**Kubernetes 学习文档**

# 知识回顾

## 什么是 Linux 容器？

Linux 容器是与系统其他部分隔离开的一系列进程。运行这些进程所需的所有文件都由另一个镜像提供，这意味着从开发到测试再到生产的整个过程中，Linux 容器都具有可移植性和一致性。因而，相对于依赖重复传统测试环境的开发渠道，容器的运行速度要快得多。

假设您在开发一个应用。您使用的是一台笔记本电脑，而且您的开发环境具有特定的配置。其他开发人员身处的环境配置可能稍有不同。您正在开发的应用不止依赖于您当前的配置，还需要某些特定的库、依赖项和文件。与此同时，您的企业还拥有标准化的开发和生产环境，有着自己的配置和一系列支持文件。您希望尽可能多在本地模拟这些环境，而不产生重新创建服务器环境的开销。因此，您要如何确保应用能够在这些环境中运行和通过质量检测，并且在部署过程中不出现令人头疼的问题，也无需重新编写代码和进行故障修复？答案就是使用容器。

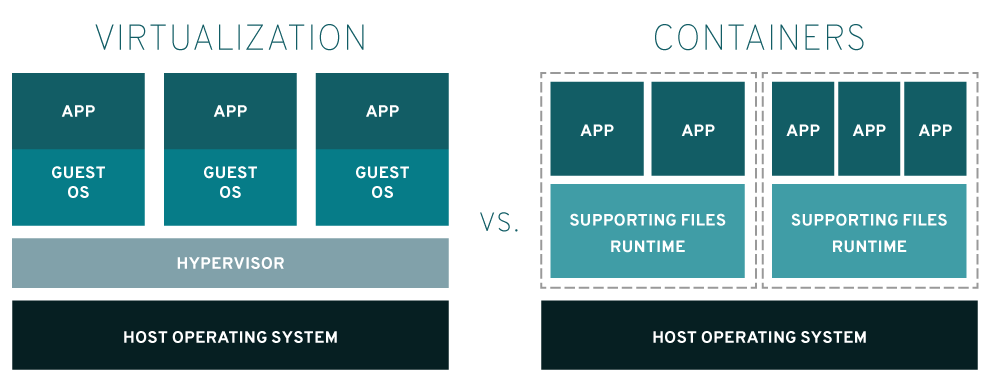
容器可以确保您的应用拥有必需的库、依赖项和文件，让您可以在生产中自如地迁移这些应用，无需担心会出现任何负面影响。实际上，您可以将容器镜像中的内容，视为 Linux 发行版的一个安装实例，因为其中完整包含 RPM 软件包、配置文件等内容。但是，安装容器镜像发行版，要比安装新的操作系统副本容易得多。这样可以避免危机，做到皆大欢喜。

* **容器不就是虚拟化吗？**

不完全如此。更确切的说法应该是：两者为互补关系。我们用一种简单方式来思考一下：

虚拟化使得您的操作系统（Windows 或 Linux）可同时在单个硬件系统上运行。

容器则可共享同一个操作系统内核，将应用进程与系统其他部分隔离开。例如：ARM Linux 系统运行 ARM Linux 容器，x86 Linux 系统运行 x86 Linux 容器，x86 Windows 系统运行 x86 Windows 容器。Linux 容器具有极佳的可移植性，但前提是它们必须与底层系统兼容。



这意味着什么？虚拟化会使用虚拟机监控程序模拟硬件，从而使多个操作系统能够并行运行。但这不如容器轻便。事实上，在仅拥有容量有限的有限资源时，您需要能够可以进行密集部署的轻量级应用。Linux 容器在本机操作系统上运行，与所有容器中共享该操作系统，因此应用和服务能够保持轻巧，并行化快速运行。

Linux 容器是我们开发、部署和管理应用方式的又一次飞跃。Linux 容器镜像提供了可移植性和版本控制，确保能够在开发人员的笔记本电脑上运行的应用，同样也能在生产环境中正常运行。相较于虚拟机，Linux 容器在运行时所占用的资源更少，使用的是标准接口（启动、停止、环境变量等），并会与应用隔离开；此外，作为（包含多个容器）大型应用的一部分时更加易于管理，而且这些多容器应用可以跨多个云环境进行编排。

* **容器简史**

容器并非起源于 Linux，但开源世界的最精彩之处就在于借鉴、修改和改进，容器也不例外。

我们现在称为容器技术的概念最初出现在 2000 年，时称 FreeBSD jail，这种技术可将 FreeBSD 系统分区为多个子系统（也称为 Jail）。Jail 是作为安全环境而开发的，系统管理员可与企业内部或外部的多个用户共享这些 Jail。Jail 的目的是让进程在经过修改的 chroot 环境中创建，而不会脱离和影响整个系统 — 在 chroot 环境中，对文件系统、网络和用户的访问都实现了虚拟化。尽管 Jail 在实施方面存在局限性，但最终人们找到了脱离这种隔离环境的方法。

但这个概念非常有吸引力。

2001 年，通过 Jacques Gélinas 的 VServer 项目，隔离环境的实施进入了 Linux 领域。正如 Gélinas 所说，这项工作的目的是“在高度独立且安全的单一环境中运行多个通用 Linux 服务器 [sic]。” 在完成了这项针对 Linux 中多个受控制用户空间的基础性工作后，Linux 容器开始逐渐成形并最终发展成了现在的模样。

**容器变得具有实用性**

很快，更多技术结合进来，让这种隔离方法从构想变为现实。控制组 (cgroups) 是一项内核功能，能够控制和限制一个进程或多组进程的资源使用。而 systemd 初始化系统可设置用户空间，并且管理它们的进程，cgroups 使用该系统来更严密地控制这些隔离进程。这两种技术在增加对 Linux 的整体控制的同时，也成为了保持环境隔离的重要框架。

内核命名空间的改进，推动了容器的进一步发展。利用内核命名空间，从进程 ID 到网络名称，一切都可在 Linux 内核中实现虚拟化。新增的用户命名空间“使得用户和组 ID 可以按命名空间进行映射。对于容器而言，这意味着用户和组可以在容器内部拥有执行某些操作的特权，而在容器外部则没有这种特权。”Linux 容器项目 (LXC) 还添加了用户急需的一些工具、模板、库和语言绑定，从而推动了这些进步，进而改善了使用容器的用户体验。LXC 使得用户能够通过简单的命令行界面轻松地启动容器。

**进入 Docker 技术时代**

2008 年，Docker 公司凭借与公司同名的容器技术通过 dotCloud 登上了舞台。Docker 技术带来了很多新的概念和工具，包括可运行和构建新的分层镜像的简单命令行界面、服务器守护进程、含有预构建容器镜像的库以及注册表服务器概念。通过综合运用这些技术，用户可以快速构建新的分层容器，并轻松地与他人共享这些容器。

红帽意识到了在这个全新的生态系统中协作能够产生的巨大力量，因而在我们的 OpenShift 容器平台中采用了底层技术。为了避免如此重要的技术被单个供应商掌控，Docker Inc. 向社区主导型开源项目提供了很多底层组件（runc 源自开放容器计划，containerd 已移交给 CNCF）。

我们可通过三个主要标准，来确保各种容器技术间的互操作性，即 OCI 镜像、分发和运行时规范。通过遵循上述规范，社区项目、商用产品和云技术提供商可以构建可互操作的容器技术（可将您自行构建的镜像，推送至云技术提供商的注册表服务器——完成这一操作后，镜像才能正常工作）。当前，红帽和 Docker 等公司都是开放容器计划（OCI）的成员，致力于实现容器技术的开放行业标准化。

## 什么是 docker ？

IT 软件中的 “Docker” 是指容器化技术，用于创建和使用 [Linux](https://www.redhat.com/zh/topics/containers)[®](https://www.redhat.com/zh/topics/containers)[容器](https://www.redhat.com/zh/topics/containers)。

借助 Docker，您可将容器当做轻巧、模块化的虚拟机使用。同时，您还将获得高度的灵活性，从而可以高效地创建、部署和复制容器，并能将其从一个环境顺利迁移至另一个环境。

## docker 是如何工作的？

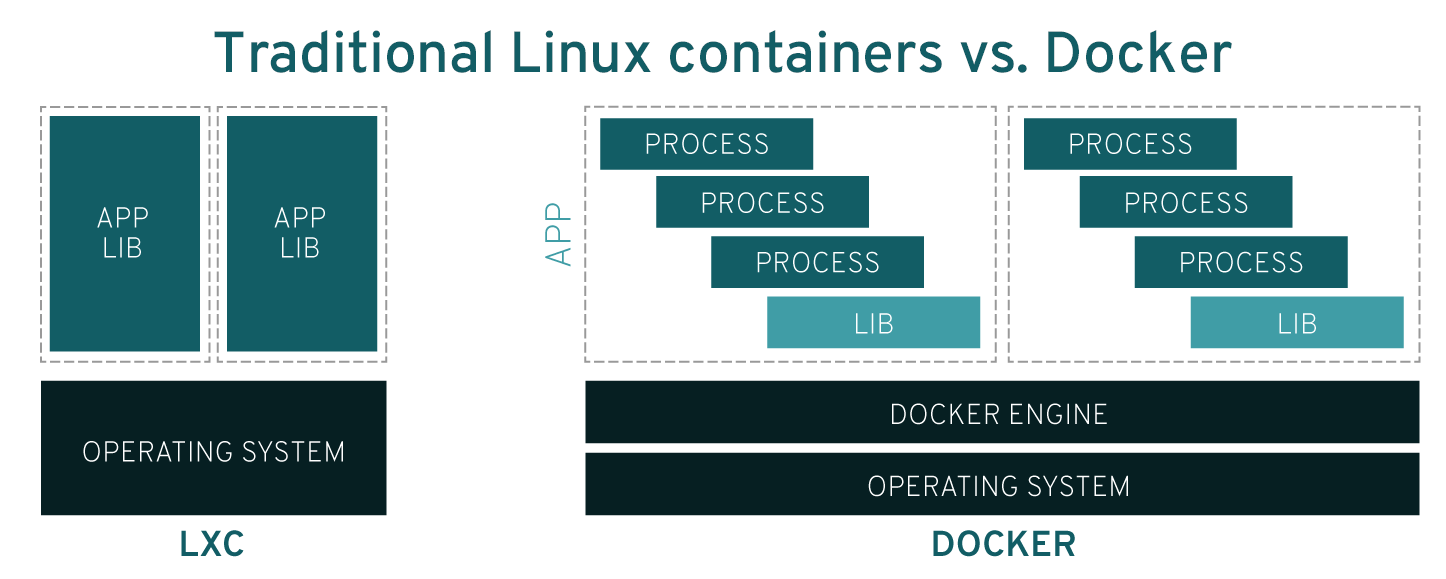
Docker 技术使用 Linux 内核和内核功能（例如 [Cgroups](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html) 和 [namespaces](https://lwn.net/Articles/528078/)）来分隔进程，以便各进程相互独立运行。这种独立性正是采用容器的目的所在；它可以独立运行多个进程、多个应用，更加充分地发挥基础架构的作用，同时保持各个独立系统的安全性。

容器工具（包括 Docker）可提供基于镜像的部署模式。这使得它能够轻松跨多种环境，与其依赖程序共享应用或服务组。Docker 还可在这一容器环境中自动部署应用程序（或者合并多种流程，以构建单个应用程序）。

此外，由于这些工具基于 Linux 容器构建，使得 Docker 既易于使用，又别具一格 -- 它可为用户提供前所未有的应用程访问权限、快速部署以及版本控制和分发能力。

## Docker与传统的 Linux 容器相同？

不同。Docker 技术最初是基于 [LXC](https://linuxcontainers.org/) 技术构建（大多数人都会将这一技术与“传统的” Linux 容器联系在一起），但后来它逐渐摆脱了对这种技术的依赖。就轻量级[虚拟化](https://www.redhat.com/zh/topics/virtualization)这一功能来看，LXC 非常有用，但它无法提供优良的开发人员或用户体验。除了运行容器之外，Docker 技术还具备其他多项功能，包括简化用于构建容器、传输镜像以及控制镜像版本的流程。



传统的 Linux 容器使用 init 系统来管理多种进程。这意味着，所有应用程序都作为一个整体运行。与此相反，Docker 技术力争让应用程序各自独立运行其进程，并提供相应工具，帮助实现这一功能。这种精细化运作模式自有其优势。

## Docker 容器的优势

**模块化**

Docker 容器化方法非常注重在不停止整个应用程序的情况下，单独截取部分应用程序进行更新或修复的能力。除了这种基于微服务的方法，您还可以采用与[面向服务的架构](https://en.wikipedia.org/wiki/Service-oriented_architecture)（SOA）类似的使用方法，在多个应用程序间共享进程。

**层和镜像版本控制**

每个 Docker 镜像文件都包含多个层。这些层组合在一起，构成单个镜像。每当镜像发生改变时，就会创建一个新的镜像层。用户每次发出命令（例如 *run* 或 *copy*）时，都会创建一个新的镜像层。

Docker 重复使用这些层来构建新容器，借此帮助加快流程构建。镜像之间会共享中间变化，从而进一步提升速度、[规模](http://developers.redhat.com/blog/2016/03/09/more-about-docker-images-size/)以及效率。版本控制是镜像层本身自带的能力。每次发生新的更改时，您基本上都会得到一个内置的更改日志，实现对容器镜像的全盘管控。

**回滚**

回滚也许是层最值得一提的功能。每个镜像都拥有多个层。举例而言，如果你不喜欢迭代后的镜像版本，完全可以通过回滚，返回之前的版本。这一功能还支持敏捷开发方法，帮助持续实施集成和部署（[CI](https://en.wikipedia.org/wiki/Continuous_integration)/[CD](https://en.wikipedia.org/wiki/Continuous_delivery)），使其在工具层面成为一种现实。

**快速部署**

启动和运行新硬件、实施部署并投入使用，这在过去一般需要数天时间。投入的心力和成本往往也让人不堪重负。基于 Docker 的容器可将部署时间缩短到几秒。通过为每个进程构建容器，您可以快速将这些类似进程应用到新的应用程序中。而且，由于无需启动操作系统即可添加或移动容器，因此大幅缩短了部署时间。除此之外，得益于这种部署速度，您可以轻松无虞、经济高效地创建和销毁容器创建的数据。

因此，Docker 技术是一种更加精细、可控、基于微服务的技术，可为企业提供更高的效率价值。

## 使用 Docker 时有哪些限制？

Docker 本身非常适合管理单个容器。但随着您开始使用越来越多的容器和容器化应用程序，并把它们划分成数百个部分，很可能会导致管理和编排变得非常困难。最终，您需要后退一步，对容器实施分组，以便跨所有容器提供网络、安全、遥测等服务。于是，Kubernetes 应运而生。

# kubernetes 前言篇

Kubernetes，又称为 k8s（首字母为 k、首字母与尾字母之间有 8 个字符、尾字母为 s，所以简称 k8s）或者简称为 “kube” ，是一种可自动实施 Linux 容器操作的开源平台。它可以帮助用户省去应用容器化过程的许多手动部署和扩展操作。也就是说，您可以将运行 Linux 容器的多组主机聚集在一起，由 Kubernetes 帮助您轻松高效地管理这些集群。而且，这些集群可跨公共云、私有云或混合云部署主机。因此，对于要求快速扩展的云原生应用而言（例如借助 Apache Kafka 进行的实时数据流处理），Kubernetes 是理想的托管平台。

Kubernetes 最初由 Google 的工程师基于 go 语言开发和设计出来并于2014年6月开源。Google 是最早研发 Linux 容器技术的企业之一，曾公开分享介绍 Google 如何将一切都运行于容器之中（这是 Google 云服务背后的技术）。Google 每周会启用超过 20 亿个容器——全都由内部平台 Borg 支撑。Borg 是 Kubernetes 的前身，多年来开发 Borg 的经验教训成了影响 Kubernetes 中许多技术的主要因素。

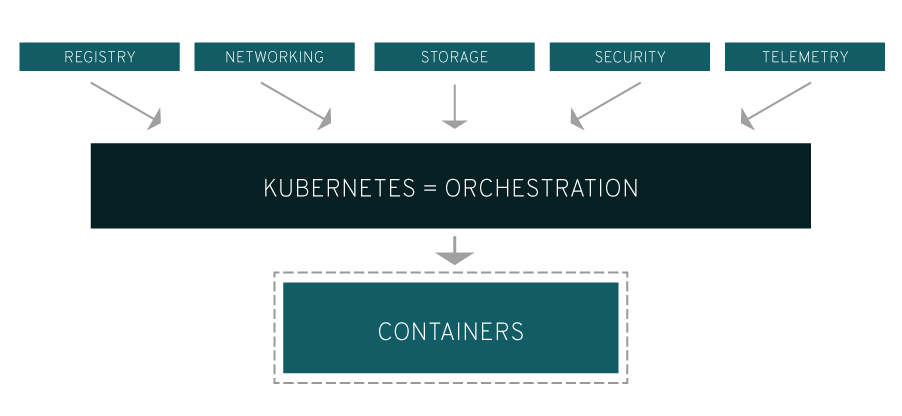
趣闻：Kubernetes 徽标的七个轮辐代表着项目最初的名称“九之七项目”（Project Seven of Nine）。

红帽是第一批与 Google 合作研发 Kubernetes 的公司之一，作为 Kubernetes 上游项目的第二大贡献者，我们甚至在这个项目启动之前就已参与其中。2015 年，Google 将 Kubernetes 项目捐赠给了新成立的云原生计算基金会。

## 为什么需要 kubernetes

真正的生产型应用会涉及多个容器。这些容器必须跨多个服务器主机进行部署。Kubernetes 可以提供所需的编排和管理功能，以便您针对这些工作负载大规模部署容器。借助 Kubernetes 编排功能，您可以构建跨多个容器的应用服务、跨集群调度、扩展这些容器，并长期持续管理这些容器的健康状况。

Kubernetes 还需要与联网、存储、安全性、遥测和其他服务集成整合，以提供全面的容器基础架构。



当然，这取决于您如何在您的环境中使用容器。Linux 容器中的基本应用将它们视作高效、快速的虚拟机。一旦把它部署到生产环境或扩展为多个应用，您显然需要许多托管在相同位置的容器来协同提供各种服务。随着这些容器的累积，您运行环境中容器的数量会急剧增加，复杂度也随之增长。

Kubernetes 通过将容器分类组成 “容器集” （pod），解决了容器增殖带来的许多常见问题容器集为分组容器增加了一个抽象层，可帮助您调用工作负载，并为这些容器提供所需的联网和存储等服务。Kubernetes 的其它部分可帮助您在这些容器集之间达成负载平衡，同时确保运行正确数量的容器，充分支持您的工作负载。

如果能正确实施 Kubernetes，再辅以其它开源项目（例如 Atomic 注册表、Open vSwitch、heapster、OAuth 以及 SELinux），您就能够轻松编排容器基础架构的各个部分。

## kubernetes 有哪些用途

在您生产环境中使用 Kubernetes 的主要优势在于，它提供了一个便捷有效的平台，让您可以在物理机和虚拟机集群上调度和运行容器。更广泛一点说，它可以帮助您在生产环境中，完全实施并依托基于容器的基础架构运营。由于 Kubernetes 的实质在于实现操作任务自动化，所以您可以将其它应用平台或管理系统分配给您的许多相同任务交给容器来执行。

利用 Kubernetes，您能够达成以下目标：

* 跨多台主机进行容器编排。
* 更加充分地利用硬件，最大程度获取运行企业应用所需的资源。
* 有效管控应用部署和更新，并实现自动化操作。
* 挂载和增加存储，用于运行有状态的应用。
* 快速、按需扩展容器化应用及其资源。
* 对服务进行声明式管理，保证所部署的应用始终按照部署的方式运行。
* 利用自动布局、自动重启、自动复制以及自动扩展功能，对应用实施状况检查和自我修复。

但是，Kubernetes 需要依赖其它项目来全面提供这些经过编排的服务。因此，借助其它开源项目可以帮助您将 Kubernetes 的全部功用发挥出来。这些功能包括：

* 注册表，通过 Atomic 注册表或 Docker 注册表等项目实现。
* 联网，通过 OpenvSwitch 和智能边缘路由等项目实现。
* 遥测，通过 heapster、kibana、hawkular 和 elastic 等项目实现。
* 安全性，通过 LDAP、SELinux、RBAC 和 OAUTH 等项目以及多租户层来实现。
* 自动化，参照 Ansible 手册进行安装和集群生命周期管理。
* 服务，可通过自带预建版常用应用模式的丰富内容目录来提供。

## kubernetes 核心概念

Kubernetes 有各类资源对象来描述整个集群的运行状态（Node、Pod、Replication Controller、Service等都可以看作一种“资源对象”）。这些对象都需要通过调用 kubernetes api 来进行创建、修改、删除并将其保存在etcd中持久化存储，可以通过 kubectl 命令工具，也可以直接调用 k8s api，或者使用对象语言的客户端库（例如：golang ， pythion ）。

从这个角度来看，Kubernetes其实是一个高度自动化的资源控制系统，它通过跟踪对比etcd库里保存的“资源期望状态”与当前环境中的“实际资源状态”的差异来实现自动控制和自动纠错的高级功能。

每个 kubernetes 对象都会包含两个关键字段：Object Spec 和 Object Status。spec 描述了对象所期望达到的状态，status 描述了该对象的实际状态。

在介绍资源对象之前，我们先了解一下Kubernetes集群的两种管理角色：Master和Node。

### Master

 Kubernetes里的Master指的是集群控制节点，每个Kubernetes集群里需要有一个Master节点来负责整个集群的管理和控制，基本上Kubernetes的所有控制命令都发给它，它来负责具体的执行过程，我们后面执行的所有命令基本都是在Master节点上运行的。Master节点通常会占据一个独立的服务器（高可用部署建议用3台服务器），其主要原因是它太重要了，是整个集群的“首脑”，如果宕机或者不可用，那么对集群内容器应用的管理都将失效。

Master节点上运行着以下一组关键进程：

* Kubernetes API Server (kube-apiserver)：提供了HTTP Rest接口的关键服务进程，是Kubernetes里所有资源的增、删、改、查等操作的唯一入口，也是集群控制的入口进程。
* Kubernetes Controller Manager (kube-controller-manager)：Kubernetes里所有资源对象的自动化控制中心，可以理解为资源对象的“大总管”。
* Kubernetes Scheduler (kube-scheduler)：负责资源调度（Pod调度）的进程，相当于公交公司的“调度室”。

 另外，在Master节点上还需要启动一个etcd服务，因为Kubernetes里的所有资源对象的数据全部是保存在etcd中的。

### Node

 除了Master，Kubernetes集群中的其他机器被称为Node节点，在较早的版本中也被称为Minion。与Master一样，Node节点可以是一台物理主机，也可以是一台虚拟机。Node节点才是Kubernetes集群中的工作负载节点，每个Node都会被Master分配一些工作负载（Docker容器），当某个Node宕机时，其上的工作负载会被Master自动转移到其他节点上去。

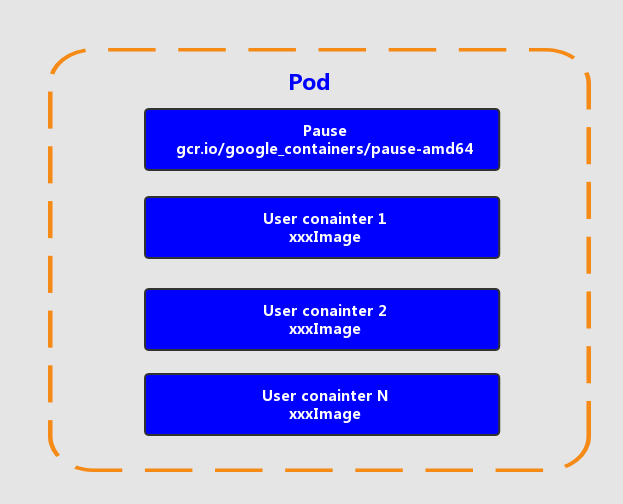
每个Node节点上都运行着以下一组关键进程：

* kubelet：负责Pod对应的容器的创建、启停等任务，同时与Master节点密切协作，实现集群管理的基本功能。
* kube-proxy：实现Kubernetes Service的通信与负载均衡机制的重要组件。
* Docker Engine （docker）：Docker引擎，负责本机的容器创建和管理工作。

 Node节点可以在运行期间动态增加到Kubernetes集群中，前提是这个节点上已经正确安装、配置和启动了上述关键进程，在默认情况下kubelet会向Master注册自己，这也是Kubernetes推荐的Node管理方式。一旦Node被纳入集群管理范围，kubelet进程就会定时向Master节点汇报自身的情报，例如操作系统、Docker版本、机器的CPU和内存情况，以及当前有哪些Pod在运行等，这样Master可以获知每个Node的资源使用情况，并实现高效均衡等资源调度策略。而某个Node超过指定时间不上报信息时，会被Master判断为“失联”，Node的状态被标记为不可用（Not Ready），随后Master会触发“工作负载大转移”的自动流程。

### Pod

 Pod是Kubernetes的最重要也最基本的概念，如下图所示是Pod的组成示意图，我们看到每个Pod都有一个特殊的被成为“根容器”的Pause容器。Pause容器对应的镜像属于Kubernetes平台的一部分，除了Pause容器，每个Pod还包含一个或多个紧密相关的用户业务容器。



Pod的组成

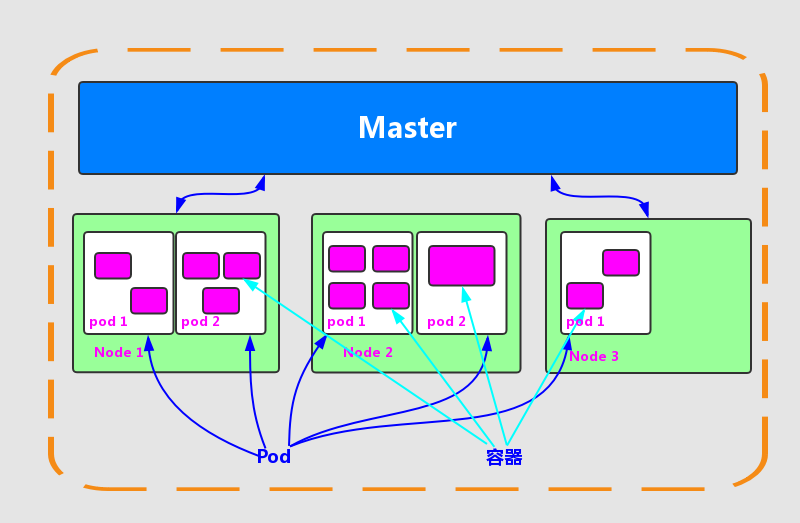
为什么Kubernetes会设计出一个全新的Pod概念并且Pod有这样特殊的组成结构？

 原因之一：在一组容器作为一个单元的情况下，我们难以对“整体”简单地进行判断及有效地进行行动。比如，一个容器死亡了，此时算是整体死亡么？引入业务无关并且不易死亡的Pause容器作为Pod的根容器，以它的状态代表整体容器组的状态，就简单、巧妙地解决了这个难题。

 原因之二：Pod里的多个业务容器共享Pause容器的IP，共享Pause容器挂接的Volume，这样既简化了密切关联的业务容器之间的通信问题，也很好地解决了它们之间的文件共享问题。

 Kubernetes为每个Pod都分配了唯一的IP地址，称之为Pod IP，一个Pod里的多个容器共享Pod IP地址。Kubernetes要求底层网络支持集群内任意两个Pod之间的TCP/IP直接通信，这通常采用虚拟而层网络技术来实现，例如Flannel、Open vSwitch等，因此我们需要牢记一点：在Kubernetes里，一个Pod里的容器与另外主机上的Pod容器能够直接通信。

 Pod其实有两种类型：普通的Pod及静态Pod（Static Pod），后者比较特殊，它并不存放在Kubernetes的etcd存储里，而是存放在某个具体的Node上的一个具体文件中，并且只在此Node上启动运行。而普通的Pod一旦被创建，就会被放入到etcd中存储，随后会被Kubernetes Master调度到某个具体的Node上并进行绑定（Binding），随后该Pod被对应的Node上的kubelet进程实例化成一组相关的Docker容器并且启动起来。在默认情况下，当Pod里的某个容器停止时，Kubernetes会自动检测到这个问题并且重新启动这个Pod（重启Pod里的所有容器），如果Pod所在的Node宕机，则会将这个Node上的所有Pod重新调度到其他节点上。Pod、容器与Node的关系图如下图所示。



Pod、容器与Node的关系

### Label（标签）

 Label是Kubernetes系统中另外一个核心概念。一个Label是一个key=value的键值对，其中key与vaue由用户自己指定。Label可以附加到各种资源对象上，例如Node、Pod、Service、RC等，一个资源对象可以定义任意数量的Label，同一个Label也可以被添加到任意数量的资源对象上去，Label通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除。

 我们可以通过指定的资源对象捆绑一个或多个不同的Label来实现多维度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或者监控和分析应用（日志记录、监控、告警）等。一些常用等label示例如下。

版本标签："release" : "stable" , "release" : "canary"...

环境标签："environment" : "dev" , "environment" : "production"

架构标签："tier" : "frontend" , "tier" : "backend" , "tier" : "middleware"

分区标签："partition" : "customerA" , "partition" : "customerB"...

质量管控标签："track" : "daily" , "track" : "weekly"

 Label相当于我们熟悉的“标签”，給某个资源对象定义一个Label，就相当于給它打了一个标签，随后可以通过Label Selector（标签选择器）查询和筛选拥有某些Label的资源对象，Kubernetes通过这种方式实现了类似SQL的简单又通用的对象查询机制。

### Replication Controller

 RC是Kubernetes系统中的核心概念之一，简单来说，它其实是定义了一个期望的场景，即声明某种Pod的副本数量在任意时刻都符合某个预期值，所以RC的定义包括如下几个部分。

* Pod期待的副本数（replicas）。
* 用于筛选目标Pod的Label Selector。
* 当Pod的副本数量小于预期数量时，用于创建新Pod的Pod模版（template）。

### Deployment

 Deployment是Kubernetes v1.2引入的概念，引入的目的是为了更好地解决Pod的编排问题。为此，Deployment在内部使用了Replica Set来实现目的，无论从Deployment的作用与目的，它的YAML定义，还是从它的具体命令行操作来看，我们都可以把它看作RC的一次升级，两者相似度超过90%。

 Deployment相对于RC的一个最大升级是我们随时知道当前Pod“部署”的进度。实际上由于一个Pod的创建、调度、绑定节点及在目标Node上启动对应的容器这一完整过程需要一定的时间，所以我们期待系统启动N个Pod副本的目标状态，实际上是一个连续变化的“部署过程”导致的最终状态。

 Deployment的典型使用场景有以下几个。

* 创建一个Deployment对象来生成对应的Replica Set并完成Pod副本的创建过程。
* 检查Deployment的状态来看部署动作是否完成（Pod副本的数量是否达到预期的值）。
* 更新Deployment以创建新的Pod（比如镜像升级）。
* 如果当前Deployment不稳定，则回滚到一个早先的Deployment版本。
* 暂停Deployment以便于一次性修改多个PodTemplateSpec的配置项，之后再恢复Deployment，进行新的发布。
* 扩展Deployment以应对高负载。
* 查看Deployment的状态，以此作为发布是否成功的指标。
* 清理不再需要的旧版本ReplicaSets。

### StatefulSet

 在Kubernetes系统中，Pod的管理对象RC、Deployment、DaemonSet和Job都是面向无状态的服务。但现实中有很多服务是有状态的，特别是一些复杂的中间件集群，例如MySQL集群、MongoDB集群、Kafka集群、Zookeeper集群等，这些应用集群有以下一些共同点。

每个节点都有固定的身份ID，通过这个ID，集群中的成员可以相互发现并且通信。

集群的规模是比较固定的，集群规模不能随意变动。

集群里的每个节点都是有状态的，通常会持久化数据到永久存储中。

如果磁盘损坏，则集群里的某个节点无法正常运行，集群功能受损。

 如果用RC/Deployment控制Pod副本数的方式来实现上述有状态的集群，则我们会发现第一点是无法满足的，因为Pod的名字是随机产生的，Pod的IP地址也是在运行期才确定且可能有变动的，我们事先无法为每个Pod确定唯一不变的ID，为了能够在其他节点上恢复某个失败的节点，这种集群中的Pod需要挂接某种共享存储，为了解决这个问题，Kubernetes从v1.4版本开始引入了PetSet这个新的资源对象，并且在v1.5版本时更名为StatefulSet，StatefulSet从本质上来说，可以看作Deployment/RC的一个特殊变种，它有如下一些特性。

StatefulSet里的每个Pod都有稳定、唯一的网络标识，可以用来发现集群内的其他成员。假设StatefulSet的名字叫kafka，那么第一个Pod叫kafak-0，第二个Pod叫kafak-1，以此类推。

StatefulSet控制的Pod副本的启停顺序是受控的，操作第n个Pod时，前n-1个Pod已经时运行且准备好的状态。

StatefulSet里的Pod采用稳定的持久化存储卷，通过PV/PVC来实现，删除Pod时默认不会删除与StatefulSet相关的存储卷（为了保证数据的安全）。

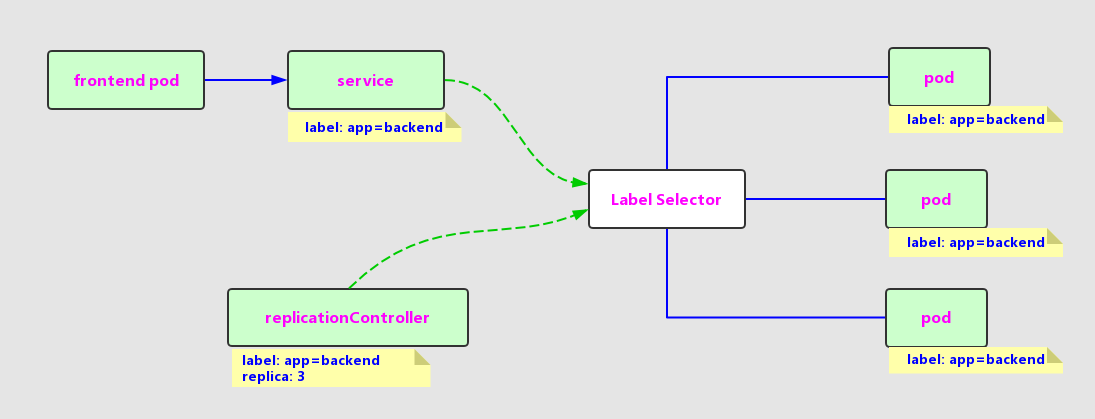
 StatefulSet除了要与PV卷捆绑使用以存储Pod的状态数据，还要与Headless Service配合使用，即在每个StatefulSet的定义中要声明它属于哪个Headless Service。Headless Service与普通Service的关键区别在于，它没有Cluster IP，如果解析Headless Service的DNS域名，则返回的是该Service对应的全部Pod的Endpoint列表。StatefulSet在Headless Service的基础上又为StatefulSet控制的每个Pod实例创建了一个DNS域名，这个域名的格式为：

$(podname).$(headless service name)

 比如一个3节点的Kafka的StatefulSet集群，对应的Headless Service的名字为kafka，StatefulSet的名字为kafka，则StatefulSet里面的3个Pod的DNS名称分别为kafka-0.kafka、kafka-1.kafka、kafka-3.kafka，这些DNS名称可以直接在集群的配置文件中固定下来。

### Service(服务)

 Service也是Kubernetes里的最核心的资源对象之一，Kubernetes里的每个Service其实就是我们经常提起的微服务架构中的一个“微服务”，之前我们所说的Pod、RC等资源对象其实都是为这节所说的“服务”------Kubernetes Service作“嫁衣”的。下图显示了Pod、RC与Service的逻辑关系。



Pod、RC与Service的关系

 从图中我们看到，Kubernetes的Service定义了一个服务的访问入口地址，前端的应用（Pod）通过这个入口地址访问其背后的一组由Pod副本组成的集群实例，Service与其后端Pod副本集群之间则是通过Label Selector来实现“无缝对接”的。而RC的作用实际上是保证Service的服务能力和服务质量始终处于预期的标准。

### Volume(存储卷)

 Volume是Pod中能够被多个容器访问的共享目录。Kubernetes的Volume概念、用途和目的与Docker的Volume比较类似，但两者不能等价。首先，Kubernetes中的Volume定义在Pod上，然后被一个Pod里的多个容器挂载到具体的文件目录下；其次，Kubernetes中的Volume中的数据也不会丢失。最后，Kubernetes支持多种类型的Volume，例如Gluster、Ceph等先进的分布式文件系统。

### Namespace

 Namespace（命名空间）是Kubernetes系统中的另一个非常重要的概念，Namespace在很多情况下用于实现多租户的资源隔离。Nameaspace通过将集群内部的资源对象“分配”到不同的Namespce中，形成逻辑上分组的不同项目、小组或用户组，便于不同的分组在共享使用整个集群的资源的同时还能被分别管理。

### Annotation（注解）

 Annotation与Label类似，也使用key/value键值对的形式进行定义。不同的是Label具有严格的命名规则，它定义的是Kubernetes对象的元数据（Metadata），并且用于Label Selector。而Annotation则是用户任意定义的“附加”信息，以便于外部工具进行查找，很多时候，Kubernetes的模块自身会通过Annotation的方式标记资源对象的特殊信息。

 通常来说，用Annotation来记录的信息如下。

* build信息、release信息、Docker镜像信息等，例如时间戳、release id号、PR号、镜像hash值、docker registry地址等。
* 日志库、监控库、分析库等资源库的地址信息。
* 程序调试工具信息，例如工具、版本号等。
* 团队等联系信息，例如电话号码、负责人名称、网址等。

### 其他

* **Kubelet**

运行在节点上的服务，可读取容器清单（container manifest），确保指定的容器启动并运行。

* **kubectl**

Kubernetes 的命令行配置工具。

上述组件是Kubernetes系统的核心组件，它们共同构成了Kubernetes系统的框架和计算模型。通过对它们进行灵活组合，用户就可以快速、方便地对容器集群进行配置、创建和管理。除了本章所介绍的核心组件，在Kubernetes系统中还有许多辅助配置的资源对象，例如LimitRange、Resurce。另外，一些系统内部使用的对象Binding、Event等请参考Kubernetes的API文档。

## kubernetes 术语

官方把 kubernetes 术语分为 12 个分类：

系统结构、社区、核心对象、扩展、基础、网络、操作、安全、存储、工具、用户类型、工作负载

由于 k8s 的术语实在太多了，想要全部记住作为新学者还是有点压力，所以我们课程中只讲解一些常用的术语。想要了解更多 kubernetes 术语，请参照官方术语表。

地址：<https://kubernetes.io/docs/reference/glossary/?fundamental=true>

### Pods

在Kubernetes中，最小的管理元素不是一个个独立的容器，而是Pod,Pod是最小的，管理，创建，计划的最小单元。

### Labels

标签其实就一对 key/value ，被关联到对象上，比如Pod；标签的使用我们倾向于能够标示对象的特殊特点，并且对用户而言是有意义的（就是一眼就看出了这个Pod是数据库），但是标签对内核系统是没有直接意义的。标签可以用来划分特定组的对象（比如，所有女的），标签可以在创建一个对象的时候直接给与，也可以在后期随时修改，每一个对象可以拥有多个标签，但是，key值必须是唯一的

"labels": {

"key1" : "value1",

"key2" : "value2"

}

### Namespace

Namespace 是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。常见的pods, services, replication controllers和deployments等都是属于某一个namespace的（默认是default），而node, persistentVolumes等则不属于任何namespace。

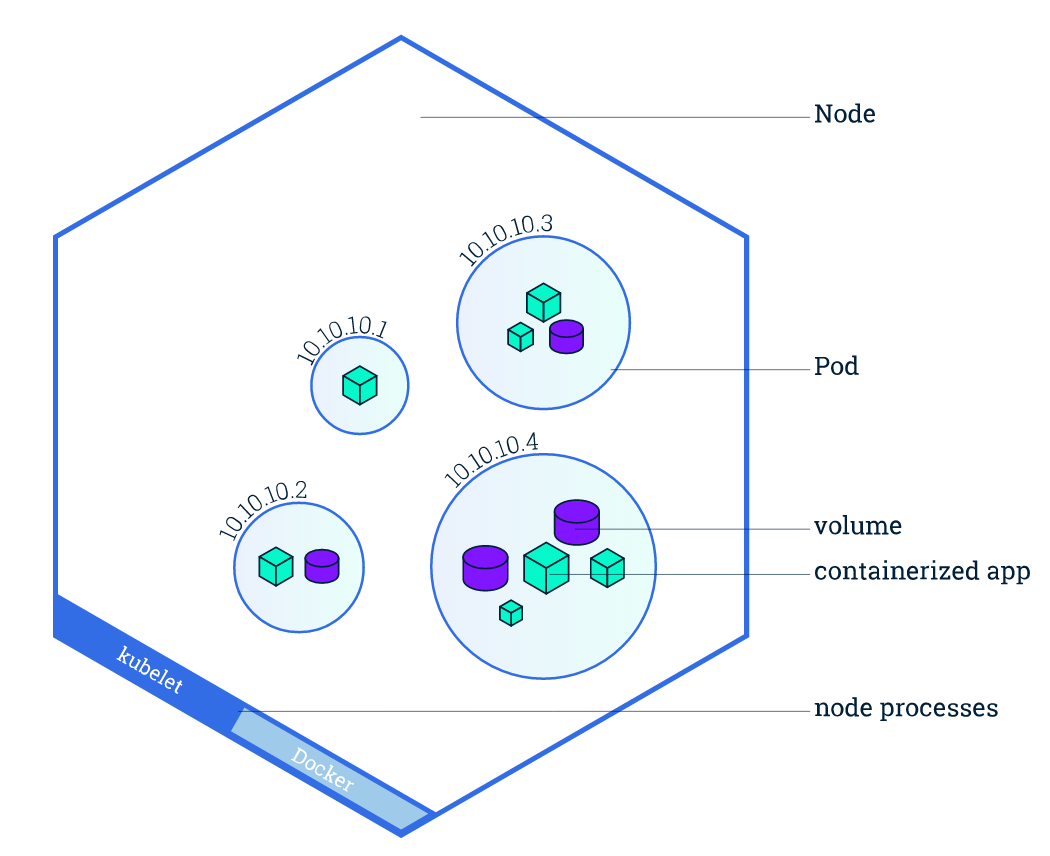
Namespace常用来隔离不同的用户，比如Kubernetes自带的服务一般运行在kube-system namespace中。

### Replication Controller

Replication Controller 保证了在所有时间内，都有特定数量的Pod副本正在运行，如果太多了，Replication Controller就杀死几个，如果太少了，Replication Controller会新建几个，和直接创建的pod不同的是，Replication Controller会替换掉那些删除的或者被终止的pod，不管删除的原因是什么（维护阿，更新啊，Replication Controller都不关心）。基于这个理由，我们建议即使是只创建一个pod，我们也要使用Replication Controller。Replication Controller 就像一个进程管理器，监管着不同node上的多个pod,而不是单单监控一个node上的pod，Replication Controller 会委派本地容器来启动一些节点上服务（Kubelet ,Docker）。

### Node

Node是Pod真正运行的主机，可以物理机，也可以是虚拟机。为了管理Pod，每个Node节点上至少要运行container runtime（比如docker或者rkt）、kubelet和kube-proxy服务。



### ReplicaSets

ReplicaSet是下一代复本控制器。ReplicaSet和 Replication Controller之间的唯一区别是现在的选择器支持。Replication Controller只支持基于等式的selector（env=dev或environment!=qa），但ReplicaSet还支持新的，基于集合的selector（version in (v1.0, v2.0)或env notin (dev, qa)）。在试用时官方推荐ReplicaSet。

### Services

Kubernetes Pod是平凡的，它门会被创建，也会死掉（生老病死），并且他们是不可复活的。 ReplicationControllers动态的创建和销毁Pods(比如规模扩大或者缩小，或者执行动态更新)。每个pod都由自己的ip，这些IP也随着时间的变化也不能持续依赖。这样就引发了一个问题：如果一些Pods（让我们叫它作后台，后端）提供了一些功能供其它的Pod使用（让我们叫作前台），在kubernete集群中是如何实现让这些前台能够持续的追踪到这些后台的？

答案是：**Service**

Kubernete Service 是一个定义了一组Pod的策略的抽象，我们也有时候叫做宏观服务。这些被服务标记的Pod都是（一般）通过label Selector决定的（下面我们会讲到我们为什么需要一个没有label selector的服务）

举个例子，我们假设后台是一个图形处理的后台，并且由3个副本。这些副本是可以相互替代的，并且前台并需要关心使用的哪一个后台Pod，当这个承载前台请求的pod发生变化时，前台并不需要直到这些变化，或者追踪后台的这些副本，服务是这些去耦

对于Kubernete原生的应用，Kubernete提供了一个简单的Endpoints API，这个Endpoints api的作用就是当一个服务中的pod发生变化时，Endpoints API随之变化，对于哪些不是原生的程序，Kubernetes提供了一个基于虚拟IP的网桥的服务，这个服务会将请求转发到对应的后台pod。

### Volumes

容器中的磁盘的生命周期是短暂的，这就带来了一系列的问题，第一，当一个容器损坏之后，kubelet 会重启这个容器，但是文件会丢失-这个容器会是一个全新的状态，第二，当很多容器在同一Pod中运行的时候，很多时候需要数据文件的共享。Kubernete Volume解决了这个问题。

### PV/PVC/StorageClass

PersistentVolume（PV）是集群中已由管理员配置的一段网络存储。 集群中的资源就像一个节点是一个集群资源。 PV是诸如卷之类的卷插件，但是具有独立于使用PV的任何单个pod的生命周期。 该API对象捕获存储的实现细节，即NFS，iSCSI或云提供商特定的存储系统。

PersistentVolumeClaim（PVC）是用户存储的请求。 它类似于pod。 Pod消耗节点资源，PVC消耗光伏资源。 荚可以请求特定级别的资源（CPU和内存）。 权利要求可以请求特定的大小和访问模式（例如，可以一旦读/写或只读许多次安装）。

虽然PersistentVolumeClaims允许用户使用抽象存储资源，但是常见的是，用户需要具有不同属性（如性能）的PersistentVolumes，用于不同的问题。 群集管理员需要能够提供多种不同于PersistentVolumes的PersistentVolumes，而不仅仅是大小和访问模式，而不会使用户了解这些卷的实现细节。 对于这些需求，存在StorageClass资源。

StorageClass为管理员提供了一种描述他们提供的存储的“类”的方法。 不同的类可能映射到服务质量级别，或备份策略，或者由群集管理员确定的任意策略。 Kubernetes本身对于什么类别代表是不言而喻的。 这个概念有时在其他存储系统中称为“配置文件”

例子

<https://kubernetes.io/docs/user-guide/persistent-volumes/walkthrough/>

### Deployment

Deployment为Pod和ReplicaSet提供了一个声明式定义(declarative)方法，用来替代以前的ReplicationController来方便的管理应用。典型的应用场景包括：

* 定义Deployment来创建Pod和ReplicaSet
* 滚动升级和回滚应用
* 扩容和缩容
* 暂停和继续Deployment

比如一个简单的nginx应用可以定义为：

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

扩容：

kubectl scale deployment nginx-deployment --replicas 10

如果集群支持 horizontal pod autoscaling 的话，还可以为Deployment设置自动扩展：

kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80

更新镜像也比较简单:

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

回滚：

kubectl rollout undo deployment/nginx-deployment

### Secret

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

Secret有三种类型：

* Service Account：用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的/run/secrets/kubernetes.io/serviceaccount目录中；
* Opaque：base64编码格式的Secret，用来存储密码、密钥等；
* kubernetes.io/dockerconfigjson：用来存储私有docker registry的认证信息。

### StatefulSet

StatefulSet是为了解决有状态服务的问题（对应Deployments和ReplicaSets是为无状态服务而设计），其应用场景包括：

* 稳定的持久化存储，即Pod重新调度后还是能访问到相同的持久化数据，基于PVC来实现
* 稳定的网络标志，即Pod重新调度后其PodName和HostName不变，基于Headless Service（即没有Cluster IP的Service）来实现
* 有序部署，有序扩展，即Pod是有顺序的，在部署或者扩展的时候要依据定义的顺序依次依次进行（即从0到N-1，在下一个Pod运行之前所有之前的Pod必须都是Running和Ready状态），基于init containers来实现
* 有序收缩，有序删除（即从N-1到0）

从上面的应用场景可以发现，StatefulSet由以下几个部分组成：

* 用于定义网络标志（DNS domain）的Headless Service
* 用于创建PersistentVolumes的volumeClaimTemplates
* 定义具体应用的StatefulSet

### DaemonSet

DaemonSet保证在每个Node上都运行一个容器副本，常用来部署一些集群的日志、监控或者其他系统管理应用。典型的应用包括：

* 日志收集，比如fluentd，logstash等
* 系统监控，比如Prometheus Node Exporter，collectd，New Relic agent，Ganglia gmond等
* 系统程序，比如kube-proxy, kube-dns, glusterd, ceph等

### Service Account

Service account是为了方便Pod里面的进程调用Kubernetes API或其他外部服务而设计的。

### CronJob

CronJob即定时任务，就类似于Linux系统的crontab，在指定的时间周期运行指定的任务。在Kubernetes 1.5，使用CronJob需要开启batch/v2alpha1 API，即–runtime-config=batch/v2alpha1。

### Job

Job负责批量处理短暂的一次性任务 (short lived one-off tasks)，即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

Kubernetes支持以下几种Job：

* 非并行Job：通常创建一个Pod直至其成功结束
* 固定结束次数的Job：设置.spec.completions，创建多个Pod，直到.spec.completions个Pod成功结束
* 带有工作队列的并行Job：设置.spec.Parallelism但不设置.spec.completions，当所有Pod结束并且至少一个成功时，Job就认为是成功

### Security Context 和 PSP

Security Context的目的是限制不可信容器的行为，保护系统和其他容器不受其影响。

Kubernetes提供了三种配置Security Context的方法：

* Container-level Security Context：仅应用到指定的容器
* Pod-level Security Context：应用到Pod内所有容器以及Volume
* Pod Security Policies（PSP）：应用到集群内部所有Pod以及Volume

Pod Security Policies（PSP）是集群级的Pod安全策略，自动为集群内的Pod和Volume设置Security Context。

使用PSP需要API Server开启extensions/v1beta1/podsecuritypolicy，并且配置PodSecurityPolicyadmission控制器。

### Resource Quotas

资源配额（Resource Quotas）是用来限制用户资源用量的一种机制。

它的工作原理为：

* 资源配额应用在Namespace上，并且每个Namespace最多只能有一个ResourceQuota对象
* 开启计算资源配额后，创建容器时必须配置计算资源请求或限制（也可以用LimitRange设置默认值）
* 用户超额后禁止创建新的资源

### Network Policy

Network Policy提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。

在使用Network Policy之前，需要注意：

* apiserver开启extensions/v1beta1/networkpolicies
* 网络插件要支持Network Policy，如Calico、Romana、Weave Net和trireme等

### Ingress

通常情况下，service和pod的IP仅可在集群内部访问。集群外部的请求需要通过负载均衡转发到service在Node上暴露的NodePort上，然后再由kube-proxy将其转发给相关的Pod。

而Ingress就是为进入集群的请求提供路由规则的集合，如下图所示



Ingress可以给service提供集群外部访问的URL、负载均衡、SSL终止、HTTP路由等。为了配置这些Ingress规则，集群管理员需要部署一个Ingress controller，它监听Ingress和service的变化，并根据规则配置负载均衡并提供访问入口。

### ThirdPartyResources

ThirdPartyResources是一种无需改变代码就可以扩展Kubernetes API的机制，可以用来管理自定义对象。

hirdPartyResources将在v1.7弃用

ThirdPartyResources将在v1.7弃用，并在未来版本中删除。建议从v1.7开始，迁移到CustomResourceDefinition。

### ConfigMap

ConfigMap用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。ConfigMap跟secret很类似，但它可以更方便地处理不包含敏感信息的字符串。

### PodPreset

PodPreset用来给指定标签的Pod注入额外的信息，如环境变量、存储卷等。这样，Pod模板就不需要为每个Pod都显式设置重复的信息。

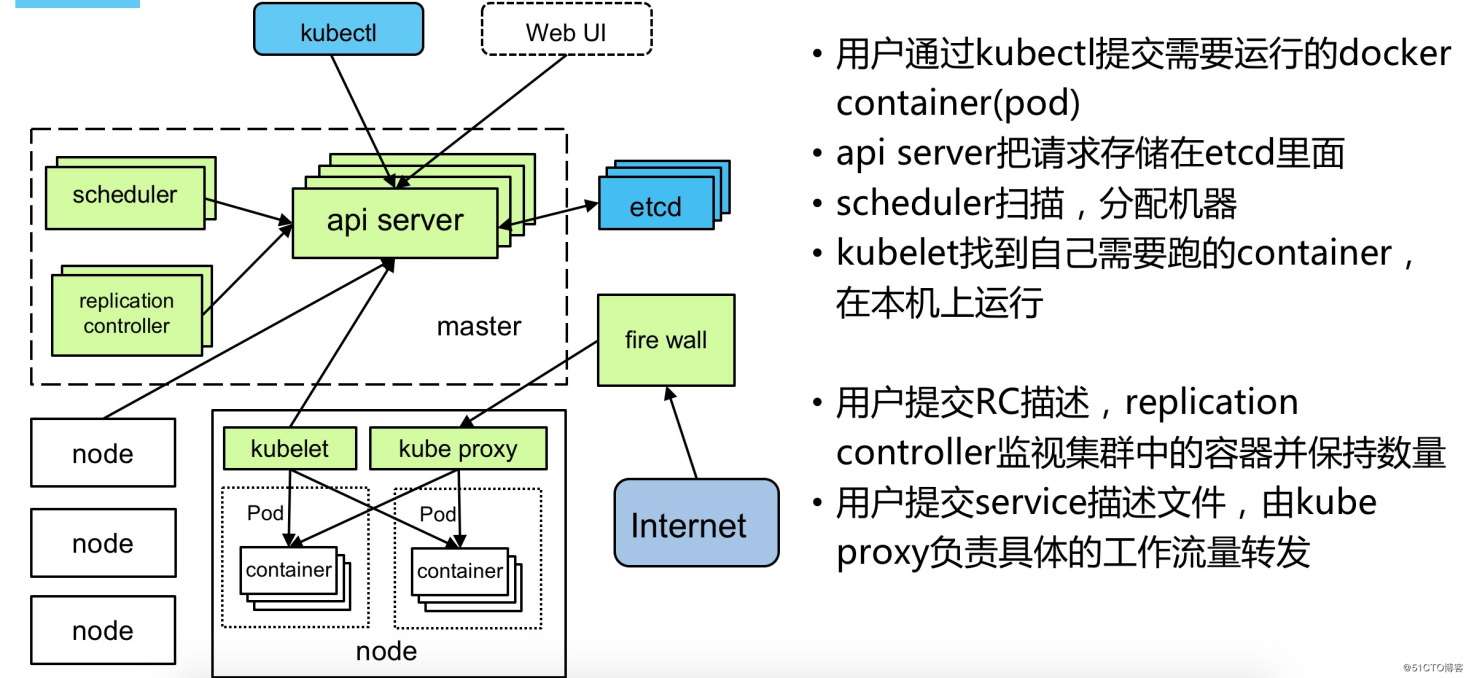
# kubernetes 入门

在了解了 kubernetes 的一些核心概念和名词术语后，本章节中我们深入学习下 kubernetes 的集群架构及环境搭建以及运行一个简单 docker 容器。

在学习下面的章节前，希望大家掌握以下知识和准备以下环境：

* Linux 操作系统原理
* Linux 常用命令
* Docker 容器原理及基本操作
* 一台可以上网的电脑，Mac/Windows/Linux 皆可
* 安装 Docker

## K8S 集群架构方案



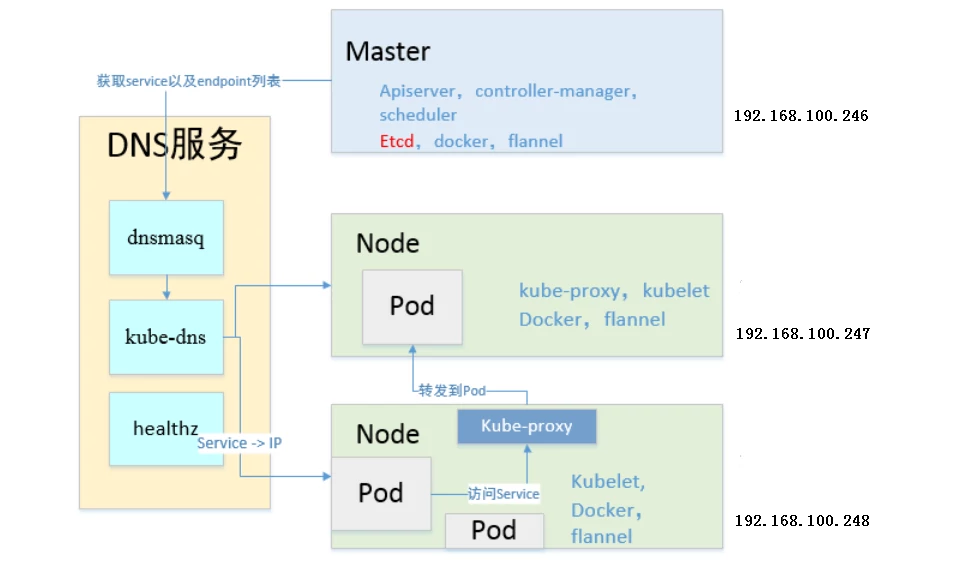
kubernetes集群架构

Kubernetes 集群组件：

* etcd 一个高可用的K/V键值对存储和服务发现系统
* flannel 实现夸主机的容器网络的通信
* kube-apiserver 提供kubernetes集群的API调用
* kube-controller-manager 确保集群服务
* kube-scheduler 调度容器，分配到Node
* kubelet 在Node节点上按照配置文件中定义的容器规格启动容器
* kube-proxy 提供网络代理服务

**Kubernetes 集群部署方案**

如下是集群部署策略，1个master + 2个node。存储集群etcd是单点集群（真实环境不推荐此做法，需要集群）。网络使用的是flannel虚拟二次网络。



K8s集群部署架构

**Kubernetes具有完备的集群管理能力：**

1. 包括多层次的安全防护和准入机制
2. 多租户应用支撑能力
3. 透明的服务注册和服务发现机制
4. 内建智能负载均衡器
5. 强大的故障发现和自我修复能力
6. 服务滚动升级和在线扩容能力
7. 可扩展的资源自动调度机制
8. 以及多粒度的资源管理能力

同时，kubernetes提供了完善的管理工具，这些工具涵盖了包括开发、部署测试、运维监控在内的各个环节。

在kubernetes中，service（服务）是分布式集群架构的核心，一个service对象拥有如下关键特征：

拥有一个唯一指定的名字（比如mysql-service）。

拥有一个虚拟IP（Cluster IP、service IP或VIP）和端口号。

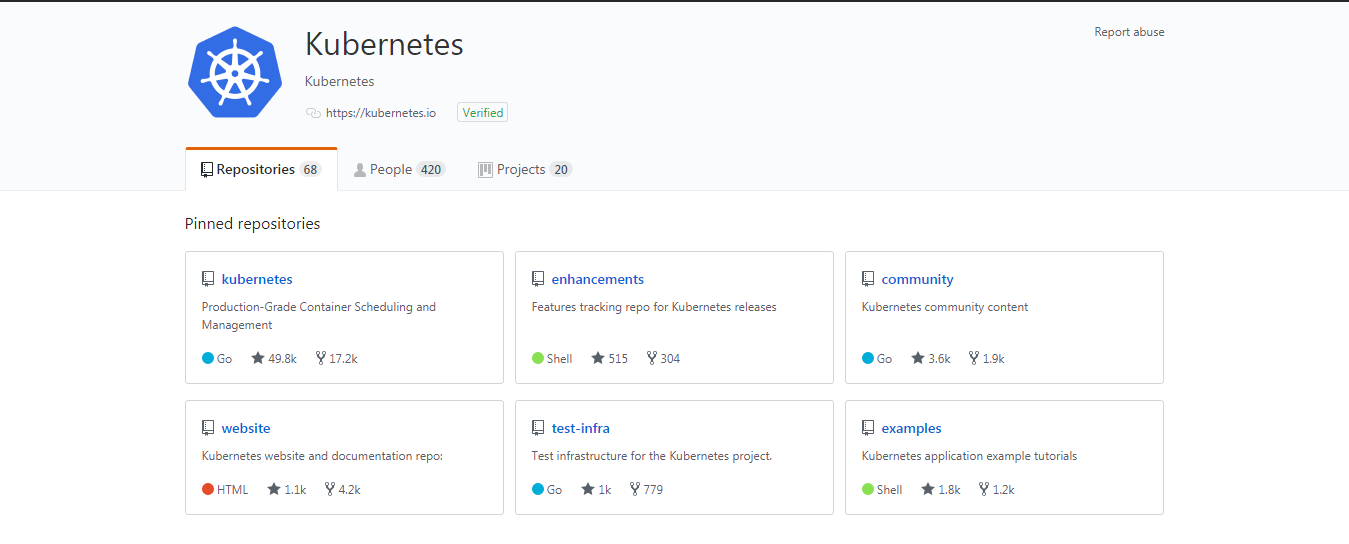
能够提供某种远程服务能力。

被映射到了提供这种服务能力的一组容器应用上。

Kubernetes.io开发了一个交互式教程，通过WEB浏览器就能使用预先部署好的一个Kubernetes集群，快速体验kubernetes的功能和应用场景。

链接：<https://kubernetes.io/docs/tutorials/kubernetes-basics/>

K8s官方下载地址：<https://github.com/kubernetes>



**环境准备**

|  |  |  |
| --- | --- | --- |
| 节点 | ip 地址 | 操作系统 |
| master | **~~192.168.100.246~~** | CentOS 7.3-x86\_64 |
| node1 | **~~192.168.100.247~~** | CentOS 7.3-x86\_64 |
| node2 | **~~192.168.100.248~~** | CentOS 7.3-x86\_64 |
| harbor | **~~192.168.100.241~~** | CentOS 7.3-x86\_64 |

**集群详情**

* OS：CentOS Linux release 7.3.1611 (Core) 3.10.0-514.el7.x86\_64
* Kubernetes 1.6.0+（最低的版本要求是1.6）
* Docker：建议使用 Docker CE
* Etcd 3.3.10
* Flannel 0.7.1 vxlan或者host-gw 网络
* TLS 认证通信 (所有组件，如 etcd、kubernetes master 和 node)
* RBAC 授权
* kubelet TLS BootStrapping
* kubedns、dashboard、heapster(influxdb、grafana)、EFK(elasticsearch、fluentd、kibana) 集群插件
* 私有docker镜像仓库harbor（请自行部署，harbor提供离线安装包，直接使用docker-compose启动即可），不会的请参官文档：

安装文档：<https://github.com/goharbor/harbor/blob/master/docs/installation_guide.md>

配置 https访问：<https://github.com/goharbor/harbor/blob/master/docs/configure_https.md>

**环境说明**

在下面的步骤中，我们将在三台CentOS系统的物理机上部署具有三个节点的kubernetes1.12.3集群。

角色分配如下：

镜像仓库：~~192.168.100.241~~，为私有镜像仓库，请替换为公共仓库或你自己的镜像仓库地址。

Master：**~~192.168.100.246~~**

Node：~~192.168.100.247~~、~~192.168.100.248~~

注意：**~~192.168.100.246~~** 这台主机 master 和 node 复用。所有生成证书、执行 kubectl 命令的操作都在这台节点上执行。一旦 node 加入到 kubernetes 集群之后就不需要再登陆node节点了。

## 集群环境搭建

本文档介绍使用二进制部署最新 kubernetes v1.12.3 集群的所有步骤，而不是使用 kubeadm 等自动化方式来部署集群。

在部署的过程中，将详细列出各组件的启动参数，它们的含义和可能遇到的问题。

部署完成后，你将理解系统各组件的交互原理，进而能快速解决实际问题。

所以本文档主要适合于那些有一定 kubernetes 基础，想通过一步步部署的方式来学习和了解系统配置、运行原理的人。

**提醒**

1. 本文档适用于 CentOS 7.x、Ubuntu 18.x 及以上版本系统
2. 由于启用了 TLS 双向认证、RBAC 授权等严格的安全机制，建议从头开始部署，而不要从中间开始，否则可能会认证、授权等失败！
3. 部署过程中需要有很多证书的操作，请大家耐心操作，不明白的操作可以参考本书中的其他章节的解释。
4. 该部署操作仅是搭建成了一个可用 kubernetes 集群，而很多地方还需要进行优化，heapster 插件、EFK 插件不一定会用于真实的生产环境中，但是通过部署这些插件，可以让大家了解到如何部署应用到集群上。

**特别提醒**

**有同学没有按照老师的k8s版本（1.12.3）安装，自己升级了版本；有些地方k8s升级后官方默认功能做了调整，比如kube-apiserver中的1.14版本 不支持Initializers插件 等......；**

**由于k8s升级和功能迭代快，所以同学们如果必要请不要自行升级版本，自行升级将可能导致参照文档安装提示报错。**

**文档需要根据个人环境修改的地方都用 ~~中划线~~ +【粗体 or 标红】表示，大家看到这种信息的时候记得修改，务必不能使用文档相同的信息。。。。。。**

### 组件版本和配置策略

#### 组件版本

* Kubernetes 1.12.3
* Docker 18.09.0-ce
* Etcd 3.3.10
* Flanneld 0.10.0
* 插件：
  + Coredns
  + Dashboard
  + Heapster (influxdb、grafana)
  + Metrics-Server
  + EFK (elasticsearch、fluentd、kibana)
* 镜像仓库：
* docker registry
* harbor

#### 主要配置策略

kube-apiserver：

* 使用节点本地 nginx 4 层透明代理实现高可用；
* 关闭非安全端口 8080 和匿名访问；
* 在安全端口 6443 接收 https 请求；
* 严格的认证和授权策略 (x509、token、RBAC)；
* 开启 bootstrap token 认证，支持 kubelet TLS bootstrapping；
* 使用 https 访问 kubelet、etcd，加密通信；

kube-controller-manager：

* + 3 节点高可用；
  + 关闭非安全端口，在安全端口 10252 接收 https 请求；
  + 使用 kubeconfig 访问 apiserver 的安全端口；
  + 自动 approve kubelet 证书签名请求 (CSR)，证书过期后自动轮转；
  + 各 controller 使用自己的 ServiceAccount 访问 apiserver；

kube-scheduler：

* 3 节点高可用；
* 使用 kubeconfig 访问 apiserver 的安全端口；

kubelet：

* 使用 kubeadm 动态创建 bootstrap token，而不是在 apiserver 中静态配置；
* 使用 TLS bootstrap 机制自动生成 client 和 server 证书，过期后自动轮转；
* 在 KubeletConfiguration 类型的 JSON 文件配置主要参数；
* 关闭只读端口，在安全端口 10250 接收 https 请求，对请求进行认证和授权，拒绝匿名访问和非授权访问；
* 使用 kubeconfig 访问 apiserver 的安全端口；

kube-proxy：

* 使用 kubeconfig 访问 apiserver 的安全端口；
* 在 KubeProxyConfiguration 类型的 JSON 文件配置主要参数；
* 使用 ipvs 代理模式；

集群插件：

* DNS：使用功能、性能更好的 coredns；
* Dashboard：支持登录认证；
* Metric：heapster、metrics-server，使用 https 访问 kubelet 安全端口；
* Log：Elasticsearch、Fluend、Kibana；
* Registry 镜像库：docker-registry、harbor；

harbor私有镜像仓库：参考：<https://github.com/goharbor/harbor>

### 系统初始化和全局变量

#### 集群机器

master：**~~192.168.100.246~~**

node1：**~~192.168.100.247~~**

node2：**~~192.168.100.248~~**

请提前在 VirtualBox 中安装三台基于 centos 7 镜像的虚拟机，本文档中的 etcd 集群、master 节点、worker 节点均使用这三台机器。

注意：

1. 需要在所有机器上执行本文档的初始化命令；
2. 需要使用具有 root 权限的账号执行这些命令。

#### 主机名

设置永久主机名称，然后重新登录:

hostnamectl set-hostname master # 将 master 替换为当前主机名

* 设置的主机名保存在 /etc/hostname 文件中；

如果 DNS 不支持解析主机名称，则需要修改每台机器的 /etc/hosts 文件，添加主机名和 IP 的对应关系：

cat >> /etc/hosts <<EOF

~~192.168.100.246~~ master

~~192.168.100.247~~ node1

~~192.168.100.248~~ node2

EOF

#### 添加 docker 账户

在每台机器上添加 docker 账户：

useradd –m docker

#### 无密码 ssh 登录其它节点

如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令，所以需要添加该节点到其它节点的 ssh 信任关系。

设置 master 可以无密码登录所有节点的 root 账户：

ssh-keygen -t rsa

ssh-copy-id root@master

ssh-copy-id root@node1

ssh-copy-id root@node2

#### 将可执行文件路径 /opt/k8s/bin 添加到 PATH 变量中

在每台机器上添加环境变量：

echo 'PATH=/opt/k8s/bin:$PATH' >>/root/.bashrc

Source /root/.bashrc

#### 安装依赖包

在每台机器上安装依赖包：

CentOS:

yum install -y epel-release

yum install -y conntrack ntpdate ntp ipvsadm ipset jq iptables curl sysstat libseccomp wget

/usr/sbin/modprobe ip\_vs

Ubuntu:

apt-get install -y conntrack ipvsadm ntp ipset jq iptables curl sysstat libseccomp

/usr/sbin/modprobe ip\_vs

* ipvs 依赖 ipset；
* ntp 保证各机器系统时间同步；

#### 关闭防火墙

在每台机器上关闭防火墙，清理防火墙规则，设置默认转发策略：

systemctl stop firewalld

systemctl disable firewalld

iptables -F && iptables -X && iptables -F -t nat && iptables -X -t nat

iptables -P FORWARD ACCEPT

#### 关闭 swap 分区

如果开启了 swap 分区，kubelet 会启动失败(可以通过将参数 --fail-swap-on 设置为 false 来忽略 swap on)，故需要在每台机器上关闭 swap 分区。同时注释 /etc/fstab 中相应的条目，防止开机自动挂载 swap 分区：

swapoff -a

sed -i '/ swap / s/^\(.\*\)$/#\1/g' /etc/fstab

#### 关闭 SELinux

关闭 SELinux，否则后续 K8S 挂载目录时可能报错 Permission denied：

setenforce 0

sed -i 's/^SELINUX=.\*/SELINUX=disabled/' /etc/selinux/config

#### 关闭 dnsmasq（可选）

linux 系统开启了 dnsmasq 后(如 GUI 环境)，将系统 DNS Server 设置为 127.0.0.1，这会导致 docker 容器无法解析域名，需要关闭它：

systemctl stop dnsmasq

systemctl disable dnsmasq

#### 加载内核模块

modprobe ip\_vs\_rr

modprobe br\_netfilter

#### 优化内核参数

这一步如果执行有一些报错不用理会，继续往下执行。

cat > kubernetes.conf <<EOF

net.bridge.bridge-nf-call-iptables=1

net.bridge.bridge-nf-call-ip6tables=1

net.ipv4.ip\_forward=1

net.ipv4.tcp\_tw\_recycle=0

vm.swappiness=0 # 禁止使用 swap 空间，只有当系统 OOM 时才允许使用它

vm.overcommit\_memory=1 # 不检查物理内存是否够用

vm.panic\_on\_oom=0 # 开启 OOM

fs.inotify.max\_user\_instances=8192

fs.inotify.max\_user\_watches=1048576

fs.file-max=52706963

fs.nr\_open=52706963

net.ipv6.conf.all.disable\_ipv6=1

net.netfilter.nf\_conntrack\_max=2310720

EOF

cp kubernetes.conf /etc/sysctl.d/kubernetes.conf

sysctl -p /etc/sysctl.d/kubernetes.conf

* 必须关闭 tcp\_tw\_recycle，否则和 NAT 冲突，会导致服务不通；
* 关闭 IPV6，防止触发 docker BUG；

#### 设置系统时区

# 调整系统 TimeZone

timedatectl set-timezone Asia/Shanghai

# 将当前的 UTC 时间写入硬件时钟

timedatectl set-local-rtc 0

# 重启依赖于系统时间的服务

systemctl restart rsyslog

systemctl restart crond

#### 更新系统时间（可选）

ntpdate cn.pool.ntp.org

#### 关闭无关的服务

systemctl stop postfix && systemctl disable postfix

#### 设置 rsyslogd 和 systemd journal

systemd 的 journald 是 Centos 7 缺省的日志记录工具，它记录了所有系统、内核、Service Unit 的日志。

相比 systemd，journald 记录的日志有如下优势：

1. 可以记录到内存或文件系统；(默认记录到内存，对应的位置为 /run/log/jounal)
2. 可以限制占用的磁盘空间、保证磁盘剩余空间；
3. 可以限制日志文件大小、保存的时间；

journald 默认将日志转发给 rsyslog，这会导致日志写了多份，/var/log/messages 中包含了太多无关日志，不方便后续查看，同时也影响系统性能。

mkdir /var/log/journal # 持久化保存日志的目录

mkdir /etc/systemd/journald.conf.d

cat > /etc/systemd/journald.conf.d/99-prophet.conf <<EOF

[Journal]

# 持久化保存到磁盘

Storage=persistent

# 压缩历史日志

Compress=yes

SyncIntervalSec=5m

RateLimitInterval=30s

RateLimitBurst=1000

# 最大占用空间 10G

SystemMaxUse=10G

# 单日志文件最大 200M

SystemMaxFileSize=200M

# 日志保存时间 2 周

MaxRetentionSec=2week

# 不将日志转发到 syslog

ForwardToSyslog=no

EOF

systemctl restart systemd-journald

#### 创建相关目录

创建目录：

mkdir -p /opt/k8s/{bin,work} /etc/{kubernetes,etcd}/cert

#### 升级内核

rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-3.el7.elrepo.noarch.rpm

# 安装完成后检查 /boot/grub2/grub.cfg 中对应内核 menuentry 中是否包含 initrd16 配置，如果没有，再安装一次！

yum --enablerepo=elrepo-kernel install -y kernel-lt

# 设置开机从新内核启动

grub2-set-default 0

安装内核源文件（可选，在升级完内核并重启机器后执行）:

# yum erase kernel-headers

yum --enablerepo=elrepo-kernel install kernel-lt-devel-$(uname -r) kernel-lt-headers-$(uname -r)

#### 关闭 NUMA

cp /etc/default/grub{,.bak}

vim /etc/default/grub # 在 GRUB\_CMDLINE\_LINUX 一行添加 `numa=off` 参数，如下所示：

diff /etc/default/grub.bak /etc/default/grub

6c6

< GRUB\_CMDLINE\_LINUX="crashkernel=auto rd.lvm.lv=centos/root rhgb quiet"

---

> GRUB\_CMDLINE\_LINUX="crashkernel=auto rd.lvm.lv=centos/root rhgb quiet numa=off"

重新生成 grub2 配置文件：

cp /boot/grub2/grub.cfg{,.bak}

grub2-mkconfig -o /boot/grub2/grub.cfg

#### 检查系统内核和模块是否适合运行 docker (仅适用于 linux 系统)

curl https://raw.githubusercontent.com/docker/docker/master/contrib/check-config.sh > check-config.sh

bash ./check-config.sh

#### 分发集群环境变量定义脚本（扩容时不需要执行该步骤）

后续的部署步骤将使用 environment.sh（文件在gitlab附件中） 文件中定义的全局环境变量，请根据自己的机器、网络情况修改：

然后，把全局变量定义脚本拷贝到所有节点的 /opt/k8s/bin 目录：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp /opt/k8s/bin/environment.sh root@${node\_ip}:/opt/k8s/bin/

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

#### 参考

系统内核相关参数参考：<https://docs.openshift.com/enterprise/3.2/admin_guide/overcommit.html>

### 创建 CA 证书和秘钥

为确保安全，kubernetes 系统各组件需要使用 x509 证书对通信进行加密和认证。

CA (Certificate Authority) 是自签名的根证书，用来签名后续创建的其它证书。

本文档使用 CloudFlare 的 PKI 工具集 [cfssl](https://github.com/cloudflare/cfssl) 创建所有证书。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 安装 cfssl 工具

sudo mkdir -p /opt/k8s/work/cert && cd /opt/k8s/work

wget https://pkg.cfssl.org/R1.2/cfssl\_linux-amd64

mv cfssl\_linux-amd64 /opt/k8s/bin/cfssl

wget https://pkg.cfssl.org/R1.2/cfssljson\_linux-amd64

mv cfssljson\_linux-amd64 /opt/k8s/bin/cfssljson

wget https://pkg.cfssl.org/R1.2/cfssl-certinfo\_linux-amd64

mv cfssl-certinfo\_linux-amd64 /opt/k8s/bin/cfssl-certinfo

chmod +x /opt/k8s/bin/\*

export PATH=/opt/k8s/bin:$PATH

#### 创建根证书 (CA)

CA 证书是集群所有节点共享的，**只需要创建一个 CA 证书**，后续创建的所有证书都由它签名。

#### 创建配置文件

CA 配置文件用于配置根证书的使用场景 (profile) 和具体参数 (usage，过期时间、服务端认证、客户端认证、加密等)，后续在签名其它证书时需要指定特定场景。

cd /opt/k8s/work/cert

cat > ca-config.json <<EOF

{

"signing": {

"default": {

"expiry": "87600h"

},

"profiles": {

"kubernetes": {

"usages": [

"signing",

"key encipherment",

"server auth",

"client auth"

],

"expiry": "87600h"

}

}

}

}

EOF

* signing：表示该证书可用于签名其它证书，生成的 ca.pem 证书中 CA=TRUE；
* server auth：表示 client 可以用该该证书对 server 提供的证书进行验证；
* client auth：表示 server 可以用该该证书对 client 提供的证书进行验证；

#### 创建证书签名请求文件

cd /opt/k8s/work/cert

cat > ca-csr.json <<EOF

{

"CN": "kubernetes",

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* CN：Common Name，kube-apiserver 从证书中提取该字段作为请求的用户名 (User Name)，浏览器使用该字段验证网站是否合法；
* O：Organization，kube-apiserver 从证书中提取该字段作为请求用户所属的组 (Group)；
* kube-apiserver 将提取的 User、Group 作为 RBAC 授权的用户标识；

#### 生成 CA 证书和私钥

cd /opt/k8s/work/cert

cfssl gencert -initca ca-csr.json | cfssljson -bare ca

ls ca\*

#### 分发证书文件

将生成的 CA 证书、秘钥文件、配置文件拷贝到所有节点的 /etc/kubernetes/cert 目录下：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh # 导入 NODE\_IPS 环境变量

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p /etc/kubernetes/cert"

scp ca\*.pem ca-config.json root@${node\_ip}:/etc/kubernetes/cert

done

#### 参考

各种 CA 证书类型：

<https://github.com/kubernetes-incubator/apiserver-builder/blob/master/docs/concepts/auth.md>

### 部署 kubectl 命令行工具

kubectl 是 kubernetes 集群的命令行管理工具，本小节介绍安装和配置它的步骤。

kubectl 默认从 ~/.kube/config 文件读取 kube-apiserver 地址、证书、用户名等信息，如果没有配置，执行 kubectl 命令时可能会出错：

kubectl get pods

The connection to the server localhost:8080 was refused - did you specify the right host or port?

注意：

1. 如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。
2. 本文档只需要部署一次，生成的 kubeconfig 文件是通用的，可以拷贝到需要执行 kubeclt 命令的机器上。

#### 下载和分发 kubectl 命令行工具

下载和解压：

cd /opt/k8s/work

wget https://dl.k8s.io/v1.12.3/kubernetes-client-linux-amd64.tar.gz

tar -xzvf kubernetes-client-linux-amd64.tar.gz

分发到所有使用 kubectl 的节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kubernetes/client/bin/kubectl root@${node\_ip}:/opt/k8s/bin/

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

#### 创建 admin 证书和私钥

kubectl 与 apiserver https 安全端口通信，apiserver 对提供的证书进行认证和授权。

kubectl 作为集群的管理工具，需要被授予最高权限。这里创建具有最高权限的 admin 证书。

创建证书签名请求：

cd /opt/k8s/work/cert

cat > admin-csr.json <<EOF

{

"CN": "admin",

"hosts": [],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "system:masters",

"OU": "study163"

}

]

}

EOF

* 为 system:masters，kube-apiserver 收到该证书后将请求的 Group 设置为 system:masters；
* 预定义的 ClusterRoleBinding cluster-admin 将 Group system:masters 与 Role cluster-admin 绑定，该 Role 授予所有 API的权限；
* 该证书只会被 kubectl 当做 client 证书使用，所以 hosts 字段为空；

生成证书和私钥：

cd /opt/k8s/work/cert

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes admin-csr.json | cfssljson -bare admin

ls admin\*

#### 创建 kubeconfig 文件

kubeconfig 为 kubectl 的配置文件，包含访问 apiserver 的所有信息，如 apiserver 地址、CA 证书和自身使用的证书；

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

# 设置集群参数

kubectl config set-cluster kubernetes \

--certificate-authority=/opt/k8s/work/cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=kubectl.kubeconfig

# 设置客户端认证参数

kubectl config set-credentials admin \

--client-certificate=/opt/k8s/work/cert/admin.pem \

--client-key=/opt/k8s/work/cert/admin-key.pem \

--embed-certs=true \

--kubeconfig=kubectl.kubeconfig

# 设置上下文参数

kubectl config set-context kubernetes \

--cluster=kubernetes \

--user=admin \

--kubeconfig=kubectl.kubeconfig

# 设置默认上下文

kubectl config use-context kubernetes --kubeconfig=kubectl.kubeconfig

* --certificate-authority：验证 kube-apiserver 证书的根证书；
* --client-certificate、--client-key：刚生成的 admin 证书和私钥，连接 kube-apiserver 时使用；
* --embed-certs=true：将 ca.pem 和 admin.pem 证书内容嵌入到生成的 kubectl.kubeconfig 文件中(不加时，写入的是证书文件路径)；

#### 分发 kubeconfig 文件

分发到所有使用 kubectl 命令的节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ~/.kube"

scp kubectl.kubeconfig root@${node\_ip}:~/.kube/config

done

* 保存到用户的 ~/.kube/config 文件；

### 部署 etcd 集群

etcd 是基于 Raft 的分布式 key-value 存储系统，由 CoreOS 开发，常用于服务发现、共享配置以及并发控制（如 leader 选举、分布式锁等）。kubernetes 使用 etcd 存储所有运行数据。

本文档介绍部署一个三节点高可用 etcd 集群的步骤：

* 下载和分发 etcd 二进制文件；
* 创建 etcd 集群各节点的 x509 证书，用于加密客户端(如 etcdctl) 与 etcd 集群、etcd 集群之间的数据流；
* 创建 etcd 的 systemd unit 文件，配置服务参数；
* 检查集群工作状态；

etcd 集群各节点的名称和 IP 如下：

* master：~~192.168.100.246~~
* node1：~~192.168.100.247~~
* node2：~~192.168.100.248~~

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 下载和分发 etcd 二进制文件

到 <https://github.com/coreos/etcd/releases> 页面下载最新版本的发布包：

cd /opt/k8s/work

wget https://github.com/coreos/etcd/releases/download/v3.3.10/etcd-v3.3.10-linux-amd64.tar.gz

tar -xvf etcd-v3.3.10-linux-amd64.tar.gz

分发二进制文件到集群所有节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp etcd-v3.3.10-linux-amd64/etcd\* root@${node\_ip}:/opt/k8s/bin

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

#### 创建 etcd 证书和私钥

创建证书签名请求：

cd /opt/k8s/work/cert

cat > etcd-csr.json <<EOF

{

"CN": "etcd",

"hosts": [

"127.0.0.1",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~"

],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* hosts 字段指定授权使用该证书的 etcd 节点 IP 或域名列表，这里将 etcd 集群的三个节点 IP 都列在其中；

生成证书和私钥：

cd /opt/k8s/work/cert

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes etcd-csr.json | cfssljson -bare etcd

ls etcd\*pem

分发生成的证书和私钥到各 etcd 节点：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p /etc/etcd/cert"

scp etcd\*.pem root@${node\_ip}:/etc/etcd/cert/

done

#### 创建 etcd 的 systemd unit 模板文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat > etcd.service.template <<EOF

[Unit]

Description=Etcd Server

After=network.target

After=network-online.target

Wants=network-online.target

Documentation=https://github.com/coreos

[Service]

Type=notify

WorkingDirectory=${ETCD\_DATA\_DIR}

ExecStart=/opt/k8s/bin/etcd \\

--data-dir=${ETCD\_DATA\_DIR} \\

--wal-dir=${ETCD\_WAL\_DIR} \\

--name=##NODE\_NAME## \\

--cert-file=/etc/etcd/cert/etcd.pem \\

--key-file=/etc/etcd/cert/etcd-key.pem \\

--trusted-ca-file=/etc/kubernetes/cert/ca.pem \\

--peer-cert-file=/etc/etcd/cert/etcd.pem \\

--peer-key-file=/etc/etcd/cert/etcd-key.pem \\

--peer-trusted-ca-file=/etc/kubernetes/cert/ca.pem \\

--peer-client-cert-auth \\

--client-cert-auth \\

--listen-peer-urls=https://##NODE\_IP##:2380 \\

--initial-advertise-peer-urls=https://##NODE\_IP##:2380 \\

--listen-client-urls=https://##NODE\_IP##:2379,http://127.0.0.1:2379 \\

--advertise-client-urls=https://##NODE\_IP##:2379 \\

--initial-cluster-token=etcd-cluster-0 \\

--initial-cluster=${ETCD\_NODES} \\

--initial-cluster-state=new \\

--auto-compaction-mode=periodic \\

--auto-compaction-retention=1 \\

--max-request-bytes=33554432 \\

--quota-backend-bytes=6442450944 \\

--heartbeat-interval=250 \\

--election-timeout=2000

Restart=on-failure

RestartSec=5

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

EOF

* WorkingDirectory、--data-dir：指定工作目录和数据目录为 ${ETCD\_DATA\_DIR}，需在启动服务前创建这个目录；
* --wal-dir：指定 wal 目录，为了提高性能，一般使用 SSD 或者和 --data-dir 不同的磁盘；
* --name：指定节点名称，当 --initial-cluster-state 值为 new 时，--name 的参数值必须位于 --initial-cluster 列表中；
* --cert-file、--key-file：etcd server 与 client 通信时使用的证书和私钥；
* --trusted-ca-file：签名 client 证书的 CA 证书，用于验证 client 证书；
* --peer-cert-file、--peer-key-file：etcd 与 peer 通信使用的证书和私钥；
* --peer-trusted-ca-file：签名 peer 证书的 CA 证书，用于验证 peer 证书；

#### 为各节点创建和分发 etcd systemd unit 文件

替换模板文件中的变量，为各节点创建 systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for (( i=0; i < 3; i++ ))

do

sed -e "s/##NODE\_NAME##/${NODE\_NAMES[i]}/" -e "s/##NODE\_IP##/${NODE\_IPS[i]}/" etcd.service.template > etcd-${NODE\_IPS[i]}.service

done

ls \*.service

* NODE\_NAMES 和 NODE\_IPS 为相同长度的 bash 数组，分别为节点名称和对应的 IP；

分发生成的 systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp etcd-${node\_ip}.service root@${node\_ip}:/usr/lib/systemd/system/etcd.service

done

* 文件重命名为 etcd.service;

#### 启动 etcd 服务

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${ETCD\_DATA\_DIR} ${ETCD\_WAL\_DIR}"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable etcd && systemctl restart etcd " &

done

* 必须创建 etcd 数据目录和工作目录;
* etcd 进程首次启动时会等待其它节点的 etcd 加入集群，命令 systemctl start etcd 会卡住一段时间，为正常现象。

#### 检查启动结果

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status etcd|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u etcd

#### 验证服务状态

部署完 etcd 集群后，在任一 etc 节点上执行如下命令：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ETCDCTL\_API=3 /opt/k8s/bin/etcdctl \

--endpoints=https://${node\_ip}:2379 \

--cacert=/opt/k8s/work/cert/ca.pem \

--cert=/etc/etcd/cert/etcd.pem \

--key=/etc/etcd/cert/etcd-key.pem endpoint health

done

预期输出：

>>> ~~192.168.100.246~~

https://~~192.168.100.246~~:2379 is healthy: successfully committed proposal: took = 3.5037ms

>>> ~~192.168.100.247~~

https://~~192.168.100.247~~:2379 is healthy: successfully committed proposal: took = 17.918242ms

>>> ~~192.168.100.248~~

https://~~192.168.100.248~~:2379 is healthy: successfully committed proposal: took = 10.445815ms

输出均为 healthy 时表示集群服务正常。

#### 查看当前的 leader

source /opt/k8s/bin/environment.sh

ETCDCTL\_API=3 /opt/k8s/bin/etcdctl \

-w table --cacert=/opt/k8s/work/cert/ca.pem \

--cert=/etc/etcd/cert/etcd.pem \

--key=/etc/etcd/cert/etcd-key.pem \

--endpoints=${ETCD\_ENDPOINTS} endpoint status

输出：

+------------------------------+------------------+---------+---------+-----------+-----------+------------+

| ENDPOINT | ID | VERSION | DB SIZE | IS LEADER | RAFT TERM | RAFT INDEX |

+------------------------------+------------------+---------+---------+-----------+-----------+------------+

| https://~~192.168.100.246~~:2379 | 52bfdc32535efaaa | 3.3.10 | 20 kB | true | 9 | 11 |

| https://~~192.168.100.247~~:2379 | 13db042075766644 | 3.3.10 | 20 kB | false | 9 | 11 |

| https://~~192.168.100.248~~:2379 | 770b9415f8cc927b | 3.3.10 | 20 kB | false | 9 | 11 |

+------------------------------+------------------+---------+---------+-----------+-----------+------------+

* 可见，当前的 leader 为 ~~192.168.100.246~~。

### 部署 flannel 网络

kubernetes 要求集群内各节点(包括 master 节点)能通过 Pod 网段互联互通。flannel 使用 vxlan 技术为各节点创建一个可以互通的 Pod 网络，使用的端口为 **UDP 8472，需要开放该端口**（如公有云 AWS 等）。

flanneld 第一次启动时，从 etcd 获取配置的 Pod 网段信息，为本节点分配一个未使用的地址段，然后创建 flannedl.1 网络接口（也可能是其它名称，如 flannel1 等）。

flannel 将分配给自己的 Pod 网段信息写入 /run/flannel/docker 文件，docker 后续使用这个文件中的环境变量设置 docker0 网桥，从而从这个地址段为本节点的所有 Pod 容器分配 IP。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 下载和分发 flanneld 二进制文件

到 https://github.com/coreos/flannel/releases 页面下载最新版本的发布包：

cd /opt/k8s/work

mkdir flannel

wget https://github.com/coreos/flannel/releases/download/v0.10.0/flannel-v0.10.0-linux-amd64.tar.gz

tar -xzvf flannel-v0.10.0-linux-amd64.tar.gz -C flannel

分发 flanneld 二进制文件到集群所有节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp flannel/{flanneld,mk-docker-opts.sh} root@${node\_ip}:/opt/k8s/bin/

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

#### 创建 flannel 证书和私钥

flannel 从 etcd 集群存取网段分配信息，而 etcd 集群启用了双向 x509 证书认证，所以需要为 flanneld 生成证书和私钥。

创建证书签名请求：

cd /opt/k8s/work/cert

cat > flanneld-csr.json <<EOF

{

"CN": "flanneld",

"hosts": [],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* 该证书只会被 kubectl 当做 client 证书使用，所以 hosts 字段为空；

生成证书和私钥：

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes flanneld-csr.json | cfssljson -bare flanneld

ls flannel\*pem

将生成的证书和私钥分发到所有节点（master 和 worker）：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p /etc/flanneld/cert"

scp flanneld\*.pem root@${node\_ip}:/etc/flanneld/cert

done

#### 向 etcd 写入集群 Pod 网段信息

**注意：本步骤只需执行一次。**

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

etcdctl \

--endpoints=${ETCD\_ENDPOINTS} \

--ca-file=/opt/k8s/work/cert/ca.pem \

--cert-file=/opt/k8s/work/cert/flanneld.pem \

--key-file=/opt/k8s/work/cert/flanneld-key.pem \

set ${FLANNEL\_ETCD\_PREFIX}/config '{"Network":"'${CLUSTER\_CIDR}'", "SubnetLen": 21, "Backend": {"Type": "vxlan"}}'

* flanneld 当前版本 (v0.10.0) 不支持 etcd v3，故使用 etcd v2 API 写入配置 key 和网段数据；
* 写入的 Pod 网段 ${CLUSTER\_CIDR} 地址段如 /16 必须小于 SubnetLen，必须与 kube-controller-manager 的 --cluster-cidr 参数值一致；

#### 创建 flanneld 的 systemd unit 文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat > flanneld.service << EOF

[Unit]

Description=Flanneld overlay address etcd agent

After=network.target

After=network-online.target

Wants=network-online.target

After=etcd.service

Before=docker.service

[Service]

Type=notify

ExecStart=/opt/k8s/bin/flanneld \\

-etcd-cafile=/etc/kubernetes/cert/ca.pem \\

-etcd-certfile=/etc/flanneld/cert/flanneld.pem \\

-etcd-keyfile=/etc/flanneld/cert/flanneld-key.pem \\

-etcd-endpoints=${ETCD\_ENDPOINTS} \\

-etcd-prefix=${FLANNEL\_ETCD\_PREFIX} \\

-iface=${IFACE} \\

-ip-masq

ExecStartPost=/opt/k8s/bin/mk-docker-opts.sh -k DOCKER\_NETWORK\_OPTIONS -d /run/flannel/docker

Restart=always

RestartSec=5

StartLimitInterval=0

[Install]

WantedBy=multi-user.target

RequiredBy=docker.service

EOF

* mk-docker-opts.sh 脚本将分配给 flanneld 的 Pod 子网网段信息写入 /run/flannel/docker 文件，后续 docker 启动时使用这个文件中的环境变量配置 docker0 网桥；
* flanneld 使用系统缺省路由所在的接口与其它节点通信，对于有多个网络接口（如内网和公网）的节点，可以用 -iface 参数指定通信接口;
* flanneld 运行时需要 root 权限；
* -ip-masq: flanneld 为访问 Pod 网络外的流量设置 SNAT 规则，同时将传递给 Docker 的变量 --ip-masq（/run/flannel/docker 文件中）设置为 false，这样 Docker 将不再创建 SNAT 规则； Docker 的 --ip-masq 为 true 时，创建的 SNAT 规则比较“暴力”：将所有本节点 Pod 发起的、访问非 docker0 接口的请求做 SNAT，这样访问其他节点 Pod 的请求来源 IP 会被设置为 flannel.1 接口的 IP，导致目的 Pod 看不到真实的来源 Pod IP。 flanneld 创建的 SNAT 规则比较温和，只对访问非 Pod 网段的请求做 SNAT。

#### 分发 flanneld systemd unit 文件到所有节点

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp flanneld.service root@${node\_ip}:/usr/lib/systemd/system/

done

#### 启动 flanneld 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable flanneld && systemctl restart flanneld"

done

#### 检查启动结果

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status flanneld|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u flanneld

#### 检查分配给各 flanneld 的 Pod 网段信息

查看集群 Pod 网段(/16)：

source /opt/k8s/bin/environment.sh

etcdctl \

--endpoints=${ETCD\_ENDPOINTS} \

--ca-file=/etc/kubernetes/cert/ca.pem \

--cert-file=/etc/flanneld/cert/flanneld.pem \

--key-file=/etc/flanneld/cert/flanneld-key.pem \

get ${FLANNEL\_ETCD\_PREFIX}/config

输出：

{"Network":"172.30.0.0/16", "SubnetLen": 21, "Backend": {"Type": "vxlan"}}

查看已分配的 Pod 子网段列表(/24):

source /opt/k8s/bin/environment.sh

etcdctl \

--endpoints=${ETCD\_ENDPOINTS} \

--ca-file=/etc/kubernetes/cert/ca.pem \

--cert-file=/etc/flanneld/cert/flanneld.pem \

--key-file=/etc/flanneld/cert/flanneld-key.pem \

ls ${FLANNEL\_ETCD\_PREFIX}/subnets

输出（结果是部署情况而定，网段可能与下面不一样）：

/kubernetes/network/subnets/172.30.240.0-21

/kubernetes/network/subnets/172.30.120.0-21

/kubernetes/network/subnets/172.30.192.0-21

查看某一 Pod 网段对应的节点 IP 和 flannel 接口地址:

source /opt/k8s/bin/environment.sh

etcdctl \

--endpoints=${ETCD\_ENDPOINTS} \

--ca-file=/etc/kubernetes/cert/ca.pem \

--cert-file=/etc/flanneld/cert/flanneld.pem \

--key-file=/etc/flanneld/cert/flanneld-key.pem \

get ${FLANNEL\_ETCD\_PREFIX}/subnets/~~172.30.240.0~~-21

输出（结果是部署情况而定，网段可能与下面不一样）：

{"PublicIP":"~~192.168.100.246~~","BackendType":"vxlan","BackendData":{"VtepMAC":"fa:32:14:34:86:74"}}

* 172.30.240.0/21 被分配给节点 node2（192.168.100.252）；
* VtepMAC 为 node2 节点的 flannel.1 网卡 MAC 地址；

#### 检查节点 flannel 网络信息

ip addr show

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid\_lft forever preferred\_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP group default qlen 1000

link/ether 00:22:0d:33:89:75 brd ff:ff:ff:ff:ff:ff

inet 172.27.137.240/20 brd 172.27.143.255 scope global dynamic eth0

valid\_lft 100647283sec preferred\_lft 100647283sec

3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc noqueue state UNKNOWN group default

link/ether ce:9c:a9:08:50:03 brd ff:ff:ff:ff:ff:ff

inet 172.30.80.0/32 scope global flannel.1

valid\_lft forever preferred\_lft forever

* flannel.1 网卡的地址为分配的 Pod 子网段的第一个 IP（.0），且是 /32 的地址；

ip route show |grep flannel.1

172.30.32.0/21 via 172.30.32.0 dev flannel.1 onlink

172.30.184.0/21 via 172.30.184.0 dev flannel.1 onlink

* 到其它节点 Pod 网段请求都被转发到 flannel.1 网卡；
* flanneld 根据 etcd 中子网段的信息，如 ${FLANNEL\_ETCD\_PREFIX}/subnets/172.30.80.0-21 ，来决定进请求发送给哪个节点的互联 IP；

#### 验证各节点能通过 Pod 网段互通

在各节点上部署 flannel 后，检查是否创建了 flannel 接口(名称可能为 flannel0、flannel.0、flannel.1 等)：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "/usr/sbin/ip addr show flannel.1|grep -w inet"

done

输出：

>>> ~~192.168.100.246~~

inet 172.30.240.0/32 scope global flannel.1

>>> ~~192.168.100.247~~

inet 172.30.120.0/32 scope global flannel.1

>>> ~~192.168.100.248~~

inet 172.30.192.0/32 scope global flannel.1

在各节点上 ping 所有 flannel 接口 IP，确保能通：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "ping -c 1 ~~172.30.240.0~~"

ssh ${node\_ip} "ping -c 1 ~~172.30.120.0~~"

ssh ${node\_ip} "ping -c 1 ~~172.30.192.0~~"

done

### kube-apiserver 高可用之 nginx 代理

kuebectl 的配置文件中静态指定了某个 kube-apiserver IP，如果该 apiserver 实例挂掉，可能引起服务异常。

本章节讲解使用 nginx 4 层透明代理功能实现 K8S 节点( master 节点和 worker 节点)高可用访问 kube-apiserver 的方案。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 基于 nginx 代理的 kube-apiserver 高可用方案

* 控制节点的 kube-controller-manager、kube-scheduler 是多实例部署，所以只要有一个实例正常，就可以保证高可用；
* 集群内的 Pod 使用域名 kubernetes 访问 kube-apiserver， kube-dns 会自动解析出多个 kube-apiserver 节点的 IP，所以也是高可用的；
* kubelet、kube-proxy、controller-manager、scheduler 通过本地的 kube-nginx（监听 127.0.0.1）访问 kube-apiserver，从而实现 kube-apiserver 的高可用。
* kube-nginx 会对所有 kube-apiserver 实例做健康检查和负载均衡；

#### 下载和编译 nginx

下载源码：

cd /opt/k8s/work

wget http://nginx.org/download/nginx-1.15.3.tar.gz

tar -xzvf nginx-1.15.3.tar.gz

安装nginx必须的依赖包

yum -y install gcc openssl-devel pcre-devel zlib-devel

配置编译参数：

cd /opt/k8s/work/nginx-1.15.3

mkdir /opt/k8s/kube-nginx

./configure --prefix=/opt/k8s/kube-nginx --with-stream --without-http --without-http\_uwsgi\_module --without-http\_scgi\_module --without-http\_fastcgi\_module

* --with-stream：开启 4 层透明转发(TCP Proxy)功能；
* --without-xxx：关闭所有其他功能，这样生成的动态链接二进制程序依赖最小；

输出：

Configuration summary

+ PCRE library is not used

+ OpenSSL library is not used

+ zlib library is not used

nginx path prefix: "/opt/k8s/kube-nginx"

nginx binary file: "/opt/k8s/kube-nginx/sbin/nginx"

nginx modules path: "/opt/k8s/kube-nginx/modules"

nginx configuration prefix: "/opt/k8s/kube-nginx/conf"

nginx configuration file: "/opt/k8s/kube-nginx/conf/nginx.conf"

nginx pid file: "/opt/k8s/kube-nginx/logs/nginx.pid"

nginx error log file: "/opt/k8s/kube-nginx/logs/error.log"

nginx http access log file: "/opt/k8s/kube-nginx/logs/access.log"

nginx http client request body temporary files: "client\_body\_temp"

nginx http proxy temporary files: "proxy\_temp"

编译和安装：

cd /opt/k8s/work/nginx-1.15.3

make && make install

#### 验证编译的 nginx

cd /opt/k8s/kube-nginx

./sbin/nginx -v

输出：

nginx version: nginx/1.15.3

查看 nginx 动态链接的库：

ldd ./sbin/nginx

linux-vdso.so.1 => (0x00007ffe11794000)

libdl.so.2 => /lib64/libdl.so.2 (0x00007f29eb5d5000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f29eb3b9000)

libc.so.6 => /lib64/libc.so.6 (0x00007f29eafeb000)

/lib64/ld-linux-x86-64.so.2 (0x00007f29eb7e0000)

* 由于只开启了 4 层透明转发功能，所以除了依赖 libc 等操作系统核心 lib 库外，没有对其它 lib 的依赖(如 libz、libssl 等)，这样可以方便部署到各版本操作系统中；

#### 安装和部署 nginx

配置 nginx，开启 4 层透明转发功能：

cat > /opt/k8s/kube-nginx/conf/nginx.conf <<EOF

worker\_processes 1;

events {

worker\_connections 1024;

}

stream {

upstream backend {

hash $remote\_addr consistent;

server ~~192.168.100.246~~:6443 max\_fails=3 fail\_timeout=30s;

server ~~192.168.100.247~~:6443 max\_fails=3 fail\_timeout=30s;

server ~~192.168.100.248~~:6443 max\_fails=3 fail\_timeout=30s;

}

server {

listen 8443;

proxy\_connect\_timeout 1s;

proxy\_pass backend;

}

}

EOF

* 需要根据集群 kube-apiserver 的实际情况，替换 backend 中 server 列表；

#### 配置 systemd unit 文件，启动服务

配置 kube-nginx systemd unit 文件：

cd /opt/k8s/work

cat > kube-nginx.service <<EOF

[Unit]

Description=kube-apiserver nginx proxy

After=network.target

After=network-online.target

Wants=network-online.target

[Service]

Type=forking

ExecStartPre=/opt/k8s/kube-nginx/sbin/nginx -c /opt/k8s/kube-nginx/conf/nginx.conf -p /opt/k8s/kube-nginx -t

ExecStart=/opt/k8s/kube-nginx/sbin/nginx -c /opt/k8s/kube-nginx/conf/nginx.conf -p /opt/k8s/kube-nginx

ExecReload=/opt/k8s/kube-nginx/sbin/nginx -c /opt/k8s/kube-nginx/conf/nginx.conf -p /opt/k8s/kube-nginx -s reload

PrivateTmp=true

Restart=always

RestartSec=5

StartLimitInterval=0

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

EOF

启动 kube-nginx 服务：

cp kube-nginx.service /usr/lib/systemd/system/

systemctl daemon-reload && systemctl enable kube-nginx && systemctl restart kube-nginx

#### 检查 kube-nginx 服务运行状态

systemctl status kube-nginx |grep 'Active:'

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u kube-nginx

### 部署 master 节点

kubernetes master 节点运行如下组件：

* kube-apiserver
* kube-scheduler
* kube-controller-manager
* kube-nginx

1. kube-scheduler 和 kube-controller-manager 会自动选举产生一个 leader 实例，其它实例处于阻塞模式，当 leader 挂了后，重新选举产生新的 leader，从而保证服务可用性；
2. kube-apiserver 是无状态的，需要通过 kube-nginx 进行代理访问，从而保证服务可用性；

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 安装和配置 kube-nginx

参考 [apiserver高可用之nginx代理](#_apiserver_高可用之_nginx)

#### 下载最新版本二进制文件

从 [CHANGELOG页面](https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG.md) 下载 server tarball 文件。

cd /opt/k8s/work

wget https://dl.k8s.io/v1.12.3/kubernetes-server-linux-amd64.tar.gz

tar -xzvf kubernetes-server-linux-amd64.tar.gz

cd kubernetes

tar -xzvf kubernetes-src.tar.gz

将二进制文件拷贝到所有 master 节点：

cd /opt/k8s/work/kubernetes

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp server/bin/\* root@${node\_ip}:/opt/k8s/bin/

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

#### apiserver 集群

本章节讲解部署一个三实例 kube-apiserver 集群的步骤，它们通过 kube-nginx 进行代理访问，从而保证服务可用性。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

##### 准备工作

注意：下载最新版本的二进制文件、安装和配置 flanneld 参考：[部署master节点](#_部署master节点)

##### 创建 kubernetes 证书和私钥

创建证书签名请求：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh

cat > kubernetes-csr.json <<EOF

{

"CN": "kubernetes",

"hosts": [

"127.0.0.1",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~",

"${CLUSTER\_KUBERNETES\_SVC\_IP}",

"kubernetes",

"kubernetes.default",

"kubernetes.default.svc",

"kubernetes.default.svc.cluster",

"kubernetes.default.svc.cluster.local"

],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* hosts 字段指定授权使用该证书的 **IP 或域名列表**，这里列出了 VIP 、apiserver 节点 IP、kubernetes 服务 IP 和域名；
* 域名最后字符不能是 .(如不能为 kubernetes.default.svc.cluster.local.)，否则解析时失败，提示： x509: cannot parse dnsName "kubernetes.default.svc.cluster.local."；
* 如果使用非 cluster.local 域名，如 opsnull.com，则需要修改域名列表中的最后两个域名为：kubernetes.default.svc.opsnull、kubernetes.default.svc.opsnull.com
* kubernetes 服务 IP 是 apiserver 自动创建的，一般是 --service-cluster-ip-range 参数指定的网段的第一个IP，后续可以通过如下命令获取：

kubectl get svc kubernetes

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes 10.254.0.1 <none> 443/TCP 1d

生成证书和私钥：

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes

ls kubernetes\*pem

将生成的证书和私钥文件拷贝到 master 节点：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p /etc/kubernetes/cert"

scp kubernetes\*.pem root@${node\_ip}:/etc/kubernetes/cert/

done

##### 创建加密配置文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat > encryption-config.yaml <<EOF

kind: EncryptionConfig

apiVersion: v1

resources:

- resources:

- secrets

providers:

- aescbc:

keys:

- name: key1

secret: ${ENCRYPTION\_KEY}

- identity: {}

EOF

将加密配置文件拷贝到 master 节点的 /etc/kubernetes 目录下：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp encryption-config.yaml root@${node\_ip}:/etc/kubernetes/

done

##### 创建 kube-apiserver systemd unit 模板文件

**特别提醒：有同学没有按照老师的k8s版本安装，自己升级了版本，有些地方k8s升级后官方默认功能做了调整，比如kube-apiserver中的1.14版本 不支持Initializers插件。**

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat > kube-apiserver.service.template <<EOF

[Unit]

Description=Kubernetes API Server

Documentation=https://github.com/GoogleCloudPlatform/kubernetes

After=network.target

[Service]

WorkingDirectory=${K8S\_DIR}/kube-apiserver

ExecStart=/opt/k8s/bin/kube-apiserver \\

--enable-admission-plugins=Initializers,NamespaceLifecycle,NodeRestriction,LimitRanger,ServiceAccount,DefaultStorageClass,ResourceQuota \\

--anonymous-auth=false \\

--experimental-encryption-provider-config=/etc/kubernetes/encryption-config.yaml \\

--advertise-address=##NODE\_IP## \\

--bind-address=##NODE\_IP## \\

--insecure-port=0 \\

--authorization-mode=Node,RBAC \\

--runtime-config=api/all \\

--enable-bootstrap-token-auth \\

--service-cluster-ip-range=${SERVICE\_CIDR} \\

--service-node-port-range=${NODE\_PORT\_RANGE} \\

--tls-cert-file=/etc/kubernetes/cert/kubernetes.pem \\

--tls-private-key-file=/etc/kubernetes/cert/kubernetes-key.pem \\

--client-ca-file=/etc/kubernetes/cert/ca.pem \\

--kubelet-certificate-authority=/etc/kubernetes/cert/ca.pem \\

--kubelet-client-certificate=/etc/kubernetes/cert/kubernetes.pem \\

--kubelet-client-key=/etc/kubernetes/cert/kubernetes-key.pem \\

--kubelet-https=true \\

--service-account-key-file=/etc/kubernetes/cert/ca.pem \\

--etcd-cafile=/etc/kubernetes/cert/ca.pem \\

--etcd-certfile=/etc/kubernetes/cert/kubernetes.pem \\

--etcd-keyfile=/etc/kubernetes/cert/kubernetes-key.pem \\

--etcd-servers=${ETCD\_ENDPOINTS} \\

--enable-swagger-ui=true \\

--allow-privileged=true \\

--max-mutating-requests-inflight=2000 \\

--max-requests-inflight=4000 \\

--apiserver-count=3 \\

--audit-log-maxage=30 \\

--audit-log-maxbackup=3 \\

--audit-log-maxsize=100 \\

--audit-log-path=${K8S\_DIR}/kube-apiserver/audit.log \\

--event-ttl=168h \\

--logtostderr=true \\

--v=2

Restart=on-failure

RestartSec=5

Type=notify

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

EOF

* --experimental-encryption-provider-config：启用加密特性；
* --authorization-mode=Node,RBAC： 开启 Node 和 RBAC 授权模式，拒绝未授权的请求；
* --enable-admission-plugins：启用 ServiceAccount 和 NodeRestriction；
* --service-account-key-file：签名 ServiceAccount Token 的公钥文件，kube-controller-manager 的 --service-account-private-key-file 指定私钥文件，两者配对使用；
* --tls-\*-file：指定 apiserver 使用的证书、私钥和 CA 文件。--client-ca-file 用于验证 client (kue-controller-manager、kube-scheduler、kubelet、kube-proxy 等)请求所带的证书；
* --kubelet-client-certificate、--kubelet-client-key：如果指定，则使用 https 访问 kubelet APIs；需要为证书对应的用户(上面 kubernetes\*.pem 证书的用户为 kubernetes) 用户定义 RBAC 规则，否则访问 kubelet API 时提示未授权；
* --bind-address： 不能为 127.0.0.1，否则外界不能访问它的安全端口 6443；
* --insecure-port=0：关闭监听非安全端口(8080)；
* --service-cluster-ip-range： 指定 Service Cluster IP 地址段；
* --service-node-port-range： 指定 NodePort 的端口范围；
* --runtime-config=api/all=true： 启用所有版本的 APIs，如 autoscaling/v2alpha1；
* --enable-bootstrap-token-auth：启用 kubelet bootstrap 的 token 认证；
* --apiserver-count=3：指定 apiserver 的实例数量；

##### 为各节点创建和分发 kube-apiserver systemd unit 文件

替换模板文件中的变量，为各节点创建 systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for (( i=0; i < 3; i++ ))

do

sed -e "s/##NODE\_NAME##/${NODE\_NAMES[i]}/" -e "s/##NODE\_IP##/${NODE\_IPS[i]}/" kube-apiserver.service.template > kube-apiserver-${NODE\_IPS[i]}.service

done

ls kube-apiserver\*.service

* NODE\_NAMES 和 NODE\_IPS 为相同长度的 bash 数组，分别为节点名称和对应的 IP；

分发生成的 systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-apiserver-${node\_ip}.service root@${node\_ip}:/usr/lib/systemd/system/kube-apiserver.service

done

* 文件重命名为 kube-apiserver.service;

##### 启动 kube-apiserver 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${K8S\_DIR}/kube-apiserver"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable kube-apiserver && systemctl restart kube-apiserver"

done

* 必须创建工作目录；

##### 检查 kube-apiserver 运行状态

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status kube-apiserver |grep 'Active:'"

done

确保状态为 active (running)，否则到 master 节点查看日志，确认原因：

journalctl -u kube-apiserver

##### 打印 kube-apiserver 写入 etcd 的数据

source /opt/k8s/bin/environment.sh

ETCDCTL\_API=3 etcdctl \

--endpoints=${ETCD\_ENDPOINTS} \

--cacert=/opt/k8s/work/cert/ca.pem \

--cert=/opt/k8s/work/cert/etcd.pem \

--key=/opt/k8s/work/cert/etcd-key.pem \

get /registry/ --prefix --keys-only

##### 检查集群信息

kubectl cluster-info

Kubernetes master is running at https://127.0.0.1:8443

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

kubectl get all --all-namespaces

NAMESPACE NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

default service/kubernetes ClusterIP 10.254.0.1 <none> 443/TCP 4m27s

kubectl get componentstatuses

NAME STATUS MESSAGE ERROR

controller-manager Unhealthy Get http://127.0.0.1:10252/healthz: dial tcp 127.0.0.1:10252: connect: connection refused

scheduler Unhealthy Get http://127.0.0.1:10251/healthz: dial tcp 127.0.0.1:10251: connect: connection refused

etcd-2 Healthy {"health":"true"}

etcd-0 Healthy {"health":"true"}

etcd-1 Healthy {"health":"true"}

注意：

1. 如果执行 kubectl 命令式时输出如下错误信息，则说明使用的 ~/.kube/config 文件不对，请切换到正确的账户后再执行该命令：

The connection to the server localhost:8080 was refused - did you specify the right host or port?

1. 执行 kubectl get componentstatuses 命令时，apiserver 默认向 127.0.0.1 发送请求。当 controller-manager、scheduler 以集群模式运行时，有可能和 kube-apiserver **不在一台机器上**，这时 controller-manager 或 scheduler 的状态为 Unhealthy，但实际上它们工作**正常**。

##### 检查 kube-apiserver 监听的端口

sudo netstat -lnpt|grep kube-api

tcp 0 0 ~~192.168.100.246~~:6443 0.0.0.0:\* LISTEN 51903/kube-apiserve

* 6443: 接收 https 请求的安全端口，对所有请求做认证和授权；
* 由于关闭了非安全端口，故没有监听 8080；

##### 授予 kubernetes 证书访问 kubelet API 的权限

在执行 kubectl exec、run、logs 等命令时，apiserver 会转发到 kubelet。这里定义 RBAC 规则，授权 apiserver 调用 kubelet API。

kubectl create clusterrolebinding kube-apiserver:kubelet-apis --clusterrole=system:kubelet-api-admin --user kubernetes

##### 参考

关于证书域名最后字符不能是 . 的问题，实际和 Go 的版本有关，1.9 不支持这种类型的证书：<https://github.com/kubernetes/ingress-nginx/issues/2188>

#### controller-manager 集群

本文档介绍部署高可用 kube-controller-manager 集群的步骤。

该集群包含 3 个节点，启动后将通过竞争选举机制产生一个 leader 节点，其它节点为阻塞状态。当 leader 节点不可用后，剩余节点将再次进行选举产生新的 leader 节点，从而保证服务的可用性。

为保证通信安全，本文档先生成 x509 证书和私钥，kube-controller-manager 在如下两种情况下使用该证书：

1. 与 kube-apiserver 的安全端口通信时;
2. 在安全端口(https，10252) 输出 prometheus 格式的 metrics；

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

##### 准备工作

下载最新版本的二进制文件、安装和配置 flanneld 参考：[部署master节点](#_部署master节点)

##### 创建 kube-controller-manager 证书和私钥

cd /opt/k8s/work/cert

cat > kube-controller-manager-csr.json <<EOF

{

"CN": "system:kube-controller-manager",

"key": {

"algo": "rsa",

"size": 2048

},

"hosts": [

"127.0.0.1",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~"

],

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "system:kube-controller-manager",

"OU": "study163"

}

]

}

EOF

* hosts 列表包含所有 kube-controller-manager 节点 IP；
* CN 为 system:kube-controller-manager、O 为 system:kube-controller-manager，kubernetes 内置的 ClusterRoleBindings system:kube-controller-manager 赋予 kube-controller-manager 工作所需的权限。

生成证书和私钥：

cd /opt/k8s/work/cert

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes kube-controller-manager-csr.json | cfssljson -bare kube-controller-manager

ls kube-controller-manager\*pem

将生成的证书和私钥分发到所有 master 节点：

cd /opt/k8s/work/cert

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-controller-manager\*.pem root@${node\_ip}:/etc/kubernetes/cert/

done

##### 创建和分发 kubeconfig 文件

kubeconfig 文件包含访问 apiserver 的所有信息，如 apiserver 地址、CA 证书和自身使用的证书；

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

kubectl config set-cluster kubernetes \

--certificate-authority=/opt/k8s/work/cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-credentials system:kube-controller-manager \

--client-certificate=cert/kube-controller-manager.pem \

--client-key=cert/kube-controller-manager-key.pem \

--embed-certs=true \

--kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-context system:kube-controller-manager \

--cluster=kubernetes \

--user=system:kube-controller-manager \

--kubeconfig=kube-controller-manager.kubeconfig

kubectl config use-context system:kube-controller-manager --kubeconfig=kube-controller-manager.kubeconfig

分发 kubeconfig 到所有 master 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-controller-manager.kubeconfig root@${node\_ip}:/etc/kubernetes/

done

##### 创建和分发 kube-controller-manager systemd unit 文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat > kube-controller-manager.service <<EOF

[Unit]

Description=Kubernetes Controller Manager

Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]

WorkingDirectory=${K8S\_DIR}/kube-controller-manager

ExecStart=/opt/k8s/bin/kube-controller-manager \\

--port=0 \\

--secure-port=10252 \\

--bind-address=127.0.0.1 \\

--kubeconfig=/etc/kubernetes/kube-controller-manager.kubeconfig \\

--authentication-kubeconfig=/etc/kubernetes/kube-controller-manager.kubeconfig \\

--authorization-kubeconfig=/etc/kubernetes/kube-controller-manager.kubeconfig \\

--service-cluster-ip-range=${SERVICE\_CIDR} \\

--cluster-name=kubernetes \\

--cluster-signing-cert-file=/etc/kubernetes/cert/ca.pem \\

--cluster-signing-key-file=/etc/kubernetes/cert/ca-key.pem \\

--experimental-cluster-signing-duration=8760h \\

--root-ca-file=/etc/kubernetes/cert/ca.pem \\

--service-account-private-key-file=/etc/kubernetes/cert/ca-key.pem \\

--leader-elect=true \\

--controllers=\*,bootstrapsigner,tokencleaner \\

--horizontal-pod-autoscaler-use-rest-clients=true \\

--horizontal-pod-autoscaler-sync-period=10s \\

--tls-cert-file=/etc/kubernetes/cert/kube-controller-manager.pem \\

--tls-private-key-file=/etc/kubernetes/cert/kube-controller-manager-key.pem \\

--use-service-account-credentials=true \\

--kube-api-qps=1000 \\

--kube-api-burst=2000 \\

--logtostderr=true \\

--v=2

Restart=on-failure

RestartSec=5

[Install]

WantedBy=multi-user.target

EOF

* --port=0：关闭监听 http /metrics 的请求，同时 --address 参数无效，--bind-address 参数有效；
* --secure-port=10252、--bind-address=0.0.0.0: 在所有网络接口监听 10252 端口的 https /metrics 请求；
* --kubeconfig：指定 kubeconfig 文件路径，kube-controller-manager 使用它连接和验证 kube-apiserver；
* --authentication-kubeconfig 和 --authorization-kubeconfig：kube-controller-manager 使用它连接 apiserver，对 client 的请求进行认证和授权。kube-controller-manager 不再使用 --tls-ca-file 对请求 https metrics 的 Client 证书进行校验。如果没有配置这两个 kubeconfig 参数，则 client 连接 kube-controller-manager https 端口的请求会被拒绝(提示权限不足)。
* --cluster-signing-\*-file：签名 TLS Bootstrap 创建的证书；
* --experimental-cluster-signing-duration：指定 TLS Bootstrap 证书的有效期；
* --root-ca-file：放置到容器 ServiceAccount 中的 CA 证书，用来对 kube-apiserver 的证书进行校验；
* --service-account-private-key-file：签名 ServiceAccount 中 Token 的私钥文件，必须和 kube-apiserver 的 --service-account-key-file 指定的公钥文件配对使用；
* --service-cluster-ip-range ：指定 Service Cluster IP 网段，必须和 kube-apiserver 中的同名参数一致；
* --leader-elect=true：集群运行模式，启用选举功能；被选为 leader 的节点负责处理工作，其它节点为阻塞状态；
* --controllers=\*,bootstrapsigner,tokencleaner：启用的控制器列表，tokencleaner 用于自动清理过期的 Bootstrap token；
* --horizontal-pod-autoscaler-\*：custom metrics 相关参数，支持 autoscaling/v2alpha1；
* --tls-cert-file、--tls-private-key-file：使用 https 输出 metrics 时使用的 Server 证书和秘钥；
* --use-service-account-credentials=true: kube-controller-manager 中各 controller 使用 serviceaccount 访问 kube-apiserver；

分发 systemd unit 文件到所有 master 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-controller-manager.service root@${node\_ip}:/usr/lib/systemd/system/

done

##### kube-controller-manager 的权限

ClusteRole: system:kube-controller-manager 的**权限很小**，只能创建 secret、serviceaccount 等资源对象，各 controller 的权限分散到 ClusterRole system:controller:XXX 中。

需要在 kube-controller-manager 的启动参数中添加 --use-service-account-credentials=true 参数，这样 main controller 会为各 controller 创建对应的 ServiceAccount XXX-controller。

内置的 ClusterRoleBinding system:controller:XXX 将赋予各 XXX-controller ServiceAccount 对应的 ClusterRole system:controller:XXX 权限。

另外，--authentication-kubeconfig 和 --authorization-kubeconfig 参数指定的证书需要有创建 "subjectaccessreviews" 的权限，否则提示:

curl --cacert /opt/k8s/work/cert/ca.pem --cert /opt/k8s/work/cert/admin.pem --key /opt/k8s/work/cert/admin-key.pem https://127.0.0.1:10252/metrics

Internal Server Error: "/metrics": subjectaccessreviews.authorization.k8s.io is forbidden: User "system:kube-controller-manager" cannot create resource "subjectaccessreviews" in API group "authorization.k8s.io" at the cluster scope

解决办法是创建一个 ClusterRoleBinding，赋予相应的权限：

kubectl create clusterrolebinding controller-manager:system:auth-delegator --user system:kube-controller-manager --clusterrole system:auth-delegator

clusterrolebinding.rbac.authorization.k8s.io/controller-manager:system:auth-delegator created

参考：<https://github.com/kubernetes/kubeadm/issues/1285>

##### 启动 kube-controller-manager 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${K8S\_DIR}/kube-controller-manager"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable kube-controller-manager && systemctl restart kube-controller-manager"

done

* 必须创建工作目录；

##### 检查服务运行状态

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status kube-controller-manager|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u kube-controller-manager

##### 查看输出的 metric

注意：以下命令在 kube-controller-manager 节点上执行。

kube-controller-manager 监听 10252 端口，接收 https 请求：

sudo netstat -lnpt|grep kube-controll

tcp 0 0 127.0.0.1:10252 0.0.0.0:\* LISTEN 55755/kube-controll

curl --cacert /opt/k8s/work/cert/ca.pem --cert /opt/k8s/work/cert/admin.pem --key /opt/k8s/work/cert/admin-key.pem https://127.0.0.1:10252/metrics

{

"kind": "Status",

"apiVersion": "v1",

"metadata": {

},

"status": "Failure",

"message": "forbidden: User \"system:anonymous\" cannot get path \"/metrics\"",

"reason": "Forbidden",

"details": {

},

"code": 403

}

将 kube-controller-manager 的日志级别设置为 4 后，可以看到原因是：

journalctl -u kube-controller-manager -f |grep /metrics

2月 22 19:07:28 m7-inf-prod01 kube-controller-manager[1416748]: I0222 19:07:28.003325 1416748 authorization.go:73] Forbidden: "/metrics", Reason: "no RBAC policy matched"

2月 22 19:07:28 m7-inf-prod01 kube-controller-manager[1416748]: I0222 19:07:28.003472 1416748 wrap.go:42] GET /metrics: (2.600519ms) 403 [curl/7.29.0 127.0.0.1:36324]

这是由于没有部署 metrics-server 的缘故。后续在 metrics-server插件 一文中将介绍部署 metrics-server 的步骤。

参考：<https://github.com/kubernetes-incubator/metrics-server/issues/85>

##### 测试 kube-controller-manager 集群的高可用

停掉一个或两个节点的 kube-controller-manager 服务，观察其它节点的日志，看是否获取了 leader 权限。

##### 查看当前的 leader

kubectl get endpoints kube-controller-manager --namespace=kube-system -o yaml

apiVersion: v1

kind: Endpoints

metadata:

annotations:

control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":"node1\_084534e2-6cc4-11e8-a418-5254001f5b65","leaseDurationSeconds":15,"acquireTime":"2018-06-10T15:40:33Z","renewTime":"2018-06-10T16:19:08Z","leaderTransitions":12}'

creationTimestamp: 2018-06-10T13:59:42Z

name: kube-controller-manager

namespace: kube-system

resourceVersion: "4540"

selfLink: /api/v1/namespaces/kube-system/endpoints/kube-controller-manager

uid: 862cc048-6cb6-11e8-96fa-525400ba84c6

可见，当前的 leader 为 master 节点。

##### 参考

关于 controller 权限和 use-service-account-credentials 参数：

<https://github.com/kubernetes/kubernetes/issues/48208>

kublet 认证和授权：

[https://kubernetes.io/docs/admin/kubelet-authentication-authorization/#kubelet-authorization](https://kubernetes.io/docs/admin/kubelet-authentication-authorization/" \l "kubelet-authorization)

#### scheduler 集群

本章节介绍部署高可用 kube-scheduler 集群的步骤。

该集群包含 3 个节点，启动后将通过竞争选举机制产生一个 leader 节点，其它节点为阻塞状态。当 leader 节点不可用后，剩余节点将再次进行选举产生新的 leader 节点，从而保证服务的可用性。

为保证通信安全，本文档先生成 x509 证书和私钥，kube-scheduler 在如下两种情况下使用该证书：

1. 与 kube-apiserver 的安全端口通信;
2. 在**安全端口**(https，10251) 输出 prometheus 格式的 metrics；

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

##### 准备工作

下载最新版本的二进制文件、安装和配置 flanneld 参考：[部署master节点](#_部署master节点)

##### 创建 kube-scheduler 证书和私钥

创建证书签名请求：

cd /opt/k8s/work/cert

cat > kube-scheduler-csr.json <<EOF

{

"CN": "system:kube-scheduler",

"hosts": [

"127.0.0.1",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~"

],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "system:kube-scheduler",

"OU": "study163"

}

]

}

EOF

* hosts 列表包含**所有** kube-scheduler 节点 IP；
* CN 为 system:kube-scheduler、O 为 system:kube-scheduler，kubernetes 内置的 ClusterRoleBindings system:kube-scheduler 将赋予 kube-scheduler 工作所需的权限。

生成证书和私钥：

cd /opt/k8s/work/cert

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes kube-scheduler-csr.json | cfssljson -bare kube-scheduler

ls kube-scheduler\*pem

##### 创建和分发 kubeconfig 文件

kubeconfig 文件包含访问 apiserver 的所有信息，如 apiserver 地址、CA 证书和自身使用的证书；

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

kubectl config set-cluster kubernetes \

--certificate-authority=cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=kube-scheduler.kubeconfig

kubectl config set-credentials system:kube-scheduler \

--client-certificate=cert/kube-scheduler.pem \

--client-key=cert/kube-scheduler-key.pem \

--embed-certs=true \

--kubeconfig=kube-scheduler.kubeconfig

kubectl config set-context system:kube-scheduler \

--cluster=kubernetes \

--user=system:kube-scheduler \

--kubeconfig=kube-scheduler.kubeconfig

kubectl config use-context system:kube-scheduler --kubeconfig=kube-scheduler.kubeconfig

* 上一步创建的证书、私钥以及 kube-apiserver 地址被写入到 kubeconfig 文件中；

分发 kubeconfig 到所有 master 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-scheduler.kubeconfig root@${node\_ip}:/etc/kubernetes/

done

##### 创建 kube-scheduler 配置文件

cat <<EOF | sudo tee kube-scheduler.yaml

apiVersion: componentconfig/v1alpha1

kind: KubeSchedulerConfiguration

clientConnection:

kubeconfig: "/etc/kubernetes/kube-scheduler.kubeconfig"

leaderElection:

leaderElect: true

EOF

* --kubeconfig：指定 kubeconfig 文件路径，kube-scheduler 使用它连接和验证 kube-apiserver；
* --leader-elect=true：集群运行模式，启用选举功能；被选为 leader 的节点负责处理工作，其它节点为阻塞状态；

分发 kube-scheduler 配置文件到所有 master 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-scheduler.yaml root@${node\_ip}:/etc/kubernetes/

done

##### 创建和分发 kube-scheduler systemd unit 文件

cd /opt/k8s/work

cat > kube-scheduler.service <<EOF

[Unit]

Description=Kubernetes Scheduler

Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]

WorkingDirectory=${K8S\_DIR}/kube-scheduler

ExecStart=/opt/k8s/bin/kube-scheduler \\

--config=/etc/kubernetes/kube-scheduler.yaml \\

--address=127.0.0.1 \\

--kube-api-qps=100 \\

--logtostderr=true \\

--v=2

Restart=always

RestartSec=5

StartLimitInterval=0

[Install]

WantedBy=multi-user.target

EOF

* --address：在 127.0.0.1:10251 端口接收 http /metrics 请求；kube-scheduler 目前还不支持接收 https 请求；

分发 systemd unit 文件到所有 master 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp kube-scheduler.service root@${node\_ip}:/etc/systemd/system/

done

##### 启动 kube-scheduler 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${K8S\_DIR}/kube-scheduler"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable kube-scheduler && systemctl restart kube-scheduler"

done

* 必须先创建工作目录；

##### 检查服务运行状态

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status kube-scheduler|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u kube-scheduler

##### 查看输出的 metric

注意：以下命令在 kube-scheduler 节点上执行。

kube-scheduler 监听 10251 端口，接收 http 请求：

sudo netstat -lnpt|grep kube-sche

tcp 0 0 127.0.0.1:10251 0.0.0.0:\* LISTEN 23783/kube-schedule

curl -s http://127.0.0.1:10251/metrics |head

# HELP apiserver\_audit\_event\_total Counter of audit events generated and sent to the audit backend.

# TYPE apiserver\_audit\_event\_total counter

apiserver\_audit\_event\_total 0

# HELP apiserver\_client\_certificate\_expiration\_seconds Distribution of the remaining lifetime on the certificate used to authenticate a request.

# TYPE apiserver\_client\_certificate\_expiration\_seconds histogram

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="0"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="21600"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="43200"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="86400"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="172800"} 0

##### 测试 kube-scheduler 集群的高可用

随便找一个或两个 master 节点，停掉 kube-scheduler 服务，看其它节点是否获取了 leader 权限（systemd 日志）。

##### 查看当前的 leader

kubectl get endpoints kube-scheduler --namespace=kube-system -o yaml

apiVersion: v1

kind: Endpoints

metadata:

annotations:

control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":"k8s-1\_1ea20a43-4f8e-11e9-9ac5-000c2955d204","leaseDurationSeconds":15,"acquireTime":"2019-03-26T06:12:24Z","renewTime":"2019-03-26T06:23:54Z","leaderTransitions":0}'

creationTimestamp: 2019-03-26T06:12:24Z

name: kube-scheduler

namespace: kube-system

resourceVersion: "72504"

selfLink: /api/v1/namespaces/kube-system/endpoints/kube-scheduler

uid: 1f4b7394-4f8e-11e9-a869-000c2955d204

可见，当前的 leader 为 master 节点。

### 部署 worker 节点

kubernetes work 节点运行如下组件：

* docker
* kubelet
* kube-proxy
* flanneld
* kube-nginx

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

**前提条件**：

安装和配置 flanneld

参考 [部署flannel网络](#_安装flannel网络插件)

安装和配置 kube-nginx

参考 [apiserver高可用之nginx代理](#_apiserver_高可用之nginx代理)

**安装依赖包：**

ssh root@${node\_ip} "yum install -y epel-release"

ssh root@${node\_ip} "yum install -y conntrack ipvsadm ipset jq iptables curl sysstat libseccomp && /usr/sbin/modprobe ip\_vs"

#### 部署 docker 组件

docker 是容器的运行环境，管理它的生命周期。kubelet 通过 Container Runtime Interface (CRI) 与 docker 进行交互。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

docker 有多种安装方式：本章节中我们使用最简单的安装方式 脚本安装，使用脚本安装不需要我们关注依赖运行脚本就能自动安装。

不过这种方式不安全，而且必须要依赖网络，不推荐在生产环境中使用；生成环境建议使用二进制脚本安装。

##### 安装依赖包

参考 [部署worker节点](#_部署_worker_节点)

##### 下载和分发 docker 安装文件

**使用脚本安装docker：**

curl -fLsS https://get.docker.com/ | sh

等待安装完成，也可以指定使用 Aliyun 镜像

curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun

**使用二进制文件安装docker：**

到 <https://download.docker.com/linux/static/stable/x86_64/> 页面下载最新发布包：

cd /opt/k8s/work

wget https://download.docker.com/linux/static/stable/x86\_64/docker-18.09.0.tgz

tar -xvf docker-18.09.0.tgz

分发二进制文件到所有 worker 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp docker/\* root@${node\_ip}:/opt/k8s/bin/

ssh root@${node\_ip} "chmod +x /opt/k8s/bin/\*"

done

##### 创建和分发 systemd unit 文件

cd /opt/k8s/work

cat > docker.service <<"EOF"

[Unit]

Description=Docker Application Container Engine

Documentation=http://docs.docker.io

[Service]

WorkingDirectory=##DOCKER\_DIR##

Environment="PATH=/opt/k8s/bin:/bin:/sbin:/usr/bin:/usr/sbin"

EnvironmentFile=-/run/flannel/docker

ExecStart=/opt/k8s/bin/dockerd $DOCKER\_NETWORK\_OPTIONS

ExecReload=/bin/kill -s HUP $MAINPID

Restart=on-failure

RestartSec=5

LimitNOFILE=infinity

LimitNPROC=infinity

LimitCORE=infinity

Delegate=yes

KillMode=process

[Install]

WantedBy=multi-user.target

EOF

* EOF 前后有双引号，这样 bash 不会替换文档中的变量，如 $DOCKER\_NETWORK\_OPTIONS；
* dockerd 运行时会调用其它 docker 命令，如 docker-proxy，所以需要将 docker 命令所在的目录加到 PATH 环境变量中；
* flanneld 启动时将网络配置写入 /run/flannel/docker 文件中，dockerd 启动前读取该文件中的环境变量 DOCKER\_NETWORK\_OPTIONS ，然后设置 docker0 网桥网段；
* 如果指定了多个 EnvironmentFile 选项，则必须将 /run/flannel/docker 放在最后(确保 docker0 使用 flanneld 生成的 bip 参数)；
* docker 需要以 root 用于运行；
* docker 从 1.13 版本开始，可能将 iptables FORWARD chain的默认策略设置为DROP，从而导致 ping 其它 Node 上的 Pod IP 失败，遇到这种情况时，需要手动设置策略为 ACCEPT：

sudo iptables -P FORWARD ACCEPT

并且把以下命令写入 /etc/rc.local 文件中，防止节点重启iptables FORWARD chain的默认策略又还原为DROP

/sbin/iptables -P FORWARD ACCEPT

分发 systemd unit 文件到所有 worker 机器:

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

sed -i -e "s|##DOCKER\_DIR##|${DOCKER\_DIR}|" docker.service

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp docker.service root@${node\_ip}:/usr/lib/systemd/system/

done

##### 配置和分发 docker 配置文件

使用国内的仓库镜像服务器以加快 pull image 的速度，同时增加下载的并发数 (需要重启 dockerd 生效)：

cd /opt/k8s/work

cat > docker-daemon.json <<EOF

{

"registry-mirrors": ["https://hub-mirror.c.163.com", "https://docker.mirrors.ustc.edu.cn"],

"insecure-registries": ["docker02:35000"],

"max-concurrent-downloads": 20,

"live-restore": true,

"max-concurrent-uploads": 10,

"debug": true,

"data-root": "/data/k8s/docker/data",

"exec-root": "/data/k8s/docker/exec",

"log-opts": {

"max-size": "100m",

"max-file": "5"

}

}

EOF

分发 docker 配置文件到所有 work 节点：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p /etc/docker/ ${DOCKER\_DIR}/{data,exec}"

scp docker-daemon.json root@${node\_ip}:/etc/docker/daemon.json

done

##### 启动 docker 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl stop firewalld && systemctl disable firewalld"

ssh root@${node\_ip} "/usr/sbin/iptables -F && /usr/sbin/iptables -X && /usr/sbin/iptables -F -t nat && /usr/sbin/iptables -X -t nat"

ssh root@${node\_ip} "/usr/sbin/iptables -P FORWARD ACCEPT"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable docker && systemctl restart docker"

#ssh root@${node\_ip} 'for intf in /sys/devices/virtual/net/docker0/brif/\*; do echo 1 > $intf/hairpin\_mode; done'

ssh root@${node\_ip} "sudo sysctl -p /etc/sysctl.d/kubernetes.conf"

done

* 关闭 firewalld(centos7)/ufw(ubuntu16.04)，否则可能会重复创建 iptables 规则；
* 清理旧的 iptables rules 和 chains 规则；
* 开启 docker0 网桥下虚拟网卡的 hairpin 模式;

##### 检查服务运行状态

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status docker|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u docker

##### 检查 docker0 网桥

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "/usr/sbin/ip addr show flannel.1 && /usr/sbin/ip addr show docker0"

done

确认各 work 节点的 docker0 网桥和 flannel.1 接口的 IP 处于同一个网段中(如下 172.30.112.0/32 位于 172.30.112.1/21 中)：

>>> ~~192.168.100.246~~

4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1450 qdisc noqueue state UNKNOWN group default

link/ether fa:32:14:34:86:74 brd ff:ff:ff:ff:ff:ff

inet 172.30.240.0/32 scope global flannel.1

valid\_lft forever preferred\_lft forever

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default

link/ether 02:42:7f:58:af:18 brd ff:ff:ff:ff:ff:ff

inet 172.30.240.1/21 brd 172.30.247.255 scope global docker0

valid\_lft forever preferred\_lft forever

#### 部署kubelet组件

kublet 运行在每个 worker 节点上，接收 kube-apiserver 发送的请求，管理 Pod 容器，执行交互式命令，如 exec、run、logs 等。

kublet 启动时自动向 kube-apiserver 注册节点信息，内置的 cadvisor 统计和监控节点的资源使用情况。

为确保安全，本文档只开启接收 https 请求的安全端口，对请求进行认证和授权，拒绝未授权的访问(如 apiserver、heapster)。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

##### 下载和分发 kubelet 二进制文件

参考 [部署master节点](#_部署master节点_1)

##### 安装依赖包

参考 [部署worker节点](#_部署_worker_节点)

##### 创建 kubelet bootstrap kubeconfig 文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_name in ${NODE\_NAMES[@]}

do

echo ">>> ${node\_name}"

# 创建 token

export BOOTSTRAP\_TOKEN=$(kubeadm token create \

--description kubelet-bootstrap-token \

--groups system:bootstrappers:${node\_name} \

--kubeconfig ~/.kube/config)

# 设置集群参数

kubectl config set-cluster kubernetes \

--certificate-authority=/etc/kubernetes/cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=kubelet-bootstrap-${node\_name}.kubeconfig

# 设置客户端认证参数

kubectl config set-credentials kubelet-bootstrap \

--token=${BOOTSTRAP\_TOKEN} \

--kubeconfig=kubelet-bootstrap-${node\_name}.kubeconfig

# 设置上下文参数

kubectl config set-context default \

--cluster=kubernetes \

--user=kubelet-bootstrap \

--kubeconfig=kubelet-bootstrap-${node\_name}.kubeconfig

# 设置默认上下文

kubectl config use-context default --kubeconfig=kubelet-bootstrap-${node\_name}.kubeconfig

done

* 证书中写入 Token 而非证书，证书后续由 kube-controller-manager 创建。

查看 kubeadm 为各节点创建的 token：

kubeadm token list --kubeconfig ~/.kube/config

TOKEN TTL EXPIRES USAGES DESCRIPTION EXTRA GROUPS

33xdd2.b04al5zmia3tnnf8 23h 2019-04-01T14:06:46+08:00 authentication,signing kubelet-bootstrap-token system:bootstrappers:node2

ch6fro.v9nps3d5z0lcmyrd 23h 2019-04-01T14:06:25+08:00 authentication,signing kubelet-bootstrap-token system:bootstrappers:master

napqqj.wi1qhq2gjel1dvia 23h 2019-04-01T14:06:35+08:00 authentication,signing kubelet-bootstrap-token system:bootstrappers:node1

* 创建的 token 有效期为 1 天，超期后将不能再被使用，且会被 kube-controller-manager 的 tokencleaner 清理(如果启用该 controller 的话)；
* kube-apiserver 接收 kubelet 的 bootstrap token 后，将请求的 user 设置为 system:bootstrap:，group 设置为 system:bootstrappers；

查看各 token 关联的 Secret：

kubectl get secrets -n kube-system|grep bootstrap-token

bootstrap-token-503xx3 bootstrap.kubernetes.io/token 7 40m

bootstrap-token-cjadiv bootstrap.kubernetes.io/token 7 45m

bootstrap-token-iifyk9 bootstrap.kubernetes.io/token 7 40m

##### 分发 bootstrap kubeconfig 文件到所有 worker 节点

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_name in ${NODE\_NAMES[@]}

do

echo ">>> ${node\_name}"

scp kubelet-bootstrap-${node\_name}.kubeconfig root@${node\_name}:/etc/kubernetes/kubelet-bootstrap.kubeconfig

done

##### 创建和分发 kubelet 参数配置文件

从 v1.10 开始，kubelet 部分参数需在配置文件中配置，kubelet --help 会提示：

DEPRECATED: This parameter should be set via the config file specified by the Kubelet's --config flag

创建 kubelet 参数配置模板文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

cat <<EOF | tee kubelet-config.yaml.template

kind: KubeletConfiguration

apiVersion: kubelet.config.k8s.io/v1beta1

authentication:

anonymous:

enabled: false

webhook:

enabled: true

x509:

clientCAFile: "/etc/kubernetes/cert/ca.pem"

authorization:

mode: Webhook

clusterDomain: "${CLUSTER\_DNS\_DOMAIN}"

clusterDNS:

- "${CLUSTER\_DNS\_SVC\_IP}"

podCIDR: "${CLUSTER\_CIDR}"

maxPods: 220

serializeImagePulls: false

hairpinMode: promiscuous-bridge

cgroupDriver: cgroupfs

runtimeRequestTimeout: "15m"

rotateCertificates: true

serverTLSBootstrap: true

readOnlyPort: 0

port: 10250

address: "##NODE\_IP##"

EOF

* address：API 监听地址，不能为 127.0.0.1，否则 kube-apiserver、heapster 等不能调用 kubelet 的 API；
* readOnlyPort=0：关闭只读端口(默认 10255)，等效为未指定；
* authentication.anonymous.enabled：设置为 false，不允许匿名访问 10250 端口；
* authentication.x509.clientCAFile：指定签名客户端证书的 CA 证书，开启 HTTP 证书认证；
* authentication.webhook.enabled=true：开启 HTTPs bearer token 认证；
* 对于未通过 x509 证书和 webhook 认证的请求(kube-apiserver 或其他客户端)，将被拒绝，提示 Unauthorized；
* authroization.mode=Webhook：kubelet 使用 SubjectAccessReview API 查询 kube-apiserver 某 user、group 是否具有操作资源的权限(RBAC)；
* featureGates.RotateKubeletClientCertificate、featureGates.RotateKubeletServerCertificate：自动 rotate 证书，证书的有效期取决于 kube-controller-manager 的 --experimental-cluster-signing-duration 参数；
* 需要 root 账户运行；

为各节点创建和分发 kubelet 配置文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

sed -e "s/##NODE\_IP##/${node\_ip}/" kubelet-config.yaml.template > kubelet-config-${node\_ip}.yaml.template

scp kubelet-config-${node\_ip}.yaml.template root@${node\_ip}:/etc/kubernetes/kubelet-config.yaml

done

##### 创建和分发 kubelet systemd unit 文件

创建 kubelet systemd unit 文件模板：

cd /opt/k8s/work

cat > kubelet.service.template <<EOF

[Unit]

Description=Kubernetes Kubelet

Documentation=https://github.com/GoogleCloudPlatform/kubernetes

After=docker.service

Requires=docker.service

[Service]

WorkingDirectory=${K8S\_DIR}/kubelet

ExecStart=/opt/k8s/bin/kubelet \\

--root-dir=${K8S\_DIR}/kubelet \\

--bootstrap-kubeconfig=/etc/kubernetes/kubelet-bootstrap.kubeconfig \\

--cert-dir=/etc/kubernetes/cert \\

--kubeconfig=/etc/kubernetes/kubelet.kubeconfig \\

--config=/etc/kubernetes/kubelet-config.yaml \\

--hostname-override=##NODE\_NAME## \\

--pod-infra-container-image=xiaochunping/pause:3.1

--allow-privileged=true \\

--event-qps=0 \\

--kube-api-qps=1000 \\

--kube-api-burst=2000 \\

--registry-qps=0 \\

--image-pull-progress-deadline=30m \\

--logtostderr=true \\

--v=2

Restart=always

RestartSec=5

StartLimitInterval=0

[Install]

WantedBy=multi-user.target

EOF

* 如果设置了 --hostname-override 选项，则 kube-proxy 也需要设置该选项，否则会出现找不到 Node 的情况；
* --bootstrap-kubeconfig：指向 bootstrap kubeconfig 文件，kubelet 使用该文件中的用户名和 token 向 kube-apiserver 发送 TLS Bootstrapping 请求；
* K8S approve kubelet 的 csr 请求后，在 --cert-dir 目录创建证书和私钥文件，然后写入 --kubeconfig 文件；
* --pod-infra-container-image 不使用 redhat 的 pod-infrastructure:latest 镜像，它不能回收容器的僵尸；

为各节点创建和分发 kubelet systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_name in ${NODE\_NAMES[@]}

do

echo ">>> ${node\_name}"

sed -e "s/##NODE\_NAME##/${node\_name}/" kubelet.service.template > kubelet-${node\_name}.service

scp kubelet-${node\_name}.service root@${node\_name}:/usr/lib/systemd/system/kubelet.service

done

##### Bootstrap Token Auth 和授予权限

kublet 启动时查找配置的 --kubeletconfig 文件是否存在，如果不存在则使用 --bootstrap-kubeconfig 向 kube-apiserver 发送证书签名请求 (CSR)。

kube-apiserver 收到 CSR 请求后，对其中的 Token 进行认证（事先使用 kubeadm 创建的 token），认证通过后将请求的 user 设置为 system:bootstrap:<TokenID>，group 设置为 system:bootstrappers，这一过程称为 Bootstrap Token Auth。

默认情况下，这个 user 和 group 没有创建 CSR 的权限，kubelet 启动失败，错误日志如下：

sudo journalctl -u kubelet -a |grep -A 2 'certificatesigningrequests'

May 06 06:42:36 k8s-1 kubelet[26986]: F0506 06:42:36.314378 26986 server.go:233] failed to run Kubelet: cannot create certificate signing request: certificatesigningrequests.certificates.k8s.io is forbidden: User "system:bootstrap:lemy40" cannot create certificatesigningrequests.certificates.k8s.io at the cluster scope

May 06 06:42:36 k8s-1 systemd[1]: kubelet.service: Main process exited, code=exited, status=255/n/a

May 06 06:42:36 k8s-1 systemd[1]: kubelet.service: Failed with result 'exit-code'.

解决办法是：创建一个 clusterrolebinding，将 group system:bootstrappers 和 clusterrole system:node-bootstrapper 绑定：

kubectl create clusterrolebinding kubelet-bootstrap --clusterrole=system:node-bootstrapper --group=system:bootstrappers

##### 启动 kubelet 服务

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${K8S\_DIR}/kubelet"

ssh root@${node\_ip} "/usr/sbin/swapoff -a"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable kubelet && systemctl restart kubelet"

done

* 必须创建工作目录；
* 关闭 swap 分区，否则 kubelet 会启动失败；

journalctl -u kubelet |tail

3月 29 15:01:53 k8s-1 systemd[1]: Job kubelet.service/start failed with result 'dependency'.

3月 29 15:03:23 k8s-1 systemd[1]: Dependency failed for Kubernetes Kubelet.

3月 29 15:03:23 k8s-1 systemd[1]: Job kubelet.service/start failed with result 'dependency'.

-- Reboot --

3月 30 18:48:42 k8s-1 systemd[1]: Cannot add dependency job for unit kubelet.service, ignoring: Unit not found.

3月 31 14:41:17 k8s-1 systemd[1]: [/usr/lib/systemd/system/kubelet.service:10] Unknown lvalue '--allow-privileged' in section 'Service'

3月 31 14:41:17 k8s-1 systemd[1]: [/usr/lib/systemd/system/kubelet.service:10] Unknown lvalue '--allow-privileged' in section 'Service'

3月 31 14:41:17 k8s-1 systemd[1]: Started Kubernetes Kubelet.

3月 31 14:41:17 k8s-1 kubelet[29768]: I0331 14:41:17.880649 29768 server.go:408] Version: v1.12.3

3月 31 14:41:17 k8s-1 kubelet[29768]: I0331 14:41:17.881049 29768 plugins.go:99] No cloud provider specified.

kubelet 启动后使用 --bootstrap-kubeconfig 向 kube-apiserver 发送 CSR 请求，当这个 CSR 被 approve 后，kube-controller-manager 为 kubelet 创建 TLS 客户端证书、私钥和 --kubeletconfig 文件。

注意：kube-controller-manager 需要配置 --cluster-signing-cert-file 和 --cluster-signing-key-file 参数，才会为 TLS Bootstrap 创建证书和私钥。

kubectl get csr

NAME AGE REQUESTOR CONDITION

node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M 22h system:bootstrap:f9isxw Pending

node-csr-IHHUpaUceWdFY7QPRw48ICdr\_pyt1VC-KgLM9SfNWE8 23h system:bootstrap:8w1g0a Pending

node-csr-xqKuMm\_MsfUGrMiwti1EaaJMsOpWwvlOQIT\_N\_3a02k 22h system:bootstrap:2b84b8

kubectl get nodes

No resources found.

* 三个 work 节点的 csr 均处于 pending 状态；

##### 检查服务运行状态

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status kubelet|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u kubelet

##### approve kubelet CSR 请求

可以手动或自动 approve CSR 请求。推荐使用自动的方式，因为从 v1.8 版本开始，可以自动轮转approve csr 后生成的证书。

###### 手动 approve CSR 请求

查看 SCR 列表：

kubectl get csr

NAME AGE REQUESTOR CONDITION

node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M 22h system:bootstrap:f9isxw Pending

node-csr-IHHUpaUceWdFY7QPRw48ICdr\_pyt1VC-KgLM9SfNWE8 23h system:bootstrap:8w1g0a Pending

node-csr-xqKuMm\_MsfUGrMiwti1EaaJMsOpWwvlOQIT\_N\_3a02k 22h system:bootstrap:2b84b8

approve SCR：

kubectl certificate approve node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M

certificatesigningrequest.certificates.k8s.io "node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M" approved

查看 Approve 结果：

kubectl describe csr node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M

###### 自动 approve CSR 请求

创建三个 ClusterRoleBinding，分别用于自动 approve client、renew client、renew server 证书：

cd /opt/k8s/work

cat > csr-crb.yaml <<EOF

# Approve all CSRs for the group "system:bootstrappers"

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: auto-approve-csrs-for-group

subjects:

- kind: Group

name: system:bootstrappers

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: system:certificates.k8s.io:certificatesigningrequests:nodeclient

apiGroup: rbac.authorization.k8s.io

---

# To let a node of the group "system:nodes" renew its own credentials

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: node-client-cert-renewal

subjects:

- kind: Group

name: system:nodes

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: system:certificates.k8s.io:certificatesigningrequests:selfnodeclient

apiGroup: rbac.authorization.k8s.io

---

# A ClusterRole which instructs the CSR approver to approve a node requesting a

# serving cert matching its client cert.

kind: ClusterRole

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: approve-node-server-renewal-csr

rules:

- apiGroups: ["certificates.k8s.io"]

resources: ["certificatesigningrequests/selfnodeserver"]

verbs: ["create"]

---

# To let a node of the group "system:nodes" renew its own server credentials

kind: ClusterRoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: node-server-cert-renewal

subjects:

- kind: Group

name: system:nodes

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: approve-node-server-renewal-csr

apiGroup: rbac.authorization.k8s.io

EOF

* auto-approve-csrs-for-group：自动 approve node 的第一次 CSR； 注意第一次 CSR 时，请求的 Group 为 system:bootstrappers；
* node-client-cert-renewal：自动 approve node 后续过期的 client 证书，自动生成的证书 Group 为 system:nodes;
* node-server-cert-renewal：自动 approve node 后续过期的 server 证书，自动生成的证书 Group 为 system:nodes;

生效配置：

kubectl apply -f csr-crb.yaml

clusterrolebinding.rbac.authorization.k8s.io/auto-approve-csrs-for-group created

clusterrolebinding.rbac.authorization.k8s.io/node-client-cert-renewal created

clusterrole.rbac.authorization.k8s.io/approve-node-server-renewal-csr created

clusterrolebinding.rbac.authorization.k8s.io/node-server-cert-renewal created

##### 查看 kublet 的情况

等待一段时间(1-10 分钟)，三个节点的 CSR 都被自动 approved：

kubectl get csr

NAME AGE REQUESTOR CONDITION

node-csr--BjlTzxB5Y4op\_6wYlDKbbQj1NtX-IOBMLmWhkupEWA 4m system:bootstrap:8galm1 Approved,Issued

node-csr-a68FhmUgprTJkaLwnJOLQLOkDQuAviDdBy91ByVtWt0 4m system:bootstrap:4ef7hj Approved,Issued

node-csr-a7DI6d0QjBiPh58IBGYFPUKAZvKs6sfbqlnoc22erRs 4m system:bootstrap:ai162m Approved,Issued

* Pending 的 CSR 用于创建 kubelet server 证书，需要手动 approve，下文介绍。

所有节点均 ready：

kubectl get nodes

NAME STATUS ROLES AGE VERSION

master Ready <none> 2m58s v1.12.3

node1 Ready <none> 2m54s v1.12.3

node2 Ready <none> 2m56s v1.12.3

kube-controller-manager 为各 node 生成了 kubeconfig 文件和公私钥：

ls -l /etc/kubernetes/kubelet.kubeconfig

-rw------- 1 root root 2298 3月 28 17:45 /etc/kubernetes/kubelet.kubeconfig

ls -l /etc/kubernetes/cert/|grep kubelet

-rw------- 1 root root 1265 3月 28 17:45 kubelet-client-2019-03-28-17-45-48.pem

lrwxrwxrwx 1 root root 58 3月 28 17:45 kubelet-client-current.pem -> /etc/kubernetes/cert/kubelet-client-2019-03-28-17-45-48.pem

* 没有自动生成 kubelet server 证书；

##### 手动 approve server cert csr

基于[安全性考虑](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/" \l "kubelet-configuration)，CSR approving controllers 默认不会自动 approve kubelet server 证书签名请求，需要手动 approve。

kubectl get csr

NAME AGE REQUESTOR CONDITION

csr-42fr8 7m45s system:node:master Pending

csr-5gwjj 7m43s system:node:node2 Pending

csr-klkcp 7m41s system:node:node1 Pending

node-csr-7YSbJGsZGmAH5IrdOmJVcGDrUK3iGtFiDHuRMVdJk5M 23h system:bootstrap:f9isxw Approved,Issued

node-csr-IHHUpaUceWdFY7QPRw48ICdr\_pyt1VC-KgLM9SfNWE8 24h system:bootstrap:8w1g0a Approved,Issued

node-csr-xqKuMm\_MsfUGrMiwti1EaaJMsOpWwvlOQIT\_N\_3a02k 23h system:bootstrap:2b84b8 Approved,Issued

kubectl certificate approve csr-42fr8

kubectl certificate approve csr-5gwjj

kubectl certificate approve csr-klkcp

certificatesigningrequest.certificates.k8s.io/csr-42fr8 approved

certificatesigningrequest.certificates.k8s.io/csr-5gwjj approved

certificatesigningrequest.certificates.k8s.io/csr-klkcp approved

ls -l /etc/kubernetes/cert/kubelet-\*

-rw------- 1 root root 1265 3月 28 17:45 /etc/kubernetes/cert/kubelet-client-2019-03-28-17-45-48.pem

lrwxrwxrwx 1 root root 58 3月 28 17:45 /etc/kubernetes/cert/kubelet-client-current.pem -> /etc/kubernetes/cert/kubelet-client-2019-03-28-17-45-48.pem

-rw------- 1 root root 1301 3月 28 17:58 /etc/kubernetes/cert/kubelet-server-2019-03-28-17-58-23.pem

lrwxrwxrwx 1 root root 58 3月 28 17:58 /etc/kubernetes/cert/kubelet-server-current.pem -> /etc/kubernetes/cert/kubelet-server-2019-03-28-17-58-23.pem

##### kubelet 提供的 API 接口

kublet 启动后监听多个端口，用于接收 kube-apiserver 或其它组件发送的请求：

sudo netstat -lnpt|grep kubelet

tcp 0 0 ~~192.168.100.246~~:10250 0.0.0.0:\* LISTEN 8958/kubelet

tcp 0 0 127.0.0.1:41848 0.0.0.0:\* LISTEN 8958/kubelet

tcp 0 0 127.0.0.1:10248 0.0.0.0:\* LISTEN 8958/kubelet

* 10248: healthz http 服务；
* 10250: https API 服务；注意：未开启只读端口 10255；

例如执行 kubectl exec -it nginx-ds-5rmws -- sh 命令时，kube-apiserver 会向 kubelet 发送如下请求：

POST /exec/default/nginx-ds-5rmws/my-nginx?command=sh&input=1&output=1&tty=1

kubelet 接收 10250 端口的 https 请求：

* /pods、/runningpods
* /metrics、/metrics/cadvisor、/metrics/probes
* /spec
* /stats、/stats/container
* /logs
* /run/、"/exec/", "/attach/", "/portForward/", "/containerLogs/" 等管理；

详情参考：[https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/server/server.go#L434:3](https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/server/server.go" \l "L434:3)

由于关闭了匿名认证，同时开启了 webhook 授权，所有访问 10250 端口 https API 的请求都需要被认证和授权。

预定义的 ClusterRole system:kubelet-api-admin 授予访问 kubelet 所有 API 的权限(kube-apiserver 使用的 kubernetes 证书 User 授予了该权限)：

kubectl describe clusterrole system:kubelet-api-admin

Name: system:kubelet-api-admin

Labels: kubernetes.io/bootstrapping=rbac-defaults

Annotations: rbac.authorization.kubernetes.io/autoupdate: true

PolicyRule:

Resources Non-Resource URLs Resource Names Verbs

--------- ----------------- -------------- -----

nodes/log [] [] [\*]

nodes/metrics [] [] [\*]

nodes/proxy [] [] [\*]

nodes/spec [] [] [\*]

nodes/stats [] [] [\*]

nodes [] [] [get list watch proxy]

##### kublet api 认证和授权

kublet 配置了如下认证参数：

* authentication.anonymous.enabled：设置为 false，不允许匿名访问 10250 端口；
* authentication.x509.clientCAFile：指定签名客户端证书的 CA 证书，开启 HTTPs 证书认证；
* authentication.webhook.enabled=true：开启 HTTPs bearer token 认证；

同时配置了如下授权参数：

* authroization.mode=Webhook：开启 RBAC 授权；

kubelet 收到请求后，使用 clientCAFile 对证书签名进行认证，或者查询 bearer token 是否有效。如果两者都没通过，则拒绝请求，提示 Unauthorized：

curl -s --cacert /etc/kubernetes/cert/ca.pem https://~~192.168.100.246~~:10250/metrics

Unauthorized

curl -s --cacert /etc/kubernetes/cert/ca.pem -H "Authorization: Bearer 123456" https://~~192.168.100.246~~:10250/metrics

Unauthorized

通过认证后，kubelet 使用 SubjectAccessReview API 向 kube-apiserver 发送请求，查询证书或 token 对应的 user、group 是否有操作资源的权限(RBAC)；

###### 证书认证和授权

# 权限不足的证书

sudo curl -s --cacert /etc/kubernetes/cert/ca.pem --cert /etc/kubernetes/cert/kube-controller-manager.pem --key /etc/kubernetes/cert/kube-controller-manager-key.pem [https://](https://192.168.100.246:10250/metrics)~~[192.168.100.246](https://192.168.100.246:10250/metrics)~~[:10250/metrics](https://192.168.100.246:10250/metrics)

Forbidden (user=system:kube-controller-manager, verb=get, resource=nodes, subresource=metrics)

# 使用部署 kubectl 命令行工具时创建的、具有最高权限的 admin 证书；

sudo curl -s --cacert /etc/kubernetes/cert/ca.pem --cert /opt/k8s/work/cert/admin.pem --key /opt/k8s/work/cert/admin-key.pem https://~~192.168.100.246~~:10250/metrics|head

# HELP apiserver\_client\_certificate\_expiration\_seconds Distribution of the remaining lifetime on the certificate used to authenticate a request.

# TYPE apiserver\_client\_certificate\_expiration\_seconds histogram

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="0"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="21600"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="43200"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="86400"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="172800"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="345600"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="604800"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="2.592e+06"} 0

* --cacert、--cert、--key 的参数值必须是文件路径，如上面的 ./admin.pem 不能省略 ./，否则返回 401 Unauthorized；

###### bear token 认证和授权

创建一个 ServiceAccount，将它和 ClusterRole system:kubelet-api-admin 绑定，从而具有调用 kubelet API 的权限：

kubectl create sa kubelet-api-test

kubectl create clusterrolebinding kubelet-api-test --clusterrole=system:kubelet-api-admin --serviceaccount=default:kubelet-api-test

SECRET=$(kubectl get secrets | grep kubelet-api-test | awk '{print $1}')

TOKEN=$(kubectl describe secret ${SECRET} | grep -E '^token' | awk '{print $2}')

echo ${TOKEN}

curl -s --cacert /etc/kubernetes/cert/ca.pem -H "Authorization: Bearer ${TOKEN}" https://~~192.168.100.246~~:10250/metrics|head

# HELP apiserver\_client\_certificate\_expiration\_seconds Distribution of the remaining lifetime on the certificate used to authenticate a request.

# TYPE apiserver\_client\_certificate\_expiration\_seconds histogram

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="0"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="21600"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="43200"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="86400"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="172800"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="345600"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="604800"} 0

apiserver\_client\_certificate\_expiration\_seconds\_bucket{le="2.592e+06"} 0

###### cadvisor 和 metrics

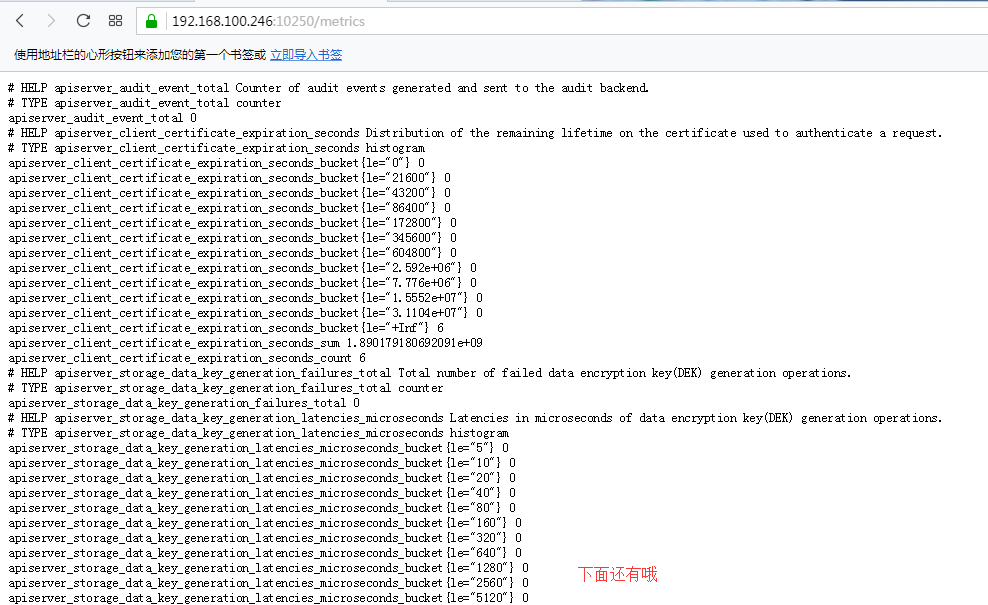
cadvisor 统计所在节点各容器的资源(CPU、内存、磁盘、网卡)使用情况，分别在自己的 http web 页面(4194 端口)和 10250 以 promehteus metrics 的形式输出。

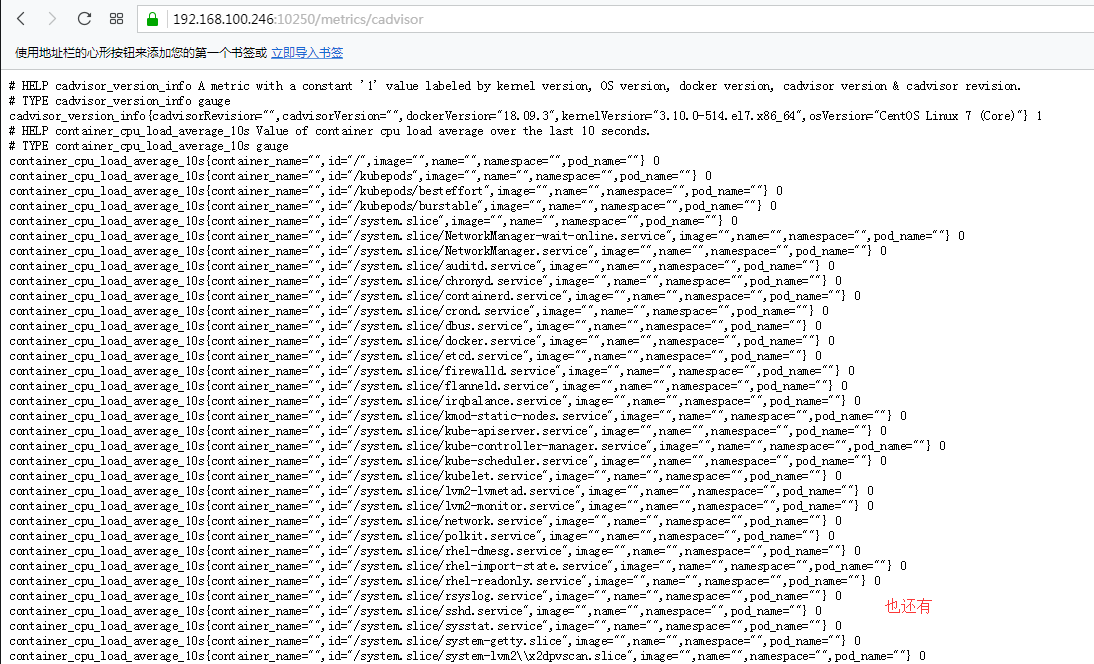
浏览器访问 ：

[https://](https://192.168.100.246:10250/metrics)~~[192.168.100.246](https://192.168.100.246:10250/metrics)~~[:10250/metrics](https://192.168.100.246:10250/metrics) 和

[https://](https://192.168.100.246:10250/metrics/cadvisor)~~[192.168.100.246](https://192.168.100.246:10250/metrics/cadvisor)~~[:10250/metrics/cadvisor](https://192.168.100.246:10250/metrics/cadvisor)

分别返回 kublet 和 metrics 和 cadvisor 。





注意：

* kublet.config.json 设置 authentication.anonymous.enabled 为 false，不允许匿名证书访问 10250 的 https 服务；
* 参考 [浏览器访问kube-apiserver安全端口](#_浏览器访问_kube-apiserver_安全端口)，创建和导入相关证书，然后访问上面的 10250 端口；

##### 获取 kublet 的配置

从 kube-apiserver 获取各 node 的配置：

# 使用部署 kubectl 命令行工具时创建的、具有最高权限的 admin 证书；

sudo curl -sSL --cacert /etc/kubernetes/cert/ca.pem --cert /opt/k8s/work/cert/admin.pem --key /opt/k8s/work/cert/admin-key.pem ${KUBE\_APISERVER}/api/v1/nodes/master/proxy/configz | jq '.kubeletconfig|.kind="KubeletConfiguration"|.apiVersion="kubelet.config.k8s.io/v1beta1"'

{

"syncFrequency": "1m0s",

"fileCheckFrequency": "20s",

"httpCheckFrequency": "20s",

"address": "~~192.168.100.246~~",

"port": 10250,

"rotateCertificates": true,

"serverTLSBootstrap": true,

"authentication": {

"x509": {

"clientCAFile": "/etc/kubernetes/cert/ca.pem"

},

"webhook": {

"enabled": true,

"cacheTTL": "2m0s"

},

"anonymous": {

"enabled": false

}

},

"authorization": {

"mode": "Webhook",

"webhook": {

"cacheAuthorizedTTL": "5m0s",

"cacheUnauthorizedTTL": "30s"

}

},

"registryPullQPS": 0,

"registryBurst": 10,

"eventRecordQPS": 0,

"eventBurst": 10,

"enableDebuggingHandlers": true,

"healthzPort": 10248,

"healthzBindAddress": "127.0.0.1",

"oomScoreAdj": -999,

"clusterDomain": "cluster.local",

"clusterDNS": [

"10.254.0.2"

],

"streamingConnectionIdleTimeout": "4h0m0s",

"nodeStatusUpdateFrequency": "10s",

"nodeLeaseDurationSeconds": 40,

"imageMinimumGCAge": "2m0s",

"imageGCHighThresholdPercent": 85,

"imageGCLowThresholdPercent": 80,

"volumeStatsAggPeriod": "1m0s",

"cgroupsPerQOS": true,

"cgroupDriver": "cgroupfs",

"cpuManagerPolicy": "none",

"cpuManagerReconcilePeriod": "10s",

"runtimeRequestTimeout": "15m0s",

"hairpinMode": "promiscuous-bridge",

"maxPods": 220,

"podCIDR": "172.30.0.0/16",

"podPidsLimit": -1,

"resolvConf": "/etc/resolv.conf",

"cpuCFSQuota": true,

"cpuCFSQuotaPeriod": "100ms",

"maxOpenFiles": 1000000,

"contentType": "application/vnd.kubernetes.protobuf",

"kubeAPIQPS": 1000,

"kubeAPIBurst": 2000,

"serializeImagePulls": false,

"evictionHard": {

"imagefs.available": "15%",

"memory.available": "100Mi",

"nodefs.available": "10%",

"nodefs.inodesFree": "5%"

},

"evictionPressureTransitionPeriod": "5m0s",

"enableControllerAttachDetach": true,

"makeIPTablesUtilChains": true,

"iptablesMasqueradeBit": 14,

"iptablesDropBit": 15,

"failSwapOn": true,

"containerLogMaxSize": "10Mi",

"containerLogMaxFiles": 5,

"configMapAndSecretChangeDetectionStrategy": "Watch",

"enforceNodeAllocatable": [

"pods"

],

"kind": "KubeletConfiguration",

"apiVersion": "kubelet.config.k8s.io/v1beta1"

}

或者参考代码中的注释：

<https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/apis/kubeletconfig/v1beta1/types.go>

##### 参考

kubelet 认证和授权：

<https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/>

#### 部署 kube-proxy 组件

kube-proxy 运行在所有 worker 节点上，，它监听 apiserver 中 service 和 Endpoint 的变化情况，创建路由规则来进行服务负载均衡。

本文档讲解部署 kube-proxy 的部署，使用 ipvs 模式。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

##### 下载和分发 kube-proxy 二进制文件

参考 [部署master节点](#_部署master节点_1)

##### 安装依赖包

各节点需要安装 ipvsadm 和 ipset 命令，加载 ip\_vs 内核模块。

参考 [部署worker节点](#_部署_worker_节点)

##### 创建 kube-proxy 证书

创建证书签名请求：

cd /opt/k8s/work/cert

cat > kube-proxy-csr.json <<EOF

{

"CN": "system:kube-proxy",

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* CN：指定该证书的 User 为 system:kube-proxy；
* 预定义的 RoleBinding system:node-proxier 将User system:kube-proxy 与 Role system:node-proxier 绑定，该 Role 授予了调用 kube-apiserver Proxy 相关 API 的权限；
* 该证书只会被 kube-proxy 当做 client 证书使用，所以 hosts 字段为空；

生成证书和私钥：

cd /opt/k8s/work/cert

cfssl gencert -ca=/opt/k8s/work/cert/ca.pem \

-ca-key=/opt/k8s/work/cert/ca-key.pem \

-config=/opt/k8s/work/cert/ca-config.json \

-profile=kubernetes kube-proxy-csr.json | cfssljson -bare kube-proxy

ls kube-proxy\*

##### 创建和分发 kubeconfig 文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

kubectl config set-cluster kubernetes \

--certificate-authority=/opt/k8s/work/cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=kube-proxy.kubeconfig

kubectl config set-credentials kube-proxy \

--client-certificate=/opt/k8s/work/cert/kube-proxy.pem \

--client-key=/opt/k8s/work/cert/kube-proxy-key.pem \

--embed-certs=true \

--kubeconfig=kube-proxy.kubeconfig

kubectl config set-context default \

--cluster=kubernetes \

--user=kube-proxy \

--kubeconfig=kube-proxy.kubeconfig

kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig

* --embed-certs=true：将 ca.pem 和 admin.pem 证书内容嵌入到生成的 kubectl-proxy.kubeconfig 文件中(不加时，写入的是证书文件路径)；

分发 kubeconfig 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_name in ${NODE\_NAMES[@]}

do

echo ">>> ${node\_name}"

scp kube-proxy.kubeconfig root@${node\_name}:/etc/kubernetes/

done

##### 创建 kube-proxy 配置文件

从 v1.10 开始，kube-proxy 部分参数可以配置文件中配置。可以使用 --write-config-to 选项生成该配置文件，或者参考 [kubeproxyconfig 的类型定义源文件](https://github.com/kubernetes/kubernetes/blob/master/pkg/proxy/apis/kubeproxyconfig/types.go)

创建 kube-proxy config 文件模板：

cd /opt/k8s/work

cat <<EOF | tee kube-proxy-config.yaml.template

kind: KubeProxyConfiguration

apiVersion: kubeproxy.config.k8s.io/v1alpha1

clientConnection:

kubeconfig: "/etc/kubernetes/kube-proxy.kubeconfig"

bindAddress: ##NODE\_IP##

clusterCIDR: ${CLUSTER\_CIDR}

healthzBindAddress: ##NODE\_IP##:10256

hostnameOverride: ##NODE\_NAME##

metricsBindAddress: ##NODE\_IP##:10249

mode: "ipvs"

EOF

* bindAddress: 监听地址；
* clientConnection.kubeconfig: 连接 apiserver 的 kubeconfig 文件；
* clusterCIDR: kube-proxy 根据 --cluster-cidr 判断集群内部和外部流量，指定 --cluster-cidr 或 --masquerade-all 选项后 kube-proxy 才会对访问 Service IP 的请求做 SNAT；
* hostnameOverride: 参数值必须与 kubelet 的值一致，否则 kube-proxy 启动后会找不到该 Node，从而不会创建任何 ipvs 规则；
* mode: 使用 ipvs 模式；

为各节点创建和分发 kube-proxy 配置文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for (( i=0; i < 3; i++ ))

do

echo ">>> ${NODE\_NAMES[i]}"

sed -e "s/##NODE\_NAME##/${NODE\_NAMES[i]}/" -e "s/##NODE\_IP##/${NODE\_IPS[i]}/" kube-proxy-config.yaml.template > kube-proxy-config-${NODE\_NAMES[i]}.yaml.template

scp kube-proxy-config-${NODE\_NAMES[i]}.yaml.template root@${NODE\_NAMES[i]}:/etc/kubernetes/kube-proxy-config.yaml

done

##### 创建和分发 kube-proxy systemd unit 文件

cd /opt/k8s/work

cat > kube-proxy.service <<EOF

[Unit]

Description=Kubernetes Kube-Proxy Server

Documentation=https://github.com/GoogleCloudPlatform/kubernetes

After=network.target

[Service]

WorkingDirectory=/data/k8s/k8s/kube-proxy

ExecStart=/opt/k8s/bin/kube-proxy \\

--config=/etc/kubernetes/kube-proxy-config.yaml \\

--logtostderr=true \\

--v=2

Restart=on-failure

RestartSec=5

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

EOF

分发 kube-proxy systemd unit 文件：

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_name in ${NODE\_NAMES[@]}

do

echo ">>> ${node\_name}"

scp kube-proxy.service root@${node\_name}:/usr/lib/systemd/system/

done

##### 启动 kube-proxy 服务

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "mkdir -p ${K8S\_DIR}/kube-proxy"

ssh root@${node\_ip} "systemctl daemon-reload && systemctl enable kube-proxy && systemctl restart kube-proxy"

done

* 必须先创建工作目录；

##### 检查启动结果

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "systemctl status kube-proxy|grep Active"

done

确保状态为 active (running)，否则查看日志，确认原因：

journalctl -u kube-proxy

##### 查看监听端口和 metrics

sudo netstat -lnpt|grep kube-prox

tcp 0 0 ~~192.168.100.246~~:10256 0.0.0.0:\* LISTEN 20482/kube-proxy

tcp 0 0 ~~192.168.100.246~~:10249 0.0.0.0:\* LISTEN 20482/kube-proxy

* 10249：http prometheus metrics port;
* 10256：http healthz port;

##### 查看 ipvs 路由规则

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh root@${node\_ip} "/usr/sbin/ipvsadm -ln"

done

预期输出：

>>> ~~192.168.100.246~~

IP Virtual Server version 1.2.1 (size=4096)

Prot LocalAddress:Port Scheduler Flags

-> RemoteAddress:Port Forward Weight ActiveConn InActConn

TCP 10.254.0.1:443 rr

-> ~~192.168.100.246~~:6443 Masq 1 0 0

-> ~~192.168.100.247~~:6443 Masq 1 0 0

-> ~~192.168.100.248~~:6443 Masq 1 0 0

>>> ~~192.168.100.247~~

IP Virtual Server version 1.2.1 (size=4096)

Prot LocalAddress:Port Scheduler Flags

-> RemoteAddress:Port Forward Weight ActiveConn InActConn

TCP 10.254.0.1:443 rr

-> ~~192.168.100.246~~:6443 Masq 1 0 0

-> ~~192.168.100.247~~:6443 Masq 1 0 0

-> ~~192.168.100.248~~:6443 Masq 1 0 0

>>> ~~192.168.100.248~~

IP Virtual Server version 1.2.1 (size=4096)

Prot LocalAddress:Port Scheduler Flags

-> RemoteAddress:Port Forward Weight ActiveConn InActConn

TCP 10.254.0.1:443 rr

-> ~~192.168.100.246~~:6443 Masq 1 0 0

-> ~~192.168.100.247~~:6443 Masq 1 0 0

-> ~~192.168.100.248~~:6443 Masq 1 0 0

可见将所有到 kubernetes cluster ip 443 端口的请求都转发到 kube-apiserver 的 6443 端口；

### 验证集群功能

本文档使用 daemonset 验证 master 和 worker 节点是否工作正常。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行，然后远程分发文件和执行命令。

#### 检查节点状态

kubectl get nodes

NAME STATUS ROLES AGE VERSION

master Ready <none> 4h35m v1.12.3

node1 Ready <none> 4h35m v1.12.3

node2 Ready <none> 4h35m v1.12.3

都为 Ready 时正常。

#### 创建测试文件

cd /opt/k8s/work

cat > nginx-ds.yml <<EOF

apiVersion: v1

kind: Service

metadata:

name: nginx-ds

labels:

app: nginx-ds

spec:

type: NodePort

selector:

app: nginx-ds

ports:

- name: http

port: 80

targetPort: 80

---

apiVersion: extensions/v1beta1

kind: DaemonSet

metadata:

name: nginx-ds

labels:

addonmanager.kubernetes.io/mode: Reconcile

spec:

template:

metadata:

labels:

app: nginx-ds

spec:

containers:

- name: my-nginx

image: nginx:1.7.9

ports:

- containerPort: 80

EOF

#### 执行定义文件

kubectl create -f nginx-ds.yml

#### 检查各 Node 上的 Pod IP 连通性

kubectl get pods -o wide|grep nginx-ds

nginx-ds-69hp5 1/1 Running 0 2m19s 172.30.8.5 node1 <none>

nginx-ds-rm744 1/1 Running 0 2m19s 172.30.48.7 master <none>

nginx-ds-vpzk2 1/1 Running 0 2m19s 172.30.144.6 node2 <none>

可见，nginx-ds 的 Pod IP 分别是 172.30.8.5、172.30.48.7、172.30.144.6，在所有 Node 上分别 ping 这三个 IP，看是否连通：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "ping -c 1 172.30.8.5"

ssh ${node\_ip} "ping -c 1 172.30.48.7"

ssh ${node\_ip} "ping -c 1 172.30.144.6"

done

#### 检查服务 IP 和端口可达性

kubectl get svc |grep nginx-ds

nginx-ds NodePort 10.254.150.98 <none> 80:30437/TCP 5m

可见：

* Service Cluster IP：10.254.150.98
* 服务端口：80
* NodePort 端口：30437

在所有 Node 上 curl Service IP：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "curl ~~10.254.150.98~~"

done

预期输出 nginx 欢迎页面内容。

#### 检查服务的 NodePort 可达性

在所有 Node 上执行：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "curl ${node\_ip}:~~30437~~"

done

预期输出 nginx 欢迎页面内容。

### 部署集群插件

插件是集群的附件组件，丰富和完善了集群的功能。下面章节我们来学习一些常用的插件：

[coredns](#_dns_插件)

[Dashboard](#_dashboard_插件)

[Heapster (influxdb、grafana)](#_heapster_插件)

[Metrics Server](#_metrics-server_插件)

[EFK (elasticsearch、fluentd、kibana)](#_ELK_插件)

#### dns 插件

特别重要：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "docker pull coredns/coredns:1.2.2 && docker tag coredns/coredns:1.2.2 k8s.gcr.io/coredns:1.2.2"

done

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行。

##### 修改配置文件

将下载的 kubernetes-server-linux-amd64.tar.gz 解压后，再解压其中的 kubernetes-src.tar.gz 文件。

coredns 对应的目录是：cluster/addons/dns。

cd /opt/k8s/work/kubernetes/cluster/addons/dns/coredns

cp coredns.yaml.base coredns.yaml

source /opt/k8s/bin/environment.sh

sed -i -e "s/\_\_PILLAR\_\_DNS\_\_DOMAIN\_\_/${CLUSTER\_DNS\_DOMAIN}/" -e "s/\_\_PILLAR\_\_DNS\_\_SERVER\_\_/${CLUSTER\_DNS\_SVC\_IP}/" coredns.yaml

##### 插件 coredns

kubectl create -f coredns.yaml

##### 检查 coredns 功能

kubectl get all -n kube-system

NAME READY STATUS RESTARTS AGE

pod/coredns-779ffd89bd-v75k9 0/1 ImagePullBackOff 0 4m54s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kube-dns ClusterIP 10.254.0.2 <none> 53/UDP,53/TCP 4m54s

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

deployment.apps/coredns 1 1 1 0 4m54s

NAME DESIRED CURRENT READY AGE

replicaset.apps/coredns-779ffd89bd 1 1 0 4m54sa

新建一个 Deployment

cd /opt/k8s/work

cat > my-nginx.yaml <<EOF

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: my-nginx

spec:

replicas: 2

template:

metadata:

labels:

run: my-nginx

spec:

containers:

- name: my-nginx

image: nginx:1.7.9

ports:

- containerPort: 80

EOF

kubectl create -f my-nginx.yam

Export 该 Deployment, 生成 my-nginx 服务：

kubectl expose deploy my-nginx

service "my-nginx" exposed

kubectl get services --all-namespaces |grep my-nginx

default my-nginx ClusterIP 10.254.206.78 <none> 80/TCP 7s

创建另一个 Pod，查看 /etc/resolv.conf 是否包含 kubelet 配置的 --cluster-dns 和 --cluster-domain，是否能够将服务 my-nginx 解析到上面显示的 Cluster IP 10.254.206.78

cd /opt/k8s/work

cat > dnsutils-ds.yml <<EOF

apiVersion: v1

kind: Service

metadata:

name: dnsutils-ds

labels:

app: dnsutils-ds

spec:

type: NodePort

selector:

app: dnsutils-ds

ports:

- name: http

port: 80

targetPort: 80

---

apiVersion: extensions/v1beta1

kind: DaemonSet

metadata:

name: dnsutils-ds

labels:

addonmanager.kubernetes.io/mode: Reconcile

spec:

template:

metadata:

labels:

app: dnsutils-ds

spec:

containers:

- name: my-dnsutils

image: tutum/dnsutils:latest

command:

- sleep

- "3600"

ports:

- containerPort: 80

EOF

kubectl create -f dnsutils-ds.yml

查看 pod 节点：

kubectl get pods

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup kubernetes

Server: 10.254.0.2

Address: 10.254.0.2#53

Name: kubernetes.default.svc.cluster.local

Address: 10.254.0.1

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup www.baidu.com # 解析外部域名时，需要以 . 结尾

Server: 10.254.0.2

Address: 10.254.0.2#53

Non-authoritative answer:

\*\*\* Can't find www.baidu.com: No answer

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup www.baidu.com.

Server: 10.254.0.2

Address: 10.254.0.2#53

Non-authoritative answer:

www.baidu.com canonical name = www.a.shifen.com.

Name: www.a.shifen.com

Address: 61.135.169.125

Name: www.a.shifen.com

Address: 61.135.169.121

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup my-nginx

Server: 10.254.0.2

Address: 10.254.0.2#53

Name: my-nginx.default.svc.cluster.local

Address: 10.254.229.163

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup kube-dns.kube-system.svc.cluster

Server: 10.254.0.2

Address: 10.254.0.2#53

Non-authoritative answer:

\*\*\* Can't find kube-dns.kube-system.svc.cluster: No answer

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup kube-dns.kube-system.svc

Server: 10.254.0.2

Address: 10.254.0.2#53

Name: kube-dns.kube-system.svc.cluster.local

Address: 10.254.0.2

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup kube-dns.kube-system.svc.cluster.local

Server: 10.254.0.2

Address: 10.254.0.2#53

Non-authoritative answer:

\*\*\* Can't find kube-dns.kube-system.svc.cluster.local: No answer

kubectl exec ~~dnsutils-ds-9rm5v~~ nslookup kube-dns.kube-system.svc.cluster.local.

Server: 10.254.0.2

Address: 10.254.0.2#53

Name: kube-dns.kube-system.svc.cluster.local

Address: 10.254.0.2

#### dashboard 插件

特别重要：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "docker pull fishchen/kubernetes-dashboard-amd64:v1.8.3 && docker tag fishchen/kubernetes-dashboard-amd64:v1.8.3 k8s.gcr.io/kubernetes-dashboard-amd64:v1.8.3"

done

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行。

修改配置文件

将下载的 kubernetes-server-linux-amd64.tar.gz 解压后，再解压其中的 kubernetes-src.tar.gz 文件。

dashboard 对应的目录是：cluster/addons/dashboard。

cd /opt/k8s/work/kubernetes/cluster/addons/dashboard

##### 修改配置文件

cat dashboard-service.yaml

apiVersion: v1

kind: Service

metadata:

name: kubernetes-dashboard

namespace: kube-system

labels:

k8s-app: kubernetes-dashboard

kubernetes.io/cluster-service: "true"

addonmanager.kubernetes.io/mode: Reconcile

spec:

type: NodePort # 增加这一行

selector:

k8s-app: kubernetes-dashboard

ports:

- port: 443

targetPort: 8443

* 指定端口类型为 NodePort，这样外界可以通过地址 nodeIP:nodePort 访问 dashboard；

##### 执行所有定义文件

$ ls \*.yaml

dashboard-configmap.yaml dashboard-controller.yaml dashboard-rbac.yaml dashboard-secret.yaml dashboard-service.yaml

$ kubectl create -f .

##### 查看分配的 NodePort

$ kubectl get deployment kubernetes-dashboard -n kube-system

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

kubernetes-dashboard 1 1 1 1 2m

$ kubectl --namespace kube-system get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

coredns-77cd44d8df-4fmfc 1/1 Running 0 1h 172.30.200.2 node2 <none>

kubernetes-dashboard-69db8c7745-jtvpj 1/1 Running 0 55s 172.30.112.3 master <none>

$ kubectl get services kubernetes-dashboard -n kube-system

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes-dashboard NodePort 10.254.9.198 <none> 443:32014/TCP 1m

* NodePort 32014 映射到 dashboard pod 443 端口；

dashboard 的 --authentication-mode 支持 token、basic，默认为 token。如果使用 basic，则 kube-apiserver 必须配置 '--authorization-mode=ABAC' 和 '--basic-auth-file' 参数。

##### 查看 dashboard 支持的命令行参数

kubectl exec --namespace kube-system -it ~~kubernetes-dashboard-659798bd99-x5j6d~~ -- /dashboard --help # kubernetes-dashboard-659798bd99-x5j6d 为 pod 名称

2019/04/01 13:12:03 Starting overwatch

Usage of /dashboard:

--alsologtostderr log to standard error as well as files

--apiserver-host string The address of the Kubernetes Apiserver to connect to in the format of protocol://address:port, e.g., http://localhost:8080. If not specified, the assumption is that the binary runs inside a Kubernetes cluster and local discovery is attempted.

--authentication-mode stringSlice Enables authentication options that will be reflected on login screen. Supported values: token, basic. Default: token.Note that basic option should only be used if apiserver has '--authorization-mode=ABAC' and '--basic-auth-file' flags set. (default [token])

--auto-generate-certificates When set to true, Dashboard will automatically generate certificates used to serve HTTPS. Default: false.

--bind-address ip The IP address on which to serve the --secure-port (set to 0.0.0.0 for all interfaces). (default 0.0.0.0)

--default-cert-dir string Directory path containing '--tls-cert-file' and '--tls-key-file' files. Used also when auto-generating certificates flag is set. (default "/certs")

--disable-settings-authorizer When enabled, Dashboard settings page will not require user to be logged in and authorized to access settings page.

--enable-insecure-login When enabled, Dashboard login view will also be shown when Dashboard is not served over HTTPS. Default: false.

--heapster-host string The address of the Heapster Apiserver to connect to in the format of protocol://address:port, e.g., http://localhost:8082. If not specified, the assumption is that the binary runs inside a Kubernetes cluster and service proxy will be used.

--insecure-bind-address ip The IP address on which to serve the --port (set to 0.0.0.0 for all interfaces). (default 127.0.0.1)

--insecure-port int The port to listen to for incoming HTTP requests. (default 9090)

--kubeconfig string Path to kubeconfig file with authorization and master location information.

--log\_backtrace\_at traceLocation when logging hits line file:N, emit a stack trace (default :0)

--log\_dir string If non-empty, write log files in this directory

--logtostderr log to standard error instead of files

--metric-client-check-period int Time in seconds that defines how often configured metric client health check should be run. Default: 30 seconds. (default 30)

--port int The secure port to listen to for incoming HTTPS requests. (default 8443)

--stderrthreshold severity logs at or above this threshold go to stderr (default 2)

--system-banner string When non-empty displays message to Dashboard users. Accepts simple HTML tags. Default: ''.

--system-banner-severity string Severity of system banner. Should be one of 'INFO|WARNING|ERROR'. Default: 'INFO'. (default "INFO")

--tls-cert-file string File containing the default x509 Certificate for HTTPS.

--tls-key-file string File containing the default x509 private key matching --tls-cert-file.

--token-ttl int Expiration time (in seconds) of JWE tokens generated by dashboard. Default: 15 min. 0 - never expires (default 900)

-v, --v Level log level for V logs

--vmodule moduleSpec comma-separated list of pattern=N settings for file-filtered logging

command terminated with exit code 2

##### 访问 dashboard

为了集群安全，从 1.7 开始，dashboard 只允许通过 https 访问，如果使用 kube proxy 则必须监听 localhost 或 127.0.0.1，对于 NodePort 没有这个限制，但是仅建议在开发环境中使用。

对于不满足这些条件的登录访问，在登录成功后浏览器不跳转，始终停在登录界面。

参考： https://github.com/kubernetes/dashboard/wiki/Accessing-Dashboard---1.7.X-and-above <https://github.com/kubernetes/dashboard/issues/2540>

1. kubernetes-dashboard 服务暴露了 NodePort，可以使用 https://NodeIP:NodePort 地址访问 dashboard；
2. 通过 kube-apiserver 访问 dashboard；
3. 通过 kubectl proxy 访问 dashboard：

如果使用了 VirtualBox，需要启用 VirtualBox 的 ForworadPort 功能将虚机监听的端口和 Host 的本地端口绑定。

###### 通过 kubectl proxy 访问 dashboard

启动代理：

kubectl proxy --address='localhost' --port=8086 --accept-hosts='^\*$'

Starting to serve on 127.0.0.1:8086

* --address 必须为 localhost 或 127.0.0.1；
* 需要指定 --accept-hosts 选项，否则浏览器访问 dashboard 页面时提示 “Unauthorized”；

浏览器访问 URL：

<http://127.0.0.1:8086/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy>

###### 通过 kube-apiserver 访问 dashboard

获取集群服务地址列表：

kubectl cluster-info

Kubernetes master is running at https://~~192.168.100.246~~:8443

CoreDNS is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

kubernetes-dashboard is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

* 由于 apiserver 通过本地的 kube-nginx 做了代理，所以上面显示的 127.0.0.1:8443 为本地的 kube-nginx 的 IP 和 Port，浏览器访问时需要替换为 kube-apiserver 实际监听的 IP 和端口，如 ~~192.168.100.246~~:6443；
* 必须通过 kube-apiserver 的安全端口(https)访问 dashbaord，访问时浏览器需要使用自定义证书，否则会被 kube-apiserver 拒绝访问。
* 创建和导入自定义证书的步骤，参考：[浏览器访问kube-apiserver安全端口](#_浏览器访问_kube-apiserver_安全端口)

浏览器访问 URL：

[https://](https://172.27.128.149:6443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy)~~[192.168.100.246](https://172.27.128.149:6443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy)~~[:6443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy](https://172.27.128.149:6443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy) 对于 virtuabox 做了端口映射：

<http://127.0.0.1:6443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/>



##### 创建登录 dashboard 的 token 和 kubeconfig 配置文件

上面提到，Dashboard 默认只支持 token 认证，所以如果使用 KubeConfig 文件，需要在该文件中指定 token，不支持使用 client 证书认证。

##### 创建登录 token

kubectl create sa dashboard-admin -n kube-system

kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin --serviceaccount=kube-system:dashboard-admin

ADMIN\_SECRET=$(kubectl get secrets -n kube-system | grep dashboard-admin | awk '{print $1}')

DASHBOARD\_LOGIN\_TOKEN=$(kubectl describe secret -n kube-system ${ADMIN\_SECRET} | grep -E '^token' | awk '{print $2}')

echo ${DASHBOARD\_LOGIN\_TOKEN}

eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9..CKm0cnBGmKvW19TdLWprxHqSAMYClIDRxfvMagA4\_J\_ogi2QGXQGWVcd9n0rV\_QL\_s98E1I3A2399aK-FJTBAw5ZyrwiyNVYYa8aSS3ROI7zyKr1xcubGCgvdKGv1WETTPjH6xcYGXhtdR0P6MReuqft6wX0ZucjmlaRtMjpEba\_N4YmLG\_YOI-qjziAqgE1vv9NU6DjWM8Enyh6cN6CzeI3qtEkt25PPFfE3BsGVMbpnkGyBgcJC-HKd\_VafAxSYaG1xICxgDGPWc5PiMIq7sQSFbX4wZAnBIozFG8IBtzgxrFhSdaJS\_MJROm5gYZf5mRezAJf-QRovX-e2xcfkQ

使用输出的 token 登录 Dashboard。

##### 创建使用 token 的 KubeConfig 文件

cd /opt/k8s/work

source /opt/k8s/bin/environment.sh

# 设置集群参数

kubectl config set-cluster kubernetes \

--certificate-authority=/etc/kubernetes/cert/ca.pem \

--embed-certs=true \

--server=${KUBE\_APISERVER} \

--kubeconfig=dashboard.kubeconfig

# 设置客户端认证参数，使用上面创建的 Token

kubectl config set-credentials dashboard\_user \

--token=${DASHBOARD\_LOGIN\_TOKEN} \

--kubeconfig=dashboard.kubeconfig

# 设置上下文参数

kubectl config set-context default \

--cluster=kubernetes \

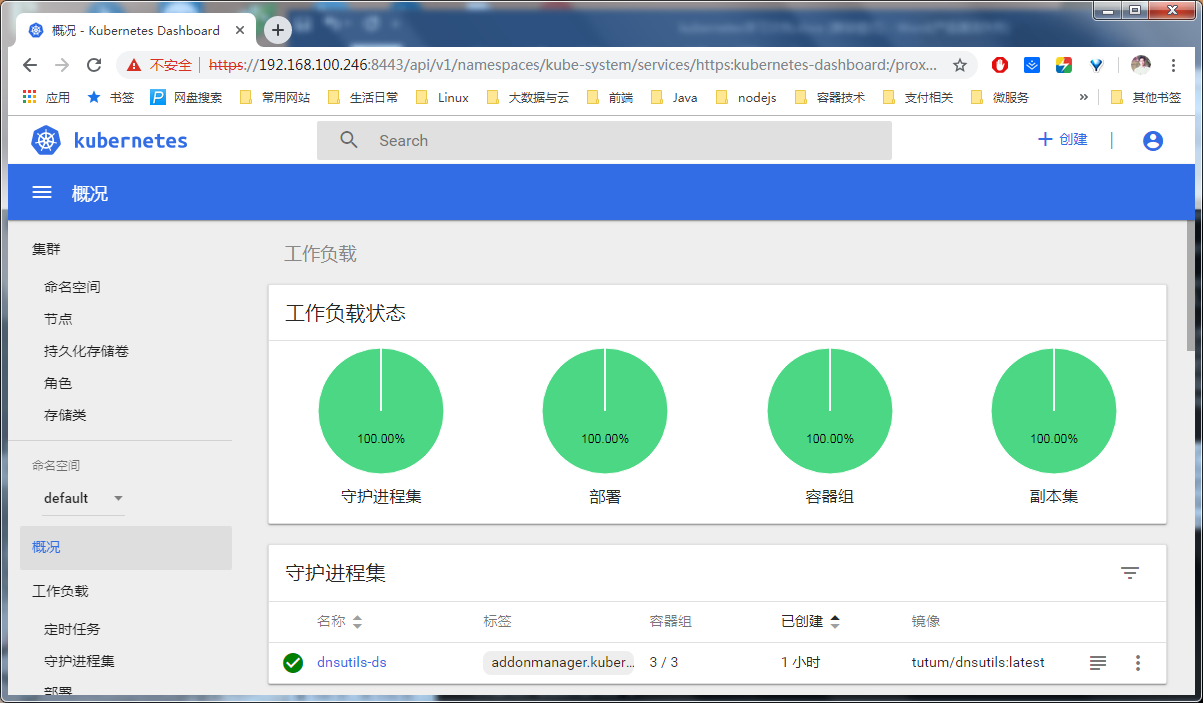
--user=dashboard\_user \

--kubeconfig=dashboard.kubeconfig

# 设置默认上下文

kubectl config use-context default --kubeconfig=dashboard.kubeconfig

用生成的 dashboard.kubeconfig 登录 Dashboard。



由于缺少 Heapster 插件，当前 dashboard 不能展示 Pod、Nodes 的 CPU、内存等统计数据和图表；

#### heapster 插件

特别重要：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

ssh ${node\_ip} "docker pull fishchen/heapster-grafana-amd64:v4.4.3 && docker tag fishchen/heapster-grafana-amd64:v4.4.3 gcr.io/google\_containers/heapster-grafana-amd64:v4.4.3"

ssh ${node\_ip} "docker pull fishchen/heapster-amd64:v1.5.3 && docker tag fishchen/heapster-amd64:v1.5.3 gcr.io/google\_containers/heapster-amd64:v1.5.3"

ssh ${node\_ip} "docker pull fishchen/heapster-influxdb-amd64:v1.3.3 && docker tag fishchen/heapster-influxdb-amd64:v1.3.3 gcr.io/google\_containers/heapster-influxdb-amd64:v1.3.3"

done

Heapster是一个收集者，将每个Node上的cAdvisor的数据进行汇总，然后导到第三方工具(如InfluxDB)。

Heapster 是通过调用 kubelet 的 http API 来获取 cAdvisor 的 metrics 数据的。

由于 kublet 只在 10250 端口接收 https 请求，故需要修改 heapster 的 deployment 配置。同时，需要赋予 kube-system:heapster ServiceAccount 调用 kubelet API 的权限。

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行。

##### 下载 heapster 文件

到 [heapster release](https://github.com/kubernetes/heapster/releases) 页面 下载最新版本的 heapster

cd /opt/k8s/work

wget https://github.com/kubernetes/heapster/archive/v1.5.4.tar.gz

tar -xzvf v1.5.4.tar.gz

mv v1.5.4.tar.gz heapster-1.5.4.tar.gz

官方文件目录： heapster-1.5.4/deploy/kube-config/influxdb

##### 修改配置

cd heapster-1.5.4/deploy/kube-config/influxdb

cp grafana.yaml{,.orig}

diff grafana.yaml.orig grafana.yaml

67c67

< # type: NodePort

---

> type: NodePort

* 开启 NodePort；

cp heapster.yaml{,.orig}

$ diff heapster.yaml.orig heapster.yaml

< - --source=kubernetes:https://kubernetes.default

---

> - --source=kubernetes:https://kubernetes.default?kubeletHttps=true&kubeletPort=10250

* 由于 kubelet 只在 10250 监听 https 请求，故添加相关参数；

##### 执行所有定义文件

$ cd /opt/k8s/work/heapster-1.5.4/deploy/kube-config/influxdb

$ ls \*.yaml

grafana.yaml heapster.yaml influxdb.yaml

$ kubectl create -f .

$ cd ../rbac/

$ cp heapster-rbac.yaml{,.orig}

$ diff heapster-rbac.yaml.orig heapster-rbac.yaml

12a13,26

> ---

> kind: ClusterRoleBinding

> apiVersion: rbac.authorization.k8s.io/v1beta1

> metadata:

> name: heapster-kubelet-api

> roleRef:

> apiGroup: rbac.authorization.k8s.io

> kind: ClusterRole

> name: system:kubelet-api-admin

> subjects:

> - kind: ServiceAccount

> name: heapster

> namespace: kube-system

$ kubectl create -f heapster-rbac.yaml

* 将 serviceAccount kube-system:heapster 与 ClusterRole system:kubelet-api-admin 绑定，授予它调用 kubelet API 的权限；

如果不修改，默认的 ClusterRole system:heapster 权限不足：

E1128 10:00:05.010716 1 manager.go:101] Error in scraping containers from kubelet:~~192.168.100.246~~:10250: failed to get all container stats from Kubelet URL "[https://](https://172.27.128.150:10250/stats/container/)~~[192.168.100.246](https://172.27.128.150:10250/stats/container/)~~[:10250/stats/container/](https://172.27.128.150:10250/stats/container/)": request failed - "403 Forbidden", response: "Forbidden (user=system:serviceaccount:kube-system:heapster, verb=create, resource=nodes, subresource=stats)" E1128 10:00:05.018556 1 manager.go:101] Error in scraping containers from kubelet:172.27.128.149:10250: failed to get all container stats from Kubelet URL "https:// ~~192.168.100.247~~:10250/stats/container/": request failed - "403 Forbidden", response: "Forbidden (user=system:serviceaccount:kube-system:heapster, verb=create, resource=nodes, subresource=stats)" E1128 10:00:05.022664 1 manager.go:101] Error in scraping containers from kubelet:172.27.128.148:10250: failed to get all container stats from Kubelet URL "https:// 192.168.100.248:10250/stats/container/": request failed - "403 Forbidden", response: "Forbidden (user=system:serviceaccount:kube-system:heapster, verb=create, resource=nodes, subresource=stats)" W1128 10:00:25.000467 1 manager.go:152] Failed to get all responses in time (got 0/3)

##### 检查执行结果

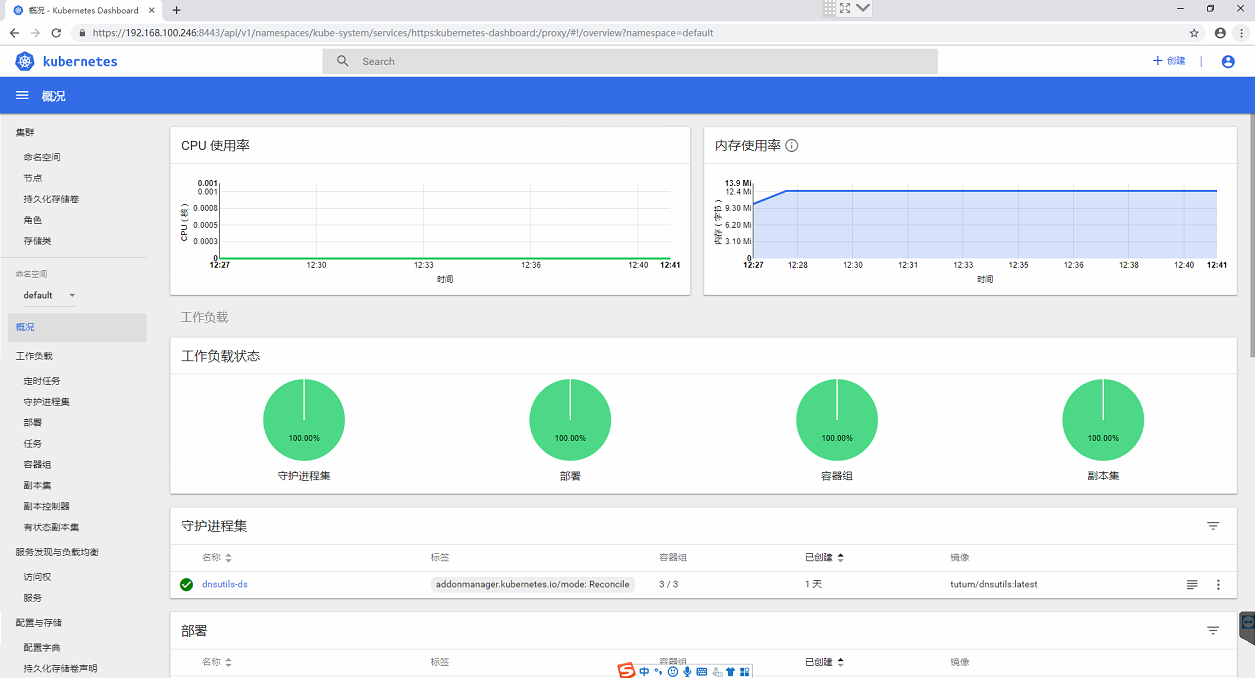
$ kubectl get pods -n kube-system | grep -E 'heapster|monitoring'

heapster-56c9dc749-j7hvz 1/1 Running 0 1m

monitoring-grafana-c797777db-lnwnc 1/1 Running 0 1m

monitoring-influxdb-cf9d95766-5wd28 1/1 Running 0 1m

检查 kubernets dashboard 界面，可以正确显示各 Nodes、Pods 的 CPU、内存、负载等统计数据和图表：



##### 访问 grafana

1. 通过 kube-apiserver 访问：

获取 monitoring-grafana 服务 URL：

kubectl cluster-info

Kubernetes master is running at https://~~192.168.100.246~~:8443

Heapster is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/heapster/proxy

CoreDNS is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

kubernetes-dashboard is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy

monitoring-grafana is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy

monitoring-influxdb is running at https://~~192.168.100.246~~:8443/api/v1/namespaces/kube-system/services/monitoring-influxdb/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

浏览器访问 URL：

[https://](https://172.27.128.252:8443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy)~~[192.168.100.246](https://172.27.128.252:8443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy)~~[:8443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy](https://172.27.128.252:8443/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy)

对于 virtuabox 做了端口映射：

<http://127.0.0.1:8080/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy>

1. 通过 kubectl proxy 访问：

创建代理

kubectl proxy --address='~~192.168.100.246~~' --port=8086 --accept-hosts='^\*$'

Starting to serve on 172.27.129.150:8086

浏览器访问 URL：

[http://](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy/?orgId=1)~~[192.168.100.246](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy/?orgId=1)~~[:8086/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy/?orgId=1](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy/?orgId=1)

对于 virtuabox 做了端口映射：

<http://127.0.0.1:8086/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy/?orgId=1>

1. 通过 NodePort 访问：

kubectl get svc -n kube-system|grep -E 'monitoring|heapster'

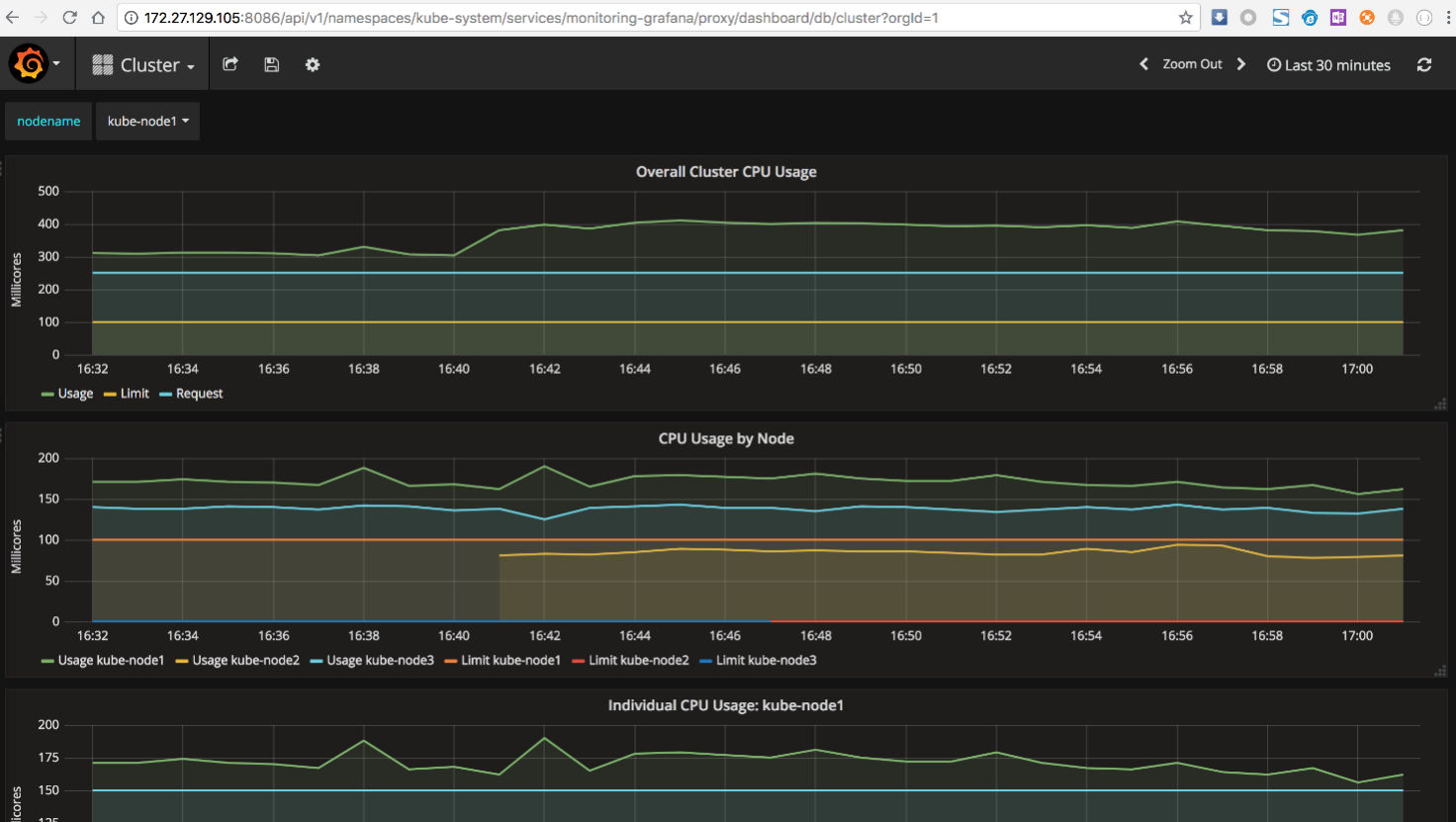
heapster ClusterIP 10.254.199.65 <none> 80/TCP 3m

monitoring-grafana NodePort 10.254.116.161 <none> 80:31470/TCP 3m

monitoring-influxdb ClusterIP 10.254.250.185 <none> 8086/TCP 3m

* grafana 监听 NodePort 31470；

浏览器访问 URL：http://~~192.168.100.246~~:31470/?orgId=1



##### 参考

1. 配置 heapster：<https://github.com/kubernetes/heapster/blob/master/docs/source-configuration.md>

#### metrics-server 插件

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行。

##### 创建 metrics-server 使用的证书

cd /opt/k8s/work/cert

cat > metrics-server-csr.json <<EOF

{

"CN": "aggregator",

"hosts": [],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* 注意： CN 名称为 aggregator，需要与 kube-apiserver 的 --requestheader-allowed-names 参数配置一致；

生成 metrics-server 证书和私钥：

cfssl gencert -ca=/etc/kubernetes/cert/ca.pem \

-ca-key=/etc/kubernetes/cert/ca-key.pem \

-config=/etc/kubernetes/cert/ca-config.json \

-profile=kubernetes metrics-server-csr.json | cfssljson -bare metrics-server

将生成的证书和私钥文件拷贝到 kube-apiserver 节点：

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]}

do

echo ">>> ${node\_ip}"

scp metrics-server\*.pem root@${node\_ip}:/etc/kubernetes/cert/

done

##### 修改 kubernetes 控制平面组件的配置以支持 metrics-server

###### kube-apiserver

添加如下配置参数：

--requestheader-client-ca-file=/etc/kubernetes/cert/ca.pem

--requestheader-allowed-names=""

--requestheader-extra-headers-prefix="X-Remote-Extra-"

--requestheader-group-headers=X-Remote-Group

--requestheader-username-headers=X-Remote-User

--proxy-client-cert-file=/etc/kubernetes/cert/metrics-server.pem

--proxy-client-key-file=/etc/kubernetes/cert/metrics-server-key.pem

--runtime-config=api/all=true

* --requestheader-XXX、--proxy-client-XXX 是 kube-apiserver 的 aggregator layer 相关的配置参数，metrics-server & HPA 需要使用；
* --requestheader-client-ca-file：用于签名 --proxy-client-cert-file 和 --proxy-client-key-file 指定的证书；在启用了 metric aggregator 时使用；
* 如果 --requestheader-allowed-names 不为空，则--proxy-client-cert-file 证书的 CN 必须位于 allowed-names 中，默认为 aggregator;

如果 kube-apiserver 机器没有运行 kube-proxy，则还需要添加 --enable-aggregator-routing=true 参数；

关于 --requestheader-XXX 相关参数，参考：

<https://github.com/kubernetes-incubator/apiserver-builder/blob/master/docs/concepts/auth.md>

<https://docs.bitnami.com/kubernetes/how-to/configure-autoscaling-custom-metrics/>

注意：requestheader-client-ca-file 指定的 CA 证书，必须具有 client auth and server auth；

###### kube-controllr-manager

添加如下配置参数(从 v1.12 开始，该选项默认为 true，不需要再添加)：

--horizontal-pod-autoscaler-use-rest-clients=true

用于配置 HPA 控制器使用 REST 客户端获取 metrics 数据。

##### 整体架构

暂缺

##### 修改插件配置文件配置文件

metrics-server 插件位于 kubernetes 的 cluster/addons/metrics-server/ 目录下。

修改 metrics-server-deployment 文件：

$ cp metrics-server-deployment.yaml{,.orig}

$ diff metrics-server-deployment.yaml.orig metrics-server-deployment.yaml

51c51

< image: mirrorgooglecontainers/metrics-server-amd64:v0.2.1

---

> image: k8s.gcr.io/metrics-server-amd64:v0.2.1

54c54

< - --source=kubernetes.summary\_api:''

---

> - --source=kubernetes.summary\_api:https://kubernetes.default?kubeletHttps=true&kubeletPort=10250

60c60

< image: siriuszg/addon-resizer:1.8.1

---

> image: k8s.gcr.io/addon-resizer:1.8.1

* metrics-server 的参数格式与 heapster 类似。由于 kubelet 只在 10250 监听 https 请求，故添加相关参数；

授予 kube-system:metrics-server ServiceAccount 访问 kubelet API 的权限：

$ cat auth-kubelet.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: metrics-server:system:kubelet-api-admin

labels:

kubernetes.io/cluster-service: "true"

addonmanager.kubernetes.io/mode: Reconcile

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: system:kubelet-api-admin

subjects:

- kind: ServiceAccount

name: metrics-server

namespace: kube-system

* 新建一个 ClusterRoleBindings 定义文件，授予相关权限；

##### 创建 metrics-server

$ cd /opt/k8s/kubernetes/cluster/addons/metrics-server

$ ls -l \*.yaml

-rw-rw-r-- 1 k8s k8s 398 Jun 5 07:17 auth-delegator.yaml

-rw-rw-r-- 1 k8s k8s 404 Jun 16 18:02 auth-kubelet.yaml

-rw-rw-r-- 1 k8s k8s 419 Jun 5 07:17 auth-reader.yaml

-rw-rw-r-- 1 k8s k8s 393 Jun 5 07:17 metrics-apiservice.yaml

-rw-rw-r-- 1 k8s k8s 2640 Jun 16 17:54 metrics-server-deployment.yaml

-rw-rw-r-- 1 k8s k8s 336 Jun 5 07:17 metrics-server-service.yaml

-rw-rw-r-- 1 k8s k8s 801 Jun 5 07:17 resource-reader.yaml

$ kubectl create -f .

##### 查看运行情况

$ kubectl get pods -n kube-system |grep metrics-server

metrics-server-v0.2.1-7486f5bd67-v95q2 2/2 Running 0 45s

$ kubectl get svc -n kube-system|grep metrics-server

metrics-server ClusterIP 10.254.115.120 <none> 443/TCP 1m

##### 查看 metrcs-server 输出的 metrics

metrics-server 输出的 APIs：

<https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/resource-metrics-api.md>

1. 通过 kube-apiserver 或 kubectl proxy 访问：

https://~~192.168.100.246~~:6443/apis/metrics.k8s.io/v1beta1/nodes https://~~192.168.100.246~~:6443/apis/metrics.k8s.io/v1beta1/nodes/ https://~~192.168.100.246~~:6443/apis/metrics.k8s.io/v1beta1/pods https://~~192.168.100.246~~:6443/apis/metrics.k8s.io/v1beta1/namespace//pods/

1. 直接使用 kubectl 命令访问：

kubectl get --raw apis/metrics.k8s.io/v1beta1/nodes kubectl get --raw apis/metrics.k8s.io/v1beta1/pods kubectl get --raw apis/metrics.k8s.io/v1beta1/nodes/ kubectl get --raw apis/metrics.k8s.io/v1beta1/namespace//pods/

$ kubectl get --raw "/apis/metrics.k8s.io/v1beta1" | jq .

{

"kind": "APIResourceList",

"apiVersion": "v1",

"groupVersion": "metrics.k8s.io/v1beta1",

"resources": [

{

"name": "nodes",

"singularName": "",

"namespaced": false,

"kind": "NodeMetrics",

"verbs": [

"get",

"list"

]

},

{

"name": "pods",

"singularName": "",

"namespaced": true,

"kind": "PodMetrics",

"verbs": [

"get",

"list"

]

}

]

}

$ kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .

{

"kind": "NodeMetricsList",

"apiVersion": "metrics.k8s.io/v1beta1",

"metadata": {

"selfLink": "/apis/metrics.k8s.io/v1beta1/nodes"

},

"items": [

{

"metadata": {

"name": "node2",

"selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/node2",

"creationTimestamp": "2018-06-16T10:24:03Z"

},

"timestamp": "2018-06-16T10:23:00Z",

"window": "1m0s",

"usage": {

"cpu": "133m",

"memory": "1115728Ki"

}

},

{

"metadata": {

"name": "master",

"selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/master",

"creationTimestamp": "2018-06-16T10:24:03Z"

},

"timestamp": "2018-06-16T10:23:00Z",

"window": "1m0s",

"usage": {

"cpu": "221m",

"memory": "6799908Ki"

}

},

{

"metadata": {

"name": "node1",

"selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/node1",

"creationTimestamp": "2018-06-16T10:24:03Z"

},

"timestamp": "2018-06-16T10:23:00Z",

"window": "1m0s",

"usage": {

"cpu": "76m",

"memory": "1130180Ki"

}

}

]

}

* /apis/metrics.k8s.io/v1beta1/nodes 和 /apis/metrics.k8s.io/v1beta1/pods 返回的 usage 包含 CPU 和 Memory；

#### EFK 插件

EFK 对应的目录：kubernetes/cluster/addons/fluentd-elasticsearch

$ cd /opt/k8s/work/kubernetes/cluster/addons/fluentd-elasticsearch

$ ls \*.yaml

es-service.yaml es-statefulset.yaml fluentd-es-configmap.yaml fluentd-es-ds.yaml kibana-deployment.yaml kibana-service.yaml

注意：如果没有特殊指明，本文档的所有操作均在 master 节点上执行。

##### 拉取镜像

注意：因为elk插件使用的是谷歌镜像，由于众所周知的原因我们无法使用，只能通过先拉去对于的其他公开镜像然后改名。或者直接修改yml文件中定义的谷歌镜像名（推荐使用第一种）。

source /opt/k8s/bin/environment.sh

for node\_ip in ${NODE\_IPS[@]};

do

echo ">>> ${node\_ip}";

ssh ${node\_ip} "docker pull xiaochunping/elasticsearch:v6.2.5 && docker tag xiaochunping/elasticsearch:v6.2.5 k8s.gcr.io/elasticsearch:v6.2.5";

ssh ${node\_ip} "docker pull xiaochunping/fluentd-elasticsearch:v2.2.0 && docker tag xiaochunping/fluentd-elasticsearch:v2.2.0 k8s.gcr.io/fluentd-elasticsearch:v2.2.0";

done

命令只需要在master节点上执行即可。

##### 给 Node 设置标签

DaemonSet fluentd-es 只会调度到设置了标签 beta.kubernetes.io/fluentd-ds-ready=true 的 Node，需要在期望运行 fluentd 的 Node 上设置该标签；

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

master Ready <none> 3d v1.10.4

node1 Ready <none> 3d v1.10.4

node2 Ready <none> 3d v1.10.4

$ kubectl label nodes node2 beta.kubernetes.io/fluentd-ds-ready=true

node "node2" labeled

##### 执行定义文件

$ cd /opt/k8s/kubernetes/cluster/addons/fluentd-elasticsearch

$ ls \*.yaml

es-service.yaml es-statefulset.yaml fluentd-es-configmap.yaml fluentd-es-ds.yaml kibana-deployment.yaml kibana-service.yaml

$ kubectl create -f .

##### 检查执行结果

$ kubectl get pods -n kube-system -o wide|grep -E 'elasticsearch|fluentd|kibana'

elasticsearch-logging-0 1/1 Running 0 5m 172.30.81.7 master

elasticsearch-logging-1 1/1 Running 0 2m 172.30.39.8 node2

fluentd-es-v2.0.4-hntfp 1/1 Running 0 5m 172.30.39.6 node2

kibana-logging-7445dc9757-pvpcv 1/1 Running 0 5m 172.30.39.7 node2

$ kubectl get service -n kube-system|grep -E 'elasticsearch|kibana'

elasticsearch-logging ClusterIP 10.254.50.198 <none> 9200/TCP 5m

kibana-logging ClusterIP 10.254.255.190 <none> 5601/TCP 5m

kibana Pod 第一次启动时会用\*\*较长时间(0-20分钟)\*\*来优化和 Cache 状态页面，可以 tailf 该 Pod 的日志观察进度：

$ kubectl logs kibana-logging-7445dc9757-pvpcv -n kube-system -f

{"type":"log","@timestamp":"2018-06-16T11:36:18Z","tags":["info","optimize"],"pid":1,"message":"Optimizing and caching bundles for graph, ml, kibana, stateSessionStorageRedirect, timelion and status\_page. This may take a few minutes"}

{"type":"log","@timestamp":"2018-06-16T11:40:03Z","tags":["info","optimize"],"pid":1,"message":"Optimization of bundles for graph, ml, kibana, stateSessionStorageRedirect, timelion and status\_page complete in 224.57 seconds"}

注意：只有当的 Kibana pod 启动完成后，才能查看 kibana dashboard，否则会提示 refuse。

##### 访问 kibana

1. 通过 kube-apiserver 访问：

$ kubectl cluster-info|grep -E 'Elasticsearch|Kibana'

Elasticsearch is running at https://~~192.168.100.246~~:6443/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy

Kibana is running at https://~~192.168.100.246~~:6443/api/v1/namespaces/kube-system/services/kibana-logging/proxy

浏览器访问 URL：

[https://](https://172.27.128.150:6443/api/v1/namespaces/kube-system/services/kibana-logging/proxy)~~[192.168.100.246](https://172.27.128.150:6443/api/v1/namespaces/kube-system/services/kibana-logging/proxy)~~[:6443/api/v1/namespaces/kube-system/services/kibana-logging/proxy](https://172.27.128.150:6443/api/v1/namespaces/kube-system/services/kibana-logging/proxy)

对于 virtuabox 做了端口映射：

<http://127.0.0.1:8080/api/v1/namespaces/kube-system/services/kibana-logging/proxy>

1. 通过 kubectl proxy 访问：

创建代理

$ kubectl proxy --address='~~192.168.100.246~~' --port=8086 --accept-hosts='^\*$'

Starting to serve on 172.27.129.150:8086

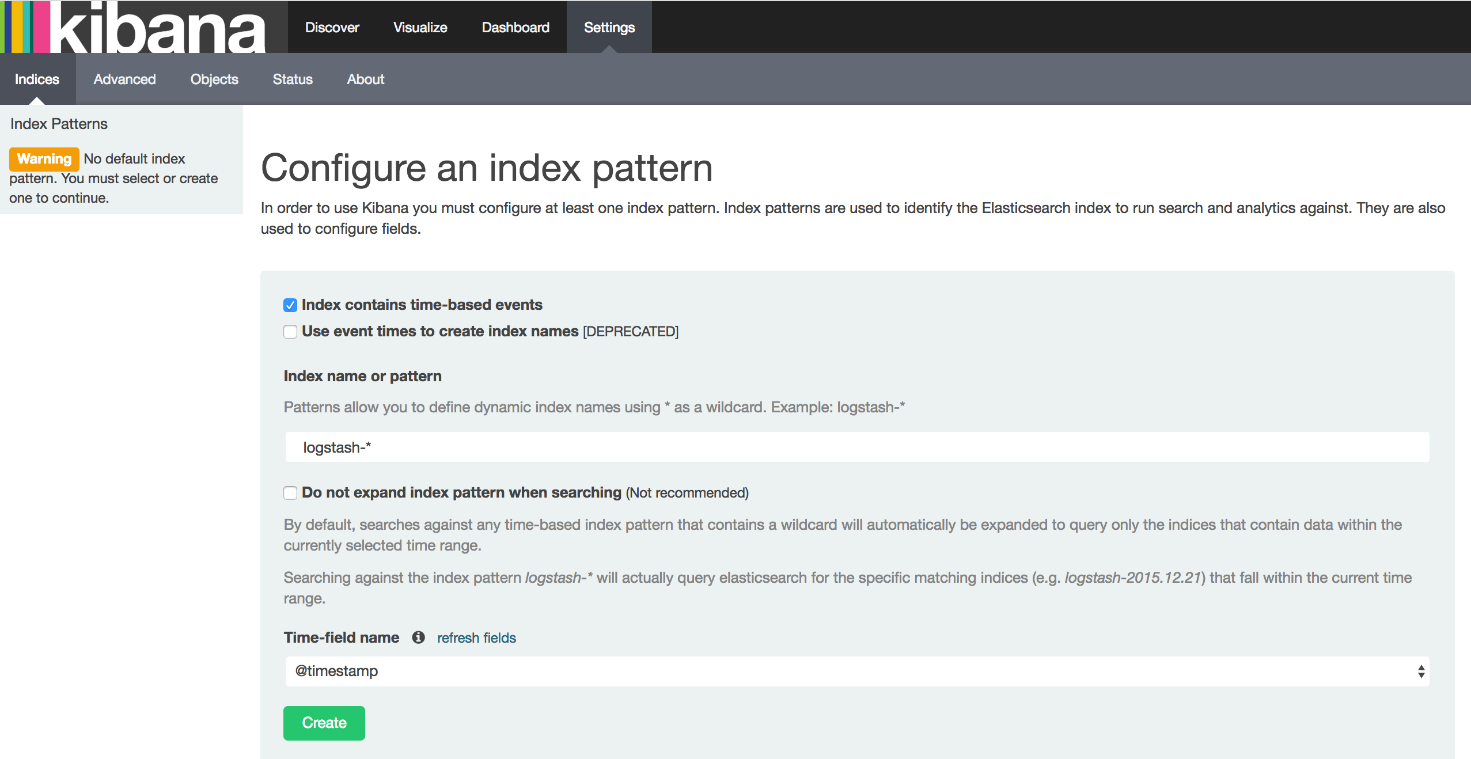
浏览器访问 URL：

[http://](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/kibana-logging/proxy)~~[192.168.100.246](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/kibana-logging/proxy)~~[:8086/api/v1/namespaces/kube-system/services/kibana-logging/proxy](http://172.27.128.150:8086/api/v1/namespaces/kube-system/services/kibana-logging/proxy)

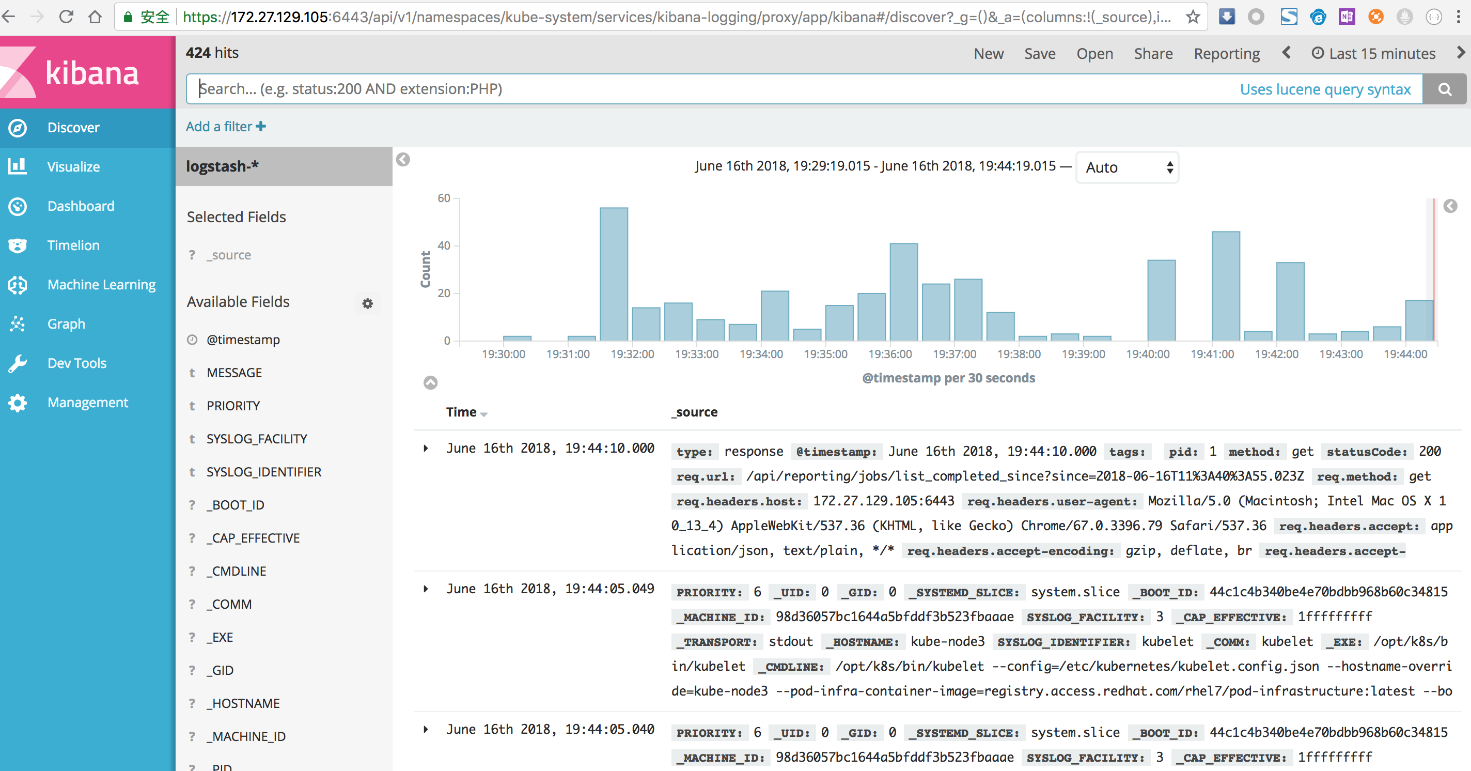
对于 virtuabox 做了端口映射：

<http://127.0.0.1:8086/api/v1/namespaces/kube-system/services/kibana-logging/proxy>

在 Settings -> Indices 页面创建一个 index（相当于 mysql 中的一个 database），选中 Index contains time-based events，使用默认的 logstash-\* pattern，点击 Create ;



创建 Index 后，稍等几分钟就可以在 Discover 菜单下看到 ElasticSearch logging 中汇聚的日志；

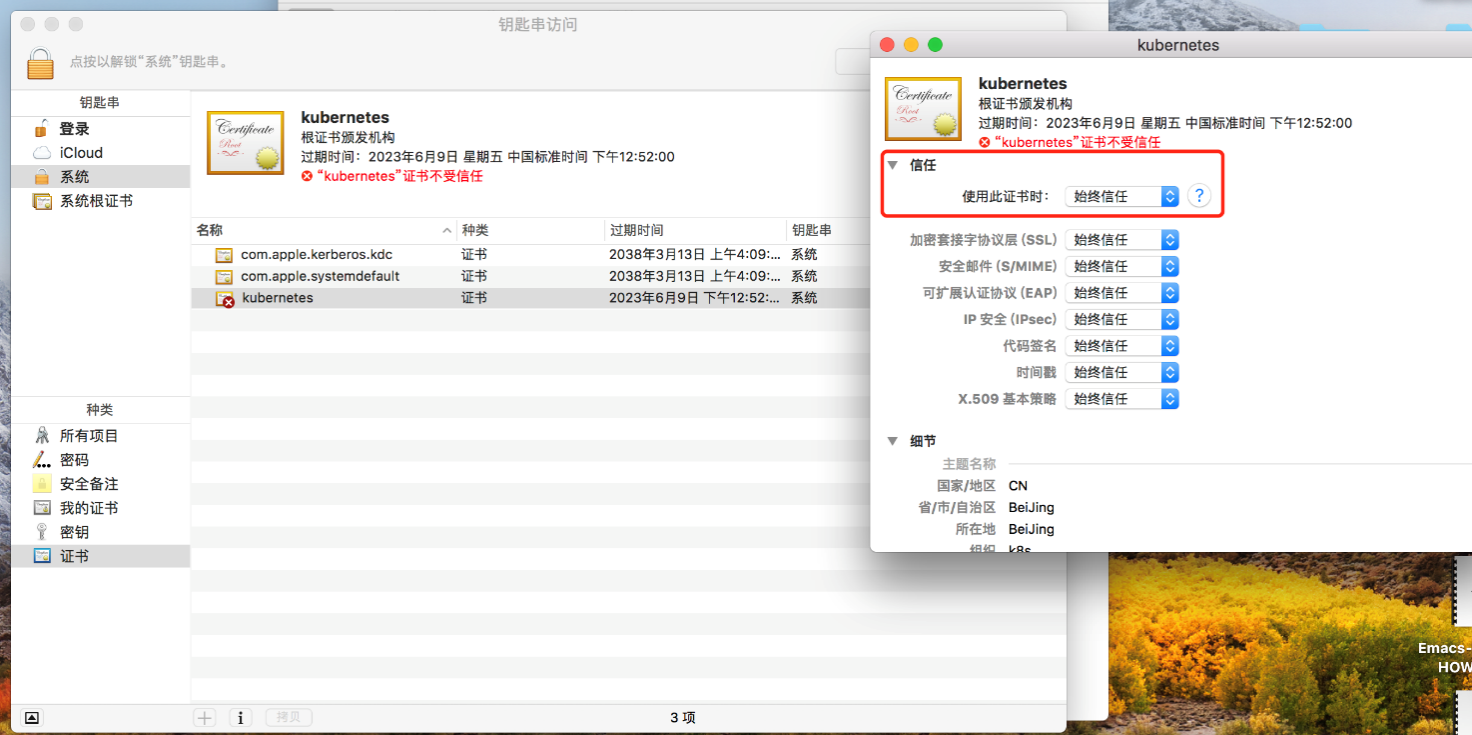


### **浏览器访问kube-apiserver安全端口**

浏览器访问 kube-apiserver 的安全端口 6443 时，提示证书不被信任：



这是因为 kube-apiserver 的 server 证书是我们创建的根证书 ca.pem 签名的，需要将根证书 ca.pem 导入操作系统，并设置永久信任。对于 Mac，操作如下：



对于windowns使用以下命令导入ca.pem

keytool -import -v -trustcacerts -alias appmanagement -file "PATH...\\ca.pem" -storepass password -keystore cacerts

再次访问 https://~~192.168.100.246~~:6443/，已信任，但提示 401，未授权的访问：

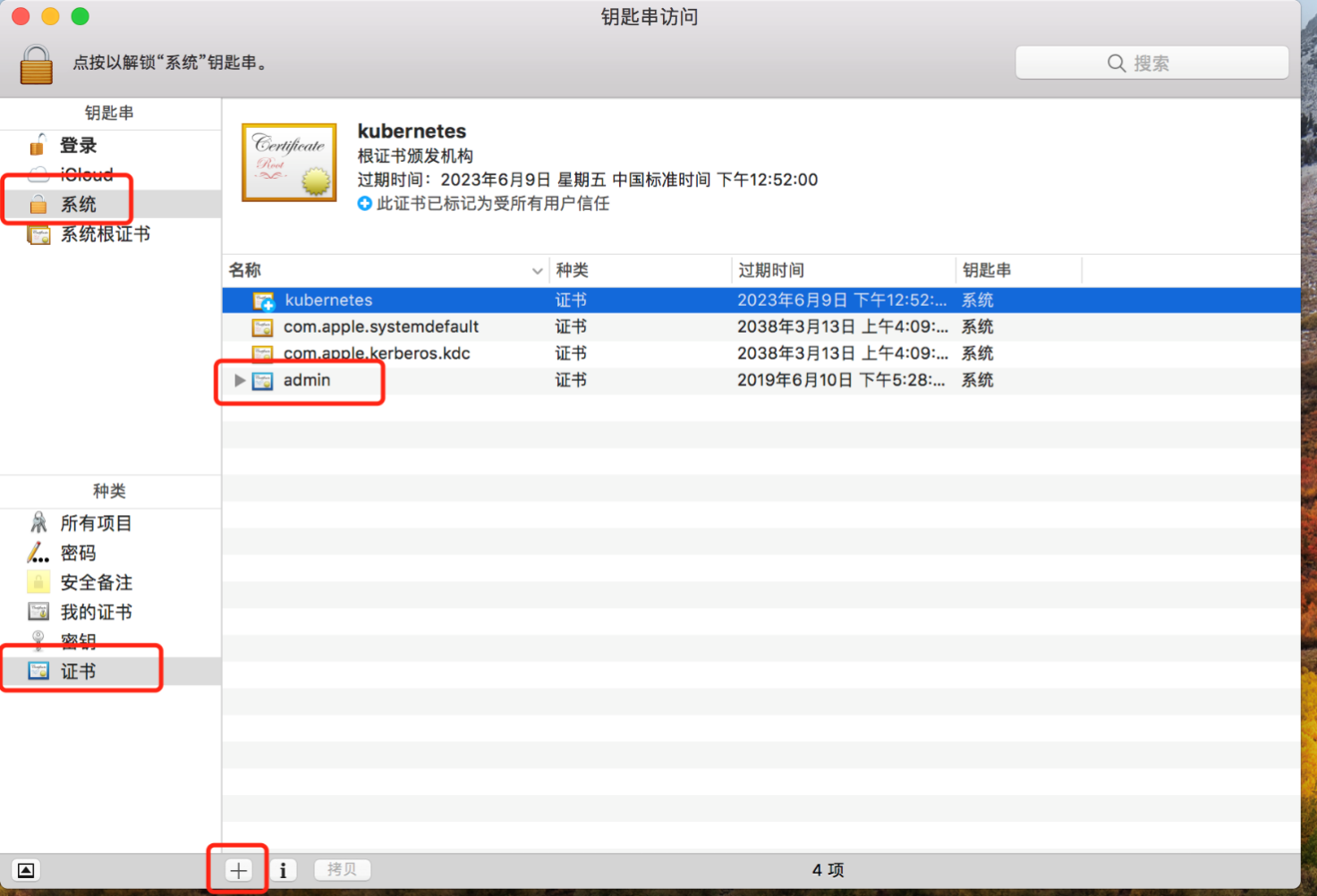


我们需要给浏览器生成一个 client 证书，访问 apiserver 的 6443 https 端口时使用。

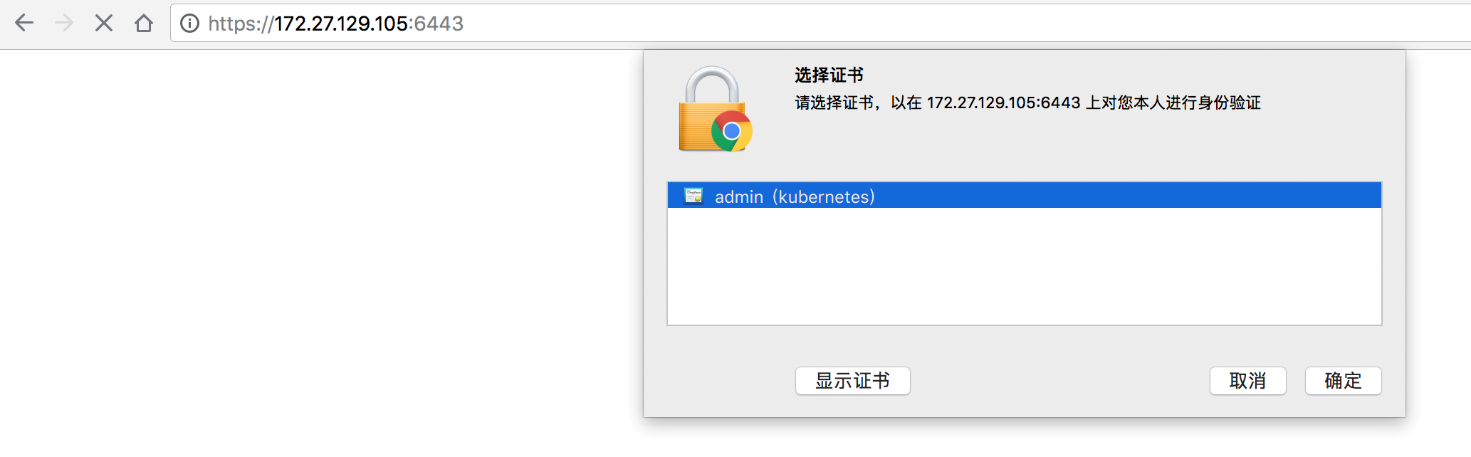
这里使用部署 kubectl 命令行工具时创建的 admin 证书、私钥和上面的 ca 证书，创建一个浏览器可以使用 PKCS#12/PFX 格式的证书：

$ openssl pkcs12 -export -out admin.pfx -inkey admin-key.pem -in admin.pem -certfile ca.pem

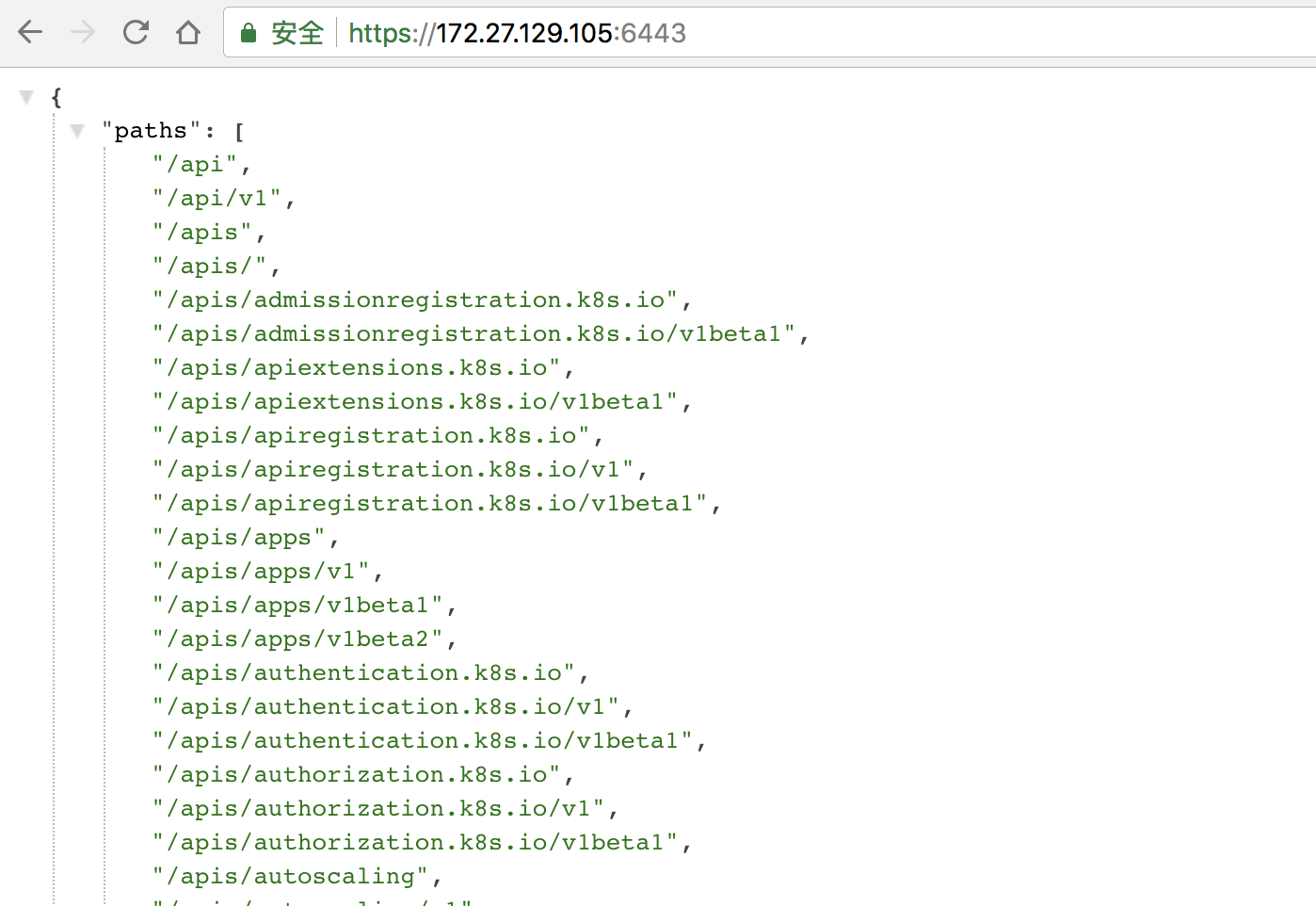
将创建的 admin.pfx 导入到系统的证书中。对于 Mac，操作如下：



重启浏览器，再次访问 https://172.27.128.150:6443/，提示选择一个浏览器证书，这里选中上面导入的 admin.pfx：



这一次，被授权访问 kube-apiserver 的安全端口：



**客户端选择证书的原理**

1. 证书选择是在客户端和服务端 SSL/TLS 握手协商阶段商定的；
2. 服务端如果要求客户端提供证书，则在握手时会向客户端发送一个它接受的 CA 列表；
3. 客户端查找它的证书列表(一般是操作系统的证书，对于 Mac 为 keychain)，看有没有被 CA 签名的证书，如果有，则将它们提供给用户选择（证书的私钥）；
4. 用户选择一个证书私钥，然后客户端将使用它和服务端通信；

**参考**

https://github.com/kubernetes/kubernetes/issues/31665

https://www.sslshopper.com/ssl-converter.html

<https://stackoverflow.com/questions/40847638/how-chrome-browser-know-which-client-certificate-to-prompt-for-a-site>

### **Weave Scope 监控集群**

创建 Kubernetes 集群并部署容器化应用只是第一步。一旦集群运行起来，我们需要确保一起正常，所有必要组件就位并各司其职，有足够的资源满足应用的需求。Kubernetes 是一个复杂系统，运维团队需要有一套工具帮助他们获知集群的实时状态，并为故障排查提供及时和准确的数据支持。

#### 安装

Weave Scope 是 Docker 和 Kubernetes 可视化监控工具。Scope 提供了至上而下的集群基础设施和应用的完整视图，用户可以轻松对分布式的容器化应用进行实时监控和问题诊断。

在 K8s 集群中安装 Scope 的方法很简单，使用下面的命令：

kubectl apply -f "https://cloud.weave.works/k8s/scope.yaml?k8s-version=$(kubectl version | base64 | tr -d '\n')&k8s-service-type=NodePort"

namespace/weave created

serviceaccount/weave-scope created

clusterrole.rbac.authorization.k8s.io/weave-scope created

clusterrolebinding.rbac.authorization.k8s.io/weave-scope created

deployment.apps/weave-scope-app created

service/weave-scope-app created

daemonset.extensions/weave-scope-agent created

部署成功后，有如下相关组件：

kubectl get pod -n weave

NAME READY STATUS RESTARTS AGE

weave-scope-agent-6l4wn 1/1 Running 0 4m55s

weave-scope-agent-dfpdl 1/1 Running 0 4m55s

weave-scope-agent-spxb4 1/1 Running 0 4m55s

weave-scope-app-6979884cc6-hpkdz 1/1 Running 0 4m55s

kubectl get svc -n weave

NAME READY UP-TO-DATE AVAILABLE AGE

weave-scope-app 1/1 1 1 5m13s

kubectl get deploy -n weave

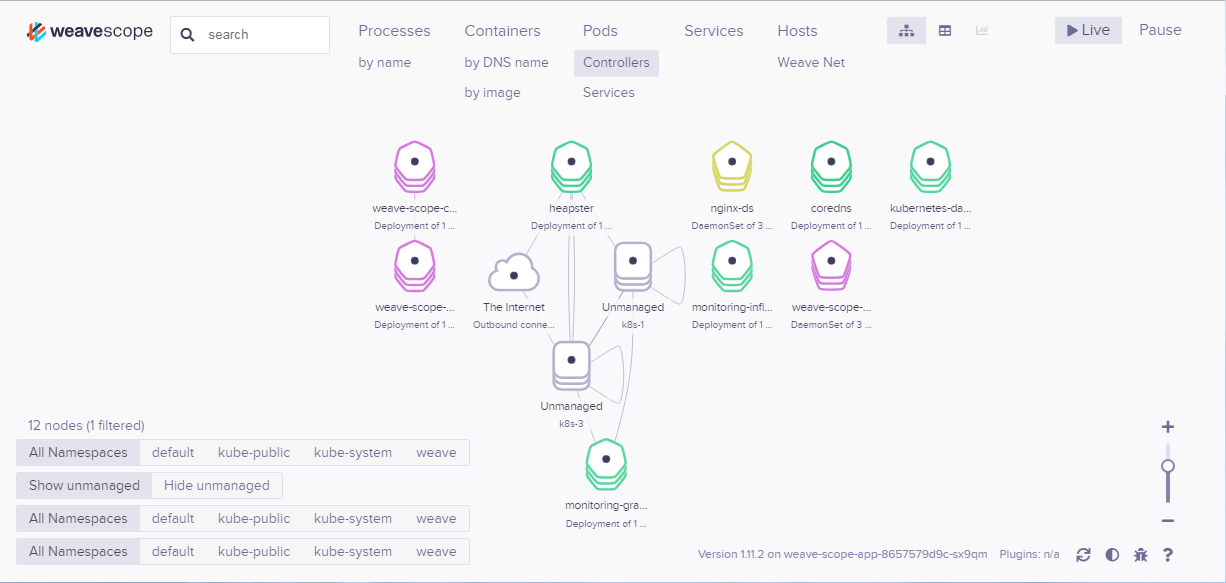
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

weave-scope-app NodePort 10.103.94.215 <none> 80:30229/TCP 5m

1. DaemonSet weave-scope-agent，集群每个节点上都会运行的 scope agent 程序，负责收集数据。
2. Deployment weave-scope-app，scope 应用，从 agent 获取数据，通过 Web UI 展示并与用户交互。
3. Service weave-scope-app，默认是 ClusterIP 类型，我们已经在上面的命令中添加了参数k8s-service-type=NodePort修改为 NodePort。

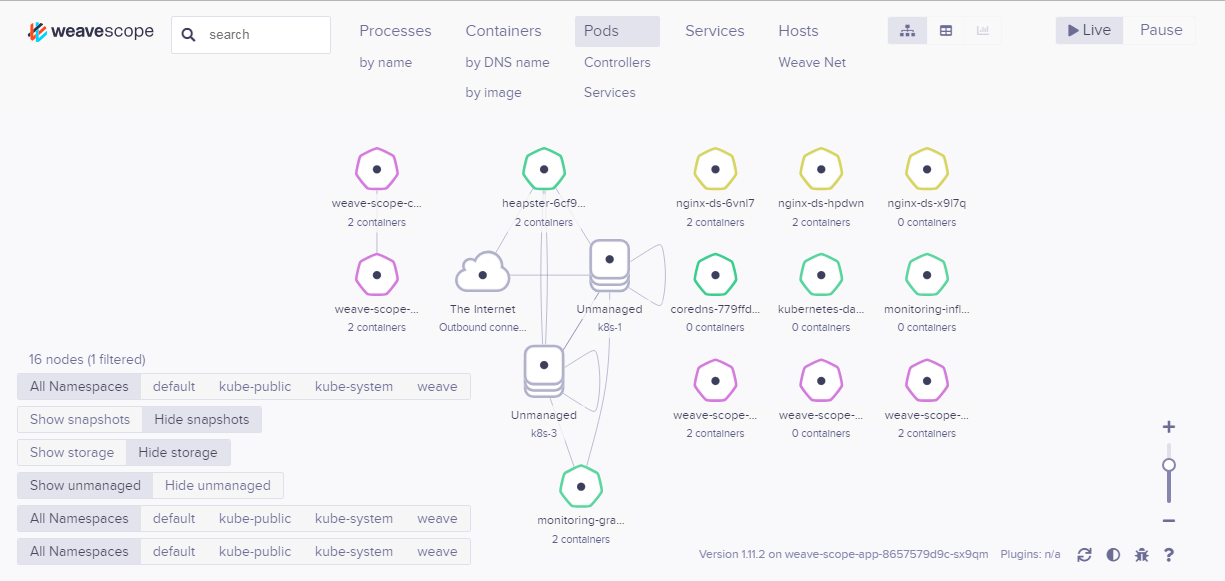
#### 使用 Scope

浏览器访问 http://MASTER\_IP:30229/，Scope 默认显示当前所有的 Controller（Deployment、DaemonSet 等）。

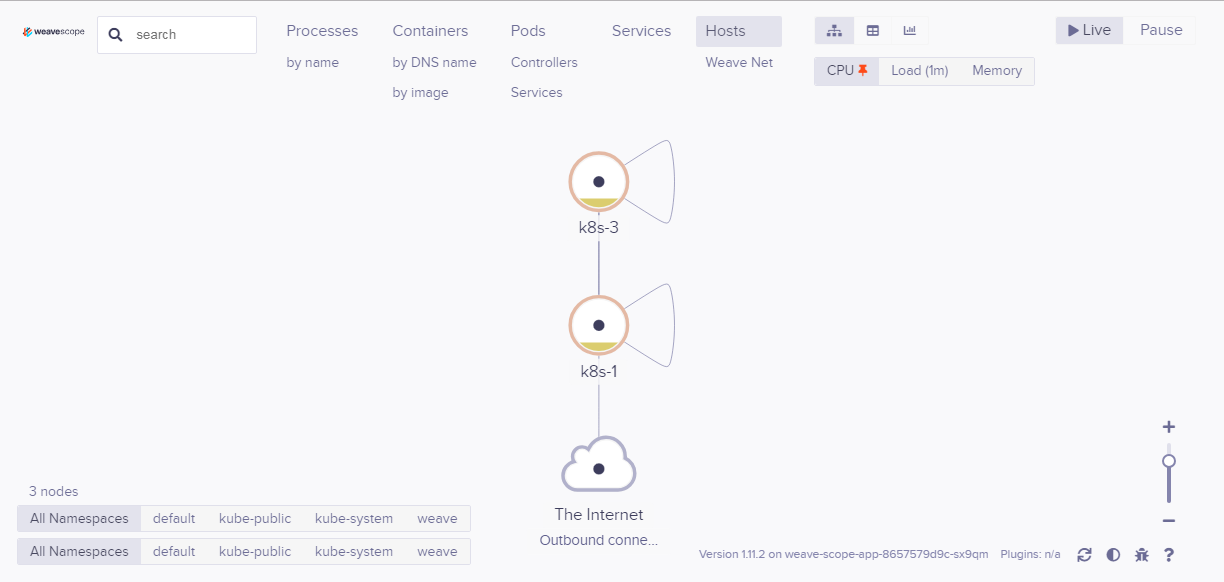


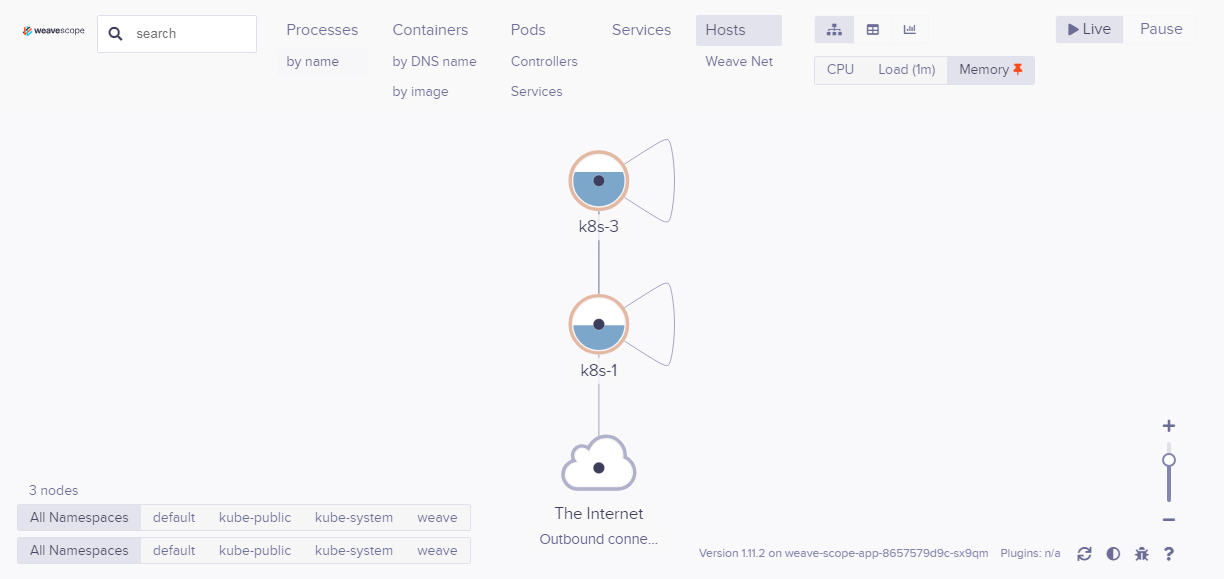
* **拓扑结构**

Scope 会自动构建应用和集群的逻辑拓扑。比如点击顶部 Pods，会显示所有 Pod 以及 Pod 之间的依赖关系。



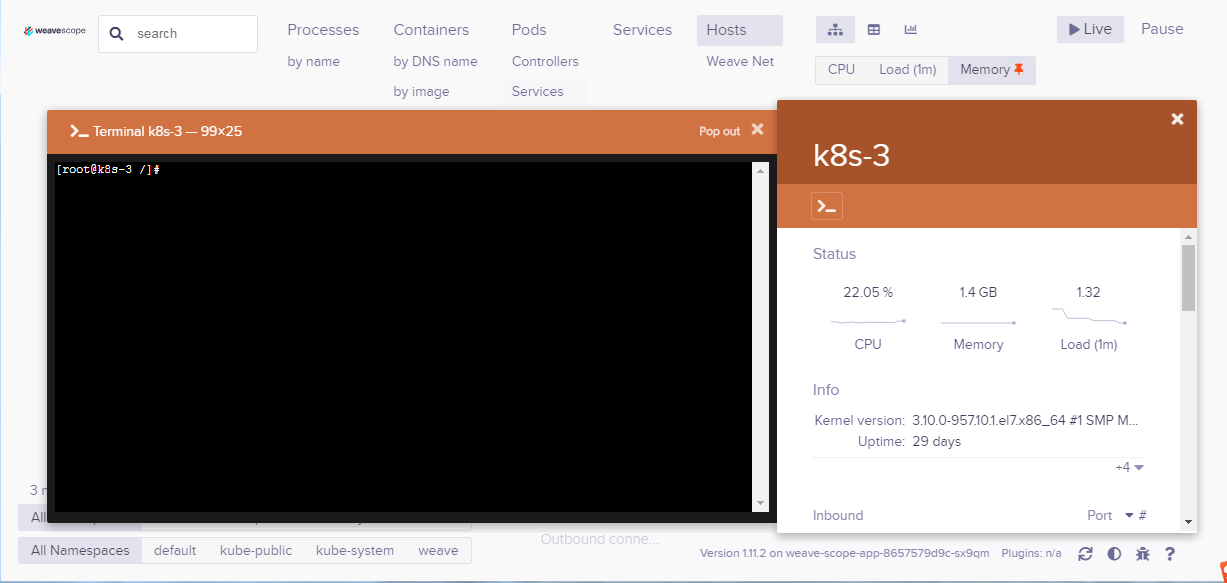
点击 Hosts，会显示各个节点之间的关系，可以在 Scope 中查看资源的 CPU 和内存使用情况。



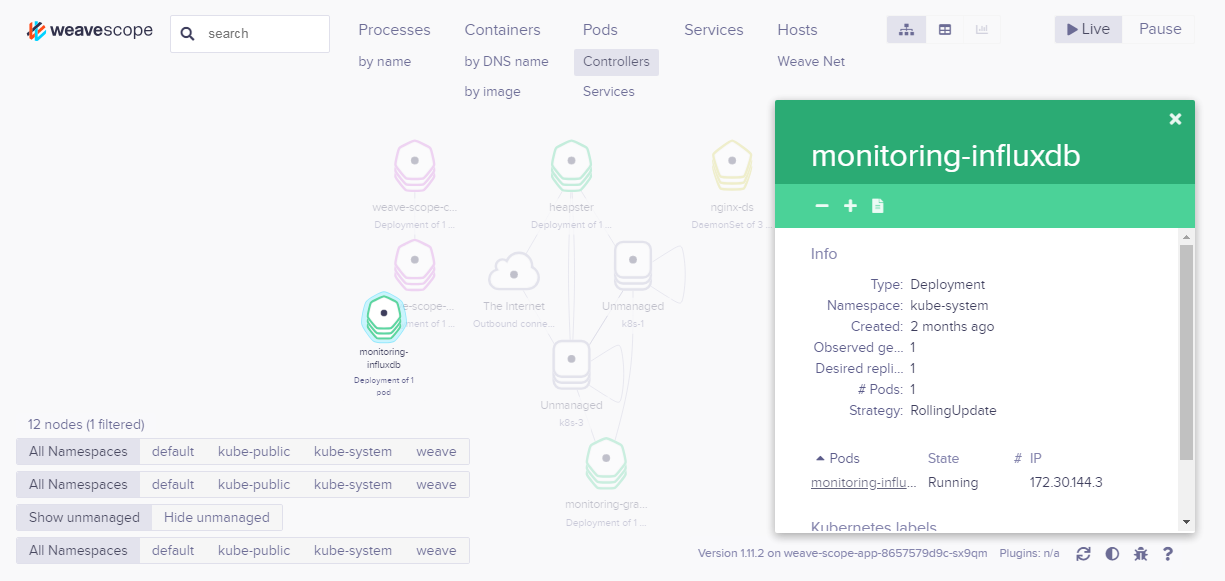


* **在线操作**

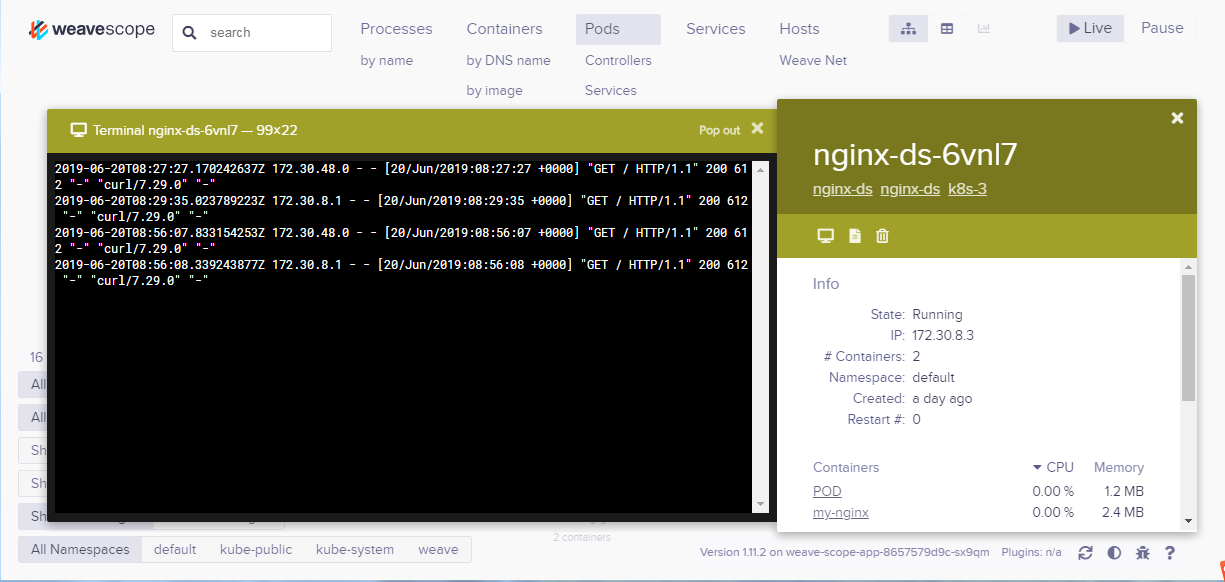
Scope 还提供了便捷的在线操作功能，比如选中某个 Host，点击 >\_ 按钮可以直接在浏览器中打开节点的命令行终端：



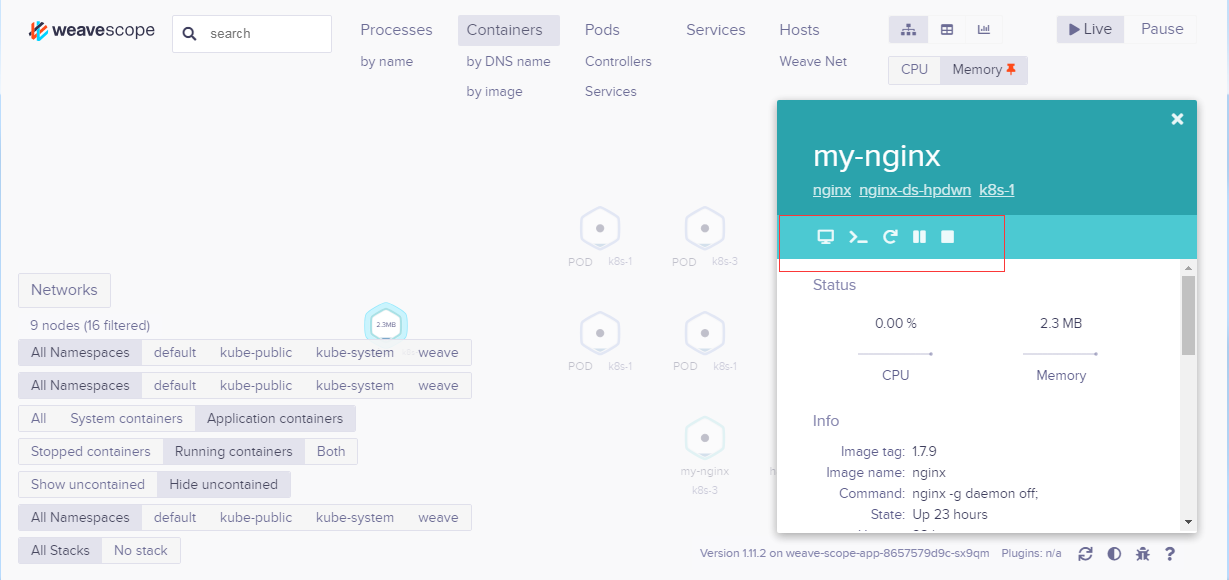
点击 Deployment 的 + 可以执行 Scale Up 操作：



可以查看 Pod 的日志：

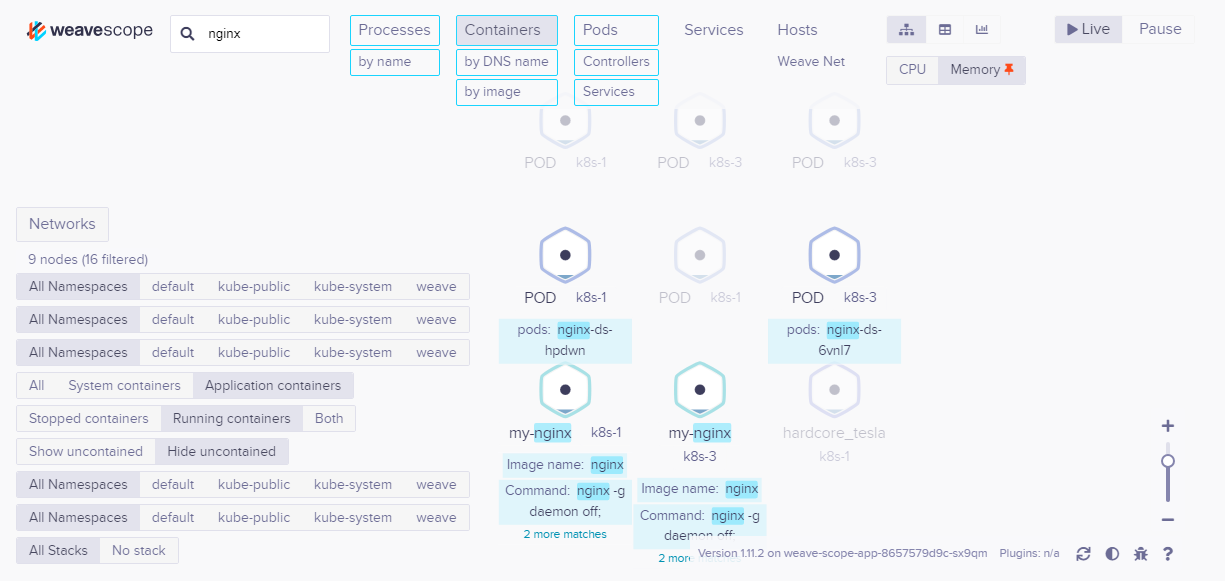


可以 attach、restart、stop 容器，以及直接在 Scope 中排查问题：

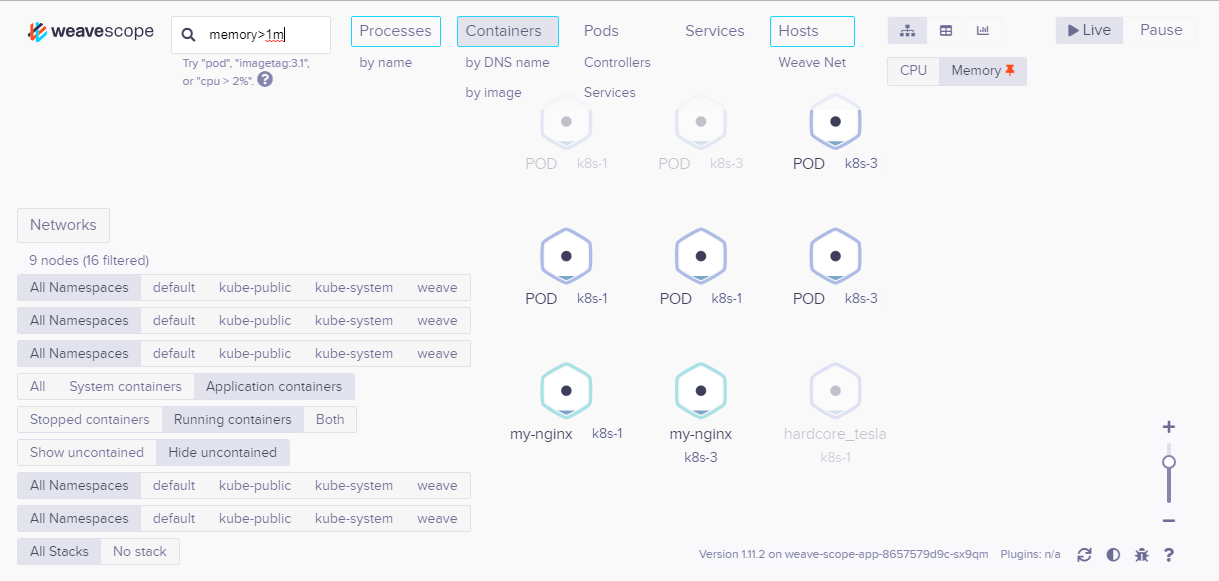


* **强大的搜索功能**

Scope 支持关键字搜索和定位资源。



还可以进行条件搜索，比如查找和定位 MEMORY > 1M 的 Containers。



Weave Scope 界面极其友好，操作简洁流畅，更多功能留给大家去探索。

官方文档：[https://www.weave.works/docs/scope/latest/installing/#k8s](https://www.weave.works/docs/scope/latest/installing/" \l "k8s)

# kubernetes 进阶

## 搭建 K8S 高可用集群

参考

集群环境搭建

## K8S 网络方案详解

Kubernetes中的网络可以说对初次接触Kubernetes或者没有网络方面经验的人来说可能是其中最难的部分。Kubernetes本身并不提供网络功能，只是把网络接口开放出来，通过插件的形式实现。

### 网络要解决的问题

既然Kubernetes中将容器的联网通过插件的方式来实现，那么该如何解决容器的联网问题呢？

如果您在本地单台机器上运行docker容器的话会注意到所有容器都会处在docker0网桥自动分配的一个网络IP段内（172.17.0.1/16）。该值可以通过docker启动参数--bip来设置。这样所有本地的所有的容器都拥有了一个IP地址，而且还是在一个网段内彼此就可以互相通信了。

但是Kubernetes管理的是集群，Kubernetes中的网络要解决的核心问题就是每台主机的IP地址网段划分，以及单个容器的IP地址分配。概括为：

* 保证每个Pod拥有一个集群内唯一的IP地址
* 保证不同节点的IP地址划分不会重复
* 保证跨节点的Pod可以互相通信
* 保证不同节点的Pod可以与跨节点的主机互相通信

为了解决该问题，出现了一系列开源的Kubernetes中的网络插件与方案，如：

* flannel
* calico
* contiv
* weave net
* kube-router
* cilium
* canal

还有很多就不一一列举了，只要实现Kubernetes官方的设计的CNI - Container Network Interface（容器网络接口）就可以自己写一个网络插件。

### k8s 网络模型

#### 基础原则

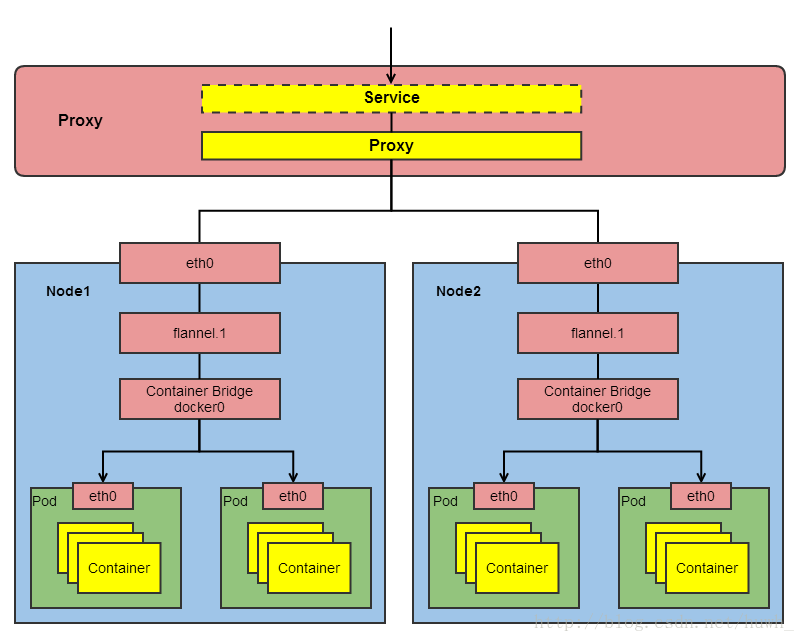
1. 每个Pod都拥有一个独立的IP地址，而且假定所有Pod都在一个可以直接连通的、扁平的网络空间中，不管是否运行在同一Node上都可以通过Pod的IP来访问。
2. k8s中Pod的IP是最小粒度IP。同一个Pod内所有的容器共享一个网络堆栈，该模型称为IP-per-Pod模型。
3. Pod由docker0实际分配的IP，Pod内部看到的IP地址和端口与外部保持一致。同一个Pod内的不同容器共享网络，可以通过localhost来访问对方的端口，类似同一个VM内的不同进程。
4. IP-per-Pod模型从端口分配、域名解析、服务发现、负载均衡、应用配置等角度看，Pod可以看作是一台独立的VM或物理机。

#### k8s 对集群网络要求

1. 所有容器都可以不用NAT的方式同别的容器通信。
2. 所有节点都可以在不同NAT的方式下同所有容器通信，反之亦然。
3. 容器的地址和别人看到的地址是同一个地址。

以上的集群网络要求可以通过第三方开源的Kubernetes网络插件实现。具体参考上面介绍的开源实现方式。

#### 网络架构图



#### k8s 集群 IP 概念汇总

由集群外部到集群内部：

|  |  |
| --- | --- |
| **IP类型** | **说明** |
| Proxy-IP | 代理层公网地址IP，外部访问应用的网关服务器。[实际需要关注的IP] |
| Service-IP | Service的固定虚拟IP，Service-IP是内部，外部无法寻址到。 |
| Node-IP | 容器宿主机的主机IP。 |
| Container-Bridge-IP | 容器网桥（docker0）IP，容器的网络都需要通过容器网桥转发。 |
| Pod-IP | Pod的IP，等效于Pod中网络容器的Container-IP。 |
| Container-IP | 容器的IP，容器的网络是个隔离的网络空间。 |

### docker 的网络基础

#### Linx 网络名词解释

* **Network Namespace**

不同的网络命名空间中，协议栈是独立的，完全隔离，彼此之间无法通信。同一个网络命名空间有独立的路由表和独立的Iptables/Netfilter来提供包的转发、NAT、IP包过滤等功能。

* 网络命名空间的实现

将与网络协议栈相关的全局变量变成一个Net Namespace变量的成员，然后在调用协议栈函数中加入一个Namepace参数。

* 网络命名空间的操作

1、创建网络命名空间

ip netns add

2、在命名空间内执行命令

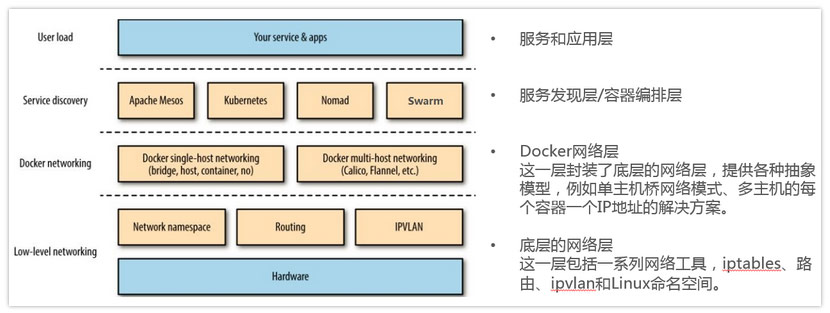
ip netns exec

3、进入命名空间

ip netns exec bash

* veth 设备对：Veth设备对的引入是为了实现在不同网络命名空间的通信。
* Iptables/Netfilter：Netfilter负责在内核中执行各种挂接的规则(过滤、修改、丢弃等)，运行在内核 模式中；Iptables模式是在用户模式下运行的进程，负责协助维护内核中Netfilter的各种规则表；通过二者的配合来实现整个Linux网络协议栈中灵活的数据包处理机制。
* 网桥：网桥是一个二层网络设备,通过网桥可以将linux支持的不同的端口连接起来,并实现类似交换机那样的多对多的通信。
* 路由：Linux系统包含一个完整的路由功能，当IP层在处理数据发送或转发的时候，会使用路由表来决定发往哪里。

#### Docker网络在整个Docker生态技术栈中的位置



### docker 的网络实现

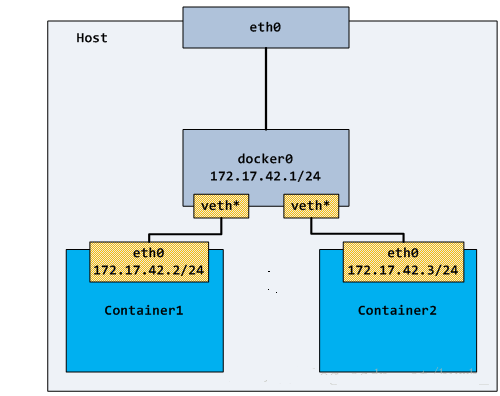
1、单机网络模式：Bridge 、Host、Container、None，下面我们会重点介绍。

2、多机网络模式：一类是 Docker 在 1.9 版本中引入Libnetwork项目，对跨节点网络的原生支持；一类是通过插件（plugin）方式引入的第三方实现方案，比如 Flannel，Calico 等等。

#### 容器网络

Docker使用Linux桥接，在宿主机虚拟一个Docker容器网桥(docker0)，Docker启动一个容器时会根据Docker网桥的网段分配给容器一个IP地址，称为Container-IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能够通过容器的Container-IP直接通信。

Docker网桥是宿主机虚拟出来的，并不是真实存在的网络设备，外部网络是无法寻址到的，这也意味着外部网络无法通过直接Container-IP访问到容器。如果容器希望外部访问能够访问到，可以通过映射容器端口到宿主主机（端口映射），即docker run创建容器时候通过 -p 或 -P 参数来启用，访问容器的时候就通过[宿主机IP]:[容器端口]访问容器。



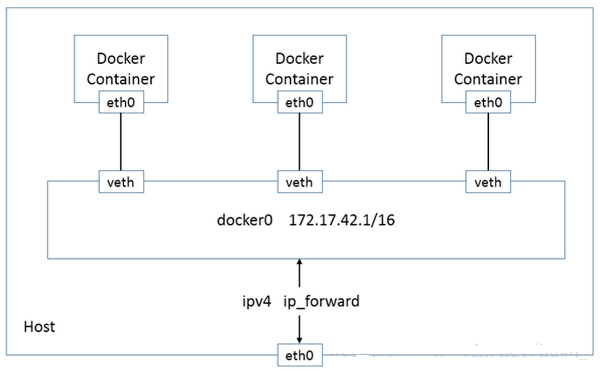
#### 4种网络模式

|  |  |  |
| --- | --- | --- |
| **Docker网络模式** | **配置** | **说明** |
| host模式 | –net=host | 容器和宿主机共享Network namespace。 |
| container模式 | –net=container:  NAME\_or\_ID | 容器和另外一个容器共享Network namespace。 kubernetes中的pod就是多个容器共享一个Network namespace。 |
| none模式 | –net=none | 容器有独立的Network namespace，但并没有对其进行任何网络设置，如分配veth pair 和网桥连接，配置IP等。 |
| bridge模式 | –net=bridge  （默认） |  |

#### bridge模式

k8s只使用bridge模式。

在bridge模式下，Docker Daemon首次启动时会创建一个虚拟网桥docker0，地址通常为172.x.x.x开头，在私有的网络空间中给这个网络分配一个子网。由Docker创建处理的每个容器，都会创建一个虚拟以太设备（Veth设备对），一端关联到网桥，另一端使用Namespace技术映射到容器内的eth0设备，然后从网桥的地址段内给eth0接口分配一个IP地址。



一般情况，宿主机IP与docker0 IP、容器IP是不同的IP段，默认情况，外部看不到docker0和容器IP，对于外部来说相当于docker0和容器的IP为内网IP。

### k8s 的网络实现

k8s网络场景

1. 容器与容器之间的直接通信。
2. Pod与Pod之间的通信。
3. Pod到Service之间的通信。
4. 集群外部与内部组件之间的通信。

#### pod 网络

Pod作为kubernetes的最小调度单元，Pod是容器的集合，是一个逻辑概念，Pod包含的容器都运行在同一个宿主机上，这些容器将拥有同样的网络空间，容器之间能够互相通信，它们能够在本地访问其它容器的端口。 实际上Pod都包含一个网络容器，它不做任何事情，只是用来接管Pod的网络，业务容器通过加入网络容器的网络从而实现网络共享。Pod网络本质上还是容器网络，所以Pod-IP就是网络容器的Container-IP。

一般将容器云平台的网络模型打造成一个扁平化网络平面，在这个网络平面内，Pod作为一个网络单元同Kubernetes Node的网络处于同一层级。

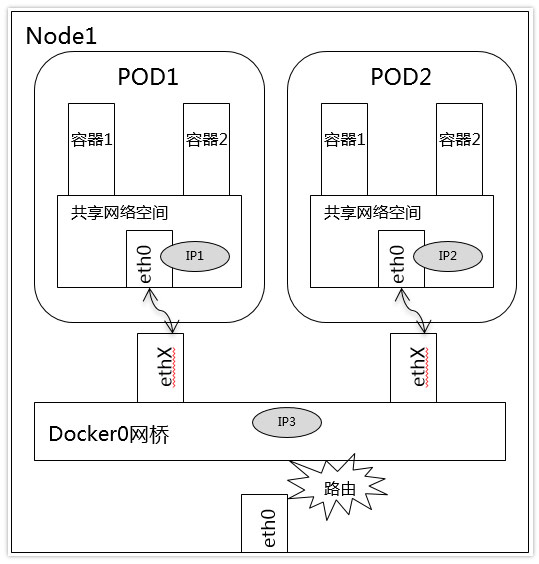
#### 容器之间的通信

同一个Pod之间的不同容器因为共享同一个网络命名空间，所以可以直接通过localhost直接通信。

#### pod之间的通信

##### 同 node 的 pod之间的通信

同一个Node内，不同的Pod都有一个全局IP，可以直接通过Pod的IP进行通信。Pod地址和docker0在同一个网段。

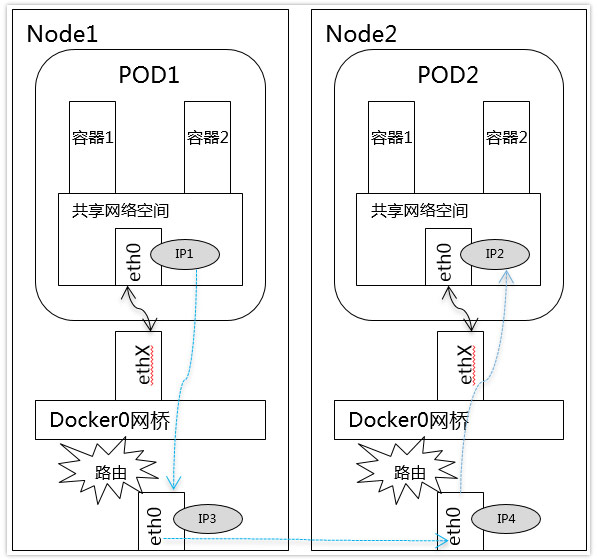


##### 不同 node 的 pod 之间的通信

不同的Node之间，Node的IP相当于外网IP，可以直接访问，而Node内的docker0和Pod的IP则是内网IP，无法直接跨Node访问。需要通过Node的网卡进行转发。

所以不同Node之间的通信需要达到两个条件：

1. 对整个集群中的Pod-IP分配进行规划，不能有冲突（可以通过第三方开源工具来管理，例如flannel）。
2. 将Node-IP与该Node上的Pod-IP关联起来，通过Node-IP再转发到Pod-IP。



* **pod 间实现通信**

例如：Pod1和Pod2（同主机），Pod1和Pod3(跨主机)能够通信

实现：因为Pod的Pod-IP是Docker网桥分配的，Pod-IP是同Node下全局唯一的。所以将不同Kubernetes Node的 Docker网桥配置成不同的IP网段即可。

* **node 与 pod 间实现通信**

例如：Node1和Pod1/ Pod2(同主机)，Pod3(跨主机)能够通信

实现：在容器集群中创建一个覆盖网络(Overlay Network)，联通各个节点，目前可以通过第三方网络插件来创建覆盖网络，比如Flannel和Open vSwitch等。

#### service 网络

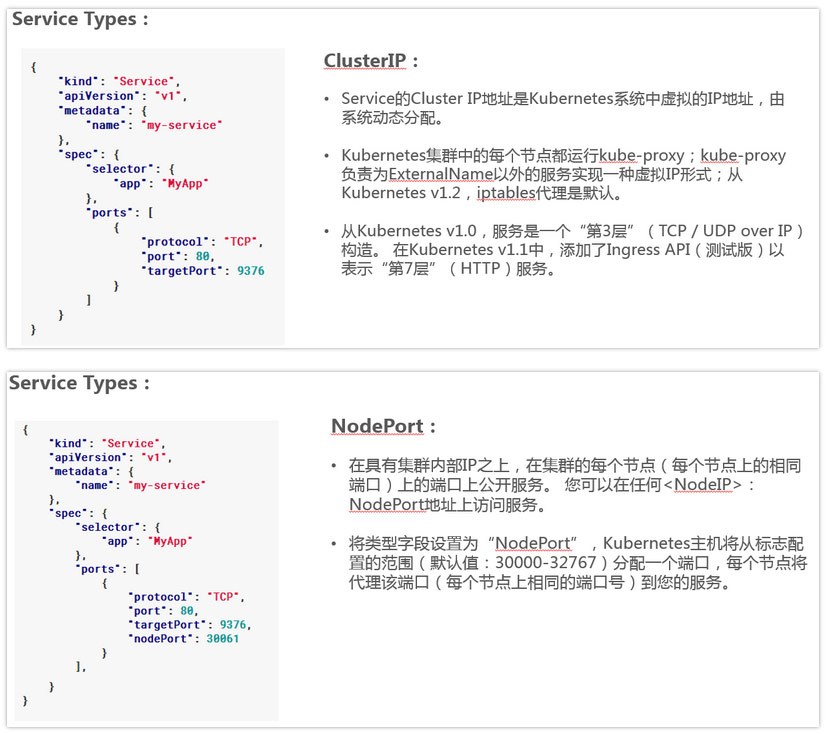
Service的就是在Pod之间起到服务代理的作用，对外表现为一个单一访问接口，将请求转发给Pod，Service的网络转发是Kubernetes实现服务编排的关键一环。Service都会生成一个虚拟IP，称为Service-IP， Kuberenetes Porxy组件负责实现Service-IP路由和转发，在容器覆盖网络之上又实现了虚拟转发网络。

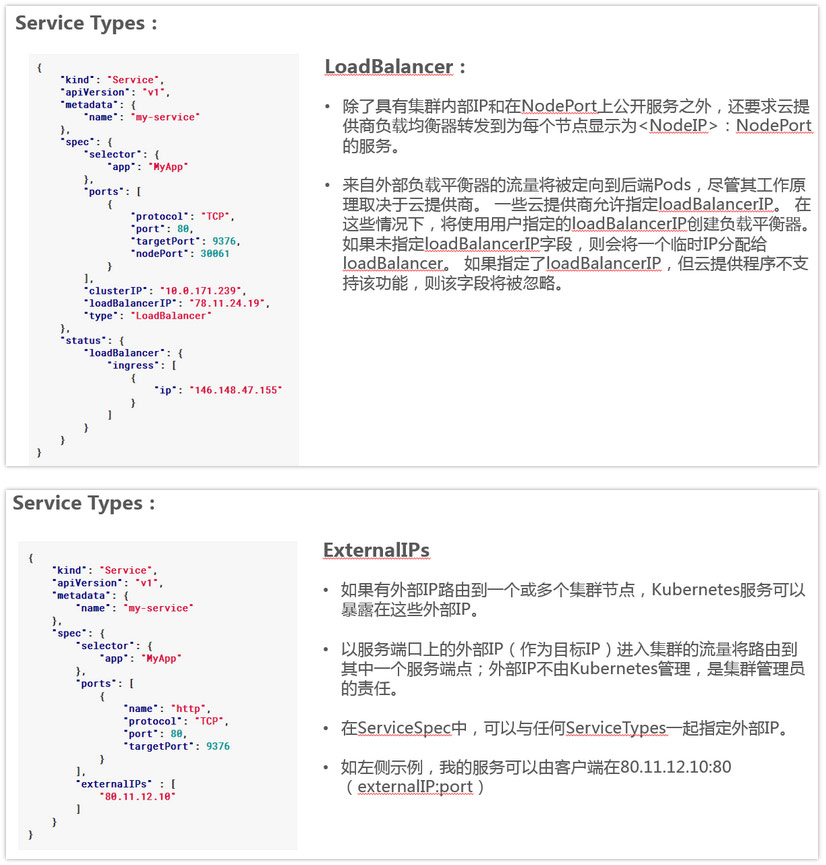
Kubernetes Porxy实现了以下功能：

1. 转发访问Service的Service-IP的请求到Endpoints(即Pod-IP)。
2. 监控Service和Endpoints的变化，实时刷新转发规则。
3. 负载均衡能力。

**简单点理解**：

Service是一组Pod的服务抽象，相当于一组Pod的LB，负责将请求分发给对应的Pod；Service会为这个LB提供一个IP，一般称为ClusterIP。





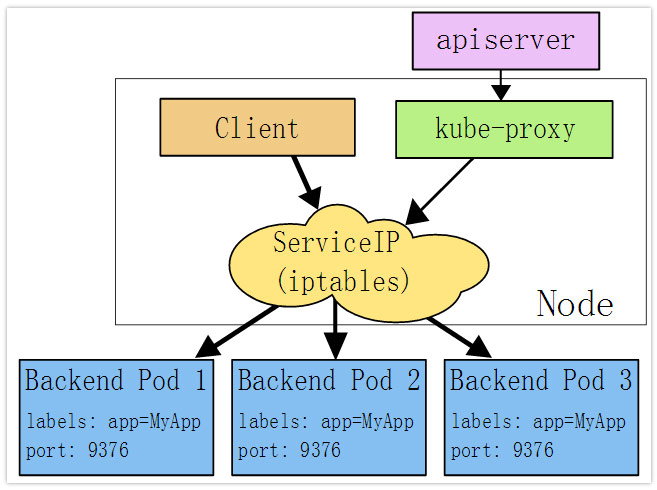
#### kube-proxy

Kube-proxy是一个简单的网络代理和负载均衡器，它的作用主要是负责Service的实现，具体来说，就是实现了内部从Pod到Service和外部的从NodePort向Service的访问。

实现方式：

* userspace是在用户空间，通过kuber-proxy实现LB的代理服务，这个是kube-proxy的最初的版本，较为稳定，但是效率也自然不太高。
* iptables是纯采用iptables来实现LB，是目前kube-proxy默认的方式。

下面是iptables模式下Kube-proxy的实现方式：



* 在这种模式下，kube-proxy监视Kubernetes主服务器添加和删除服务和端点对象。对于每个服务，它安装iptables规则，捕获到服务的clusterIP（虚拟）和端口的流量，并将流量重定向到服务的后端集合之一。对于每个Endpoints对象，它安装选择后端Pod的iptables规则。
* 默认情况下，后端的选择是随机的。可以通过将service.spec.sessionAffinity设置为“ClientIP”（默认为“无”）来选择基于客户端IP的会话关联。
* 与用户空间代理一样，最终结果是绑定到服务的IP:端口的任何流量被代理到适当的后端，而客户端不知道关于Kubernetes或服务或Pod的任何信息。这应该比用户空间代理更快，更可靠。然而，与用户空间代理不同，如果最初选择的Pod不响应，则iptables代理不能自动重试另一个Pod，因此它取决于具有工作准备就绪探测。

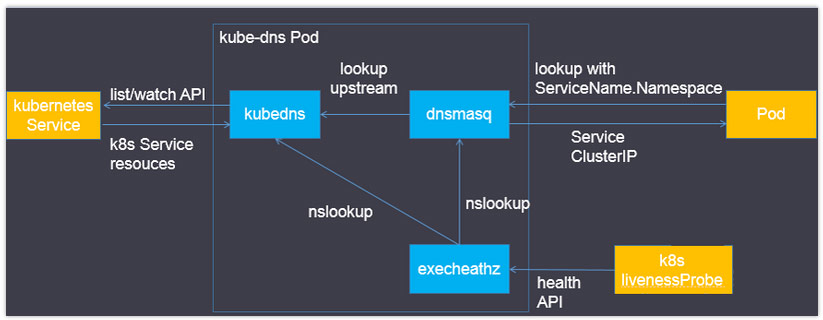
#### kube-dns

Kube-dns用来为kubernetes service分配子域名，在集群中可以通过名称访问service；通常kube-dns会为service赋予一个名为“service名称.namespace.svc.cluster.local”的A记录，用来解析service的clusterip。

Kube-dns组件：

在Kubernetes v1.4版本之前由“Kube2sky、Etcd、Skydns、Exechealthz”四个组件组成。

在Kubernetes v1.4版本及之后由“Kubedns、dnsmasq、exechealthz”三个组件组成。



**Kubedns**

接入SkyDNS，为dnsmasq提供查询服务。

替换etcd容器，使用树形结构在内存中保存DNS记录。

通过K8S API监视Service资源变化并更新DNS记录。

服务10053端口。

**Dnsmasq**

Dnsmasq是一款小巧的DNS配置工具。

在kube-dns插件中的作用是：

通过kubedns容器获取DNS规则，在集群中提供DNS查询服务

提供DNS缓存，提高查询性能

降低kubedns容器的压力、提高稳定性

Dockerfile在GitHub上Kubernetes组织的contrib仓库中，位于dnsmasq目录下。

在kube-dns插件的编排文件中可以看到，dnsmasq通过参数–server=127.0.0.1:10053指定upstream为kubedns。

**Exechealthz**

在kube-dns插件中提供健康检查功能。

源码同样在contrib仓库中，位于exec-healthz目录下。

新版中会对两个容器都进行健康检查，更加完善。

### 开源的网络组件

容器网络发展到现在，形成了两大阵营，就是Docker的**CNM**和Google、CoreOS、Kuberenetes主导的**CNI**。首先明确一点，CNM和CNI并不是网络实现，他们是网络规范和网络体系，从研发的角度他们就是一堆接口，你底层是用Flannel也好、用Calico也好，他们并不关心，CNM和CNI关心的是网络管理的问题。

#### CNM（Docker LibnetworkContainer Network Model）

Docker Libnetwork的优势就是原生，而且和Docker容器生命周期结合紧密；缺点也可以理解为是原生，被Docker“绑架”。

* Docker Swarm overlay
* Macvlan & IP networkdrivers
* Calico
* Contiv
* Weave

#### CNI（Container NetworkInterface）

CNI的优势是兼容其他容器技术（e.g. rkt）及上层编排系统（Kubernetes & Mesos)，而且社区活跃势头迅猛，Kubernetes加上CoreOS主推；缺点是非Docker原生。

* Kubernetes
* Weave
* Macvlan
* Calico
* Flannel
* Contiv
* Mesos CNI

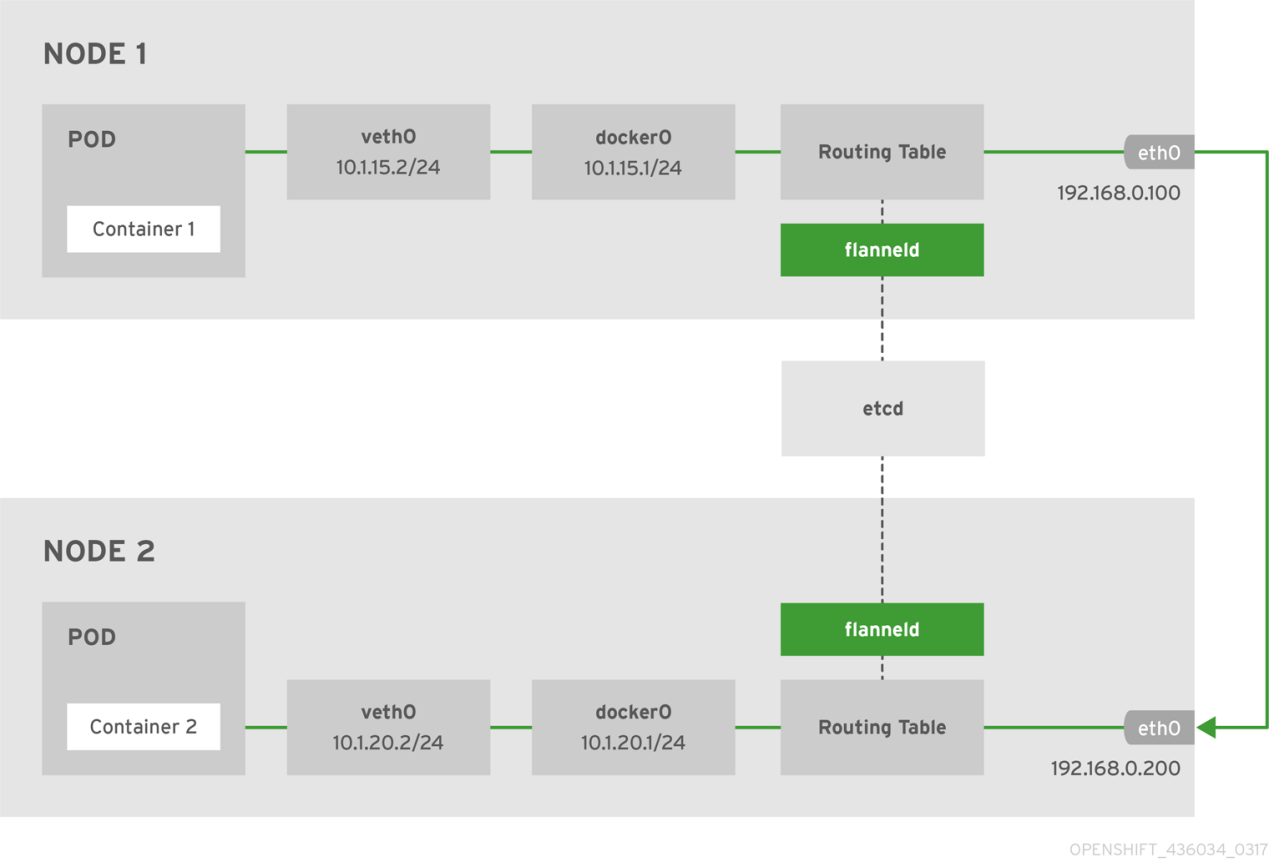
#### Flannel 容器网络

Flannel之所以可以搭建kubernets依赖的底层网络，是因为它可以实现以下两点：

* 它给每个node上的docker容器分配相互不想冲突的IP地址；
* 它能给这些IP地址之间建立一个覆盖网络，同过覆盖网络，将数据包原封不动的传递到目标容器内。

Flannel介绍

* Flannel是CoreOS团队针对Kubernetes设计的一个网络规划服务，简单来说，它的功能是让集群中的不同节点主机创建的Docker容器都具有全集群唯一的虚拟IP地址。
* 在默认的Docker配置中，每个节点上的Docker服务会分别负责所在节点容器的IP分配。这样导致的一个问题是，不同节点上容器可能获得相同的内外IP地址。并使这些容器之间能够之间通过IP地址相互找到，也就是相互ping通。
* Flannel的设计目的就是为集群中的所有节点重新规划IP地址的使用规则，从而使得不同节点上的容器能够获得“同属一个内网”且”不重复的”IP地址，并让属于不同节点上的容器能够直接通过内网IP通信。
* Flannel实质上是一种“覆盖网络(overlaynetwork)”，也就是将TCP数据包装在另一种网络包里面进行路由转发和通信，目前已经支持udp、vxlan、host-gw、aws-vpc、gce和alloc路由等数据转发方式，默认的节点间数据通信方式是UDP转发。



#### Calico 容器网络

Calico介绍

* Calico是一个纯3层的数据中心网络方案，而且无缝集成像OpenStack这种IaaS云架构，能够提供可控的VM、容器、裸机之间的IP通信。Calico不使用重叠网络比如flannel和libnetwork重叠网络驱动，它是一个纯三层的方法，使用虚拟路由代替虚拟交换，每一台虚拟路由通过BGP协议传播可达信息（路由）到剩余数据中心。
* Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的workload的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。
* Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。
* Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。



Calico主要由Felix、etcd、BGP client、BGP Route Reflector组成。

Etcd：负责存储网络信息

BGP client：负责将Felix配置的路由信息分发到其他节点

Felix：Calico Agent，每个节点都需要运行，主要负责配置路由、配置ACLs、报告状态

BGP Route Reflector：大规模部署时需要用到，作为BGP client的中心连接点，可以避免每个节点互联

## 内置的负载均衡机制

Kubernetes在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了Service资源，并通过kube-proxy配合cloud provider来适应不同的应用场景。随着kubernetes用户的激增，用户场景的不断丰富，又产生了一些新的负载均衡机制。目前，kubernetes中的负载均衡大致可以分为以下几种机制，每种机制都有其特定的应用场景：

* Service：直接用Service提供cluster内部的负载均衡，并借助cloud provider提供的LB提供外部访问
* Ingress Controller：还是用Service提供cluster内部的负载均衡，但是通过自定义LB提供外部访问
* Service Load Balancer：把load balancer直接跑在容器中，实现Bare Metal的Service Load Balancer
* Custom Load Balancer：自定义负载均衡，并替代kube-proxy，一般在物理部署Kubernetes时使用，方便接入公司已有的外部服务

### Service

Service是对一组提供相同功能的Pods的抽象，并为它们提供一个统一的入口。借助Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service通过标签来选取服务后端，一般配合Replication Controller或者Deployment来保证后端容器的正常运行。

Service目前有三种类型：

* ClusterIP：默认类型，自动分配一个仅cluster内部可以访问的虚拟IP
* NodePort：在ClusterIP基础上为Service在每台机器上绑定一个端口，这样就可以通过<NodeIP>:NodePort来访问该服务
* LoadBalancer：在NodePort的基础上，借助cloud provider创建一个外部的负载均衡器，并将请求转发到<NodeIP>:NodePort



service 类型

Service 中常见的三种端口的含义：

* port

Service 暴露在 Cluster IP 上的端口，也就是虚拟 IP 要绑定的端口。port 是提供给集群内部客户访问 Service 的入口。

* nodePort

nodePort 是 Kubernetes 提供给集群外部客户访问 Service 的入口。

* targetPort

targetPort 是 Pod 里容器的端口，从 port 和 nodePort 上进入的数据最终会经过 Kube-Proxy 流入到后端 Pod 里容器的端口。如果 targetPort 没有被显式声明，那么会默认转发到 Service 接受请求的端口（和 port 端口的值一样）。

总的来说，port 和 nodePort 都是 Service 的端口，前者暴露给集群内客户访问服务，后者暴露给集群外客户访问服务。从这两个端口到来的数据都需要经过反向代理 Kube-Proxy 流入后端 Pod 里容器的端口，从而到达 Pod 上的容器内。

另外，也可以将已有的服务以Service的形式加入到Kubernetes集群中来，只需要在创建Service的时候不指定Label selector，而是在Service创建好后手动为其添加endpoint。

### Ingress Controller

Service虽然解决了服务发现和负载均衡的问题，但它在使用上还是有一些限制，比如

* 对外访问的时候，NodePort类型需要在外部搭建额外的负载均衡，而LoadBalancer要求kubernetes必须跑在支持的cloud provider上面

Ingress就是为了解决这些限制而引入的新资源，主要用来将服务暴露到cluster外面，并且可以自定义服务的访问策略。比如想要通过负载均衡器实现不同子域名到不同服务的访问：

foo.bar.com --| |-> foo.bar.com s1:80

| 178.91.123.132 |

bar.foo.com --| |-> bar.foo.com s2:80

可以这样来定义Ingress：

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test

spec:

rules:

- host: foo.bar.com

http:

paths:

- backend:

serviceName: s1

servicePort: 80

- host: bar.foo.com

http:

paths:

- backend:

serviceName: s2

servicePort: 80

**注意**： Ingress本身并不会自动创建负载均衡器，cluster中需要运行一个ingress controller来根据Ingress的定义来管理负载均衡器。目前社区提供了nginx和gce的参考实现。

### Service Load Balancer

在Ingress出现以前，Service Load Balancer是推荐的解决Service局限性的方式。Service Load Balancer将haproxy跑在容器中，并监控service和endpoint的变化，通过容器IP对外提供4层和7层负载均衡服务。

社区提供的Service Load Balancer支持四种负载均衡协议：TCP、HTTP、HTTPS和SSL TERMINATION，并支持ACL访问控制。

像阿里云、aws、azure、openstack、gce都提供了loadbalance服务。

### Custom Load Balancer

虽然Kubernetes提供了丰富的负载均衡机制，但在实际使用的时候，还是会碰到一些复杂的场景是它不能支持的，比如：

* 接入已有的负载均衡设备
* 多租户网络情况下，容器网络和主机网络是隔离的，这样kube-proxy就不能正常工作

这个时候就可以自定义组件，并代替kube-proxy来做负载均衡。基本的思路是监控kubernetes中service和endpoints的变化，并根据这些变化来配置负载均衡器。比如weave flux、nginx plus、kube2haproxy等。

### Endpoints

有几种情况下需要用到没有selector的service。

* 使用kubernetes集群外部的数据库时
* service中用到了其他namespace或kubernetes集群中的service
* 在kubernetes的工作负载与集群外的后端之间互相迁移

可以这样定义一个没有selector的service。

kind: Service

apiVersion: v1

metadata:

name: my-service

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

定义一个Endpoints来对应该service。

kind: Endpoints

apiVersion: v1

metadata:

name: my-service

subsets:

- addresses:

- ip: 1.2.3.4

ports:

- port: 9376

访问没有selector的service跟访问有selector的service时没有任何区别。

使用kubernetes时有一个很常见的需求，就是当数据库部署在kubernetes集群之外的时候，集群内的service如何访问数据库呢？当然你可以直接使用数据库的IP地址和端口号来直接访问，有没有什么优雅的方式呢？你需要用到ExternalName Service。

kind: Service

apiVersion: v1

metadata:

name: my-service

namespace: prod

spec:

type: ExternalName

externalName: my.database.example.com

ports:

- port: 12345

这个例子中，在kubernetes集群内访问my-service实际上会重定向到my.database.example.com:12345这个地址。

## 自定义 CustomResourceDefinition 拓展

我们知道，Kubernetes 中一切都可视为资源，它提供了很多默认资源类型，如 Pod、Deployment、Service、Volume等一系列资源，能够满足大多数日常系统部署和管理的需求。但是，在一些特殊的需求场景下，这些现有资源类型就满足不了，那么这些就可以抽象为 Kubernetes 的自定义资源，在 Kubernetes 1.7 之后增加了对 CRD 自定义资源二次开发能力来扩展 Kubernetes API，通过 CRD 我们可以向 Kubernetes API 中增加新资源类型，而不需要修改 Kubernetes 源码或创建自定义的 API server，该功能大大提高了 Kubernetes 的扩展能力。它是 (TPR) ThirdPartyResource 的替代者，在 1.9 以上版本 TPR 将被废弃。

自定义资源是对Kubernetes API的扩展，kubernetes中的每个资源都是一个API对象的集合，例如我们在YAML文件里定义的那些spec都是对kubernetes中的资源对象的定义，所有的自定义资源可以跟kubernetes中内建的资源一样使用kubectl操作。

### 自定义示例

resourcedefinition.yaml：

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:

# 名称必须符合下面的格式：<plural>.<group>

name: crontabs.stable.example.com

spec:

# REST API使用的组名称：/apis/<group>/<version>

group: stable.example.com

# REST API使用的版本号：/apis/<group>/<version>

version: v1

# Namespaced或Cluster

scope: Namespaced

names:

# URL中使用的复数名称: /apis/<group>/<version>/<plural>

plural: crontabs

# CLI中使用的单数名称

singular: crontab

# CamelCased格式的单数类型。在清单文件中使用

kind: CronTab

# CLI中使用的资源简称

shortNames:

- ct

创建该CRD：

kubectl create -f resourcedefinition.yaml

访问RESTful API端点如http://172.20.0.113:8080将看到如下API端点已创建：

/apis/stable.example.com/v1/namespaces/\*/crontabs/...

**创建自定义对象**

apiVersion: "stable.example.com/v1"

kind: CronTab

metadata:

name: my-new-cron-object

spec:

cronSpec: "\* \* \* \* /5"

image: my-awesome-cron-image

引用该自定义资源的API创建对象。

**终止器**

可以为自定义对象添加一个终止器，如下所示：

apiVersion: "stable.example.com/v1"

kind: CronTab

metadata:

finalizers:

- finalizer.stable.example.com

删除自定义对象前，异步执行的钩子。对于具有终止器的一个对象，删除请求仅仅是为metadata.deletionTimestamp字段设置一个值，而不是删除它，这将触发监控该对象的控制器执行他们所能处理的任意终止器。

详情参考：[Extend the Kubernetes API with CustomResourceDefinitions](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/)

**自定义控制器**

单纯设置了自定义资源，并没有什么用，只有跟自定义控制器结合起来，才能讲资源对象中的声明式API翻译成用户所期望的状态。自定义控制器可以用来管理任何资源类型，但是一般是跟自定义资源结合使用。

### 官方示例

接下来，我们通过官方示例 [sample-controller](https://github.com/kubernetes/sample-controller) 来演示下如何使用 Kubernetes CRD。

通过该示例 sample-controller，我们可以清楚的了解到：

* 如何通过 CRD 创建一个新的自定义资源类型 Foo
* 如何创建、获取、List 该新资源类型 Foo 的实例
* 如何在资源处理创建、更新、删除事件上设置 Controller

具体部署请参考 k8s 官网：<https://github.com/kubernetes/sample-controller>

## 部署 Docker-Registry

注意：本文档介绍使用 docker 官方的 registry v2 镜像部署私有仓库的步骤，你也可以部署 Harbor 私有仓库（[部署 Harbor 私有仓库](#_部署_Harbor-Registry)）。

本文档讲解部署一个 TLS 加密、HTTP Basic 认证的私有 docker registry 步骤；

示例机器 IP 如下：

docker registry: 192.168.100.249

### 创建 docker registry

创建 registry 使用的 x509 证书

mkdir -p registry/{auth,certs}

cat > registry-csr.json <<EOF

{

"CN": "registry",

"hosts": [

"127.0.0.1",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~",

"~~192.168.100.249~~"

],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

cfssl gencert -ca=/etc/kubernetes/cert/ca.pem \

-ca-key=/etc/kubernetes/cert/ca-key.pem \

-config=/etc/kubernetes/cert/ca-config.json \

-profile=kubernetes registry-csr.json | cfssljson -bare registry

* 这里复用以前创建的 CA 证书和秘钥文件；
* hosts 字段指定 registry 的 NodeIP；

创建 HTTP Baisc 认证文件

docker run --entrypoint htpasswd registry:2 -Bbn study study > registry/auth/htpasswd

cat registry/auth/htpasswd

study:$2y$05$iZaM45Jxlcg0DJKXZMggLOibAsHLGybyU.CgU9AHqWcVDyBjiScN.

配置 registry 参数

cat > config.yml << EOF

# https://docs.docker.com/registry/configuration/#list-of-configuration-options

version: 0.1

log:

level: info

fromatter: text

fields:

service: registry

storage:

filesystem:

rootdirectory: /var/lib/registry

maxthreads: 100

cache:

blobdescriptor: inmemory

delete:

enabled: true

auth:

htpasswd:

realm: basic-realm

path: /auth/htpasswd

http:

addr: 0.0.0.0:8000

headers:

X-Content-Type-Options: [nosniff]

tls:

certificate: /certs/registry.pem

key: /certs/registry-key.pem

health:

storagedriver:

enabled: true

interval: 10s

threshold: 3

EOF

* auth.htpasswd 指定了 HTTP Basic 认证的 token 文件路径；
* http.tls 指定了 registry http 服务器的证书和秘钥文件路径；

创建 docker registry

ssh root@192.168.100.249

docker run -d -p 8000:8000 --privileged \

-v /opt/k8s/registry/auth/:/auth \

-v /opt/k8s/registry/certs:/certs \

-v /opt/k8s/registry/config.yml:/etc/docker/registry/config.yml \

--name registry registry:2

* 执行该 docker run 命令的机器 IP 为 192.168.100.249；

### 向 registry push image

将签署 registry 证书的 CA 证书拷贝到 /etc/docker/certs.d/192.168.100.249:8000 目录下

sudo mkdir -p /etc/docker/certs.d/192.168.100.249:8000

sudo cp /etc/kubernetes/cert/ca.pem /etc/docker/certs.d/192.168.100.249:8000/ca.crt

登陆私有 registry

docker login 192.168.100.249:8000

Username: study

Password: study

Login Succeeded

登陆信息被写入 ~/.docker/config.json 文件

cat ~/.docker/config.json

将本地的 image 打上私有 registry 的 tag

docker tag busybox 192.168.100.249:8000/busybox

docker images |grep busybox

将 image push 到私有 registry

docker push 192.168.100.249:8000/busybox

### 私有 registry 的运维操作

* 查询私有镜像中的 images

curl --user study:study --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/\_catalog

* 查询某个镜像的 tags 列表

curl --user study:study --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/busybox/tags/list

* 获取 image 或 layer 的 digest

向 v2/<repoName>/manifests/<tagName> 发 GET 请求，从响应的头部 Docker-Content-Digest 获取 image digest，从响应的 body 的 fsLayers.blobSum 中获取 layDigests;

注意，必须包含请求头：Accept: application/vnd.docker.distribution.manifest.v2+json：

curl -v -H "Accept: application/vnd.docker.distribution.manifest.v2+json" --user study:study --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/busybox/manifests/latest

* 删除 image

向 /v2/<name>/manifests/<reference> 发送 DELETE 请求，reference 为上一步返回的 Docker-Content-Digest 字段内容：

curl -X DELETE --user study:study --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/busybox/manifests/sha256:68effe31a4ae8312e47f54bec52d1fc925908009ce7e6f734e1b54a4169081c5

* 删除 layer

向 /v2/<name>/blobs/<digest>发送 DELETE 请求，其中 digest 是上一步返回的 fsLayers.blobSum 字段内容：

curl -X DELETE --user study:study --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/busybox/blobs/sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4

curl -X DELETE --cacert /etc/docker/certs.d/192.168.100.249\:8000/ca.crt https://192.168.100.249:8000/v2/busybox/blobs/sha256:04176c8b224aa0eb9942af765f66dae866f436e75acef028fe44b8a98e045515

### 常见问题

* login 失败 416

执行 <http://docs.ceph.com/docs/master/install/install-ceph-gateway/> 里面的 s3 test.py 程序失败：

python s3test.py Traceback (most recent call last): File "s3test.py", line 12, in bucket = conn.create\_bucket('my-new-bucket') File "/usr/lib/python2.7/site-packages/boto/s3/connection.py", line 625, in create\_bucket response.status, response.reason, body) boto.exception.S3ResponseError: S3ResponseError: 416 Requested Range Not Satisfiable

解决版办法：

1. 在管理节点上修改 ceph.conf
2. ceph-deploy config push master node1 node2

For anyone who is hitting this issue set default pg\_num and pgp\_num to lower value(8 for example), or set mon\_max\_pg\_per\_osd to a high value in ceph.conf radosgw-admin doesn' throw proper error when internal pool creation fails, hence the upper level error which is very confusing.

<https://tracker.ceph.com/issues/21497>

* login 失败 503

docker login 192.168.100.249:8000 Username: study Password: Error response from daemon: login attempt to https://192.168.100.249:8000/v2/ failed with status: 503 Service Unavailable

原因： docker run 缺少 --privileged 参数；

## 部署 Harbor-Registry

本文档介绍使用 docker-compose 部署 harbor 私有仓库的步骤，你也可以使用 docker 官方的 registry 镜像部署私有仓库([部署 Docker Registry](#_部署_Docker-Registry))。

### 下载文件

从 [docker compose 发布页面](https://github.com/docker/compose/releases)下载最新的 docker-compose 二进制文件（这里不再重复安装步骤了）

从 [harbor 发布页面](https://github.com/vmware/harbor/releases)下载最新的 harbor 离线安装包

wget --continue https://storage.googleapis.com/harbor-releases/release-1.7.0/harbor-offline-installer-v1.7.0.tgz

tar -xzvf harbor-offline-installer-v1.7.0.tgz

### 创建 harbor nginx 服务器使用的 x509 证书

创建 harbor 证书签名请求：

cd /opt/k8s/work/cert

cat > harbor-csr.json <<EOF

{

"CN": "harbor",

"hosts": [

"127.0.0.1",

"192.168.100.241",

"192.168.100.242",

"192.168.100.243",

"192.168.100.244",

"192.168.100.245",

"~~192.168.100.246~~",

"~~192.168.100.247~~",

"~~192.168.100.248~~",

"~~192.168.100.249~~"

],

"key": {

"algo": "rsa",

"size": 2048

},

"names": [

{

"C": "CN",

"ST": "BeiJing",

"L": "BeiJing",

"O": "k8s",

"OU": "study163"

}

]

}

EOF

* hosts 字段指定授权使用该证书的当前部署节点 IP，如果后续使用域名访问 harbor 则还需要添加域名；

生成 harbor 证书和私钥：

cfssl gencert -ca=/etc/kubernetes/cert/ca.pem \

-ca-key=/etc/kubernetes/cert/ca-key.pem \

-config=/etc/kubernetes/cert/ca-config.json \

-profile=kubernetes harbor-csr.json | cfssljson -bare harbor

sudo mkdir -p /etc/harbor/ssl

sudo mv harbor\*.pem /etc/harbor/ssl

rm harbor.csr harbor-csr.json

### 修改 harbor.cfg 文件

cp harbor.cfg{,.bak}

vim harbor.cfg

diff harbor.cfg{,.bak}

8c8

< hostname = 192.168.100.249

---

> hostname = reg.mydomain.com

12c12

< ui\_url\_protocol = https

---

> ui\_url\_protocol = http

24,25c24,25

< ssl\_cert = /etc/harbor/ssl/harbor.pem

< ssl\_cert\_key = /etc/harbor/ssl/harbor-key.pem

---

> ssl\_cert = /data/cert/server.crt

> ssl\_cert\_key = /data/cert/server.key

cp prepare{,.bak}

vim prepare

diff prepare{,.bak}

453a454

> print("%s %w", args, kw)

490c491

< empty\_subj = "/"

---

> empty\_subj = "/C=/ST=/L=/O=/CN=/"

* 需要修改 prepare 脚本的 empyt\_subj 参数，否则后续 install 时出错退出：

Fail to generate key file: ./common/config/ui/private\_key.pem, cert file: ./common/config/registry/root.crt

参考：<https://github.com/vmware/harbor/issues/2920>

### 加载和启动 harbor 镜像

docker-compose up

✔ ----Harbor has been installed and started successfully.----

Now you should be able to visit the admin portal at https://192.168.100.249.

For more details, please visit https://github.com/vmware/harbor .

### 访问管理界面

确认所有组件都工作正常：

docker-compose ps

Name Command State Ports

-------------------------------------------------------------------------------------------------------------------------------------

harbor-adminserver /harbor/start.sh Up (healthy)

harbor-db /usr/local/bin/docker-entr ... Up (healthy) 3306/tcp

harbor-jobservice /harbor/start.sh Up

harbor-log /bin/sh -c /usr/local/bin/ ... Up (healthy) 127.0.0.1:1514->10514/tcp

harbor-ui /harbor/start.sh Up (healthy)

nginx nginx -g daemon off; Up (healthy) 0.0.0.0:443->443/tcp, 0.0.0.0:4443->4443/tcp, 0.0.0.0:80->80/tcp

redis docker-entrypoint.sh redis ... Up 6379/tcp

registry /entrypoint.sh serve /etc/ ... Up (healthy) 5000/tcp

浏览器访问 https://127.0.0.1:443，用账号 admin 和 harbor.cfg 配置文件中的默认密码 Harbor12345 登陆系统。



### harbor 运行时产生的文件、目录

harbor 将日志打印到 /var/log/harbor 的相关目录下，使用 docker logs XXX 或 docker-compose logs XXX 将看不到容器的日志。

$ # 日志目录

$ ls /var/log/harbor

adminserver.log jobservice.log mysql.log proxy.log registry.log ui.log

$ # 数据目录，包括数据库、镜像仓库

$ ls /data/

ca\_download config database job\_logs registry secretkey

### docker 客户端登陆

将签署 harbor 证书的 CA 证书拷贝到 /etc/docker/certs.d/192.168.100.249 目录下

$ sudo mkdir -p /etc/docker/certs.d/192.168.100.249

$ sudo cp /etc/kubernetes/cert/ca.pem /etc/docker/certs.d/192.168.100.249/ca.crt

登陆 harbor

$ docker login 192.168.100.249

Username: admin

Password:

认证信息自动保存到 ~/.docker/config.json 文件。

### 其它操作

下列操作的工作目录均为 解压离线安装文件后 生成的 harbor 目录。

$ # 停止 harbor

$ docker-compose down -v

$ # 修改配置

$ vim harbor.cfg

$ # 更修改的配置更新到 docker-compose.yml 文件

$ ./prepare

$ sudo chmod -R 666 common ## 防止容器进程没有权限读取生成的配置

$ # 启动 harbor

$ docker-compose up -d

## 清理集群

### 清理 Node 节点

停相关进程：

systemctl stop kubelet kube-proxy flanneld docker kube-nginx

清理文件：

source /opt/k8s/bin/environment.sh

# umount kubelet 和 docker 挂载的目录

mount | grep "${K8S\_DIR}" | awk '{print $3}'|xargs sudo umount

# 删除 kubelet 工作目录

sudo rm -rf ${K8S\_DIR}/kubelet

# 删除 docker 工作目录

sudo rm -rf ${DOCKER\_DIR}

# 删除 flanneld 写入的网络配置文件

sudo rm -rf /var/run/flannel/

# 删除 docker 的一些运行文件

sudo rm -rf /var/run/docker/

# 删除 systemd unit 文件

sudo rm -rf /etc/systemd/system/{kubelet,docker,flanneld,kube-nginx}.service

# 删除程序文件

sudo rm -rf /opt/k8s/bin/\*

# 删除证书文件

sudo rm -rf /etc/flanneld/cert /etc/kubernetes/cer

清理 kube-proxy 和 docker 创建的 iptables：

iptables -F && sudo iptables -X && sudo iptables -F -t nat && sudo iptables -X -t nat

删除 flanneld 和 docker 创建的网桥：

ip link del flannel.1

ip link del docker0

### 清理 Master 节点

停相关进程：

systemctl stop kube-apiserver kube-controller-manager kube-scheduler kube-nginx

清理文件：

# 删除 kube-apiserver 工作目录

rm -rf /var/run/kubernetes

# 删除 systemd unit 文件

rm -rf /etc/systemd/system/{kube-apiserver,kube-controller-manager,kube-scheduler}.service

# 删除程序文件

rm -rf /opt/k8s/bin/{kube-apiserver,kube-controller-manager,kube-scheduler}

# 删除证书文件

rm -rf /etc/flanneld/cert /etc/kubernetes/cert

### 清理 etcd 集群

停相关进程：

systemctl stop etcd

清理文件：

source /opt/k8s/bin/environment.sh

# 删除 etcd 的工作目录和数据目录

sudo rm -rf ${ETCD\_DATA\_DIR} ${ETCD\_WAL\_DIR}

# 删除 systemd unit 文件

sudo rm -rf /etc/systemd/system/etcd.service

# 删除程序文件

sudo rm -rf /opt/k8s/bin/etcd

# 删除 x509 证书文件

sudo rm -rf /etc/etcd/cert/\*

# kubernetes 最佳实践

本章节主要讲解 k8s 在生产级环境下的使用及网易对 k8s 的探索和实践。

## k8s 可视化自动化工具

参考

部署集群插件 dashboard 插件

## K8S 日志收集

参考

部署集群插件 elk 插件

## 结合 K8S 的系统整体监控方案

参考

部署集群插件 heapster 插件

## 生产级的 kubernetes 简化安装工具 kubeadm

[kubeadm](https://www.kubernetes.org.cn/tags/kubeadm)是Kubernetes官方提供的用于快速安装Kubernetes集群的工具，伴随Kubernetes每个版本的发布都会同步更新，kubeadm会对集群配置方面的一些实践做调整，通过实验kubeadm可以学习到Kubernetes官方在集群配置上一些新的最佳实践。

在Kubernetes的文档[Creating a single master cluster with kubeadm](https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/)中已经给出了目前kubeadm的主要特性已经处于beta状态了，在2018年将进入GA状态，说明kubeadm离可以在生产环境中使用的距离越来越近了。

官网参考资料：

<https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/>

<https://kubernetes.io/zh/docs/setup/independent/install-kubeadm/>

### 准备

#### 系统配置

在安装之前，需要先做如下准备。两台CentOS 7.x主机如下：

cat /etc/hosts

192.168.100.243 node1

192.168.100.244 node2

执行前面章节的 [集群环境搭建 – 系统初始化和全局变量](#_系统初始化和全局变量) 配置步骤

#### 安装docker

Kubernetes从1.6开始使用CRI(Container Runtime Interface)容器运行时接口。默认的容器运行时仍然是Docker，使用的是kubelet中内置dockershim CRI实现。

Kubernetes 1.12已经针对Docker的1.11.1, 1.12.1, 1.13.1, 17.03, 17.06, 17.09, 18.06等版本做了验证，需要注意Kubernetes 1.12最低支持的Docker版本是1.11.1。 我们这里在各节点安装docker的18.09.0版本。

在测试或开发环境中 Docker 官方为了简化安装流程，提供了一套便捷的安装脚本，CentOS 系统上可以使用这套脚本安装：

curl -fsSL get.docker.com -o get-docker.sh

sudo sh get-docker.sh --mirror Aliyun

执行这个命令后，脚本就会自动的将一切准备工作做好，并且把 Docker CE 的 Edge 版本安装在系统中。

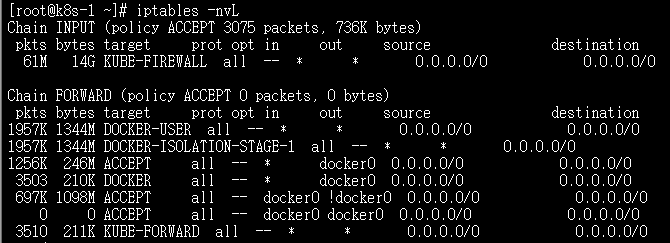
启动docker

sudo systemctl enable docker

sudo systemctl start docker

确认一下iptables filter表中FOWARD链的默认策略(pllicy)为ACCEPT。

iptables -nvL



如果不是，请参考集群环境搭建中的 [关闭防火墙](#_关闭防火墙) 步骤。

Docker从1.13版本开始调整了默认的防火墙规则，禁用了iptables filter表中FOWARD链，这样会引起Kubernetes集群中跨Node的Pod无法通信。

### 使用kubeadm部署kubernetes

#### 安装kubeadm、kubelet和kubectl

需要在每台机器上都安装以下的软件包：

* kubeadm: 用来初始化集群的指令。
* kubelet: 在集群中的每个节点上用来启动 pod 和 container 等。
* kubectl: 用来与集群通信的命令行工具。

替换阿里云的源安装kubernetes.repo

cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86\_64/

enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg

https://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg

EOF

yum makecache

查看可用版本

yum list --showduplicates | grep 'kubeadm\|kubectl\|kubelet'

安装 kubeadm1.14.1, kubelet1.14.1 和 kubectl1.14.1

yum install -y kubelet-1.14.1 kubeadm-1.14.1 kubectl-1.14.1 --disableexcludes=kubernetes

从安装结果可以看出还安装了cri-tools, kubernetes-cni, socat三个依赖：

* socat是kubelet的依赖
* cri-tools是CRI(Container Runtime Interface)容器运行时接口的命令行工具

运行kubelet –help可以看到原来kubelet的绝大多数命令行flag参数都被DEPRECATED了。

而官方推荐我们使用–config指定配置文件，并在配置文件中指定原来这些flag所配置的内容。具体内容可以查看这里[Set Kubelet parameters via a config file](https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/)。这也是Kubernetes为了支持动态Kubelet配置（Dynamic Kubelet Configuration）才这么做的，参考[Reconfigure a Node’s Kubelet in a Live Cluster](https://kubernetes.io/docs/tasks/administer-cluster/reconfigure-kubelet/)。

kubelet的配置文件必须是json或yaml格式，具体可查看[这里](https://github.com/kubernetes/kubernetes/blob/release-1.10/pkg/kubelet/apis/kubeletconfig/v1beta1/types.go)。

查看/etc/systemd/system/kubelet.service.d/10-kubeadm.conf，看到了下面的内容：

# Note: This dropin only works with kubeadm and kubelet v1.11+

[Service]

Environment="KUBELET\_KUBECONFIG\_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf

--kubeconfig=/etc/kubernetes/kubelet.conf"

Environment="KUBELET\_CONFIG\_ARGS=--config=**/var/lib/kubelet/config.yaml**"

# This is a file that "kubeadm init" and "kubeadm join" generates at runtime, populating the KUBELET\_KUBEADM\_ARGS variable dynamically

EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env

# This is a file that the user can use for overrides of the kubelet args as a last resort. Preferably, the user should use

# the .NodeRegistration.KubeletExtraArgs object in the configuration files instead. KUBELET\_EXTRA\_ARGS should be sourced from this file.

EnvironmentFile=-/etc/sysconfig/kubelet

ExecStart=

ExecStart=/usr/bin/kubelet $KUBELET\_KUBECONFIG\_ARGS $KUBELET\_CONFIG\_ARGS $KUBELET\_KUBEADM\_ARGS $KUBELET\_EXTRA\_ARGS

上面显示kubeadm部署的kubelet的配置文件–config=/var/lib/kubelet/config.yaml，实际去查看/var/lib/kubelet和这个config.yaml的配置文件都没有被创建。 可以猜想肯定是运行kubeadm初始化集群时会自动生成这个配置文件，而如果我们不关闭Swap的话，第一次初始化集群肯定会失败的。

所以还是老老实实的回到使用kubelet的启动参数–fail-swap-on=false去掉必须关闭Swap的限制。

查看 kubelet 安装了哪些文件？

$ rpm -ql kubelet

/etc/kubernetes/manifests # 清单目录

/etc/sysconfig/kubelet # 配置文件

/etc/systemd/system/kubelet.service # unit file

/usr/bin/kubelet # 主程序

修改/etc/sysconfig/kubelet，加入：

sed -i "s/KUBELET\_EXTRA\_ARGS=/KUBELET\_EXTRA\_ARGS=\"--fail-swap-on=false\"/" /etc/sysconfig/kubelet

cat /etc/sysconfig/kubelet

KUBELET\_EXTRA\_ARGS="--fail-swap-on=false"

#### 使用kubeadm init初始化集群

在各节点开机启动kubelet服务：

systemctl enable kubelet.service

如果是第一次运行，下载 Docker 镜像后再运行 kubeadm init会比较慢，也可以通过 kubeadm config images pull 命令先将镜像下载到本地。kubeadm init 首先会执行一系列的运行前检查来确保机器满足运行 Kubernetes 的条件。这些检查会抛出警告并在发现错误的时候终止整个初始化进程。 然后 kubeadm init 会下载并安装集群的控制面组件，这可能会花费几分钟时间。

命令执行完以后，会自动启动 kubelet Docker 镜像，由于不能连接到k8s 的镜像网站，所以需要从老师的hub.docker 上拉取再改名字。

docker pull xiaochunping/kube-apiserver:v1.14.1

docker pull xiaochunping/kube-controller-manager:v1.14.1

docker pull xiaochunping/kube-scheduler:v1.14.1

docker pull xiaochunping/kube-proxy:v1.14.1

docker pull xiaochunping/pause:3.1

docker pull xiaochunping/etcd:3.3.10

docker pull xiaochunping/coredns:1.3.1

docker tag xiaochunping/kube-apiserver:v1.14.1 k8s.gcr.io/kube-apiserver:v1.14.1

docker tag xiaochunping/kube-controller-manager:v1.14.1 k8s.gcr.io/kube-controller-manager:v1.14.1

docker tag xiaochunping/kube-scheduler:v1.14.1 k8s.gcr.io/kube-scheduler:v1.14.1

docker tag xiaochunping/kube-proxy:v1.14.1 k8s.gcr.io/kube-proxy:v1.14.1

docker tag xiaochunping/pause:3.1 k8s.gcr.io/pause:3.1

docker tag xiaochunping/etcd:3.3.10 k8s.gcr.io/etcd:3.3.10

docker tag xiaochunping/coredns:1.3.1 k8s.gcr.io/coredns:1.3.1

选择node1作为Master Node，在node1上执行下面的命令：

kubeadm init \

--kubernetes-version=v1.14.1 \

--pod-network-cidr=10.244.0.0/16 \

--apiserver-advertise-address=192.168.100.243 \

--ignore-preflight-errors=Swap

因为我们选择flannel作为Pod网络插件，所以上面的命令指定–pod-network-cidr=10.244.0.0/16。

执行初始化，根据输出的内容基本上可以看出手动初始化安装一个Kubernetes集群所需要的关键步骤。

其中有以下关键内容：

* [kubelet] 生成kubelet的配置文件”/var/lib/kubelet/config.yaml”
* [certificates]生成相关的各种证书
* [kubeconfig]生成相关的kubeconfig文件
* [bootstraptoken]生成token记录下来，后边使用kubeadm join往集群中添加节点时会用到
* 下面的命令是配置常规用户如何使用kubectl访问集群：

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

* 最后给出了将节点加入集群的命令kubeadm join 192.168.100.243:6443 –token zalj3i.q831ehufqb98d1ic –discovery-token-ca-cert-hash sha256:6ee48b19ba61a2dda77f6b60687c5fd11072ab898cfdfef32a68821d1dbe8efa

请备份好 kubeadm init 输出中的 kubeadm join 命令，因为您会需要这个命令来给集群添加节点。

查看一下集群状态：

kubectl get cs,nodes

确认个组件都处于healthy状态。

集群初始化如果遇到问题，可以使用下面的命令进行清理：

kubeadm reset

ifconfig cni0 down

ip link delete cni0

ifconfig flannel.1 down

ip link delete flannel.1

rm -rf /var/lib/cni/

#### 部署 Flannel

接下来安装flannel network add-on：

mkdir -p ~/k8s/

cd ~/k8s

# 下载配置文件

wget <https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml>

#启动 flannel 服务

kubectl apply -f kube-flannel.yml

这里注意kube-flannel.yml这个文件里的flannel的镜像是0.11.0，quay.io/coreos/flannel:v0.11.0-amd64

如果Node有多个网卡的话，参考[flannel issues 39701](https://github.com/kubernetes/kubernetes/issues/39701)，目前需要在kube-flannel.yml中使用–iface参数指定集群主机内网网卡的名称，否则可能会出现dns无法解析。需要将kube-flannel.yml下载到本地，flanneld启动参数加上–iface=<iface-name>

......

containers:

- name: kube-flannel

image: quay.io/coreos/flannel:v0.11.0-amd64

command:

- /opt/bin/flanneld

args:

- --ip-masq

- --kube-subnet-mgr

- --iface=eth1

......

本次按上面的步骤部署flannel，发现没有效果，查看一下集群中的daemonset：

kubectl get ds -l app=flannel -n kube-system

使用kubectl get pod –all-namespaces -o wide确保所有的Pod都处于Running状态。

kubectl get pod --all-namespaces -o wide

查看节点状态

kubectl get cs,node

确保node1的STATUS已经是Ready。

#### 向Kubernetes集群中添加Node节点

将新节点添加到集群为每一台机器上执行以下操作:

kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash>

如果忘记了 Master 的 Token，可以在 Master 上输入以下命令查看：

kubeadm token list

默认情况下 Token 过期是时间是24小时，如果 Token 过期以后，可以输入以下命令，生成新的 Token

kubeadm token create

--discovery-token-ca-cert-hash 的查看方法，在 Master 运行以下命令查看

openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.\* //'

接下来我们开始正式将 Node 节点加入到 Master ，输入以下命令

kubeadm join 192.168.100.243:6443 --token cvj70p.iv3k58e0fb6lrn9y \

--discovery-token-ca-cert-hash sha256:fae63a0a1ef01058f6173bfc8983fc3c976b6f9f6d55fd8c4ab2140c7596c4fc

在 Master 查看节点加入情况，其他节点加入以后：

kubectl get nodes

在 Node 节点上查看 Docker 容器运行状态

docker ps

确保node节点上docker启动了flanneld、kube-proxy、pause容器。

最后在 Master 节点上查看 Pod 运行状态，可以的看到 kube-flannel 和 kube-flannel 在每一个 Node 节点上都有运行

kubectl get pods -n kube-system -o wide

#### 从集群中删除Node

如果需要从集群中移除node2这个Node执行下面的命令：

在master节点上执行：

kubectl drain node2 --delete-local-data --force --ignore-daemonsets

kubectl delete node node2

在 Node 被删除,需要重启所有 kubeadm 安装状态:：

kubeadm reset

ifconfig cni0 down

ip link delete cni0

ifconfig flannel.1 down

ip link delete flannel.1

rm -rf /var/lib/cni/

出于安全原因，您的群集不会在主服务器上安排pod。如果您希望能够在主服务器上安排pod，例如对于用于开发的单机Kubernetes集群，可以运行以下命令

# 所有主服务器都安排 pod

kubectl taint nodes --all node-role.kubernetes.io/master-

# 指定 hadoop1 服务器都安排 pod

kubectl taint nodes hadoop1 node-role.kubernetes.io/master-

### 部署whoami

whoami 是一个简单的HTTP docker服务，用于打印容器ID

在 Master 运行部署 Whoami

kubectl create deployment whoami --image=idoall/whoami

查看 Whoami 部署状态

kubectl get deployments

查看 Whoami 的部署信息

kubectl describe deployment whoami

查看 Whoami 的容器日志

kubectl describe po whoami

为 Whoami 扩展端口；创建一个可以通过互联网访问的 Whoami 容器

kubectl create service nodeport whoami --tcp=80:80

上面的命令将在主机上为 Whoami 部署创建面向公众的服务。

由于这是一个节点端口部署，因此 kubernetes 会将此服务分配给32000+范围内的主机上的端口。

查看当前的服务状态

kubectl get svc,pods -o wide

上面的服务可以看到 Whoami 运行在 32707 端口，通过 http://10.0.0.101:32707 访问

测试 Whoami 服务是否运行正常

curl hadoop2:32707

扩展部署应用

kubectl scale --replicas=5 deployment/whoami

查看扩展后的结果,可以看到 Whoami 在 hadoop1、hadoop2、hadoop3上面都有部署

kubectl scale --replicas=5 deployment/whoami

测试扩展后的结果

curl c0:32707

ClusterIP 模式会提供一个集群内部的虚拟IP（与Pod不在同一网段)，以供集群内部的 Pod 之间通信使用。

删除 Whoami 部署

kubectl delete deployment whoami

kubectl get deployments

### 部署Dashboard

从版本1.7开始，仪表板不再具有默认授予的完全管理员权限。所有权限都被撤销，并且只授予了使 Dashboard 工作所需的最小权限。

我们使用官方提供的 v1.12.3 版本的配置文件；使用前面章节中配置dashboard 的配置：

cd /opt/k8s/work/kubernetes/cluster/addons/dashboard

复制所有的yaml文件。

创建dashboard服务

kubectl apply -f kubernetes-dashboard.yaml

注意：**前面章节中的配置中已经添加了NodePort**；NodePort 模式下 Kubernetes 将会在每个 Node 上打开一个端口并且每个 Node 的端口都是一样的，通过 <NodeIP>:NodePort 的方式 Kubernetes 集群外部的程序可以访问 Service。

通过下面的命令，可以查看到，服务已在服务器的端口31230（HTTPS）上公开。现在，您可以从以下浏览器访问它：https://10.0.0.100:30779。

kubectl -n kube-system get service kubernetes-dashboard

查看 Dashboard 状态

kubectl get pods --all-namespaces | grep kubernetes-dashboard

查看 Dashboard 日志

kubectl logs -f kubernetes-dashboard-57df4db6b-6scvx -n kube-system

**创建访问Dashboard的token**

需要创建一个 Admin 用户并授予 Admin 角色绑定，使用下面的 yaml文件 创建 admin 用户并赋予管理员权限，然后可以通过 Token 访问 kubernetes

您可以通过创建以下 ClusterRoleBinding 来授予 Dashboard 服务 Admin 管理员权限。

kubectl get secret -o wide --all-namespaces | grep kubernetes-dashboard-token

kubectl -n kube-system describe secret kubernetes-dashboard-token-fbl6l

上面的token 就是登录用的密码

也可以通过 jsonpath 直接获取 token

kubectl -n kube-system get secret kubernetes-dashboard-token-fbl6l -o jsonpath={.data.token}|base64 -d

也可以使用下面的命令，直接获取 kubernetes-dashboard-token 的值，然后直接打印输出

k8tokenvalue=`kubectl get secret -o wide --all-namespaces | grep kubernetes-dashboard-token | awk '{print $2}'`;kubectl -n kube-system get secret $k8tokenvalue -o jsonpath={.data.token}|base64 -d | awk '{print $1}'

### 部署Heaster

参照前面章节中的 [部署集群插件 – heapster插件](#_heapster_插件)

## kubectl

kubectl是用于针对Kubernetes集群运行命令的命令行接口。本章节涵盖kubectl语法，描述命令操作，并提供常见的示例。

官网最详细的命令使用文档（强烈推荐大家学习）：

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>

中文文档：

<http://docs.kubernetes.org.cn/683.html>

### 语法

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

其中command，TYPE，NAME，和flags分别是：

**command**: 指定要在一个或多个资源进行操作，例如create，get，describe，delete。

**TYPE**：指定资源类型。资源类型区分大小写，您可以指定单数，复数或缩写形式。例如，以下命令产生相同的输出：

$ kubectl get pod pod1 $ kubectl get pods pod1 $ kubectl get po pod1

**NAME**：指定资源的名称。名称区分大小写。如果省略名称，则会显示所有资源的详细信息,比如$ kubectl get pods。

在多个资源上执行操作时，可以按类型和名称指定每个资源，或指定一个或多个文件：

按类型和名称指定资源：

kubectl get pod example-pod1 example-pod2

kubectl get pod -f ./pod.yaml

**flags**：指定可选标志。例如，您可以使用-s或--serverflags来指定Kubernetes API服务器的地址和端口。 重要提示：从命令行指定的标志将覆盖默认值和任何相应的环境变量。

### 操作

下表包括所有kubectl操作的简短描述和一般语法：

| Operation | Syntax | Description |
| --- | --- | --- |
| **annotate** | `kubectl annotate (-f FILENAME | TYPE NAME |
| **api-versions** | **kubectl api-versions [flags]** | 列出可用的API版本。 |
| **apply** | **kubectl apply -f FILENAME [flags]** | 对文件或标准输入流更改资源应用配置。 |
| **attach** | **kubectl attach POD -c CONTAINER [-i] [-t] [flags]** | attach 到正在运行的容器来查看输出流或与容器（stdin）进行交互。 | |
| **autoscale** | `kubectl autoscale (-f FILENAME | TYPE NAME |
| **cluster-info** | **kubectl cluster-info [flags]** | 显示有关集群中master节点和服务的端点信息。 |
| **config** | **kubectl config SUBCOMMAND [flags]** | 修改kubeconfig文件。有关详细信息，请参阅各个子命令。 |
| **create** | **kubectl create -f FILENAME [flags]** | 从文件或stdin创建一个或多个资源。 |
| **delete** | `kubectl delete (-f FILENAME | TYPE [NAME |
| **describe** | `kubectl describe (-f FILENAME | TYPE [NAME\_PREFIX |
| **edit** | `kubectl edit (-f FILENAME | TYPE NAME |
| **exec** | **kubectl exec POD [-c CONTAINER] [-i] [-t] [flags] [-- COMMAND [args...]]** | 对pod中的容器执行命令 |
| **explain** | **kubectl explain [--include-extended-apis=true] [--recursive=false] [flags]** | 获取各种资源的文档。例如 pods, nodes, services 等. |
| **expose** | `kubectl expose (-f FILENAME | TYPE NAME |
| **get** | `kubectl get (-f FILENAME | TYPE [NAME |
| **label** | `kubectl label (-f FILENAME | TYPE NAME |
| **logs** | **kubectl logs POD [-c CONTAINER] [--follow] [flags]** | 在pod的容器中打印日志。 |
| **patch** | `kubectl patch (-f FILENAME | TYPE NAME |
| **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]** | 运行一个代理到Kubernetes API服务器。 |
| **replace** | **kubectl replace -f FILENAME** | 从文件或stdin替换资源。 |
| **rolling-update** | `kubectl rolling-update OLD\_CONTROLLER\_NAME ([NEW\_CONTROLLER\_NAME] –image=NEW\_CONTAINER\_IMAGE | -f NEW\_CONTROLLER\_SPEC) [flags]` |
| **run** | **kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [flags]** | 在集群上运行指定的镜像。 |
| **scale** | `kubectl scale (-f FILENAME | TYPE NAME |
| **stop** | **kubectl stop** | 已弃用: 相应的, 请查看 **kubectl delete**. |
| **version** | **kubectl version [--client] [flags]** | 显示在客户端和服务器上运行的Kubernetes版本。 |

有关命令操作的更多信息，请参阅[kubectl](https://kubernetes.io/docs/user-guide/kubectl)参考文档。

### 资源类型

下表包括所有支持的资源类型及其缩写别名的列表：

| 资源类型 | 缩写别名 |
| --- | --- |
| **apiservices** |  |
| **certificatesigningrequests** | **csr** |
| **clusters** |  |
| **clusterrolebindings** |  |
| **clusterroles** |  |
| **componentstatuses** | **cs** |
| **configmaps** | **cm** |
| **controllerrevisions** |  |
| **cronjobs** |  |
| **customresourcedefinition** | **crd**, **crds** |
| **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** |  |

### 选项

--alsologtostderr 同时输出日志到标准错误控制台和文件

--as string 以指定用户执行操作

--as-group stringArray 模拟操作的组，可以使用这个标识来指定多个组。

--cache-dir string 默认 HTTP 缓存目录（默认值 "/home/username/.kube/http-cache" ）

--certificate-authority string 用于进行认证授权的 .cert 文件路径

--client-certificate string TLS 使用的客户端证书路径

--client-key string TLS 使用的客户端密钥文件路径

--cluster string 指定要使用的 kubeconfig 文件中集群名

--context string 指定要使用的 kubeconfig 文件中上下文

-h, --help kubectl 帮助

--insecure-skip-tls-verify 值为 true，则不会检查服务器的证书的有效性。 这将使您的HTTPS连接不安全

--kubeconfig string CLI 请求使用的 kubeconfig 配置文件路径。

--log-backtrace-at traceLocation 当日志长度超出规定的行数时，忽略堆栈信息（默认值：0）

--log-dir string 如果不为空，则将日志文件写入此目录

--logtostderr 日志输出到标准错误控制台而不输出到文件

--match-server-version 要求客户端版本和服务端版本相匹配

-n, --namespace string 如果存在，CLI 请求将使用此命名空间

--request-timeout string 放弃一个简单服务请求前的等待时间，非零值需要包含相应时间单位(例如：1s, 2m, 3h)。零值则认为不做超时请求。 (默认值 "0")

-s, --server string Kubernetes API server 的地址和端口

--stderrthreshold severity 等于或高于此阈值的日志将输出标准错误控制台（默认值2）

--token string 用于 API server 进行身份认证的承载令牌

--user string 指定使用的 kubeconfig 配置文件中的用户名

-v, --v Level 指定输出日志的日志级别

--vmodule moduleSpec 指定输出日志的模块，格式如下：pattern=N，使用逗号分隔

### 输出选项

请使用以下部分查看如何格式化或排序某些命令的输出的信息，有关哪些命令支持各种输出选项的详细信息，请参阅kubectl参考文档。

#### 格式化输出

所有kubectl命令的默认输出格式是可读的纯文本格式。要以特定格式将详细信息输出到终端窗口，您可以将一个-o或多个-output标志添加到支持的kubectl命令中。

语法

kubectl [command] [TYPE] [NAME] -o=<output\_format>

根据kubectl操作，支持以下输出格式:

| 输出格式 | 描述 |
| --- | --- |
| **-o=custom-columns=<spec>** | 输入使用逗号分隔的列表打印表格 [custom columns](https://kubernetes.io/zh/docs/user-guide/kubectl-overview/" \l "custom-columns)。 |
| **-o=custom-columns-file=<filename>** | 使用文件中的[自定义列模板](https://kubernetes.io/zh/docs/user-guide/kubectl-overview/" \l "custom-columns)打印表**<filename>**。 |
| **-o=json** | 输出JSON格式的API对象。 |
| **-o=jsonpath=<template>** | 打印在[jsonpath](https://kubernetes.io/docs/user-guide/jsonpath)表达式中定义的字段 |
| **-o=jsonpath-file=<filename>** | 打印由文件中的[jsonpath](https://kubernetes.io/docs/user-guide/jsonpath)表达式定义的字段**<filename>**。 |
| **-o=name** | 仅打印资源名称，没有其他的。 |
| **-o=wide** | 以纯文本格式输出任何附加信息。对于pod，包括节点名称。 |
| **-o=yaml** | 输出YAML格式的API对象。 |

**例子**

在此示例中，以下命令将单个pod的详细信息作为YAML格式化对象输出： $ kubectl get pod web-pod-13je7 -o=yaml

#### 排序列表对象

要将对象输出到终端窗口中的排序列表，可以将该--sort-by标志添加到支持的kubectl命令中。通过使用--sort-by标志指定任何数字或字符串字段来对对象进行排序。要指定一个字段，请使用jsonpath表达式。

语法

kubectl [command] [TYPE] [NAME] --sort-by=<jsonpath\_exp>

例子

打印按名称排序的列表，请运行：

$ kubectl get pods --sort-by=.metadata.name

### 示例：常用操作

使用以下一组示例来帮助您熟悉运行常用kubectl操作：

// 使用在example-service.yaml中的定义创建一个service.

$ kubectl create -f example-service.yaml

// 使用在example-controller.yaml中的定义创建一个replication controller.

$ kubectl create -f example-controller.yaml

// 使用在<directory>目录下的any .yaml, .yml, or .json文件创建对象.

$ kubectl create -f <directory>

kubectl get - 列出一个或更多资源.

// 使用文本格式列出所有的pods.

$ kubectl get pods

// 使用文本格式列出所有的信息，包含一些额外的信息(比如节点名称).

$ kubectl get pods -o wide

// 使用文本格式列出指定名称的replicationcontroller. 注: 你可以缩短 'replicationcontroller' 资源类型使用别名 'rc'.

$ kubectl get replicationcontroller <rc-name>

// 使用文本格式列出所有的rc,services.

$ kubectl get rc,services

`kubectl describe` - 显示一个或多个资源的详情状态.

// 显示指定节点名称的详情信息.

$ kubectl describe nodes <node-name>

// 显示指定pods名称的详情信息.

$ kubectl describe pods/<pod-name>

// 显示所有被名称为<rc-name>的replication controller管理的所有pods的详情信息.

// 请记住: 任何被replication controller的pod的名称前缀为replication controller的名称.

$ kubectl describe pods <rc-name>

kubectl delete - 从文件, stdin,或者指定的label selectors, 名称,资源选择器, 或者资源去删除资源.

// 通过 pod.yaml 文件中的资源类型和名称删除一个pod.

$ kubectl delete -f pod.yaml

// 删除所有label名称为name=<label-name>的pods和services.

$ kubectl delete pods,services -l name=<label-name>

// 删除所有pods.

$ kubectl delete pods --all

kubectl exec - 针对pod中的某个容器执行命令.

// 在名称为<pod-name>的pod允许 'date' 命令获得输出. 默认返回的是pod中第一个容器的终端.

$ kubectl exec <pod-name> date

// 在<pod-name>的pod中的<container-name>容器中运行'date'获取输出.

$ kubectl exec <pod-name> -c <container-name> date

// 从名称为<pod-name>的pod获取一个交互的终端和运行/bin/bash. 默认返回的是pod中第一个容器的终端.

$ kubectl exec -ti <pod-name> /bin/bash

kubectl logs - 输出一个pod的容器日志.

// 从名称为<pod-name>的pod返回日志快照.

$ kubectl logs <pod-name>

// 从名称为<pod-name>的pod中获取日志流. 这个和Linux命令'tail -f'相似.

$ kubectl logs -f <pod-name>

### 示例：Docker用户使用kubectl命令指南

本小节，我们将向 docker-cli 用户介绍 Kubernetes 命令行如何与 api 进行交互。该命令行工具 — kubectl，被设计成 docker-cli 用户所熟悉的样子，但是它们之间又存在一些必要的差异。该文档将向您展示每个 docker 子命令和 kubectl 与其等效的命令。

#### docker run

如何运行一个 nginx Deployment 并将其暴露出来？ 查看 [kubectl run](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "run) 。

使用 docker 命令：

docker run -d --restart=always -e DOMAIN=cluster --name nginx-app -p 80:80 nginx

docker ps # 列表docker运行的容器

使用 kubectl 命令：

# 启动运行 nginx 的 pod

kubectl run --image=nginx nginx-app --port=80 --env="DOMAIN=cluster"

在 1.2 及以上版本的 Kubernetes 集群中，使用kubectl run 命令将创建一个名为 “nginx-app” 的 Deployment。如果您运行的是老版本，将会创建一个 replication controller。 如果您想沿用旧的行为，使用 --generation=run/v1 参数，这样就会创建 replication controller。

查看 kubectl run 获取更多详细信息。

# 通过服务暴露端口

kubectl expose deployment nginx-app --port=80 --name=nginx-http

在 kubectl 命令中，我们创建了一个 Deployment，这将保证有 N 个运行 nginx 的 pod（N 代表 spec 中声明的 replica 数，默认为 1）。我们还创建了一个 service，使用 selector 匹配具有相应的 selector 的 Deployment。查看 [快速开始](https://kubernetes.io/docs/user-guide/quick-start) 获取更多信息。

默认情况下镜像会在后台运行，与docker run -d ... 类似，如果您想在前台运行，使用：

kubectl run [-i] [--tty] --attach <name> --image=<image>

与 docker run ... 不同的是，如果指定了 --attach ，我们将连接到 stdin，stdout 和 stderr，而不能控制具体连接到哪个输出流（docker -a ...）。

因为我们使用 Deployment 启动了容器，如果您终止了连接到的进程（例如 ctrl-c），容器将会重启，这跟 docker run -it 不同。 如果想销毁该 Deployment（和它的 pod），您需要运行 kubectl delete deployment <name>。

#### docker ps

如何列出哪些正在运行？查看 [kubectl get](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "get)。

使用 docker 命令：

docker ps

使用 kubectl 命令：

kubectl get po

#### docker attach

如何连接到已经运行在容器中的进程？查看 [kubectl attach](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "attach)。

使用 docker 命令：

docker ps

docker attach a9ec34d98787

使用 kubectl 命令：

kubectl get pods

kubectl attach -it nginx-app-5jyvm

#### docker exec

如何在容器中执行命令？查看 [kubectl exec](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "exec)。

使用 docker 命令：

docker ps

docker exec a9ec34d98787 cat /etc/hostname

使用 kubectl 命令：

kubectl get po

kubectl exec nginx-app-5jyvm -- cat /etc/hostname

执行交互式命令怎么办？

使用 docker 命令：

docker exec -ti a9ec34d98787 /bin/sh

使用 kubectl 命令：

kubectl exec -ti nginx-app-5jyvm -- /bin/sh

更多信息请查看[获取运行中容器的 Shell 环境](https://kubernetes.io/docs/tasks/kubectl/get-shell-running-container/)。

#### docker logs

如何查看运行中进程的 stdout/stderr？查看 [kubectl logs](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "logs)。

使用 docker 命令：

docker logs -f a9e

使用 kubectl 命令：

kubectl logs -f nginx-app-zibvs

现在是时候提一下 pod 和容器之间的细微差别了；默认情况下如果 pod 中的进程退出 pod 也不会终止，相反它将会重启该进程。这类似于 docker run 时的 --restart=always 选项， 这是主要差别。在 docker 中，进程的每个调用的输出都是被连接起来的，但是对于 kubernetes，每个调用都是分开的。要查看以前在 kubernetes 中执行的输出，请执行以下操作：

kubectl logs --previous nginx-app-zibvs

查看[记录和监控集群活动](https://kubernetes.io/docs/concepts/cluster-administration/logging/)获取更多信息。

#### docker stop 和 docker rm

如何停止和删除运行中的进程？查看 [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "delete)。

使用 docker 命令：

docker ps

docker stop a9ec34d98787

docker rm a9ec34d98787

使用 kubectl 命令：

kubectl get deployment nginx-app

kubectl get po -l run=nginx-app

kubectl delete deployment nginx-app

kubectl get po -l run=nginx-app

请注意，我们不直接删除 pod。使用 kubectl 命令，我们要删除拥有该 pod 的 Deployment。如果我们直接删除 pod，Deployment 将会重新创建该 pod。

#### docker login

在 kubectl 中没有对 docker login 的直接模拟。如果您有兴趣在私有镜像仓库中使用 Kubernetes，请参阅[使用私有镜像仓库](https://kubernetes.io/docs/concepts/containers/images/" \l "using-a-private-registry)。

#### docker version

如何查看客户端和服务端的版本？查看 [kubectl version](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "version)。

使用 docker 命令：

docker version

使用 kubectl 命令：

kubectl version

#### docker info

如何获取有关环境和配置的各种信息？查看 [kubectl cluster-info](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/" \l "cluster-info)。

使用 docker 命令：

docker info

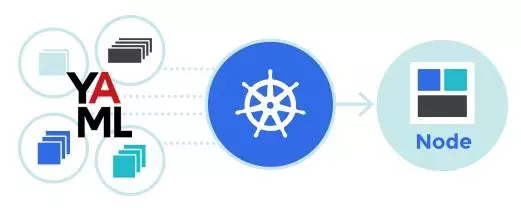
使用 kubectl 命令：

kubectl cluster-info

## YAML资源描述文件

生活中，随处可见，几乎每一款产品都会附带一份说明书，说明书可以记录产品的使用方法，也可以记录产品的配方。有了说明书，我们完全可以窥探一款产品的全貌。

在 K8S 中，yaml 配置文件就是 K8S 资源对象的说明书，定义了对象包含的元素及采取的动作，每种对象都可以通过 yaml 配置文件来创建。



### yaml是什么

yaml 是一种用来写配置文件的语言，没错，它是一门语言。如果你用过 json，那么对它就不会陌生，yaml 又被称为是 json 的超集，使用起来比 json 更方便。

结构上它有两种可选的类型：Lists 和 Maps。

List 用 -（破折号） 来定义每一项，Map 则是一个 key:value 的键值对来表示。如下是一个 json 文件到 yaml 文件的转换：

json:

{

"apiVersion": "v1",

"kind": "Pod",

"metadata": {

"name": "xx"

}

"spec": {

"containers": [{

"name": "front-end",

"image": "nginx",

"ports": [{

"containerPort": "80"

}]

}, {

"name": "flaskapp-demo",

"image": "jcdemo/flaskapp",

"ports": [{

"containerPort": "5000"

}]

}]

}

}

yaml：

---

apiVersion: v1

kind: Pod

metadata:

name: xx

spec:

containers:

- name: front-end

image: nginx

ports:

- containerPort: 80

- name: flaskapp-demo

image: jcdemo/flaskapp

ports: 8080

这个文件简单地定义了一个 Pod 对象，包含两个容器，我们可以很清晰地看到两者是如何转换的。

### k8s创建资源的两种方式

在 K8S 中，有两种创建资源的方式：kubectl 命令和 yaml 配置文件。

两种方式各有各的好处。命令行的方式最为简单，一条命令就万事大吉，但缺点也很明显，你并不知道这条命令背后到底做了哪些事，配置文件就提供了一种让你知其然更知其所以然的方式。总的来说，它有以下好处：

* 完整性：配置文件描述了一个资源的完整状态，可以很清楚地知道一个资源的创建背后究竟做了哪些事；
* 灵活性：配置文件可以创建比命令行更复杂的结构；
* 可维护性：配置文件提供了创建资源对象的模板，能够重复使用；
* 可扩展性：适合跨环境、规模化的部署。
* ......

当然，复杂的东西对用户就难以做到友好，我们需要熟悉它的配置文件的语法，有一定难度。下面举几个例子，让你对 yaml 配置文件有一个基本的认识。

### 例子

下面，我们分别来看看 deployment、pod、service 这三种资源的说明书都长啥样。

由于 K8S 对每种资源的定义非常庞杂，限于篇幅，我们只看一些必选的参数，目的是通过这几个例子，读懂 yaml 配置文件。

#### deployment

定义 deployment 配置文件，命名为：nginx-deployment.yaml

apiVersion: apps/v1 # 1.9.0 之前的版本使用 apps/v1beta2，可通过命令 kubectl api-versions 查看

kind: Deployment #指定创建资源的角色/类型

metadata: #资源的元数据/属性

name: nginx-deployment #资源的名字，在同一个namespace中必须唯一

spec:

replicas: 2 #副本数量2

selector: #定义标签选择器

matchLabels:

app: web-server

template: #这里Pod的定义

metadata:

labels: #Pod的label

app: web-server

spec: # 指定该资源的内容

containers:

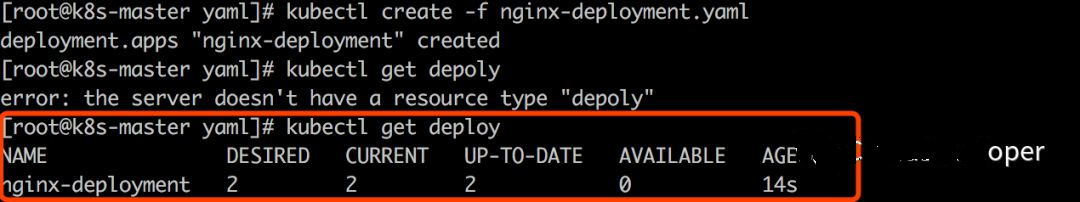
- name: nginx #容器的名字

image: nginx:1.12.1 #容器的镜像地址

ports:

- containerPort: 80 #容器对外的端口

执行kubectl create -f nginx.yaml创建 deployment 资源：



#### pod

定义 pod 配置文件，命名为 redis-pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: pod-redis

labels:

name: redis

spec:

containers:

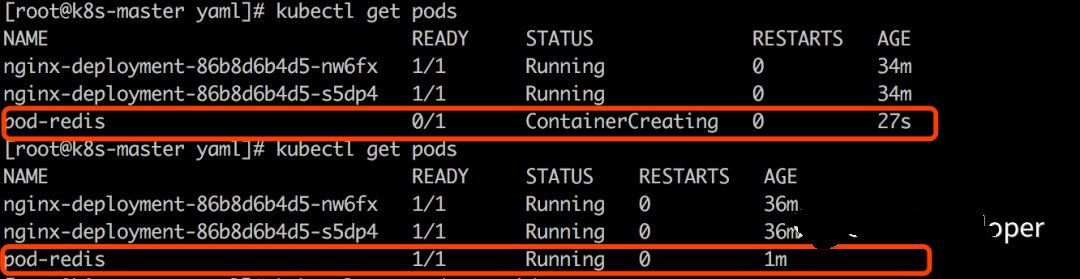
- name: pod-redis

image: docker.io/redis

ports:

- containerPort: 80 #容器对外的端口

执行kubectl create -f pod-redis.yaml创建 pod 资源：



可以看到，成功创建一个 Pod，ContainerCreating表示 Pod 中的容器正在执行镜像的下载和安装过程，过一会儿，就显示Running了，表明 Pod 应用部署完成。

#### service

定义 service 配置文件，命名为 httpd-svc.yaml

apiVersion: v1

kind: Service # 指明资源类型是 service

metadata:

name: httpd-svc # service 的名字是 httpd-svc

labels:

name: httpd-svc

spec:

ports: # 将 service 8080 端口映射到 pod 的 80 端口，使用 TCP 协议

- port: 8080

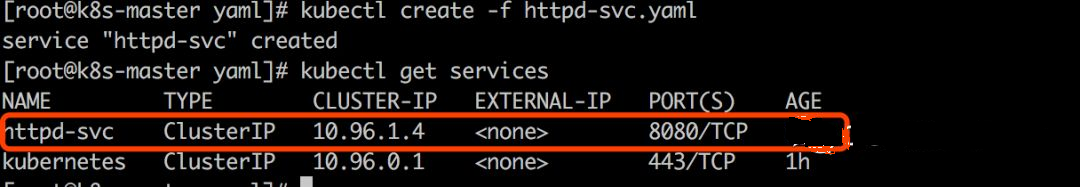
targetPort: 80

protocol: TCP

selector:

run: httpd # 指明哪些 label 的 pod 作为 service 的后端

执行kubectl create -f httpd-svc.yaml创建 service 资源：



可以看到，service httpd-svc 分配到一个 Cluster-IP 10.96.0.1，我们可以通过该 IP 访问 service 所维护的后端 Pod。

另外，还有一个 service kubernetes，这个是 Kubernetes API Server 的 service，Cluster 内部的各组件就是通过这个 service 来访问 API Server。

#### 一个完整的yaml格式pod定义文件

apiVersion: v1 #必选，版本号，例如v1

kind: Pod #必选，Pod

metadata: #必选，元数据

name: string #必选，Pod名称

namespace: string #必选，Pod所属的命名空间

labels: #自定义标签

- name: string #自定义标签名字

annotations: #自定义注释列表

- name: string

spec: #必选，Pod中容器的详细定义

containers: #必选，Pod中容器列表

- name: string #必选，容器名称

image: string #必选，容器的镜像名称

imagePullPolicy: [Always | Never | IfNotPresent] #获取镜像的策略 Alawys表示下载镜像 IfnotPresent表示优先使用本地镜像，否则下载镜像，Nerver表示仅使用本地镜像

command: [string] #容器的启动命令列表，如不指定，使用打包时使用的启动命令

args: [string] #容器的启动命令参数列表

workingDir: string #容器的工作目录

volumeMounts: #挂载到容器内部的存储卷配置

- name: string #引用pod定义的共享存储卷的名称，需用volumes[]部分定义的的卷名

mountPath: string #存储卷在容器内mount的绝对路径，应少于512字符

readOnly: boolean #是否为只读模式

ports: #需要暴露的端口库号列表

- name: string #端口号名称

containerPort: int #容器需要监听的端口号

hostPort: int #容器所在主机需要监听的端口号，默认与Container相同

protocol: string #端口协议，支持TCP和UDP，默认TCP

env: #容器运行前需设置的环境变量列表

- name: string #环境变量名称

value: string #环境变量的值

resources: #资源限制和请求的设置

limits: #资源限制的设置

cpu: string #Cpu的限制，单位为core数，将用于docker run --cpu-shares参数

memory: string #内存限制，单位可以为Mib/Gib，将用于docker run --memory参数

requests: #资源请求的设置

cpu: string #Cpu请求，容器启动的初始可用数量

memory: string #内存清楚，容器启动的初始可用数量

livenessProbe: #对Pod内个容器健康检查的设置，当探测无响应几次后将自动重启该容器，检查方法有exec、httpGet和tcpSocket，对一个容器只需设置其中一种方法即可

exec: #对Pod容器内检查方式设置为exec方式

command: [string] #exec方式需要制定的命令或脚本

httpGet: #对Pod内个容器健康检查方法设置为HttpGet，需要制定Path、port

path: string

port: number

host: string

scheme: string

HttpHeaders:

- name: string

value: string

tcpSocket: #对Pod内个容器健康检查方式设置为tcpSocket方式

port: number

initialDelaySeconds: 0 #容器启动完成后首次探测的时间，单位为秒

timeoutSeconds: 0 #对容器健康检查探测等待响应的超时时间，单位秒，默认1秒

periodSeconds: 0 #对容器监控检查的定期探测时间设置，单位秒，默认10秒一次

successThreshold: 0

failureThreshold: 0

securityContext:

privileged:false

restartPolicy: [Always | Never | OnFailure]#Pod的重启策略，Always表示一旦不管以何种方式终止运行，kubelet都将重启，OnFailure表示只有Pod以非0退出码退出才重启，Nerver表示不再重启该Pod

nodeSelector: obeject #设置NodeSelector表示将该Pod调度到包含这个label的node上，以key：value的格式指定

imagePullSecrets: #Pull镜像时使用的secret名称，以key：secretkey格式指定

- name: string

hostNetwork:false #是否使用主机网络模式，默认为false，如果设置为true，表示使用宿主机网络

volumes: #在该pod上定义共享存储卷列表

- name: string #共享存储卷名称 （volumes类型有很多种）

emptyDir: {} #类型为emtyDir的存储卷，与Pod同生命周期的一个临时目录。为空值

hostPath: string #类型为hostPath的存储卷，表示挂载Pod所在宿主机的目录

path: string #Pod所在宿主机的目录，将被用于同期中mount的目录

secret: #类型为secret的存储卷，挂载集群与定义的secre对象到容器内部

scretname: string

items:

- key: string

path: string

configMap: #类型为configMap的存储卷，挂载预定义的configMap对象到容器内部

name: string

items:

- key: string

path: string

### 简化k8s yaml文件创建

如果需要自动化处理复杂的Kubernetes任务，常常需要编写Yaml配置文件。由于Yaml文件格式比较复杂，即使是老手有时也不免会犯错或需要查询文档，也有人开玩笑这是使用 Yaml 编程。下面我们来学习几个方法来帮助大家来简化 Kubernetes Yaml 文件创建。

#### 模拟命令执行

kubectl中很多命令支持 --dry-run 和 -o yaml 参数，可以方便地模拟命令执行，并输出yaml格式的命令请求，这样我们就可以将执行结果 Copy & Paste到自己的编辑器中，修改完成自己的配置文件。

$ kubectl run myapp --image=nginx --dry-run -o yaml

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

creationTimestamp: null

labels:

run: myapp

name: myapp

spec:

replicas: 1

selector:

matchLabels:

run: myapp

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

run: myapp

spec:

containers:

- image: nginx

name: myapp

resources: {}

status: {}

$ kubectl create secret generic mysecret --from-literal=quiet-phrase="Shh! Dont' tell" -o yaml --dry-run

apiVersion: v1

data:

quiet-phrase: U2hoISBEb250JyB0ZWxs

kind: Secret

metadata:

creationTimestamp: null

name: mysecret

#### 导出资源描述

kubectl get <resource-type> <resource> --export -o yaml 命令会以Yaml格式导出系统中已有资源描述

比如，我们可以将系统中 mysql 部署的描述导成 Yaml 文件

$ kubectl get deployment mysql --export -o yaml > mysql.yaml

#### Kompose转换

Docker Compose是最为流行的容器编排开发工具，可以方便地在Docker/Docker Swarm环境下开发、部署容器编排模板。

Kompose是Kubernetes社区开发的一个转换工具，可以方便地将简单的Docker Compose模板转化成为Kubernetes的Yaml描述文件，并在Kubernetes集群上部署和管理应用

我们可以通过 https://github.com/kubernetes/kompose 获取最新的kompose 工具

如下 docker-compose.yaml 模板描述了一个 WordPress 编排，其中包含两个服务，wordpress服务和mysql服务，为了将 wordpress服务暴露给外部访问，我们增加了一个 Kompose 特有的标签 kompose.service.type: nodeport，将服务以NodePort方式提供对外访问方式。

version: '2'

services:

wordpress:

image: wordpress:4

environment:

- WORDPRESS\_DB\_PASSWORD=password

ports:

- 80:80

depends\_on:

- mysql

labels:

kompose.service.type: nodeport

mysql:

image: mysql:5.7

environment:

- MYSQL\_ROOT\_PASSWORD=password

我们可以通过kompose convert命令转换为 Kubernetes 的 Deployment和Service的资源描述，

$ kompose convert

也可以通过kompose convert --stdout命令输出到标准输出

## ConfigMap

很多情况下我们为某一应用做好镜像，当我们想修改其中的一些参数的时候，就变得比较麻烦，又要重新制作镜像，我们是不是有一种方式，让镜像根据不同的场景调用我们不同的配置文件呢，那我们就需要用到 k8s 的另外一种资源，那就是 ConfigMap。

我们知道，在几乎所有的应用开发中，都会涉及到配置文件的变更，比如说在web的程序中，需要连接数据库，缓存甚至是队列等等。而我们的一个应用程序从写第一行代码开始，要经历开发环境、测试环境、预发布环境只到最终的线上环境。而每一个环境都要定义其独立的各种配置。如果我们不能很好的管理这些配置文件，你的运维工作将顿时变的无比的繁琐。为此业内的一些大公司专门开发了自己的一套配置管理中心，如360的Qcon，百度的disconf等。kubernetes也提供了自己的一套方案，即ConfigMap。kubernetes通过ConfigMap来实现对容器中应用的配置管理。

ConfigMap API资源提供了将配置数据注入容器的机制，同时保持容器不受kubernetes的影响。ConfigMap可用于存储细粒度信息如单个属性，或粗粒度信息如整个配置文件或JSON对象。

ConfigMap API资源保存配置数据的键值对，可以在pods中使用或者可以用于存储系统组件的配置数据。ConfigMap类似于Secrets，但是旨在更方便的使用不包含敏感信息的字符串。

注意：ConfigMap不打算充当属性文件的替换者。

### 创建ConfigMap

ConfigMap是用来存储配置文件的kubernetes资源对象，所有的配置内容都存储在etcd中。

创建ConfigMap的方式有4种：

* 通过直接在命令行中指定configmap参数创建，即--from-literal；
* 通过指定文件创建，即将一个配置文件创建为一个ConfigMap，--from-file=<文件>;
* 通过一个文件内多个键值对，--from-env-file=<文件>；
* 事先写好标准的configmap的yaml文件，然后kubectl create -f 创建

#### 通过 --from-literal

kubectl create configmap test-config1 --from-literal=db.host=172.18.8.200 --from-literal=db.port='3306'

查看配置内容。

kubectl get cm test-config1 -o yaml

#### 通过 --from-file

也可以通过--from-file指定文件，并将他多次传递给kubectc。下面的命令和上面的示例产生相同的结果：

echo -n 172.18.8.200 > ./db.host

echo -n 3306 > ./db.port

kubectl create cm test-config2 --from-file=./db.host --from-file=./db.port

查看配置内容：

kubectl get cm test-config2 -o yaml

每个文件内容对应一个信息条目。

#### 通过 --from-env-file

cat << EOF > env.txt

db.host=172.18.8.200

db.port=3306

EOF

kubectl create cm test-config3 --from-env-file=env.txt

查看配置内容：

kubectl get cm test-config3 -o yaml

#### 通过YAML配置文件

配置文件内容如下。

apiVersion: v1

kind: ConfigMap

metadata:

name: test-config4

data:

db.host: 172.18.8.200

db.port: "3306"

创建并查看其内容。

kubectl apply -f db.yaml

kubectl get cm test-config4 -o yaml

### ConfigMap使用

ConfigMap有以下几种使用方式：

* 通过环境变量的方式，填充环境变量的值，传递给pod
* 设置容器内的命令行参数
* 作为volume的方式挂载到pod内

#### 通过环境变量使用

使用valueFrom、configMapKeyRef、name、key指定要用的key。

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: busybox

args: [ "/bin/sh", "-c", "sleep 3000" ]

env:

- name: DB\_HOST

valueFrom:

configMapKeyRef:

name: test-config4

key: db.host

- name: DB\_PORT

valueFrom:

configMapKeyRef:

name: test-config4

key: db.port

还可以通过envFrom、configMapRef、name使得configmap中的所有key/value对都自动变成环境变量。

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: busybox

args: [ "/bin/sh", "-c", "sleep 3000" ]

envFrom:

- configMapRef:

name: test-config3

#### 作为volume挂载使用

把test-config4所有key/value挂载进来：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: busybox

args: [ "/bin/sh", "-c", "sleep 3000" ]

volumeMounts:

- name: db

mountPath: "/etc/db"

readOnly: true

volumes:

- name: db

configMap:

name: test-config4

进入容器查看，看到在db文件夹下以每一个key为文件名value为值创建了多个文件。

kubectl exec -it mypod -- /bin/sh

---

# cd /etc/db

# ls -al

# cat db.h

#### ConfigMap的热更新

使用该 ConfigMap 挂载的 Env 不会同步更新；

使用该 ConfigMap 挂载的 Volume 中的数据需要一段时间（实测大概10秒）才能同步更新。

#### 最佳使用方法

大多数情况下，配置信息都以文件形式提供，所以在创建 ConfigMap 时通常采用 --from-file 或 YAML 方式，读取 ConfigMap 时通常采用 Volume 方式。

##### 使用ConfigMap配置MySQL

比如我们的 MySQL 配置文件/etc/my.cnf。

[mysqld]

datadir=/var/lib/mysql

socket=/var/lib/mysql/mysql.sock

symbolic-links=0

[mysqld\_safe]

log-error=/var/log/mariadb/mariadb.log

pid-file=/var/run/mariadb/mariadb.pid

!includedir /etc/my.cnf.d

创建CongifMap。

kubectl create cm mysql-cm --from-file=/etc/my.cnf

查看创建好的cm。

kubectl get cm mysql-cm -o yaml

在 Pod 中使用此 ConfigMap，配置文件为：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: busybox

args: [ "/bin/sh", "-c", "sleep 3000" ]

volumeMounts:

- name: mysql

mountPath: "/tmp"

volumes:

- name: mysql

configMap:

name: mysql-cm

items:

- key: my.cnf

path: mysql/my.cnf

创建 Pod 并读取配置信息：

kubectl exec -it mypod sh

# cat /tmp/mysql/my.cnf

关于挂在路径大家可以自行进行修改。

##### 使用ConfigMap配置Redis

让我看另一个真实示例：使用ConfigMap配置redis。加入我们想用推荐的配置注入redis作为缓存。redis的配置文件应该包含：

maxmemory 2mb

maxmemory-policy allkeys-lru

这个文件在/root/k8s-configmap/redis，我们可以用下面的命令新建一个ConfigMap：

$ kubectl create configmap example-redis-config --from-file=/root/k8s-configmap/redis/redis-config

$ kubectl get configmap example-redis-config -o yaml

apiVersion: v1

data:

redis-config: |

maxmemory 2mb

maxmemory-policy allkeys-lru

kind: ConfigMap

metadata:

creationTimestamp: 2016-03-30T18:14:41Z

name: example-redis-config

namespace: default

resourceVersion: "24686"

selfLink: /api/v1/namespaces/default/configmaps/example-redis-config

uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002

现在我们新建一个pod使用这个配置：

apiVersion: v1

kind: Pod

metadata:

name: redis

spec:

containers:

- name: redis

image: kubernetes/redis:v1

env:

- name: MASTER

value: "true"

ports:

- containerPort: 6379

resources:

limits:

cpu: "0.1"

volumeMounts:

- mountPath: /redis-master-data

name: data

- mountPath: /redis-master

name: config

volumes:

- name: data

emptyDir: {}

- name: config

configMap:

name: example-redis-config

items:

- key: redis-config

path: redis.conf

注意，这个pod有一个ConfigMap卷，它将example-redis-config ConfigMap的redis-config键的值放入一个名为redis.conf的文件中。这个卷是挂载到redis容器里面的/redis-master目录里面，我们的配置文件在/redis-master/redis.conf，这个redis注解点在镜像里面的配置文件。

kubectl create -f /root/k8s-configmap/redis/redis-pod.yaml

运行kubectl exec进入到这个pod里面并运行redis-cli工具，可以检查我们的配置是否正确应用：

kubectl exec -it redis redis-cli

127.0.0.1:6379> CONFIG GET maxmemory

1) "maxmemory"

2) "2097152"

127.0.0.1:6379> CONFIG GET maxmemory-policy

1) "maxmemory-policy"

2) "allkeys-lru"

### 限制

ConfigMap必须在pod消耗他之前创建，除非被标记为可选。对于不存在的ConfigMap将阻止它启动。控制器可能被写入以容忍丢失数据，可以个根据具体情况查看通过ConfigMap配置的各个组件。

通过configMapKeyRef引用命名ConfigMap中不存在的键将阻止他的启动。

ConfigMaps用于填充环境变量通过envFrom,它们被认为具有无效的环境变量名称的键将跳过这些键。该pod被允许启动。将会有一个事件，原因是InvalidVariabelNames,该消息包含被跳过的无效的键的列表。该示例显示一个pod它指的是 default/myconfig ConfigMap包含两个无效的键,1badkey和2alsobad。

kubectl.sh get events

ConfigMap驻留到namespace里面。他们可以被引用和pod相同的namespace。ConfigMap配额大小是一个计划功能。

kubelet仅支持pod使用ConfigMap获取从API server。这包括使用kubectl创建每个pod,或者间接通过副本控制器。

## 业务系统部署方案

本章节我们来学习使用k8s发布服务

### docker、k8s部署一个简单应用

下面使用Docker+Kubernetes搭建一个简单的Hello World Node.js应用程序。

#### 准备工作

准备好一个k8s集群（前面章节已经准备好了，准备系统资源足够），安装nodejs环境。

sudo yum install nodejs

#### 编写node.js应用程序

将此代码保存在以文件名命名hellonode 的文件夹中server.js

var http = require('http');

var handleRequest = function(request, response) {

    console.log('Received request for URL: ' + request.url);

    response.writeHead(200);

    response.end('Hello World!');

};

var www = http.createServer(handleRequest);

www.listen(8080);

这可以先试用node server.js来运行一下，应该能够在http：// localhost：8080 /上看到“Hello World！”消息。按Ctrl-C停止正在运行的Node.js服务器。

#### 将应用程序打包到docker容器中

这里需要使用Dockerfile文件来创建容器，在server.js文件夹内创建Dockerfile文件，如下：

FROM node:6.9.2

EXPOSE 8080

COPY server.js .

CMD node server.js

Docker镜像的这个配置从Docker注册表中的官方Node.js LTS镜像开始，公开端口8080，将server.js文件复制到镜像并启动Node.js服务器。

#### 使用build指令根据Dockerfile文件创建镜像

使用Docker守护进程构建Docker镜像（注意尾随点）：

docker build -t hello-node:v1 .

#### 创建部署

Kubernetes Pod是一个由一个或多个容器组成的组合，用于管理和联网。本教程中的Pod只有一个Container。Kubernetes 部署会检查Pod的运行状况，并在Pod终止时重新启动Pod的容器。部署是管理Pod的创建和扩展的推荐方法。使用该kubectl run命令创建管理Pod的Deployment。Pod根据hello-node:v1Docker镜像运行Container 。将 --image-pull-policy标志设置Never为始终使用本地映像，而不是从Docker注册表中提取它（因为您还没有将它推到那里）：

kubectl run hello-node --image=hello-node:v1 --port=8080 --image-pull-policy=Never

查看部署：

kubectl get deployments

查看Pod：

kubectl get pods

#### 创建服务

默认情况下，Pod只能通过Kubernetes集群中的内部IP地址访问。要从hello-nodeKubernetes虚拟网络外部访问Container，须将Pod公开为Kubernetes 服务。可以使用以下kubectl expose命令将Pod公开到公共Internet ：

kubectl expose deployment hello-node --type=LoadBalancer

查看刚刚创建的服务：

kubectl get services

--type=LoadBalancer标志表示在群集外部公开的服务。

#### 访问应用

使用本地IP地址自动打开浏览器窗口，并显示“Hello World”消息。

#### 更新应用

这编辑server.js文件以返回新消息：

response.end(' www.study.163.com');

#### 重新构建版本的镜像

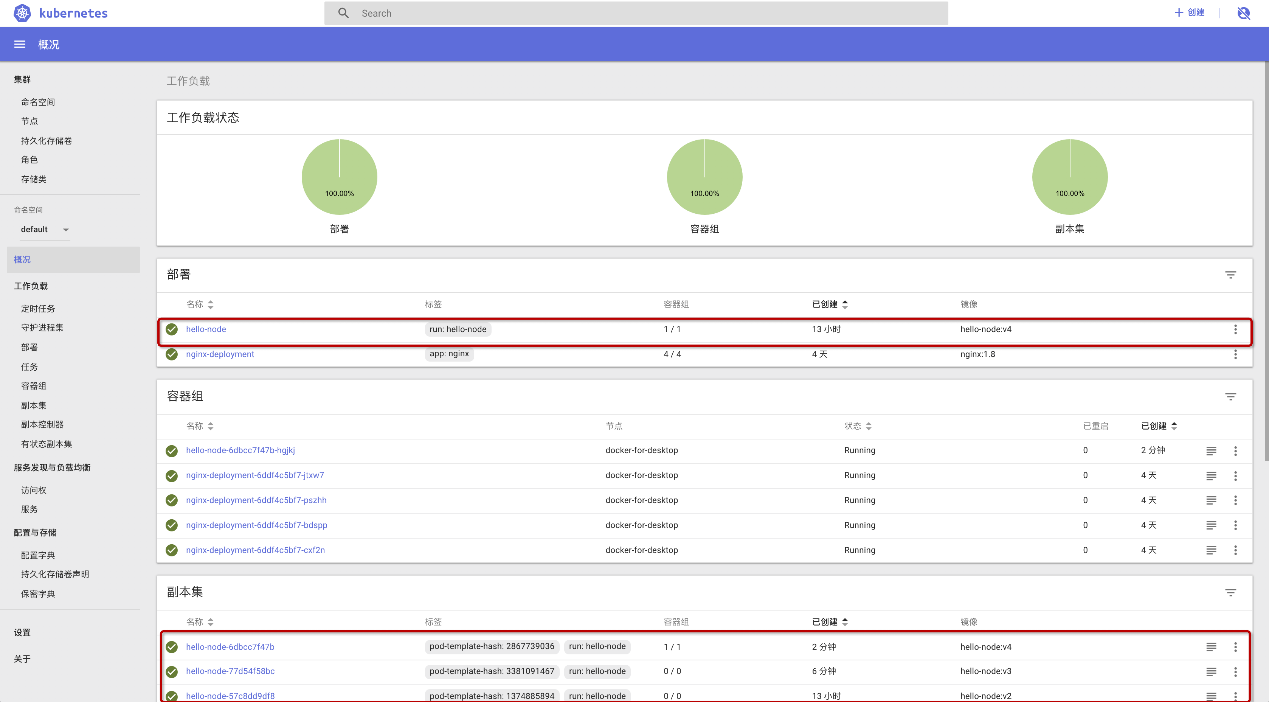
docker build -t hello-node:v2 .

#### 更新部署的镜像

kubectl set image deployment/hello-node hello-node=hello-node:v2

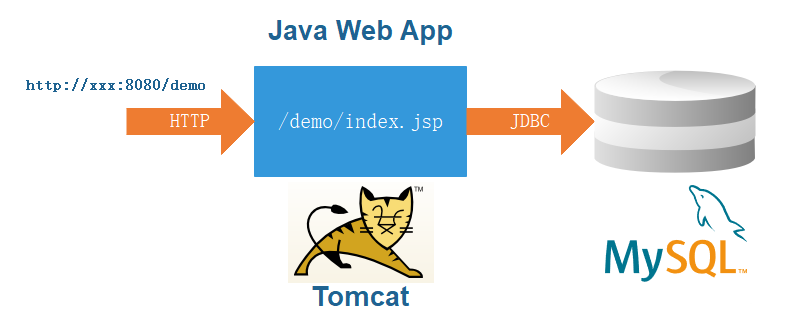
再次运行您的应用以查看新消息

#### k8s仪表盘



### docker、k8s部署一个java web应用

本章节部署tomcat+mysql实现Jave Web应用。此应用是JSP页面通过JDBC访问Mysql数据库，只要程序正确连接到数据库上，就会自动完成对应的Table的创建与初始化数据的准备工作。当我们通过浏览器访问此应用时，就会显示一个表格的页面，数据则来自数据库。



此应用需要启动两个容器：Web App容器和MySQL容器，并且Web App容器需要访问MySQL容器。现在我们就来看看通过Kubernetes是如何实现Java Web应用的。

#### 创建并配置mysql容器

##### 创建mysql的Deployment定义文件

mysql-dep.yaml文件内容如下：

apiVersion: extensions/v1beta1 #apiserver的版本

kind: Deployment #副本控制器deployment，管理pod和RS

metadata:

name: mysql #deployment的名称，全局唯一

spec:

replicas: 1 #Pod副本期待数量

selector:

matchLabels: #定义RS的标签

app: mysql #符合目标的Pod拥有此标签

strategy: #定义升级的策略

type: RollingUpdate #滚动升级，逐步替换的策略

template: #根据此模板创建Pod的副本（实例）

metadata:

labels:

app: mysql #Pod副本的标签，对应RS的Selector

spec:

containers: #Pod里容器的定义部分

- name: mysql #容器的名称

image: mysql:5.7 #容器对应的docker镜像

volumeMounts: #容器内挂载点的定义部分

- name: time-zone #容器内挂载点名称

mountPath: /etc/localtime #容器内挂载点路径，可以是文件或目录

- name: mysql-data

mountPath: /var/lib/mysql #容器内mysql的数据目录

- name: mysql-logs

mountPath: /var/log/mysql #容器内mysql的日志目录

ports:

- containerPort: 3306 #容器暴露的端口号

env: #写入到容器内的环境容量

- name: MYSQL\_ROOT\_PASSWORD #定义了一个mysql的root密码的变量

value: "123456"

volumes: #本地需要挂载到容器里的数据卷定义部分

- name: time-zone #数据卷名称，需要与容器内挂载点名称一致

hostPath:

path: /etc/localtime #挂载到容器里的路径，将localtime文件挂载到容器里，可让容器使用本地的时区

- name: mysql-data

hostPath:

path: /data/mysql/data #本地存放mysql数据的目录

- name: mysql-logs

hostPath:

path: /data/mysql/logs #本地存入mysql日志的目录

* apiVersion：定义使用apiserver的哪个版本，可通过kubectl api-versions命令查看apiserver有哪些版本；
* kind：用来表明此资源对象的类型，比如这里的值为“Deployment”，表示这是一个deployment；
* spec：RS相关属性定义，spec.selector是RS的Pod标签（Label）选择器，即监控和管理拥有这些标签的Pod实例，确保当前集群上始终有且仅有replicas个Pod实例在运行，这里设置replicas=1表示只能运行一个Mysql Pod实例。
* spec.strategy：定义Pod的升级方案，Recreate表示删除所有已存在的Pod，重新创建新的；RollingUpdate表示滚动升级，逐步替换的策略，滚动升级时支持更多的附加参数，例如设置最大不可用Pod数量，最小升级间隔时间等等。
* spec.template：当集群中运行的Pod数量小于replicas时，RS会根据spec.template中定义的Pod模板来生成一个新的Pod实例，spec.template.metadata.labels指定了该Pod的标签，需要特别注意的是，这里的labels必须匹配之前的spec.selector。
* spec.template.spec.containers：容器的定义部分，包括容器的名称、使用的docker镜像、挂载数据卷、服务的端口号、变量等内容。
* spec.template.spec.volumes：需要挂载到容器里的本地数据卷的定义部分，数据卷的名称要与容器内挂载点的名称一致，path定义本地的数据卷路径。

##### 创建deployment、RS、Pod和容器

创建过程需要先下载镜像，时间会比较久，可喝杯茶撩撩旁边的妹子，哈哈，请耐心等待

kubectl create -f mysql-dep.yaml

##### 查看创建好的deployment运行情况

kubectl get deployment

注：都是1表示运行正常

##### 查看ReplicaSet（RS）的运行情况

kubectl get rs

注：都是1表示运行正常

##### 查看Pod的运行情况

kubectl get pod

注：READY的值是1/1，并且STATUS的值是Running，表示运行正常

由于Pod的创建需要花费一些时间，在还没有创建好容器时，STATUS的状态会是ContainerCreating，表示正在创建容器，这时只需要等待。Pod创建好后，STATUS的状态会是Running，这时可以通过docker ps命令查看容器运行的情况。

##### 查看容器的运行情况

docker ps

##### 查看Pod里容器的时间，检查时间是否与本地时间一致

kubectl exec mysql-3238461207-vvwt8 date

注：exec后面跟pod的名称

##### 创建mysql的service定义文件

mysql-svc.yaml文件内容如下：

apiVersion: v1

kind: Service #表示Kubernetes Service

metadata:

name: mysql #Service的名称

spec:

ports:

- port: 3306 #Service提供服务的端口号

selector:

app: mysql #Service对应的Pod的标签

* metadata.name：Service的服务名称
* spec.ports：Service提供的服务端口号，对应容器的服务端口号
* spec.selector：确定哪些Pod副本（实例）对应到此Service

##### 创建service

kubectl create -f mysql-svc.yaml

##### 查看Service的运行情况

kubectl get svc

kubernetes会给Service分配一个Cluster IP，这是个虚拟IP地址，此后集群中的其他新创建的Pod就可以通过此Cluster IP+端口号的方式来连接和访问mysql服务了。

#### 创建并配置tomcat容器

##### 创建tomcat的Deployment定义文件

myweb-dep.yaml文件的内容如下：

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: myweb

spec:

replicas: 1

selector:

matchLabels:

app: myweb

strategy:

type: RollingUpdate

template:

metadata:

labels:

app: myweb

spec:

containers:

- name: myweb

image: kubeguide/tomcat-app:v1

volumeMounts:

- name: time-zone

mountPath: /etc/localtime

- name: tomcat-logs

mountPath: /usr/local/tomcat/logs

ports:

- containerPort: 8080

env:

- name: MYSQL\_SERVICE\_HOST

value: '10.254.144.64' #此处为mysql服务的Cluster IP

- name: MYSQL\_SERVICE\_PORT

value: '3306'

volumes:

- name: time-zone

hostPath:

path: /etc/localtime

- name: tomcat-logs

hostPath:

path: /data/tomcat/logs

##### 创建tomcat的deployment、RS、Pod和容器

kubectl create -f myweb-dep.yaml

创建过程比较久，请耐心等待，pod的STATUS状态为Running时表示创建成功。

##### 创建tomcat的Service定义文件

myweb-svc.yaml文件的内容如下：

apiVersion: v1

kind: Service

metadata:

name: myweb

spec:

type: NodePort

ports:

- port: 8080

nodePort: 30001

selector:

app: myweb

此Service开启了NodePort方式的外网访问模式，端口为30001，此端口会映射到tomcat容器的8080端口上。

##### 创建Service

kubectl create -f myweb-svc.yaml

##### 查看Service的运行情况

kubectl get svc

可使用curl命令测试tomcat服务是否能正常访问

curl http://192.168.2.238:30001

#### 通过浏览器访问网页

1、通过浏览器访问[http://](http://192.168.100.246:30001/demo/)~~[192.168.100.246](http://192.168.100.246:30001/demo/)~~[:30001/demo/](http://192.168.100.246:30001/demo/)

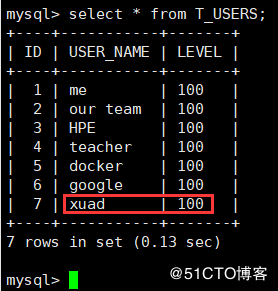
2、登陆mysql数据库验证

docker exec -it 5252cd76009a /bin/bash

mysql -uroot -p123456

use HPE\_APP

select \* from T\_USERS;



### K8s部署微服务

后续以录播方式提供会好很多，课件中不在展示部署过程。

## 服务滚动升级

当有镜像发布新版本，新版本服务上线时如何实现服务的滚动和平滑升级？

如果你使用ReplicationController创建的pod可以使用kubectl rollingupdate命令滚动升级，如果使用的是Deployment创建的Pod可以直接修改yaml文件后执行kubectl apply即可。

Deployment已经内置了RollingUpdate strategy，因此不用再调用kubectl rollingupdate命令，升级的过程是先创建新版的pod将流量导入到新pod上后销毁原来的旧的pod。

Rolling Update适用于Deployment、Replication Controller，官方推荐使用Deployment而不再使用Replication Controller。

使用ReplicationController时的滚动升级请参考官网说明：<https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/>

**ReplicationController与Deployment的关系**

ReplicationController和Deployment的RollingUpdate命令有些不同，但是实现的机制是一样的，关于这两个kind的关系我引用了[ReplicationController与Deployment的区别](https://segmentfault.com/a/1190000008232770)中的部分内容如下，详细区别请查看原文。

**ReplicationController**

Replication Controller为Kubernetes的一个核心内容，应用托管到Kubernetes之后，需要保证应用能够持续的运行，Replication Controller就是这个保证的key，主要的功能如下：

* 确保pod数量：它会确保Kubernetes中有指定数量的Pod在运行。如果少于指定数量的pod，Replication Controller会创建新的，反之则会删除掉多余的以保证Pod数量不变。
* 确保pod健康：当pod不健康，运行出错或者无法提供服务时，Replication Controller也会杀死不健康的pod，重新创建新的。
* 弹性伸缩 ：在业务高峰或者低峰期的时候，可以通过Replication Controller动态的调整pod的数量来提高资源的利用率。同时，配置相应的监控功能（Hroizontal Pod Autoscaler），会定时自动从监控平台获取Replication Controller关联pod的整体资源使用情况，做到自动伸缩。
* 滚动升级：滚动升级为一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始化升级的时候就可以及时发现和解决问题，避免问题不断扩大。

**Deployment**

Deployment同样为Kubernetes的一个核心内容，主要职责同样是为了保证pod的数量和健康，90%的功能与Replication Controller完全一样，可以看做新一代的Replication Controller。但是，它又具备了Replication Controller之外的新特性：

* Replication Controller全部功能：Deployment继承了上面描述的Replication Controller全部功能。
* 事件和状态查看：可以查看Deployment的升级详细进度和状态。
* 回滚：当升级pod镜像或者相关参数的时候发现问题，可以使用回滚操作回滚到上一个稳定的版本或者指定的版本。
* 版本记录: 每一次对Deployment的操作，都能保存下来，给予后续可能的回滚使用。
* 暂停和启动：对于每一次升级，都能够随时暂停和启动。
* 多种升级方案：Recreate：删除所有已存在的pod,重新创建新的; RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用pod数量，最小升级间隔时间等等。

**创建测试镜像**

我们来创建一个特别简单的web服务，当你访问网页时，将输出一句版本信息。通过区分这句版本信息输出我们就可以断定升级是否完成。

所有配置和代码见gitlab源码项目[subject-3-k8s](http://59.111.92.219/java/subject-3-k8s)中的rolling-update-test目录；

这里我们还是拿前面讲过的nodejs代码来做演示，server.js：

var http = require('http');

var handleRequest = function(request, response) {

    console.log('Received request for URL: ' + request.url);

    response.writeHead(200);

    response.end('Hello World!');

};

var www = http.createServer(handleRequest);

www.listen(8080);

**创建Dockerfile**

FROM node:6.9.2

EXPOSE 8080

COPY server.js .

CMD node server.js

注意修改添加的文件的名称。

**创建Makefile**

修改镜像仓库的地址为你自己的私有镜像仓库地址。

修改Makefile中的TAG为新的版本号。

all: build push

.PHONY: build push

TAG = v1

# Build for linux amd64

build:

docker build -t 192.168.100.249/study/hello-node:${TAG} .

# Push to harbor

push:

docker push 192.168.100.249/study/hello-node:${TAG}

**编译**

make all

分别修改server.js中的输出语句、Makefile中的TAG，创建两个版本的镜像。

**测试**

我们使用Deployment部署服务来测试。

配置文件rolling-update-test.yaml：

all: build push

.PHONY: build push

TAG = v1

# Build for linux amd64

build:

docker build -t 192.168.100.249/study/hello-node:${TAG} .

# Push to harbor

push:

docker push 192.168.100.249/study/hello-node:${TAG}apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: rolling-update-test

namespace: default

spec:

replicas: 1

template:

metadata:

labels:

app: rolling-update-test

spec:

containers:

- name: rolling-update-test

image: 192.168.100.249/study/hello-node:v1

ports:

- containerPort: 8080

protocol: TCP

---

apiVersion: v1

kind: Service

metadata:

name: rolling-update-test

labels:

app: rolling-update-test

namespace: default

spec:

ports:

- nodePort: 30321

port: 8080

protocol: TCP

selector:

app: rolling-update-test

type: LoadBalancer

**部署service**

kubectl create -f rolling-update-test.yaml

打开浏览器访问 http://~~192.168.100.246~~:30321 将会看到以下输出：

Hello World！

**滚动升级**

只需要将rolling-update-test.yaml文件中的image改成新版本的镜像名，然后执行：

kubectl apply -f rolling-update-test.yaml

也可以参考[Kubernetes Deployment Concept](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)中的方法，直接设置新的镜像。

kubectl set image deployment/rolling-update-test rolling-update-test=192.168.100.249/study/hello-node:v2

或者使用kubectl edit deployment/rolling-update-test修改镜像名称后保存。

使用以下命令查看升级进度：

kubectl rollout status deployment/rolling-update-test

升级完成后在浏览器中刷新 http://~~192.168.100.246~~:30321 将会看到以下输出：

www.study.163.com

说明滚动升级成功。

**使用ReplicationController创建的Pod如何RollingUpdate**

以上讲解使用Deployment创建的Pod的RollingUpdate方式，那么如果使用传统的ReplicationController创建的Pod如何Update呢？

举个例子：

kubectl -n node-cluster rolling-update node-controller --image 192.168.100.249/study/hello-node:v2

只需要指定新的镜像即可，当然你可以配置RollingUpdate的策略。

## 简化 Kubernetes 应用部署工具 - Helm

## 在 Kubernetes 使用 ceph