷供应使用StorageClass进行实现，其允许存储卷按需被创建。如果没有动态存储供应，Kubernetes集群的管理员将不得不通过手工的方式类创建新的存储卷。通过动态存储卷，Kubernetes将能够按照用户的需要，自动创建其需要的存储。

基于StorageClass的动态存储供应整体过程如下图所示：



1）集群管理员预先创建存储类（StorageClass）；

2）用户创建使用存储类的持久化存储声明(PVC：PersistentVolumeClaim)；

3）存储持久化声明通知系统，它需要一个持久化存储(PV: PersistentVolume)；

4）系统读取存储类的信息；

5）系统基于存储类的信息，在后台自动创建PVC需要的PV；

6）用户创建一个使用PVC的Pod；

7）Pod中的应用通过PVC进行数据的持久化；

8）而PVC使用PV进行数据的最终持久化处理。

##### ****定义存储类****

每一个存储类都包含provisioner、parameters和reclaimPolicy这三个参数域，当一个属于某个类的PersistentVolume需要被动态提供时，将会使用上述的参数域。

存储类对象的名称非常重要，用户通过名称类请求特定的存储类。管理员创建存储类对象时，会设置类的名称和其它的参数，存储类的对象一旦被创建，将不能被更新。管理员能够为PVC指定一个默认的存储类。

**kind: StorageClass**

**apiVersion: storage.k8s.io/v1**

**metadata:**

**name: standard**

# 指定存储类的供应者

**provisioner: kubernetes.io/aws-ebs**

**parameters:**

**type: gp2**

# 指定回收策略

**reclaimPolicy: Retain**

**mountOptions:**

**- debug**

##### ****供应者****

存储类有一个供应者的参数域，此参数域决定PV使用什么存储卷插件。参数必需进行设置：

|  |  |  |
| --- | --- | --- |
| **存储卷** | **内置供应者** | **配置例子** |
| AWSElasticBlockStore | ✓ | [AWS](https://kubernetes.io/docs/concepts/storage/storage-classes/#aws) |
| AzureFile | ✓ | [Azure File](https://kubernetes.io/docs/concepts/storage/storage-classes/#azure-file) |
| AzureDisk | ✓ | [Azure Disk](https://kubernetes.io/docs/concepts/storage/storage-classes/#azure-disk) |
| CephFS | – | – |
| Cinder | ✓ | [OpenStack Cinder](https://kubernetes.io/docs/concepts/storage/storage-classes/#openstack-cinder) |
| FC | – | – |
| FlexVolume | – | – |
| Flocker | ✓ | – |
| GCEPersistentDisk | ✓ | [GCE](https://kubernetes.io/docs/concepts/storage/storage-classes/#gce) |
| Glusterfs | ✓ | [Glusterfs](https://kubernetes.io/docs/concepts/storage/storage-classes/#glusterfs) |
| iSCSI | – | – |
| PhotonPersistentDisk | ✓ | – |
| Quobyte | ✓ | [Quobyte](https://kubernetes.io/docs/concepts/storage/storage-classes/#quobyte) |
| NFS | – | – |
| RBD | ✓ | [Ceph RBD](https://kubernetes.io/docs/concepts/storage/storage-classes/#ceph-rbd) |
| VsphereVolume | ✓ | [vSphere](https://kubernetes.io/docs/concepts/storage/storage-classes/#vsphere) |
| PortworxVolume | ✓ | [Portworx Volume](https://kubernetes.io/docs/concepts/storage/storage-classes/#portworx-volume) |
| ScaleIO | ✓ | [ScaleIO](https://kubernetes.io/docs/concepts/storage/storage-classes/#scaleio) |
| StorageOS | ✓ | [StorageOS](https://kubernetes.io/docs/concepts/storage/storage-classes/#storageos) |
| Local | – | [Local](https://kubernetes.io/docs/concepts/storage/storage-classes/#local) |

Kubernetes的存储类并不局限于表中的“interneal”供应者，“interneal”供应者的名称带有“kubernetes.io”前缀；也可以允许和指定外部的供应者，外部供应者通过独立的程序进行实现。外部供应者的作者对代码在何处生存、如何供应、如何运行、使用什么卷插件（包括Flex）等有充分的判断权，kubernetes-incubator/external-storage仓库中存在编写外部提供者的类库。例如，NFS不是内部的供应者，但也是可以使用。在kubernetes-incubator/external-storage仓库中以列表的形式展示了一些外部的供应者，一些第三方供应商也提供了他们自己的外部供应者。

##### ****提供者的参数****

存储类存在很多描述存储卷的参数，依赖不同的提供者可能有不同的参数。例如，对于type参数，它的值可能为io1。当一个参数被省略，则使用默认的值。

##### ****回收策略****

通过存储类创建的持久化存储卷通过reclaimPolicy参数来指定，它的值可以是Delete或者Retain，默认为Delete。对于通过手工创建的，并使用存储类进行管理的持久化存储卷，将使用任何在创建时指定的存储卷。

##### ****挂接选项****

通过存储类动态创建的持久化存储卷，会存在一个通过mountOptions参数指定的挂接选择。如果存储卷插件不支持指定的挂接选项，这提供存储供应就会失败，在存储类或者PV中都不会对挂接选项进行验证，因此需要在设置时进行确认。

#### ****使用存储类****

动态存储卷供应基于StorageClass的API对象的来实现，集群管理员能够按需定义StorageClass对象，每一个StorageClass对象能够指定一个存储卷插件（即供应者）。集群管理员能够在一个集群中定义各种存储卷供应，用户不需要了解存储的细节和复杂性，就能够选择符合自己要求的存储。

##### ****启用动态供应****

为了启用动态供应，集群管理员需要预先为用户创建一个或者多个存储类对象。存储类对象定义了使用哪个供应者，以及供应者相关的参数。下面是存储类的一个示例，它创建一个名称为slow的存储类，使用gce供应者：

**apiVersion: storage.k8s.io/v1**

**kind: StorageClass**

**metadata:**

**name: slow**

**provisioner: kubernetes.io/gce-pd**

**parameters:**

**type: pd-standard**

下面创建了一个名为“fast”的存储类，其提供类似固态磁盘的存储卷磁盘：

**apiVersion: storage.k8s.io/v1**

**kind: StorageClass**

**metadata:**

**name: fast**

**provisioner: kubernetes.io/gce-pd**

**parameters:**

**type: pd-ssd**

##### ****使用动态供应****

用户通过在PersistentVolumeClaim中包含一个存储类，来请求动态供应存储。在Kubernetes v1.6之前的版本，通过volume.beta.kubernetes.io/storage-class注释类请求动态供应存储；在v1.6版本之后，用户应该使用PersistentVolumeClaim对象的storageClassName参数来请求动态存储。

下面是请求fast存储类的持久化存储卷声明的YAML配置文件示例：

**apiVersion: v1**

**kind: PersistentVolumeClaim**

**metadata:**

**name: claim1**

**spec:**

**accessModes:**

**- ReadWriteOnce**

# 指定所使用的存储类，此存储类将会自动创建符合要求的PV

**storageClassName: fast**

**resources:**

**requests:**

**storage: 30Gi**

此声明将使用类似于固态存储磁盘，当持久化存储卷声明被删除后，存储卷也将会被销毁。

##### ****默认行为****

如果Kubernetes的集群中没有指定存储类，集群管理员可以通过执行下面的设置，启用默认的存储类：

* 标记一个默认的StorageClass对象；
* 确定API server中DefaultStorage接入控制器已被启用。

管理员能够通过添加storageclass.kubernetes.io/is-default-class注释，标记一个特定的StorageClass作为默认的存储类。在集群中，如果存在一个默认的StorageClass，系统将能够在不指定storageClassName 的情况下创建一个PersistentVolume，DefaultStorageClass接入控制器会自动将storageClassName指向默认的存储类。注意：在一个集群中，最多只能有一个默认的存储类，如果没有默认的存储类，那么如果在PersistentVolumeClaim中没有显示指定storageClassName，则将无法创建PersistentVolume。

#### ****NFS存储类示例****

##### ****部署nfs-provisioner****

为nfs-provisioner实例选择存储状态和数据的存储卷，并将存储卷挂接到容器的/export 命令。

...

 volumeMounts:

    - name: export-volume

      mountPath: /export

volumes:

  - name: export-volume

    hostPath:

      path: /tmp/nfs-provisioner

...

为StorageClass选择一个供应者名称，并在deploy/kubernetes/deployment.yaml进行设置。

args:

  - "-provisioner=example.com/nfs"

...

完整的deployment.yaml文件内容如下：

kind: Service

apiVersion: v1

metadata:

  name: nfs-provisioner

  labels:

    app: nfs-provisioner

spec:

  ports:

    - name: nfs

      port: 2049

    - name: mountd

      port: 20048

    - name: rpcbind

      port: 111

    - name: rpcbind-udp

      port: 111

      protocol: UDP

  selector:

    app: nfs-provisioner

---

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: nfs-provisioner

spec:

  replicas: 1

  strategy:

    type: Recreate

  template:

    metadata:

      labels:

        app: nfs-provisioner

    spec:

      containers:

        - name: nfs-provisioner

          image: quay.io/kubernetes\_incubator/nfs-provisioner:v1.0.8

          ports:

            - name: nfs

              containerPort: 2049

            - name: mountd

              containerPort: 20048

            - name: rpcbind

              containerPort: 111

            - name: rpcbind-udp

              containerPort: 111

              protocol: UDP

          securityContext:

            capabilities:

              add:

                - DAC\_READ\_SEARCH

                - SYS\_RESOURCE

          args:

# 定义提供者的名称，存储类通过此名称指定提供者

            - "-provisioner=nfs-provisioner"

          env:

            - name: POD\_IP

              valueFrom:

                fieldRef:

                  fieldPath: status.podIP

            - name: SERVICE\_NAME

              value: nfs-provisioner

            - name: POD\_NAMESPACE

              valueFrom:

                fieldRef:

                  fieldPath: metadata.namespace

          imagePullPolicy: "IfNotPresent"

          volumeMounts:

            - name: export-volume

              mountPath: /export

      volumes:

        - name: export-volume

          hostPath:

            path: /srv

在设置好deploy/kubernetes/deployment.yaml文件后，通过kubectl create命令在Kubernetes集群中部署nfs-provisioner。

$ kubectl create -f {path}/deployment.yaml

##### ****创建StorageClass****

下面是example-nfs的StorageClass配置文件，此配置文件定义了一个名称为nfs-storageclass的存储类，此存储类的提供者为nfs-provisioner。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: nfs-storageclass

provisioner: nfs-provisioner

通过kubectl create -f命令使用上面的配置文件创建：

$ kubectl create -f deploy/kubernetes/class.yaml

storageclass “example-nfs” created

在存储类被正确创建后，就可以创建PersistenetVolumeClaim来请求StorageClass，而StorageClass将会为PersistenetVolumeClaim自动创建一个可用PersistentVolume。

##### 创建PersistenetVolumeClaim

PersistenetVolumeClaim是对PersistenetVolume的声明，即PersistenetVolume为存储的提供者，而PersistenetVolumeClaim为存储的消费者。下面是PersistentVolumeClaim的YAML配置文件，此配置文件通过*spec.storageClassName*字段指定所使用的存储储类。

在此配置文件中，使用nfs-storageclass存储类为PersistenetVolumeClaim创建PersistenetVolume，所要求的PersistenetVolume存储空间大小为1Mi，可以被多个容器进行读取和写入操作。

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-pvc

spec:

accessModes:

- ReadWriteMany

**storageClassName： nfs-storageclass**

resources:

requests:

storage: 1Mi

通过kubectl create命令创建上述的持久化存储卷声明：

$ kubectl create -f {path}/claim.yaml

##### ****创建使用PersistenVolumeClaim的部署****

在这里定义名为busybox-deployment的部署YAML配置文件，使用的镜像为busybox。基于busybox镜像的容器需要对**/mnt**目录下的数据进行持久化，在YAML文件指定使用名称为nfs的PersistenVolumeClaim对容器的数据进行持久化。

# This mounts the nfs volume claim into /mnt and continuously

# overwrites /mnt/index.html with the time and hostname of the pod.

apiVersion: v1

kind: Deployment

metadata:

name: busybox-deployment

spec:

replicas: 2

selector:

name: busybox-deployment

template:

metadata:

labels:

name: busybox-deployment

spec:

containers:

- image: busybox

command:

- sh

- -c

- 'while true; do date > /mnt/index.html; hostname >> /mnt/index.html; sleep $(($RANDOM % 5 + 5)); done'

imagePullPolicy: IfNotPresent

name: busybox

volumeMounts:

# name must match the volume name below

- name: nfs

mountPath: "/mnt"

#

volumes:

- name: nfs

persistentVolumeClaim:

claimName: nfs-pvc

通过kubectl create创建busy-deployment部署：

$ kubectl create -f {path}/nfs-busybox-deployment.yaml

### Secret（保密文件）

### ConfigMap（配置 2019-04-09 Y）

#### ConfigMap概述

在生产环境中，应用程序的配置可能会很复杂，因此会需要多个配置文件、命令行参数和环境变量的组合。在使用容器进行部署时，应该把配置从应用程序的镜像中解耦出来，从而能够有效的保证镜像的可移植性。在1.2版本之后，Kubernetes引入了ConfigMap来处理这种类型的配置数据。

ConfigMap被设计用来存储通用的配置变量，它类似于配置文件，实现将分布式系统中的环境变量集中到一个地方进行管理。从数据角度来看，ConfigMap保存的只是键值对组，用于存储被Pod或者其他资源对象（如Deployment、StatefulSet等）使用和访问的数据信息。这与Secret的设计理念类似，主要区别在于secret用于存储敏感信息，ConfigMap只用于存储简单的文本信息。

创建Pod时，对Configmap进行绑定，Pod内的应用就可以直接引用ConfigMap的配置内容。ConfigMap即可以用来保存单个属性，也可以用来保存整个配置文件或者JSON格式的二进制大对象。

ConfigMap API以键值对的方式存储配置数据，ConfigMap的数据可以被Pod和Deployment、StatefulSet等控制器的系统组件所使用。ConfigMap和Secret的作用相似，但ConfigMap用于存储不包含敏感信息的数据。在ConfigMap中，通过data域来配置数据。

如下面例子所示，ConfigMap即可以包含细粒度的配置项，如：example.property.1；也可以包含粗粒度的配置文件，如：example.property.file。

kind: ConfigMap

apiVersion: v1

metadata:

  name: example-configmap

  namespace: default

data:

  # 通过具体值创建的例子

  example.property.1: hello

  example.property.2: world

  # 通过文件创建的例子

  example.property.file: |-

    property.1=value-1

    property.2=value-2

使用ConfigMap时，需要注意下面的事项：

* 在Pod 规格中应用ConfigMap之前，ConfigMap必须已经存在。如果Pod引用的ConfigMap不存在，Pod将不能正常启动。
* ConfigMap只能被在同一个命名空间中的Pod所引用。

#### 创建ConfigMap

在Kubernetes中，可以使用kubectl create configmap命令，通过目录、文件和指定值(literal value)来创建ConfigMap：

$ kubectl create configmap <map-name> <data-source>

这里的 <map-name> 是希望创建的ConfigMap的名称，<data-source>是一个目录、文件和具体值。

在ConfigMap中，键值对的数据源如下所述：

* key = 文件名或者在命令行中提供的键
* value = 文件内容或者在命令行中提供的具体值

能够使用kubectl describe或者kubectl get获取ConfigMap的信息。

##### 通过目录创建ConfigMaps

在Kubernetes中，创建ConfigMap的第一种方式是可以通过目录来创建。例如，D:\configmap\目录下存在game.properties和ui.properties文件。

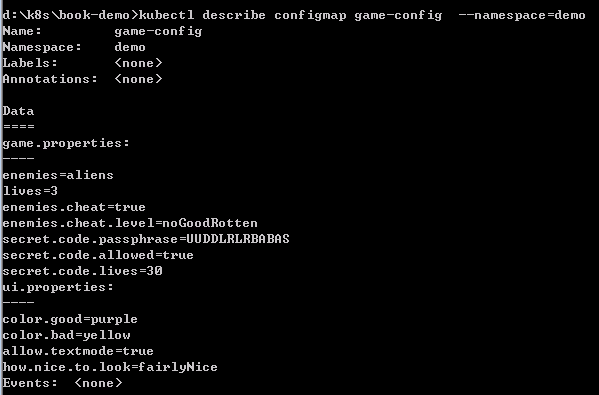
1）通过在一个目录下的多个文件创建ConfigMap,例如：

$kubectl create configmap game-config \

--from-file=d:\configmap --namespace=demo

2）查看game-config ConfigMap的信息:

$ kubectl describe configmaps game-config --namespace=demo



##### 通过文件创建ConfigMaps

创建ConfigMap的第二种方式是可以通过文件来创建，可以通过单个或者多个文件来创建ConfigMap。

###### 通过单个文件创建

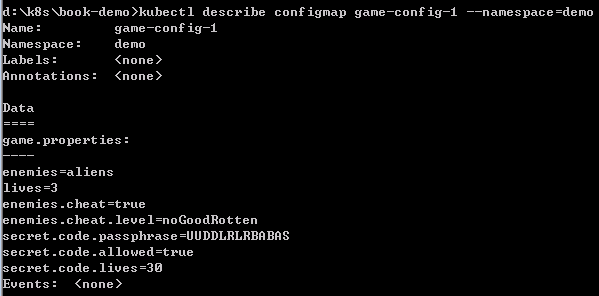
1）通过单个文件创建ConfigMap，例如：

$ kubectl create configmap game-config-1 \

--from-file=d:\configmap\game.properties --namespace=demo

2）查看生产的ConfigMap信息：

$ kubectl describe configmaps game-config-1 --namespace=demo



###### 通过多个文件生成

1）通过多个文件创建ConfigMap

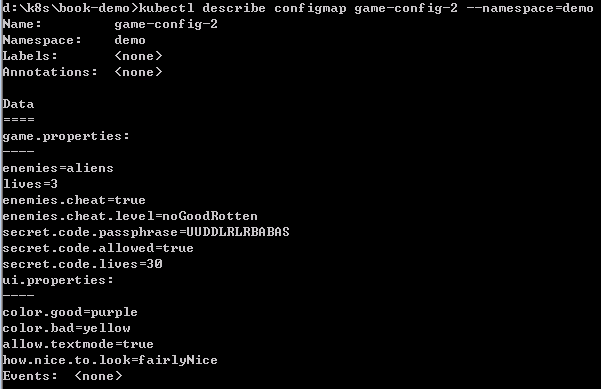
$ kubectl create configmap game-config-2 \

--from-file=/docs/tasks/configure-pod-container/configmap/kubectl/game.properties \

--from-file=/docs/tasks/configure-pod-container/configmap/kubectl/ui.properties --namespace=demo

2）查看生产的ConfigMap信息：

$ kubectl describe configmaps game-config-2 --namespace=demo



###### 通过文件创建并定义键

通过文件创建ConfigMap时，键默认为文件的名称，文件的内容为键值对的值。此方式允许用户设置自己的键，而不是默认使用文件的名称作为键。

$ kubectl create configmap game-config-3 --from-file=<my-key-name>=<path-to-file>

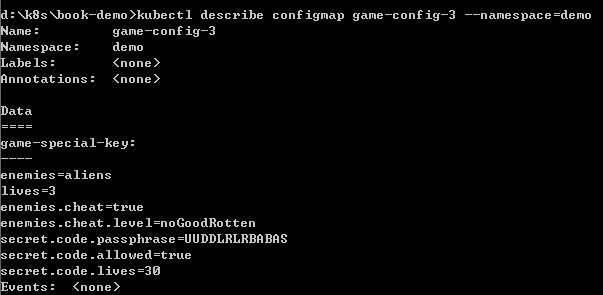
这里的<my-key-name>是在ConfigMap里想要被使用的键，<path-to-file>是文件的路径，例如：

$ kubectl create configmap game-config-3 \

--from-file=game-special-key=d:/configmap/game.properties -–namespace=demo

2）查看生产的ConfigMap信息：

$ kubectl describe configmaps game-config-3 --namespace=demo



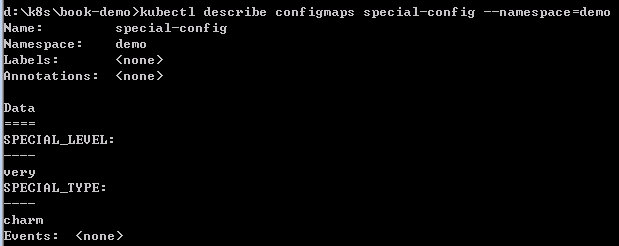
##### 通过具体值创建ConfigMaps

通过kubectl create configmap命令，能够使用–from-literal参数来定义具体值来创建ConfigMap：

$ kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm --namespace=demo

在命令行中可以输入多个键-值对。每一一个键值对会成为ConfigMap data部分的一条记录。

$ kubectl describe configmaps special-config --namespace=demo



#### 使用ConfigMap

在集群中创建好ConfigMap后，就可以由Pod等资源对ConfigMap中的信息进行使用。在Kubernetes中，使用ConfigMap的两种典型场景是：使用ConfigMap数据定义Pod环境变量和添加ConfigMap数据值存储卷。

##### 使用ConfigMap数据定义Pod环境变量

用户可以通过ConfigMap数据来定义Pod的环境变量，这里有四种应用场景：Pod环境变量的值来自于一个或者多个ConfigMap文件，使用ConfigMap中所有的键值对做Pod的环境变量，在Pod命令中使用ConfigMap定义的环境变量。

###### Pod环境变量的值来自于单一ConfigMap

此场景是通过单一的ConfigMap文件为容器中的应用设置环境变量，这是最简单的应用模式。

1）在ConfigMap中定义一个环境变量作为键值对：

$ kubectl create configmap special-config --from-literal=special.how=very

2）指派ConfigMap中定义的special.how的值给Pod中SPECIAL\_LEVEL\_KEY环境变量：

$ kubectl edit pod configmap-test-pod

apiVersion:v1

kind:Pod

metadata:

name:dapi-test-pod

spec:

containers:

- name:test-container

image:k8s.gcr.io/busybox

command:["/bin/sh","-c","env"]

env:

# 定义环境变量，环境变量名为SPECIAL\_LEVEL\_KEY

- name:SPECIAL\_LEVEL\_KEY

**valueFrom:**

**configMapKeyRef:**

**# ConfigMap的名称为special-config**

**name:special-config**

**# 指定所要从ConfigMap获取值的键**

**key:special.how**

restartPolicy:Never

保存Pod规格的变化，Pod将输出SPECIAL\_LEVEL\_KEY=very。

###### Pod环境变量的值来自于多个ConfigMap

此场景是通过两个的ConfigMap文件为容器中的应用设置环境变量，首先通过YAML文件创建两个ConfigMap，其中一个名称为special-config.yaml，另一个名称为env-config。

下面是special-config的yaml文件内容：

apiVersion:v1

kind:ConfigMap

metadata:

   name:special-config

   namespace:default

data:

   special.how:very

下面是env-config的yaml文件内容：

apiVersion:v1

kind:ConfigMap

metadata:

   name:env-config

   namespace:default

data:

   log\_level:INFO

在Pod规格中定义环境变量：

apiVersion:v1

kind:Pod

metadata:

  name:dapi-test-pod

spec:

  containers:

 - name:test-container

    image:k8s.gcr.io/busybox

    command:["/bin/sh","-c","env"]

    env:

    # 定义SPECIAL\_LEVEL\_KEY环境变量

- name:SPECIAL\_LEVEL\_KEY

      valueFrom:

        configMapKeyRef:

           name:special-config

           key:special.how

    # 定义LOG\_LEVEL环境变量

- name:LOG\_LEVEL

       valueFrom:

          configMapKeyRef:

              name:env-config

              key:log\_level

  restartPolicy:Never

保存变更后的Pod，Pod将会输出SPECIAL\_LEVEL\_KEY=very和LOG\_LEVEL=info。

###### 使用ConfigMap中所有键值对作为Pod的环境变量

此场景是直接使用ConfigMap中的所有键值对作为容器中应用的环境变量，此能力在Kubernetes v1.6+可用。

首先，创建包含多个键-值对的ConfigMap。

apiVersion:v1

kind:ConfigMap

metadata:

  name:special-config

  namespace:default

data:

  SPECIAL\_LEVEL:very

  SPECIAL\_TYPE:charm

使用envFrom定义所有的ConfigMap数据作为Pod的环境变量，来自于ConfigMap的键成为Pod中环境变量的名。

apiVersion:v1

kind:Pod

metadata:

  name:dapi-test-pod

spec:

  containers:

  - name:test-container

    image:k8s.gcr.io/busybox

    command:["/bin/sh","-c","env"]

      # 定义环境，环境变量直接来自于ConfigMap

envFrom:

      - configMapRef:

        name:special-config

  restartPolicy:Never

保存Pod规范的变更，现在Pod的输出包括：

SPECIAL\_LEVEL=very

SPECIAL\_TYPE=charm

###### 在Pod命令行中使用ConfigMap定义的环境变量

在Pod规范的command 中使用$(VAR\_NAME) ，可以获取ConfigMap定义的环境变量。例如：下面的Pod规范：

apiVersion:v1

kind:Pod

metadata:

  name:dapi-test-pod

spec:

  containers:

 - name:test-container

   image:k8s.gcr.io/busybox

    command:["/bin/sh","-c","echo $(SPECIAL\_LEVEL\_KEY) $(SPECIAL\_TYPE\_KEY)"]

    env:

    - name:SPECIAL\_LEVEL\_KEY

      valueFrom:

         configMapKeyRef:

            name:special-config

            key:SPECIAL\_LEVEL

    - name:SPECIAL\_TYPE\_KEY

      valueFrom:

          configMapKeyRef:

           name:special-config

            key:SPECIAL\_TYPE

  restartPolicy:Never

test-container容器的输入如下：

very charm

##### 添加ConfigMap数据至存储卷

ConfigMap的另外一个应用场景是将文件中的数据添加到容器中，下面是名称为volume-config的配置文件。此文件有两个键值对，即special.how: very和special.type: charm。

apiVersion: v1

kind: ConfigMap

metadata:

name: volume-config

namespace: default

data:

special.how: very

special.type: charm

通过下面的命令，在集群中创建此配置文件：

$ kubectl create -f volume-config.yaml --namespace=demo

此场景用于将ConfigMap中的数据添加到容器中，通过在Pod的存储卷区域指定ConfigMap的名称，这将会添加ConfigMap数据到volumeMounts.mountPath指定的目录下（在此例子为/etc）。

apiVersion: apps/v1beta2

kind: Deployment

metadata:

name: tomcat-deployment

labels:

app: tomcat

spec:

replicas: 1

selector:

matchLabels:

app: tomcat

template:

metadata:

labels:

app: tomcat

spec:

containers:

- name: tomcat

image: tomcat

ports:

- containerPort: 8080

volumeMounts:

- name: config-volume

mountPath: /mnt

volumes:

- name: config-volume

configMap:

# 提供ConfigMap名称，此ConfigMap包含添加至容器的文件名称

name: volume-config

当Pod运行时，进入容器，并定位到/mnt目录，能够看到有两个文件：

special.level

special.type

这两个文件时volume-config配置文件中两个键，即为每一个键创建一个文件。而键的值为对于键的内容：

在/mnt目录下，执行下面的命令，将会返回very：

$ mnt special.how



注意：如果在/etc/config/目录下已存在文件，将不会删除。

# Kubernetes安全 Y

本章节从身份认证、访问授权、性能监控和日志管理这个几个方面讲述Kubernetes的安全。身份认证用于保证只有合法的用户才能够访问Kubernetes，并与Kubernetes进行交互；访问控制用于保证只有经过合适授权的用户才能够访问特定的资源，并对资源进行授权的操作；日志管理用于记录用户和系统的行为，帮着管理员跟踪用户在系统中的行为，以便于后续的行为审计和问题处理。

## 身份认证(2019-04-12 )

身份认证用于对访问Kubernetes的用户合法性进行检查，只有合法的用户才能够与访问Kubernetes。在Kubernetes中提供了客户端证书、令牌和静态密码等多种认证策略，用户可以根据自身的需要使用某一或多种认证策略。

### ****Kubernetes中的用户****

由于处于安全考虑，所有的系统都存在访问和使用其的用户，Kubernetes也不例外。在Kubernetes的集群中有存在两类用户：

* service accounts：由Kubernetes进行管理的特殊用户；
* 普通用户：普通用户是由外部应用进行管理的用户。

对于普通用户，Kubernetes管理员只负责为其分配私钥。普通用户可能来自于Keystone或google等外部系统中，或者甚至是存储在文件中的用户名和密码列表。在Kubernetes中，没有表达普通用户的对象，因此，也就不能通过API将普通用户添加到集群中。

而Service Account是由Kubernetes API管理的用户，它们被绑定到特定的命名空间中，并由API服务器自动创建或通过API调用手动创建。Service Account与存储在Secrets的一组证书相关联，这些凭据被挂载到Pod中，以允许集群中的进程与Kubernetes API进行通信。

API请求要么来自于普通用户或Service Account，或来自于匿名请求。这就意味着集群内外部的所有进程（从来自于用户使用kubectl输入的请求，或来自于Nodes中kubelet的请求，或来自控制板的成员的请求）都需要进行认证才能与API server进行交互。

### ****认证策略（Authentication strategies）****

Kubernetes的用户可以使用客户端证书、Bearer Token、身份验证代理或HTTP基本认证，通过身份验证插件来验证API请求。比如，当HTTP请求到达API server，插件尝试将以下的属性与请求进行关联：

* Username：用户名，标识最终用户的字符串。通常，Username的值可能像“kube-admin”或者“jane@example.com”。
* UID：用户的唯一标识标识。
* Groups：用户组组名。
* Extra fields：记录用户其他信息的属性。

上述所有值对于认证系统都是不透明的，只有在被授权者解释时才有意义。可以同时启用上面的多个认证方法。通常至少使用两种：

* service accounts使用serive account tokens。
* 用户认证至少使用另外一种方法。

当同时启用多个认证模块时，根据短路径评估，使用第一个认证模块成功地认证了请求。API server不保证接下来的认证是通过的。system:authenticated 组被包括在所有已认证用户的组列表中。可以通过使用authenticating.proxy或者authenticaiton webhook与其他的认证协议进行集成（LDAP, SAML, Kerberos, alternate x509 schemes, etc）

### ****X509客户端证书****

客户端证书身份认证模式通过在API Server中设置–client-ca-file = SOMEFILE选项来启用。所引用的文件中必须包含一个或多个证书管理机构，用以验证提交给API服务器的客户端证书。如果客户端提交的证书通过验证，主体的通用名称将被用作请求的用户名。从Kubernetes的1.4版本起，客户端证书也可以通过证书的组织（organization）区域指定用户的组成员资格。客户端证书认证叫作TLS双向认证，也就是服务器客户端互相验证证书的正确性，在都正确的情况下协调通信加密方案。CA\_CERTIFICATE\_FILE肯定包括一个或者多个认证中心，可以被用来验证呈现给api－server的客户端证书。客户端证书的／CN将作为用户名。

例如，使用openssl命令行工具生成证书签名请求：

$ openssl req -new -key jbeda.pem -out jbeda-csr.pem -subj ”/CN=jbeda/O=app1/O=app2”

这将会创建一个用户名为“jbeda”，所属组为”app1”和“app2”的签名请求。 使用客户端证书身份验证时，可以通过easyrsa、OpenSSL或cfssl手动生成证书，x509证书一般会用到三类文件，key(私用密钥)，csr(证书请求文件，用于申请证书)，crt(CA认证后的证书文件)。

数字证书则是由证书认证机构（CA）对证书申请者真实身份验证之后，用CA的根证书对申请人的一些基本信息以及申请人的公钥进行签名（相当于加盖发证书机构的公章）后形成的一个数字文件。数字证书包含证书中所标识的实体的公钥（就是说你的证书里有你的公钥），由于证书将公钥与特定的个人匹配，并且该证书的真实性由颁发机构保证（就是说可以让大家相信你的证书是真的），因此，数字证书为如何找到用户的公钥并知道它是否有效这一问题提供了解决方案。

此处以Openssl为例进行阐述，通过openssl手动创建。首先，生成一个CA根证书；然后，用CA根证书来签发用户证书。  用户的证书过程是：先生成一个私钥；然后，用私钥生成证书请求(证书请求里应含有公钥信息)；最后，利用CA根证书来签发用户证书。通过Openssl生成证书的过程如下：

#### ****生成CA根证书****

通过下述两个步骤，以生产CA根证书，：

1）用2048位生成一个ca私钥：

**$ openssl genrsa** -out **ca**.**key** 2048

2）根据ca私钥生成证书ca根证书（用天设置证书的有效时间）：

**$ openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER\_IP}" -days 10000 -out ca.crt**

#### ****生成服务器端用户证书****

在生成CA根证书后，通过下面的步骤生成服务器端用户证书：

1）使用2048位生成服务器端用户私钥：

$ openssl genrsa -out server.key 2048

2）为证书签名请求(CSR)创建一个配置文件（例如：csr.conf）。在将真实的值保持到文件之前，一定要用要尖括号把这些值标起来（例如< master\_ip >）。注意，**MASTER\_CLUSTER\_IP**的值是服务集群IP服务器如前所描述的服务群集IP。

[ req ]default\_bits = 2048 prompt = no default\_md = sha256 req\_extensions = req\_ext distinguished\_name = dn

[ dn ]C = <country> ST = <state> L = <city> O = <organization> OU = <organization unit> CN = <MASTER\_IP>

[ req\_ext ]subjectAltName = @alt\_names

[ alt\_names ]DNS.1 = kubernetes DNS.2 = kubernetes.default DNS.3 = kubernetes.default.svc

DNS.4 = kubernetes.default.svc.cluster DNS.5 = kubernetes.default.svc.cluster.local IP.1 = <MASTER\_IP>

IP.2 = <MASTER\_CLUSTER\_IP>

[ v3\_ext ]authorityKeyIdentifier=keyid,issuer:alwaysbasicConstraints=CA:FALSEkeyUsage=keyEncipherment,

dataEnciphermentextendedKeyUsage=serverAuth,clientAuthsubjectAltName=@alt\_names

3）基于配置文件和私钥生成证书签名请求：

$ openssl req -new -key server.key -out server.csr -config csr.conf

4）使用ca根证书，ca私钥和服务器端用户证书请求创建服务器用户证书：

$ openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \-CAcreateserial -out server.crt -days 10000 \-extensions v3\_ext -extfile csr.conf

5）查看服务器端用户证书：

$ openssl x509 -noout -text -in ./server.crt

最后，将相同的参数添加到API Server的启动参数。

6）生成pem格式证书：

有时可能需要用到pem格式的证书，可以用以下方式合并证书文件（crt）和私钥文件（key）来生成

$cat server.crt server.key > server.pem

### ****Service Account令牌****

在有些情况下，希望在 Pod 内部访问 API server，以获取集群的信息，以及对集群进行改动。针对这种情况，kubernetes 提供了一种特殊的认证方式：Service Account令牌。 Service Account 是限定命名空间的。在创建命名空间的时候，kubernetes 会为每一个命名空间创建一个默认的 Service Account；这个默认的 Service Account 只能访问该命名空间内的资源。Service Account 和 Pod、Service、Deployment 一样是 Kubernetes 集群中的一种资源，用户也可以通过手动的方式创建Service Account。

Service Account是一个自动启用的认证器，它使用被签名的Bearer Token对请求进行认证，该插件接受两个可选参数：

* + --service-account-key-file： 一个包含用于对Bearer Token进行签名的PEM编码密钥文件。如果不指定，将使用API服务器的TLS私钥。
  + --service-account-lookup ：如果启用，从API中删除的tokens将会被废除。

Service Account通常由API服务器自动创建，并通过ServiceAccount [Admission Controller](https://kubernetes.io/docs/admin/admission-controllers/)与集群中的Pods进行关联。 Bearer tokens被挂载到pod中众所周知的位置，从使集群中的进程可以与API服务器进行通信。Service Account可以使用PodSpec的serviceAccountName字段来关联到Pod中。

注意：通常省略serviceAccountName字段，因为一般是自动完成的。

apiVersion: apps/v1 # this apiVersion is relevant as of Kubernetes 1.9

kind: Deployment

metadata:

  name: nginx-deployment

  namespace: default

spec:

  replicas: 3

  template:

    metadata:

    # ...

    spec:

      serviceAccountName: bob-the-bot

      containers:

      - name: nginx

        image: nginx:1.7.9

Service Account的bearer tokens也在集群外部使用，可以用于创建希望与API进行通信的身份。通过使用kubectl create serviceaccount（NAME）命令，可以创建Service Account，并会创建一个关联的密钥。

$ kubectl create serviceaccount jenkins

serviceaccount "jenkins" created

$ kubectl get serviceaccounts jenkins -o yaml

创建的证书保存了API server公共CA和签名的JSON Web Token（JWT）。

$ kubectl get secret jenkins-token-mk89c -o yaml

apiVersion: v1

data:

  # ca证书

ca.crt: (APISERVER'S CA BASE64 ENCODED)

  # 命名空间

namespace: ZGVmYXVsdA==

# 令牌

  token: (BEARER TOKEN BASE64 ENCODED)

kind: Secret

metadata:

  # ...

type: kubernetes.io/service-account-token

注意：保密字典的值是基于base64编码的，因为通常秘钥都是使用base64进行编码的。

签名的JWT可以作为bearer token来验证给定的service account。通常，这些秘钥被挂载到Pods中，用于集群中访问API服务器，但也可以在群集外部使用。

Service Account使用用户名进行验证 system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)，并分配给组 [system:serviceaccounts](http://systemserviceaccounts/) 和 system:serviceaccounts:(NAMESPACE)。由于service account 令牌存储在秘钥中，任何具有对这些秘钥的读取访问权限的用户都可以作为service account 进行身份验证。

ServiceAccount 主要包含了三个内容：**命名空间、令牌** **和 CA**。命名空间指定了 Pod 所在的 命名空间；CA是用于验证 api server 的证书；令牌用作身份验证。它们都通过挂接的方式保存在 pod 的文件系统中，其中令牌保存的路径是: /var/run/secrets/[kubernetes.io/serviceaccount/token](http://kubernetes.io/serviceaccount/token) ，这是 apiserver 使用base64 编码通过私钥签的令牌； CA 保存的路径是 /var/run/secrets/[kubernetes.io/serviceaccount/ca.crt](http://kubernetes.io/serviceaccount/ca.crt) ，命名空间保存的路径是 /var/run/secrets/[kubernetes.io/serviceaccount/namespace](http://kubernetes.io/serviceaccount/namespace) ，也是使用 base64 编码。

如果令牌能够通过认证，那么请求的用户名将被设置为 system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT) ，而请求的组名有两个：[system:serviceaccounts](http://systemserviceaccounts/) 和 system:serviceaccounts:(NAMESPACE)。

### ****静态Tokent文件****

API server通过-token-auth-file=SOMEFILE选择读取不记名的Token。当前，token是无期限持续的，除非重启API server。Token文件是一个至少包含3列的csv文件： token, user name, user uid，后跟可选的组名。注意，如果您有多个组，则列必须是双引号，例如：

token,user,uid,"group1,group2,group3"

用token唯一标识请求者，只要apiserver存在该token，则认为认证通过，但是如果需要新增Token，则需要重启kube-apiserver组件，实际效果不是很好。

当通过客户端使用 bearer token 认证时，API服务器需要一个值为带有Bearer THETOKEN值的Authorization头。bearer token必须是一个字符序列，能够放在HTTP请求头中。例如：如果bearer token是31ada4fd-adec-460c-809a-9e56ceb75269，它将会在HTTP头中按下面的方式呈现：

Authorization: Bearer 31ada4fd-adec-460c-809a-9e56ceb75269

### ****Bootstrap令牌****

此特性目前还是alpha阶段。为了能够在新的集群中使用bootstrapping认证，Kubernetes需要包括一个动态管理的Bearer token（成为Bootstrap Token），这种token以Secrets的方式存储在kube-system命名空间中，在这个命名空间token可以被动态的管理和创建。Controller Manager有一个TokenCleaner控制器，通过此控制器删除过期了的bootstrap token。

token证书格式为：[a-z0-9]{6}.[a-z0-9]{16}，Token的第一部分是一个Token ID，第二部分是token的秘钥。你需要在http协议头中加上类似的信息：

Authorization: Bearer 781292.db7bc3a58fc5f07e

如果要使用Bootstrap，需要在API Sever中设置–experimental-bootstrap-token-auth。同时，必须在Controller Manager中设置–controllers=\*,tokencleaner来启用TokenCleaner控制器。

在使用kubeadm部署Kubernetes时，kubeadm会自动创建默认token，可通过kubeadm token list命令查询。

### ****静态密码文件****

基础认证模式通过在API Server中设置–basic-auth-file=SOMEFILE来启用。一旦API server服务启动，加载的用户名和密码信息就不会发生改变，任何对源文件的修改必须重启 apiserver 才能生效。

静态密码文件是 CSV 格式的文件，每行对应一个用户的信息，前面3列为：密码、用户名、用户 ID，这些是是必须的；第四列是可选的组名（如果有多个组，必须用双引号）：

password,user,uid,"group1,group2,group3"

当Http客户端使用基础认证时，API Server需要一个带有Basic BASE64ENCODED(USER:PASSWORD) 值的 Authorization 头。API Server 解析出客户端提供的用户名和密码，如果和文件中的某一行匹配，就认为认证成功。

### ****Keystone密码****

Kubernetes也可以使用Openstack的Keystone组件进行身份认证和授权，这个方法对于已经使用 openstack 来搭建 IaaS 平台的公司比较适用，直接使用 keystone 可以保证 Iaas 和 Caas 平台保持一致的用户体系。

需要API Server在启动时指定–experimental-keystone-url=<AuthURL>，而https时还需要设置–experimental-keystone-ca-file=SOMEFILE。

### ****匿名请求****

如果用户请求没有Kubernetes任何方式的身份认证，在正常情况下，Kubernetes会直接返回 “401” 错误信息。但是，在 kubernetes 还提供另外一种方案，即给没有通过认证的请求一个特殊的用户名 ：[system:anonymous](http://systemanonymous/) 和组名： [system:unauthenticated](http://systemunauthenticated/) 。这样的话，就可以跟授权模式结合起来，为匿名请求设置一些特殊的权限，比如，只能读取当前 namespace 的 pod 信息，以方便用户访问。

在Kubernetes的1.5.1-1.5.x版本，在默认情况下匿名请求是不可用的，但能够通过在API Server中设置–anonymous-auth=true来启用。

在Kubernetes的1.6+版本，如果使用AlwaysAllow以外的收取模式，则匿名请求默认开启，但可用通过设置–anonymous-auth=false来禁止匿名请求。从1.6版本开始，ABAC和RBAC授权需要显示对[system:anonymous](http://systemanonymous/)用户和[system:unauthenticated](http://systemunauthenticated/)的组进行授权，通过\* user或\*group这种方式进行访问授权将不包含匿名的用户。

## 访问授权(2019-04-15 Y)

对于具有合法访问Kubernetes的用户来说，并不意味着就能够对Kubernetes中的资源进行访问和操作。只有经过授权的用户，才能够对特定的资源进行访问和操作。Kubernetes的访问授权有多种模式，在本文中主要讲述基于RBAC的授权模式。

### ****RBAC介绍****

在Kubernetes中，授权有ABAC（基于属性的访问控制）、RBAC（基于角色的访问控制）、Webhook、Node、AlwaysDeny（一直拒绝）和AlwaysAllow（一直允许）这6种模式。从1.6版本起，Kubernetes 默认启用RBAC访问控制策略。Kubernetes通过设置–authorization-mode=RBAC，从而启用RABC访问控制策略。在RABC API中，通过如下的步骤进行授权：

1）定义角色：在定义角色时会指定此角色对于资源的访问控制的规则；

2）绑定角色：将主体与角色进行绑定，对用户进行访问授权。

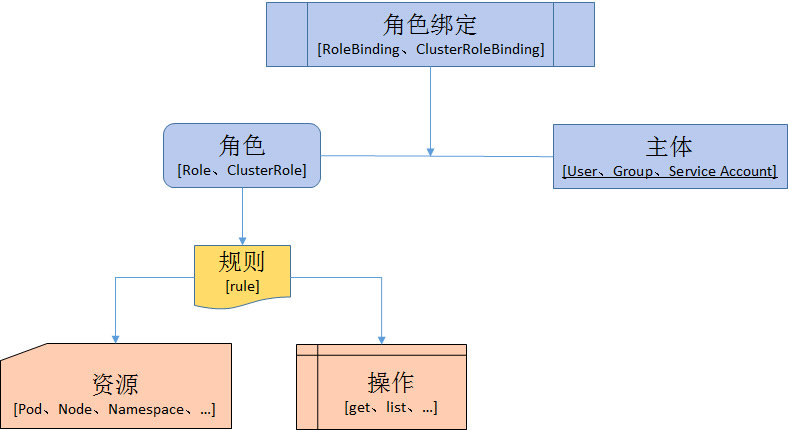


图6-1 基于RBAC的授权模式

#### ****角色和集群角色****

在RBAC API中，角色包含代表权限集合的规则。在Kubernetes中，权限只能被授予，而不能被拒绝。在Kubernetes中有两类角色，即普通角色(Role)和集群角色(ClusterRole)。可以通过Role定义一个命名空间中的角色，或者可以使用ClusterRole定义集群范围的角色。一个角色只能被用来授予访问单一命令空间中的资源。下面是在“default”命令空间中定义了一个名为“pod-reader”的角色，此角色能够访问在“default”命名空间中的Pod：

**kind: Role**

**apiVersion: rbac.authorization.k8s.io/v1**

**metadata:**

**namespace: default**

**name: pod-reader**

**rules:**

**- apiGroups: [""] # "" indicates the core API group**

**resources: ["pods"]**

**verbs: ["get", "watch", "list"]**

集群角色(ClusterRole)能够被授予如下资源的权限：

* 集群范围的资源（类似于Node）
* 非资源端点（类似于”/healthz”）
* 集群中所有命名空间的资源（类似Pod）

下面是授予集群角色读取秘密字典文件访问权限的例子：

**kind:ClusterRole**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**# "namespace" omitted since ClusterRoles are not namespaced**

**name:secret-reader**

**rules:**

**- apiGroups:[""]**

**resources:["secrets"] #明确资源类型**

**verbs:["get","watch","list"]**

#### ****角色绑定和集群角色绑定****

角色绑定用于将角色与一个或一组用户进行绑定，从而实现将对用户进行授权的目的。主体分为用户、组和服务帐户。角色绑定也分为角色普通角色绑定和集群角色绑定。角色绑定只能引用同一个命名空间下的角色。在下面的例子中，在”default”命名空间中角色绑定将‘jane’用户和“pod-reader”角色进行了绑定，这就授予了“jane”能够访问“default”命名空间下的Pod。

**# This role binding allows "jane" to read pods in the "default" namespace.**

**kind:RoleBinding**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**name:read-pods**

**namespace:default**

**subjects: #主体**

**- kind:User**

**name:jane**

**apiGroup:rbac.authorization.k8s.io**

**roleRef: #引用的角色**

**kind:Role**

**name:pod-reader**

**apiGroup:rbac.authorization.k8s.io**

角色绑定也可以通过引用集群角色授予访问权限，当主体对资源的访问仅限与本命名空间，这就允许管理员定义整个集群的公共角色集合，然后在多个命名空间中进行复用。例如，下面的角色绑定引用了集群角色，但是“dave”用户也仅仅只能读取“development”命名空间中的secrets资源：

**# This role binding allows "dave" to read secrets in the "development" namespace.**

**kind:RoleBinding**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**name:read-secrets**

**namespace:development# This only grants permissions within the "development" namespace.**

**subjects:**

**- kind:User**

**name:dave**

**apiGroup:rbac.authorization.k8s.io**

**roleRef:**

**kind:ClusterRole**

**name:secret-reader**

**apiGroup:rbac.authorization.k8s.io**

集群角色可以被用来在集群层面和整个命名空间进行授权。下面的示例允许在“manager”组的用户能够访问所有命名空间中的保密字典资源。

**# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.**

**kind:ClusterRoleBinding**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**name:read-secrets-global**

**subjects:**

**- kind:Group**

**name:manager**

**apiGroup:rbac.authorization.k8s.io**

**roleRef:**

**kind:ClusterRole**

**name:secret-reader**

**apiGroup:rbac.authorization.k8s.io**

#### ****资源****

在Kubernets中，主要的资源包括：Pods、Nodes、Services、Deployment、Replicasets、Statefulsets、Namespace、Persistents、Secrets和ConfigMaps等。另外，有些资源下面存在子资源，例如：Pod下就存在log子资源：

GET /api/v1/namespaces/{namespace}/pods/{name}/log

下面的例子显示，“**pod-and-pod-logs-reader**”角色能够对“pods”和“pods/log”进行访问：

**kind:Role**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**namespace:default**

**name:pod-and-pod-logs-reader**

**rules:**

**- apiGroups:[""]**

**resources:["pods","pods/log"]**

**verbs:["get","list"]**

也可以通过resourceNamess指定特定的资源实例，以限制角色只能够对实例进行访问控制：

**kind:Role**

**apiVersion:rbac.authorization.k8s.io/v1**

**metadata:**

**namespace:default**

**name:configmap-updater**

**rules:**

**- apiGroups:[""]**

**resources:["configmaps"]**

**resourceNames:["my-configmap"]**

**verbs:["update","get"]**

#### ****主体****

RBAC授权中的主体可以是组，用户或者服务帐户。用户通过字符串表示，比如“alice”、 “bob@example.com”等，具体的形式取决于管理员在认证模块中所配置的用户名。system:被保留作为用来Kubernetes系统使用，因此不能作为用户的前缀。组也有认证模块提供，格式与用户类似。**在角色绑定主体的例子：**

1）名称为 “alice@example.com”用户：

**subjects:**

**- kind:User**

**name:"alice@example.com"**

**apiGroup:rbac.authorization.k8s.io**

2）名称为“frontend-admins”的组：

**subjects**:

- **kind**:Group

**name**:"frontend-admins"

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

3）在kube-system命名空间中，名称为“default”的服务帐户：

**subjects**:

- **kind**:ServiceAccount

**name**:default

namespace:**kube**-**system**

4）在所有命名空间中，名称为qa的服务帐户：

**subjects**:

- **kind**:Group

**name**:**system**:**serviceaccounts**:**qa**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

5）所有的服务帐户：

**subjects**:

- **kind**:Group

**name**:**system**:**serviceaccounts**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

6）所有被认证的用户 (version 1.5+):

**subjects**:

- **kind**:Group

**name**:**system**:**authenticated**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

7）所有未被认证的用户 (version 1.5+):

**subjects**:

- **kind**:Group

**name**:**system**:**unauthenticated**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

8）所有用户(version 1.5+):

**subjects**:

- **kind**:Group

**name**:**system**:**authenticated**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

- **kind**:Group

**name**:**system**:**unauthenticated**

**apiGroup**:**rbac**.**authorization**.**k8s**.**io**

### ****命令行工具****

Kubernetes可以通过命令工具进行角色绑定，即进行访问授权。

#### ****创建rolebinding****

在指定的命名空间中进行角色绑定：

1）在“acme”命名空间中，将“admin”集群角色授予“bob”用户：

$ kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob --namespace=acme

2）在“acme”命名空间中，将“admin”集群角色授予“acme:myapp”服务帐户：

$ kubectl create rolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp --namespace=acme

#### ****创建clusterrolebinding****

在整个集群中进行角色绑定：

1）在整个集群中，授予”cluster-admin”集群角色给”root”用户：

$ kubectl create clusterrolebinding root-cluster-admin-binding --clusterrole=cluster-admin --user=root

2）在整个集群中，授予”system:node”集群角色给“kubelet”用户：

$ kubectl create clusterrolebinding kubelet-node-binding --clusterrole=system:node --user=kubelet

3）在整个集群中，授予”view”集群角色给”acme:myapp”服务帐户：

$ kubectl create clusterrolebinding myapp-view-binding --clusterrole=view --serviceaccount=acme:myapp

### ****服务帐户权限****

默认情况下，RBAC策略授予控制组件、Node和控制器作用域的权限，但是未授予“kube-system”命名空间外服务帐户的访问权限。这就允许管理员按照需要将特定角色授予服务帐户。

从最安全到最不安全的顺序，方法如下：

**1）授予角色给一个指定应用的服务帐户(最佳实践)**

这要求在Pod规格中指定serviveAccountName，同时此服务帐户已被创建(通过API、kubectl create serviceaccount等)。例如，在“my-namespace”命名空间内，授予”my-sa”服务帐户“view”集群角色：

kubectl create rolebinding my-sa-view \

--clusterrole=view \

--serviceaccount=my-namespace:my-sa \

--namespace=my-namespace

**2）在一个命名空间授予“view”集群角色给“default”服务帐户**

如果应用没有指定serviceAccountName，它将使用”default” 服务帐户。例如，例如，在“my-namespace”命名空间内，授予”default”服务帐户“view”集群角色：

kubectl create rolebinding default-view \

--clusterrole=view \

--serviceaccount=my-namespace:default \

--namespace=my-namespace

当前，在”kube-system“命名空间中，很多插件作为”default“服务帐户进行运行。为了允许超级用户访问这些插件，在“kube-system”命名空间中授予”cluster-admin“角色给”default”帐户。

$ kubectl create clusterrolebinding add-on-cluster-admin \

--clusterrole=cluster-admin \

--serviceaccount=kube-system:default

**3）在一个命名空间中，授予角色给所有的服务帐户：**

如果希望在一个命名空间中的所有应用都拥有一个角色，而不管它们所使用的服务帐户，可以授予角色给服务帐户组。例如，在“my-namespace”命名空间中，将”view“集群角色授予“system:serviceaccounts:my-namespace“组：

$ kubectl create rolebinding serviceaccounts-view \

--clusterrole=view \

--group=system:serviceaccounts:my-namespace \

--namespace=my-namespace

**4）在整个集群中授予一个角色给所有的服务帐户 (不推荐)**

如果不想按照每个命名空间管理权限，可以在整个集群的访问进行授权。例如，在整个集群层面，将”view“集群角色授予“sytem:serviceaccounts“：

$ kubectl create clusterrolebinding serviceaccounts-view \

--clusterrole=view \

--group=system:serviceaccounts

**5）在整个集群中授予超级用户访问所有的服务帐户 (强烈不推荐)**

如果对访问权限不太重视，可以授予超级用户访问所有的服务帐户。

$ kubectl create clusterrolebinding serviceaccounts-cluster-admin \

--clusterrole=cluster-admin \

--group=system:serviceaccounts

**6）宽松的RBAC权限**

下面的策略允许所有的服务帐户作为集群管理员。在容器中运行的应用将自动的收取到服务帐户证书，并执行所有的API行为。包括查看保密字典恩将和修改权限，这是不被推荐的访问策略。

$ kubectl create clusterrolebinding permissive-binding \

--clusterrole=cluster-admin \

--user=admin \

--user=kubelet \

--group=system:serviceaccounts

## 日志管理

日志管理用于记录Kubernetes中用户和系统的行为，用于帮助管理员进行行为审计和问题处理。日志管理不属于Kubernetes本身的功能范畴，因此需要借助于第三方的功能组件进行实现。本章节从日志管理的整体方案入手，描述如何为Kubernetes搭建完整日志管理。

### 统一日志管理的整体方案

通过应用和系统日志可以了解Kubernetes集群内所发生的事情，对于调试问题和监视集群活动来说日志非常有用。对于大部分的应用来说，都会具有某种日志机制。因此，大多数容器引擎同样被设计成支持某种日志机制。对于容器化应用程序来说，最简单和最易接受的日志记录方法是将日志内容写入到标准输出和标准错误流。但是，容器引擎或运行时提供的本地功能通常不足以支撑完整的日志记录解决方案。例如，如果一个容器崩溃、一个Pod被驱逐、或者一个Node死亡，应用相关者可能仍然需要访问应用程序的日志。因此，日志应该具有独立于Node、Pod或者容器的单独存储和生命周期，这个概念被称为群集级日志记录。群集级日志记录需要一个独立的后端来存储、分析和查询日志。Kubernetes本身并没有为日志数据提供原生的存储解决方案，但可以将许多现有的日志记录解决方案集成到Kubernetes集群中。在Kubernetes中，有三个层次的日志：

* 基础日志
* Node级别的日志
* 群集级别的日志架构

#### 基础日志

kubernetes基础日志即将日志数据输出到标准输出流，可以使用kubectl  logs命令获取容器日志信息。如果Pod中有多个容器，可以通过将容器名称附加到命令来指定要访问哪个容器的日志。例如，在Kubernetes集群中的devops命名空间下有一个名称为nexus3-f5b7fc55c-hq5v7的Pod，就可以通过如下的命令获取日志：

$ kubectl logs nexus3-f5b7fc55c-hq5v7 --namespace=devops

#### Node级别的日志

容器化应用写入到stdout和stderr的所有内容都是由容器引擎处理和重定向的。例如，docker容器引擎会将这两个流重定向到日志记录驱动，在Kubernetes中该日志驱动被配置为以json格式写入文件。docker json日志记录驱动将每一行视为单独的消息。当使用docker日志记录驱动时，并不支持多行消息，因此需要在日志代理级别或更高级别上处理多行消息。

默认情况下，如果容器重新启动，kubectl将会保留一个已终止的容器及其日志。如果从Node中驱逐Pod，那么Pod中所有相应的容器也会连同它们的日志一起被驱逐。Node级别的日志中的一个重要考虑是实现日志旋转，这样日志不会消耗Node上的所有可用存储。Kubernetes目前不负责旋转日志，部署工具应该建立一个解决方案来解决这个问题。

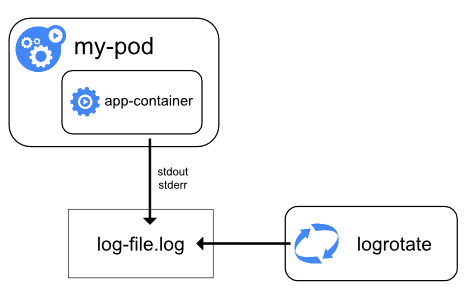


图6-2 日志管理解决方案

在Kubernetes中有两种类型的系统组件：运行在容器中的组件和不在容器中运行的组件。例如：

* Kubernetes调度器和kube-proxy在容器中运行。
* kubelet和容器运行时，例如docker，不在容器中运行。

在带有systemd的机器上，kubelet和容器运行时写入journaId。如果systemd不存在，它们会在/var/log目录中写入.log文件。在容器中的系统组件总是绕过默认的日志记录机制，写入到/var/log目录，它们使用golg日志库。可以找到日志记录中开发文档中那些组件记录严重性的约定。  
类似于容器日志，在/var/log目录中的系统组件日志应该被旋转。这些日志被配置为每天由logrotate进行旋转，或者当大小超过100mb时进行旋转。

#### 集群级别的日志架构

Kubernetes本身没有为群集级别日志记录提供原生解决方案，但有几种常见的方法可以采用：

* 使用运行在每个Node上的Node级别的日志记录代理；
* 在应用Pod中包含一个用于日志记录的sidecar；
* 将日志直接从应用内推到后端。

经过综合考虑，本文采用通过在每个Node上包括Node级别的日志记录代理来实现群集级别日志记录。日志记录代理暴露日志或将日志推送到后端的专用工具。通常，logging-agent是一个容器，此容器能够访问该Node上的所有应用程序容器的日志文件。

因为日志记录必须在每个Node上运行，所以通常将它作为DaemonSet副本、或一个清单Pod或Node上的专用本机进程。然而，后两种方法后续将会被放弃。使用Node级别日志记录代理是Kubernetes集群最常见和最受欢迎的方法，因为它只为每个节点创建一个代理，并且不需要对节点上运行的应用程序进行任何更改。但是，Node级别日志记录仅适用于应用程序的标准输出和标准错误。

Kubernetes本身并没有指定日志记录代理，但是有两个可选的日志记录代理与Kubernetes版本打包发布：和谷歌云平台一起使用的Stackdriver和Elasticsearch，两者都使用自定义配置的fluentd作为Node上的代理。在本文的方案中，Logging-agent 采用 Fluentd，而 Logging Backend 采用 Elasticsearch，前端展示采用Grafana。即通过 Fluentd 作为 Logging-agent 收集日志，并推送给后端的Elasticsearch；Grafana从Elasticsearch中获取日志，并进行统一的展示。

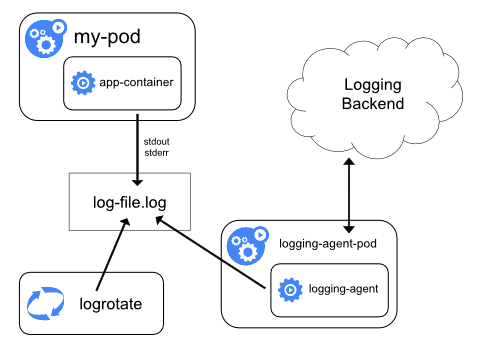


图6-3 基于Elasticseach的日志管理解决方案

### 安装统一日志管理的组件

在本文中采用使用Node日志记录代理的方面进行Kubernetes的统一日志管理，相关的工具采用：

* 日志记录代理（logging-agent）：日志记录代理用于从容器中获取日志信息，使用**Fluentd**；
* 日志记录后台（Logging-Backend）：日志记录后台用于处理日志记录代理推送过来的日志，使用**Elasticsearch**；
* 日志记录展示：日志记录展示用于向用户显示统一的日志信息，使用**Kibana**。

在Kubernetes中通过了Elasticsearch 附加组件，此组件包括Elasticsearch、Fluentd和Kibana。Elasticsearch是一种负责存储日志并允许查询的搜索引擎。Fluentd从Kubernetes中获取日志消息，并发送到Elasticsearch；而Kibana是一个图形界面，用于查看和查询存储在Elasticsearch中的日志。

#### 安装部署Elasticsearch

Elasticsearch是一个基于Apache Lucene(TM)的开源搜索和数据分析引擎引擎，Elasticsearch使用Java进行开发，并使用Lucene作为其核心实现所有索引和搜索的功能。它的目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。Elasticsearch不仅仅是Lucene和全文搜索，它还提供如下的能力：

* 分布式的实时文件存储，每个字段都被索引并可被搜索；
* 分布式的实时分析搜索引擎；
* 可以扩展到上百台服务器，处理PB级结构化或非结构化数据。

在Elasticsearch中，包含多个索引（Index），相应的每个索引可以包含多个类型（Type），这些不同的类型每个都可以存储多个文档（Document），每个文档又有多个属性。索引 (index) 类似于传统关系数据库中的一个数据库，是一个存储关系型文档的地方。Elasticsearch 使用的是标准的 RESTful API 和 JSON。此外，还构建和维护了很多其他语言的客户端，例如 Java, Python, .NET, 和 PHP。

下面是Elasticsearch的YAML配置文件，在此配置文件中，定义了一个名称为elasticsearch-logging的ServiceAccount，并授予其能够对命名空间、服务和端点读取的访问权限；并以StatefulSet类型部署Elasticsearch。

# RBAC authn and authz

# 为elastic定义一个ServiceAccount

apiVersion: v1

kind: ServiceAccount

metadata:

  name: elasticsearch-logging

  namespace: kube-system

  labels:

    k8s-app: elasticsearch-logging

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

---

# 为elastic定义一个集群角色

kind: ClusterRole

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  name: elasticsearch-logging

  labels:

    k8s-app: elasticsearch-logging

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

rules:

- apiGroups:

  - ""

  resources:

  - "services"

  - "namespaces"

  - "endpoints"

  verbs:

  - "get"

---

# 将用户和角色进行绑定，即授予用户角色所承担权限

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  namespace: kube-system

  name: elasticsearch-logging

  labels:

    k8s-app: elasticsearch-logging

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

subjects:

- kind: ServiceAccount

  name: elasticsearch-logging

  namespace: kube-system

  apiGroup: ""

roleRef:

  kind: ClusterRole

  name: elasticsearch-logging

  apiGroup: ""

---

# 将elastic部署为StatefulSet类型的应用

apiVersion: apps/v1

kind: StatefulSet

metadata:

  name: elasticsearch-logging

  namespace: kube-system

  labels:

    k8s-app: elasticsearch-logging

    version: v6.2.5

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

spec:

  serviceName: elasticsearch-logging

  replicas: 2

  selector:

    matchLabels:

      k8s-app: elasticsearch-logging

      version: v6.2.5

  template:

    metadata:

      labels:

        k8s-app: elasticsearch-logging

        version: v6.2.5

        kubernetes.io/cluster-service: "true"

    spec:

      # 在集群内部使用elasticsearch-logging内部用户

serviceAccountName: elasticsearch-logging

      containers:

      - image: k8s.gcr.io/elasticsearch:v6.2.5

        name: elasticsearch-logging

        resources:

          # 设置本应用所使用的资源

          limits:

            cpu: 1000m

          requests:

            cpu: 100m

        ports:

        - containerPort: 9200

          name: db

          protocol: TCP

        - containerPort: 9300

          name: transport

          protocol: TCP

        volumeMounts:

        - name: elasticsearch-logging

          mountPath: /data

        env:

        - name: "NAMESPACE"

          valueFrom:

            fieldRef:

              fieldPath: metadata.namespace

      volumes:

      - name: elasticsearch-logging

        emptyDir: {}

      initContainers:

      - image: alpine:3.6

        command: ["/sbin/sysctl", "-w", "vm.max\_map\_count=262144"]

        name: elasticsearch-logging-init

        securityContext:

          privileged: true

通过执行如下的命令部署Elasticsearch：

$ kubectl create -f {path}/es-statefulset.yaml

下面Elasticsearch的代理服务YAML配置文件，代理服务暴露的端口为9200。

apiVersion: v1

kind: Service

metadata:

name: elasticsearch-logging

namespace: kube-system

labels:

k8s-app: elasticsearch-logging

kubernetes.io/cluster-service: "true"

addonmanager.kubernetes.io/mode: Reconcile

kubernetes.io/name: "Elasticsearch"

spec:

ports:

- port: 9200

protocol: TCP

targetPort: db

selector:

k8s-app: elasticsearch-logging

通过执行如下的命令部署Elasticsearch的代理服务：

$ kubectl create -f {path}/es-service.yaml

#### 安装部署Fluentd

Fluentd是一个开源数据收集器，通过它能对数据进行统一收集和消费，能够更好地使用和理解数据。Fluentd将数据结构化为JSON，从而能够统一处理日志数据，包括：收集、过滤、缓存和输出。Fluentd是一个基于插件体系的架构，包括输入插件、输出插件、过滤插件、解析插件、格式化插件、缓存插件和存储插件，通过插件可以扩展和更好的使用Fluentd。Fluentd的整体处理过程如下，通过Input插件获取数据，并通过Engine进行数据的过滤、解析、格式化和缓存，最后通过Output插件将数据输出给特定的终端。

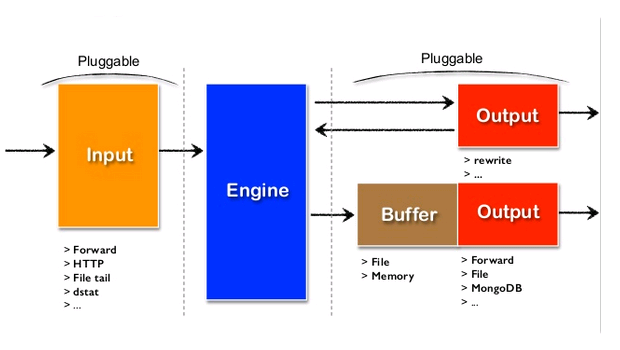


图6-4 Fluented数据处理过程

在本文中， Fluentd 作为 Logging-agent 进行日志收集，并将收集到的日子推送给后端的Elasticsearch。对于Kubernetes来说，DaemonSet确保所有（或一些）Node会运行一个Pod副本。因此，Fluentd被部署为DaemonSet，它将在每个节点上生成一个pod，以读取由kubelet，容器运行时和容器生成的日志，并将它们发送到Elasticsearch。为了使Fluentd能够工作，每个Node都必须标记beta.kubernetes.io/fluentd-ds-ready=true。

下面是Fluentd的ConfigMap配置文件，此文件定义了Fluentd所获取的日志数据源，以及将这些日志数据输出到Elasticsearch中。

kind: ConfigMap

apiVersion: v1

metadata:

  name: fluentd-es-config-v0.1.4

  namespace: kube-system

  labels:

    addonmanager.kubernetes.io/mode: Reconcile

data:

  system.conf: |-

    <system>

      root\_dir /tmp/fluentd-buffers/

    </system>

  containers.input.conf: |-

    <source>

      @id fluentd-containers.log

      @type tail

      path /var/log/containers/\*.log

      pos\_file /var/log/es-containers.log.pos

      time\_format %Y-%m-%dT%H:%M:%S.%NZ

      tag raw.kubernetes.\*

      read\_from\_head true

      <parse>

        @type multi\_format

        <pattern>

          format json

          time\_key time

          time\_format %Y-%m-%dT%H:%M:%S.%NZ

        </pattern>

        <pattern>

          format /^(?<time>.+) (?<stream>stdout|stderr) [^ ]\* (?<log>.\*)$/

          time\_format %Y-%m-%dT%H:%M:%S.%N%:z

        </pattern>

      </parse>

    </source>

    # Detect exceptions in the log output and forward them as one log entry.

    <match raw.kubernetes.\*\*>

      @id raw.kubernetes

      @type detect\_exceptions

      remove\_tag\_prefix raw

      message log

      stream stream

      multiline\_flush\_interval 5

      max\_bytes 500000

      max\_lines 1000

    </match>

  output.conf: |-

    # Enriches records with Kubernetes metadata

    <filter kubernetes.\*\*>

      @type kubernetes\_metadata

    </filter>

    <match \*\*>

      @id elasticsearch

      @type elasticsearch

      @log\_level info

      include\_tag\_key true

      host elasticsearch-logging

      port 9200

      logstash\_format true

      <buffer>

        @type file

        path /var/log/fluentd-buffers/kubernetes.system.buffer

        flush\_mode interval

        retry\_type exponential\_backoff

        flush\_thread\_count 2

        flush\_interval 5s

        retry\_forever

        retry\_max\_interval 30

        chunk\_limit\_size 2M

        queue\_limit\_length 8

        overflow\_action block

      </buffer>

    </match>

通过执行如下的命令创建Fluentd的ConfigMap：

$ kubectl create -f {path}/fluentd-es-configmap.yaml

Fluentd本身的YAML配置文件如下所示：

apiVersion: v1

kind: ServiceAccount

metadata:

  name: fluentd-es

  namespace: kube-system

  labels:

    k8s-app: fluentd-es

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

---

kind: ClusterRole

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  name: fluentd-es

  labels:

    k8s-app: fluentd-es

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

rules:

- apiGroups:

  - ""

  resources:

  - "namespaces"

  - "pods"

  verbs:

  - "get"

  - "watch"

  - "list"

---

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  name: fluentd-es

  labels:

    k8s-app: fluentd-es

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

subjects:

- kind: ServiceAccount

  name: fluentd-es

  namespace: kube-system

  apiGroup: ""

roleRef:

  kind: ClusterRole

  name: fluentd-es

  apiGroup: ""

---

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: fluentd-es-v2.2.0

  namespace: kube-system

  labels:

    k8s-app: fluentd-es

    version: v2.2.0

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

spec:

  selector:

    matchLabels:

      k8s-app: fluentd-es

      version: v2.2.0

  template:

    metadata:

      labels:

        k8s-app: fluentd-es

        kubernetes.io/cluster-service: "true"

        version: v2.2.0

      # This annotation ensures that fluentd does not get evicted if the node

      # supports critical pod annotation based priority scheme.

      # Note that this does not guarantee admission on the nodes (#40573).

      annotations:

        scheduler.alpha.kubernetes.io/critical-pod: ''

        seccomp.security.alpha.kubernetes.io/pod: 'docker/default'

    spec:

      priorityClassName: system-node-critical

      serviceAccountName: fluentd-es

      containers:

      - name: fluentd-es

        image: k8s.gcr.io/fluentd-elasticsearch:v2.2.0

        env:

        - name: FLUENTD\_ARGS

          value: --no-supervisor -q

        resources:

          limits:

            memory: 500Mi

          requests:

            cpu: 100m

            memory: 200Mi

        volumeMounts:

        - name: varlog

          mountPath: /var/log

        - name: varlibdockercontainers

          mountPath: /var/lib/docker/containers

          readOnly: true

        - name: config-volume

          mountPath: /etc/fluent/config.d

      nodeSelector:

        beta.kubernetes.io/fluentd-ds-ready: "true"

      terminationGracePeriodSeconds: 30

      volumes:

      - name: varlog

        hostPath:

          path: /var/log

      - name: varlibdockercontainers

        hostPath:

          path: /var/lib/docker/containers

      - name: config-volume

        configMap:

          name: fluentd-es-config-v0.1.4

通过执行如下的命令部署Fluentd：

$ kubectl create -f {path}/fluentd-es-ds.yaml

#### 安装部署Kibana

Kibana是一个开源的分析与可视化平台，被设计用于和Elasticsearch一起使用的。通过kibana可以搜索、查看和交互存放在Elasticsearch中的数据，利用各种不同的图表、表格和地图等，Kibana能够对数据进行分析与可视化。Kibana部署的YAML如下所示，通过环境变量ELASTICSEARCH\_URL，指定所获取日志数据的Elasticsearch服务，此处为：http://elasticsearch-logging:9200，elasticsearch.cattle-logging是elasticsearch在Kubernetes中代理服务的名称。在Fluented配置文件中，有下面的一些关键指令：

* **source**指令确定输入源。
* **match**指令确定输出目标。
* **filter**指令确定事件处理管道。
* **system**指令设置系统范围的配置。
* **label**指令将输出和过滤器分组以进行内部路由
* **@include**指令包含其他文件。

apiVersion: apps/v1

kind: Deployment

metadata:

  name: kibana-logging

  namespace: kube-system

  labels:

    k8s-app: kibana-logging

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

spec:

  replicas: 1

  selector:

    matchLabels:

      k8s-app: kibana-logging

  template:

    metadata:

      labels:

        k8s-app: kibana-logging

      annotations:

        seccomp.security.alpha.kubernetes.io/pod: 'docker/default'

    spec:

      containers:

      - name: kibana-logging

        image: docker.elastic.co/kibana/kibana-oss:6.2.4

        resources:

          # need more cpu upon initialization, therefore burstable class

          limits:

            cpu: 1000m

          requests:

            cpu: 100m

        env:

          - name: ELASTICSEARCH\_URL

            value: http://elasticsearch-logging:9200

        ports:

        - containerPort: 5601

          name: ui

          protocol: TCP

通过执行如下的命令部署Kibana的代理服务：

$ kubectl create -f {path}/kibana-deployment.yaml

下面Kibana的代理服务YAML配置文件，代理服务的类型为NodePort。

apiVersion: v1

kind: Service

metadata:

  name: kibana-logging

  namespace: kube-system

  labels:

    k8s-app: kibana-logging

    kubernetes.io/cluster-service: "true"

    addonmanager.kubernetes.io/mode: Reconcile

    kubernetes.io/name: "Kibana"

spec:

  type: NodePort

  ports:

  - port: 5601

    protocol: TCP

    targetPort: ui

  selector:

    k8s-app: kibana-logging

通过执行如下的命令部署Kibana的代理服务：

$ kubectl create -f {path}/kibana-service.yaml

### 日志数据展示

通过如下命令获取Kibana的对外暴露的端口：

$ kubectl get svc --namespace=kube-system



图6-5 获取kibana-logging的外部端口

从输出的信息可以知道，kibana对外暴露的端口为30471，因此在Kubernetes集群外可以通过：http://{NodeIP}:30471 访问kibana。

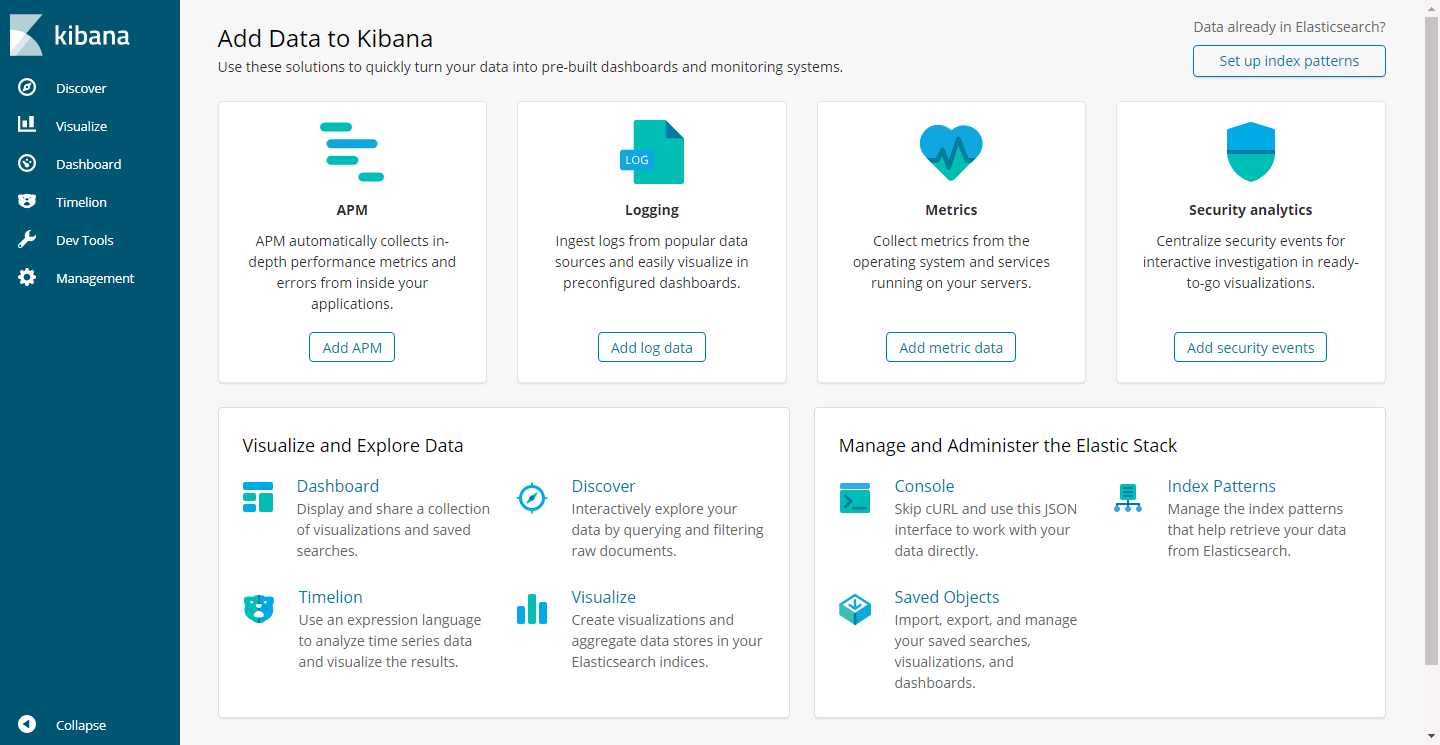


图6-6 kibana界面

通过点击“Discover”，就能够实时看看从容器中获取到的日志信息：

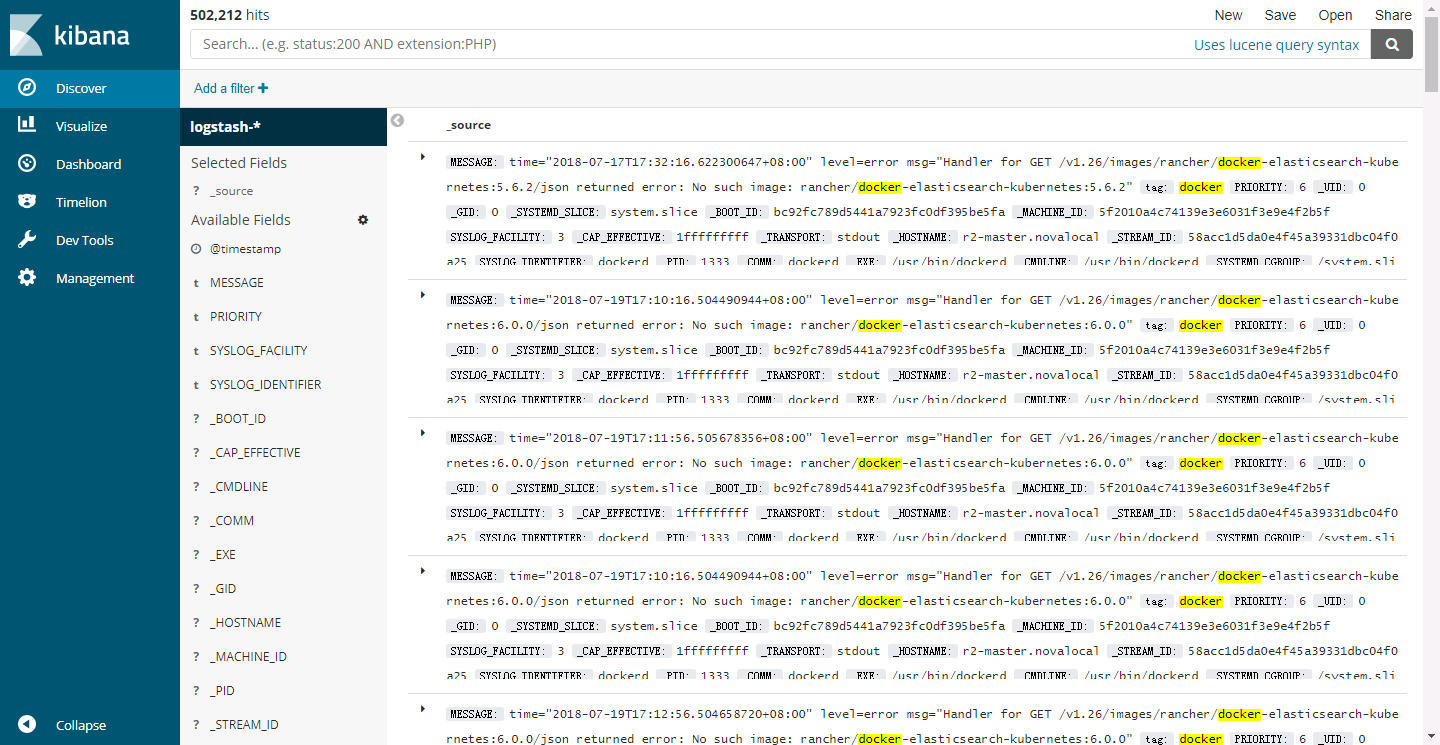


图6-7 通过kibana展示的容器日志信息

# Kubernetes客户端和工具(2019-04-10 Y)

用户可以通过多种途径与Kubernetes进行交互，这些途径就是Kubernetes的客户端和相关的工具。在此章节，对Kubectl命令行工具、Dashboard界面和Helm部署这三个工具进行了讲解。

## Kubectl命令行工具

Kubectl命令行工具是用户与kubernetes进行交互的最核心工具，包括查看Kubernetes运行的相关信息，以及在Kubernetes部署容器化应用。在这里会对kubectl进行整体介绍，并带来用户进行kubectl安装部署和提供常用命令的示例。

### kubectl概述

kubectl是Kubernetes的命令行工具，通过kubectl能够对Kuberentes集群本身进行管理，并能够在集群上进行容器化应用的安装部署。运行kubectl命令的语法如下所示：

$ kubectl [command] [TYPE] [NAME] [flags]

这里的command，TYPE、NAME和flags为：

* **comand**：是指定要对资源执行的操作，例如create、get、describe和delete；
* **TYPE**：指定所有进行操作的资源类型，资源类型是大小写敏感的，开发者能够以单数、复数和缩略的形式。例如：

$ kubectl get pod pod1

$ kubectl get pods pod1

$ kubectl get po pod1

* **NAME**：指定资源的名称，名称也大小写敏感的。如果省略名称，则会显示所有的资源，例如:

 $ kubectl get pods

* **flags**：指定可选的参数。例如，可以使用-s或者–server参数指定Kubernetes API server的地址和端口。

另外，可以通过运行kubectl help命令能够获取更多的信息。

#### kubectl操作

kubectl作为kubernetes的命令行工具，主要的职责就是对集群中的资源的对象进行操作，这些操作包括对资源对象的创建、删除和查看等。下表中显示了kubectl支持的所有操作，以及这些操作的语法和描述信息：

表 7-1 kubectl操作语法

| **操作** | **语法** | **描述** |
| --- | --- | --- |
| annotate | kubectl annotate (-f FILENAME \| TYPE NAME \| TYPE/NAME) KEY\_1=VAL\_1 … KEY\_N=VAL\_N [–overwrite] [–all] [–resource-version=version] [flags] | 添加或更新一个或多个资源的注释 |
| api-versions | kubectl api-versions [flags] | 列出可用的API版本 |
| apply | kubectl apply -f FILENAME [flags] | 将来自于文件或stdin的配置变更应用到主要对象中 |
| attach | kubectl attach POD -c CONTAINER [-i] [-t] [flags] | 连接到正在运行的容器上，以查看输出流或与容器交互（stdin） |
| autoscale | kubectl autoscale (-f FILENAME \| TYPE NAME \| TYPE/NAME) [–min=MINPODS] –max=MAXPODS [–cpu-percent=CPU] [flags] | 自动扩宿容由副本控制器管理的Pod |
| cluster-info | kubectl cluster-info [flags] | 显示群集中的主节点和服务 |
| config | kubectl config SUBCOMMAND [flags] | 修改kubeconfig文件 |
| create | kubectl create -f FILENAME [flags] | 从文件或stdin中创建一个或多个资源对象 |
| delete | kubectl delete (-f FILENAME \| TYPE [NAME \| /NAME \| -l label \| –all]) [flags] | 删除资源对象。 |
| describe | kubectl describe (-f FILENAME \| TYPE [NAME\_PREFIX \| /NAME \| -l label]) [flags] | 显示一个或者多个资源对象的详细信息 |
| edit | kubectl edit (-f FILENAME \| TYPE NAME \| TYPE/NAME) [flags] | 通过默认编辑器编辑和更新服务器上的一个或多个资源对象 |
| exec | kubectl exec POD [-c CONTAINER] [-i] [-t] [flags] [– COMMAND [args…]] | 在Pod的容器中执行一个命令 |
| explain | kubectl explain [–include-extended-apis=true] [–recursive=false] [flags] | 获取Pod、Node和服务等资源对象的文档 |
| expose | kubectl expose (-f FILENAME \| TYPE NAME \| TYPE/NAME) [–port=port] [–protocol=TCP\|UDP] [–target-port=number-or-name] [–name=name] [—-external-ip=external-ip-of-service] [–type=type] [flags] | 为副本控制器、服务或Pod等暴露一个新的服务 |
| get | kubectl get (-f FILENAME \| TYPE [NAME \| /NAME \| -l label]) [–watch] [–sort-by=FIELD] [[-o \| –output]=OUTPUT\_FORMAT] [flags] | 列出一个或多个资源 |
| label | kubectl label (-f FILENAME \| TYPE NAME \| TYPE/NAME) KEY\_1=VAL\_1 … KEY\_N=VAL\_N [–overwrite] [–all] [–resource-version=version] [flags] | 为一个或者多个资源对象添加或更新标签 |
| logs | kubectl logs POD [-c CONTAINER] [–follow] [flags] | 显示Pod中一个容器的日志 |
| patch | kubectl patch (-f FILENAME \| TYPE NAME \| TYPE/NAME) –patch PATCH [flags] | 使用策略合并补丁过程更新资源对象中的一个或多个字段 |
| port-forward | kubectl port-forward POD [LOCAL\_PORT:]REMOTE\_PORT […[LOCAL\_PORT\_N:]REMOTE\_PORT\_N] [flags] | 将一个或多个本地端口转发到Pod |
| proxy | kubectl proxy [–port=PORT] [–www=static-dir] [–www-prefix=prefix] [–api-prefix=prefix] [flags] | 为Kubernetes API服务器运行一个代理 |
| replace | kubectl replace -f FILENAME | 从文件或stdin中替换资源对象 |
| rolling-update | kubectl rolling-update OLD\_CONTROLLER\_NAME ([NEW\_CONTROLLER\_NAME] –image=NEW\_CONTAINER\_IMAGE \| -f NEW\_CONTROLLER\_SPEC) [flags] | 通过逐步替换指定的副本控制器和Pod的方式来进行滚动升级 |
| run | kubectl run NAME –image=image [–env=”key=value”] [–port=port] [–replicas=replicas] [–dry-run=bool] [–overrides=inline-json] [flags] | 在集群上运行一个指定的镜像 |
| scale | kubectl scale (-f FILENAME \| TYPE NAME \| TYPE/NAME) –replicas=COUNT [–resource-version=version] [–current-replicas=count] [flags] | 扩宿容副本集的数量 |
|  |  |  |
| version | kubectl version [–client] [flags] | 显示Kubernetes客户端和服务器端的版本信息 |

#### 资源对象类型

在kubernetes中，提供了很多的资源对象，开发和运维人员可以通过这些对象对容器进行编排。在下表中，是kubectl所支持的资源对象类型，以及它们的缩略别名。

表 7-2 资源对象列表

| **资源对象类型** | **缩略别名** |
| --- | --- |
| apiservices |  |
| certificatesigningrequests | csr |
| clusters |  |
| clusterrolebindings |  |
| clusterroles |  |
| componentstatuses | cs |
| configmaps | cm |
| controllerrevisions |  |
| cronjobs |  |
| customresourcedefinition | crd |
| daemonsets | ds |
| deployments | deploy |
| endpoints | ep |
| events | ev |
| horizontalpodautoscalers | hpa |
| ingresses | ing |
| jobs |  |
| limitranges | limits |
| namespaces | ns |
| networkpolicies | netpol |
| nodes | no |
| persistentvolumeclaims | pvc |
| persistentvolumes | pv |
| poddisruptionbudget | pdb |
| podpreset |  |
| pods | po |
| podsecuritypolicies | psp |
| podtemplates |  |
| replicasets | rs |
| replicationcontrollers | rc |
| resourcequotas | quota |
| rolebindings |  |
| roles |  |
| secrets |  |
| serviceaccounts | sa |
| services | svc |
| statefulsets |  |
| storageclasses |  |

#### 输出选项

默认情况下，kubectl的输出格式为纯文本格式。为了信息查看更加友好和可读，可以通过-o或者–output字段指定命令的输出格式。

$ kubectl [command] [TYPE] [NAME] -o=<output\_format>

表 7-3 kubectl的输出格式

| **输出格式** | **描述** |
| --- | --- |
| -o=custom-columns=<spec> | 使用以逗号分隔的自定义列打印表格 |
| -o=custom-columns-file=<filename> | 使用文件中自定义列打印表格 |
| -o=json | 输出JSON格式的API对象 |
| -o=jsonpath=<template> | 打印在jsonpath表达式中定义的字段 |
| -o=jsonpath-file=<filename> | 打印文件中以jsonpath表达式定义的字段 |
| -o=name | 仅仅输出资源对象的名称 |
| -o=wide | 输出带有附加信息的纯文本格式。对于Pod对象，将会包含Node名称 |
| -o=yaml | 输出YAML格式的API对象 |

### kubectl安装部署

#### ****安装kubectl****

在本文中，kubectl客户端在Windows操作系统下安装，

**1）下载kubectl**

此处是在Windows操作系统下安装，因此下载kubectl.exe，下载地址：<https://storage.googleapis.com/kubernetes-release/release/v1.9.0/bin/windows/amd64/kubectl.exe>。

并将kubectl.exe所在的地址添加至Windows的环境变量的Path中。

其他操作系统下安装kubectl，请参考：<https://kubernetes.io/docs/tasks/tools/install-kubectl/#tabset-2>

**2）配置kubeconfig文件**

在操作系统当前用户的目录下创建.kube文件夹和config文件，并将这个文件复制到Kubernetes的kubeconfig内容到~/.kube/config。在完成配置工作后，就可以在本地使用kubectl。

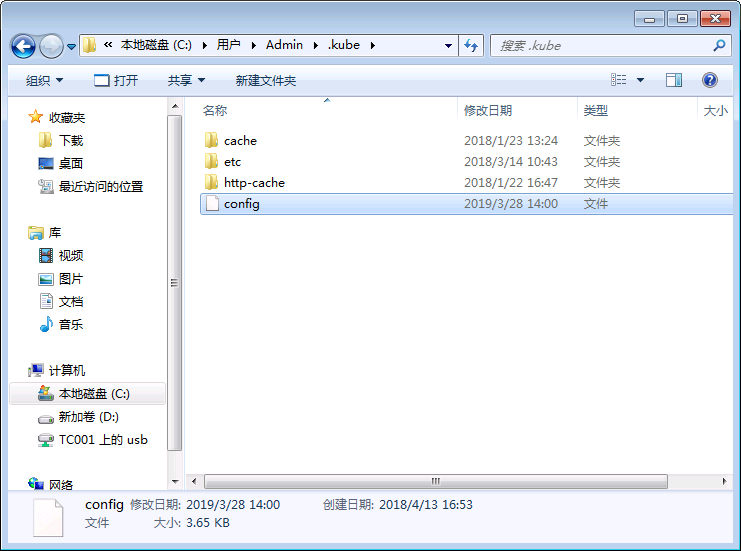


图7-1 .kube目录

#### ****验证****

在安装好kubectl后，需要认证安装是否正确。这里通过执行kubectl命令来获取nodes的信息，如果返回信息正确，则确认安装和配置没有问题。

$ kubectl get nodes

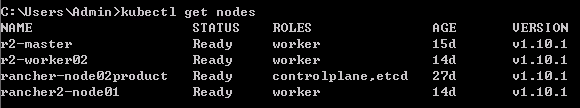


图7-2 集群中nodes的信息

### kubectl的常用命令示例

在此部分将提供常用命令的示例，以帮助读者快速了解和使用kubectl。

1）kubectl create命令

此命令通过文件或者stdin创建一个资源对象，假设这里存在一个nginx部署的YAML配置文件，可以通过执行下面的命令创建部署对象。

$ kubectl create -f nginx-deployment.yaml

nginx部署的YAML配置文件的示例代码如下：

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: nginx

spec:

  replicas: 10

  selector:

    matchLabels:

      app: nginx

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: nginx

    spec:

      imagePullSecrets:

        - name: dc-hspfd

      containers:

      # 应用的镜像

      - image: nginx

        name: nginx

        imagePullPolicy: IfNotPresent

        # 应用的内部端口

        ports:

        - containerPort: 80

          name: nginx80

        # 持久化挂接位置，在docker中

        volumeMounts:

        - mountPath: /usr/share/nginx/html

          name: nginx-data

        - mountPath: /etc/nginx

          name: nginx-conf

      volumes:

      # 宿主机上的目录

      - name: nginx-data

        nfs:

          path: /k8s-nfs/nginx

          server: 192.168.8.150

      - name: nginx-conf

        nfs:

          path: /k8s-nfs/nginx/conf

          server: 192.168.8.150

**2）kubectl get 命令**

通过此命令列出一个或多个资源对象，在这里通过kubectl get命令获取default命名空间下的所有部署。



图7-3 集群中部署的信息

**3）kubectl describe命令**

此命令用于显示一个或多个资源对象的详细信息，在这里通过获取上述nginx部署的信息。

$ kubectl describe deployments/nginx



图7-4 特定部署的详细信息

**4）kubectl exec命令**

此命令用于在Pod中的容器上执行一个命令，此处在nginx的一个容器上执行/bin/bash命令。

$ kubectl exec -it nginx-5847748bf9-49k5k /bin/bash

**5）kubectl logs命令**

此命令用于获取Pod中一个容器的日志信息，此处获取nginx一个容器的日志信息。

$ kubectl logs nginx-5847748bf9-49k5k

**6）kubectl delete命令**

此命令用于删除集群中已存在的资源对象，可以通过指定名称、标签选择器、资源选择器等。此处删除前面创建的nginx部署。

$ kubectl delete deployments/nginx

## Dashboard界面

此章节用于介绍Kubernetes的Web控制台，相对于kubectl来说，Dashboard界面更加易用，但功能没有kubectl强大。

### ****部署Dashboard用户界面****

作为Kubernetes的Web用户界面，用户可以通过Dashboard在Kubernetes集群中部署容器化的应用，对应用进行问题处理和管理，并对集群本身进行管理。通过Dashboard，用户可以查看集群中应用的运行情况，同时也能够基于Dashboard创建或修改部署、任务、服务等Kubernetes的资源。通过部署向导，用户能够对部署进行扩缩容，进行滚动更新、重启Pod和部署新应用。当然，通过Dashboard也能够查看Kubernetes资源的状态。下面是部署Dashboard的过程：

**1）下载kubernetes-dashboard.yaml文件**

通过：<https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml> 地址，能够下载下载kubernetes-dashboard.yaml文件。

**2）编辑kubernetes-dashboard.yaml文件**

通过编辑工具打开kubernetes-dashboard.yaml，并在此文件中的Service部分下添加type: NodePort和nodePort: 30001，添加位置如下图所示。

**3）通过执行如下的命令部署Web UI**

# kubectl create -f  {path}/kubernetes-dashboard.yaml

### ****访问Dashboard用户界面****

在Dashboard部署完成后，就可以通过浏览器访问Kubernetes，访问过程如下：

1）在浏览器中输入:https://{Master IP}:30001，此处为：[https://10.0.32.175:3001](https://10.0.32.175:3001/)，打开页面如下：

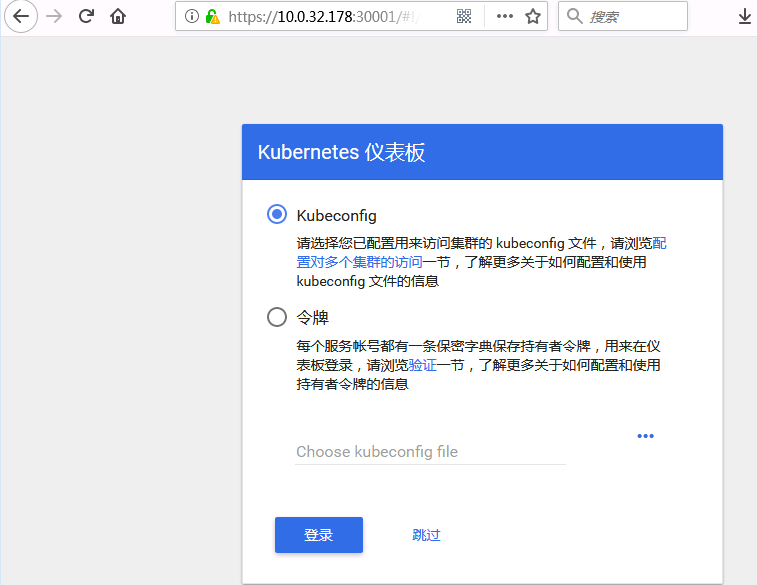


图7-6 Dashboard登录页面

2）创建一个管理员用户

创建amind-user.yaml文件，文件内容如下：

apiVersion: v1

kind: ServiceAccount

metadata:

  name: admin-user

  namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1beta1kind: ClusterRoleBinding

metadata:

  name: admin-user

  annotations:

    rbac.authorization.kubernetes.io/autoupdate: "true"

roleRef:

  apiGroup: rbac.authorization.k8s.io

  kind: ClusterRole

  name: cluster-admin

subjects:

- kind: ServiceAccount

  name: admin-user

  namespace: kube-system

通过执行如下命令创建admin-user：

#kubectl create -f {path}/admin-user.yaml

**3）获取管理员用户的Token**

通过执行如下命令获取系统Token信息：

#kubectl describe  secret admin-user --namespace=kube-system

**4）添加Token至kubeconfig文件**

通过编辑工具打开kubeconfig文件(~/.kube/config)，并添加Token。

**5）导入kubeconfig文件**

在界面中导入kubeconfig文件。

### ****使用Dashboard****

此部分用于告诉读者如何使用Dashboard进行Kubernetes管理，第一部分描述Dashboard所提供的功能；第二部分指导读者在Dashboard中如何部署容器化的应用。

#### Dashboard提供的功能

在默认情况下，Dashboard显示默认(default)命名空间下的对象，也可以通过命名空间选择器选择其他的命名空间。在Dashboard用户界面中能够显示集群大部分的对象类型。

**1）集群管理**

集群管理视图用于对节点、命名空间、持久化存储卷、角色和存储类进行管理。 节点视图显示CPU和内存的使用情况，以及此节点的创建时间和运行状态。 命名空间视图会显示集群中存在哪些命名空间，以及这些命名空间的运行状态。角色视图以列表形式展示集群中存在哪些角色，这些角色的类型和所在的命名空间。 持久化存储卷以列表的方式进行展示，可以看到每一个持久化存储卷的存储总量、访问模式、使用状态等信息；管理员也能够删除和编辑持久化存储卷的YAML文件。

**2） 工作负载**

工作负载视图显示部署、副本集、有状态副本集等所有的工作负载类型。在此视图中，各种工作负载会按照各自的类型进行组织。 工作负载的详细信息视图能够显示应用的详细信息和状态信息，以及对象之间的关系。

**3） 服务发现和负载均衡**

服务发现视图能够将集群内容的服务暴露给集群外的应用，集群内外的应用可以通过暴露的服务调用应用，外部的应用使用外部的端点，内部的应用使用内部端点。

**4） 存储**

存储视图显示被应用用来存储数据的持久化存储卷申明资源。

**5） 配置**

配置视图显示集群中应用运行时所使用配置信息，Kubernetes提供了配置字典（ConfigMaps）和秘密字典（Secrets），通过配置视图，能够编辑和管理配置对象，以及查看隐藏的敏感信息。

**6） 日志视图**

Pod列表和详细信息页面提供了查看日志视图的链接，通过日志视图不但能够查看Pod的日志信息，也能够查看Pod容器的日志信息。通过Dashboard能够根据向导创建和部署一个容器化的应用，当然也可以通过手工的方式输入指定应用信息，或者通过上传YAML和JSON文件来创建和不受应用。

#### 部署应用

在Dashboard中有两种部署容器化应用的方式：一种是通过手动在页面上填写相关信息的方式来创建应用；另外一种是通过YAML文件或者JSON文件来创建应用。

**1）手动创建应用**

通过向导创建和部署容器化应用时，需要提供如下的一些信息：

* 应用名称（App name 必需）： 需要部署的应用的名称。带有此值的标签将会被添加至部署和服务中。在当前的Kubernetes命名空间中，应用名称必须是唯一的。同时，应用名称必须以小写字母开头，以小写字母和数字结尾，可以包含字母、数字和“-”。名称最长为24个字母。
* 容器组个数（Number of pods 必需）： 希望部署的容器组数量。值必须为整数。
* 描述（Description）： 对于应用的描述，将被添加至部署的注释中，并在应用详细信息中显示。
* 标签（Labels）： 应用的默认标签为应用的名称和版本。可以指定其它的标签，这些标签将会被应用至部署、服务、容器组等资源中**。**
* 命名空间（Namespace）：在同一个物理集群中，Kubernetes支持多个虚拟集群。这些虚拟集群被称为命名空间，通过命名空间可以将资源进行逻辑上的划分。通过下列菜单可以选择已有的命名空间，当然也可以创建新的命名空间。命名空间的名称最大的字符数为63，名词可以使用字母、数字“-”，不能包含大写字母，同时也不能全部使用数字。
* 镜像拉取保密字典（Image Pull Secret）： 如果Docker容器镜像是私有的，则有可能需要保密证书。Dashboard通过下拉菜单提供了所有的可用的保密凭证，也允许创建新的保密字典。保密字典名称必须遵循DNS域名语法，例如：new.image-pull.secret。保密字典的内容必须使用基于base64进行加密的，并在.dockercfg文件中进行指定。保密字典名称最长不能超过253个字符。
* 环境变量（Environment variables）： Kubernetes通过环境变量暴露服务， 可以创建环境变量或者使用环境变量的值将参数传递给命令。环境变量能够被应用用来发现服务，环境变量的值可以通过￥（VAR\_NAME）语法被其它变量引用。

**2）上传YAML或JSON文件创建应用**

通过编译工具编写容器化应用的YAML和JSON文件，在Dashboard用户界面中通过上传文件创建和部署应用。

## Helm部署工具

在Kubernetes中部署容器云的应用也是一项有挑战性的工作，Helm的出现就是为了简化在Kubernetes中安装部署容器化应用的一个客户端工具。

### ****Helm介绍****

通过helm，能够帮助开发者定义、安装和升级Kubernetes中的容器化应用。同时，也可以通过helm进行容器化应用的分享。在Kubeapps Hub中提供了包括Redis、MySQL和Jenkins等常见的应用，通过helm可以使用一条命令就能够将其部署安装在自己的Kubernetes集群中。

helm的整体架构如下图所示，Helm架构由Helm客户端、Tiller服务器端和Chart仓库所组成；Tiller部署在Kubernetes中，Helm客户端从Chart仓库中获取Chart安装包，并将其安装部署到Kubernetes集群中。

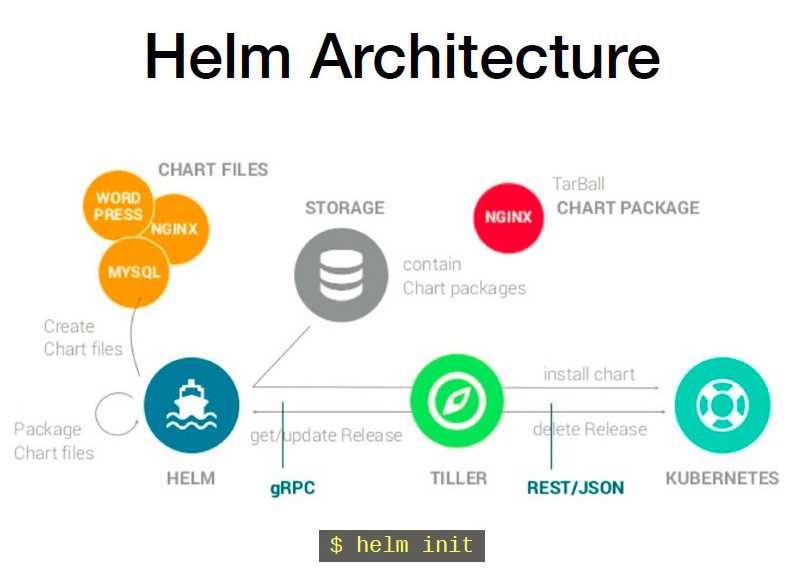


图7-7 helm架构

Helm是管理Kubernetes包的工具，Helm提供的主要能力如下所示：

* 创建新的charts
* 将charts打包成tgz文件
* 与chart仓库交互
* 安装和卸载Kubernetes的应用
* 管理使用Helm安装的charts的生命周期

在Helm中，有三个需要了解的重要概念：

* chart：是创建Kubernetes应用实例的信息集合；
* config：创建发布对象的chart的配置信息
* release：chart的运行实例，包含特定的config

#### ****Helm组件****

Helm由两部分组成，既Helm客户端和Tiller服务器。客户端负责管理chart，服务器发展管理发布。**Helm客户端**：这是一个供终端用户使用的命令行工具，客户端负责如下的工作：

* 本地chart开发
* 管理仓库
* 与Tiller服务器交互
* 发送需要被安装的charts
* 请求关于发布版本的信息
* 请求更新或者卸载已安装的发布版本

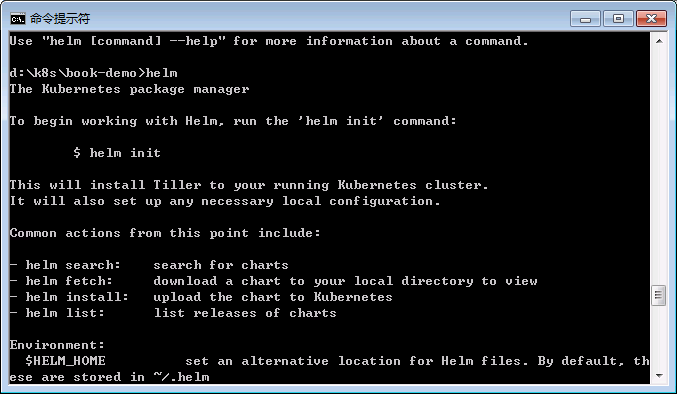


图7-8 helm客户端

**Tiller服务器：** Tiller服务部署在Kubernetes集群中，Helm客户端通过与Tiller服务器进行交互，并最终与Kubernetes API服务器进行交互。 Tiller服务器负责如下的工作：

* 监听来自于Helm客户端的请求
* 组合chart和配置来构建一个发布
* 在Kubernetes中安装，并跟踪后续的发布
* 通过与Kubernetes交互，更新或者chart

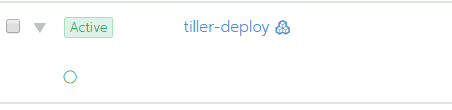


图7-9 部署在Kubernetes中的helm服务端

#### ****Helm技术实现****

Helm客户端是使用Go语言编写的，它通过gRPC协议与Tiller服务器交互。Tiller服务器也是使用Go语言编写的，它使用Kubernetes客户端类库（当前是REST+JSON）与Kubernetes进行通讯。Tiller服务器通过Kubernetes的ConfigMap存储信息，因此本身没有用于存储数据库。

### ****Helm安装部署****

Helm由客户端和服务器所组成，在本文中helm客户端安装在windows操作系统中，服务器端部署在Kubernetes中。

#### ****安装Helm客户端****

下载helm-v2.8.0-windows-amd64.tar.gz，并将其解压缩到操作系统的本地特定目录地址下，并将helm.exe所在的地址添加至Windows的环境变量的Path中。

在本书中，helm.exe放置的目录地址为：d:/helm。

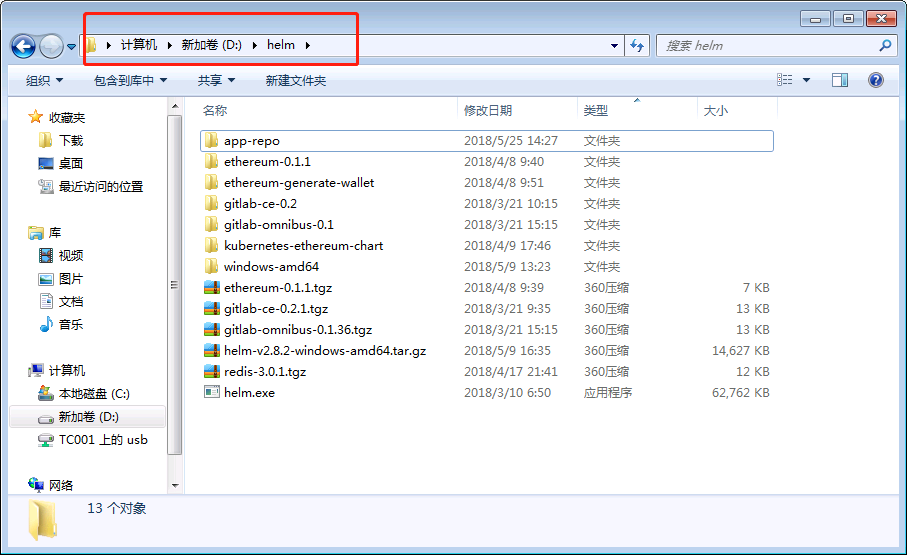


图7-10 helm客户端所在的目录

在Windows的环境变量，将其添加到Path变量的值中，如下图所示：

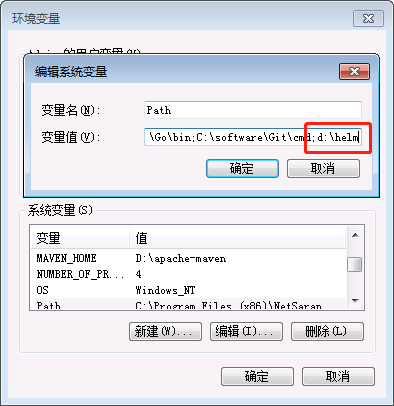


图7-11 设置helm的环境变量

#### ****安装Tiller服务器****

##### ****使用默认配置文件安装****

通过在Helm客户端中执行如下的命令来安装Tiller服务器：

$ helm init

**注意：**

* helm init命令通过$HOME/.kube/config配置文件确定在哪个Kubernetes集群中按照Tiller服务器。
* 此命令会将Tiller服务器安装Kubernetes的kube-system命名空间中。

##### ****使用Service Account安装****

1）创建一个名为tiller的Service Account

$ kubectl create serviceaccount tiller --namespace kube-system

2）授予名为tiller的Service Account集群管理员角色cluster-admin：

将tiller绑定至集群管理员角色的的yaml文件如下所示：

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRoleBinding

metadata:

name: tiller

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: tiller

namespace: kube-system

通过执行kubectl create -f将授予tiller集群管理员角色：

$ kubectl create -f rbac-config.yaml

3）安装Tiller服务器

$ helm init --service-account tiller

##### ****验证安装****

在安装完成后，可以通过执行如下命令来检查是安装成功：

$ helm version

如果正确显示Helm客户端和Tiller服务器的版本，这表示安装成功。

或者通过执行kubectl的如下命令来查看是否已正常按照Tiller服务器：

$ kubectl get pods -n kube-system

### ****使用Helm****

Helm所使用的包格式为chart，chart是描述Kubernetes资源的文件集合。通过chart能够部署类似于memcache等简单的应用，或者能够部署类似于Http服务器、数据库等复杂的应用。此文主要就chart目录、chart.yaml文件和values.yaml文件进行介绍。

#### ****Chart目录结构****

在使用Helm之前，先以wordpress为例来看一下一个Chart都包含哪些内容：

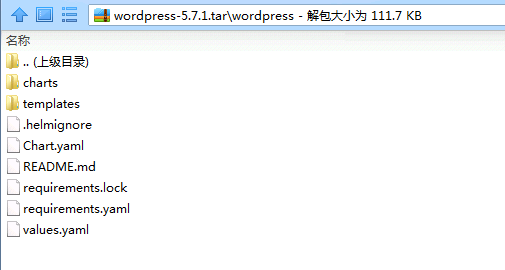


图7-12 典型的chart目录结构

**wordpress**/

* + - Chart.yaml 必需：包含关于chart信息的YAML文件
    - LICENSE 可选： chart的license描述文件
    - README.md 可选：可读的说明文件
    - requirements.yaml 可选：列示chart依赖的YAML文件
    - **values.yaml 必需：chart默认的配置值文件**
    - charts/ 可选：包含chart所有依赖的目录
    - templates/ 可选：包含模板文件的目录
    - templates/NOTES.txt 可选：部署后的使用说明

#### ****chart.yaml文件介绍****

对于chart来说，chart.yaml文件是必需的文件，此文件的内容如下：

* **name**: 必需，chart的名称
* **version**: 必需，SemVer 2的版本
* **description**: 可选，此项目的一句话描述
* **keywords**: – 可选，此项目的关键词列表
* **home**: 可选，此项目主页面的URL
* **sources**: – 可选，此项目的源代码URL列表
* **maintainers**: # 可选 – **name**: 维护者的名字
* **email**: 维护者的email地址
* **url**: 维护者的URL
* **engine**: gotpl # 可选，模板引擎的名称
* **icon**: 可选，被使用的SVG或者PNG格式图标的URL
* **appVersion**: 可选，应用的版本
* **deprecated**: 可选，标识吃chart是否将要被废弃
* **tillerVersion**: chart所要求的Tiller服务器的版本

#### ****values.yaml文件介绍****

chart的配置文件values.yaml所包含的内容如下例子所示：

imageRegistry: "quay.io/deis"

dockerTag: "latest"

pullPolicy: "Always" storage: "s3"

在chart中可以包含一个默认的values.yaml文件，Helm安装命令也允许通过–values参数指定应该YAML文件：

$ helm install --values=myvals.yaml wordpress

通过给定–values参数的方式，helm会将此参数给定的YAML文件的内容与默认values.yaml文件的内容进行合并，例如在myvals.yaml文件的内容是：

storage: "gcs"

合并后的文件内容将如下所示：

imageRegistry: "quay.io/deis"

dockerTag: "latest"

pullPolicy: "Always" storage: "gcs"

**注意**：默认的值文件必需被命名为values.yaml。

### ****Chart应用仓库****

Kubeapps Hub（[https://hub.kubeapps.com](https://hub.kubeapps.com/)）作为公共的chart应用仓库，目前在上面已经以chart的格式提供Nginx、Jenkins、Redis等常用应用。在此仓库中，可以搜索符合自己需要的Kubernetes应用，或者发布自己以chart格式构建的Kubernetes应用。

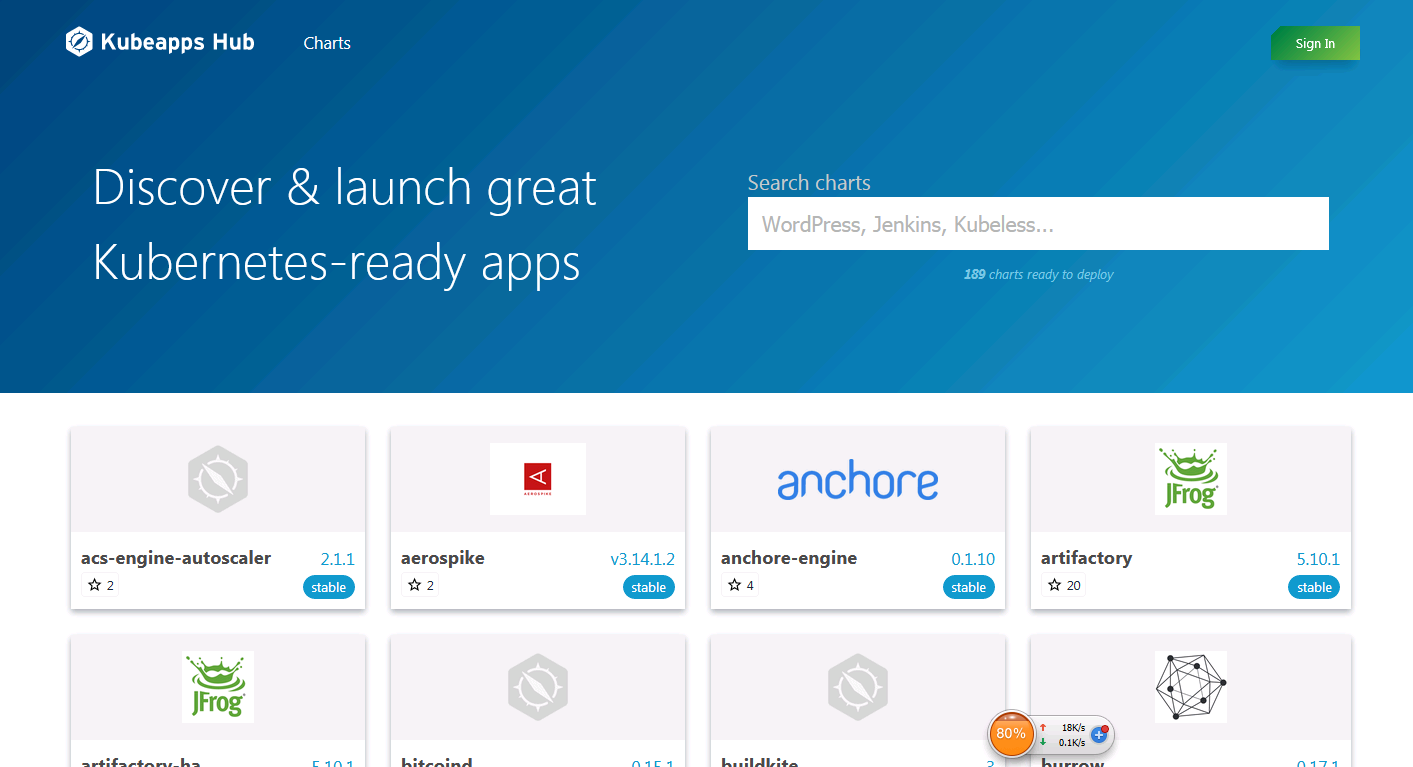


图7-13 kubeapps hub界面

# 第二章 生态环境

# 镜像仓库(2019-04-16 Y)

在Kubernetes的集群中运行的是容器，而容器的基础或者是来源是镜像。Kubernetes本身并没有对镜像进行管理，因此需要通过外部的镜像仓库对镜像对管理。根据易用性等方面的考虑，本文财Nexus作为镜像的私有仓库。

## 安装Nexus

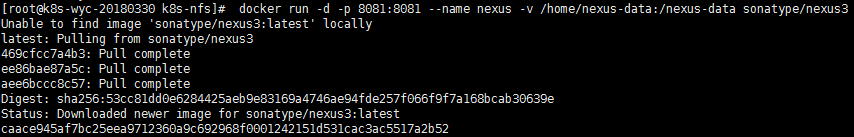
Nexus是Sonatype提供的仓库管理平台，Nuexus Repository OSS3能够支持Maven、npm、Docker、YUM、Helm等格式数据的存储和发布；并且能够与Jekins、SonaQube和Eclipse等工具进行集成。Nexus支持作为宿主和代理存储库的Docker存储库，可以直接将这些存储库暴露给客户端工具；也可以以存储库组的方式暴露给客户端工具，存储库组是合并了多个存储库的内容的存储库，能够通过一个URL将多个存储库暴露给客户端工具，从而便于用户的使用。通过nexus自建能够有效减少访问获取镜像的时间和对带宽使用，并能够通过自有的镜像仓库共享企业自己的镜像。在本文中，采用Docker模式安装部署Nexus。

首先，通过mkdir创建一个目录，用于为Nexus提供存储的空间。

$ mkdir {path}/nexus-data && chown -R 200 {path}/nexus-data

接着，就可以通过sonatype/nexus3镜像启动nexus3的容器化应用了。通过如下命令启动的nexus将对外暴露8081端口，并容器的持久化数据通过会存储在上述创建的空间中。在容器运行后，用户将可以通过http://{host\_ip}:8081访问nexus应用，其中{host\_ip}为容器所部署的宿主机的IP地址。

$ docker run -d -p 8081:8081 --name nexus -v {path}/nexus-data:/nexus-data sonatype/nexus3



## 构建私有镜像仓库

在nexus部署成功后，在浏览器中通过http://{host\_ip}:8081地址访问nexus应用。

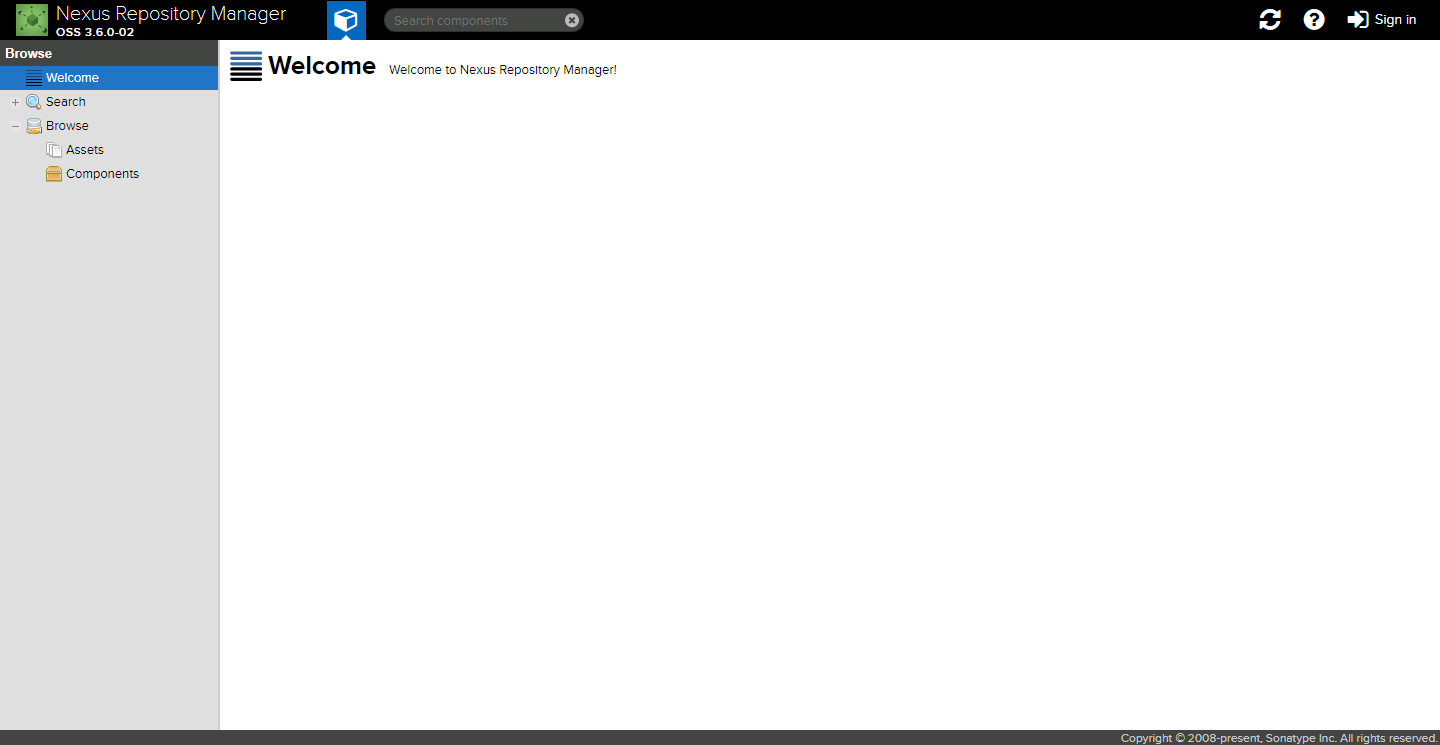


图8-1 nexus界面

1）通过管理员帐户登录nexus，并进入创建docker镜像仓库的主页：

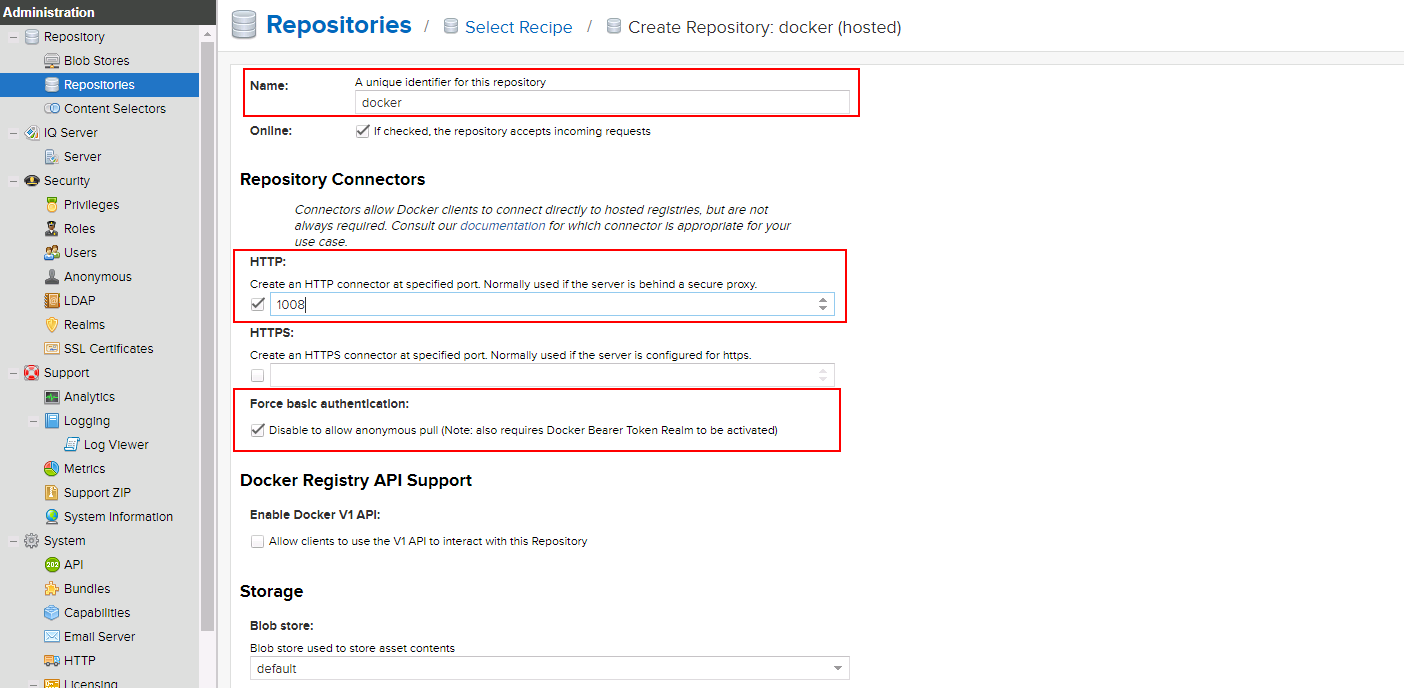


图8-2 创建docker镜像仓库界面

2）在创建镜像仓库的页面中，设置镜像仓库的相关信息，包括名称、HTTP端口、是否允许匿名拉取镜像等信息。这里需要注意的是，此处的HTTP端口(此处的值为1008)很重要，后续拉取和推送进行是使用此端口进行的，而不是nexus本身对外暴露的端口。另外，如果允许设置通过匿名的方式拉取镜像。

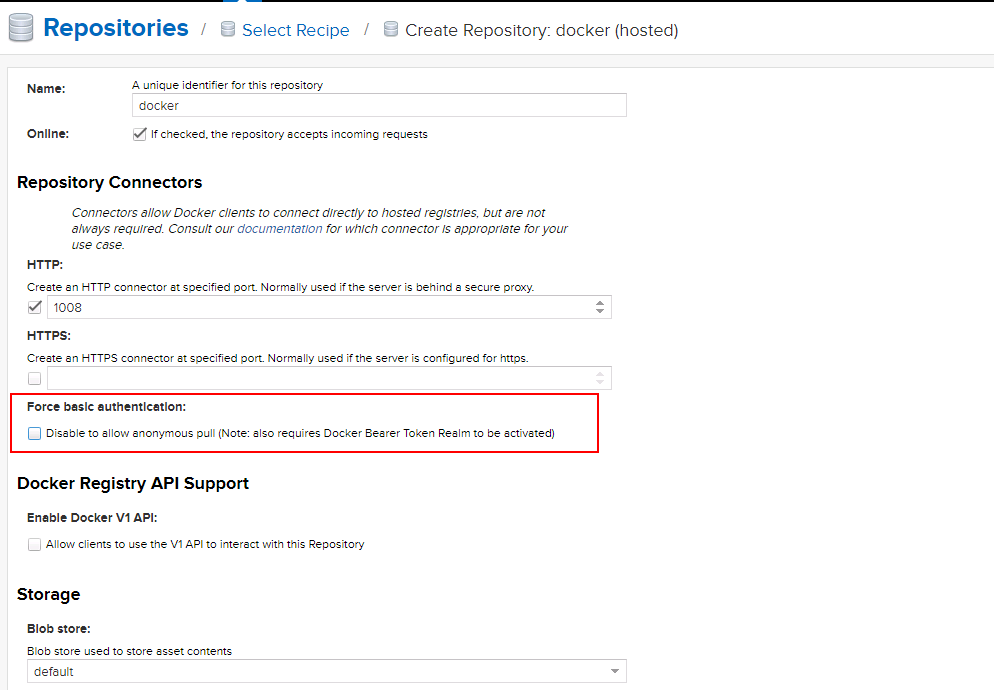


图8-3 设置镜像仓库信息

这需要在Realms主页激活Docker Bearer Token Realm，如下图所示：

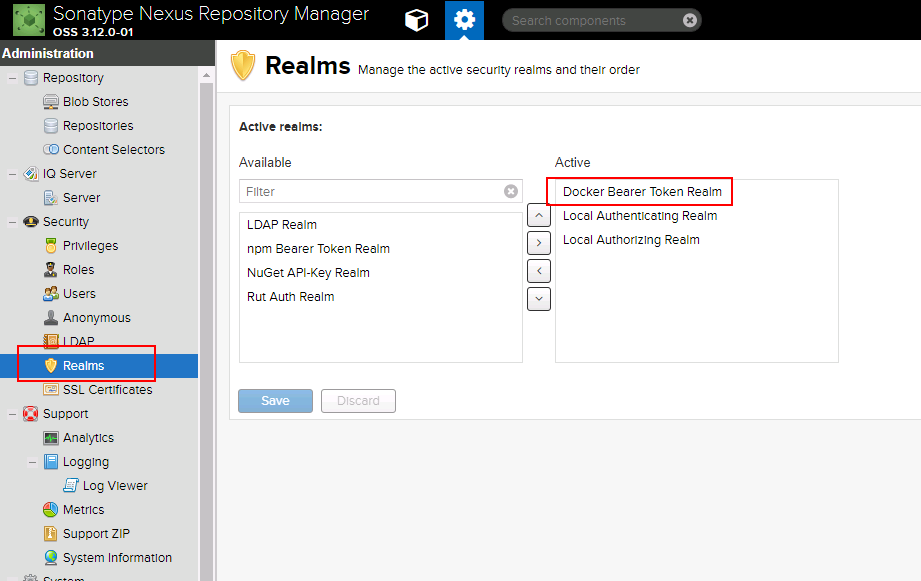


图8-4 设置realm

并对匿名方式进行设置，允许通过匿名方式访问服务器，如下图进行设置：

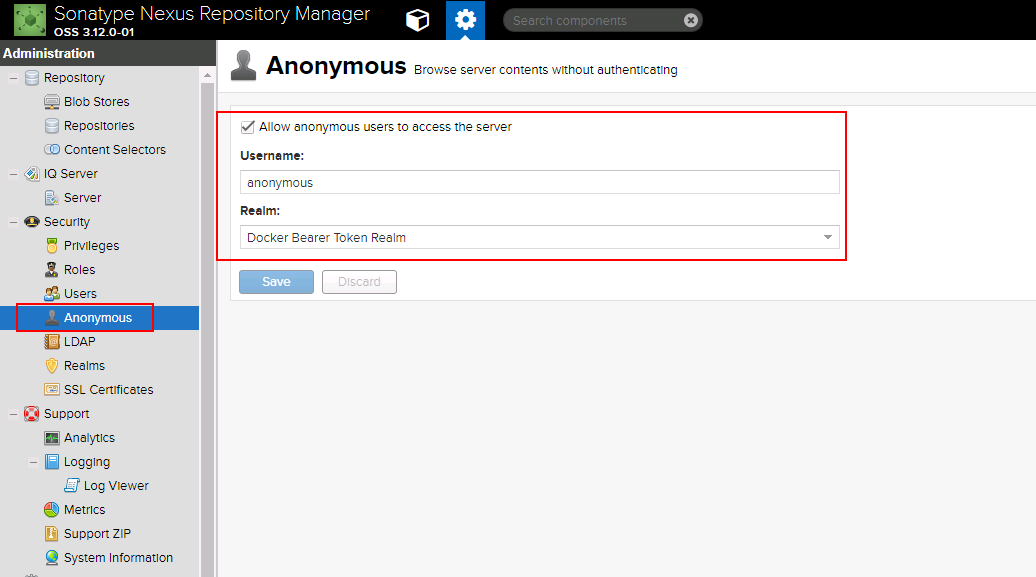


图8-5 允许匿名访问

3）在客户端的/etc/docker/daemon.json文件中添加下面的内容：

在完成私有镜像仓库的设置后，由于使用的是HTTP协议，因此需要在客户端对docker进行配置。通过编译工具打开daemon.json:

$ vi /etc/docker/daemon.json

在文件中添加如下的内容，告诉客户端私有镜像仓库是一个安全的仓库：

{

"insecure-registries":["<nexus-hostname>:<repository-port>"]

}

## 基本操作

在镜像仓库构建完成，就可以在镜像仓库中对镜像进行管理。在真正进行操作之前，需要先登录镜像仓库；登录通过后，就可以将进行推送到进行仓库进行管理；后续根据业务需要，从镜像仓库中拉取需要的镜像。

### 登录认证

在通过nexus完成私有镜像仓库的构建后，首先需要进行登录认证才能进行后续的操作，私有镜像仓库登录认证的语法和格式：docker login <nexus-hostname>:<repository-port>。假设上述的nexus部署在IP地址为10.8.32.148主机上，私有镜像的端口为1008，则通过执行如下的命令登录私有镜像仓库：

$ docker login 10.8.32.148:1008

https://www.kubernetes.org.cn/img/2018/05/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20180528165219.png

登录时，需要提供用户名和密码。认证的信息会被保存在~/.docker/config.json文件，在后续与私有镜像仓库交互时就可以被重用，而不需要每次都进行登录认证。

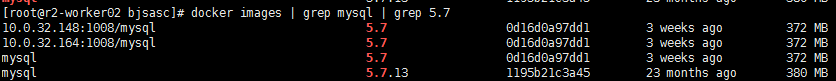
### 推送镜像

要共享一个镜像，可以通过将其发布到托管存储库，然后其它人员就可以通过存储库获取自己需要的镜像。在将镜像推送到存储库之前，需要对镜像进行标记。当标记图像时，可以使用镜像标识符（imageId）或者镜像名称（imageName）。标识镜像的语法和格式：

$ docker tag <imageId or imageName> <nexus-hostname>:<repository-port>/<image>:<tag>

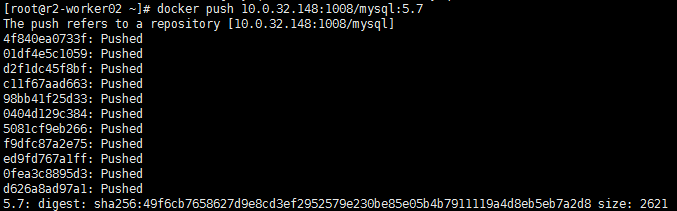
假设这里将mysql:5.7镜像标识为私有镜像仓库(10.8.32.148:1008)中的镜像，标识的执行命令如下：

$ docker tag mysql:5.7 10.8.32.148:1008/mysql:5.7



一旦镜像标识完成后，就可以通过的docker push命令将镜像推送到私有仓库中。推送镜像到私有镜像仓库的语法和格式为*docker push <nexus-hostname>:<repository-port>/<image>:<tag>，*通过下面的命令，将上述打完标签的镜像上传至私有镜像仓库：

$ docker push 10.8.32.148:1008/mysql:5.7



### 拉取镜像

Kunbernetes将会根据需要从私有镜像仓库中拉取镜像，在客户端可以通过手动拉取镜像，拉取的语法和格式：docker pull <nexus-hostname>:<repository-port>/<image>:<tag>。假设从本文构建的私有镜像仓库中拉取mysql:5.7，执行命令如下所示：

$ docker pull 10.8.32.148:1008/mysql:5.7

### 从私有镜像拉取镜像

在镜像仓库构建完成后，Kubernetes就可以从进行镜像仓库中拉取镜像，并在集群中对镜像进行实例化生成容器，为用户提供相应的应用服务。

#### 生成密钥

在使用私有镜像拉取镜像时，需要为私有镜像仓库创建一个镜像仓库的密钥，并在创建容器中进行引用。创建镜像仓库的语法和格式：

$ kubectl create secret docker–registry <**regsecret-name>** —docker–server=<your–registry–server> —docker–username=<your–name> —docker–password=<your–pword> —docker–email=<your–email>。

* <regsecret-name>：所创建私有镜像仓库密钥的名称；
* <your-registry-server>：镜像仓库服务器地址；
* <your-name>：登录镜像仓库的用户名；
* <your-pword>：登录镜像仓库的密码；
* <your-email>：用户的邮箱地址。

假设登录私有镜像仓库的用户命名为admin、密码为admin、邮箱地址为admin@aliyun.com。则可以通过执行下面的命令创建私有镜像仓库的密钥：

$ kubectl create secret docker-registry myregsecret --docker-server=10.8.32.148:1008 \

--docker-username=admin --docker-password=admin --docker-email=admin@aliyun.com

#### 定义拉取镜像的部署

在这里定义了一个名为nginx的YAML部署示例文件，此文件从私有镜像仓库拉取nginx，并使用imagePullSecrets字段来指定拉取镜像所使用的密钥：

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: nginx

spec:

  replicas: 3

  selector:

    matchLabels:

      app: nginx

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: nginx

    spec:

# 指定从私有镜像仓库拉取镜像的密钥

imagePullSecrets:

- name: myregsecret

      containers:

      # 所要拉取的镜像

      - image: 10.8.32.148：1008/nginx:1.7.9

        name: nginx

        imagePullPolicy: IfNotPresent

        ports:

        - containerPort: 80

          name: nginx80

        volumeMounts:

        - mountPath: /usr/share/nginx/html

          name: nginx-data

        - mountPath: /etc/nginx

          name: nginx-conf

      volumes:

      - name: nginx-data

        nfs:

          path: /k8s-nfs/nginx

          server: 192.168.8.150

      - name: nginx-conf

        nfs:

          path: /k8s-nfs/nginx/conf

          server: 192.168.8.150

通过执行kubectl create -f命令，在Kubernetes中基于所定义的YAML创建部署：

$ kubectl create -f {path}/nginx-deployment.yaml

 执行上述命令后，Kubernetes将会从私有镜像仓库拉取nginx:1.7.9镜像，并基于此镜像启动容器。

# 网络(2019-05-05 Y)

Kubernetes会为运行在其上的应用构建一个独立的网络，通过这个内部的网络，容器化的应用间就可以进行通讯和调用。本章节首先就Docker的网络模式进行介绍，包括Docker支持的网络模式、网络的构建过程，以及外部如何访问容器中的应用。在了解Docker网络模式的基础上，基于flanel对Kubernetes的网络模式进行讲解。

## **Docker网络模式**

在讨论Kubernetes网络之前，让我们先来看一下Docker网络。Docker采用插件化的网络模式，默认提供bridge、host、none、overlay、maclan和Network plugins这几种网络模式，运行容器时可以通过–network参数设置具体使用那一种模式。

* **bridge**：这是Docker默认的网络驱动，此模式会为每一个容器分配Network Namespace和设置IP等，并将容器连接到一个虚拟网桥上。如果未指定网络驱动，这默认使用此驱动。
* **host**：此网络驱动直接使用宿主机的网络。
* **none**：此驱动不构造网络环境。采用了none 网络驱动，那么就只能使用loopback网络设备，容器只能使用127.0.0.1的本机网络。
* **overlay**：此网络驱动可以使多个Docker daemons连接在一起，并能够使swarm服务之间进行通讯。也可以使用overlay网络进行swarm服务和容器之间、容器之间进行通讯，
* **macvlan**：此网络允许为容器指定一个MAC地址，允许容器作为网络中的物理设备，这样Docker daemon就可以通过MAC地址进行访问的路由。对于希望直接连接网络网络的遗留应用，这种网络驱动有时可能是最好的选择。
* **Network plugins**：可以安装和使用第三方的网络插件。可以在Docker Store或第三方供应商处获取这些插件**。**

在默认情况，Docker使用bridge网络模式，bridge网络驱动的示意图如下，此文以bridge模式对Docker的网络进行说明。

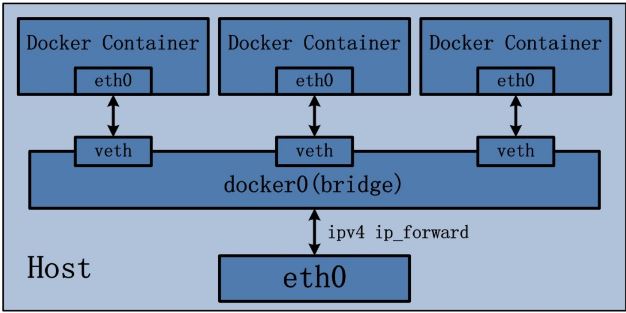


图9-1 docker的bridge模式网络模型

### ****bridge网络的构建过程****

bridge网络的构建过程主要分为两个步骤，安装docker时，会在主机上创建一个名称为docker0的虚拟网桥；运行容器时，会为容器创建虚拟网卡veth pair设备。

1）安装Docker时，创建一个名为docke0的虚拟网桥，虚拟网桥使用“10.0.0.0 -10.255.255.255 “、”172.16.0.0-172.31.255.255″和“192.168.0.0——192.168.255.255”这三个私有网络的地址范围。

通过 ifconfig 命令可以查看docker0网桥的信息：

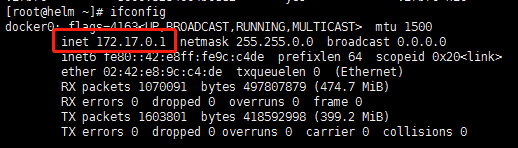


图9-2 docker0网桥信息

通过 *docker network inspect bridge* 可以查看网桥的子网网络范围和网关：

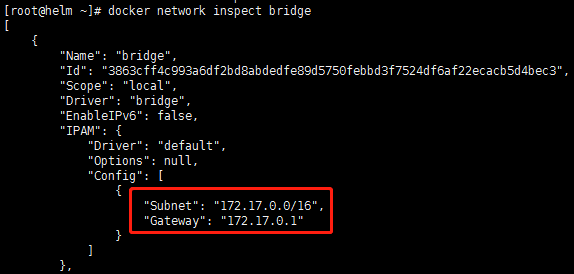
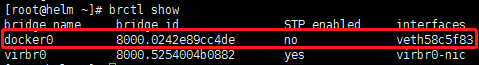


图9-3 docker0网桥信息

2）运行容器时，在宿主机上创建虚拟网卡veth pair设备。veth pair设备是成对出现的，从而组成一个数据通道，数据从一个设备进入，就会从另一个设备出来。将veth pair设备的一端放在新创建的容器中，命名为**eth0**；另一端放在宿主机的docker0中，以**veth**为前缀的名字命名。通过 brctl show 命令查看放在docker0中的veth pair设备：



### ****外部访问****

bridge的docker0是虚拟出来的网桥，因此无法被外部的网络访问。因此需要在运行容器时通过-p和-P参数对将容器的端口映射到宿主机的端口。实际上Docker是采用 NAT的方式，将容器内部的服务监听端口与宿主机的某一个端口port 进行绑定，使得宿主机外部可以将网络报文发送至容器。

1）通过-P参数，将容器的端口映射到宿主机的随机端口：

$ docker run -P {images}

2）通过-p参数，将容器的端口映射到宿主机的制定端口：

$ docker run -p {hostPort}:{containerPort} {images}

## **Kubernetes网络模式**

Kubernetes与Docker网络有些不同，Kubernetes网络需要解决下面的4个问题：

* **集群内：**
  + 容器与容器之间的通信
  + Pod和Pod之间的通信
  + Pod和服务之间的通信
* **集群外：**
  + 外部应用与服务之间的通信

因此，Kubernetes假设Pod之间能够进行通讯，这些Pod可能部署在不同的宿主机上。每一个Pod都拥有自己的IP地址，因此能够将Pod看作为物理主机或者虚拟机，从而能实现端口设置、命名、服务发现、负载均衡、应用配置和迁移。为了满足上述需求，则需要通过集群网络来实现。

在本文主要分析容器与容器之间，以及Pod和Pod之间的通信；Pod和服务之间，以及外部应用与服务之间的通信请参考《Kubernetes-核心资源之Service》和《Kubernetes-核心资源之Ingress》。

### ****同一个Pod中容器之间的通信****

这种场景对于Kubernetes来说没有任何问题，根据Kubernetes的架构设计。Kubernetes创建Pod时，首先会创建一个pause容器，为Pod指派一个唯一的IP地址。然后，以pause的网络命名空间为基础，创建同一个Pod内的其它容器（–net=container:xxx）。因此，同一个Pod内的所有容器就会共享同一个网络命名空间，在同一个Pod之间的容器可以直接使用localhost进行通信。

### ****不同Pod中容器之间的通信****

对于此场景，情况现对比较复杂一些，这就需要解决Pod间的通信问题。在Kubernetes通过flannel、calic、canal等网络插件解决Pod间的通信问题。本文以flannel为例说明在Kubernetes中网络模型，flannel是kubernetes默认提供网络插件。Flannel是由CoreOs团队开发社交的网络工具，CoreOS团队采用L3 Overlay模式设计flannel，规定宿主机下各个Pod属于同一个子网，不同宿主机下的Pod属于不同的子网。

flannel会在每一个宿主机上运行名为flanneld代理，其负责为宿主机预先分配一个子网，并为Pod分配IP地址。Flannel使用Kubernetes或etcd来存储网络配置、分配的子网和主机公共IP等信息。数据包则通过VXLAN、UDP或host-gw这些类型的后端机制进行转发。

### ****数据传递过程****

**在源容器宿主机中的数据传递过程：**

**1）源容器向目标容器发送数据，数据首先发送给docker0网桥**

在源容器内容查看路由信息：

$ kubectl exec -it -p {Podid} -c {ContainerId} -- ip route

**2）docker0网桥接受到数据后，将其转交给flannel.1虚拟网卡处理**

docker0收到数据包后，docker0的内核栈处理程序会读取这个数据包的目标地址，根据目标地址将数据包发送给下一个路由节点：

查看源容器所在Node的路由信息：

$ ip route

**3）flannel.1接受到数据后，对数据进行封装，并发给宿主机的eth0**

flannel.1收到数据后，flannelid会将数据包封装成二层以太包。

**Ethernet Header的信息：**

* From:{源容器flannel.1虚拟网卡的MAC地址}
* To:{目录容器flannel.1虚拟网卡的MAC地址}

**4）对在flannel路由节点封装后的数据，进行再封装后，转发给目标容器Node的eth0**

由于目前的数据包只是vxlan tunnel上的数据包，因此还不能在物理网络上进行传输。因此，需要将上述数据包再次进行封装，才能源容器节点传输到目标容器节点，这项工作在由linux内核来完成。

**Ethernet Header的信息：**

* From:{源容器Node节点网卡的MAC地址}
* To:{目录容器Node节点网卡的MAC地址}

**IP Header的信息：**

* From:{源容器Node节点网卡的IP地址}
* To:{目录容器Node节点网卡的IP地址}

通过此次封装，就可以通过物理网络发送数据包。

**在目标容器宿主机中的数据传递过程：**

**5）目标容器宿主机的eth0接收到数据后，对数据包进行拆封，并转发给flannel.1虚拟网卡；**

**6）flannel.1 虚拟网卡接受到数据，将数据发送给docker0网桥；**

**7）最后，数据到达目标容器，完成容器之间的数据通信。**

# 文件存储(2019-04-17 Y)

由于容器本身不提供进行数据持久化的能力，因此需要使用外部的文件存储系统进行数据持久化，同时要求文件存储系统能够支持分布式的文件存储和访问。根据上述要求，在本文中采用NFS作为文件存储系统。

## **NFS介绍**

NFS是Network File System的简写，即网络文件系统，NFS是FreeBSD支持的文件系统中的一种。NFS基于RPC(Remote Procedure Call)远程过程调用实现，允许系统在网络上与其它用户和程序共享目录和文件。通过使用NFS，用户和程序就可以像访问本地文件一样访问远端系统上的文件。NFS是一个非常稳定的和可移植的网络文件系统，具备可扩展和高性能等特性，能够满足企业级应用的标准。

### ****nfs原理****

NFS 使用RPC(Remote Procedure Call)的机制进行实现，通过RPC客户端可以调用服务端的函数。同时，由于有 VFS 的存在，客户端可以像使用其它普通文件系统一样使用 NFS 文件系统。通过操作系统的内核，将对 NFS 文件系统的调用请求通过 TCP/IP 发送至服务端的 NFS 服务。NFS服务器执行相关的操作，并将操作结果返回给客户端。

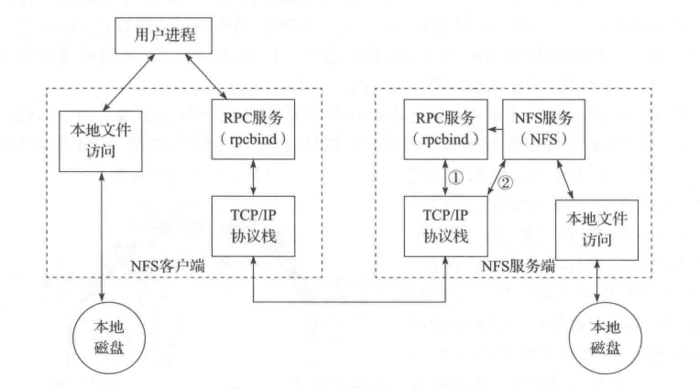


图10-1 nfs原理

NFS服务主要进程包括：

* rpc.nfsd：最主要的NFS进程，管理客户端是否可登录；
* rpc.mountd：挂载和卸载NFS文件系统，包括权限管理；
* rpc.lockd：非必要，管理文件锁，避免同时写出错；
* rpc.statd：非必要，检查文件一致性，可修复文件。

nfs的关键工具包括：

* 主要配置文件：/etc/exports；
* NFS文件系统维护命令：/usr/bin/exportfs；
* 共享资源的日志文件： /var/lib/nfs/\*tab；
* 客户端查询共享资源命令： /usr/sbin/showmount；
* 端口配置： /etc/sysconfig/nfs。

### ****共享配置****

在NFS服务器端的主要配置文件为/etc/exports时，通过此配置文件可以设置共享文件目录。每条配置记录由NFS共享目录、NFS客户端地址和参数这3部分组成，格式如下：

[NFS共享目录] [NFS客户端地址1(参数1,参数2,参数3……)] [客户端地址2(参数1,参数2,参数3……)]

* NFS共享目录：服务器上共享出去的文件目录；
* NFS客户端地址：允许其访问的NFS服务器的客户端地址，可以是客户端IP地址，也可以是一个网段(192.168.64.0/24)；
* 访问参数：括号中逗号分隔项，主要是一些权限选项。

1. **访问权限参数**

**表10-1 访问参数**

|  |  |  |
| --- | --- | --- |
| **序号** | **选项** | **描述** |
| 1 | ro | 客户端对于共享文件目录的访问权限为只读，这是默认设置。 |
| 2 | rw | 客户端对于共享文件目录具有读写权限。 |

1. **用户映射参数**

表10-2 用户映射参数

|  |  |  |
| --- | --- | --- |
| **序号** | **选项** | **描述** |
| 1 | root\_squash | 客户端使用root账户访问共享目录时，服务器端会将访问用户映射为服务器本地的匿名账号。 |
| 2 | no\_root\_squash | 客户端使用root账户访问共享目录时，服务端也使用root用户对共享目录进行操作。 |
| 3 | all\_squash | 将所有客户端用户请求映射到匿名用户或用户组（nfsnobody）。 |
| 4 | no\_all\_squash | 与上相反（默认设置）。 |
| 5 | anonuid=xxx | 将远程访问的所有用户都映射为匿名用户，并指定该用户为本地用户（UID=xxx）。 |
| 6 | anongid=xxx | 将远程访问的所有用户组都映射为匿名用户组账户，并指定该匿名用户组账户为本地用户组账户（GID=xxx）。 |

1. **其它配置参数**

表10-3 其它配置参数

|  |  |  |
| --- | --- | --- |
| 序号 | 选项 | 描述 |
| 1 | sync | 同步写操作，数据写入存储设备后返回成功信息。（默认设置） |
| 2 | async | 异步写操作，数据在未完全写入存储设备前就返回成功信息，实际还在内存。 |
| 3 | wdelay | 延迟写入选项，将多个写操请求合并后写入硬盘，减少I/O次数，NFS非正常关闭数据可能丢失（默认设置）。 |
| 4 | no\_wdelay | 与上相反，不与async同时生效，如果NFS服务器主要收到小且不相关的请求，该选项实际会降低性能。 |
| 5 | subtree | 若输出目录是一个子目录，则nfs服务器将检查其父目录的权限(默认设置)； |
| 6 | no\_subtree | 即使输出目录是一个子目录，nfs服务器也不检查其父目录的权限，这样可以提高效率 |
| 7 | secure | 限制客户端只能从小于1024的tcp/ip端口连接nfs服务器（默认设置）。 |
| 8 | insecure | 允许客户端从大于1024的tcp/ip端口连接服务器。 |

## **nfs服务端配置**

nfs作为网络文件存储系统时，需要进行进行如下的安装和配置工作。首先，需要安装nfs和rpcbind服务；接着，需要创建使用共享目录的用户；然后，需要对共享目录进行配置，这是其中相对重要和复杂的一个步骤；最后，需要启动rpcbind和nfs服务，以供应用使用。

### ****安装nfs服务****

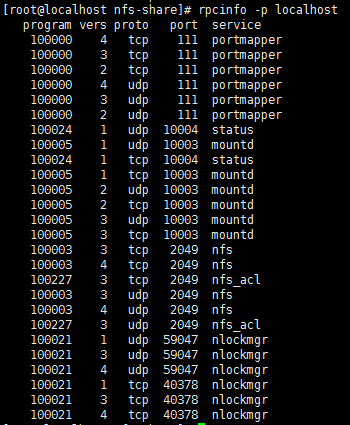
首先，需要安装nfs服务，nfs服务由nfs服务和rpcbind服务所组成。

1）通过yum目录安装nfs服务和rpcbind服务：

$ yum -y install nfs-utils rpcbind

2）检查nfs服务是否正常安装：

$ rpcinfo -p localhost



### ****创建用户和共享目录****

在nfs服务安装完成，需要为NFS服务其添加用户，并创建共享目录，以及设置共享目录的访问权限。在这里创建nfs用户，在根目录下创建nfs-share文件夹，同时授予所有人对此文件夹具有可执行的操作权限。

$ useradd -u nfs

$ mkdir -p /nfs-share

$ chmod a+w /nfs-share

### ****配置共享目录****

在创建好用户和共享目录后，需要进行共享配置才能被客户端所访问。下面是为客户端配置共享目录的示例，在

$ echo "/nfs-share \*(rw,async,no\_root\_squash)" >> /etc/exports

通过执行如下命令是配置生效：

$exportfs -r

### ****启动服务****

在共享目录配置完成后，就可以启动nfs服务了。需要注意的是，首先要启动rpcbind服务，然后在启动nfs服务。

1）由于必须先启动rpcbind服务，再启动nfs服务，这样才能让nfs服务在rpcbind服务上注册成功：

$ systemctl start rpcbind

2）启动nfs服务：

$ systemctl start nfs-server

3）设置rpcbind和nfs-server开机启动：

$ systemctl enable rpcbind

$ systemctl enable nfs-server

### ****检查nfs服务是否正常启动****

在nfs配置完成后，可以通过执行下面的命令进行验证。

$ showmount -e localhost



## **nfs作为volume**

nfs可以直接作为存储卷使用，下面是一个redis部署的YAML配置文件。在此示例中，redis在容器中的持久化数据保存在/data目录下；存储卷使用nfs，nfs的服务地址为：192.168.8.150，存储路径为：/k8s-nfs/redis/data。容器通过volumeMounts.name的值确定所使用的存储卷。

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: redis

spec:

  selector:

    matchLabels:

      app: redis

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: redis

    spec:

      containers:

      # 应用的镜像

      - image: redis

        name: redis

        imagePullPolicy: IfNotPresent

        # 应用的内部端口

        ports:

        - containerPort: 6379

          name: redis6379

        env:

        - name: ALLOW\_EMPTY\_PASSWORD

          value: "yes"

        - name: REDIS\_PASSWORD

          value: "redis"

# 持久化挂接位置，在docker中

        volumeMounts:

        - name: redis-persistent-storage

          mountPath: /data

      volumes:

      # nfs服务器信息

      - name: redis-persistent-storage

        nfs:

          path: /k8s-nfs/redis/data

          server: 192.168.8.150

## **nfs作为PersistentVolum**

在Kubernetes当前版本的中，可以创建类型为nfs的持久化存储卷，用于为PersistentVolumClaim提供存储卷。在下面的PersistenVolume YAML配置文件中，定义了一个名为nfs-pv的持久化存储卷，此存储卷提供了5G的存储空间，只能由一个PersistentVolumClaim进行可读可写操作。此持久化存储卷使用的nfs服务器地址为192.168.5.150，存储的路径为/tmp。

apiVersion: v1

kind: PersistentVolume

metadata:

name: nfs-pv

spec:

  capacity:

    storage: 5Gi

  volumeMode: Filesystem

  accessModes:

- ReadWriteOnce

  persistentVolumeReclaimPolicy: Recycle

storageClassName: slow

mountOptions:

- hard

- nfsvers=4.1

# 此持久化存储卷使用nfs插件

nfs:

# nfs共享目录为/tmp

path: /tmp

# nfs服务器的地址

server: 192.168.5.150

通过执行如下的命令可以创建上述持久化存储卷：

$ kubectl create -f {path}/nfs-pv.yaml

存储卷创建成功后将处于可用状态，等待PersistentVolumClaim使用。PersistentVolumClaim会通过访问模式和存储空间自动选择合适存储卷，并与其进行绑定。

## **nfs作为动态存储提供**

### ****部署nfs-provisioner****

为nfs-provisioner实例选择存储状态和数据的存储卷，并将存储卷挂接到容器的/export 命令。

...

 volumeMounts:

    - name: export-volume

      mountPath: /export

volumes:

  - name: export-volume

    hostPath:

      path: /tmp/nfs-provisioner

...

为StorageClass选择一个供应者名称，并在deploy/kubernetes/deployment.yaml进行设置。

args:

  - "-provisioner=example.com/nfs"

...

完整的deployment.yaml文件内容如下：

kind: Service

apiVersion: v1

metadata:

  name: nfs-provisioner

  labels:

    app: nfs-provisioner

spec:

  ports:

    - name: nfs

      port: 2049

    - name: mountd

      port: 20048

    - name: rpcbind

      port: 111

    - name: rpcbind-udp

      port: 111

      protocol: UDP

  selector:

    app: nfs-provisioner

---

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  name: nfs-provisioner

spec:

  replicas: 1

  strategy:

    type: Recreate

  template:

    metadata:

      labels:

        app: nfs-provisioner

    spec:

      containers:

        - name: nfs-provisioner

          image: quay.io/kubernetes\_incubator/nfs-provisioner:v1.0.8

          ports:

            - name: nfs

              containerPort: 2049

            - name: mountd

              containerPort: 20048

            - name: rpcbind

              containerPort: 111

            - name: rpcbind-udp

              containerPort: 111

              protocol: UDP

          securityContext:

            capabilities:

              add:

                - DAC\_READ\_SEARCH

                - SYS\_RESOURCE

          args:

# 定义提供者的名称，存储类通过此名称指定提供者

            - "-provisioner=nfs-provisioner"

          env:

            - name: POD\_IP

              valueFrom:

                fieldRef:

                  fieldPath: status.podIP

            - name: SERVICE\_NAME

              value: nfs-provisioner

            - name: POD\_NAMESPACE

              valueFrom:

                fieldRef:

                  fieldPath: metadata.namespace

          imagePullPolicy: "IfNotPresent"

          volumeMounts:

            - name: export-volume

              mountPath: /export

      volumes:

        - name: export-volume

          hostPath:

            path: /srv

在设置好deploy/kubernetes/deployment.yaml文件后，通过kubectl create命令在Kubernetes集群中部署nfs-provisioner。

$ kubectl create -f {path}/deployment.yaml

### ****创建StorageClass****

下面是example-nfs的StorageClass配置文件，此配置文件定义了一个名称为nfs-storageclass的存储类，此存储类的提供者为nfs-provisioner。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: nfs-storageclass

provisioner: nfs-provisioner

通过kubectl create -f命令使用上面的配置文件创建：

$ kubectl create -f deploy/kubernetes/class.yaml

storageclass “example-nfs” created

在存储类被正确创建后，就可以创建PersistenetVolumeClaim来请求StorageClass，而StorageClass将会为PersistenetVolumeClaim自动创建一个可用PersistentVolume。

### 创建PersistenetVolumeClaim

PersistenetVolumeClaim是对PersistenetVolume的声明，即PersistenetVolume为存储的提供者，而PersistenetVolumeClaim为存储的消费者。下面是PersistentVolumeClaim的YAML配置文件，此配置文件通过metadata.annotations[].volume.beta.kubernetes.io/storage-class字段指定所使用的存储储类。

在此配置文件中，使用nfs-storageclass存储类为PersistenetVolumeClaim创建PersistenetVolume，所要求的PersistenetVolume存储空间大小为1Mi，可以被多个容器进行读取和写入操作。

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nfs-pvc

annotations:

volume.beta.kubernetes.io/storage-class: "nfs-storageclass"

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 1Mi

通过kubectl create命令创建上述的持久化存储卷声明：

$ kubectl create -f {path}/claim.yaml

### ****创建使用PersistenVolumeClaim的部署****

在这里定义名为busybox-deployment的部署YAML配置文件，使用的镜像为busybox。基于busybox镜像的容器需要对**/mnt**目录下的数据进行持久化，在YAML文件指定使用名称为nfs的PersistenVolumeClaim对容器的数据进行持久化。

# This mounts the nfs volume claim into /mnt and continuously

# overwrites /mnt/index.html with the time and hostname of the pod.

apiVersion: v1

kind: Deployment

metadata:

name: busybox-deployment

spec:

replicas: 2

selector:

name: busybox-deployment

template:

metadata:

labels:

name: busybox-deployment

spec:

containers:

- image: busybox

command:

- sh

- -c

- 'while true; do date > /mnt/index.html; hostname >> /mnt/index.html; sleep $(($RANDOM % 5 + 5)); done'

imagePullPolicy: IfNotPresent

name: busybox

volumeMounts:

# name must match the volume name below

- name: nfs

mountPath: "/mnt"

#

volumes:

- name: nfs

persistentVolumeClaim:

claimName: nfs-pvc

通过kubectl create创建busy-deployment部署：

$ kubectl create -f {path}/nfs-busybox-deployment.yaml

# 第三章 应用实践

# 基于Dockfile构建docker镜像实践(2019-04-17 Y)

对于容器云来说，镜像是最核心的基础。在容器云中运行的基础是容器，而容器的来源是镜像。在本章节中阐述如何基于Dockerfile文件构建镜像，构建镜像之前需要先了解Dockerfile文件和核心指令；然后如何基于Docker文件构建出镜像。

## Dockerfile文件和核心指令

在Kubernetes中运行容器的前提是已存在构建好的镜像文件，而通过Dockerfile文件构建镜像是最好方式。Dockerfile是一个文本文件，在此文件中的可以设置各种指令，以通过docker build命令自动构建出需要的镜像。Dockerfile文件必需以FROM命令开始，然后按照文件中的命令顺序逐条进行执行。在文件以#开始的内容会被看做是对相关命令的注释，此命令不会被执行。

# Comment

INSTRUCTION arguments

下面是一个典型的Dockerfile文件，此Dockerfile用于构建一个docker镜像仓库的镜像。Dockerfile文件的格式如下，在文件中对于大小写是不敏感的。但是为了方便的区分命令和参数，一般以大写的方式编写命令。此镜像的基础镜像为alpine:3.4，构建一个docker镜像仓库的镜像：

# Build a minimal distribution container

FROM alpine:3.4

RUN set -ex \

&& apk add --no-cache ca-certificates apache2-utils

COPY ./registry/registry /bin/registry

COPY ./registry/config-example.yml /etc/docker/registry/config.yml

VOLUME ["/var/lib/registry"]

EXPOSE 5000

COPY docker-entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]

CMD ["/etc/docker/registry/config.yml"]

### FROM：设置基础镜像

FROM命令为后续的命令设置基础镜像，它是Dockerfile文件的第一条命令，FROM命令的格式如下：

FROM <image>[:<tag>] [AS <name>]

### RUN：设置构建镜像时执行的命令

RUN命令有两种格式，下面是shell格式的RUN命令，在Linux中RUN的默认命令是/bin/sh；在Windows中默认命令为cmd /S /C：

RUN <command>

下面是exec格式的RUN命令：

RUN ["executable", "param1", "param2"]

RUN指令将会在当前镜像顶部的新层中执行任何命令，并提交结果。提交的结果镜像将用于Dockerfile文件的下一步。分层RUN指令和生成提交符合Docker的核心概念，可以从镜像历史中的任何点镜像创建，非常类似于源代码管理。

### CMD：设置容器的默认执行命令

CMD指令的主要目的是为容器提供一个默认的执行命令，在一个Dockerfile只能有一条CMD指令，如果设置了多条CMD指令，则只有最后一条CMD指令会生效。在Dockerfile文件中，CMD指令有如下三种格式：

* exec格式，这是推荐的格式：

CMD ["executable","param1","param2"]

* 为ENTRYPOINT提供参数：

CMD ["param1","param2"]

* shell格式：

CMD command param1 param2

如果在Dockerfile中，CMD被用来为ENTRYPOINT指令提供参数，则CMD和ENTRYPOINT指令都应该使用exec格式。当基于镜像的容器运行时，将会自动执行CMD指令。如果在docker run命令中指定了参数，这些参数将会覆盖在CMD指令中设置的参数。

### ENTRYPOINT：设置容器为可执行文件

通过ENTRYPOINT指令可以将容器设置作为可执行的文件，ENTRYPOINT 有两种格式：

* exec格式，这是推荐的格式：

ENTRYPOINT ["executable", "param1", "param2"]

* shell格式：

ENTRYPOINT command param1 param2

下面是是启动一个nginx的例子，端口为80：

docker run -i -t --rm -p 80:80 nginx

docker run <image>命令行参数将会被追加到exec格式的ENTRYPOINT所有元素之后，并将会覆盖使用CMD指定的所有元素。这就允许江参数传递到入口点，例如，docker run <Image＞ -d 将通过-d 参数传递到入口点。可以使用docker run –entrypoint 字段覆盖“ENTRYPOINT ”指令。如果在Dockerfile文件设置了多条ENTRYPOINT指令，则只会生效最后的一条指令。

#### ENTRYPOINT指令exec格式示例

可以使用ENTRYPOINT 的exec形式来设置相对稳定的默认命令和参数，然后使用任何形式的CMD指令来设置可能发生变化的参数。

FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

当运行容器是，可以看到只有一个top进程在运行：

$ docker run -it --rm --name test top -H

top - 08:25:00 up 7:27, 0 users, load average: 0.00, 0.01, 0.05

Threads: 1 total, 1 running, 0 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.1 us, 0.1 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem: 2056668 total, 1616832 used, 439836 free, 99352 buffers

KiB Swap: 1441840 total, 0 used, 1441840 free. 1324440 cached Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1 root 20 0 19744 2336 2080 R 0.0 0.1 0:00.04 top

通过docker exec命令，能够参考容器的更多信息。

$ docker exec -it test ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 2.6 0.1 19752 2352 ? Ss+ 08:24 0:00 top -b -H

root 7 0.0 0.1 15572 2164 ? R+ 08:25 0:00 ps aux

下面的Dockerfile显示使用ENTRYPOINT在前台运行Apache:

FROM debian:stable

RUN apt-get update && apt-get install -y --force-yes apache2

EXPOSE 80 443

VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

#### ENTRYPOINT指令的shell格式

通过为*ENTRYPOINT*指定文本格式的参数，此参数将在*/bin /sh -c*中进行执行。这个形式将使用shell处理，而不是shell环境变量，并且将忽略任何的CMD或docker run运行命令行参数。

FROM ubuntu

ENTRYPOINT exec top -b

#### CMD和ENTRYPOINT交互

在CMD和ENTRYPOINT指令中都可以定义容器运行时所执行的命令，下面是它们之间协作的一些规则：

1）在Dockerfile至少需要设置一条CMD或者ENTRYPOINT指令；

2）当将容器作为可执行文件使用时，建议定义ENTRYPOINT指令；

3）CMD作为为ENTRYPOINT命令定义默认参数的一种方式；

4）当使用带有参数的命令运行容器时，CMD将会被覆盖。

下表是显示了不同的ENTRYPOINT / CMD指令组合的命令执行情况：

表11-1 ENTRYPOINT / CMD指令组合

|  |  |  |  |
| --- | --- | --- | --- |
|  | No ENTRYPOINT | ENTRYPOINT exec\_entry p1\_entry | ENTRYPOINT [“exec\_entry”, “p1\_entry”] |
| No CMD | 报错，这种情况不运行出现 | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry |
| CMD [“exec\_cmd”, “p1\_cmd”] | exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry exec\_cmd p1\_cmd |
| CMD [“p1\_cmd”, “p2\_cmd”] | p1\_cmd p2\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry p1\_cmd p2\_cmd |
| CMD exec\_cmd p1\_cmd | /bin/sh -c exec\_cmd p1\_cmd | /bin/sh -c exec\_entry p1\_entry | exec\_entry p1\_entry /bin/sh -c exec\_cmd p1\_cmd |

### ENV：设置环境变量

Env指令通过<键><值>对设置环境变量，“Env”指令有两种形式。第一种形式，即ENV <Key>  < value >，将一个变量设置一个值。第一个字符作为“<键>”，之后的整个字符串将被处理为“<值>”，包括空白字符。

ENV <key> <value>

第二种形式，即ENV <Key>＝Value>…，允许一次设置多个变量。与命令行解析一样，通过引用和反斜杠可在值内包含空格。

ENV <key>=<value> ...

例如：

ENV myName="John Doe" myDog=Rex\ The\ Dog \

myCat=fluffy

或者：

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat fluffy

### ADD：添加内容到容器中

ADD指令用于从当前机器或远程URL中的<src>中拷贝文件、目录，并将它们添加到镜像文件系统的<dest>中。在指令中能够设置多个<src>，*–chown*仅仅在构建Linux容器镜像时起作用，ADD指令有两种格式：

ADD [--chown=<user>:<group>] <src>... <dest>

下面的ADD指令格式可以在源和目标路径包含空格。

ADD [--chown=<user>:<group>] ["<src>",... "<dest>"]

<src>可以包含通配符，例如：

ADD hom\* /mydir/ # 添加所有以"hom"开头的文件到镜像中的/mydir目录下。

ADD hom?.txt /mydir/ # ? is replaced with any single character, e.g., "home.txt"

<dest>是容器一个绝对路径，或者是一个相对于WORKDIR的相对路径，

ADD test relativeDir/ # 添加"test"到容器中`WORKDIR`/relativeDir/

ADD test /absoluteDir/ # 添加"test"到容器中的/absoluteDir/

ADD指令遵循下面的规则：

* <src>路径必需在构建的上下文中；不能使用 ADD ../someting /someting，这是因为docker build的第一步就是发送上下文目录给docker daemon。
* 如果<src>是一个URL，并且<dest>不是以斜线结束的情况，则会从URL中下载一个文件，并将其拷贝到<dest>；
* 如果<src>是一个URL，并且<dest>以斜线结束，则会然后从URL中导出文件名，并将文件下载到<dest>/<filename>中。例如：ADD http://example.com/foobar /，则会在容器的/目录下创建foobar文件，并将URL中foobar文件中的内容复制到容器中/foobar文件中。
* 如果<src>是一个目录，那么将会拷贝整个目录下的内容，并包括文件系统的元数据。需要注意的时，拷贝时，并不会拷贝目录本身，而只是拷贝目录下内容。
* 如果<src>是本地的一个压缩(例如：gzip、bzip2、xz等格式)文件，则会对其进行解压缩。对于来自于远程的URL，则不会进行解压缩。
* 如果<src>是一个普通文件，将会直接将文件和它的元数据拷贝到镜像的<dest>目录下。
* 如果指定了多个<src>，如果这些<src>中存在目录或使用了通配符，则<Dest>必须是一个目录，并且必须以斜杠/结尾。
* 如果<dest>不是以斜杠/结尾，它将被认为是一个文件，那么<src>的内容将被写到<dest>中。

### COPY：拷贝内容到镜像中

COPY指令用于从<src>中拷贝文件或目录，并将其添加到镜像文件系统的<path>目录下。在指令中可以指定多个< src>资源，但是文件和目录的路径将被解释为相对于当前构建上下文的资源。COPY指令与ADD指令的功能基本上相似，但ADD能够从远程拷贝，以及解压缩文件。COPY指令有两种格式：

COPY [--chown=<user>:<group>] <src>... <dest>

当目录中存在空格时，请使用下面的格式：

COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]

### WORKDIR：设置当前工作目录

WORKDIR指令用于为RUN、CMD、ENTRYPOINT、COPY和ADD这些指令设置当前的工作目录。如果WORKDIR不存在，则会自动创建一个，即使后续不使用。

WORKDIR /path/to/workdir

在Dockerfile文件中，可以设置多个WORKDIR指令。如果给定了一个相对路径，则后续WORKDIR设置的路径是相对于上一个相对路径的路径：

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

在Dockerfile中，最后的pwd命令输出的为：/a/b/c

### EXPOSE：设置暴露的端口

EXPOSE指令用于告知容器运行人员，容器在运行时将会监听哪个指定的网络端口。并可以指定端口的协议是TCP或UDP，如果没有指定协议，则默认为TCP协议。EXPOSE指令的格式如下：

EXPOSE <port> [<port>/<protocol>...]

“EXPOSE”指令实际上并不发布端口，它仅仅起着在构建镜像和运行容器的人员之间进行告知的作用。要在运行容器时实际发布端口，则需要通过在docker run命令使用-p和-P来发布和映射一个或者多个端口。

### LABEL：设置镜像的元数据信息

LABEL指令用于为镜像添加一些描述的元数据，LABEL是一系列的键值对，它的格式如下：

LABEL <key>=<value> <key>=<value> <key>=<value> ...

下面是LABEL指令的示例：

LABEL "com.example.vendor"="ACME Incorporated"

LABEL com.example.label-with-value="foo"

LABEL version="1.0"

LABEL description="This text illustrates \

that label-values can span multiple lines."

通过docker inspect命令，可以查看镜像中的标签信息：

"Labels": {

"com.example.vendor": "ACME Incorporated"

"com.example.label-with-value": "foo",

"version": "1.0",

"description": "This text illustrates that label-values can span multiple lines.",

"multi.label1": "value1",

"multi.label2": "value2",

"other": "value3"

},

### VOLUME：设置存储卷

VOLUME指令用于创建一个带有指定名称的挂载点，并将其标记为来自于本地主机或其他容器的存储卷。该值可以是JSON数组、VOLUME [“/var/log/“]，或者是具有多个参数的普通字符串，例如VOLUME /var／log 或 VOLUME /var／log /var／db。

VOLUME ["/data"]

## 构建镜像

在定义后Dockerfile文件，并准备好相关的内容后，就可以通过docker build命令从Dockerfile和上下文构建docker镜像。构建的上下文是位于指定路径或URL中的文件集合。构建过程可以引用上下文中的任何文件。例如，构建可以使用复制指令来引用上下文中的文件。

docker build [OPTIONS] PATH | URL | -

### 命令选项

构建镜像时，根据情况添加如下的命令选项。

表11-2 docker build命令选项

|  |  |  |
| --- | --- | --- |
| 名称 | 默认值 | 描述 |
| --add-host |  | 添加定制host-to-IP映射(host:ip) |
| --build-arg |  | 设置构建时的变量 |
| --cache-from |  | 考虑被作为缓存源的镜像 |
| --cgroup-parent |  | 容器的可选父cgroup |
| --compress |  | 使用gzip压缩构建上下文 |
| --cpu-period |  | 限制CPU CFS（完全公平调度程序）周期 |
| --cpu-quota |  | 限制CPU CFS（完全公平调度程序）配额 |
| --cpu-shares , -c |  | CPU的份额（基于相对权重） |
| --cpuset-cpus |  | 允许执行的CPU（0-3，0，1） |
| --cpuset-mems |  | 允许执行的内存（0-3，0，1） |
| --disable-content-trust | true | 忽略镜像验证 |
| --file , -f |  | Dockerfile文件的名称(默认值为”PATH/Dockerfile“) |
| --force-rm |  | 总是移除中间容器 |
| --iidfile |  | 将镜像ID写入文件 |
| --isolation |  | 所使用的容器隔离技术 |
| --label |  | 为镜像设置元数据 |
| --memory , -m |  | 设置内存限制 |
| --memory-swap |  | Swap限制等于内存加swap：“-1”允许无限swap |
| --network |  | 在构建期间为RUN指令设置网络模式 |
| --no-cache |  | 在构建镜像时，不使用缓存 |
| --platform |  | 如果服务器是多平台能力的，设置平台 |
| --pull |  | 一直尝试拉取镜像的最新版本 |
| --quiet , -q |  | 抑制构建输出和打印镜像ID |
| --rm | true | 构建成功后移除中间容器 |
| --security-opt |  | 安全选项 |
| --shm-size |  | /dev/shm的大小 |
| --squash |  | 将新建的层挤压成一个新的层 |
| --stream |  | 流连接到服务器，以协商构建的上下文 |
| --tag , -t |  | 为构建的镜像以”name:tab“格式打上标签 |
| --target |  | 设置目标构建阶段进行构建 |
| --ulimit |  | Ulimit选项 |

### URL参数

URL参数可以引用三种资源：Git存储库、预打包的tabball上下文和纯文本文件，本文主要描述如何使用Git仓库构建镜像。当 URL 参数指向一个Git仓库的位置，仓库将作为构建的上下文。系统的递归获取库及其子模块，提交历史不保存。仓库是首先被拉取到本地主机的临时目录。成功后，此临时目录被发送给Docker daemon作为构建上下文。

Git URL接受的上下文配置，由冒号分隔：进行分割。第一部分表示Git将签出的引用，可以是分支、标签或远程引用。第二部分表示存储库内的子目录，该目录将用作构建上下文。

例如：使用container分支的docker目录构建镜像：

$ docker build https://github.com/docker/rootfs.git#container:docker

下面是通过git构建镜像的合法表达：

表11-3 通过git构建镜像的表达式

|  |  |  |
| --- | --- | --- |
| 建立语法后缀 | 提交使用 | 构建上下文使用 |
| myrepo.git | refs/heads/master | / |
| myrepo.git#mytag | refs/tags/mytag | / |
| myrepo.git#mybranch | refs/heads/mybranch | / |
| myrepo.git#pull/42/head | refs/pull/42/head | / |
| myrepo.git#:myfolder | refs/heads/master | /myfolder |
| myrepo.git#master:myfolder | refs/heads/master | /myfolder |
| myrepo.git#mytag:myfolder | refs/tags/mytag | /myfolder |
| myrepo.git#mybranch:myfolder | refs/heads/mybranch | /myfolder |

### 构建示例

下面是通过本地路径构建一个私有镜像仓库镜像的示例，在此示例中，通过-t设置了镜像的标签为registry:latest；构建上下文为当前执行命令所在的目录，Dockerfile为当前上下文中的文件。

$ docker build -t registry:latest .

下面是通过Git仓库构建镜像的示例：

$ docker build -t regiestry:latest https://github.com/docker/distribution-library-image.git

## 最佳实践

为了构建出更加合理和高效的镜像，需要遵循下面的这些最佳实践经验。

**1）不安装不必要的包**

为了减少复杂性、依赖性、文件大小和构建时间，避免安装额外的或不必要的包。

**2）最小化层的数量**

在旧版本的Docker中，最小化镜像中的层数是非常重要，这样可以确保它们的性能，添加以下特征能够减少这种限制：

* 在docker 1.10和更高版本中，只有RUN、COPY和ADD会创建新的层。而其它指令仅会创建临时的中间镜像，并且不直接增加构建的大小。
* 在docker17.05和更高版本中，可以进行多阶段构建，只将需要的工件复制到最终镜像中。这允许在中间构建阶段中包含工具和调试信息，而不增加最终镜像的大小。

**3）解耦应用**

每个容器应该只关注一个业务问题。将应用程序分解到多个容器中，从而可以更容易地进行水平扩容和重用。例如，Web应用程序栈可能由三个单独的容器组成，每个容器都有自己的镜像，以解耦的方式管理Web应用程序、数据库和内存缓存。尽最大的努力使容器尽可能保持清晰和模块化。如果容器相互依赖，可以使用docker容器的网络来确保这些容器可以进行通信。

**4）排序多行参数**

只要有可能，尽量按字母顺序排序多行参数，可以减轻以后的变化。这有助于避免重复包，并使列表更容易更新。下面是buildpack-deps镜像的一个例子：

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

**5）利用构建缓存**

在构建镜像时，Docker会通过Dockerfile文件中的指令，并按指定的顺序执行每一个指令。在检查每个指令时，Docker会在缓存中寻找可重用的现有镜像，而不是创建新的（重复的）镜像。

如果不使用缓存，可以在docker构建命令上使用–no-cache＝true选项。但是，如果让Docker使用缓存，则需要了解它何时能找到匹配的镜像。docker遵循的基本规则如下：

* 从已经存在于缓存中的父镜像开始，将下一条指令与从该基础镜像派生的所有子镜像进行比较，以查看其中是否使用完全相同的指令构建了其中的一个子镜像。如果没有，则缓存无效。
* 在大多数情况下，简单地将Dockerfile文件中的指令与其中一个子镜像中指令进行比较就足够了。然而，某些指令需要更多的检查和解释。
* 对于ADD和COPY指令，通过检查镜像中文件的内容，并为每个文件计算校验和，这些校验和中未考虑文件的最后修改和上次访问时间。在缓存查找期间，会将校验和与现有镜像中的校验和进行比较。如果文件中的任何内容（如内容和元数据）发生变化，则缓存被无效。
* 除了ADD和COPY命令之外，缓存检查并不查看容器中的文件来确定缓存匹配情况。

**6）尽量使用官方的alphine镜像作为基础镜像**

只要有可能，请使用官方的镜像作为基础镜像。并建议使用alpine镜像，因为它的尺寸会被严格控制（目前低于5 MB），但仍然是一个完整的Linux发行版。

**7）ADD和COPY的使用**

虽然ADD和COPY功能类似，一般来说，优先使用COPY，那是因为COPY比ADD更透明。

# 基于Kubernetes的Devops平台(2019-5-23 Y)

DevOps是开发Development与运维Operation的组合，是一种软件研发工程文化理念和实践指导体系，在自动化软件交付流程及基础设施变更过程中，强调开发人员与测试、运维等专业人员之间的协作与沟通。籍希望在软件研发团队中建立一种良好的文化与环境，从而使得软件的构建、测试、发布得以更加快速、频繁，以及保证软件能够更加稳定地进行。本章节以Kubernetes为基础，构建了一套解决方案。首先，本章节就devops的整体方案进行介绍；在方案介绍的基础上，详细阐述了如何基于gitlab、nexues、jenkins、maven、docker和rancher进行平台搭建；在最后，提供了使用此devops方案的示例。

## 基于kubernetes devops的整体方案

本方案以Kubernetes为基础，为基于java语言研发团队提供一套完整的devops解决方案。在此方案中，开发人员基于eclipse集成开发环境进行代码；开发人员所开发的代码交由由gitlab进行托管、版本管理和分支管理；代码的依赖更新和构建工作由Maven进行处理；为了提升工作效率和代码质量，在devops中引入SonarQube进行代码检查；对于打包构建后代码，交由docker进行镜像构建，并在私有镜像仓库中对镜像进行管理；最后，devops会将自动从私有镜像仓库从拉取镜像，并在Rancher中进行部署。

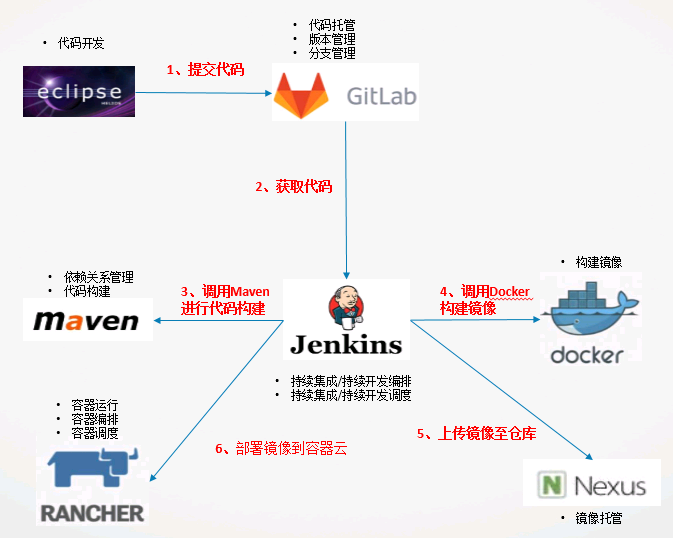


图12-1 devops整体方案

基于此devops解决方案的整体工作过程如下所示：

1）开发人员基于eclipse集成开发环境镜像代码开发的，将代码到gitlab中进行托管；

2）jenkins从gitlab拉取代码；

3）jenkins调用Maven对代码进行打包构建；

4）jenkins调用docker构建镜像；

5）jenkins将构建好的镜像上传至基于Nexus的私有镜像仓库；

6）jenkins拉取镜像，并部署镜像至Rancher中。

## 组件安装部署

此部分描述需要为devops部署的组件，根据整体方案，devops需要使用gitlab、jenkins、nexus、maven、docker和kubernetes这些组件和系统。其中，gitlab、jenkins、nexus都在kubernetes中安装部署，在jenkins中包含了maven；docker直接在物理机提供，对于docker的部署不在此部分进行阐述。

### 代码托管工具-Gitlab

在devops方案中，代码的托管基于Gitlab。下面是在Kubernetes中部署gitlab的YAML配置文件，在此文件中定义了gitlab部署和服务。gitlab部署使用的镜像为gitlab/gitlab-ce:latest，并暴露了443、80和22这三个端口，并通过NFS对配置文件、日志和数据进行持久化。在服务中，端口的类型为NodePort，即允许集群外的用户可以通过映射在主机节点上的端口对gitlab进行访问。

#------------------------定义代理服务-------------------------

apiVersion: v1

kind: Service

metadata:

  name: gitlab

spec:

  type: NodePort

  ports:

  # Port上的映射端口

  - port: 443

    targetPort: 443

    name: gitlab443

  - port: 80

    targetPort: 80

    name: gitlab80

  - port: 22

    targetPort: 22

    name: gitlab22

  selector:

    app: gitlab

---

# ------------------------定义Gitlab的部署 -----------------------

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: gitlab

spec:

  selector:

    matchLabels:

      app: gitlab

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: gitlab

    spec:

      containers:

      # 应用的镜像

      - image: gitlab/gitlab-ce

        name: gitlab

        imagePullPolicy: IfNotPresent

        # 应用的内部端口

        ports:

        - containerPort: 443

          name: gitlab443

        - containerPort: 80

          name: gitlab80

        - containerPort: 22

          name: gitlab22

        volumeMounts:

        # gitlab持久化

        - name: gitlab-persistent-config

          mountPath: /etc/gitlab

        - name: gitlab-persistent-logs

          mountPath: /var/log/gitlab

        - name: gitlab-persistent-data

          mountPath: /var/opt/gitlab

      imagePullSecrets:

      - name: devops-repo

      volumes:

      # 使用nfs互联网存储

      - name: gitlab-persistent-config

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/gitlab/config

      - name: gitlab-persistent-logs

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/gitlab/logs

      - name: gitlab-persistent-data

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/gitlab/data

通过kubectl命令工具，执行如下的命令，在kubernetes集群中部署gitlab：

$ kubectl create -f {path}/gitlab.yaml

### 镜像仓库-Nexus

在本devops方案中，采用Nexus作为docker私有镜像仓库和maven的远程仓库。下面是在Kubernetes中部署Nexus的YAML配置文件，在此文件中定义了Nexus部署和服务。Nexus部署使用的镜像为sonatype/nexus3:latest，并暴露了8081、5001这两个端口，并通过NFS对配置文件、日志和数据进行持久化。在服务中，端口的类型为NodePort，即允许集群外的用户可以通过映射在主机节点上的端口对nexus进行访问。其中，5001作为docker私有镜像仓库的端口。

#------------------------定义Nexus代理服务-------------------------

apiVersion: v1

kind: Service

metadata:

  name: nexus3

spec:

  type: NodePort

  ports:

  # Port上的映射端口

  - port: 8081

    targetPort: 8081

    name: nexus8081

  - port: 5001

    targetPort: 5001

    name: nexus5001

  selector:

    app: nexus3

---

#-----------------------定义Nexus部署-------------------------

apiVersion: apps/v1beta2

kind: Deployment

metadata:

   name: nexus3

   labels:

     app: nexus3

spec:

   replicas: 1 #副本数为3

   selector: #选择器

     matchLabels:

       app: nexus3 #部署通过app：nginx标签选择相关资源

   template: #Pod的模板规范

     metadata:

       labels:

         app: nexus3 #Pod的标签

     spec:

       imagePullSecrets:

       - name: devops-repo

       containers: #容器

       - name: nexus3

         image: sonatype/nexus3

         imagePullPolicy: IfNotPresent

         ports: #端口

         - containerPort: 8081

           name: nexus8081

         - containerPort: 5001

           name: nexus5001

         volumeMounts:

         - name: nexus-persistent-storage

           mountPath: /nexus-data

       volumes:

       # 宿主机上的目录

       - name: nexus-persistent-storage

         nfs:

           server: 192.168.8.150

           path: /k8s-nfs/nexus

通过kubectl命令工具，执行如下的命令，在kubernetes集群中部署nexus：

$ kubectl create -f {path}/nexus.yaml

### 流水线工具-Jenkins

在本devops方案中，采用jenkins作为流水线工具。下面是在Kubernetes中部署jenkins的YAML配置文件，在此文件中定义了jenkins部署和服务。jenkins部署使用的镜像为lw96/java8-jenkins-maven-git-vim:latest，并暴露了8080这个端口，并通过NFS对配置文件和数据进行持久化。在服务中，端口的类型为NodePort，即允许集群外的用户可以通过映射在主机节点上的端口对jenkins进行访问。另外，在此镜像中也提供maven和java。

#------------------------定义代理服务-------------------------

apiVersion: v1

kind: Service

metadata:

  name: jenkins-devops

spec:

  type: NodePort

  ports:

  # Port上的映射端口

  - port: 8080

    targetPort: 8080

    name: pipeline8080

  selector:

    app: jenkins-devops

---

# ------------------------定义mysql的部署 -----------------------

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

  name: jenkins-devops

spec:

  selector:

    matchLabels:

      app: jenkins-devops

  revisionHistoryLimit: 2

  template:

    metadata:

      labels:

        app: jenkins-devops

    spec:

      containers:

      # 应用的镜像

      - image: 10.0.32.148:1008/lw96/java8-jenkins-maven-git-vim

        name: jenkins-devops

        imagePullPolicy: IfNotPresent

        # 应用的内部端口

        ports:

        - containerPort: 8080

          name: pipeline8080

        volumeMounts:

        # jenkins-devops持久化

        - name: pipeline-persistent

          mountPath: /etc/localtime

        # jenkins-devops持久化

        - name: pipeline-persistent

          mountPath: /jenkins

        # jenkins-devops持久化

        - name: pipeline-persistent-repo

          mountPath: /root/.m2/repository

        - name: pipeline-persistent-mnt

          mountPath: /mnt

      volumes:

      # 使用nfs互联网存储

      - name: pipeline-persistent

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/jenkins-devops

      - name: pipeline-persistent-mnt

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/jenkins-devops/mnt

      - name: pipeline-persistent-repo

        nfs:

          server: 192.168.8.150

          path: /k8s-nfs/jenkins-devops/repo

通过kubectl命令工具，执行如下的命令，在kubernetes集群中部署jenkins：

$ kubectl create -f {path}/jenkins-devops.yaml

注意，后续将执行下面的操作：

* 将Kubernetes集群的kubeconfig文件拷贝到192.168.8.150主机的/k8s-nfs/jenkins-devops/mnt目录下；
* 将maven的settings.xml文件拷贝至到192.168.8.150主机的/k8s-nfs/jenkins-devops/repo目录下；
* 将maven的依赖插件包拷贝至到192.168.8.150主机的/k8s-nfs/jenkins-devops/repo目录下。

## devops平台搭建

在devops平台各个组件安装完成后，需要进行相应的设置。首先，需要基于nexus为devops提供maven的远程仓库和docker的私有镜像仓库；第二步，为设置maven所使用的仓库；第四步，设置docker所使用的镜像仓库和对外暴露服务，以支持构建镜像；最后，对jenkins进行设置，包括gitlab、maven、docker和kubernetes等相关的插件和对maven信息进行设置。

### nexus设置

nexus在devops中承担两个功能，作为maven的远程仓库和作为docker的私有镜像仓库。在本方案中，使用nexus默认安装的maven-snapshots、maven-releases和maven-public这三个仓库。

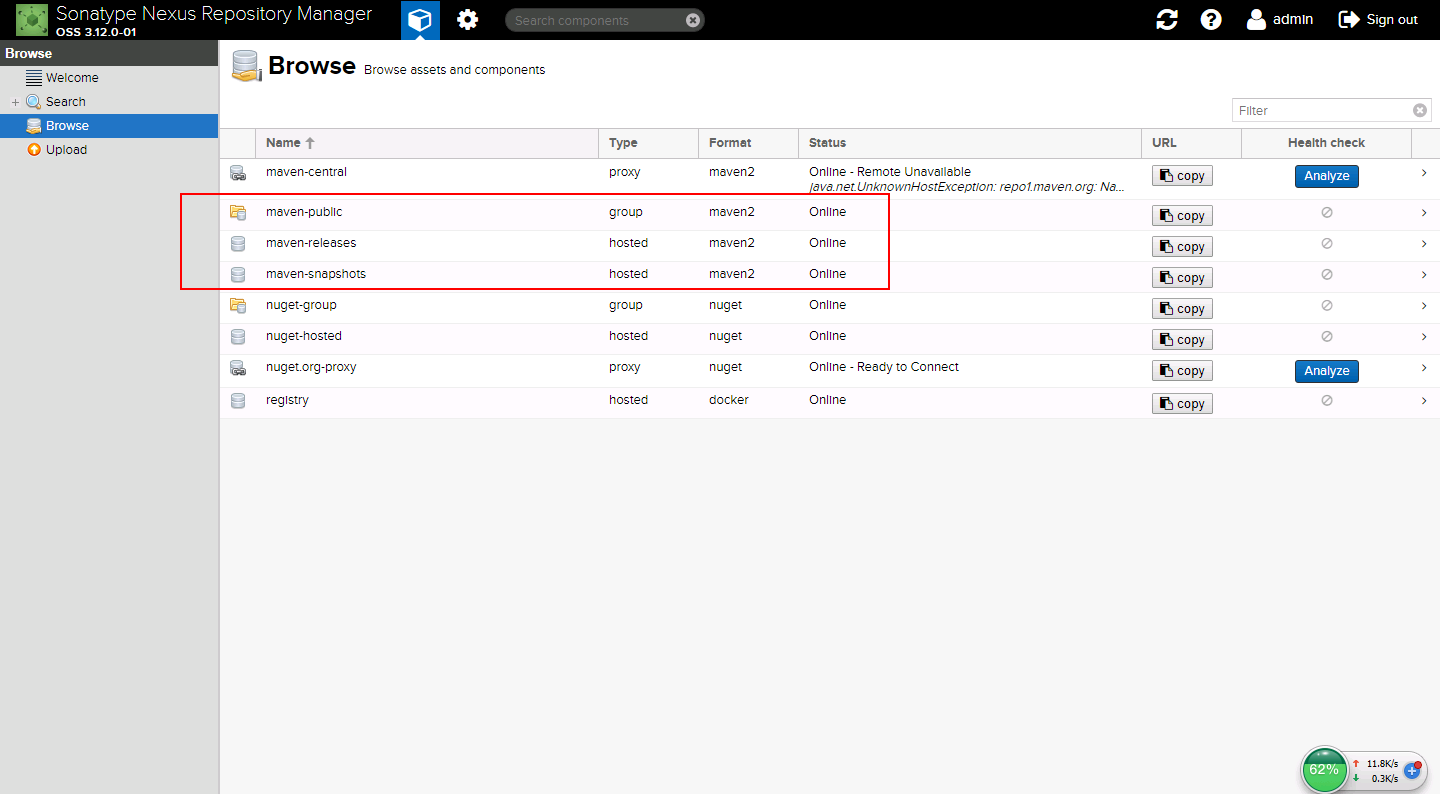


图12-2 nexus界面

并为docker创建一个名为registry的私有镜像仓库，其端口为5001：

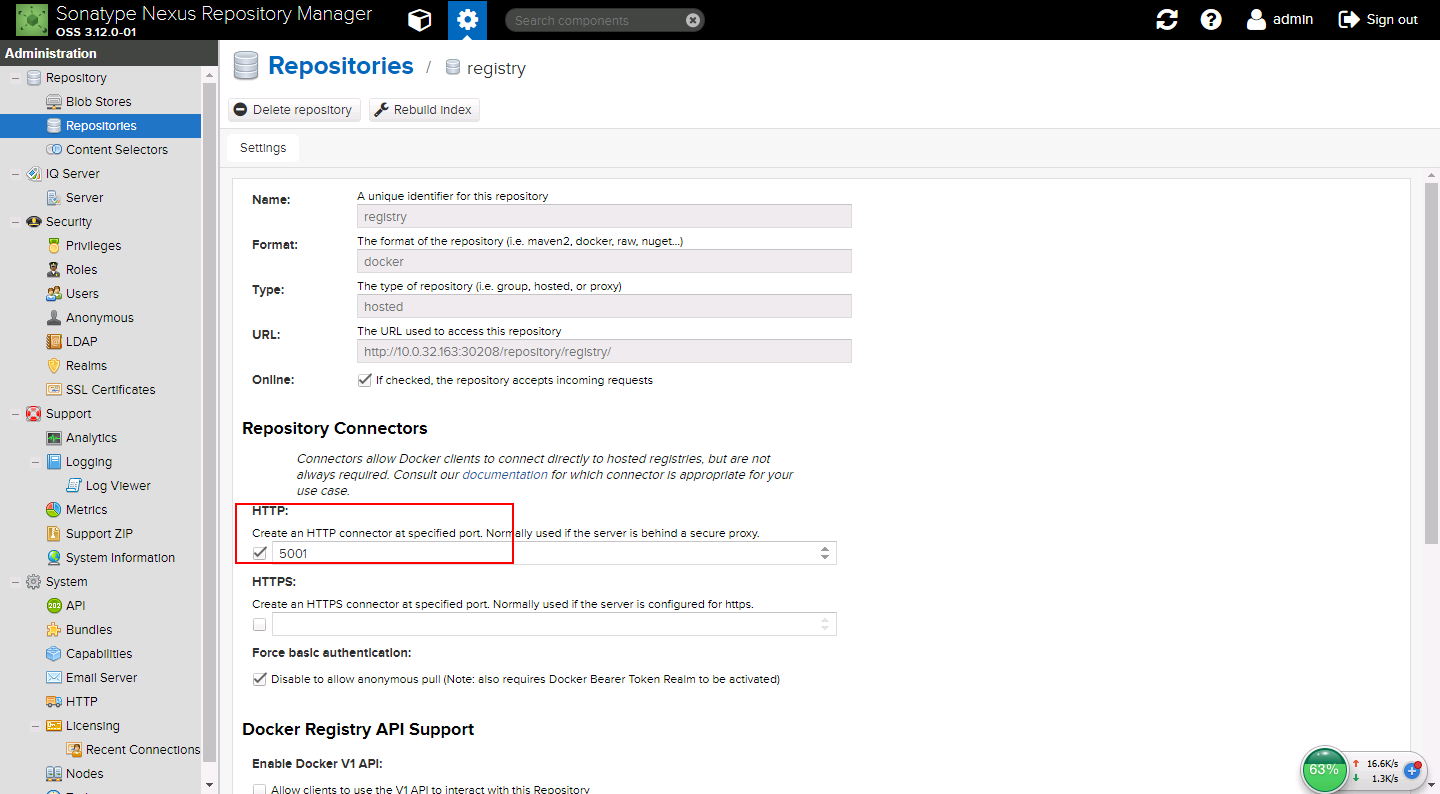


图12-3 设置docker镜像仓库信息

### maven设置

在DevOps平台中，Maven负责管理代码的依赖关系和构建。maven通过settings.xml文件设置运行环境，包括与远程仓库的连接。

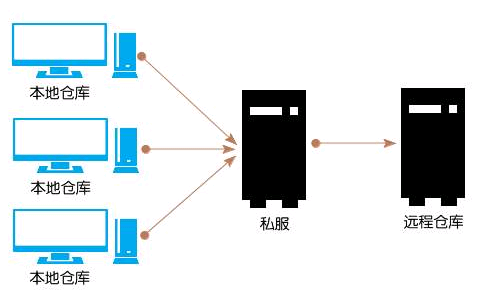


图12-4 Maven仓库体系

本文中的settings.xml如下所示，http://nexus3:8081中的nexus3是在kubernetes中的服务名称。将settings.xml拷贝至192.168.8.150机器的/k8s-nfs/jenkins-devops/repo目录下。

<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

<pluginGroups>

</pluginGroups>

<proxies>

</proxies>

<!-- server 用于配置服务端的信息-->

<servers>

<server>

<id>maven-release</id>

<username>admin</username>

<password>admin123</password>

</server>

<server>

<id>maven-snapshots</id>

<username>admin</username>

<password>admin123</password>

</server>

</servers>

<!-- mirrors 用于方便地切换远程仓库地址的途径，相当于一个拦截器，会拦截maven对remote repository的相关请求，并把请求里的remote repository地址重定向到mirror里配置的地址。-->

<mirrors>

<mirror>

<id>nexus</id>

<url>http://nexus3:8081/repository/maven-public/</url>

<mirrorOf>\*</mirrorOf>

</mirror>

</mirrors>

<!-- profiles 个性配置, 主要包括activation,repositories,pluginRepositories 和properties元素。单独定义profile后，并不会生效，需要通过满足条件来激活。-->

<profiles>

<profile>

<id>maven-release</id>

<!-- action用于激活profile -->

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<!-- repositories定义其他开发库和插件开发库。对于团队来说，肯定有自己的开发库，可以通过此配置来定义。-->

<repositories>

<!-- repositorie用于定义仓库 -->

<repository>

<id>maven-release</id>

<name>Repository for Release</name>

<url>http://nexus3:8081/repository/maven-releases/</url>

<releases>

<enabled>false</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

</repositories>

</profile>

<profile>

<id>maven-snapshots</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<repositories>

<repository>

<id>maven-release</id>

<name>Repository for Release</name>

<url>http://nexus3:8081/repository/maven-snapshots/</url>

<releases>

<enabled>false</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

</repositories>

</profile>

</profiles>

</settings>

### docker设置

为了能够支持将远程提交的代码构建成镜像，以及将构建好的镜像上传至镜像仓库需要在/etc/docker/daemon.json的文件中添加下面的内容。其中，10.0.32.163:32476为镜像仓库的地址和端口，tcp://0.0.0.0:4243为对外暴露的地址和端口。

{

"hosts":["tcp://0.0.0.0:4243","unix:///var/run/docker.sock"],

"insecure-registries":["10.0.32.163:32476"]

}

并通过执行下面的命令重启docker服务：

$ systemctl daemon-reload

$ systemctl restart docker

### jenkins设置

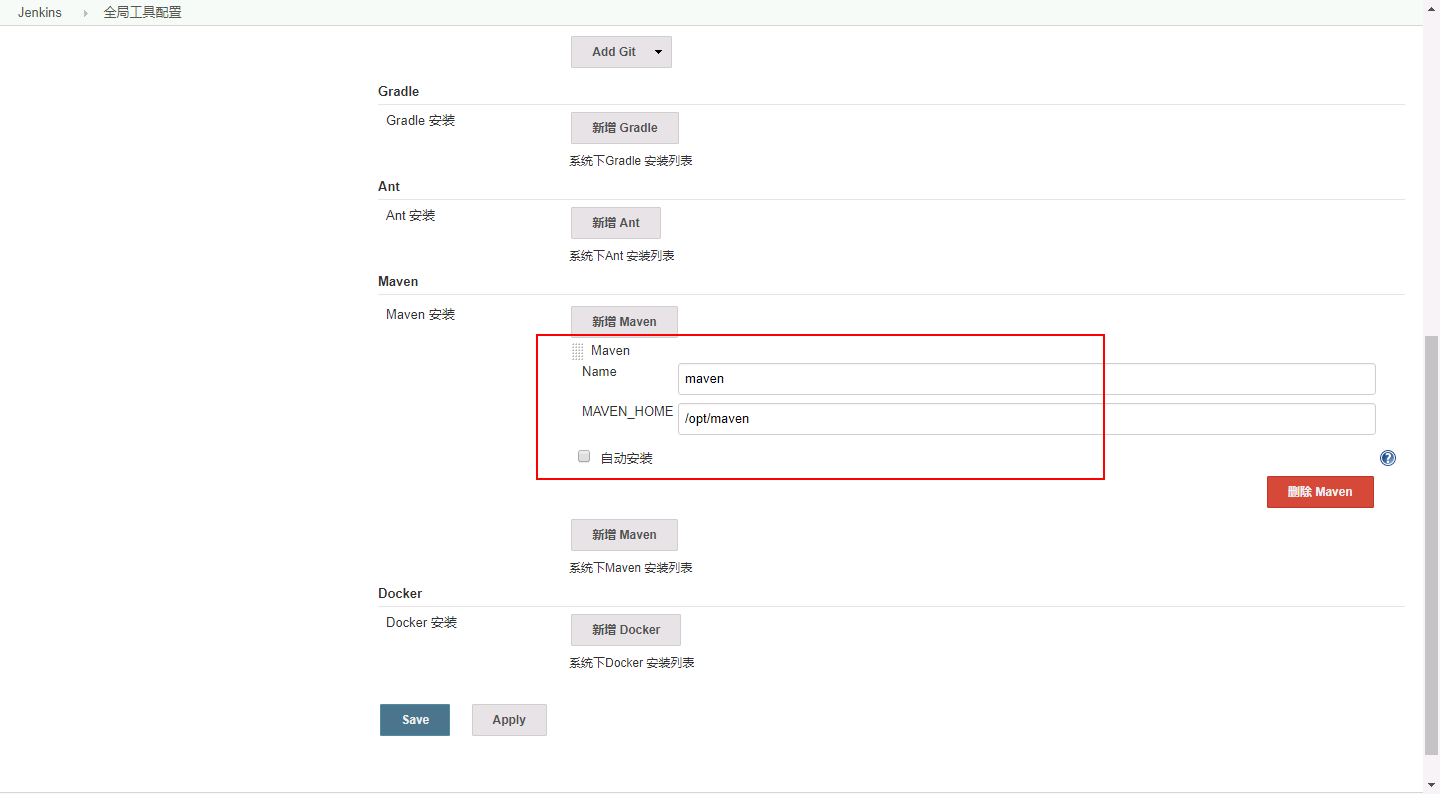
#### 安装插件

jenkins作为devops平台的流程线工具，需要从gitlab中获取代码，并提交给maven进行构建；在代码构建成功后，调用docker构建镜像，并将上传至基于Nexus的私有镜像仓库；最终，在Kubernetes中部署和运行镜像。为了实现上述能力，需要在jenkins中安装如下插件：

* git plugin：与gitlab集成的插件，用于获取代码；
* maven plugin：与maven集成的插件，用于构建代码；
* CloudBees Docker Build and Publish plugin：与docker集成的插件，用于构建docker镜像，并上传至镜像仓库；
* Kubernetes Continuous Deploy Plugin：与kubernetes集成的插件，用于将镜像部署到kubernetes环境。

#### maven设置

在jenkins中的“全局工具配置”页面，设置maven的安装信息，name可以按照自己的喜好填写，MAVEN\_HOME为maven的安装地址，此处为/opt/maven。



## devops持续集成示例

此示例用于向研发团队展示如何基于devops方案进行整体研发过程的管理，核心的步骤包括：在gitlab中创建项目，在eclipse中进行代码开发，在jenkins中设置相关信息，执行代码构建和查看部署后的应用。

**1）安装git客户端和创建密钥**

在工作计算上安装git客户端，并通过下面的命令创建ssh密钥：

ssh-keygen -t rsa -C "your.email@example.com" -b 4096

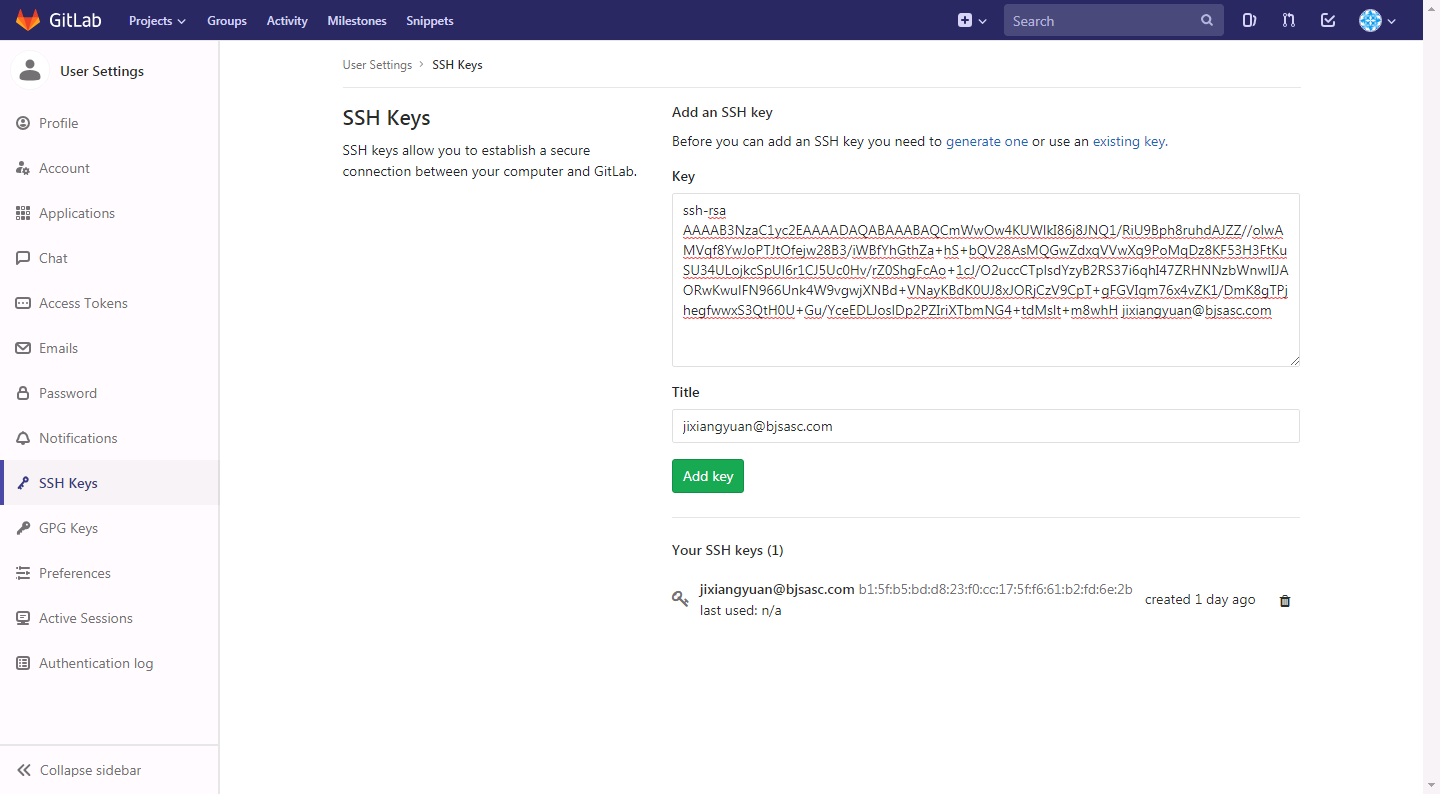
执行上述命令后，会在本地的～/.ssh目录创建id\_rsa.pub(公钥)和id\_rsa(私钥)。

**2）在gitlab创建oms项目**

进入gitlab，并创建一个名称为oms的项目。

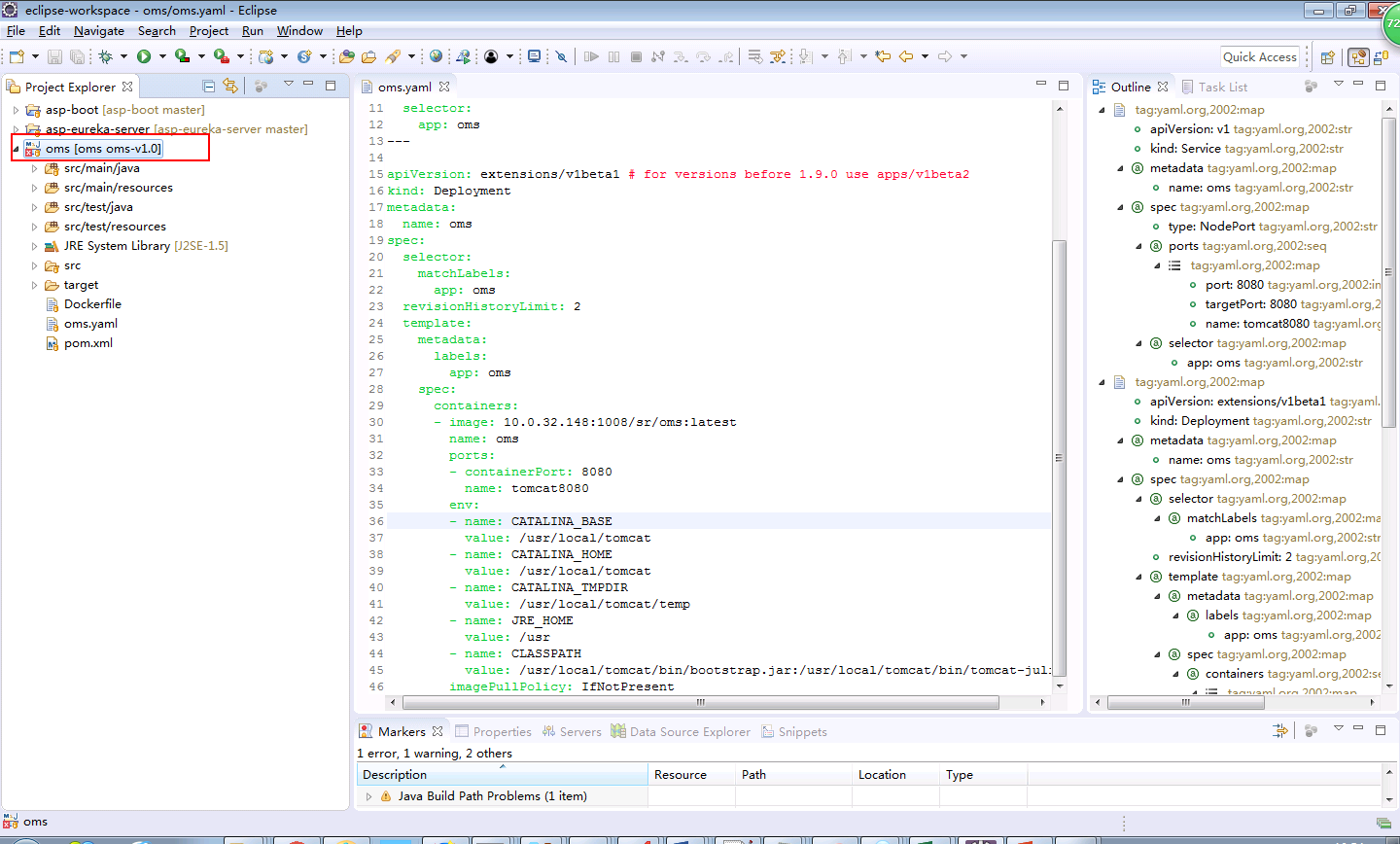


在gitlab中添加所创建的公钥：



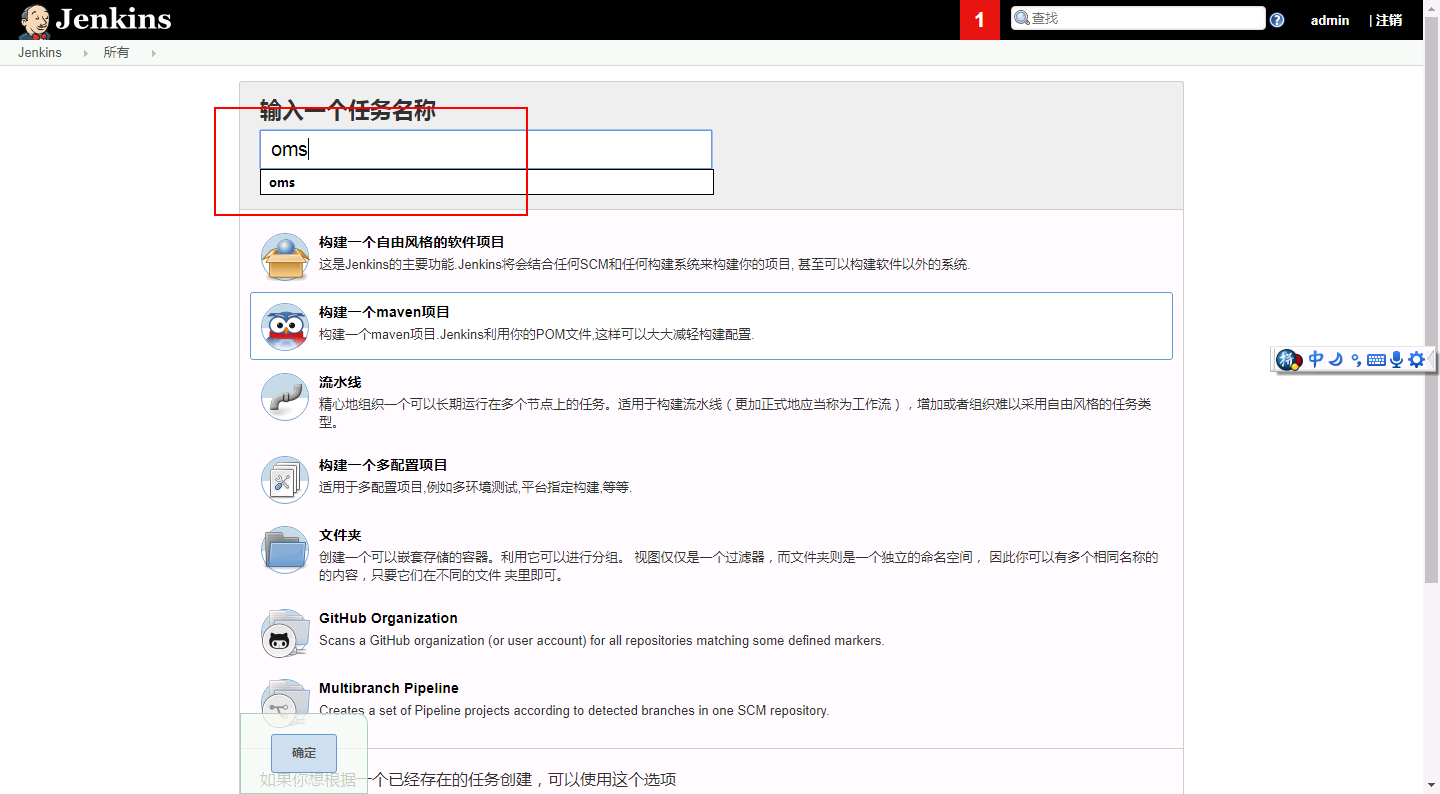
**3）在eclipse中进行代码开发**

在eclipse中创建类型为maven的oms项目，并进行代码开发。在完成代码开发后，将代码上传至gitlab中进行代码托管。



**4）在jenkins中创建oms项目**

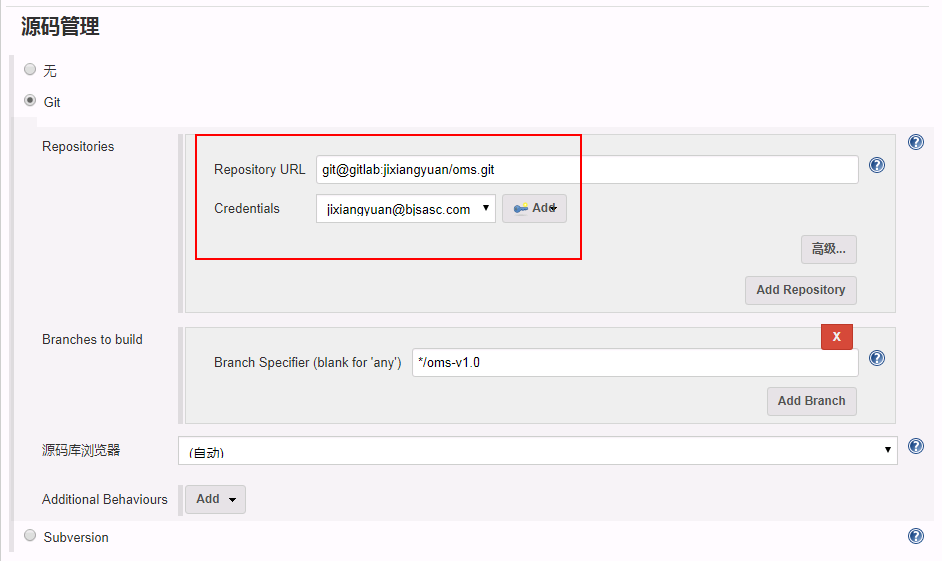
进入jenkins，创建一个名称为oms的maven项目。



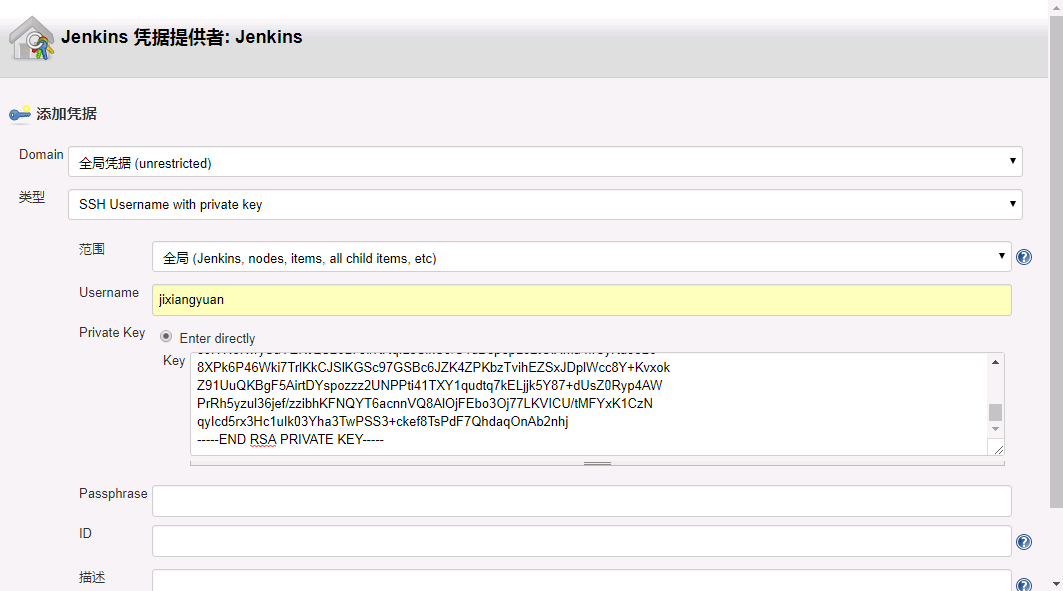
**5）在jenkins中设置获取代码信息**

在jenkins中，进入oms的配置页面，在源代码管理处设置获取源代码的相关信息。

* Repository URL：项目在gitlab中的地址；
* Credentials：访问gitlab的认证方式；
* Branch Specifier：代码的分支。



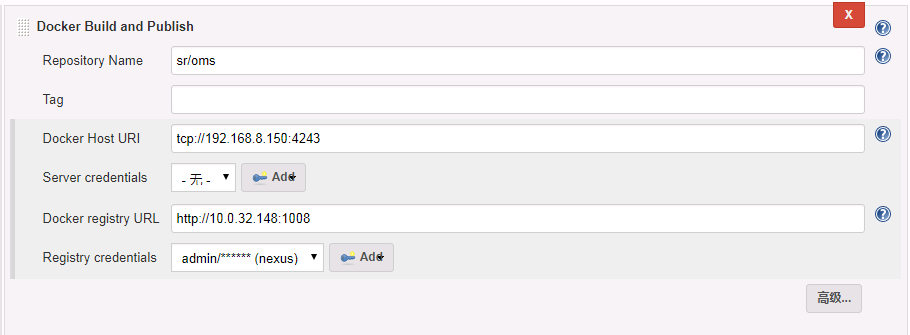
此处访问gitlab的认证方式为“SSH Username with private key”，Private Key的值来自于～/.ssh目录下id\_rsa(私钥)的内容。



**6）在jenkins中设置构建和上传镜像信息**

在oms项目的配置的“Docker Build and Publish”部分，填写如下的信息；

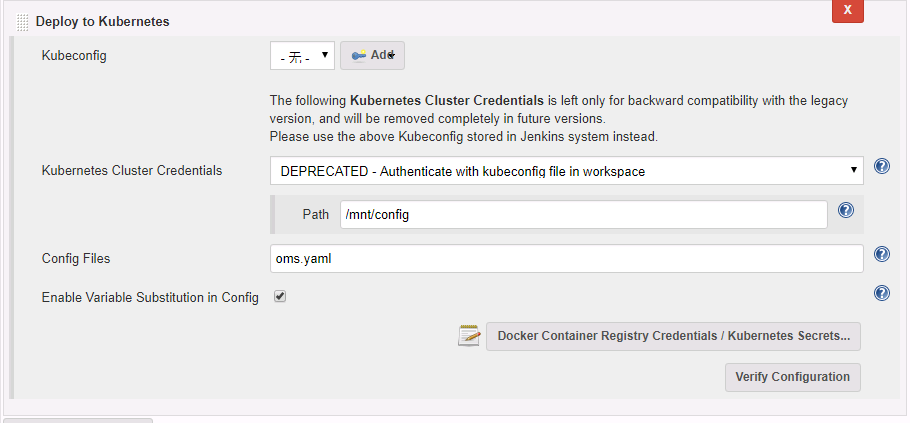
* Repository Name：镜像的名称；
* Tag：镜像的版本；
* Docker Host URI：docker服务的地址和端口；
* Docker registry URL：docker镜像仓库的地址；
* Registry credentials：镜像仓库的认证方式。



**7）在jenkins中设置部署容器信息**

在oms项目的配置的“Deploy to Kubernetes”部分，填写如下的信息；

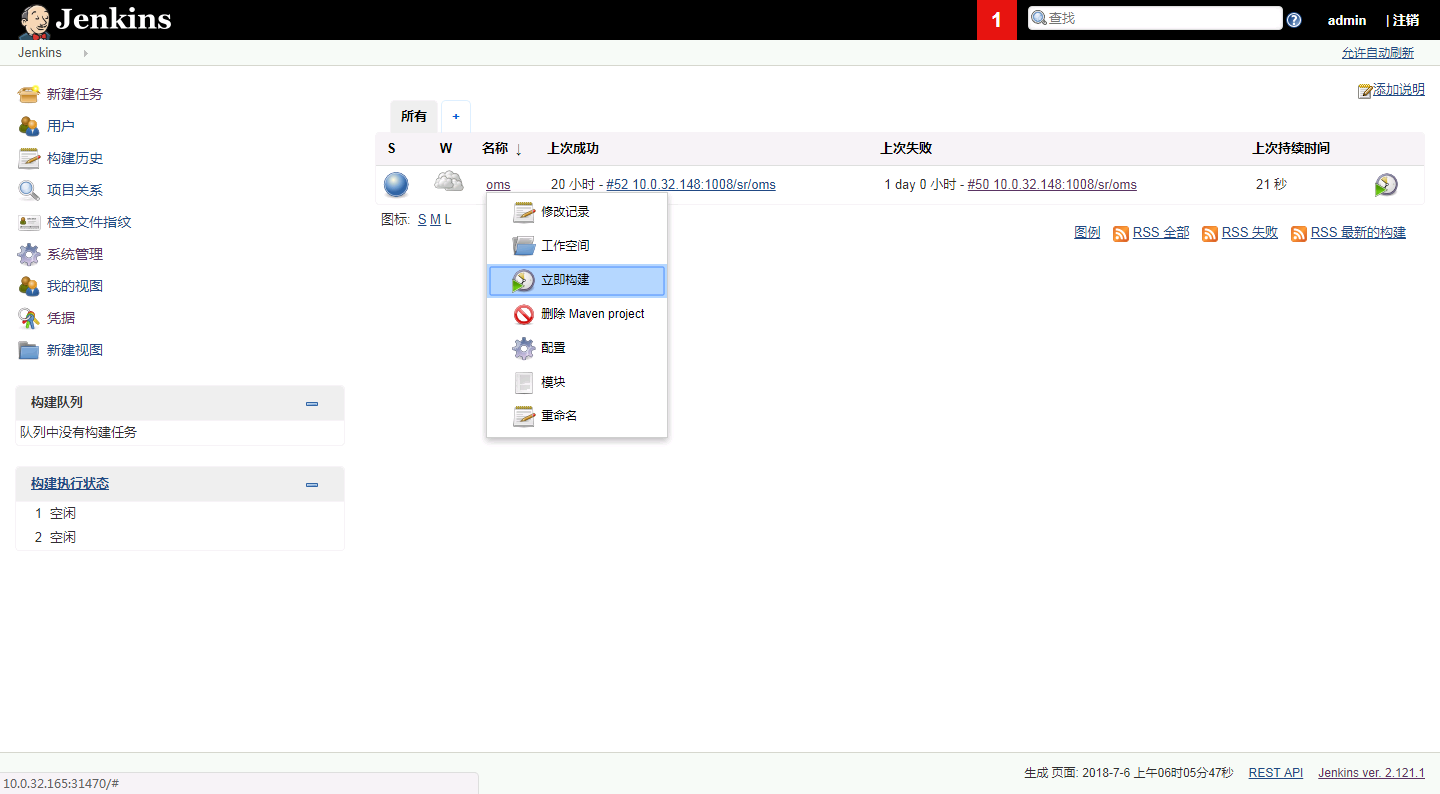
* Kubernetes Cluster Credentials：Kubernetes集群的认证方式；
* Path：kubeconfig文件的所在地址；
* Config Files：构建YAML配置文件。



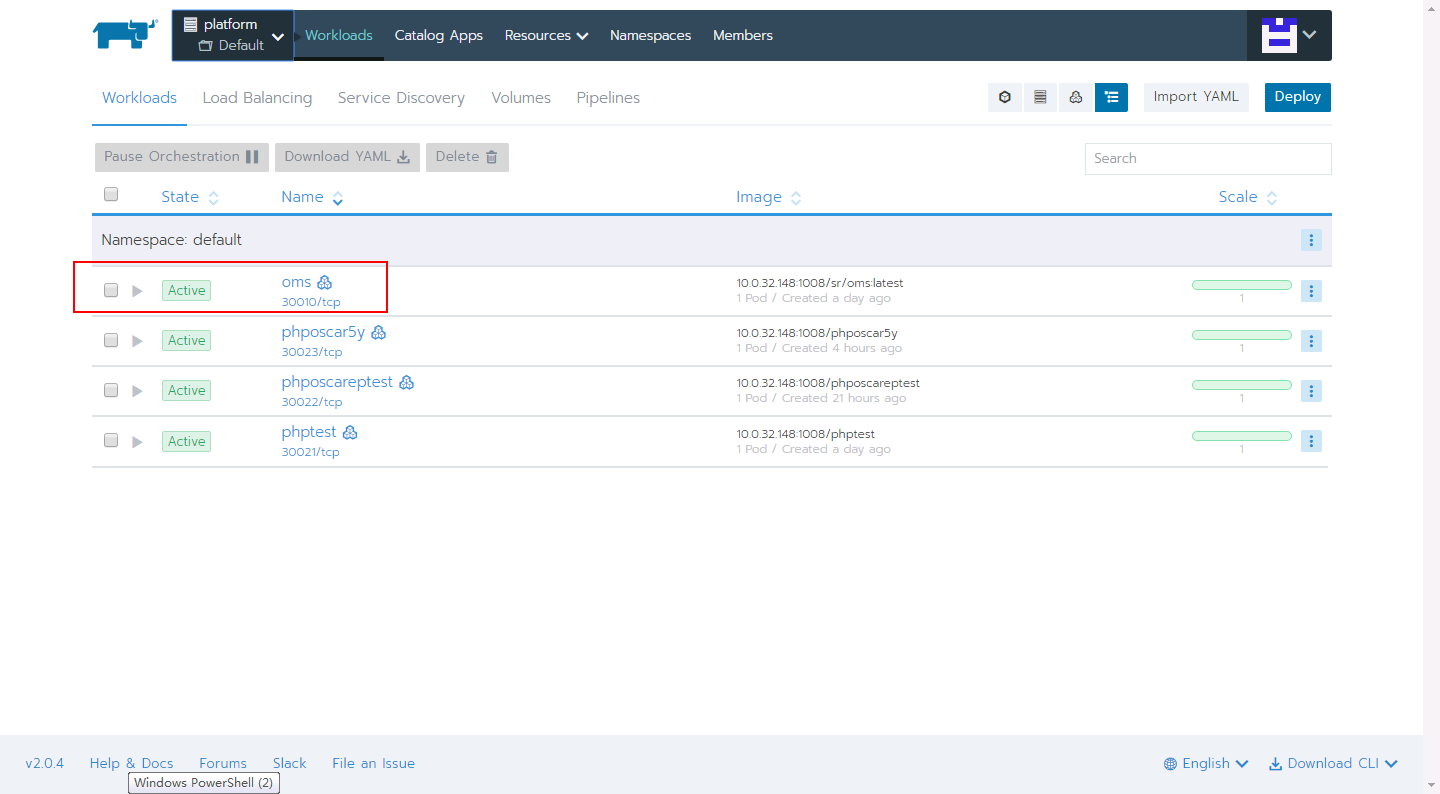
**8）在jenkins中执行oms构建**

在oms项目创建和设置完成后，可以对项目进行构建操作。通过一键操作，jenkins将会完成从构建、打包成镜像和部署的所有工作内容：

* 从gitlab中获取oms的代码；
* 提交给maven进行构建；
* 调用docker构建镜像；
* 上传镜像至Nexus的私有镜像仓库；
* 部署镜像到kubernetes集群。



**9）在kubernetes中查看部署情况**

进入kubernetes(本文使用的为Rancher系统)界面，在default命名空间下，可以看到已部署的oms。

# 部署高可用的mysql（2019-5-22 Y）

此章节用于描述如何在Kubernetes中部署一个高可用的mysql，为整个集群提供关系型的数据存储。首先，对mysql进行了简介，并就提供的高可用方案进行了描述；在了解mysql和高可用方案的基础上，基于yaml文件进行mysql高可用方案的安装部署和验证。

## **MySQL简介**

MySQL 是一个开源的关系型数据库管理系统，使用标准的sql语言，由瑞典 MySQL AB 公司开发，当前属于 Oracle 公司。能够 支持大型的数据库，可以处理上千万条的数据记录。可以运行于在Windows、Linux等多种系统上；支持C、C++、Python、Java、Perl、PHP、Eiffel、Ruby和Tcl等编程语言。对于32位系统，MySQL的表文件最大可支持4GB，对于64位系统，MySQL支持最大的表文件为8TB。

## **MySQL的高可用方案**

本文的MySQL高可用方案为主从复制+读写分离，即由单一的master和多个slave所构成。其中，客户端通过master对数据库进行写操作，通过slave端进行读操作。master出现问题后，可以将应用切换到slave端。 此方案是MySQL官方提供的一种高可用解决方案，节点间的数据同步采用MySQL Replication技术。

MySQL Replication从一个MySQL数据库服务器（master）的数据复制到一个或多个MySQL数据库服务器（slave）。在默认情况下，复制是异步的；slave不需要一直接收来自主机的更新。根据配置，可以复制数据库中的所有数据库、选定的数据库，或者特定的表。

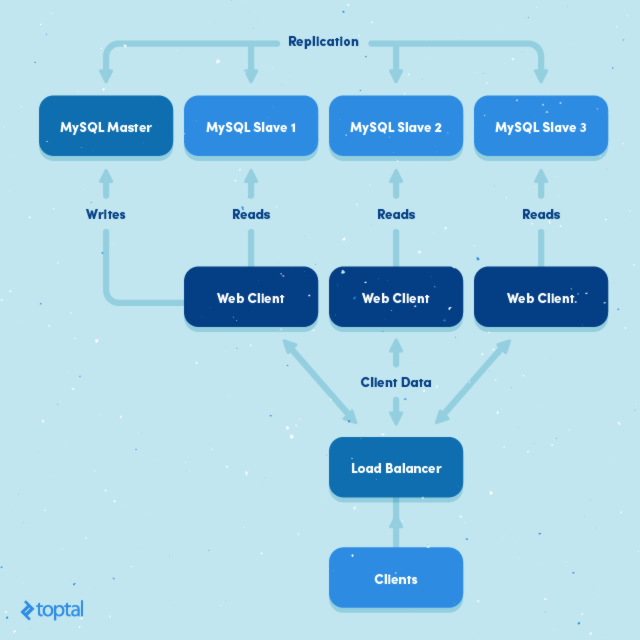


图13-1 mysql高可用方案

MySQL中复制的优点包括：

* 扩容解决方案：在多个slave之间扩展负载以提高性能。在这种模式下，所有的写入和更新操作都必须在主服务器上进行。然而，读取操作通过slave镜像。该模型可以提高写入操作的性能，同时，也能够通过增加slave的节点数量，从而显著地提升读取速度。
* 数据安全：数据从master被复制到slave，并且slave可以暂停复制过程。因此，可以在不损坏master的情况下，在slave上运行备份服务。
* 分析：现场数据可以在master上创建，而对信息的分析可以在slave进行，而不影响master的性能。
* 远程数据分发：可以使用复制为远程站点创建本地数据的副本，而不必一直通过访问master。

此高可用的解决方案适用于对数据实时性要求不是特别严格的场景，在使用时可以通过廉价的硬件来扩展slave的节点数量，将读压力分散到多台slave的机器上面。此方案能够在很大的程度上解决数据库读取数据的压力瓶颈问题，这是因为在大多的应用系统中，读压力要比写压力大很多多。

## **安装部署**

### ****环境要求****

在部署MySql之前,需要具备如下的环境要求:

* 已有Kubernetes 1.6+环境；
* 在Kubernetes中提供多个(具体数量根据有状态副本集的个数而定)容量大于10g的持久化存储卷。

### ****部署MySql****

部署此mysql的相关YAML文件包括一个ConfigMap、两个Service和一个StatefulSet所组成。ConfigMap用于设置mysql的配置信息，Service用于为Statefulset用于部署mysql本身。

#### ****创建ConfigMap****

通过YAML文件创建名为mysql的ConfigMap：

**$ kubectl create -f {path}/mysql-configmap.yaml --namespace=kube-public**

**apiVersion: v1**

**kind: ConfigMap**

**metadata:**

**name: mysql**

**labels:**

**app: mysql**

**data:**

**master.cnf: |**

**# Apply this config only on the master.**

**[mysqld]**

**log-bin**

log\_bin\_trust\_function\_creators=1

lower\_case\_table\_names=1

**slave.cnf: |**

**# Apply this config only on slaves.**

**[mysqld]**

**super-read-only**

**log\_bin\_trust\_function\_creators=1**

#### ****创建Services****

通过yaml文件创建mysql和mysql-read这两个Service：

**$ kubectl create -f {path}/mysql-services.yaml --namespace=kube-public**

**# Headless service for stable DNS entries of StatefulSet members.**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: mysql**

**labels:**

**app: mysql**

**spec:**

**ports:**

**- name: mysql**

**port: 3306**

**clusterIP: None**

**selector:**

**app: mysql**

**---**

**# Client service for connecting to any MySQL instance for reads.**

**# For writes, you must instead connect to the master: mysql-0.mysql.**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: mysql-read**

**labels:**

**app: mysql**

**spec:**

**ports:**

**- name: mysql**

**port: 3306**

**selector:**

**app: mysql**

StatefulSet控制器为Pod创建了一个DNS条目，而Headless服务为DNS条目提供一个主机。因为Headless服务的名称为mysql，其他Pod通过<pod-name>.mysql访问此Pod。客户端访问被称为mysql-read，客户端服务通过访问mysql-read读取数据。通过连接myql执行写入数据的操作。

#### ****创建StatefulSet****

通过yaml文件创建名为mysql的StatefulSet：

$ kubectl create **-**f **{**path**}/**mysql**-**statefulset**.**yaml **--**namespace**=**kube**-**public

**apiVersion: apps/v1**

**kind: StatefulSet**

**metadata:**

**name: mysql**

**spec:**

**selector:**

**matchLabels:**

**app: mysql**

**serviceName: mysql**

  replicas: 3

  template:

    metadata:

      labels:

        app: mysql

    spec:

      initContainers:

      - name: init-mysql

        image: mysql:5.7

        command:

        - bash

        - "-c"

        - |

          set -ex

          # Generate mysql server-id from pod ordinal index.

          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1

          ordinal=${BASH\_REMATCH[1]}

          echo [mysqld] > /mnt/conf.d/server-id.cnf

          # Add an offset to avoid reserved server-id=0 value.

          echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf

          # Copy appropriate conf.d files from config-map to emptyDir.

          if [[ $ordinal -eq 0 ]]; then

            cp /mnt/config-map/master.cnf /mnt/conf.d/

          else

            cp /mnt/config-map/slave.cnf /mnt/conf.d/

          fi

        volumeMounts:

        - name: conf

          mountPath: /mnt/conf.d

        - name: config-map

          mountPath: /mnt/config-map

      - name: clone-mysql

        image: gcr.io/google-samples/xtrabackup:1.0

        command:

        - bash

        - "-c"

        - |

          set -ex

          # Skip the clone if data already exists.

          [[ -d /var/lib/mysql/mysql ]] && exit 0

          # Skip the clone on master (ordinal index 0).

          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1

          ordinal=${BASH\_REMATCH[1]}

          [[ $ordinal -eq 0 ]] && exit 0

          # Clone data from previous peer.

          ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C /var/lib/mysql

          # Prepare the backup.

          xtrabackup --prepare --target-dir=/var/lib/mysql

        volumeMounts:

        - name: data

          mountPath: /var/lib/mysql

          subPath: mysql

        - name: conf

          mountPath: /etc/mysql/conf.d

      containers:

      - name: mysql

        image: mysql:5.7

        env:

        - name: MYSQL\_ALLOW\_EMPTY\_PASSWORD

          value: "1"

        ports:

        - name: mysql

          containerPort: 3306

        volumeMounts:

        - name: data

          mountPath: /var/lib/mysql

          subPath: mysql

        - name: conf

          mountPath: /etc/mysql/conf.d

        resources:

          requests:

            cpu: 500m

            memory: 1Gi

        livenessProbe:

          exec:

            command: ["mysqladmin", "ping"]

          initialDelaySeconds: 30

          periodSeconds: 10

          timeoutSeconds: 5

        readinessProbe:

          exec:

            # Check we can execute queries over TCP (skip-networking is off).

            command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]

          initialDelaySeconds: 5

          periodSeconds: 2

          timeoutSeconds: 1

      - name: xtrabackup

        image: gcr.io/google-samples/xtrabackup:1.0

        ports:

        - name: xtrabackup

          containerPort: 3307

        command:

        - bash

        - "-c"

        - |

          set -ex

          cd /var/lib/mysql

          # Determine binlog position of cloned data, if any.

          if [[ -f xtrabackup\_slave\_info ]]; then

            # XtraBackup already generated a partial "CHANGE MASTER TO" query

            # because we're cloning from an existing slave.

            mv xtrabackup\_slave\_info change\_master\_to.sql.in

            # Ignore xtrabackup\_binlog\_info in this case (it's useless).

            rm -f xtrabackup\_binlog\_info

          elif [[ -f xtrabackup\_binlog\_info ]]; then

            # We're cloning directly from master. Parse binlog position.

            [[ `cat xtrabackup\_binlog\_info` =~ ^(.\*?)[[:space:]]+(.\*?)$ ]] || exit 1

            rm xtrabackup\_binlog\_info

            echo "CHANGE MASTER TO MASTER\_LOG\_FILE='${BASH\_REMATCH[1]}',\

                  MASTER\_LOG\_POS=${BASH\_REMATCH[2]}" > change\_master\_to.sql.in

          fi

          # Check if we need to complete a clone by starting replication.

          if [[ -f change\_master\_to.sql.in ]]; then

            echo "Waiting for mysqld to be ready (accepting connections)"

            until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

            echo "Initializing replication from clone position"

            # In case of container restart, attempt this at-most-once.

            mv change\_master\_to.sql.in change\_master\_to.sql.orig

            mysql -h 127.0.0.1 <<EOF

          $(<change\_master\_to.sql.orig),

            MASTER\_HOST='mysql-0.mysql',

            MASTER\_USER='root',

            MASTER\_PASSWORD='',

            MASTER\_CONNECT\_RETRY=10;

          START SLAVE;

          EOF

          fi

          # Start a server to send backups when requested by peers.

          exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \

            "xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 --user=root"

        volumeMounts:

        - name: data

          mountPath: /var/lib/mysql

          subPath: mysql

        - name: conf

          mountPath: /etc/mysql/conf.d

        resources:

          requests:

            cpu: 100m

            memory: 100Mi

      volumes:

      - name: conf

        emptyDir: {}

      - name: config-map

        configMap:

          name: mysql

  volumeClaimTemplates:

  - metadata:

      name: data

    spec:

      accessModes: ["ReadWriteOnce"]

      resources:

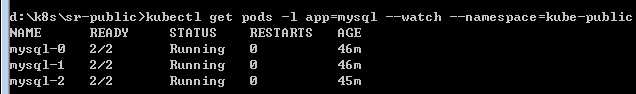
        requests:

          storage: 10Gi

通过执行如下的命令可以查看启动过程：

$ kubectl **get** pods **-**l app**=**mysql **--**watch **--namespace=**kube**-public**

在启动后，应该能够看到如下的信息：

****

## 理解有状态Pod的初始化

StatefulSet控制器按Pod的序号索引一次启动一个Pod，控制器每个Pod指派一个唯一的、稳定的名称，名称的格式为**<statefulset-name>-<ordinal-index>**。在此示例中，Pod的名称为mysql-0，此节点为master主节点；mysql-1和mysql-2，这两个节点为slave从节点。

### ****创建配置文件****

在开始启动Pod规格中的任何容器之前，Pod首先会按照YAML配置中定义的顺序运行初始化容器。

第一个初始化容器为*init-mysql*，将以顺序索引创建MySQL配置文件。

脚本从Pod名称的结尾处获取并确定它的顺序索引，顺序索引通过hostname命令获取。然后，它会按照顺序保存在conf.d目录下的server-id.cnf文件中。此行为将StatefulSet控制器提供的唯一和稳定的身份标识转为mysql服务Id的域。在init-mysql容器中，脚本使用来自于ConfigMap中master.cnf或slave.cnf。

在此例子的拓扑关系中，存在一个MySQL master节点和多个MySQL slave节点，脚本简单的指派顺序0给主节点。这能够保证MySQL主节点在创建从节点之前就已经准备就绪。

### ****克隆已存在的数据****

一般来说，当一个新的Pod加入进来作为从节点时，必须假设MySQL master已经有关于它的数据。也假设slave副本的日志必须重新开始的。这些假设对于StatefulSet的扩缩容是很关键。

第二个初始化容器是*clone-mysql*，它在空的PersistentVolume上执行克隆从节点Pod的行为。这意味着它将从已在运行的Pod中拷贝数据，因此，它的当前状态能够与从master开始的副本节点一致。

MySQL自身并没有提供能够做到上述能力的机制，因此，此例子使用开源的Percona XtraBackup工具来实现。在克隆的过程中，为了对MySQL主节点影响的最小化，脚本会要求每一个新的Pod从顺序索引值小的Pod中进行克隆。这样做的原因是，StatefulSet控制器需要一直保证Pod N需要在Pod N+1之前准备就绪。

### ****启动副本****

在初始化容器完成后，容器将正常运行。MySQL Pod由运行实际mysqld服务的MySQL容器组成，xtrabacekup容器只是作为备份的工具。xtrabackup负责监控克隆数据文件，并确定是否在从节点初始化MySQL副本。如果需要，它将等待MySQL就绪，然后执行 CHANGE MASTER TO和START SLAVE命令。

一旦一个从节点开始复制，它将记住MySQL master，并自动进行重新连接，因为从节点寻找主节点作为稳定DNS名称（mysql-0.mysql），它们自动的发现主节点。最后，在启动副本后，xtrabackup容器也监听来自于其它Pod对数据克隆的请求。

## **MySQL部署环境验证**

1）通过运行一个临时的容器(使用mysql:5.7镜像)，使用MySQL 客户端发送测试请求给MySQL master节点（主机名为mysql-0.mysql；跨命名空间的话，主机名请使用mysql-0.mysql.kube-public）

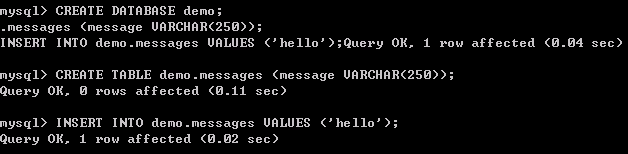
**$ kubectl run mysql-client --image=mysql:5.7 -it** **--rm** **--restart=Never -- mysql -h mysql-0.mysql.kube-public**

CREATE DATABASE demo;

CREATE TABLE demo.messages (message VARCHAR(250));

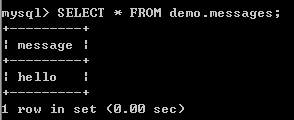
INSERT INTO demo.messages VALUES ('hello');

在master节点上创建demo数据库，并创建一个只有message字段的demo.messages的表，并为message字段插入hello值。



2）使用主机名为mysql-read来发送测试请求给服务器：

**$ kubectl run mysql-client --image=mysql:5.7 -i** **-t** **--rm** **--restart=Never -- mysql -h mysql-read.kube-public**



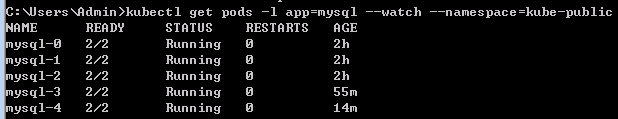
## **扩容slave的数量**

1）对于mysql副本，通过添加从节点进行扩容。

$ kubectl scale statefulset mysql --replicas=5 --namespace=kube-public

2）通过下面的命令查看新的Pod：

$ kubectl get pods -l app=mysql --watch --namespace=kube-public



 3）缩容也是无缝的：

$ kubectl scale statefulset mysql --replicas=3 --namespace=kube-public

# 部署高可用的redis（2019-05-14 Y）

本章节以Redis为例，在Kubernetes中部署一个共用的非结构化数据库。首先，本章节从整体上介绍了Redis和基于Sentinel模式的高可用方案；然后，详细描述了如何基于helm进行Redis部署，以及对部署好的Redis进行验证。

## Redis简介

Redis是一个开放源代码（BSD许可证）非结构化数据库，其在内存中存储数据，可以代理数据库、缓存和消息。Redis支持字符串、散列、列表、集合和位图等数据结构。Redis作为一个高性能的key-value数据库，在很大程度上改进了memcached这类key/value存储的不足。同时，Redis提供了Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，Python，Ruby和Erlang等语言的客户端。

Redis支持master/slave结构，数据可以从master向任意数量的slave上进行同步。Redis 与其它 key/value 缓存产品相比，具有以下三个方面特点：

* **支持内存的持久化**：可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用；
* **支持多种数据结构**：Redis不仅仅只是支持key-value类型的数据，还能够支持字符串、散列和列表等数据结构；
* **支持主从结构**：Redis支持主从结构，保证系统的高可用。

## 基于Sentinel模式的高可用方案

此处的Redis高可用方案采用Sentinel(哨兵)模式，在集群出现故障的时候自动进行故障转移，保证集群的可用性。Redis Sentinel 为Redis提供了高可用性，这意味着使用Sentinel部署的Redis，能够在没有人为干预的情况下能够抵抗某些类型的失败。Sentiel的完整功能列表如下所示：

* 监控：不间断的检查master/slave实例否是安装预期正常工作；
* 通知：当 Redis 实例出现错误的时候，会使用程序（通过 API 接口）通知管理员；
* 自动故障转移：在master发生故障时，哨兵会开启故障转移处理，将一台slave提升为master，其它的slave被重新配置使用新的master，当应用程序连接时使用新的地址配置；
* 配置信息：Sentinel作为服务发现的权威来源，客户端连接到Sentinel去获取当前Redis master的地址，如果发生故障转移，Sentinel将会汇报新的服务器地址。

Sentinel本身是一套分布式系统，它被设计成能够进行多个进程间协同工作的模式，这样的好处如下：

* 多个Sentinel一致明确给定的主机不再可用时，才会执行故障检测，这能够有效错报的概率。
* 即使只有一个Sentinel在正常运行，Redis也是可用的，从而保证系统具有较高的健壮性。

Sentinel，Redis实例（master和slave）和连接到Sentinel和Redis的客户端的数量，也是一个具有特定属性的更大的分布式系统。在本文中，定制的Redis服务器镜像会确定执行它的Pod是redis的Sentinel、master还是slave，并启动适当的服务。这个Helm chart指示Sentinel状态与环境变量。如果没有设置，新启动的Pod将查询Kunbernetes的活动master。如果不存在，则它使用一种确定的方法来检测它是否应该作为master启动，然后将“master”或“slave”写入到称为redis-role的标签中。

redis-role=master Pod是集群启动的关键。在它们完成启动，sentinel将处于等待整体。所有其他的Pod等待sentinel识别主节点。运行Pod并设置标签podIP和runID。runID是每个redis服务器生成的唯一run\_ID值的前几个字符。在正常操作中，应该只有一个redis=master Pod。如果失败，Sentinel将提名一个新的master，并适当地改变所有的redis-role的值。

通过执行如下命令可以查看Pod所承担的角色：

$ kubectl get pods -L redis-role -namespace=kube-public

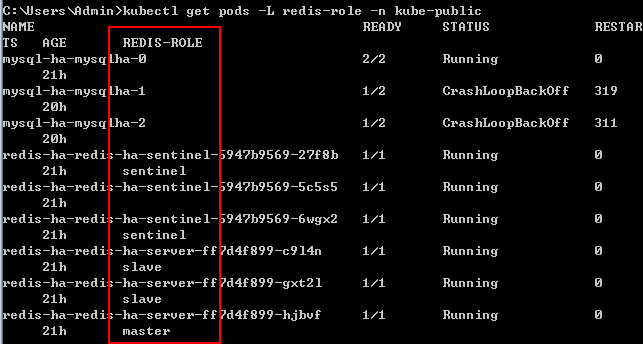


图14-1 Redis中的各角色

## 安装部署

### 环境要求

在进行Redis高可用方案部署之前，需要具备如下的环境：

* 已有Kubernetes 1.6+环境；
* 已部署helm客户端和tiller服务端（请参考：<https://docs.helm.sh/using_helm/#installing-helm>）：
* 在Kubernetes中创建了具备足够权限访问权限的service account；
* 并通过此service account在Kubernetes部署了tiller服务端（请参考：<https://docs.helm.sh/using_helm/#role-based-access-control>）。
* 在Kubernetes中提供了容量大于10g的持久化存储卷。

### Helm chart配置

下表列示了Redis chart的配置参数和默认值：

表14-1 redis chart配置参数

|  |  |  |
| --- | --- | --- |
| 参数 | 描述 | 默认值 |
| redis\_image | Redis镜像 | quay.io/smile/redis:4.0.6r2 |
| resources.master | Redis主节点CPU/内存的资源请求/限制 | Memory: 200Mi, CPU: 100m |
| resources.slave | Redis从节点CPU/内存的资源请求/限制 | Memory: 200Mi, CPU: 100m |
| resources.sentinel | 哨兵节点CPU/内存的资源请求/限制 | Memory: 200Mi, CPU: 100m |
| replicas.servers | redis master/slave pods的副本数量 | 3 |
| replicas.sentinels | sentinel pods的副本数量 | 3 |
| nodeSelector | 为Pod指派的Node标签 | {} |
| tolerations | 为Pod指派的可容忍标签 | [] |
| servers.serviceType | 设置”LoadBalancer”能够通过VPC进行访问 | ClusterIP |
| servers.annotations | 参考应用模式 | “ |
| rbac.create | 是否应该创建RBAC资源 | true |
| serviceAccount.create | 是否创建代理所使用的service account名称 | true |
| serviceAccount.name | 被使用的service account。如果未进行设置，同时如果serviceAccount.create被设置为true，则Kubernetes会在后台以模板的全名创建一个service account。 | “ |

在helm install中使用–set key=value 格式设置上述的参数值，例如：

$ helm install \

--set redis\_image=quay.io/smile/redis:4.0.6r2 \

stable/redis-ha

### 持久化

redis将持久化数据保存在容器的/redis-master-datal路径下，安装时会创建一个PersistentVolumeClaim ，并将其挂接到容器内的目录。因此，需要在Kubernetes中提前提供一个PersistentVolume。

### 通过Chart安装Redis

通过执行如下的命令，在Kubernetes中部署Redis：

$ helm install stable/redis-ha --name=redis-ha --namespace=kube-public

通过上述命令，将以默认的配置在Kubernetes中部署Redis。默认情况下，chart会安装部署3个Sentinel Pod，1个master Pod和2个slave Pod。

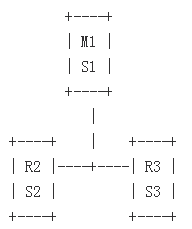
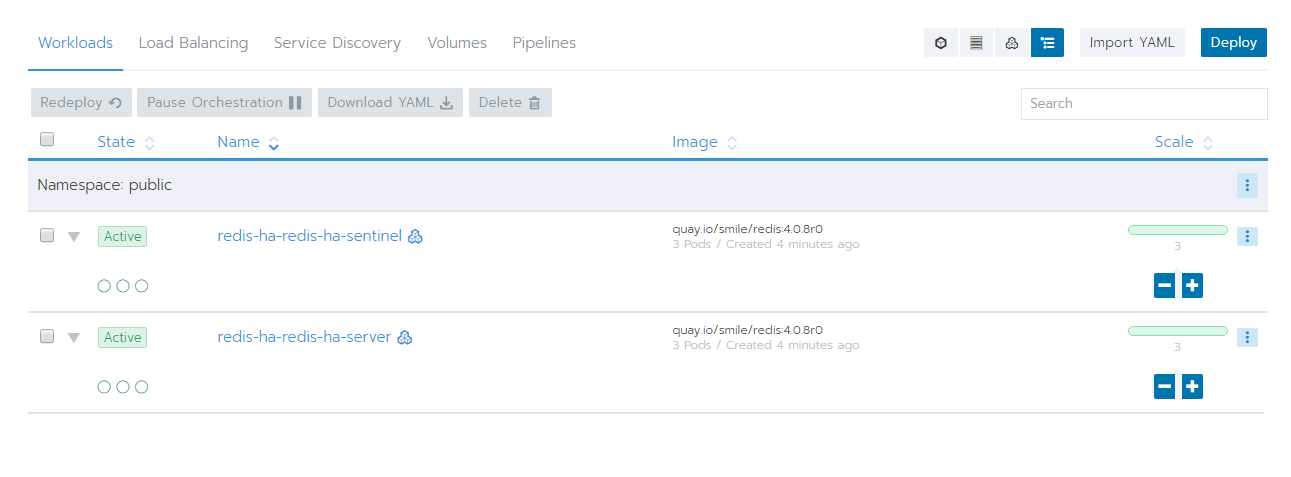


图14-1



## Helm Chart分析

MySQL Chart的目录如下，其中，values为默认的配置文件，用于为部署提供默认值。templates目录下的YAML文件是在Kubernetes进行部署的配置文件。

redis-ha

--templates # 模板目录，当与values.yaml组合时，将生成有效的Kubernetes清单文件。

----NOTES.txt

----\_helpers.tpl

----redis-auth-secret.yaml

----redis-master-service.yalm

----redis-role.yaml

----redis-rolebinding.yaml

----redis-sentinel-deployment.yaml

----redis-sentinel-service.yaml

----redis-server-deployment.yaml

----redis-serviceaccount.yaml

----redis-slave-service.yaml

--Chart.yaml # 描述chart的信息

--README.md # 可读的chart介绍文件

--values.yaml # 默认配置文件

### values.yaml

在values.yaml配置文件中设置了通过helm进行部署时的默认值。在values.yaml中，首先，定义了主Pod和哨兵Pod的请求和限制资源的要求；接着，通过nodeSelector和容忍度为Pod定义调度到哪个Node上；以及，指定容器所使用的镜像和其它的相关信息。

## Configure resource requests and limits

## ref: http://kubernetes.io/docs/user-guide/compute-resources/

##

resources:

server:

requests:

memory: 200Mi

cpu: 100m

limits:

memory: 700Mi

sentinel:

requests:

memory: 200Mi

cpu: 100m

limits:

memory: 200Mi

## Node labels and tolerations for pod assignment

## ref: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#nodeselector

## ref: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#taints-and-tolerations-beta-feature

nodeSelector: {}

tolerations: []

## Redis image version

redis\_image: quay.io/smile/redis:4.0.8r0

## replicas number for each component

replicas:

servers: 3

sentinels: 3

servers:

serviceType: ClusterIP # [ClusterIP|LoadBalancer]

annotations: {}

rbac:

# Specifies whether RBAC resources should be created

create: true

serviceAccount:

# Specifies whether a ServiceAccount should be created

create: true

# The name of the ServiceAccount to use.

# If not set and create is true, a name is generated using the fullname template

name:

## Configures redis with AUTH (requirepass & masterauth conf params)

auth: false

## Redis password

## Defaults to a random 10-character alphanumeric string if not set and auth is true

## ref: https://github.com/kubernetes/charts/blob/master/stable/redis-ha/templates/redis-auth-secret.yaml

##

## redisPassword:

### redis-server-deployment.yaml

此YAML配置文件用于定义Redis master/slave的部署。

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

# Pay attention to the redis-role label at runtime. The self-determination logic in the image sets

# this value accordingly.

name: {{ template "redis-ha.fullname" . }}-server

labels:

name: {{ template "redis-ha.fullname" . }}-server

redis-node: "true"

{{ include "labels.standard" . | indent 4 }}

spec:

replicas: {{ .Values.replicas.servers }}

template:

metadata:

labels:

app: {{ template "redis-ha.name" . }}

release: {{ .Release.Name }}

component: server

name: {{ template "redis-ha.fullname" . }}-server

redis-node: "true"

spec:

serviceAccountName: {{ template "redis-ha.serviceAccountName" . }}

{{- if .Values.nodeSelector }}

nodeSelector:

{{ toYaml .Values.nodeSelector | indent 8 }}

{{- end }}

{{- if .Values.tolerations }}

tolerations:

{{ toYaml .Values.tolerations | indent 8 }}

{{- end }}

containers:

- name: redis

image: {{ .Values.redis\_image }}

resources:

{{ toYaml .Values.resources.server | indent 10 }}

env:

- name: REDIS\_SENTINEL\_SERVICE\_HOST

value: "redis-sentinel"

- name: REDIS\_CHART\_PREFIX

value: {{ template "redis-ha.fullname" . }}-

{{- if .Values.auth }}

- name: REDIS\_PASS

valueFrom:

secretKeyRef:

name: {{ template "redis-ha.fullname" . }}

key: auth

{{- end }}

ports:

- containerPort: 6379

volumeMounts:

- mountPath: /redis-master-data

name: data

volumes:

- name: data

### ****redis-master-service.yaml****

此YAML配置文件为定义了redis master的服务，此服务暴露6379端口，以供在集群中使用。

apiVersion: v1

kind: Service

metadata:

name: {{ template "redis-ha.fullname" . }}-master-svc

labels:

{{ include "labels.standard" . | indent 4 }}

annotations:

{{ toYaml .Values.servers.annotations | indent 4 }}

spec:

ports:

- port: 6379

protocol: TCP

targetPort: 6379

selector:

app: {{ template "redis-ha.name" . }}

release: "{{ .Release.Name }}"

redis-node: "true"

redis-role: "master"

type: "{{ .Values.servers.serviceType }}"

### ****redis-slave-service.yaml****

此YAML配置文件为定义了redis slave的服务，此服务暴露6379端口，以供在集群中使用。

apiVersion: v1

kind: Service

metadata:

name: {{ template "redis-ha.fullname" . }}-slave-svc

labels:

role: service

{{ include "labels.standard" . | indent 4 }}

annotations:

{{ toYaml .Values.servers.annotations | indent 4 }}

spec:

ports:

- port: 6379

protocol: TCP

targetPort: 6379

selector:

app: {{ template "redis-ha.name" . }}

release: "{{ .Release.Name }}"

redis-node: "true"

redis-role: "slave"

type: "{{ .Values.servers.serviceType }}"

### ****redis-sentinel-deployment.yaml****

此YAML文件定义Sentinel部署，Sentinel用于监控和管理对于Redis的访问。

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: {{ template "redis-ha.fullname" . }}-sentinel

labels:

{{ include "labels.standard" . | indent 4 }}

spec:

replicas: {{ .Values.replicas.sentinels }}

template:

metadata:

labels:

app: {{ template "redis-ha.name" . }}

release: {{ .Release.Name }}

component: sentinel

name: {{ template "redis-ha.fullname" . }}-sentinel

spec:

serviceAccountName: {{ template "redis-ha.serviceAccountName" . }}

{{- if .Values.nodeSelector }}

nodeSelector:

{{ toYaml .Values.nodeSelector | indent 8 }}

{{- end }}

{{- if .Values.tolerations }}

tolerations:

{{ toYaml .Values.tolerations | indent 8 }}

{{- end }}

containers:

- name: sentinel

image: {{ .Values.redis\_image }}

resources:

{{ toYaml .Values.resources.sentinel | indent 10 }}

env:

- name: SENTINEL

value: "true"

- name: REDIS\_CHART\_PREFIX

value: {{ template "redis-ha.fullname" . }}-

{{- if .Values.auth }}

- name: REDIS\_PASS

valueFrom:

secretKeyRef:

name: {{ template "redis-ha.fullname" . }}

key: auth

{{- end }}

ports:

- containerPort: 26379

### ****redis-sentinel-service.yaml****

此YAML文件用于在集群内容暴露Sentinel部署，以供其它应用访问和调用。

apiVersion: v1

kind: Service

metadata:

name: {{ template "redis-ha.fullname" . }}-sentinel

labels:

name: {{ template "redis-ha.name" . }}-sentinel-svc

role: service

{{ include "labels.standard" . | indent 4 }}

spec:

ports:

- port: 26379

targetPort: 26379

selector:

app: {{ template "redis-ha.name" . }}

release: "{{ .Release.Name }}"

redis-role: "sentinel"

### ****redis-serviceaccount.yaml****

如果rbac.create的值为true，此YAML文件将创建一个名为{{template “redis-ha.serviceAccountName”.}}的service account。

{{- if .Values.serviceAccount.create -}}

apiVersion: v1

kind: ServiceAccount

metadata:

name: {{ template "redis-ha.serviceAccountName" . }}

labels:

app: "redis-ha"

chart: {{ .Chart.Name }}-{{ .Chart.Version }}

heritage: {{ .Release.Service }}

release: {{ .Release.Name }}

{{- end -}}

### ****redis-role.yaml****

如果rbac.create的值为true，则此YAML文件将会定义名为{{template “redis-ha.fullname” .}}一个角色，此角色拥有获取、列示和修改pods的权限。

{{- if .Values.rbac.create -}}

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: Role

metadata:

name: {{ template "redis-ha.fullname" . }}

labels:

{{ include "labels.standard" . | indent 4 }}

rules:

- apiGroups:

- ""

resources:

- pods

verbs:

- get

- list

- patch

{{- end -}}

### ****redis-rolebinding.yaml****

如果rbac.create的值为true，将上述创建的service account和角色进行绑定。

{{- if .Values.rbac.create -}}

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: RoleBinding

metadata:

name: {{ template "redis-ha.fullname" . }}

labels:

{{ include "labels.standard" . | indent 4 }}

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: Role

name: {{ template "redis-ha.fullname" . }}

subjects:

- kind: ServiceAccount

name: {{ template "redis-ha.serviceAccountName" . }}

{{- end -}}

### ****redis-auth-secret.yaml****

如果auth的值为true，则会创建一个保密字典。

{{- if .Values.auth -}}

apiVersion: v1

kind: Secret

metadata:

name: {{ template "redis-ha.fullname" . }}

labels:

{{ include "labels.standard" . | indent 4 }}

type: Opaque

data:

{{- if .Values.redisPassword }}

auth: {{ .Values.redisPassword | b64enc | quote }}

{{- else }}

auth: {{ randAlphaNum 10 | b64enc | quote }}

{{- end }}

{{- end -}}

## Redis部署环境验证

在Kubernetes集群中，可以通过DNS名称{{ template “redis-ha.fullname” . }}.{{ .Release.Namespace }}.svc.cluster.local和端口6379访问redis集群。

如果设置了认证的话，通过下面的步骤连接Redis：

1）获取随机创建的redis密码：

echo $(kubectl get secret {{ template “redis-ha.fullname” . }} -o “jsonpath={.data[‘auth’]}” | base64 -D)

2）使用客户端连接Redis master Pod:

kubectl exec -it $(kubectl get pod -o jsonpath='{range .items[\*]}{.metadata.name} {.status.containerStatuses[0].state}{“\n”}{end}’ -l redis-role=master | grep running | awk ‘{print $1}’) bash

3）在容器内使用Redis CLI连接:

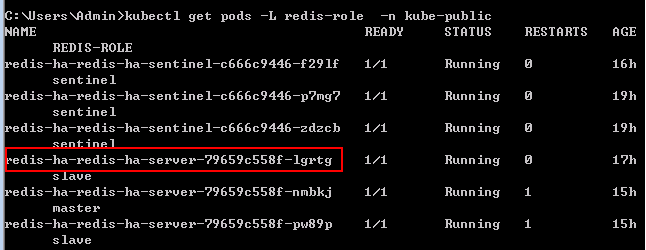
redis-cli -a <REDIS-PASS-FROM-SECRET>

如果未设置认证的话，通过下面的步骤连接Redis：

1）可以通过下面的命令运行Redis Pod，作为客户端：

获取当前系统中的Pods:

$ kubectl get pods -L redis-role --namespace=kube-public



以名称为redis-ha-redis-ha-server-79659c558f-lgrtg的Pod作为客户端：

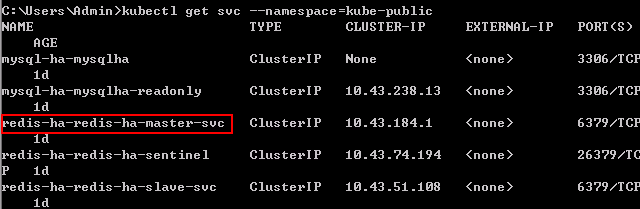
$ kubectl exec -it redis-ha-redis-ha-server-79659c558f-lgrtg --namespace=kube-public bash

https://www.kubernetes.org.cn/img/2018/05/redis_20180523085734.png

2）使用Redis CLI：

获取Redis的master服务名称：

$ kubectl get svc --namespace=kube-public



$ redis-cli -h redis-ha-redis-ha-master-svc.kube-public -p 6379

# 基于Promethues和Grafana的系统监控(2019-05-13 Y)

作为应用系统运行的底层平台，在Kubernetes上会运行大量的应用系统，同时又由于其自身的复杂性，因此需要提供一整套的监控解决方案，用于帮助管理员进行全面和系统化监控。系统监控为管理者提供对Kubernetes集群、宿主机、负载和容器的全方位监控，及时发现和处理问题，保证系统应用的正常运行。

## Prometheus介绍

Prometheus是一个开源的系统监视和警报工具包，自2012成立以来，许多公司和组织采用了Prometheus。它现在是一个独立的开源项目，并独立于任何公司维护。在2016年，Prometheus加入云计算基金会作为Kubernetes之后的第二托管项目，**Prometheus的关键特性如下：**

* 由指标名和键值对标识的时间序列数据的多维数据模型；
* 灵活的查询语言；
* 不依赖于分布式存储；单服务器节点是自治的；
* 通过HTTP上的拉模型实现时间序列收集；
* 通过中间网关支持推送时间序列；
* 通过服务发现或静态配置发现目标；
* 图形和仪表板支持的多种模式。

Prometheus生态由多个组件组成，并且这些组件大部分是可选的：

* **Prometheus服务器**，用于获取和存储时间序列数据；
* 仪表应用数据的客户端类库(**Client Library**)；
* 支持临时性工作的**推网关(Push Gateway)；**
* 特殊目的的**输出者(Exporter)**，提供被监控组件信息的 HTTP 接口，例如HAProxy、StatsD、MySQL、Nginx和Graphite等服务都有现成的输出者接口；
* 处理告警的**告警管理器（Alert Manager）；**
* 其它支持工具。

Prometheus从jobs获取指标数据，也直接或通过推送网关获取临时jobs的指标数据。它在本地存储所有被获取的样本，并在这些数据上运行规则，对现有数据进行聚合和记录新的时间序列，或生成警报。通过Grafana或其他API消费者，可以可视化的查看收集到的数据。下图显示了Pometheus的整体架构和生态组件：

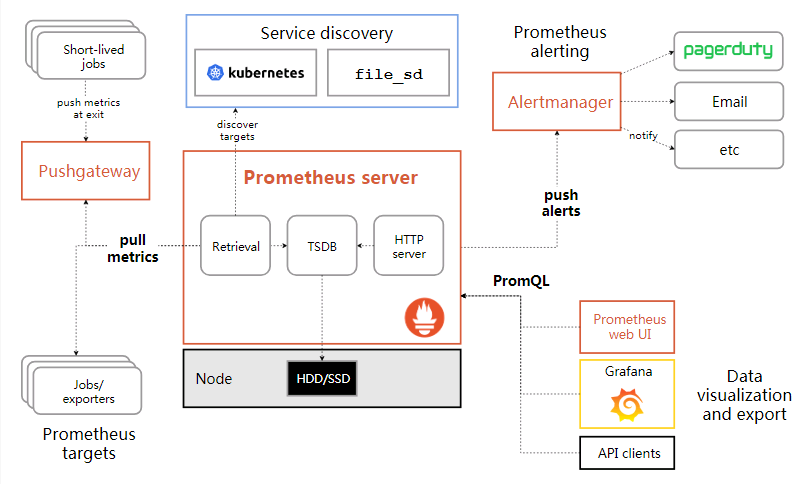


图15-1 Pormetheus整体架构图

Prometheus的整体工作流程：

1）Prometheus 服务器定期从配置好的 jobs 或者 exporters 中获取度量数据；或者接收来自推送网关发送过来的度量数据。

2）Prometheus 服务器在本地存储收集到的度量数据，并对这些数据进行聚合；

3）运行已定义好的 alert.rules，记录新的时间序列或者向告警管理器推送警报。

4）告警管理器根据配置文件，对接收到的警报进行处理，并通过email等途径发出告警。

5）Grafana等图形工具获取到监控数据，并以图形化的方式进行展示。

## Prometheus关键概念

### ****数据模型****

Prometheus从根本上将所有数据存储为时间序列：属于相同指标标准和同一组标注尺寸的时间戳值流。除了存储的时间序列之外，Prometheus可能会生成临时派生时间序列作为查询的结果。

* **指标名称和标签**：每个时间序列都是由指标标准名称和一组键值对（也称为标签）组成唯一标识。**指标名称**指定被测量的系统的特征（例如：http\_requests\_total-接收到的HTTP请求的总数）。它可以包含ASCII字母和数字，以及下划线和冒号。它必须匹配正则表达式[a-zA-Z\_:][a-zA-Z0-9\_:]\*。**标签**启用Prometheus的维度数据模型：对于相同指标标准名称，任何给定的标签组合都标识该指标标准的特定维度实例。查询语言允许基于这些维度进行筛选和聚合。更改任何标签值（包括添加或删除标签）都会创建新的时间序列。标签名称可能包含ASCII字母，数字以及下划线。他们必须匹配正则表达式[a-zA-Z\_][a-zA-Z0-9\_]\*。以\_\_开始的标签名称保留给供内部使用。
* **样本**：实际的时间序列，每个序列包括：一个 float64 的值和一个毫秒级的时间戳。
* **格式：**给定指标标准名称和一组标签，时间序列通常使用以下格式来标识：

<metric name>{<label name>=<label value>, ...}

例如，时间序列的指标名称为api\_http\_requests\_total，标签method=”POST”和handler=”/messages”，则标记为：

api\_http\_requests\_total{method="POST", handler="/messages"}

### ****指标类型****

Prometheus 客户端库主要提供Counter、Gauge、**Histogram和Summery**四种主要的 metric 类型：

* **Counter(计算器)：Counter是**一种累加的度量，它的值只能增加或在重新启动时重置为零。例如，您可以使用计数器来表示提供的请求数，已完成的任务或错误的数量。不要使用计数器来表达可减少的值。例如，不要使用Counter来计算当前正在运行的进程的数量，而是使用Gauge。
* **Gauge(测量)：**Gauge表示单个数值，表达可以任意地上升和下降的度量。Gauge通常用于测量值，例如温度或当前的内存使用情况，但也可以表达上升和下降的“计数”，如正在运行的goroutines的数量。
* **Histogram(**直方图)：Histogram样本观测（例如：请求持续时间或响应大小），并将它们计入配置的桶中。它也提供所有观测值的总和。具有<basename>基本度量标准名称的histogram的在获取数据期间会显示多个时间序列：
  + 观察桶的累计计数器，暴露为 <basename>\_bucket{le=”<upper inclusive bound>”}；
  + 所有观察值的总和，暴露为<basename>\_sum；
  + 已观察到的事件的计数，暴露为<basename>\_count（等同于<basename>\_bucket{le=”+Inf”}）。
* **Summery**：**类似于Histogram**，Summery样本观察（通常是请求持续时间和响应大小）。虽然它也提供观测总数和所有观测值的总和，但它计算滑动时间窗内的可配置分位数。在获取数据期间，具有<basename>基本度量标准名称的Summery会显示多个时间序列：
  + 流动φ分位数（0≤φ≤1）的观察事件，暴露为<basename>{quantile=”<φ>”}；
  + 所有观察值的总和，暴露为<basename>\_sum；
  + 已经观察到的事件的计数，暴露为<basename>\_count。

### ****工作和实例****

按照Prometheus的说法，可以获取数据的端点被称为*实例（instance）*，通常对应于一个单一的进程。具有相同目的的实例集合（例如为了可伸缩性或可靠性而复制的进程）称为*作业（job）*。

例如，具有四个复制实例的API服务器作业：

* 工作： api-server
  + 实例1： 1.2.3.4:5670
  + 实例2： 1.2.3.4:5671
  + 实例3： 5.6.7.8:5670
  + 实例4： 5.6.7.8:5671

当Prometheus获取目标时，它会自动附加一些标签到所获取的时间序列中，以识别获取目标：

* job：目标所属的配置作业名称。
* instance：<host>:<port>被抓取的目标网址部分。

如果这些标签中的任何一个已经存在于抓取的数据中，则行为取决于honor\_labels配置选项。

对于每个实例抓取，Prometheus会在以下时间序列中存储一个样本：

* up{job=”<job-name>”, instance=”<instance-id>”}：1 如果实例健康，即可达；或者0抓取失败。
* scrape\_duration\_seconds{job=”<job-name>”, instance=”<instance-id>”}：抓取的持续时间。
* scrape\_samples\_post\_metric\_relabeling{job=”<job-name>”, instance=”<instance-id>”}：应用度量标准重新标记后剩余的样本数。
* scrape\_samples\_scraped{job=”<job-name>”, instance=”<instance-id>”}：目标暴露的样本数量。
* up时间序列是实例可用性的监控。

## 使用Helm在Kubernetes中部署Prometheus

### 环境要求

* 已有Kubernetes 1.6+环境；
* 已部署helm客户端和tiller服务端（请参考：<https://docs.helm.sh/using_helm/#installing-helm>）：
* 在Kubernetes中创建了具备足够权限访问权限的service account；
* 并通过此service account在Kubernetes部署了tiller服务端（请参考：<https://docs.helm.sh/using_helm/#role-based-access-control>）。
* 在Kubernetes中提供2个容量大于10g的持久化存储卷。

### 通过Chart安装Prometheus

通过执行如下的命令，在Kubernetes中部署Prometheus：

$ helm install stable/prometheus --name=prometheus --namespace=kube-system

通过上述命令，将以默认的配置在Kubernetes中部署Prometheus。

## Grafana介绍和部署

### Grafana介绍

Grafana是一个开源的度量分析和可视化套件，它用于基础设施和应用分析的时间序列数据可视化。Grafana拥有种类丰富的图表，灵活的布局控制，强大的组织管理功能 (权限控制)，支持多种数据库作为数据源。社区提供了丰富的插件和APP 进行功能扩展。Grafana的主要概念如下：

* 数据源（Data Souce）

Grafana为时间序列数据（数据源）提供多种不同的存储后端。每个数据源都有一个特定的查询编辑器，可根据特定数据源公开的功能进行自定义。目前官方支持以下数据源：Graphite，InfluxDB，OpenTSDB，Prometheus，Elasticsearch，CloudWatch和KairosDB。每个数据源的查询语言和功能显然会非常不同，但可以将来自多个数据源的数据组合到单个仪表板上，每个面板都绑定到属于特定组织的特定数据源。

* 组织（Oragnization）

Grafana支持多个组织，以支持各种各样的部署模型，包括使用单个Grafana实例为多个可能不受信任的组织提供服务。在很多情况下，Grafana将与单一组织一起部署。每个组织可以有一个或多个数据源。所有仪表板都由特定组织拥有。注意：大多数度量数据库不提供任何类型的用户认证。因此，在Grafana中，数据源和仪表板可供特定组织的所有用户使用。

* 用户（User）

用户是Grafana中的指定帐户。用户可以属于一个或多个组织，并且可以通过角色分配不同级别的权限。Grafana支持多种内部和外部认证方式来对用户进行身份验证。包括来自自身的集成数据库，来自外部SQL服务器或来自外部LDAP服务器。

* 行（Row）

行是仪表板内的逻辑分隔符，用于将面板组合在一起。行总是12“单位”宽，这些单位会根据浏览器的水平分辨率进行自动缩放。也可以通过设置自己的宽度来控制一行中面板的相对宽度。利用单位抽象，可以让Grafana适应所有小型和巨大的屏幕。注意：无论您的分辨率或时间范围如何，借助MaxDataPoint功能，Grafana都可以显示完美的数据点数量。

* 面板（Panel）

面板是Grafana中的基本可视化构建模块。每个面板都提供了一个查询编辑器（取决于在面板中选择的数据源），通过利用查询编辑器可以将数据以可视化方式的显示在面板上，面板提供了各种各样的样式和格式选项。面板可以在仪表板上拖放和重新排列，他们也可以调整大小。目前有四种面板类型：Graph，Singlestat，Dashlist，Table和Text。

* 查询编辑器（Query Editor）

查询编辑器提供暴露数据源的功能，通过查询编辑器能够查询度量信息。使用查询编辑器可以在时间序列数据库中构建一个或多个查询（针对一个或多个系列）。可以在查询本身的查询编辑器中使用模板变量，这正方式提供了一种基于Dashboard上选定的模板变量动态探索数据的强大方法。Grafana允许通过查询编辑器所在的行来引用查询。如果将第二个查询添加到图形中，则只需输入#A即可引用第一个查询。这为构建复合查询提供了一种简单方便的方法。

* 仪表板（Dashboard）

仪表板是聚集所有可视化信息的地方，仪表板可以被认为是组织和排列成一个或多个行的一组或多组面板的集合。仪表板的时间段可以通过仪表板右上角的仪表板时间选择器(Dashboard time picker)来控制。仪表板可以利用模板(Templating)使它们更具动态性和交互性。仪表板可以利用注释(Annotations)在面板上显示事件数据。这有助于将面板中的时间序列数据与其他事件相关联。仪表盘（或特定面板）可以通过多种方式轻松共享。可以发送一个链接给能够登录Grafana用户。可以使用快照功能将当前正在查看的所有数据编码为静态和交互式JSON文档。仪表板可以被标记，仪表板选择器可以快速，可搜索地访问特定组织中的所有仪表板。

### 基于Helm在Kubernetes中部署Grafana

通过执行如下的命令，在Kubernetes中部署Grafana：

$ helm install stable/grafana --name=grafana --namespace=kube-system

通过上述命令，将以默认的配置在Kubernetes中部署Grfana。

## 监控Kubernetes实践

### 监控Kubernetes内容分析

对于Kubernetes来说，需要对集群内容相关资源运行的性能和健康状态这两类指标需要监控，这些资源主要包括：

* Node：主机节点
* Container：应用运行的容器
* Pod：一组容器的组合
* Deployment：无状态的应用部署
* StatefulSet：有状态的应用部署

### 在Grafana中配置数据源

登录Grafana，添加Prometheus类型的的数据源。登录Grafana的管理帐户为admin，密码从grafana保密字典的admin-password键的值中获取。其中，url地址：http://{prometheus\_server}.{namespace}:80，Access模式：proxy。

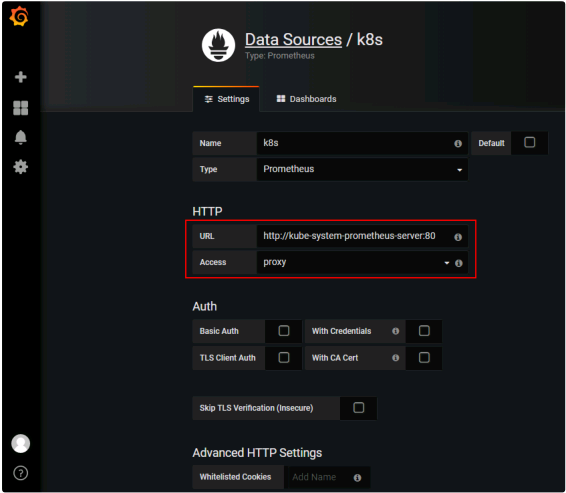


图15-2 Grafana配置页面

### 监控实践

在此文中以监控Pod和应用性能指标为例，展示Prometheus和Grafana对Kubernetes的监控实践。

1）对于Pod性能指标进行监控

进入Grafana的dashboard页面导入Kubernetes Pod Metrics模板 (<https://grafana.com/dashboards/747>)：

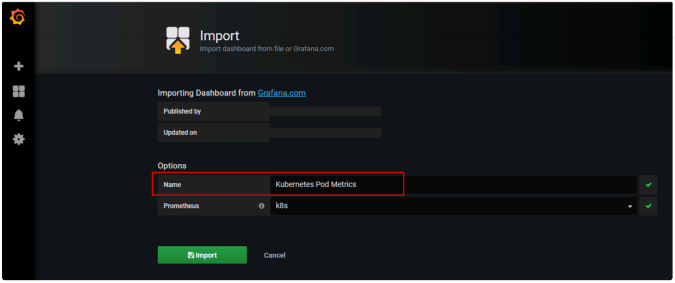


图15-3 导入Kubernetes Pod Metrics模板

导入Kubernetes Pod Metrics以后，就可以对Kubernetes中的Pod进行性能监控，包括网络I/O压力、CPU和内存使用情况等。

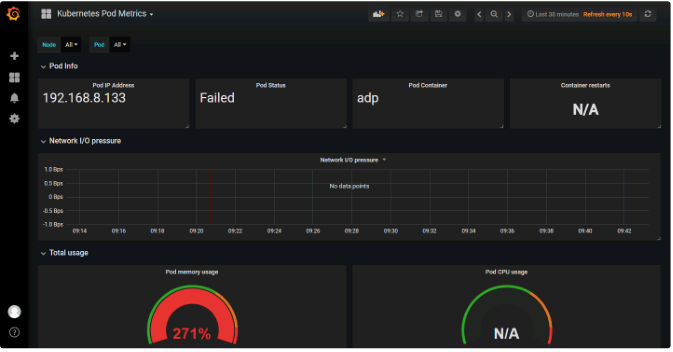


图15-4 Pod性能监控

2）对于应用性能指标进行监控

进入Grafana的dashboard页面导入Kubernetes App Metrics模板 (<https://grafana.com/dashboards/1471>)：

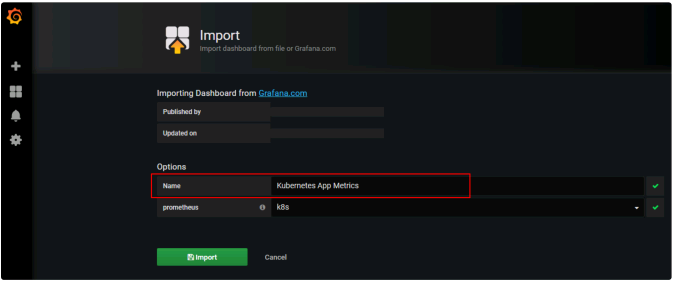


图15-5 导入Kubernetes App Metrics模板

导入Kubernetes App Metrics以后，就可以对Kubernetes中的应用进行性能监控，包括请求率、反应时间、使用Pod的数量、Pod使用情况等指标。

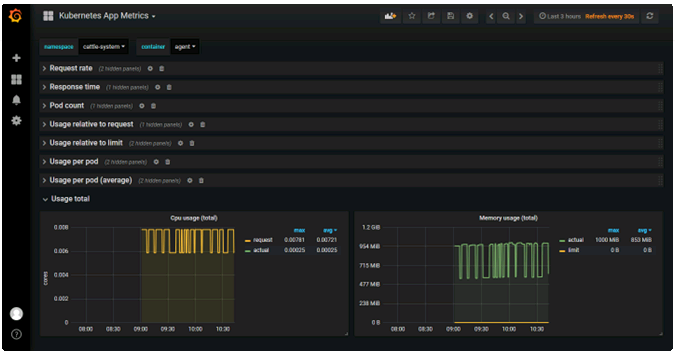


图15-6 应用性能监控

## Rancher中使用Prometheus和Grafana进行性能监控

在2.2.0版本以后，Rancher默认提供集群层面和项目层面的性能监控，所监控的指标主要为利用率、饱和度和错误率这三个方面。

### 集群性能监控

在集成层面，能够对整个集群、etcd、node和kubernetes组件等对象进行性能监控。

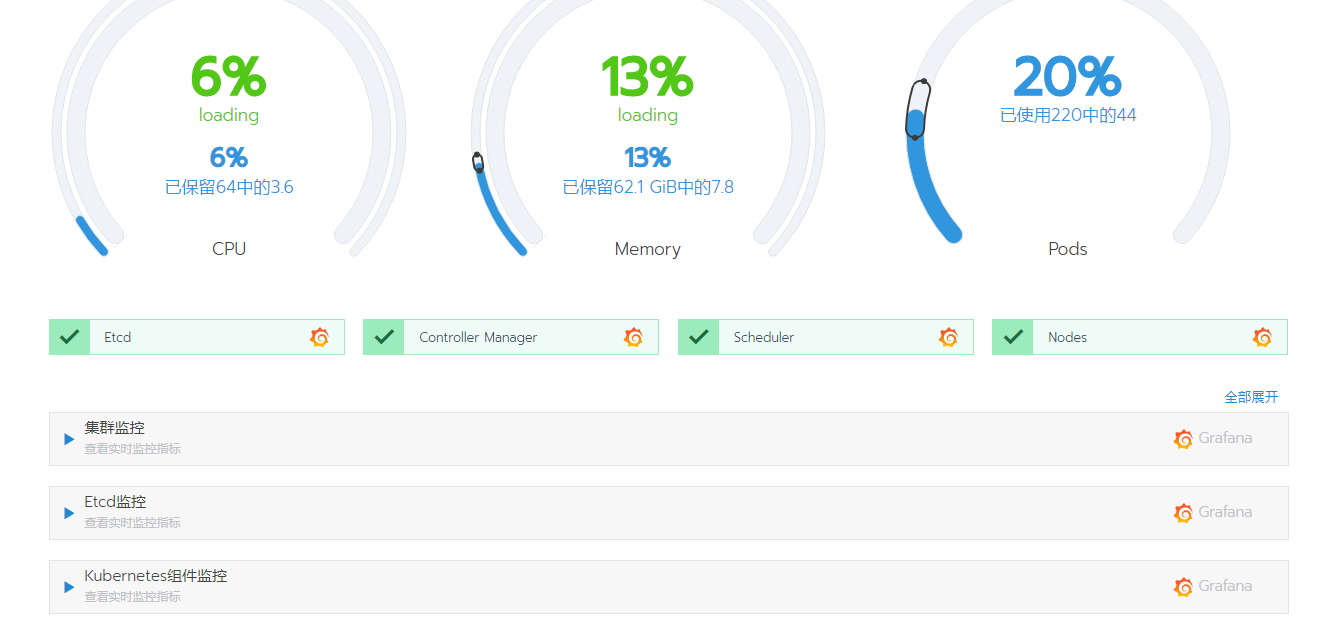


图15-7 集群性能监控

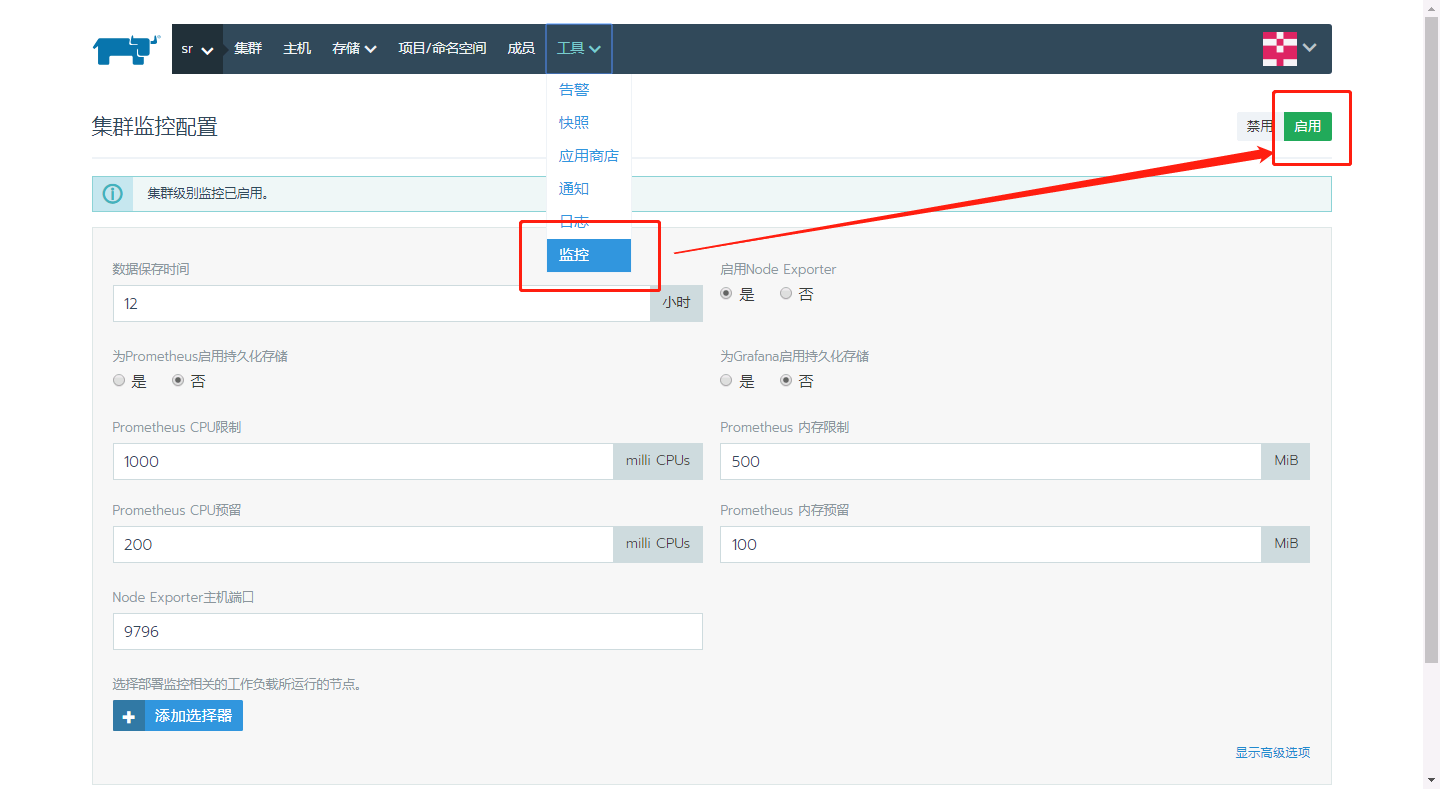
对于集群层面的性能监控，在特定集群的监控界面中进行启用。在启用监控时，需要设置监控数据保存时间、是否为Prometheus、Grafana启用持久化存储、设置Prometheus的CPU和内存的限制、以及为通过选择器为监控负载指定运行主机等信息

图15-8 启用监控

在监控启用后，能够对集群和主机节点的CPU使用率、CPU负载情况、内存使用率、磁盘使用率、磁盘I/O、网络数据包和网络I/O进行监控。集群管理者通过查询这些资源的使用情况，能够有效的监控整个集群的运行状态。

图15-9 对于集群和主机节点的性能监控

### 项目/命名空间性能监控

对于项目/命名空间层面的性能监控，在特定的项目/命名空间界面中进行启用。启用过程与集群下的监控相似，在启用监控时，需要设置监控数据保存时间、是否为Prometheus、Grafana启用持久化存储、设置Prometheus的CPU和内存的限制、以及为通过选择器为监控负载指定运行主机等信息。

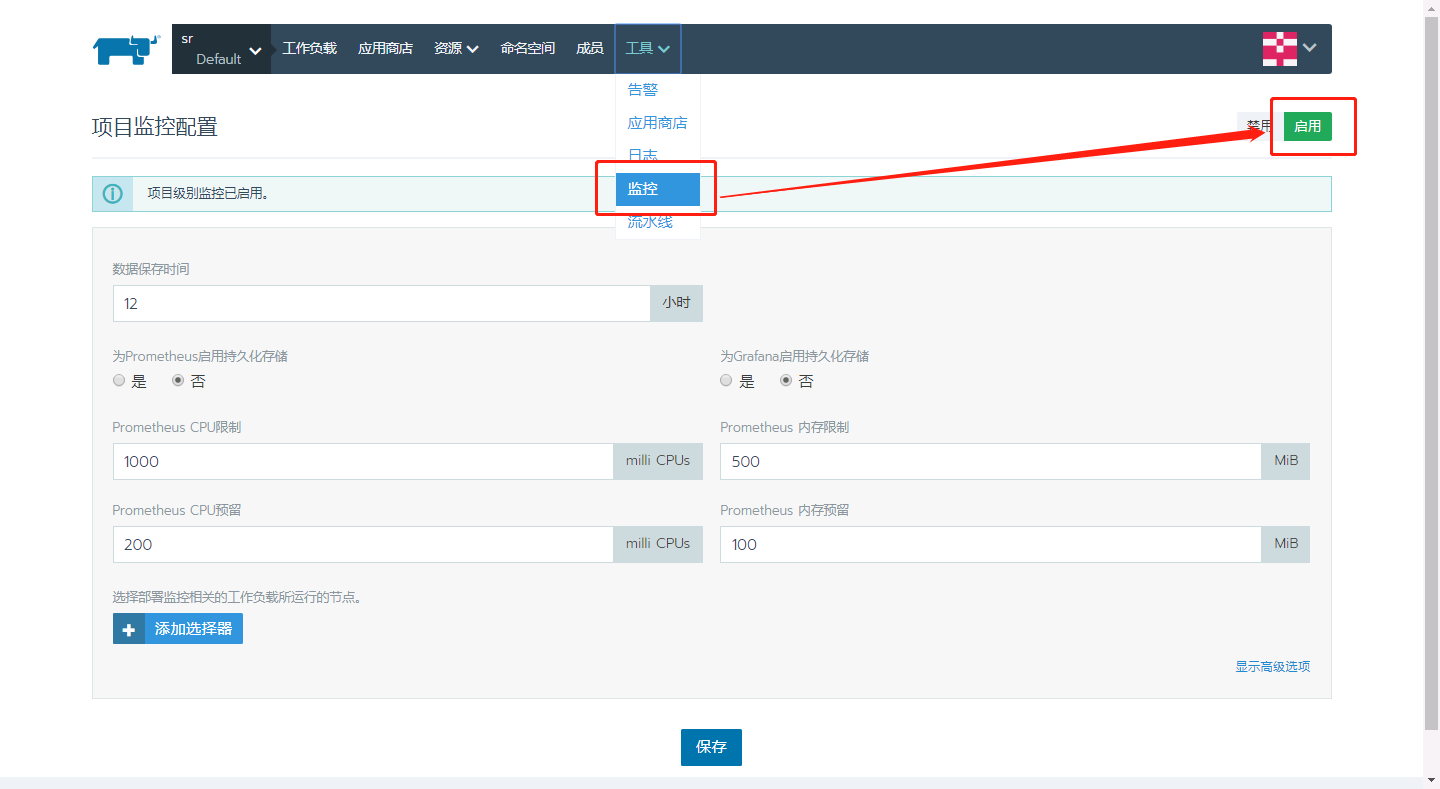


图15-10 启用监控

在项目/命名空间层面，能够对工作负载、Pod和容器等对象进行性能监控。对于工作负载和Pod来说，可以监控CPU、内存、网络数据包、网络I/O和磁盘I/O的资源使用情况。



图15-11 对工作负载、Pod和容器等对象进行性能监控

对于容器来说，将会监控CPU、内存和磁盘I/O的资源使用情况。

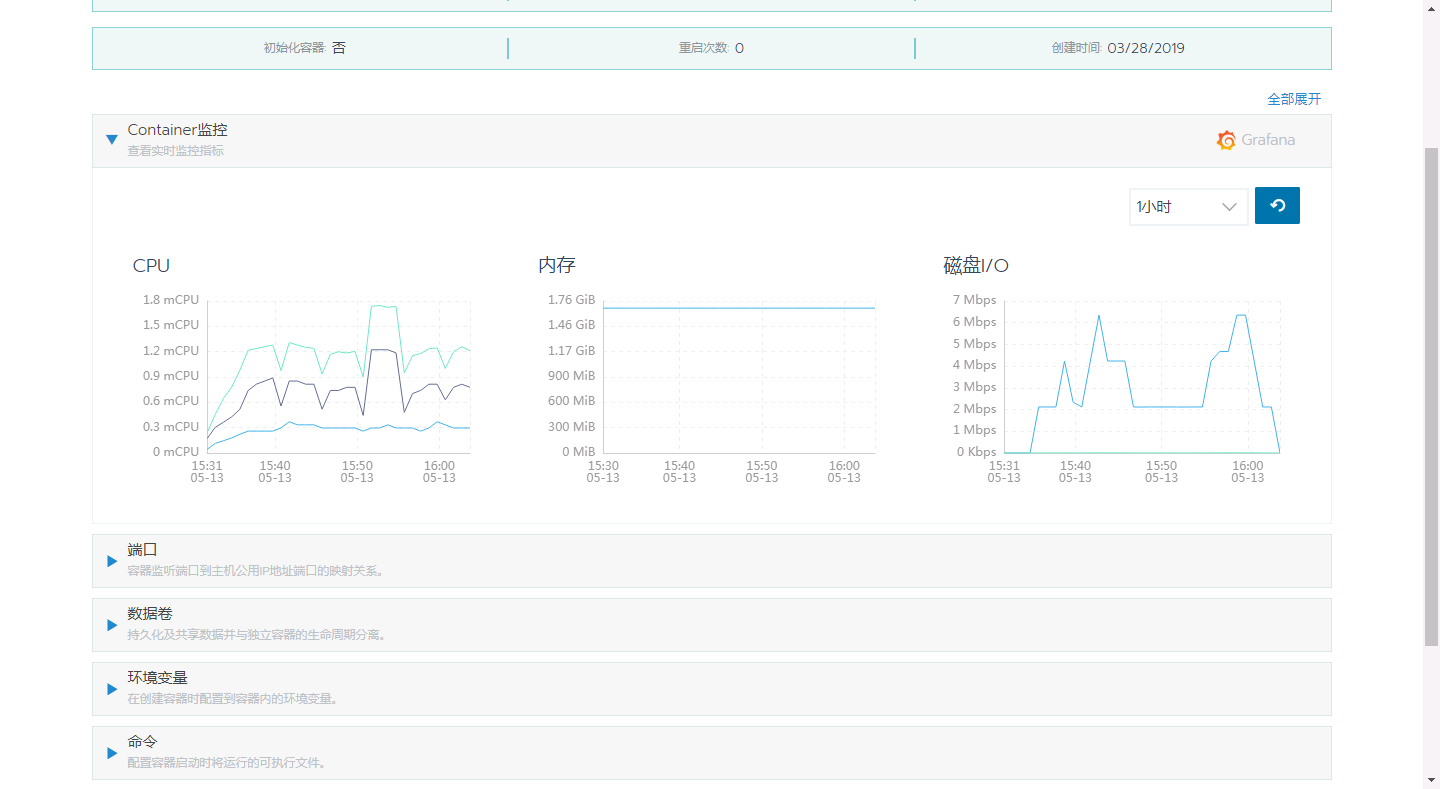


图15-12 对容器进行性能监控

# 基于client-go的Kubernetes定制开发指南

在现实应用过程中，可能由用户本身的实际需求其它原因，用户将需要对Kubernetes提供的功能进行定制开发和增加新的功能。在本章节中，首先阐述了如何安装和部署开发环境；并在部署好开发环境的基础上，提供了基于client-go的开发示例。

## 安装和配置开发环境

在本章节中，介绍的是基于client-go对Kubernetes进行开发。操作系统为Window 7，Kubernetes的版本为v1.13。为搭建此开发环境，需要安装和部署kubectl、go和client-go。

### 安装kubectl

Kubectl是用于与容器云进行交互的命令行客户端工具，本文将kubectl安装在Windows操作系统中。

1）安装部署kubectl

拷贝kubectl.exec可执行文件到特定的文件夹目录下，并将kubectl.exe所在的文件夹目录地址添加至Windows的环境变量的Path中。

2）配置kubeconfig文件

在当前用户的文件夹目录下，创建./kube文件夹，并创建config文件。rancher系统中，进入所创建的集群的主页，通过点击“Kube config File”进入kubeconfig信息页面。并将kubeconfig文件的内容拷贝到~/.kube/config中。

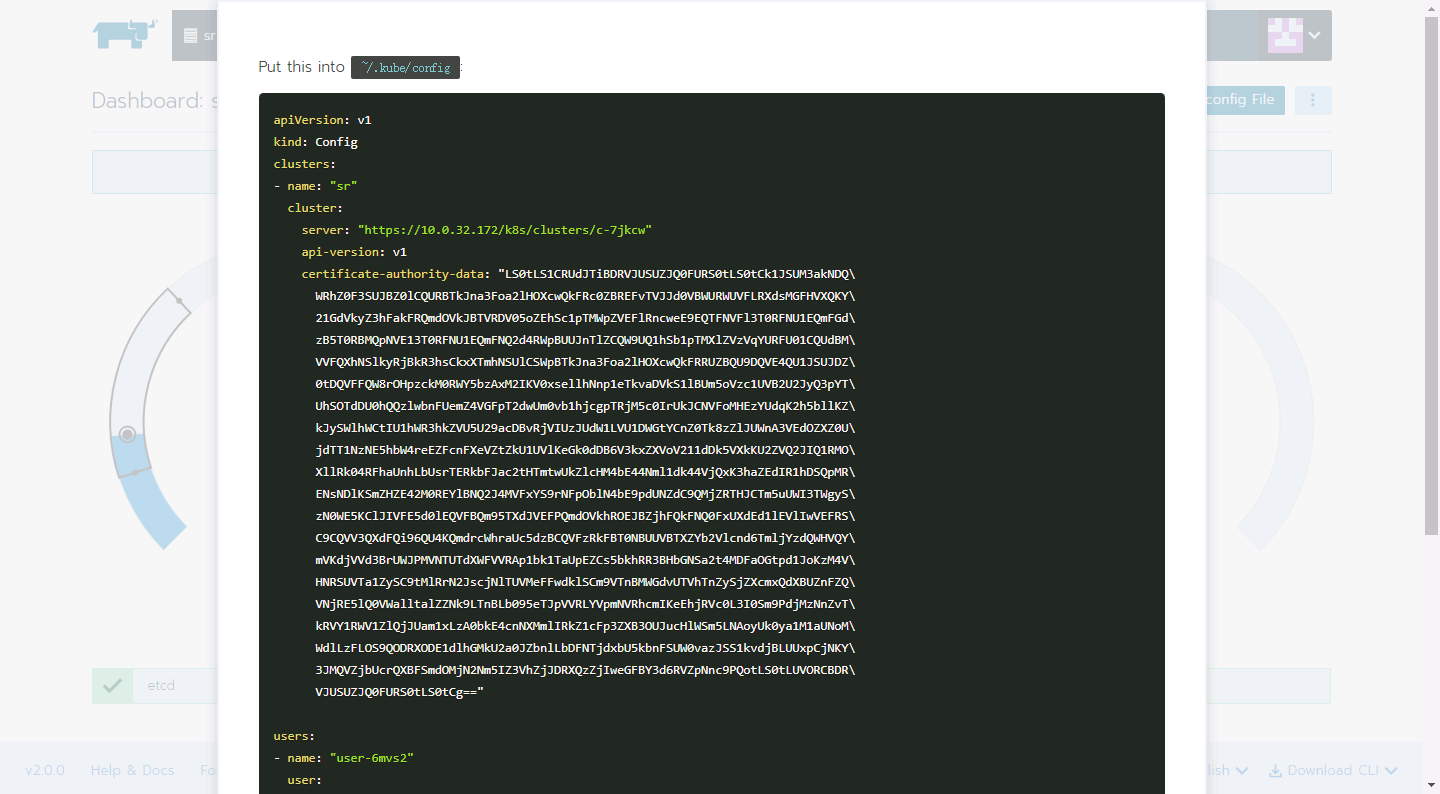


图16-1 kubeconfig文件信息

**3）验证**

通过执行如下的命令，验证kubectl配置是否成功：

$ kubectl get nodes

### 安装go-client

在部署好kubectl命令行工具后，接下来要安装go-client开发环境，这里首先对go-client进行了整体简介，随后展示了如何安装部署go-client开发环境。

#### go-client简介

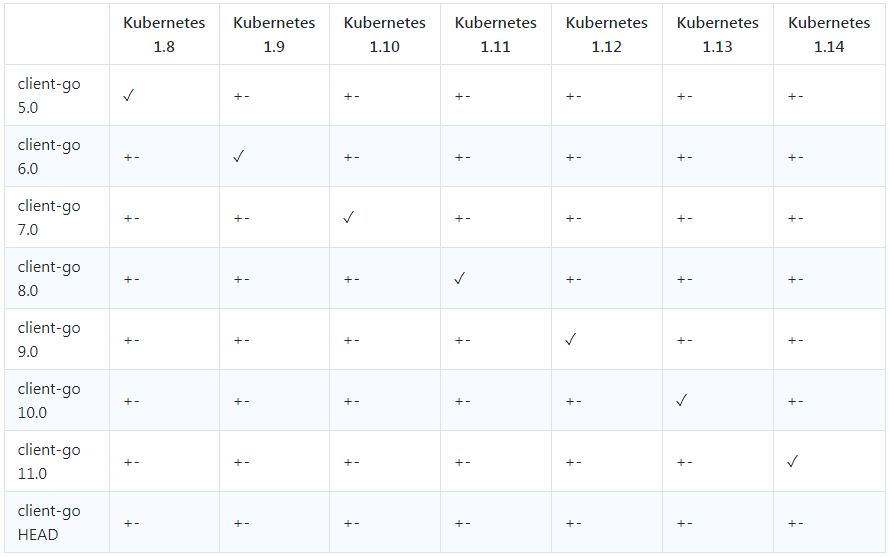
client-go是与Kubernetes进行通信的一种客户端，在client-go中有的三类 client。

* **Clientset：**Clientset 是调用Kubernetes最常用的 client，通过它能够找到 kubernetes所有原生资源对应的 client。 获取方式一般是，指定 group 然后指定特定的 version，然后根据 resource 名字来获取到对应的 client。
* **DynamicClient：**Dynamic client 是一种动态的 client，它能处理 kubernetes 所有的资源。不同于 clientset，dynamic client 返回的对象是一个 map[string]interface{}，如果一个 controller 中需要控制所有的 API，可以使用dynamic client，目前它在了 garbage collector 和 namespace controller中被使用。
* **RESTClient：**RESTClient 是 clientset 和 dynamic client 的基础，前面这两个 client 本质上都是 RESTClient，它提供了一些 RESTful 的函数如 Get()，Put()，Post()，Delete()。

如何选择 Client 的类型呢?

* 如果 Controller 只是需要控制 Kubernetes 原生的资源，如 Pods，Nodes，Deployments等，那么 Clientset 就够用了。
* 如果需要使用第三方的资源来拓展 Kubernetes 的 API，那么就需要使用 Dynamic Client 或 RESTClient。
* 需要注意的是，Dynamic Client 目前只支持 JSON 的序列化和反序列化。

当前，client-go和Kubernetes之间的版本对应关系如下：



说明：

* ✓：client-go和Kubernetes版本中的功能/ API对象完全相同。
* +：client-go具有可能不存在于Kubernetes集群中的功能或API对象，这可能是因为client-go具有其他新的API，或者服务器已删除的旧API。但是，它们共有的一切（即大多数API）都可以使用。请注意，alpha API可能会在单个版本中消失或发生重大变化。
* -：Kubernetes集群具有client-go库无法使用的功能，这可能是由于服务器具有其他新API，或者客户端已删除旧API。但是，它们共有的所有内容（即大多数API）都可以使用。

#### 安装部署

在安装部署client-go之前，需要保证已经安装了go，这里在windows操作系统下安装go 1.12.5版本，依赖关系使用Glide进行管理。

##### 安装client-go

在命令行中执行如下的命令，进行client-go的安装：

$ go get k8s.io/client-go/...

##### Glide介绍

在本文中go的依赖关系通过Glide进行管理，通过Glide将管理/vendor目录。Glide支持语义化版本；支持Git、Svn配置管理工具等；支持Go工具链；支持vendor目录；支持从Godep、GB、GPM、Gom倒入；支持私有的Repos和Forks。

Glide扫描应用程序或库的源代码以确定所需的依赖。Glide通过读取glide.yaml文件来确定依赖的版本和位置。通过这些信息，Glide会获取所需的依赖。遇到依赖包时，会扫描导入以确定依赖关系的依赖关系（传递依赖关系）。如果依赖项目包含一个glide.yaml文件，确定依赖规则中各依赖的位置和版本。此文件还可以确定导入了Godep，GB，GOM和GPM的配置。当go工作发现和使用Glide时，依赖将会被导出到vendor/目录下。glide.lock文件包含所有的依赖，以及传递。通过glide init命令能够安装新项目，通过glide update能够使用扫描和规则重新生成依赖版本，glide install会安装在glide.lock文件列示的版本。glide.yaml文件目录结构如下所示：

- $GOPATH/src/myProject (Your project)

|

|-- glide.yaml

|

|-- glide.lock

|

|-- main.go (Your main go code can live here)

|

|-- mySubpackage (You can create your own subpackages, too)

| |

| |-- foo.go

|

|-- vendor

|-- github.com

|

|-- Masterminds

|

|-- ... etc.

通过执行如下的命令安装Glide：

$ go get github.com/Masterminds/glide

在安装成功后，可以通过下面的名称确认Glide是否部署成功：

$ glide -v

Glide的核心使用命令如下：

$ glide create # 初始化一个新项目，并创建一个glide.yaml文件

$ open glide.yaml # 编辑glide.yaml文件

$ glide get github.com/Masterminds/cookoo # 获取包，并将其添加至glide.yaml

$ glide install # 安装包和依赖，从glide.lock文件安装特定的版本

$ go build # Go tools work normally

$ glide up # 下载或更新glide.yaml文件中列出的所有库，并将它们放在vendor目录中。

##### 通过Glide管理依赖关系

避免使用Glide的许多子命令，最好直接修改Glide的清单文件（glide.yaml），然后执行*glide updeate –strip-vendor*进行更新。

首先，在项目的根目录(这里的根目录为bjsasc.com/demo)下创建一个glide.yaml文件：

package: bjsasc.com/demo

import:

- package: k8s.io/client-go

version: v7.0.0

接着，在下面中添加一个Go文件，此文件导入client-go，否则将不会导入client-go依赖到下面的vendor/目录下。

## 开发示例

在本部分用于向读者展示一个基于client-go的开发示例，内容包括示例代码和示例的运行这两部分。以此示例为基础，后续读者可以进行安装自己的需要进行定制开发。

### 示例介绍

在“%GOPATH%\src\bjsasc.com\demo”目录下创建一个main.go文件，main.go(通过此链接获取：<https://github.com/kubernetes/client-go/tree/master/examples/create-update-delete-deployment>)的代码如下所示，此代码的示例演示如何通过client-go在Kubernetes中创建、更新和删除部署。

/\*

Copyright 2017 The Kubernetes Authors.

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

\*/

// Note: the example only works with the code within the same release/branch.

package main

import (

        "bufio"

        "flag"

        "fmt"

        "os"

        "path/filepath"

        appsv1 "k8s.io/api/apps/v1"

        apiv1 "k8s.io/api/core/v1"

        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"

        "k8s.io/client-go/kubernetes"

        "k8s.io/client-go/tools/clientcmd"

        "k8s.io/client-go/util/homedir"

        "k8s.io/client-go/util/retry"

        // Uncomment the following line to load the gcp plugin (only required to authenticate against GKE clusters).

        // \_ "k8s.io/client-go/plugin/pkg/client/auth/gcp"

)

func main() {

        var kubeconfig \*string

        if home := homedir.HomeDir(); home != "" {

kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"), "(optional) absolute path to the kubeconfig file")

} else {

              kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig file")

        }

        flag.Parse()

        config, err := clientcmd.BuildConfigFromFlags("", \*kubeconfig)

        if err != nil {

               panic(err)

        }

        clientset, err := kubernetes.NewForConfig(config)

        if err != nil {

               panic(err)

        }

        deploymentsClient := clientset.AppsV1().Deployments(apiv1.NamespaceDefault)

//定义部署对象

        deployment := &appsv1.Deployment{

               ObjectMeta: metav1.ObjectMeta{

                       Name: "demo-deployment",

               },

               Spec: appsv1.DeploymentSpec{

                       Replicas: int32Ptr(2),

                       Selector: &metav1.LabelSelector{

                               MatchLabels: map[string]string{

                                      "app": "demo",

                               },

                       },

                       Template: apiv1.PodTemplateSpec{

                               ObjectMeta: metav1.ObjectMeta{

                                      Labels: map[string]string{

                                              "app": "demo",

                                      },

                               },

                               Spec: apiv1.PodSpec{

                                      Containers: []apiv1.Container{

                                              {

                                                     Name:  "web",

                                                     Image: "nginx:1.12",

                                                     Ports: []apiv1.ContainerPort{

                                                             {

                                                                     Name:          "http",

                                                                     Protocol:      apiv1.ProtocolTCP,

                                                                     ContainerPort: 80,

                                                             },

                                                     },

                                              },

                                      },

                               },

                       },

               },

        }

// 创建部署

        fmt.Println("Creating deployment...")

        result, err := deploymentsClient.Create(deployment)

        if err != nil {

               panic(err)

        }

        fmt.Printf("Created deployment %q.\n", result.GetObjectMeta().GetName())

// 更新部署

        prompt()

        fmt.Println("Updating deployment...")

        //    You have two options to Update() this Deployment:

        //

        //    1. Modify the "deployment" variable and call: Update(deployment).

        //       This works like the "kubectl replace" command and it overwrites/loses changes

        //       made by other clients between you Create() and Update() the object.

        //    2. Modify the "result" returned by Get() and retry Update(result) until

        //       you no longer get a conflict error. This way, you can preserve changes made

        //       by other clients between Create() and Update(). This is implemented below

        //                     using the retry utility package included with client-go. (RECOMMENDED)

        //

        // More Info:

        // https://github.com/kubernetes/community/blob/master/contributors/devel/api-conventions.md#concurrency-control-and-consistency

        retryErr := retry.RetryOnConflict(retry.DefaultRetry, func() error {

               // Retrieve the latest version of Deployment before attempting update

               // RetryOnConflict uses exponential backoff to avoid exhausting the apiserver

               result, getErr := deploymentsClient.Get("demo-deployment", metav1.GetOptions{})

               if getErr != nil {

                       panic(fmt.Errorf("Failed to get latest version of Deployment: %v", getErr))

               }

        result.Spec.Replicas = int32Ptr(1)

// 减少副本数量

               result.Spec.Template.Spec.Containers[0].Image = "nginx:1.13" // change nginx version

               \_, updateErr := deploymentsClient.Update(result)

               return updateErr

        })

        if retryErr != nil {

               panic(fmt.Errorf("Update failed: %v", retryErr))

        }

        fmt.Println("Updated deployment...")

        // 列示部署

        prompt()

        fmt.Printf("Listing deployments in namespace %q:\n", apiv1.NamespaceDefault)

        list, err := deploymentsClient.List(metav1.ListOptions{})

        if err != nil {

               panic(err)

        }

        for \_, d := range list.Items {

               fmt.Printf(" \* %s (%d replicas)\n", d.Name, \*d.Spec.Replicas)

        }

        // 删除部署

        prompt()

        fmt.Println("Deleting deployment...")

        deletePolicy := metav1.DeletePropagationForeground

        if err := deploymentsClient.Delete("demo-deployment", &metav1.DeleteOptions{

               PropagationPolicy: &deletePolicy,

        }); err != nil {

               panic(err)

        }

        fmt.Println("Deleted deployment.")

}

func prompt() {

        fmt.Printf("-> Press Return key to continue.")

        scanner := bufio.NewScanner(os.Stdin)

        for scanner.Scan() {

               break

        }

        if err := scanner.Err(); err != nil {

               panic(err)

        }

        fmt.Println()

}

func int32Ptr(i int32) \*int32 { return &i }

### 运行示例

在glide.yaml文件所在的目录中运行如下的命令：

glide update --strip-vendor

也可以使用下面的命令：

glide up -v

执行上述命令后，k8s.io/client-go和k8s.io/apimachinery都会被添加至下面的vendor/命令下。

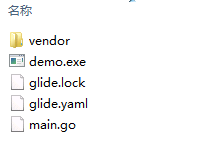
确保具有一个Kubernetes集群，并以及进行了kubectl配置：

$ kubectl get nodes

通过执行下面的命令编译此示例：

$ cd bjsasc.com/demo

$ go build



现在，使用本地kubeconfig文件在工作站上运行此应用程序：

$ demo.exe

# or specify a kubeconfig file with flag

$ demo.exe -kubeconfig=$HOME/.kube/config

运行此命令将在群集上执行以下操作：

* **创建部署：**这将创建副本数为2部署。验证 kubectl get pods。
* **更新部署：**这将通过将副本计数设置为1并将容器映像更改为更新在上一步中创建的部署资源nginx:1.13。建议您检查处理冲突的重试循环。使用确认新的副本计数和容器图像 kubectl describe deployment demo。
* **回滚部署：**这将将部署回滚到上一版本。在这种情况下，它是在步骤1中创建的修订版。kubectl describe用于验证容器图像现在nginx:1.12。另请注意，部署的副本计数仍为1; 这是因为当且仅当部署的pod模板（.spec.template）发生更改时才会创建部署修订。
* **列表部署：**这将检索default 命名空间中的部署并打印其名称和副本计数。
* **删除部署：**这将删除Deployment对象及其相关的ReplicaSet资源。验证kubectl get deployments。

每个步骤都由交互式提示分隔。您必须 Return按键才能继续下一步。您可以使用这些提示作为中断来花时间运行kubectl并检查执行的操作的结果。

您应该看到如下输出：

1）创建一个名称为“demo-deployment”的部署

Creating deployment...

Created deployment "demo-deployment".

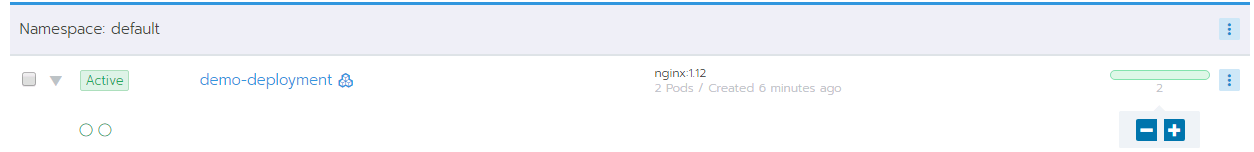


图16-2 应用部署结果

2）更新部署的版本

-> Press Return key to continue.

Updating deployment...

Updated deployment...

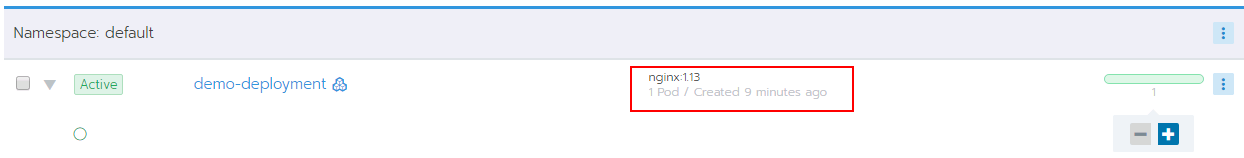


图16-3 应用更新结果

3）回滚部署

-> Press Return key to continue.

Rolling back deployment...

Rolled back deployment...

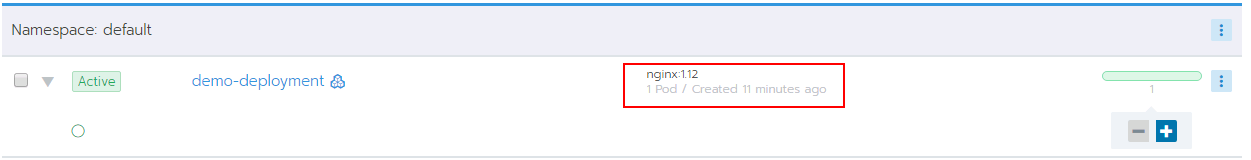


图16-4 应用回滚结果

4）以列表的形式展示“default”命名空间下的部署

-> Press Return key to continue.

Listing deployments in namespace "default":

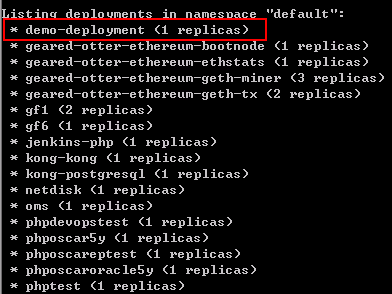


图16-5 default命名空间下的应用列表

5）删除“demo-deployment”部署

-> Press Return key to continue.

Deleting deployment...

Deleted deployment.

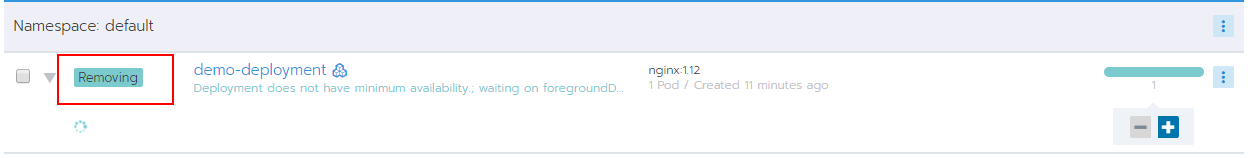


图16-5 应用部署删除过程

# 管理控制台备份和恢复

在这里Rancher Server作为Kubernetes的管理控制台，此节点是单点安装的，因此需要对其进行备份，以防止由于Rancher Server出现问题时无法使用Kubernetes集群。

## 备份

本书所介绍的Kubernetes集群是通过Rancher安装的，为了保证在Rancher服务发生故障时，或者需要升级Rancher到新版本时，能够保留目前的数据，则需要对Rancher进行数据备份。这样的话，在Rachenr发生问题，或者升级过程中遇到问题时，就可以通过备份进行数据恢复。

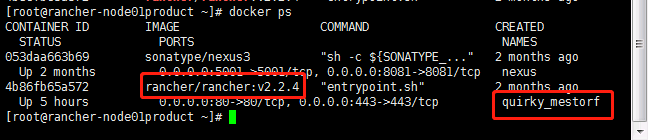
* 备份准备

在进行备份之前，需要提前准备下面的信息。

|  |  |  |
| --- | --- | --- |
| 项目 | 例子 | 说明 |
| <RANCHER\_CONTAINER\_TAG> | v2.2.4 | 初始安装Rancher时，所使用的rancher/rancher镜像版本。 |
| <RANCHER\_CONTAINER\_NAME> | quirky\_mestorf | Rancher容器名称 |
| <DATE> | 2019-08-25 | 创建备份的时间 |

在安装Rancher服务的宿主机上，通过执行下面的命令查询<RANCHER\_CONTAINER\_TAG>、<RANCHER\_CONTAINER\_NAME>和<RANCHER\_CONTAINER\_ID>。

docker ps



* 创建备份

1. 停止Rancher Server容器

停止当前运行Rancher Server的容器，此处<RANCHER\_CONTAINER\_ID>为4b86fb65a572：

docker stop 4b86fb65a572

1. 创建数据卷容器

将当前Rancher Server容器的数据备份到名为*rancher-data-20190825的*数据卷容器中：

docker create --volumes-from quirky\_mestorf --name rancher-data-20190825 rancher/rancher:v2.2.4

1. 创建数据卷容器备份

在进行Rancher Server升级期时，新的容器需要链接到数据卷容器，并且会对数据卷容器中的数据进行更新或更改。因此，在这里需要提前对数据卷容器进行备份，以备升级失败时，用于数据回滚。

docker run --volumes-from rancher-data-20190825 -v $PWD:/backup alpine tar cvf /backup/rancher-data-backup-v2.2.4-20190825.tar /var/lib/rancher

1. 重启Rancher Server容器

docker restart 4b86fb65a572

## 恢复

当Rancher Server服务发问题时，则可以通过前面备份的

1. 停止Rancher Server容器

通过下面的命令停止容器：

docker stop quirky\_mestorf

1. 恢复

复制备份文件到Rancher Server主机上，通过cd命令切换到压缩文件所在的目录，并执行以下命令：

**警告!**此命令将从Rancher Server容器中删除所有数据。

docker run --volumes-from quirky\_mestorf

-v $PWD:/backup alpine sh -c "rm /var/lib/rancher/\* -rf && tar xvf /backup/rancher-data-backup-v2.2.4-20190825.tar"

1. 重启Rancher Server容器

docker restart quirky\_mestorf

# 问题定位和处理

由于Kubernetes的定位和复杂性，因此在企业实施Kubernetes时就会遇到各种问题。在本章节，主要阐述应用部署时的问题定位和处理，以及docker守护程序的问题定位和处理。

## 应用部署问题定位和处理(2019-05-22 Y)

相对于传统的部署模式，容器化后的应用部署到Kubernetes中的过程有着自己的特殊或者说复杂性。

### 应用部署问题处理的整体思路

在将容器化的应用部署到Kubernetes集群中，可能会出现各种问题。根据Kubernetes的架构设计原理，容器化应用对外提供服务出现的主要问题在三个点上：

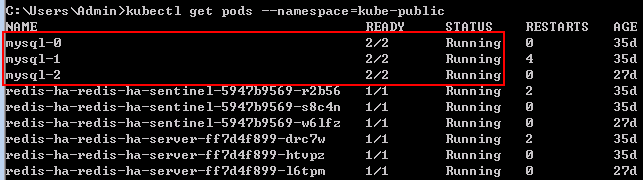
1. **应用本身的问题**：此问题为应用本身的问题，不在此文中进行详细的阐述；
2. **作为容器化应用逻辑主机的Pod的问题**：此部分的问题主要涉及到容器化应用是否在容器云中正常部署和运行，这里会涉及到CPU、内存、存储资源等问题；
3. **代理容器化应用服务的问题**：第三方服务或用户会通过代理服务访问容器化应用，如果代理服务存在问题，则容器云应用将无法对外提供服务能力，这里会涉及到服务是否存在、DNS解析是否正确等问题。

在本文中，以部署的高可用MySQL（请参考：[部署高可用的mysql（2019-5-22 Y）](#_部署高可用的mysql（2019-5-22_Y）)）为例展示如何进行问题定位和处理。另外为了能够在Kubernetes集群外访问MySQL数据库，对外暴露了MySQL master的NodePort类型服务，服务名称为mysql-0-svc。

### 调试Pods

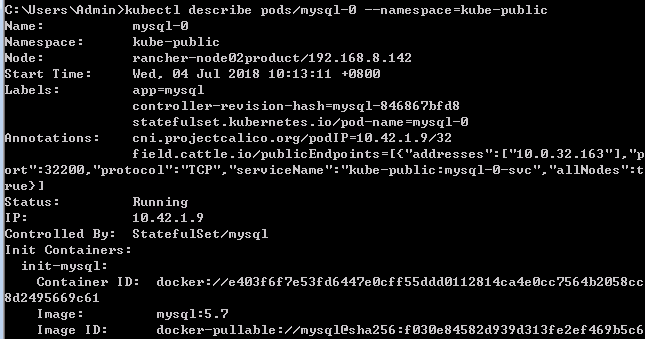
在调试Pod之前，通过kubectl get pods命令查看一下Pod的运行状态。

$ kubectl get pods --namespace=kube-public



对于特定的Pod，可以通过kubectl describe pods命令查看详细的信息。

$ kubectl describe pods/mysql-0 --namespace=kube-public



在Pod的生命周期中，有如下的几个状态：

* Pending: Pod已经被Kubernetes系统接受，但是还有一个或者多个容器镜像未被创建。这包括Pod正在被调度和从网络上下载镜像的时间。
* Running: Pod已经被绑定到了一个Node，所有的容器也已经被创建。至少有一个容器已经在运行，或者在启动或者重新启动的过程中。
* Succeeded: 在Pod中的所有的容器都已经被成功的终止，并且不会再重启。
* Failed: 在Pod中所有容器都已经被终止，并且至少有一个容器是非正常终止的。即，容器以非零状态退出或者被系统强行终止的。
* Unknown: 由于某些原因，Pod不能被获取，典型的情况是在与Pod的主机进行通信中发生了失败。
* Waiting：由于某些原因，Pod已被调度到了Node节点上，但无法正常运行。
* Crashing：由于某些原因，Pod处于崩溃状态。

根据Pod所处的状态，相应的处理方式不同。

#### Pod处于待命（Pending）状态

如果Pod被卡在待命（Pending）状态，则意味着它无法被安排到Node节点上。造成这种情况通常因为某种类型的资源不足，从而导致Pod无法被调度。通过查看kubectl describe …命令的输出内容，应该有为什么Pod无法被调度的原因。这些原因包括：

* **没有足够的资源**：集群中的CPU或内存可能已经耗尽了，在这种情况下，需要删除Pod，调整资源请求或向集群中添加新的Node节点。
* **正在使用hostPort**：将Pod绑定到了数量有限的hostPort。在大多数情况下，没有必要使用hostPort，可以尝试使用服务来暴露Pod。如果确实需要hostPort，那么只能调度与Kubernetes集群中的节点一样多的Pod。

#### Pod处于等待（Waiting）状态

如果Pod处于等待（Waiting）状态，则它已被调度到一个工作Node上，但它无法在该Node上运行。同样，通过kubectl describe …应该是能够获取有用的信息。处于等待（Waiting）状态的最常见的原因是**无法拉取镜像**。有三件事需要检查：

* 确保镜像名称正确无误。
* 确认镜像仓库中是否存在此镜像？
* 在机器上，运行docker pull <image>命令，查看是否可以拉取镜像。

#### Pod崩溃（Crashing）或其他不健康

首先，通过执行kubectl logs ${POD\_NAME} ${CONTAINER\_NAME}查看当前容器的日志：

$ kubectl logs mysql-0 mysql --namespace=kube-public

如果容器之前已崩溃，可以使用以下命令访问上一个容器的崩溃日志：

$ kubectl logs --previous mysql-0 mysql --namespace=kube-public

或者，也可以使用kubectl exec在该容器内运行命令：

$ kubectl exec ${POD\_NAME} -c ${CONTAINER\_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}

请注意，这-c ${CONTAINER\_NAME}是可选的，对于仅包含单个容器的Pod，可以省略。如果这些方法都不起作用，可以找到运行该pod的主机，并通过SSH连接到该主机。

#### Pod正在运行，但没有按照要求执行

如果Pod没有按预期运行，可能是Pod描述中存在错误，并且在创建Pod时忽略了该错误。通常可能是，Pod描述的一部分嵌套不正确，或键名不正确，因此键被忽略。例如，如果拼写错误command，commnd则将创建Pod，但不会按照希望使用命令行。

首先，要做的第一件事是删除此Pod，并尝试使用–validate选项再次创建它。例如，运行kubectl create –validate -f mypod.yaml。如果拼写错误command，commnd那么会出现如下错误：

I0805 10:43:25.129850 46757 schema.go:126] unknown field: commnd

I0805 10:43:25.129973 46757 schema.go:129] this may be a false alarm, see https://github.com/kubernetes/kubernetes/issues/6842

pods/mypod

接下来，要检查的是apiserver上的Pod是否与要创建的Pod相匹配。例如，运行kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml，然后将原始的Pod描述mypod.yaml与从apiserver返回的描述文件mypod-on-apiserver.yaml进行比较。“apiserver”版本通常会有一些不在原始版本上的内容，这不影响。但是，如果原始版本上有不在apiserver版本上的行，则可能意味着原始版本Pod描述规范存在问题。

### 调试代理服务

根据Kubernetes的架构设计，用户或其它应用通过代理服务访问容器化应用。因此需要通过调试确认代理服务是否正常，需要做的工作包括：

1）检查代理服务本身是否存在；

2）检查代理服务是否能够正常通过DNS进行解析；

3）检查代理服务本身是否正确。

#### 检查服务是否存在

在调试服务时，第一步要做的就是检查服务是否存在。在本文的前面说明了，在Kubernetes中通过NodePort类型对外暴露了MySQL master。通过执行kubectl get svc命令，可以获取是否存在相应服务：

$ kubectl get svc/mysql-0-svc --namespace=kube-public

https://www.kubernetes.org.cn/img/2018/08/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20180808131914.png

通过返回的结果可以看出，在Kubernetes集群中存在此服务。

#### 能否通过DNS解析正常解析代理服务

对于处于同一个命名空间的容器化应用，可以直接通过代理服务的名称(mysql-0-svc)访问MySQL master。

$ kubectl exec -it redis-ha-redis-ha-sentinel-5947b9569-r2b56 --namespace=kube-public -- nslookup mysql-0-svc

https://www.kubernetes.org.cn/img/2018/08/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20180808133616.png

对Kubernetes集群中不同命名空间的容器化应用，则需要通过添加命名空间名称后(mysql-0-svc.kube-public)访问MySQL master：

$ kubectl exec -it gf1-6497d5df45-98g8v -- nslookup mysql-0-svc.kube-public

https://www.kubernetes.org.cn/img/2018/08/%E5%BE%AE%E4%BF%A1%E6%88%AA%E5%9B%BE_20180809092321.png

根据返回的可以看出，通过DNS能够正确的解析代理服务。

#### DNS是否正常工作

如果通过上述的操作都无法正常解析服务，通过*kubectl exec -it  ${POD\_NAME}  — nslookup*命令检查一下Kubernetes master是否正常工作：

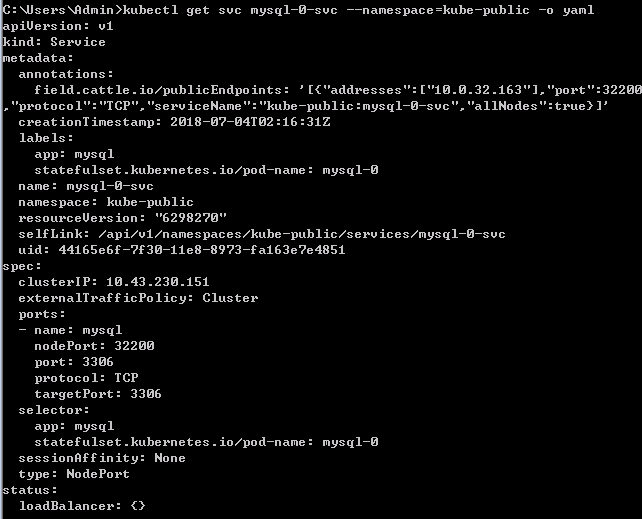
$ kubectl exec -it gf1-6497d5df45-98g8v -- nslookup kubernetes.default

如果此操作也失败，则需要检查Kubernetes集群中的DNS服务是否正常运行。

#### 代理服务本身是否正确

如果代理服务也存在，DNS解析也没有问题。则需要检查一下代理服务本身是否有问题：

$ kubectl get service mysql-0-svc -o yaml --namespace=kube-public



例如访问的端口是否正确？targetPort是否指向了正确的Pods端口？这里的端口协议是否与Pod暴露出来的端口协议一致等。

## [docker守护程序问题处理](https://www.kubernetes.org.cn/5009.html)

容器作为容器云的管理的对象，其真正运行的引擎还是docker，当然也可以使用其他的容器引擎，当前目前市场中主要使用的还是docker技术。Docker守护进程本身的问题也将会影响容器化应用的运行情况。

### 检查Docker是否正在运行

在进行具体问题定位和处理之前，最好和最简单直观的方式是先看看docker引擎是否能够正常运行。有以下几种方法可以确认docker是否处于正常运行状态：

1）通过执行docker info命令可以直接确认docker是否正常运行：

$ sudo docker info

2）也可以使用操作系统的命令来检查docker是否正常运行：

$ sudo systemctl is-active docker

# 或者

$ sudo service docker status

3）可以使用ps或top等命令检查进程的进程是否有正常运行的dockerd。

### 解决*daemon.json*与启动脚本之间的冲突

在Docker中，守护程序会将所有数据都保存在一个根目录中。通过此目录，将能够跟踪到与Docker相关的所有内容，包括容器(containers)，镜像(images)，存储卷(volumes)，服务定义(service definition)和机密(secrets)。默认情况下，此根目录为：

* Linux环境：/var/lib/docker
* Windows环境：C:\ProgramData\docker

可以使用data-root配置选项将Docker守护程序的配置为使用其他目录 。由于Docker守护程序的状态保留在此根目录中，因此需要确保每个守护程序使用专用的目录。如果两个守护程序共享同一根目录（例如，NFS共享），将可能会导致故障的无法处理的情况。

在docker引擎中，如果使用了daemon.json文件，同时通过手动或使用启动脚本将选项传递给dockerd命令，并且这些选项间存在冲突，则会导致Docker无法启动，例如：

unable to configure the Docker daemon with file /etc/docker/daemon.json:

the following directives are specified both as a flag and in the configuration

file: hosts: (from flag: [unix:///var/run/docker.sock], from file: [tcp://127.0.0.1:2376])

如果看到类似于上述内容的错误，并且使用此标志手动启动守护程序，则可能需要调整命令字段或daemon.json以解决冲突。如果使用操作系统的初始脚本启动Docker，则可能需要以特定于操作系统的方式覆盖这些脚本中的默认值。

默认情况下，Docker会监听套接字(socket)。在Debian和Ubuntu操作系统中使用启动docker时，-H字段会一直被使用。如果在daemon.json指定了一个 hosts条目，则会导致配置冲突，从而导致Docker无法启动。为了解决此问题，请/etc/systemd/system/docker.service.d/docker.conf使用以下内容创建一个新文件，以删除默认情况下启动守护程序时使用的-H参数。

[Service]

ExecStart=

ExecStart=/usr/bin/dockerd

### 查看日志

守护程序的日志可以帮助诊断问题，docker守护程序日志的具体存储位置取决于操作系统配置和使用的日志记录子系统：

表17-1 docker守护程序的日志存储位置

| 操作系统 | 目录 |
| --- | --- |
| RHEL，Oracle Linux | /var/log/messages |
| Debian | /var/log/daemon.log |
| Ubuntu 16.04+，CentOS | 使用*journalctl -u docker.service*命令检查日志 |
| Ubuntu 14.10- | /var/log/upstart/docker.log |
| macOS（Docker 18.01+） | ~/Library/Containers/com.docker.docker/Data/vms/0/console-ring |
| macOS（Docker <18.01） | ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/console-ring |
| Windows | AppData\Local |

#### 启用调试

在daemon.json文件中将 debug键的值设置为true就可以启用docker的调试模式，此方法适用于每个Docker平台。daemon.json文件通常位于/etc/docker/目录，如果该文件尚不存在，则需要手动创建此文件。

如果文件为空，请添加以下内容：

{

"debug": true

}

如果文件中已经包含JSON，则只需添加键值对”debug”: true，如果它不是文件中的最后最后一行，请在行尾添加“；”。同时需要确认log-level是否已设置为info或debug。默认值为info，可能的值包括debug，info，warn，error，fatal。

向守护程序发送hup信号以使其重新加载新的配置。在Linux主机上，使用以下命令。

$ sudo kill -SIGHUP $(pidof dockerd)

也可以手动停止Docker守护程序，并使用debug的-D字段标志重新启动它。

#### 强制记录堆栈跟踪

如果守护程序没有响应，则可以通过向SIGUSR1守护程序发送信号来强制记录完整堆栈跟踪。

$ sudo kill -SIGUSR1 $(pidof dockerd)

这会强制记录堆栈跟踪，但不会停止守护程序。守护程序日志显示堆栈跟踪或包含堆栈跟踪的文件的路径（如果已记录到文件）。处理完SIGUSR1信号并将堆栈跟踪转储到日志后，守护程序继续运行。堆栈跟踪可用于确定守护程序中所有goroutine和线程的状态。

#### 查看堆栈跟踪

可以使用以下方法之一查看Docker守护程序日志：

* 在Linux系统上执行journalctl -u docker.service命令；
* 在较旧的Linux系统上，查看/var/log/messages，/var/log/daemon.log或/var/log/docker.log

在Docker日志中查找如下消息：

...goroutine stacks written to /var/run/docker/goroutine-stacks-2017-06-02T193336z.log

...daemon datastructure dump written to /var/run/docker/daemon-data-2017-06-02T193336z.log

Docker保存这些堆栈跟踪和转储的位置取决于操作系统和配置，有时可以直接从堆栈跟踪和转储中获取有用的诊断信息。

## 处理由于服务器时间不一致带来的问题

在搭建Kubernetes时，像Etcd集群服务都需要依赖宿主机的时间，如果宿主机的系统时间不一致，可能会造成各种问题。因此需要统一集群内所有宿主机的系统时间，以防止由于时间不一致所带来各种问题。

1. **下载ntp安装介质**

需要有一台能够上网的机器，通过此机器下载ntp安装介质：

#创建一个放置安装介质的目录

mkdir /home/ntp

#下载ntp的安装介质

yum -y install ntp --downloadonly --downloaddir /home/ntp

#进入/home/ntp目录

cd /home/ntp

#将安装介质压缩成tar文件

tar -cvf ntp.tar \*.rpm

1. **安装ntp服务端**

拷贝tar文件到ntp服务端机器上，这里选择Kubernetes的master节点所在的宿主机作为ntp服务端。

# 创建放置ntp.tar文件的目录

mkdir /home/ntp

# 进入/home/ntp目录

cd /home/ntp

# 解压缩ntp.tar文件

tar –xvf ntp.tar

# 使用rpm命令安装ntp

rpm –ivh \*.rpm

在安装完成后，需要通过编辑ntp.conf文件进行服务端配置：

vi /etc/ntp.conf

下面是ntp.conf所编辑的内容：

# 由于是在内网环境下使用，注释掉以下内容

# server 0.centos.pool.ntp.org iburst

# server 1.centos.pool.ntp.org iburst

# server 2.centos.pool.ntp.org iburst

# server 3.centos.pool.ntp.org iburst

# 允许本机的任何操作

restrict 127.0.0.1

# 允许10.0.32.\*的所有机器同步时间

restrict 10.0.32.0 mask 255.255.255.0 nomodify

# 添加下面内容，将此本机作为ntp服务端

server 127.127.1.0

# 设置系统时钟的层级

fudge 127.127.1.0 stratum 10

其中：

* restrict [ip] [mask] [par]：对ntp的访问权限进行设置
  + ip：网络的ip地址
  + mask：网络的子网掩码
  + par：其他参数
    - ignore：忽略所有类型的NTP连接请求；
    - nomodify：限制客户端修改服务器端的时间；
    - noquery：不提供NTP网络时间同步服务；
    - notrap：不接受远程的登录请求；
    - notrust：不接受没有经过认证的客户端请求；
    - 如果没有用任何参数，那么表示不做任何限制。
* server [ip or hostname]：设置以那个ntp服务端作为基准同步时间。
* fudge [ip] [stratnum int]：设置系统时钟的层数，取值范围为1～16，定义了时钟的准确度。层数为1的时钟准确度最高，准确度从1到16依次递减，层数为16的时钟处于未同步状态，不能作为参考时钟。

1. **安装ntp客户端**

拷贝tar文件到ntp客户端机器上，Kubernetes集群其它节点所在的宿主机。

# 创建放置ntp.tar文件的目录

mkdir /home/ntp

# 进入/home/ntp目录

cd /home/ntp

# 解压缩ntp.tar文件

tar –xvf ntp.tar

# 使用rpm命令安装ntp

rpm –ivh \*.rpm

在安装完成后，需要通过编辑ntp.conf文件进行服务端配置：

vi /etc/ntp.conf

下面是ntp.conf所编辑的内容：

# 由于是在内网环境下使用，注释掉以下内容

# server 0.centos.pool.ntp.org iburst

# server 1.centos.pool.ntp.org iburst

# server 2.centos.pool.ntp.org iburst

# server 3.centos.pool.ntp.org iburst

# 添加下面内容，以master主机时间为基准进行时间同步

server 192.168.8.133

1. **启动ntp服务**

通过执行下面的命令启动服务端的ntp：

# 启动ntpd服务

service ntpd restart

# 设置ntpd服务为开机启动

systemctl enable ntpd.service

# 查看ntp服务的状态

ntpstat

1. **同步时间**

在客户端通过手动执行下面的命令进行时间同步：

ntpdate 192.168.8.133



另外，也可以通过与crontab一起，定时进行时间同步。通过在crontab中添加下面的内容，系统就会在每天的12点整同步一次时间。

0 12 \* \* \* \* /usr/sbin/ntpdate 192.168.0.1

其中，crontab内容的格式为：[分钟] [小时] [每月的某一天] [每年的某一月] [每周的某一天] [执行命令]。

# 企业网盘基于Kubernetes的研发和运行

在本章节，将以企业网盘这个应用系统的研发为例，阐述如何基于Kubernetes和DevOps进行应用系统的持续集成和运行维护。

## 企业网盘介绍

企业网盘是神软公司自研的一款网盘产品。用于为个人提供集中的安全的文件自我管理工具，通过企业网盘用户能够方便的存储、搜索、分享和查看文件；以及为团队提供以文件为中心的轻量级协同工作环境，基于此环境团队成员能够协同对文件维护，并进行沟通。



图18-1 企业网盘定位

以往，企业会利用共享文件夹或 FTP 等手段搭建一个简单的文件共享服务器进行文件管理，由于这些工具未充分考虑用户的使用习惯，因此操作相对比较复杂。企业网盘针对用户使用习惯，进行了合理的用户界面布局和操作序列设计，为用户提供易于使用的文件管理工具。

基于企业网盘良好的用户体验，保证了用户能够进行高效的文件维护工作。同时，基于网盘提供的协同工作能力，协同团队成员能够有效提升以文件为中心的协同办公效率。

企业网盘从存储空间和存储设备两个方面着手，进行了降低存储成本的考虑。通过计算，相同的文件在服务器上只保留一份。企业网盘的后台存储支持最常用文件系统的存储方式，低成本就可以获取超大存储空间。

企业网盘在设计之处就严格遵守分级保护安全测评的要求，提供了包括身份认证、访问控制、密级管理、传输加密、存储加密和审计日志等完整的安全防护能力。

## 基于DevOps的持续集成

### 企业网盘的产品研发团队

此产品的研发团队成员由产品经理、开发经理、用户体验设计师、开发工程师、配置管理员、测试工程师和运维人员所组成。这些团队成员各自的工作职责如下：

* 产品经理：负责企业网盘的整体规划、市场调研、竞品分析、用户需求分析、版本发布和市场推广等工作；
* 开发经理：负责企业网盘的整体技术、以及带领开发工程师根据需求完成系统的研发工作；
* 用户体验设计师：负责根据用户需求进行企业网盘功能的用户体验设计，并输出设计原型给开发工程师；
* 开发工程师：负责完成开发经理所分配的功能实现，并进行功能自测；
* 配置管理员：负责在devops中为企业网盘创建和维护相应的项目；
* 质量管理员：负责制定质量管理计划，并依据质量管理计划对企业网盘研发过程进行质量管控；
* 测试工程师：编写测试用例，并依据测试用例对企业网盘的功能和性能进行测试；
* 运维人员：负责维护DevOps环境和企业网盘的正常运行。

### 基于DevOps的研发过程

1. **代码实现**

配置管理员在gitlab中为企业网盘创建一个项目，此处的项目名称为drap-boot-NetDisk-sjs。在gitlab上创建好项目以后，配置管理员将会告知开发经理。

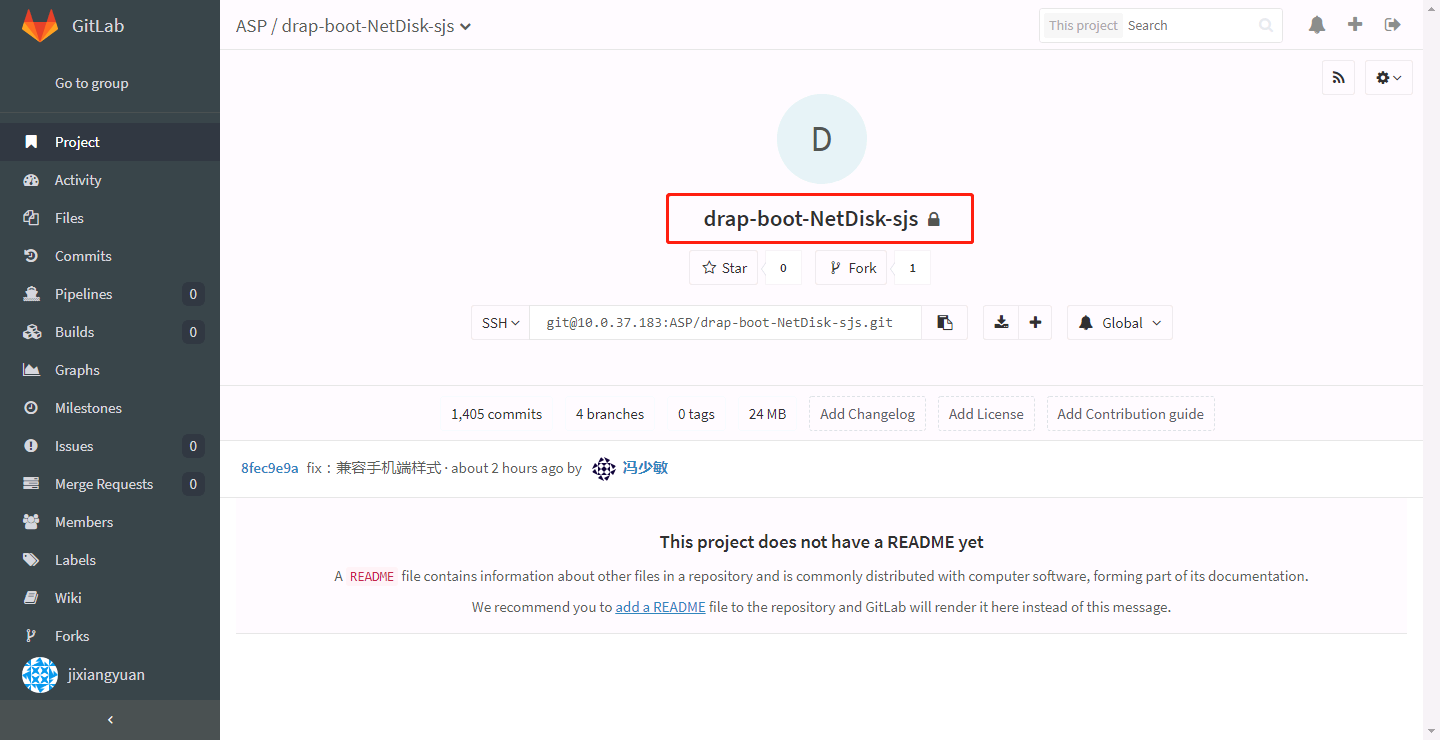


图18-2 gitlab中的企业网盘项目

开发经理在eclipse中拉取drap-boot-NetDisk-sjs项目，并根据企业网盘的功能规划构建好代码的组织结构，并将完成后的工作推送至gitlab的项目中。

各开发工程师在eclipse中拉取drap-boot-NetDisk-sjs项目，并进行自己所负责功能模块的代码开发，并在完成后推送至gitlab中进行托管。

1. **流水线配置**

在完成了代码研发工作后，配置管理员在jenkins中为网盘创建一个maven类型的项目，此项目的名称为netdisk-k8s。

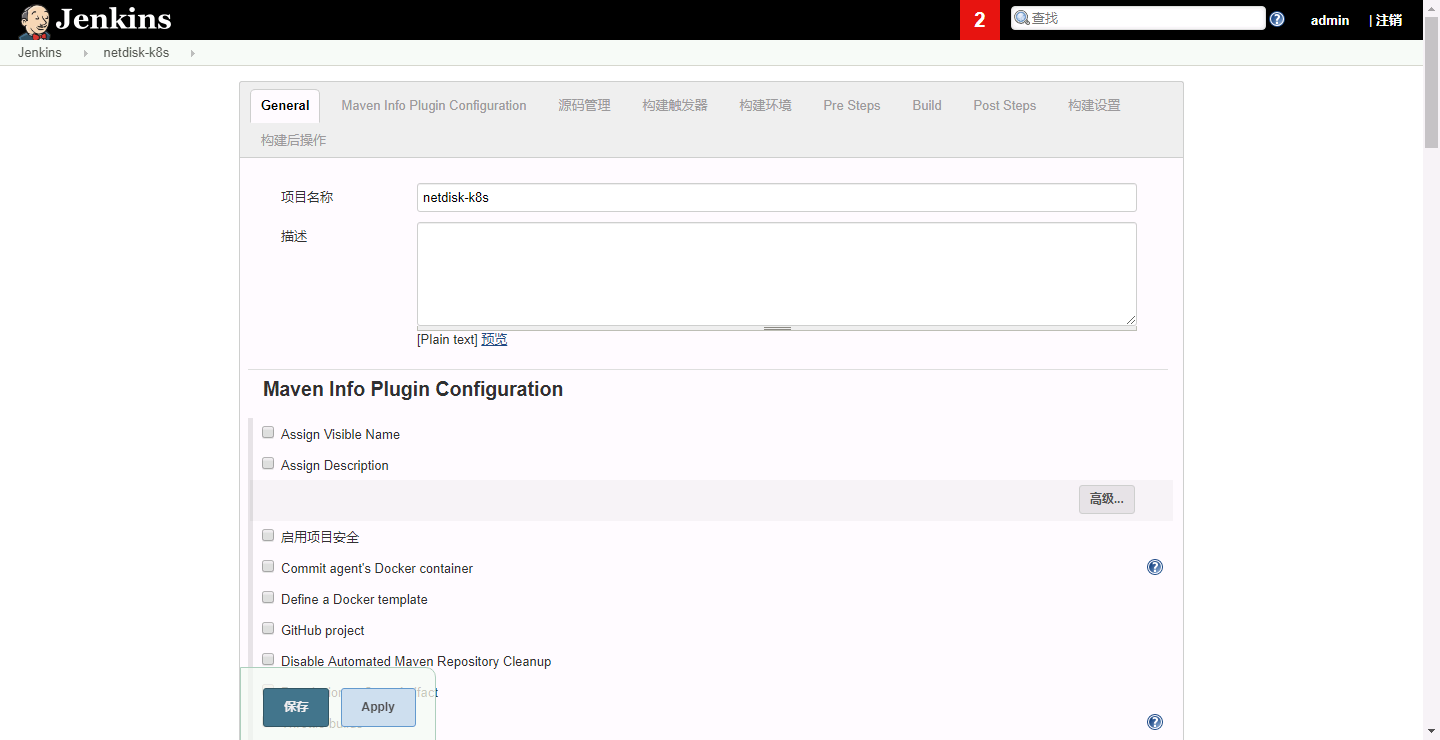


图18-3 jenkins中的企业网盘项目

netdisk-k8s项目的源代码来自于gitlab的git@10.0.37.183:ASP/drap-boot-NetDisk-sjs.git仓库。

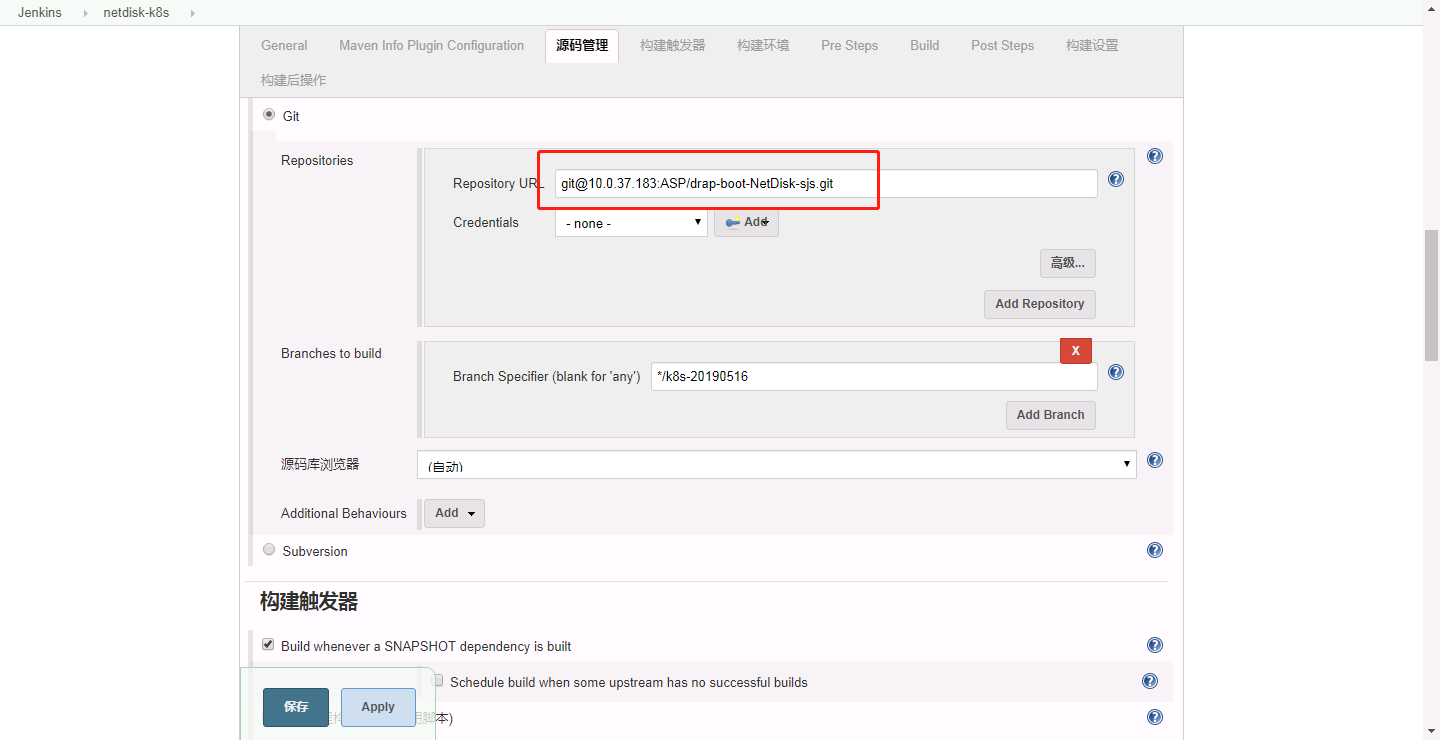


图18-4 配置源代码仓库

[netdisk-k8s项目](mailto:netdisk-k8s项目的源代码来自于gitlab的git@10.0.37.183:ASP/drap-boot-NetDisk-sjs.git)的依赖关系由maven进行管理，这些依赖关系由项目中的pom.xml文件进行配置和管理。

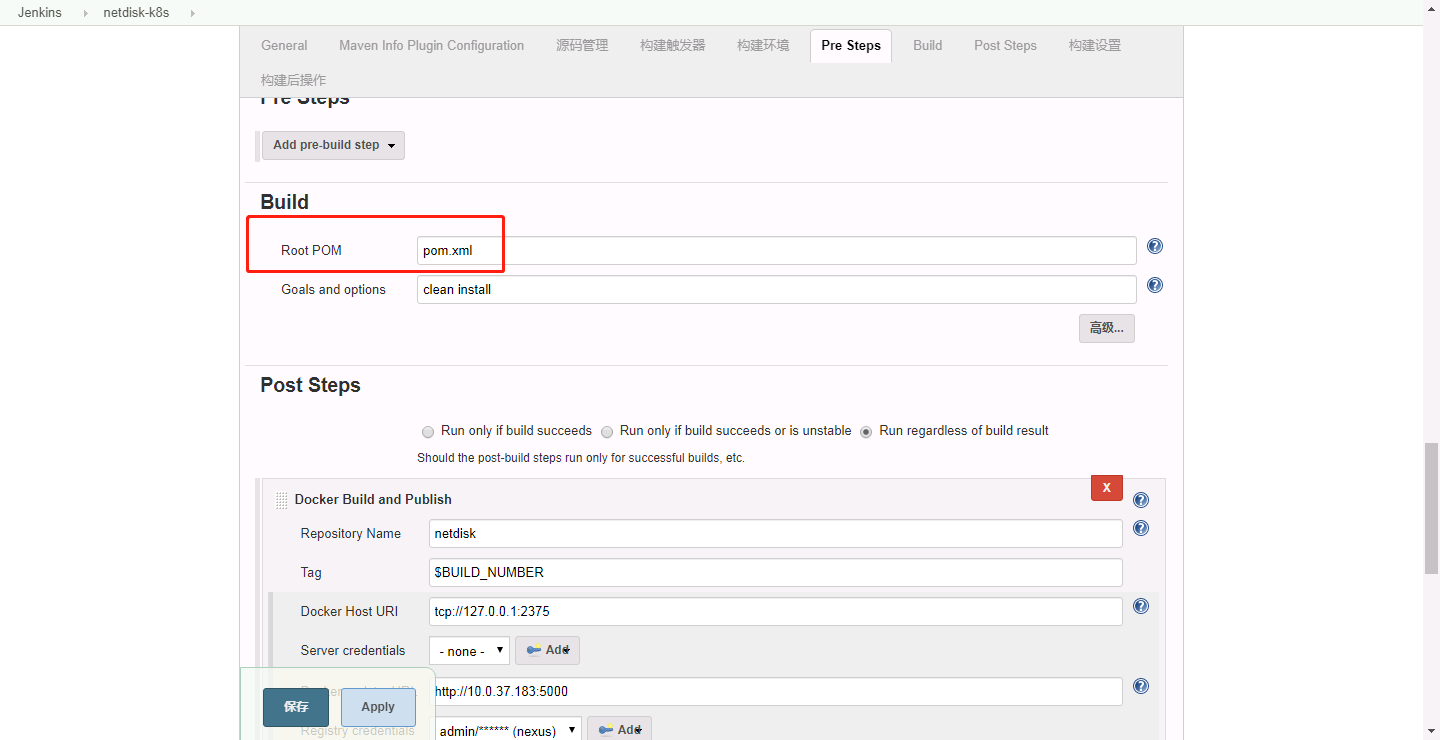


图18-5 配置maven构建信息

在代码构建完成后，由docker负责进行镜像构建，并会自动上传至私有镜像仓库中。

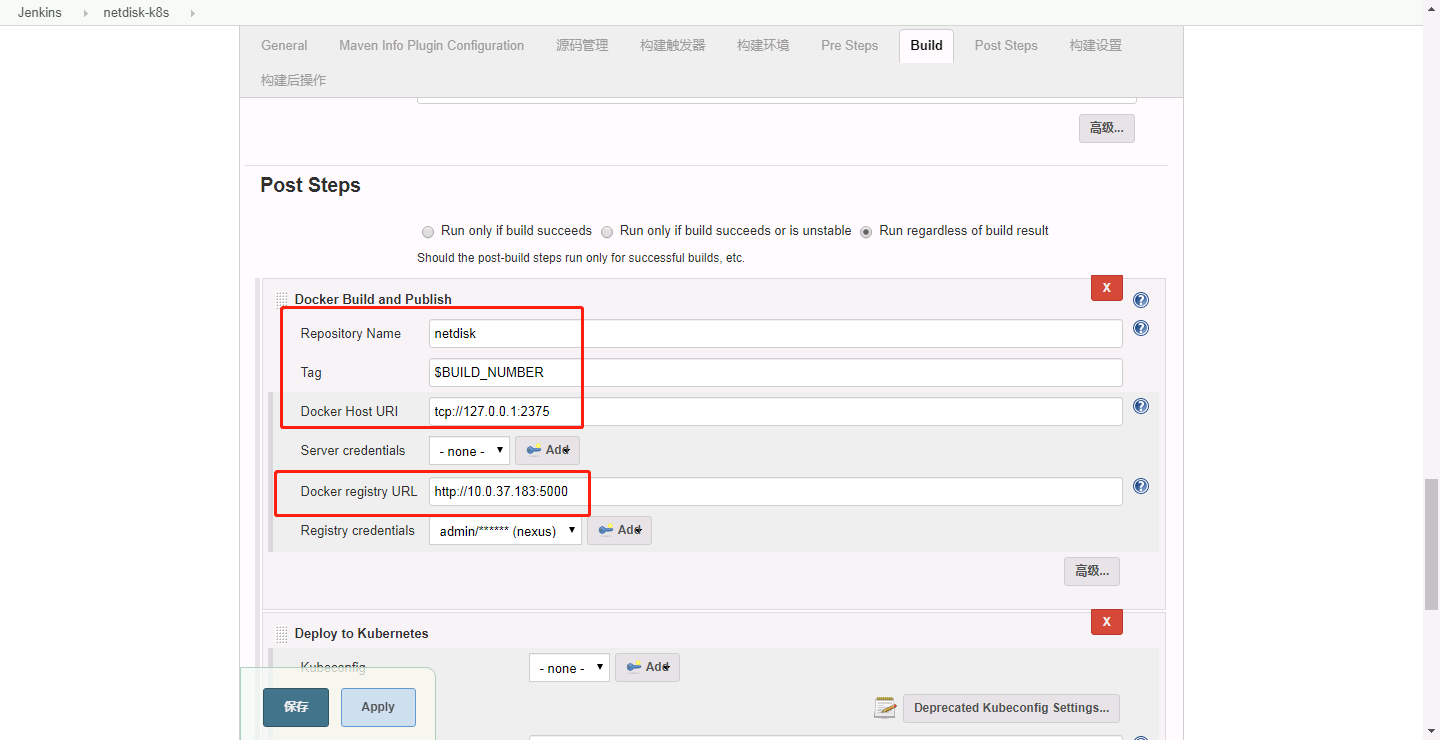


图18-6 配置docker和私有镜像仓库

最终，jenkins的流水线将会从私有镜像仓库中拉取镜像，并将其部署至Kubernetes中，此部署过程由netdisk.yaml文件定义。

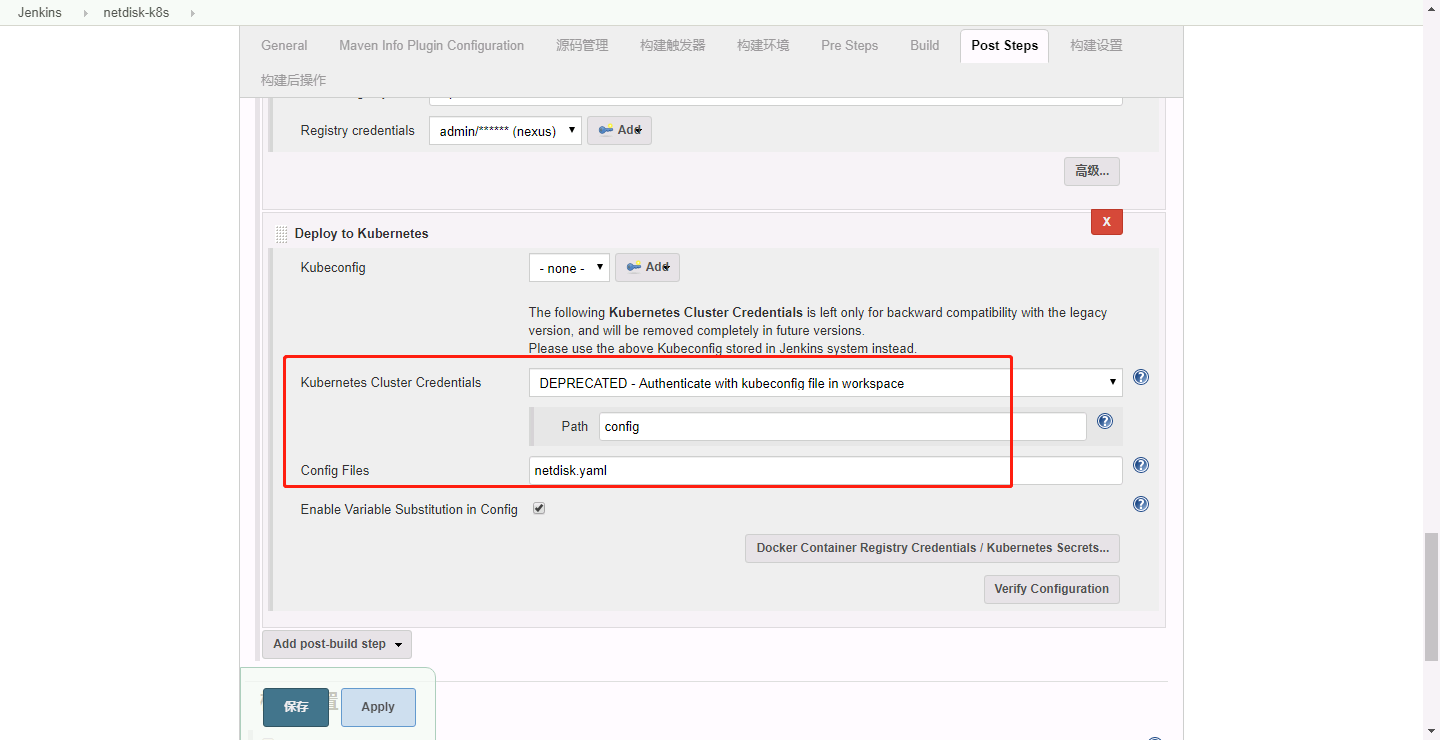


图18-7 配置Kubernetes认证和部署文件

1. **过程自动化**

在jenkins流水线配置完成后，就可以一键执行。这个过程包括从gitlab中获取源代码，基于maven进行源代码构建，打包镜像，并将镜像上传至私有镜像仓库，最后将镜像部署至Kubernetes中运行起来。

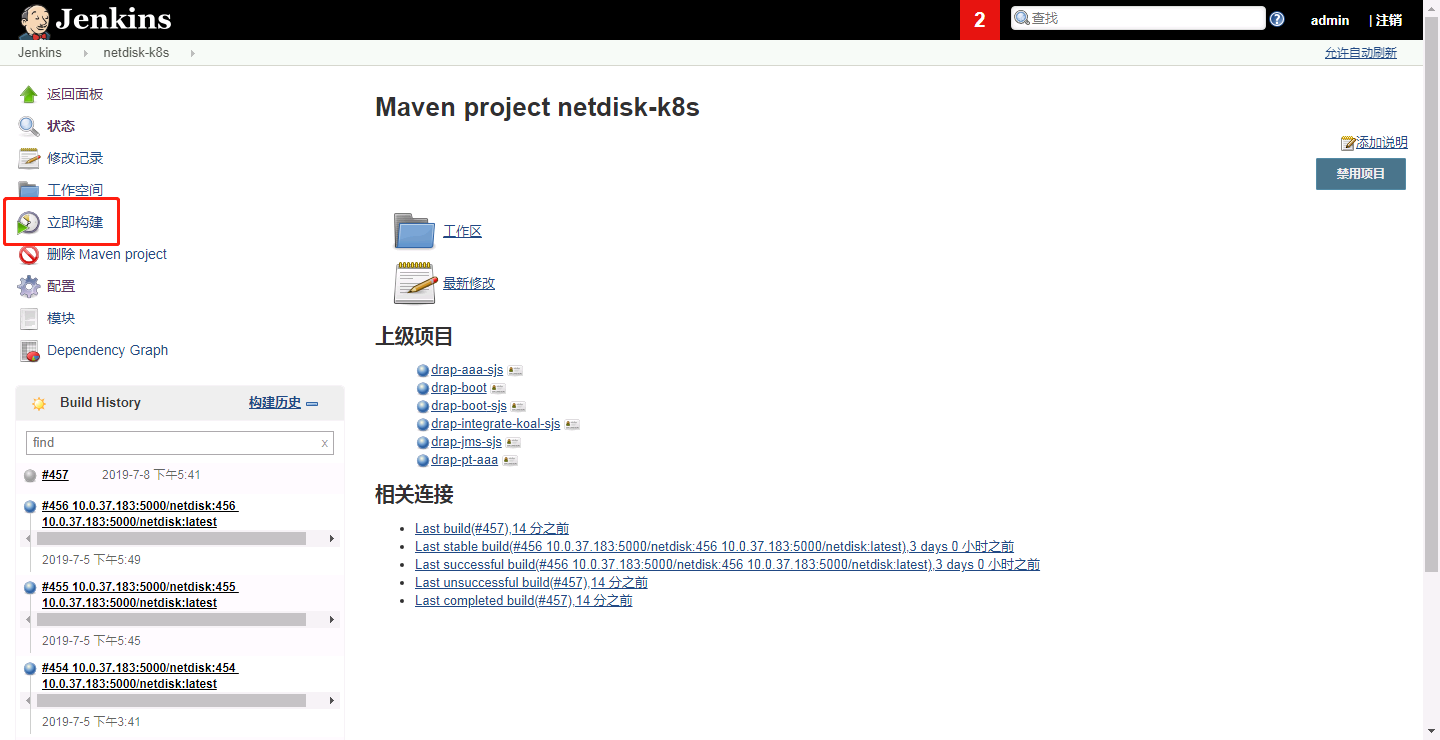


图18-8 构建过程

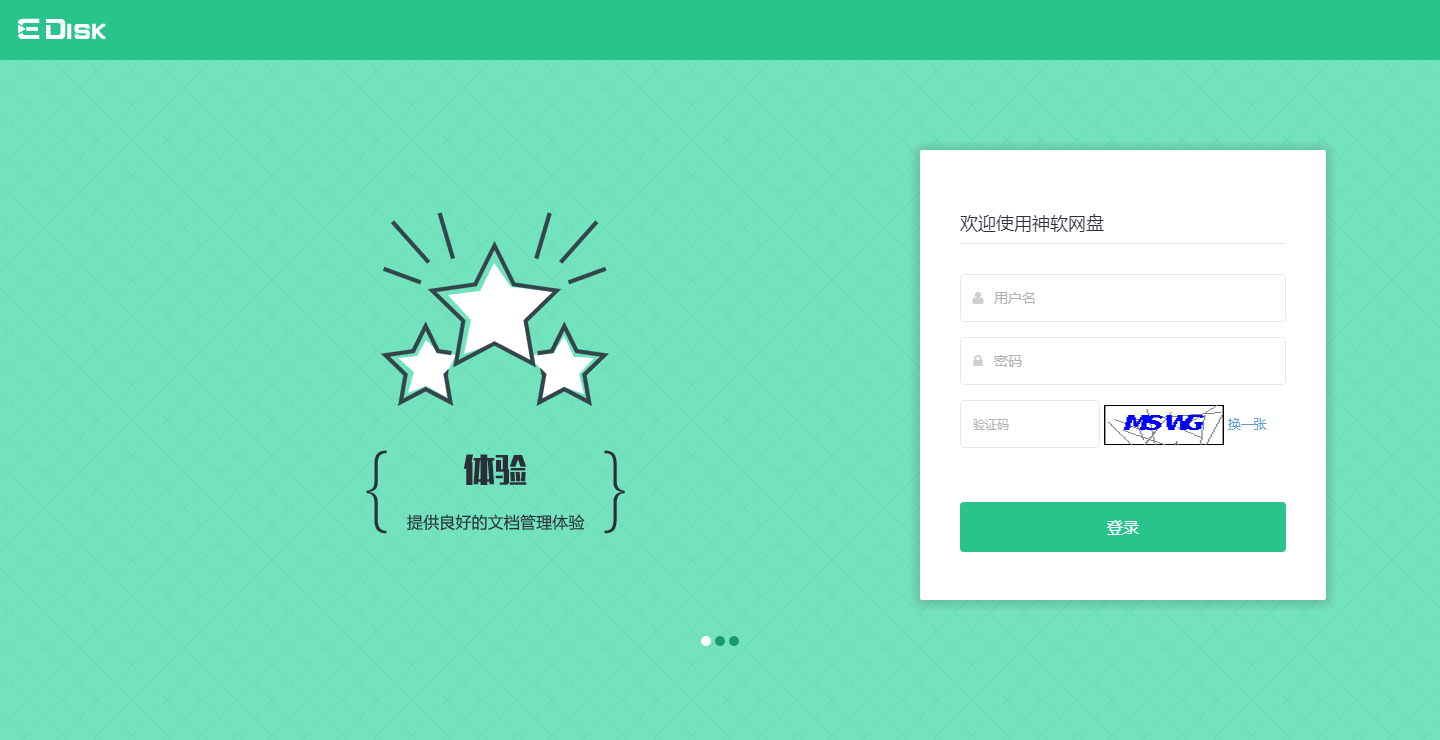


图18-9 神软-企业网盘

测试工程师可以直接基于Kuberenetes中的运行环境进行功能测试，并提交bug；开发工程师在修改好bug后，就可以重新提交代码；开发经理确认所有的问题都已解决，则可以通知配置管理员再一次执行jenkins中的流水线。DevOps系统会重新执行上述的过程，并灰度升级应用工测试工程师进行下一轮测试。

# 参考文献

[1] 胡世杰. 分布式对象存储[M]. 北京：人民邮电出版，2018.

[2] Nigel Poulton（奈吉尔·波尔顿）[英]. 深入浅出Docker[M]. 北京:人民邮电出版社，2019.

[3] 龚正. Kubernetes权威指南. 北京:电子工业出版社，2019.

[4] (美)[施瓦茨](http://search.dangdang.com/?key2=%CA%A9%CD%DF%B4%C4&medium=01&category_path=01.00.00.00.00.00),(美)[扎伊采夫](http://search.dangdang.com/?key2=%D4%FA%D2%C1%B2%C9%B7%F2&medium=01&category_path=01.00.00.00.00.00),(美)[特卡琴科](http://search.dangdang.com/?key2=%CC%D8%BF%A8%C7%D9%BF%C6&medium=01&category_path=01.00.00.00.00.00" \o "(美)施瓦茨,(美)扎伊采夫,(美)特卡琴科). [高性能MySQL（第3版）](http://product.dangdang.com/23214590.html" \o " 高性能MySQL（第3版） 领域排头 畅行全球 天团献译 从业必收" \t "_blank). **北京：**[电子工业出版社](http://search.dangdang.com/?key=&key3=%B5%E7%D7%D3%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00" \o "电子工业出版社)，2013.

[5] [李子骅](http://search.dangdang.com/?key2=%C0%EE%D7%D3%E6%E8&medium=01&category_path=01.00.00.00.00.00). [Redis入门指南（第2版）](http://product.dangdang.com/23695514.html).  北京：[人民邮电出版社](http://search.dangdang.com/?key=&key3=%C8%CB%C3%F1%D3%CA%B5%E7%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2015.

[6] [日][户根勤](http://search.dangdang.com/?key2=%BB%A7%B8%F9%C7%DA&medium=01&category_path=01.00.00.00.00.00). 网络是怎样连接的. 北京:[人民邮电出版社](http://search.dangdang.com/?key3=%C8%CB%C3%F1%D3%CA%B5%E7%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2017.

[7] [杨传辉](http://search.dangdang.com/?key2=%D1%EE%B4%AB%BB%D4&medium=01&category_path=01.00.00.00.00.00). 大规模分布式存储系统 北京：[机械工业出版社](http://search.dangdang.com/?key3=%BB%FA%D0%B5%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2013.

[8] [华为云容器服务团队](http://search.dangdang.com/?key2=%BB%AA%CE%AA%D4%C6%C8%DD%C6%F7%B7%FE%CE%F1%CD%C5%B6%D3&medium=01&category_path=01.00.00.00.00.00) [杜军](http://search.dangdang.com/?key2=%B6%C5%BE%FC&medium=01&category_path=01.00.00.00.00.00). 云原生分布式存储基石 北京：[机械工业出版社](http://search.dangdang.com/?key3=%BB%FA%D0%B5%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2018.

[9] Docker官网文档：<https://docs.docker.com/>

[10] Kubernetes官网文档：<https://kubernetes.io/docs/home/>

[11] Rancher官网文档：<https://www.cnrancher.com/docs/>

[12] 华为云软件开发服务DevCloud：<https://www.huaweicloud.com/devcloud/>

[13] 阿里云容器服务ACS：<https://www.aliyun.com/product/containerservice>

[14] [朱林](http://search.dangdang.com/?key2=%D6%EC%C1%D6&medium=01&category_path=01.00.00.00.00.00). Elasticsearch技术解析与实战. 北京：[机械工业出版社](http://search.dangdang.com/?key=&key3=%BB%FA%D0%B5%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2017.

[15] [饶琛琳](http://search.dangdang.com/?key2=%C8%C4%E8%A1%C1%D5&medium=01&category_path=01.00.00.00.00.00). ELK Stack权威指南. 北京：[机械工业出版社](http://search.dangdang.com/?key3=%BB%FA%D0%B5%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2017.

[16] [陈晓宇](http://search.dangdang.com/?key2=%B3%C2%CF%FE%D3%EE&medium=01&category_path=01.00.00.00.00.00). 深入浅出Prometheus. 北京:[电子工业出版社](http://search.dangdang.com/?key3=%B5%E7%D7%D3%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2019.

[17] Helm官网文档：<https://helm.sh/docs/>

[18] （瑞典）[Joakim](http://search.dangdang.com/?key2=Joakim&medium=01&category_path=01.00.00.00.00.00) [Verona](http://search.dangdang.com/?key2=Verona&medium=01&category_path=01.00.00.00.00.00)（[约阿基姆·维罗纳](http://search.dangdang.com/?key2=%D4%BC%B0%A2%BB%F9%C4%B7%A1%A4%CE%AC%C2%DE%C4%C9&medium=01&category_path=01.00.00.00.00.00)）著，[高清华](http://search.dangdang.com/?key2=%B8%DF%C7%E5%BB%AA&medium=01&category_path=01.00.00.00.00.00) [马博文](http://search.dangdang.com/?key2=%C2%ED%B2%A9%CE%C4&medium=01&category_path=01.00.00.00.00.00)译. DevOps 实践作者. 北京：[电子工业出版社](http://search.dangdang.com/?key3=%B5%E7%D7%D3%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2016.

[19] （美）[John](http://search.dangdang.com/?key2=John&medium=01&category_path=01.00.00.00.00.00) [Ferguson](http://search.dangdang.com/?key2=Ferguson&medium=01&category_path=01.00.00.00.00.00) Smart（[约翰.弗格森.斯马特](http://search.dangdang.com/?key2=%D4%BC%BA%B2.%B8%A5%B8%F1%C9%AD.%CB%B9%C2%ED%CC%D8&medium=01&category_path=01.00.00.00.00.00)）著，[郝树伟](http://search.dangdang.com/?key2=%BA%C2%CA%F7%CE%B0&medium=01&category_path=01.00.00.00.00.00) [于振苓](http://search.dangdang.com/?key2=%D3%DA%D5%F1%DC%DF&medium=01&category_path=01.00.00.00.00.00) [熊熠](http://search.dangdang.com/?key2=%D0%DC%EC%DA&medium=01&category_path=01.00.00.00.00.00)译. Jenkins权威指南. 北京:[电子工业出版社](http://search.dangdang.com/?key3=%B5%E7%D7%D3%B9%A4%D2%B5%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2016.

[20] 孙宏明. 完全学会Git, GitHub, Git Server的24堂课. 北京：[清华大学出版社](http://search.dangdang.com/?key3=%C7%E5%BB%AA%B4%F3%D1%A7%B3%F6%B0%E6%C9%E7&medium=01&category_path=01.00.00.00.00.00)，2016.

[21] fluentd官网文档：<https://docs.fluentd.org/>