

目录

文档说明	1.1
1.基础知识	1.2
导读	1.2.1
1.1 说透 Docker: 基础	1.2.2
1.2 说透 Docker: 虚拟化	1.2.3
1.3 了解 Docker 网络	1.2.4
1.4 Docker 和 Pod	1.2.5
1.5 K8S入门基础	1.2.6
2.部署和配置	1.3
导读	1.3.1
2.1 使用 Minikube 部署	1.3.2
2.2 使用 kubeadm 部署	1.3.3
2.3 CKAD认证中的部署教程	1.3.4
2.4 国内代理	1.3.5
2.5 Dashboard	1.3.6
3.Pod部署和调度	1.4
导读	1.4.1
3.1 Pod	1.4.2
3.2 Deployment部署	1.4.3
3.3 副本集(ReplicaSet)	1.4.4
3.34 Pod 端口映射	1.4.5
3.5 Pod 升级、回滚	1.4.6
3.6 Pod 缩放	1.4.7
3.7.Pod 标签	1.4.8
3.8 Pod 调度	1.4.9
3.9 Jobs、CronJobs	1.4.10
4.Kubernetes 网络	1.5
导读	1.5.1
4.1 Kubernetes 网络	1.5.2
4.2 Endpoint	1.5.3
4.3 ingress	1.5.4
4.4 服务发现	1.5.5
5.volumes	1.6
导读	1.6.1
5.1 卷	1.6.2

5.2 secret 和 ConfigMap 卷	1.6.3
5.3 NFS卷	1.6.4
5.4 持久化卷	1.6.5
6.安全	1.7
导读	1.7.1
7.API与对象	1.8
导读	1.8.1
8.日志和故障排除	1.9
导读	1.9.1
日志和监控工具	1.9.2
抓包	1.9.3
教程	1.9.4
Readme	1.10

- 文档说明

文档说明

作者：痴者工良

文档地址：<https://k8s.whuanle.cn>

简介：这是一本关于 **Kubernetes** 的书。

Kubernetes 标准化词汇表 <https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-app-container>

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-10-19 19:36:37

- 第一章：基础知识
 - 导读

第一章：基础知识

导读

本章将会介绍 Docker 和 Kubernetes 的一些基础知识，掌握 Docker 的组成结构以及 Docker 是怎么隔离容器、隔离硬件资源的，了解为什么用 Kubernetes，Kubernetes 的组成、结构等，我们会学习到很多 K8S 的术语。

由于本章的内容可能看起来比较枯燥，建议第一遍阅读时，只快速了解，读完后面的章节后、练习命令、上手实践，对各方面的技术了解后，再重新看第一大章。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-07 08:16:11

- 1.1 说透 Docker: 基础
 - 容器化应用
 - 什么是容器化应用
 - 应用怎么打包
 - Docker 镜像组成
 - 联合文件系统
 - Linux 内核
 - Docker 结构
 - Docker 服务与客户端
 - Docker 客户端
 - 容器运行时
 - Docker 引擎
 - Docker 引擎变化
 - Docker 引擎的架构

1.1 说透 Docker: 基础

既然要学习 K8S，相信各位读者都已经使用过 Docker 了，Docker 的入门是比较容易的，但 Docker 的网络和存储、虚拟化是相当复杂的，Docker 的技术点比较多，在本章中将会深入介绍 Docker 的各方面，期待能够帮助读者加深对 Docker 的理解。

容器化应用

什么是容器化应用

containerized applications 指容器化的应用，我们常常说使用镜像打包应用程序，使用 Docker 发布、部署应用程序，那么当你的应用成功在 Docker 上运行时，称这个应用是 containerized applications。

应用怎么打包

容器化应用的主要特征是使用镜像打包应用的运行环境以及应用程序，可以通过 Docker 启动这个镜像，进而将应用程序启动起来。

将一个应用程序打包为镜像，大约分为以下过程：

- 编写 Dockerfile 文件 -- 定义构建镜像的流程
- 选择一个基础镜像（操作系统） -- 操作系统
- 安装应用需要的环境 -- 运行环境
- 复制程序文件 -- 应用程序
- 启动 Dockerfile -- 生成镜像

sequenceDiagram 操作系统->>运行环境: Ubuntu 18.04 Note right of 运行环境:
.NET Core Runtime 3.1 运行环境-->>Web程序(C#): 安装运行环境

Docker 镜像组成

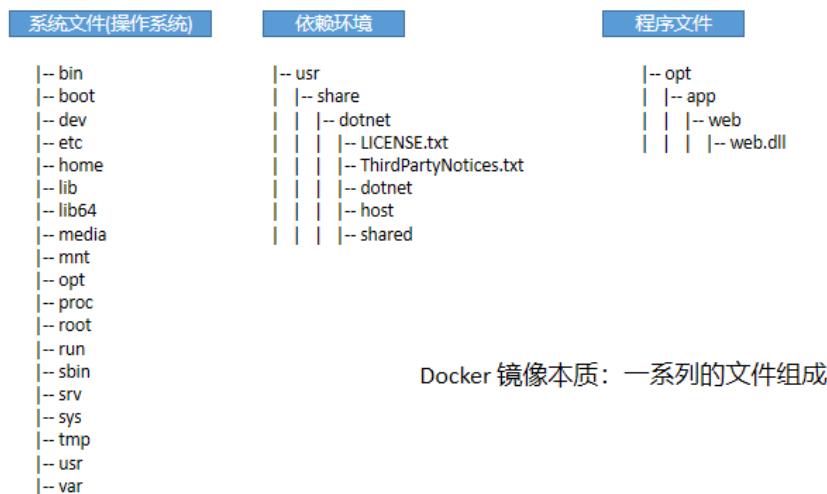
以 .NET Core(C#) 程序为例，一个 Docker 镜像的层次如下图所示：



操作系统：不包含内核、系统文件高度精简

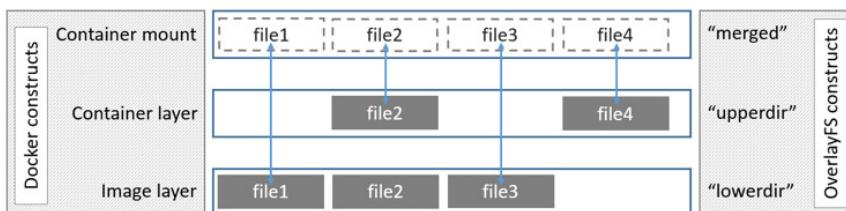
在 Docker 镜像中，操作系统是高度精简的，可能只有一个精简的 Shell，甚至没有 Shell。而且镜像中的操作系统还不包含内核，容器都是共享所在的宿主机的内核。所以有时会说容器仅包含必要的操作系统（通常只有操作系统文件和文件系统对象），容器中查看到的 Linux 内核版本与宿主机一致。

Docker 镜像是由一系统文件组成的。



联合文件系统

Linux 有名为 **Unionfs** 的文件系统服务，可以将不同文件夹中的文件联合到一个文件夹中。Unionfs 有称为分支的概念，一个分支包含了多个目录和文件，多个分支可以挂载在一起，在挂载时，可以指定一个分支优先级大于另一个分支，这样当两个分支都包含相同的文件名时，一个分支会优先于另一个分支，在合并的目录中，会看到高优先级分支的文件。



Docker 中，层层组成镜像的技术也是联合文件系统，**Union File System**。Docker 镜像中的操作系统是根文件系统，在上一小节的图片中，可以看到有 bin、boot 等目录。我们都知道，Docker 镜像是由多层文件组成的，在上面的示例图片中有三层组成：根文件系统、环境依赖包、应用程序文件。当镜像层生成后，便不能被修

改，如果再进行操作，则会在原来的基础上生成新的镜像层，层层联合，最终生成镜像。当然生成的镜像可能会因为层数太多或者操作过多，导致出现大量冗余，镜像臃肿。

Docker 的镜像分层是受 Linux Unionfs 启发而开发的，Docker 支持多种文件联合系统，如 AUFS、OverlayFS、VFS 等。

Docker 在不同系统中可以选择的联合文件系统：

Linux发行版	推荐的存储驱动程序	替代驱动程序
Ubuntu	overlay2	overlay devicemapper , aufs , zfs , vfs
Debian	overlay2	overlay , devicemapper , aufs , vfs
CentOS	overlay2	overlay , devicemapper , zfs , vfs

[info] 提示

Docker Desktop for Mac 和 Docker Desktop for Windows 不支持修改存储驱动程序，只能使用默认存储驱动程序。

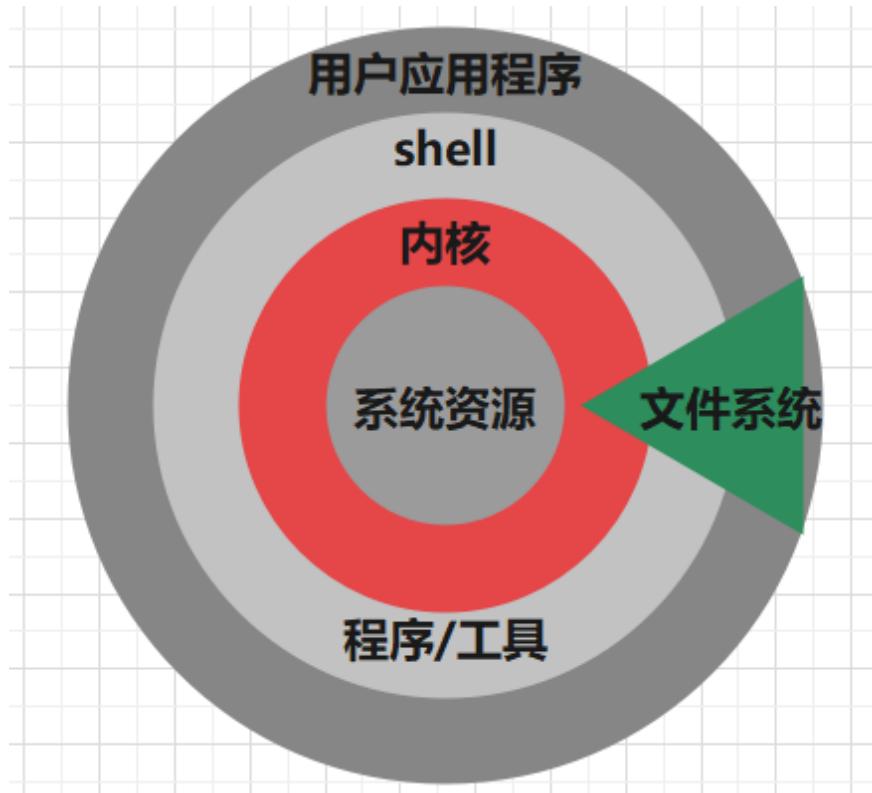
Linux 内核

既然 Docker 容器需要与 Linux 内核结合才能使用，那么我们看一下 Linux 内核的功能，稍微了解一下 Linux 内核在支撑 Docker 容器运作中起到什么作用。

Linux 内核主要包含以下功能：

- 内存管理：追踪记录有多少内存存储了什么以及存储在哪里；
- 进程管理：确定哪些进程可以使用中央处理器（CPU）、何时使用以及持续多长时间；
- 设备驱动程序：充当硬件与进程之间的调解程序/解释程序；
- 系统调用和安全防护：接受程序请求调用系统服务；
- 文件系统：操作系统中负责管理持久数据的子系统，在 Linux 中，一切皆文件。

Linux 层次结构如下：



Docker 容器中包含了一个操作系统，包含简单的 shell 或者不包含，其层次结构如图所示：



Docker 结构

本节将了解 Docker 的组成部件和结构。

Docker 服务与客户端

Docker 由 Service 和 Client 两部分组成，在服务器上可以不安装 Docker Client，可以通过 Http Api 等方式与 Docker Servie 通讯。

在安装了 Docker 的主机上执行命令 `docker version` 查看版本号。

```
Client: Docker Engine - Community
Version:          20.10.7
API version:     1.41
Go version:      go1.13.15
Git commit:      f0df350
Built:            Wed Jun 2 11:58:10 2021
OS/Arch:          linux/amd64
Context:          default
Experimental:    true

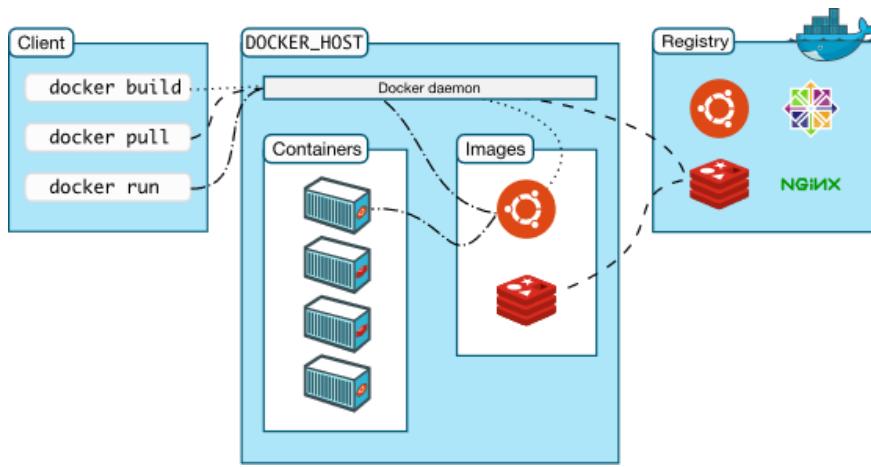
Server: Docker Engine - Community
Engine:
  Version:          20.10.7
  API version:     1.41 (minimum version 1.12)
  Go version:      go1.13.15
  Git commit:      b0f5bc3
  Built:            Wed Jun 2 11:56:35 2021
  OS/Arch:          linux/amd64
  Experimental:    false
  containerd:
    Version:        1.4.6
    GitCommit:      d71fcd7d8303cbf684402823e425e9dd2e99285d
  runc:
    Version:        1.0.0-rc95
    GitCommit:      b9ee9c6314599f1b4a7f497e1f1f856fe433d3b7
  docker-init:
    Version:        0.19.0
    GitCommit:      de40ad0
```

Docker 客户端

要想跟 Docker Server 通讯，可以使用 **Restful API**、**UNIX 套接字**或**网络接口 (Socket)**。Docker 官方的客户端是一个二进制命令行程序，使用 **Go** 语言编写，我们也可以使用 **C#**、**Java** 等语言写一个类似的程序，Docker 客户端不需要安装到 Docker Server 所在的主机，Client 跟 Server 可以远程通讯。

Docker 的客户端是许多 Docker 用户与 Docker 交互的主要方式，当我们使用 `docker run` 之类的命令时，客户端会将这些命令发送到 Docker Server，由 Docker Server 解析并执行命令。

Docker for Linux 中最为常见的同主机通讯方式是 **Unix 域套接字**。很多软件都支持使用域套接字与 Docker 通讯，例如 CI/CD 软件 **Jenkins**，使用域套接字连接 Docker，能够利用 Docker 启动容器构建应用程序以及使用 Docker 来做一些不可描述的事情。



容器运行时

容器运行时是提供运行环境并启动容器的软件，我们最常听说的是 **Docker**，此外还有 **containerd**、**CRI-O** 等。可以毫不夸张的说，整个 **Kubernetes** 建立在容器之上。

默认情况下，**Kubernetes** 使用 容器运行时接口(Container Runtime Interface, CRI) 来与服务器中容器运行时交互。所以 **Kubernetes** 支持多种容器软件，但只能使用一种容器运行时进行工作，在有多个容器运行时的情况下，我们需要指定使用何种运行时，如果你不指定运行时，则 **kubeadm** 会自动尝试检测到系统上已经安装的运行时，方法是扫描一组众所周知的 Unix 域套接字。

Linux 是多进程操作系统，为了让多个系统中的多个进程能够进行高效的通讯，出现和很多方法，其中一种是域套接字(Unix domain socket)，只能用于在同一计算机中的进程间通讯，但是其效率高于网络套接字(socket)，域套接字不需要经过网络协议处理，通过系统调用将数据从一个进程复制到另一个进程中。

域套接字使用一个 `.sock` 文件进行通讯，常见的容器软件其对应域套接字如下：

运行时	域套接字
Docker	<code>/var/run/docker.sock</code>
containerd	<code>/run/containerd/containerd.sock</code>
CRI-O	<code>/var/run/crio/crio.sock</code>

[info]

同一主机下常见进程通讯方式有 共享内存、消息队列、管道通讯(共享文件)。

Unix 域套接字是套接字和管道之间的混合物。在 Linux 中，有很多进程，为了让多个进程能够进行通讯，出现和很多方法，其中一种是套接字(socket)。一般的 socket 都是基于 TCP/IP 的，称为网络套接字，可以实现跨主机进程通讯。在 Linux 中有一种套接字，名为域套接字，只能用于在同一计算机中的进程间通讯，但是其效率高于网络套接字。域套接字使用一个 `.sock` 文件进行通讯。

当计算机中有多种容器运行时，**Kubernetes** 默认优先使用 **Docker**。

如果你想了解 CRI , 请点击:

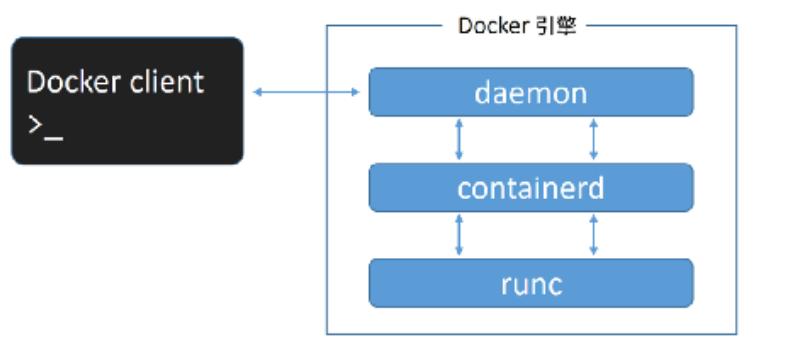
<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md>

Docker 引擎

Docker 引擎也可以说是 Docker Server, 它由 Docker 守护进程（Docker daemon）、containerd 以及 runc 组成。

当使用 Docker client 输入命令时，命令会被发送到 Docker daemon , daemon 会侦听请求并管理 Docker 对象，daemon 可以管理 镜像、容器、网络和存储卷等。

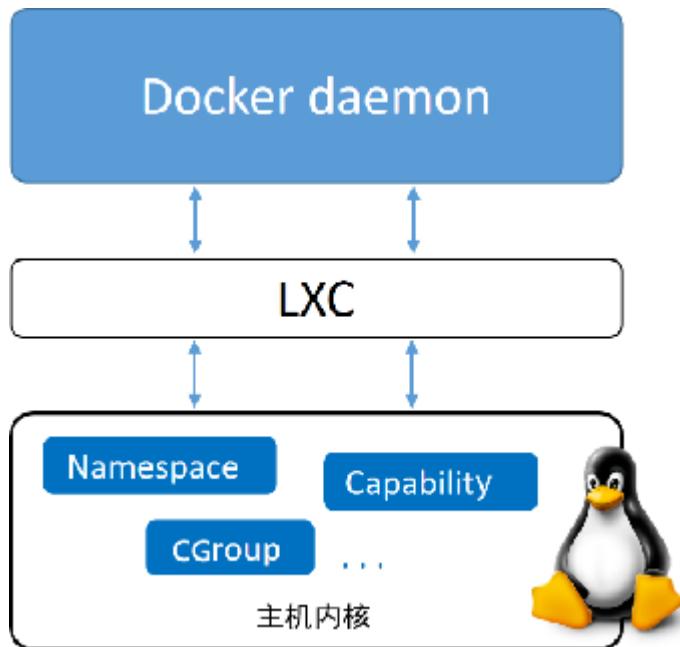
下面这个图是新 Docker 版本的结构组成。



Docker 引擎变化

Docker 首次发布时，Docker 引擎由两个核心组件构成：LXC 和 Docker daemon，这也是很多文章中称 Docker 是基于 LXC 的原因，旧版本的 Docker 利用了 LXC、cgroups、Linux 内核编写。接下来我们了解一下 LXC 。

LXC (Linux Container)是 Linux 提供的一种内核虚拟化技术，可以提供轻量级的虚拟化，以便隔离进程和资源，它是操作系统层面上的虚拟化技术。LXC 提供了对诸如命名空间(namespace) 和控制组(cgroups) 等基础工具的操作能力，它们是基于 Linux 内核的容器虚拟化技术。我们不需要深入了解这个东西。



Docker 一开始是使用 LXC 做的，LXC 是一个很牛逼的开源项目，但是随着 Docker 的成熟，Docker 开始抛弃 LXC，自己动手手撕容器引擎。

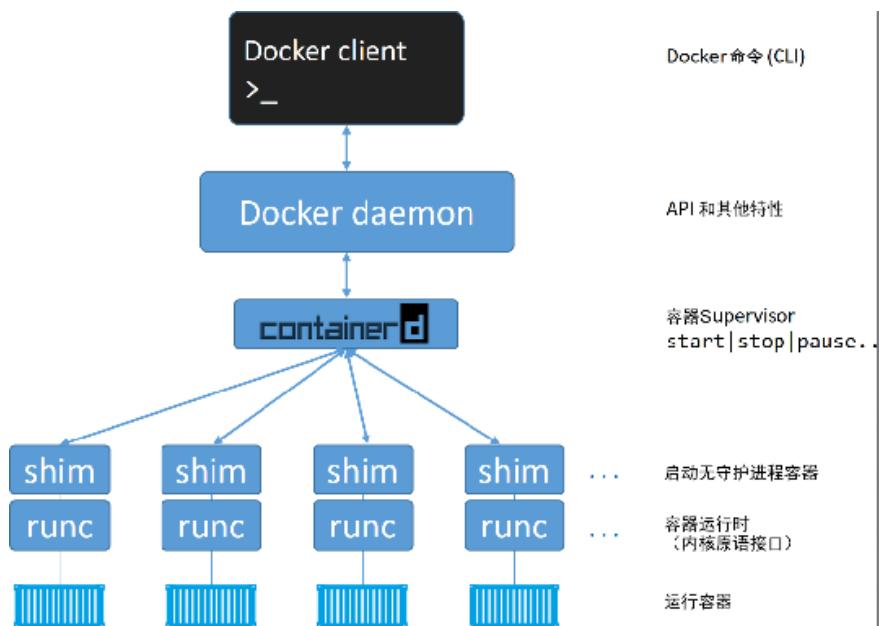
为什么 Docker 要抛弃 LXC 呢？首先，LXC 是基于 Linux 的。这对于一个立志于跨平台的 Docker 来说是个问题，离开 LXC，怎么在 MAC、Windows 下运行？其次，如此核心的组件依赖于外部工具，这会给项目带来巨大风险，甚至影响其发展。

[info]

哈哈哈。。。其实笔者觉得不支持 Windows 也罢。。。

Docker 引擎的架构

下面是一张 Docker 的架构图。



Docker client 和 Docker daemon 在前面已经介绍过了，接下来介绍其他组件。

containerd

containerd 是一个开源容器引擎，是从 Docker 开源出去的。之前有新闻说 Kubernetes 不再支持 Docker，只支持 containerd，很多人以为 Docker 不行了。

一开始 Docker 是一个“大单体”，随着 Docker 的成长，Docker 开始进行模块化，Docker 中的许多模块都是可替换的，如 Docker 网络。支持容器运行的核心代码自然也抽出来，单独做一个模块，便是 containerd。Kubernetes 不再支持 Docker，只不过是降低依赖程度，减少对其他模块的依赖，只集中在 containerd 上。当我们安装 Docker 时，自然会包含 containerd。如果我们不需要 Docker 太多组件，那么我们可以仅仅安装 containerd，由 Kubernetes 调度，只不过我们不能使用 Docker client 了。因此可以说，Kubernetes 不再支持 Docker，并不代表会排斥 Docker。

containerd 的主要任务是容器的生命周期管理，如启动容器、暂停容器、停止容器等。containerd 位于 daemon 和 runc 所在的 OCI 层之间。

shim

shim 它的作用非常单一，那就是实现 CRI 规定的每个接口，然后把具体的 CRI 请求“翻译”成对后端容器项目的请求或者操作。

这里要区别一下，dockershim 和 containerd-shim，dockershim 是一个临时性的方案，dockershim 会在 Kubernetes v1.24 中删除，这也是 Kubernetes 不再支持 Docker 的另一组件。

[info] 提示

CRI 即 Container Runtime Interface，容器运行时接口，容器引擎要支持 Kubernetes，需要实现 CRI 接口，例如 runc、crun 两种是常见的 Container Runtime。

shim 是容器进程的父进程，shim 的生命周期跟容器一样长，shim 是一个轻量级的守护进程，它与容器进程紧密相关，但是 shim 与容器中的进程完全分离。shim 可以将容器的 `stdin`、`stdout`、`stderr` 流重定向到日志中，我们使用 `docker logs` 即可看到容器输出到控制台的流。

关于 shim，我们就先了解到这里，后面会继续讲解一个示例。

runc

runc 实质上是一个轻量级的、针对 Libcontainer 进行了包装的命令行交互工具，runc 生来只有一个作用——创建容器，即 runc 是一个用于运行容器的命令行工具。

[info] 提示

Libcontainer 取代了早期 Docker 架构中的 LXC。

如果主机安装了 Docker，我们可以使用 `runc --help` 来查看使用说明。我们可以这样来理解 runc，runc 是在隔离环境生成新的进程的工具，在这个隔离环境中有一个专用的根文件系统(ubuntu、centos 等)和新的进程树，这个进程树的根进程

PID=1。

[success] 博客推荐

笔者在查阅资料时，发现了这个大佬的博客，在这个大佬的博客中学会了很多东西。

博客推荐：<https://iximiuz.com/en/>

在后面的节中，我们将继续了解 Docker 中的网络和存储，并开始探究与 Kubernetes 相关的知识点。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-10-30 13:19:42

- 1.2 说透 Docker: 虚拟化
 - Docker 虚拟化
 - 关于Docker
 - 传统虚拟化部署方式
 - Linux 虚拟化
 - Linux-Namespace
 - cgroups 硬件资源隔离
 - 聊聊虚拟化
 - 理论基础
 - 软硬件实现等效
 - 虚拟化
 - 不同层次的虚拟化

1.2 说透 Docker: 虚拟化

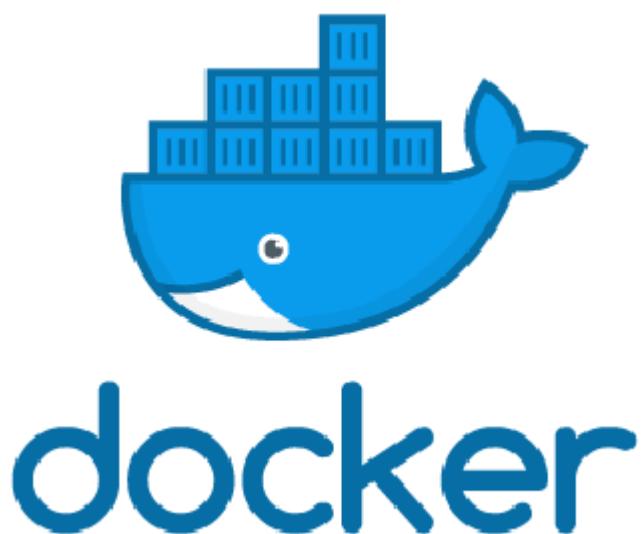
本章内容将讲解 Docker 虚拟化、虚拟化本质、namespace、cgroups。

Docker 虚拟化

关于Docker

本小节将介绍 Docker 虚拟化的一些特点。

Docker 是一个开放源代码软件项目，自动化进行应用程序容器化部署，借此在Linux操作系统上，提供一个额外的软件抽象层，以及操作系统层虚拟化的自动管理机制。 -From wiki



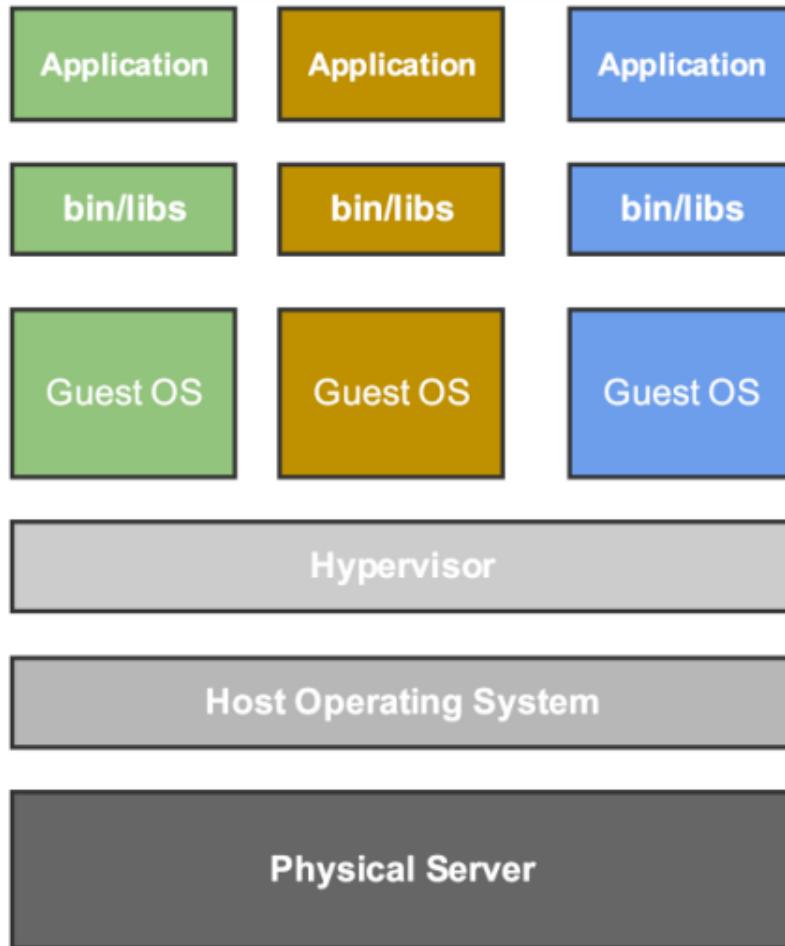
在接触 Docker 的过程中，或多或少会了解到 Docker 的虚拟化，最常见的介绍方式是对比 Docker 和虚拟机之间的差别，笔者这里也给出两者的对比表格，以便后面详细地展开来讲。

	虚拟机	Docker 容器
隔离程度	硬件级进程隔离	操作系统级进程隔离
系统	每个虚拟机都有一个单独的操作系统	每个容器可以共享操作系统（共享操作系统内核）
启动时间	需要几分钟	几秒
体积大小	虚拟机镜像GB级别	容器是轻量级的（KB/MB）
启动镜像	虚拟机镜像比较难找到	预建的 docker 容器很容易获得
迁移	虚拟机可以轻松迁移到新主机	容器被销毁并重新创建而不是移动
创建速度	创建 VM 需要相对较长的时间	可以在几秒钟内创建容器
资源使用	GB级别	MB级别

Docker 中的虚拟化是依赖于 Windows 和 Linux 内核的，在 Windows 上会要求开启 Hyper-V，在 Linux 上需要依赖 namespace 和 cgroups 等，因此这里就不过多介绍 Docker 了，后面主要介绍 Linux 上的虚拟化技术。

传统虚拟化部署方式

传统虚拟化方式是在硬件抽象级别虚拟化，其特点是 虚拟化程度高。



传统虚拟化方式的优点是：

1，虚拟机之间通过虚拟化技术隔离互不影响 2，物理机上可部署多台虚拟机，提升资源利用率 3，应用资源分配、扩容通过虚拟管理器直接可配置 4，支持快照、虚拟机克隆多种技术，快速部署、容灾减灾

传统虚拟化部署方式的缺点：

1，资源占用高，需要额外的操作系统镜像，需要占用GB级别的内存以及数十GB存储空间。2，启动速度慢，虚拟机启动需要先启动虚拟机内操作系统，然后才能启动应用。3，性能影响大，应用 => 虚拟机操作系统=> 物理机操作系统=> 硬件资源

Linux 虚拟化

本节简单地讲解 Docker 的实现原理，读者可以从中了解 Linux 是如何隔离资源的、Docker 又是如何隔离的。

我们知道，操作系统是以一个进程为单位进行资源调度的，现代操作系统为进程设置了资源边界，每个进程使用自己的内存区域等，进程之间不会出现内存混用。

Linux 内核中，有 **cgroups** 和 **namespaces** 可以为进程定义边界，使得进程彼此隔离。

Linux-Namespace

在容器中，当我们使用 `top` 命令或 `ps` 命令查看机器的进程时，可以看到进程的 `Pid`，每个进程都有一个 `Pid`，而机器的所有容器都具有一个 `Pid = 1` 的基础，但是为什么不会发生冲突？容器中的进程可以任意使用所有端口，而不同容器可以使用相同的端口，为什么不会发生冲突？这些都是资源可以设定边界的表现。

在 Linux 中，`namespace` 是 Linux 内核提供的一种资源隔离技术，可以将系统中的网络、进程环境等进行隔离，使得每个 `namespace` 中的系统资源不再是全局性的。目前有以下 6 种资源隔离，Docker 也基本在这 6 种资源上对容器环境进行隔离。

读者可以稍微记忆一下这个表格，后面会使用到。

namespace	系统调用参数	隔离内容
UTS	<code>CLONE_NEWUTS</code>	主机名和域名
IPC	<code>CLONE_NEWIPC</code>	信号量、消息队列、共享内存
PID	<code>CLONE_NEWPID</code>	进程编号
Network	<code>CLONE_NEWNET</code>	网络设备、网络栈、端口
Mount	<code>CLONE_NEWNS</code>	文件系统挂载
User	<code>CLONE_NEWUSER</code>	用户和用户组

[info] 关于 Mount

`namespace` 的 Mount 可以实现将子目录挂载为根目录。

unshare

Linux 中，`unshare` 命令行程序可以创建一个 `namespace`，并且根据参数创建在 `namespace` 中隔离各种资源，在这里我们可以使用这个工具简单地创建一个 `namespace`。

为了深刻理解 Linux 中的 `namespace`，我们可以在 Linux 中执行：

```
unshare --pid /bin/sh
```

--pid 仅隔离进程。

这命令类似于 `docker run -it {image}:{tag} /bin/sh`。当我们执行命令后，终端会进入一个 `namespace` 中，执行 `top` 命令查看进程列表。

```
PID USER      PR NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1 root      20  0 160188  8276  5488 S  0.0  0.4  9:35.58 systemd
 2 root      20  0      0      0      0 S  0.0  0.0  0:00.08 kthreadd
 3 root      20 -20      0      0      0 I  0.0  0.0  0:00.00 rcu_gp
 4 root      20 -20      0      0      0 I  0.0  0.0  0:00.00 rcu_par_gp
```

可以看到，进程 `PID` 是从 1 开始的，说明在这个 `namespace` 中，与主机的进程是隔离开来的。

这个命令中，只隔离了进程，因为并没有隔离网络，因此当我们执行 `netstat --tlap` 命令时，这个命名空间的网络跟其它命名空间的网络是相通的。

在执行 `unshare` 命令前，使用 `pstree` 命令查看进程树：

```
init---2*[init--init--bash]
|----init--init--bash--pstree
|----init--init--fsnotifier-wsl
|----init--init--server--14*[{server}]
`----2*[{init}]
```

为了方便比较，我们使用 `unshare --pid top` 创建一个 `namespace`，对比执行了 `unshare` 命令后：

```
$> pstree -lha
init
|-init
|  |-init
|  |  \_bash
|  |  \_sudo unshare --pid top
|  |  \_top
|-init
|  |-init
|  |  \_bash
|  |  \_pstree -lha
|-init
|  |-init
|  |  \_fsnotifier-wsl
|-init
|  |-init
|  |  \_bash
|-init
|  |-init
|  |  \_server --port 29687 --instance WSL-Ubuntu
|  |  \_14*[{server}]
`----2*[{init}]
```

而在 `namespace` 中，查看 `top` 显示的内容，发现：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	20	0	1904	1136	1020	S	0.0	0.0	0:08.38	init

通过进程树可以看到，不同 `namespace` 内的进程处于不同的树支，他们的进程 `PID` 也是相互独立的。其功能类似于 `Docker` 中的 `runc`。

由于笔者对 `Linux` 了解不深，这部分内容就不深入探究了。

在 `unshare` 命令中，`--pid` 参数创建隔离进程的命名空间，此外，还可以隔离多种系统资源：

- `mount`：命名空间具有独立的挂载文件系统；
- `ipc`: Inter-Process Communication (进程间通讯)命名空间，具有独立的信号量、共享内存等；
- `uts`: 命名空间具有独立的 `hostname`、`domainname`；
- `net`: 独立的网络，例如每个 `docker` 容器都有一个虚拟网卡；
- `pid`: 独立的进程空间，空间中的进程 `Pid` 都从 1 开始；

- **user:** 命名空间中有独立的用户体系，例如 Docker 中的 root 跟主机的用户不一样；
- **cgroup:** 独立的用户分组；

Go 简单实现 进程隔离

在前面我们使用了 `unshare` 创建命名空间，在这里我们可以尝试使用 Go 调用 Linux 内核的 `namespace`，通过编程代码创建隔离的资源空间。

Go 代码示例如下：

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {

    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS |
            syscall.CLONE_NEWPID |
            syscall.CLONE_NEWSOCKET |
            syscall.CLONE_NEWIPC |
            syscall.CLONE_NEWNET |
            syscall.CLONE_NEWPID |
            syscall.CLONE_NEWUSER,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if err := cmd.Run(); err != nil {
        log.Fatalln(err)
    }
}
```

[info] 提示

前面已经提到过 UTS 等资源隔离，读者可以参考表格中的说明，对照代码理解 `Cloneflags` 的作用。

代码示例参考 陈显鹭《自己动手写 Docker》一书。

在这个代码中，我们启动了 Linux 中的 `sh` 命令，开启一个新的进程，这个进程将会使用新的 IPC、PID 等隔离。

读者可以在 Linux 中，执行 `go run main.go`，即可进入新的命名空间。

```
1: lo: <LOOPBACK,NOARP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback brd 00:00:00:00:00:00
    queueingdisc global eth0
    valid_lft forever preferred_lft forever
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
        valid_lft forever preferred_lft forever
        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
            valid_lft forever preferred_lft forever
            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                valid_lft forever preferred_lft forever
                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                    valid_lft forever preferred_lft forever
                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                        valid_lft forever preferred_lft forever
                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                            valid_lft forever preferred_lft forever
                            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                valid_lft forever preferred_lft forever
                                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                    valid_lft forever preferred_lft forever
                                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                        valid_lft forever preferred_lft forever
                                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                            valid_lft forever preferred_lft forever
                                            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                valid_lft forever preferred_lft forever
                                                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                    valid_lft forever preferred_lft forever
                                                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                        valid_lft forever preferred_lft forever
                                                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                            valid_lft forever preferred_lft forever
                                                            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                valid_lft forever preferred_lft forever
                                                                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                    valid_lft forever preferred_lft forever
                                                                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                        valid_lft forever preferred_lft forever
                                                                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                            valid_lft forever preferred_lft forever
                                                                            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                valid_lft forever preferred_lft forever
                                                                                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                    valid_lft forever preferred_lft forever
                                                                                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                        valid_lft forever preferred_lft forever
                                                                                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                        inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                            valid_lft forever preferred_lft forever
                                                                                            link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                            inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                                valid_lft forever preferred_lft forever
                                                                                                link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                                inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                                    valid_lft forever preferred_lft forever
                                                                                                    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
                                                                                                    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
                                                                                                        valid_lft forever preferred_lft forever
                                                                                                        link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
................................................................
```

限于个人水平和篇幅有限，关于 namespace 的介绍就到这里。

cgroups 硬件资源隔离

前面提到的 `namespace` 是逻辑形式使得进程之间相互不可见，形成环境隔离，这跟 Docker 容器的日常使用是一样的，隔离根目录，隔离网络，隔离进程 PID 等。

当然，Docker 处理环境隔离外，还能限制每个容器使用的物理资源，如 CPU、内存等，这种硬件资源的限制是基于 Linux 内核的 `cgroups` 的。

在 Docker 中限制容器能够使用的资源量参数示例：

```
-m 4G --memory-swap 0 --cpu-period=1000000 --cpu-quota=8000000
```

`cgroups` 是 `control groups` 的缩写，是 Linux 内核提供的一种可以进程所使用的物理资源的机制。

`cgroups` 可以控制多种资源，在 `cgroups` 中每种资源限制功能对应一个子系统，可以使用命令查看：

```
mount | grep cgroup
```

```
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=system)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
```

[info] 提示

每种子系统的功能概要如下：

- `blkio` — 该子系统对进出块设备的输入/输出访问设置限制，如 USB 等。
- `cpu` — 该子系统使用调度程序来提供对 CPU 的 cgroup 任务访问。
- `cpuacct` — 该子系统生成有关 cgroup 中任务使用的 CPU 资源的自动报告。
- `cpuset` — 该子系统将单个 CPU 和内存节点分配给 cgroup 中的任务。
- `devices` — 该子系统允许或拒绝 cgroup 中的任务访问设备。
- `freezer` — 该子系统在 cgroup 中挂起或恢复任务。
- `memory` — 该子系统对 cgroup 中的任务使用的内存设置限制，并生成有关自动报告。
- `net_cls` — 允许 Linux 流量控制器（`tc`）识别源自特定 cgroup 任务的数据包。
- `net_prio` — 该子系统提供了一种动态设置每个网络接口的网络流量优先级的方法。
- `ns` — 命名空间子系统。
- `perf_event` — 该子系统识别任务的 cgroup 成员资格，可用于性能分析。

详细内容请参考：[redhat 文档](#)

我们也可以使用 `lssubsys` 命令，查看内核支持的子系统。

```
$> lssubsys -a
cpuset
cpu
cpuacct
blkio
memory
devices
freezer
net_cls
perf_event
net_prio
hugetlb
pids
rdma
```

[info] 提示

Ubuntu 可以使用 `apt install cgroup-tools` 安装工具。

为了避免篇幅过大，读者只需要知道 Docker 限制容器资源使用量、CPU 核数等操作，其原理是 Linux 内核中的 `cgroups` 即可，笔者这里不再赘述。

聊聊虚拟化

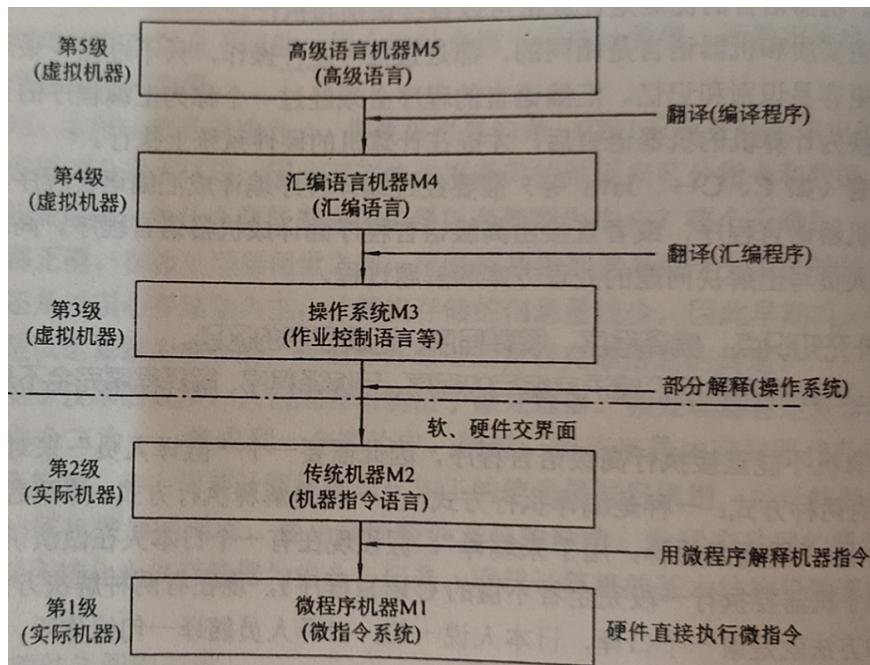
本节内容将从底层角度，聊聊虚拟化。

理论基础

计算机层次结构

从语言角度，一台由软硬件组成的通用计算机系统可以看作是按功能划分的多层次机器级组成的层次结构。

如果从语言角度来看，计算机系统的层次结构可用下图所示。



【图来源：《计算机组成原理》天勤考研 1.2.5 计算机系统的层次结构】

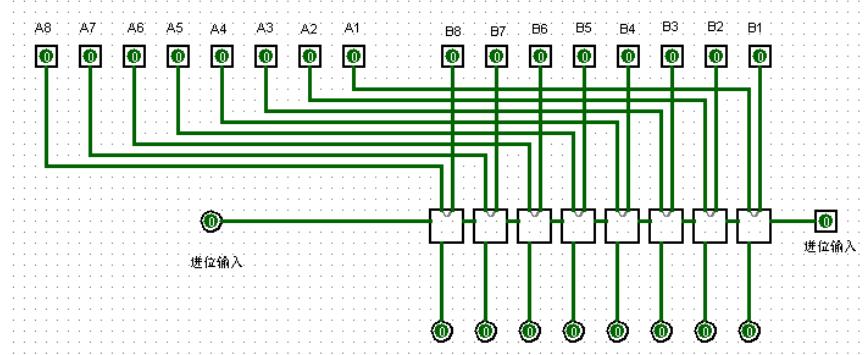
我们平时使用的笔记本、安卓手机、平板电脑、Linux 服务器等，虽然不同机器的系统和部分硬件差异很大，但是其系统结构是一致的。从 CPU 中晶体管、寄存器到 CPU 指令集，再到操作系统、汇编，现在使用的通用计算机基本上这种结构。

下面讲解一下不同层次的主要特点。

计算机的最底层是硬联逻辑级，由门电路，触发器等逻辑电路组成，特征是使用极小的元件构成，表示了计算机中的 0、1。



微程序是使用微指令编写的，一个微程序即一个机器指令，一般直接由硬件执行，它可以表示一个最简单的操作。例如一个加法指令，由多个逻辑元件构成一个加法器，其元件组成如下图所示(图中为一个 8 位全加器)。



传统机器语言机器级是处理器的指令集所在，我们熟知的 X86、ARM、MIPS、RISC-V 等指令集，便是在这个层次。程序员使用指令集中的指令编写的程序，由低一层微程序解释。

操作系统机器层是从操作系统基本功能来看的，操作系统需要负责管理计算机中的软硬件资源，如内存、设备、文件等，它是软硬件的交互界面。常用的操作系统有 **Windows**、**Linux**、**Unix** 等。这个层次使用的语言是机器语言，即 0、1 组成的二进制代码，能够由计算机直接识别和执行。

汇编语言机器层顾名思义是汇编语言所在的位置，汇编语言与处理器有关，相同类型的处理器使用的汇编语言集是一致的。汇编语言需要被汇编语言程序变换为等效的二进制代码目标程序。由于计算机中的资源被操作系统所管理，因此汇编语言需要在操作系统的控制下进行。

到了高级语言机器层，便是我们使用的 **C**、**C++** 等编程语言，高级语言是与人类思维相接近的语言。

软硬件实现等效

计算机的某些功能即可以由硬件实现，也可以由软件来实现。即软件和硬件在功能意义上是等效的。

一个功能使用硬件来实现还是使用软件来实现？

硬件实现：速度快、成本高；灵活性差、占用内存少。

软件实现：速度低、复制费用低；灵活性好、占用内存多。

虚拟化技术是将原本硬件实现的功能，使用软件来实现，它们在性能、价格、实现的难易程度是不同的。一个功能既可以使用硬件实现，也可以使用软件实现，也可以两者结合实现，可能要根据各种人力成本、研发难度、研发周期等考虑。

虚拟化

虚拟化（技术）或虚拟技术是一种资源管理技术，将计算机的各种实体资源（CPU、内存、磁盘空间、网络适配器等），予以抽象、转换后呈现出来，并可供分割、组合为一个或多个计算机配置环境。

不同层次的虚拟化

我们应该在很多书籍、文章中，了解到虚拟机跟 **Docker** 的比较，了解到 **Docker** 的优点，通过 **Docker** 打包镜像后可以随时在别的地方运行而不需要担心机器的兼容问题。但是 **Docker** 的虚拟化并不能让 **Linux** 跑 **Windows** 容器，也不能让 **Windows** 跑 **Linux** 容器，更不可能让 **x86** 机器跑 **arm** 指令集的二进制程序。但是 **VMware** 可以在 **Windows** 运行 **Linux**、**Mac** 的镜像，但 **VMWare** 也不能由 **MIPS** 指令构建的 **Linux** 系统。

Docker 和 **VMware** 都可以实现不同程度的虚拟化，但也不是随心所欲的，它们虚拟化的程度相差很大，因为它们是在不同层次进行虚拟化的。

Library Level	库级别虚拟化，如Wine、WSL
Programming Language Level	编程语言级别虚拟化，如CLR、JVM
Operating System Level	操作系统级别虚拟化，如 Docker
Hardware Abstraction Level	硬件抽象级别虚拟化，如VMware_ESXi、Hyper-V
Instruction Set Architecture Level	指令集级别虚拟化，如Bochs、QEMU

[Info] 提示

许多虚拟化软件不单单是在一个层面上，可能具有多种层次的虚拟化能力。

在指令集级别虚拟化中，从指令系统上看，就是要在一种机器上实现另一种机器的指令系统。例如，QEMU 可以实现在 X64 机器上模拟 ARM32/64、龙芯、MIPS 等处理器。

虚拟化程度在于使用硬件实现与软件实现的比例，硬件部分比例越多一般来说性能就会越强，软件部分比例越多灵活性会更强，但是性能会下降，不同层次的实现也会影响性能、兼容性等。随着现在计算机性能越来越猛，很大程度上产生了性能过剩；加之硬件研发的难度越来越高，越来越难突破，非硬件程度的虚拟化将会越来越广泛。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-10-31 07:33:23

- 1.3 了解 Docker 网络
 - Docker 的四种网络模式
 - bridge 模式
 - none 模式
 - host 模式
 - container 模式

1.3 了解 Docker 网络

本章将会简单地讲述 Docker 中的网络，对于 CNM、Libnetwork 这些，限于笔者个人水平，将不会包含在内。

Docker 的四种网络模式

Docker 有 bridge、none、host、container 四种网络模式，提供网络隔离、端口映射、容器间互通网络等各种支持，下面开门见山地直接介绍这四种网络模式。

这四种网络模式可以通过启动容器的时候指定，其命令或参数个数如下：

网络模式	参数	说明
host模式	--net=host	容器和宿主机共享 Network namespace。
container 模式	--net={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	默认为该模式，通过 -p 指定端口映射。

这四种模式可以理解成 Docker 怎么虚拟化容器的网络，隔离程度和共享程度。

bridge 模式

使用 Docker 创建一个 bridge 模式的容器命令格式如下：

```
docker run -itd -p 8080:80 nginx:latest
```

bridge 模式称为网桥模式，首先 Docker 会在主机上创建一个名为 docker0 的虚拟网桥，这个虚拟网络处于七层网络模型的数据链路层，每当创建一个新的容器时，容器都会通过 docker0 与主机的网络连接，docker0 相当于网桥。

使用 bridge 模式新创建的容器，其内部都有一个虚拟网卡，名为 eth0，容器之间可以通过 172.17.x.x 相互访问。

一般情况下，网桥默认 IP 范围是 172.17.x.x，可以在宿主机执行 ifconfig 命令查看所有网卡，里面会包含 Docker 容器的虚拟网卡，可以查看某个容器的 ip。在容器中，也可以使用 ifconfig 命令查看自身的容器 ip：

```

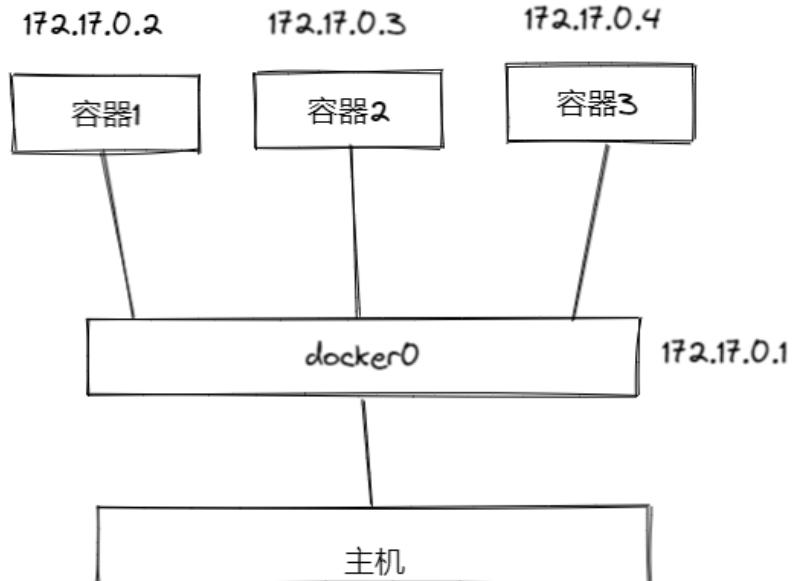
root@cda6958393cb:/var# ./ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
        ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
        RX packets 347 bytes 9507996 (9.5 MB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 278 bytes 22384 (22.3 KB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        loop txqueuelen 1000 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

可以看到，此容器的 ip 是 172.17.0.2。

使用了 **bridge** 创建的容器，其网络与主机以及其他容器隔离，以太网接口、端口、路由表以及 DNS 配置都是独立的。每个容器都好像是一个独立的主机，这便是 **bridge**（网桥）的作用。但是因为 **docker0** 的存在，对于容器来说，可以通过 ip 访问别的容器。



容器1可以通过 172.17.0.3 访问容器2，同样，主机也可以使用这个 ip 访问容器2中的服务。

[Error] 提示

`bridge` 模式是默认模式，即使是使用 `docker run -itd nginx:latest` 命令启动容器，也会创建一个虚拟 IP。

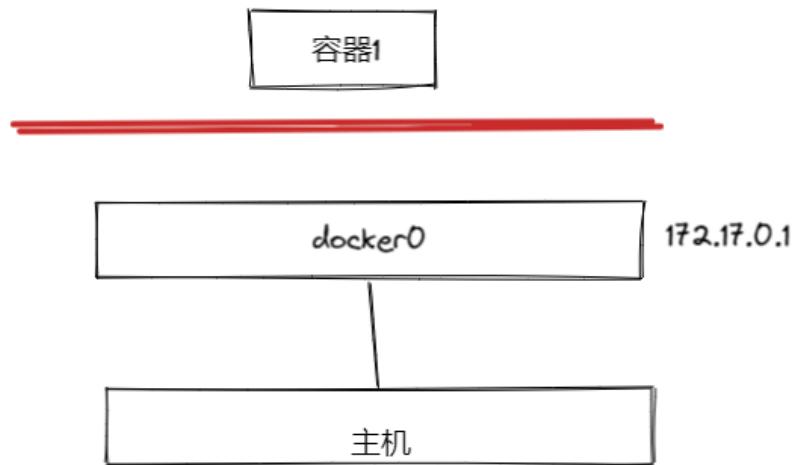
none 模式

这种网络模式下容器只有 `lo` 回环网络，没有其他网卡，这种类型的网络没有办法联网，外界也无法访问它，封闭的网络能很好地保证容器的安全性。

创建 `none` 网络的容器：

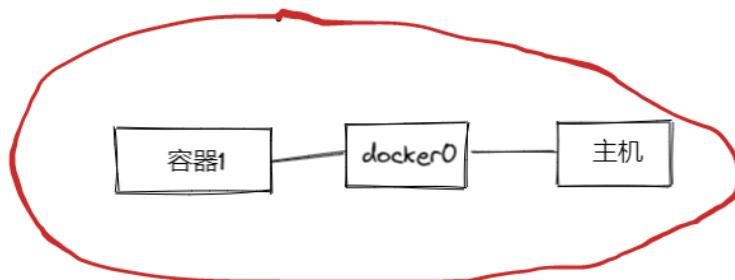
```
docker run -itd --net=none nginx:latest
```

```
root@5a67da130f62:/var# ./ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        loop txqueuelen 1000 (Local Loopback)
        RX packets 0 bytes 0 (0.0 B)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```



host 模式

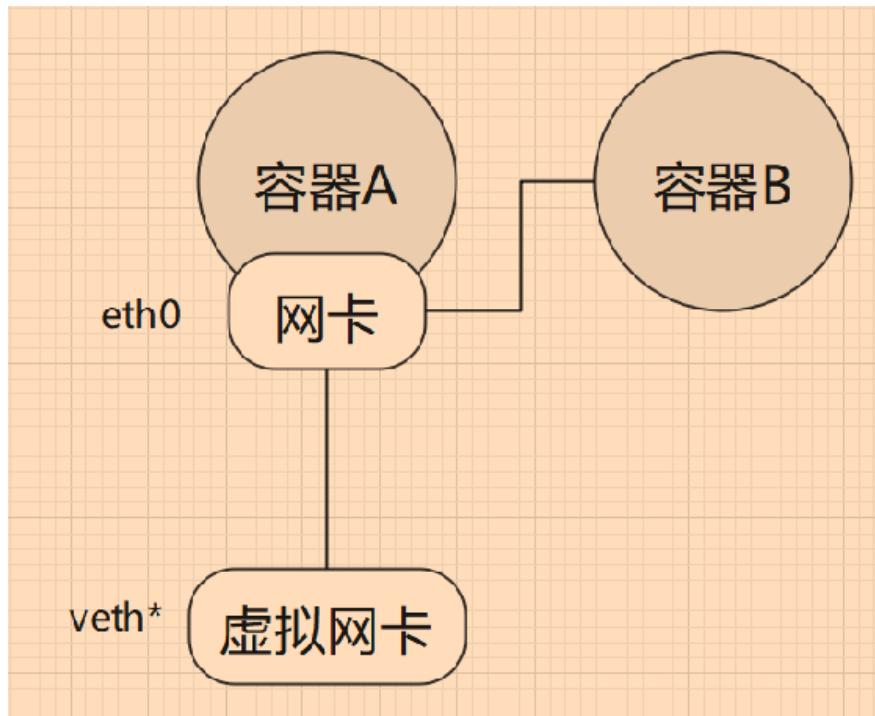
host 模式会让容器与主机共享网络，此时映射的端口可能会产生冲突，但是容器的其余部分(文件系统、进程等)依然是隔离的，此时容器与宿主机共享网络。



container 模式

container 模式可以让多个容器之间相互通讯，即容器之间共享网络。

首先启动一个 A 容器，A 一般为 bridge 网络，接着 B 使用 `--net={id}` 连接到 A 中，使用 A 的虚拟网卡，此时 A、B 共享网络，可以接着加入 B、C、D 等容器。



Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-10-30 13:09:53

- 1.4 容器与 Pod
 - 什么是容器化应用
 - Pod
 - 容器与 Pod 的区别
 - 节点

1.4 容器与 Pod

现在 Docker 的流行程度越来越高，越来越多的公司使用 Docker 打包和部署项目。但是也有很多公司只是追求新技术，将以前的单体应用直接打包为镜像，代码、配置方式等各方面保持不变，使用 Docker 后，并没有带来多大的便利，反而使得配置、启动过程变得更加繁杂，更难调试。

本章将讨论容器与 Pod 的关系，了解如何更好地将应用容器化。

什么是容器化应用

containerized applications 指容器化的应用，我们常常说使用镜像打包应用程序，使用 Docker 发布、部署应用程序，那么当你的应用成功在 Docker 上运行时，称这个应用是 **containerized applications**。

定义：

[Success] 定义

Containerized applications are bundled with their required libraries, binaries, and configuration files into a container.

容器化的应用程序与它们所需的库、二进制文件和配置文件绑定到一个容器中。

通常，容器都包含一个应用程序，以及正确执行二进制程序所需的依赖库、文件等，例如 Linux 文件系统+应用程序组成一个简单的容器。通过将容器限制为单个进程，问题诊断和更新应用程序都变得更加容易。与 VM(虚拟机)不同，容器不包含底层操作系统，因此容器被认为是轻量级的。Kubernetes 容器属于开发领域。

容器在操作系统之上，提供了 CPU、内存、网络、存储等资源的虚拟化，为应用在不同服务器里提供了一致的运行时环境。开发者可以通过容器创建一个可预测的环境，能够保证在开发、调试、生产时的环境都是一致的，减少开发团队和运维团队可以减少调试和诊断问题时，因环境差异带来的麻烦。同时，应用运行在一个沙盒中，对应用和系统进行了隔离，提高了安全性，还能限制应用程序使用的计算资源。

当然，并不是说能够将一个应用程序打包到容器中运行，就可以鼓吹产品；并不是每个应用程序都是容器化的优秀对象，例如在 DDD 设计中被称为大泥球的应用程序，具有设计复杂、依赖程度高、程序不稳定等确定，这种难以迁移、难以配置的应用程序明显是失败的产品。

在多年经验中，许多开发者对容器化技术进行了总结，这些强有力的经验、理论形成十二个云计算应用程序因素指导原则：

1. **Codebase:** One codebase tracked in revision control, many deploys

代码库：一个代码库可以在版本控制和多份部署中被跟踪。一般使用 `github` 等对代码进行管理。

2. Dependencies: Explicitly declare and isolate dependencies

依赖项：显式声明和隔离依赖项。

3. Config: Store config in the environment

配置：在环境中存储配置。

4. Backing services: Treat backing services as attached resources

支持服务：将支持服务视为附加资源(可拓展，而不是做成大泥球)。

5. Build, release, run: Strictly separate build and run stages

构建、发布、运行：严格区分构建和运行阶段(连 `Debug`、`Release` 都没有区分的产品是真的垃圾)。

6. Processes: Execute the app as one or more stateless processes

过程：应用程序作为一个或多个无状态过程执行。

7. Port binding: Export services via port binding

端口绑定：可通过端口绑定服务对外提供服务。

8. Concurrency: Scale out via the process model

并发性：通过 `process` 模型进行扩展。

9. Disposability: Maximize robustness with fast startup and graceful shutdown

可处理性：快速启动和完美关机，最大限度地增强健壮性。

10. Dev/prod parity: Keep development, staging, and production as similar as possible

开发/生产一致：尽可能保持开发中、演示时和生产时的相似性。

11. Logs: Treat logs as event streams

Logs：将日志视为事件流。

12. Admin processes: Run admin/management tasks as one-off processes

管理流程：将管理/管理任务作为一次性流程运行。

上述内容可能有笔者翻译不到位的地方，读者可阅读原文了解：

<https://12factor.net/>

容器位于开发人员技能列表之中，开发人员需要掌握如何容器化应用。

另外，在一个产品中，好的容器化规范或方法，具有以下特点：

- 使用声明式的格式进行设置自动化，以最大限度地减少新开发人员加入项目的时间和成本；
- 与底层操作系统之间有一个干净的契约(资源隔离、统一接口)，在执行环境之间提供最大的可移植性；
- 适合部署在现代云平台上，无需服务器和系统管理；

- 最大限度地减少开发和生产之间的差异，实现持续部署以实现最大敏捷性；
- 并且可以在不对工具、架构或开发实践进行重大更改的情况下进行扩展。

在制作云原生应用的过程中，可以参考云计算应用程序因素指导原则，设计更加优秀的产品。

Pod

最简单的说法就是将多个容器打包起来一起运行，这个整体就是 Pod。

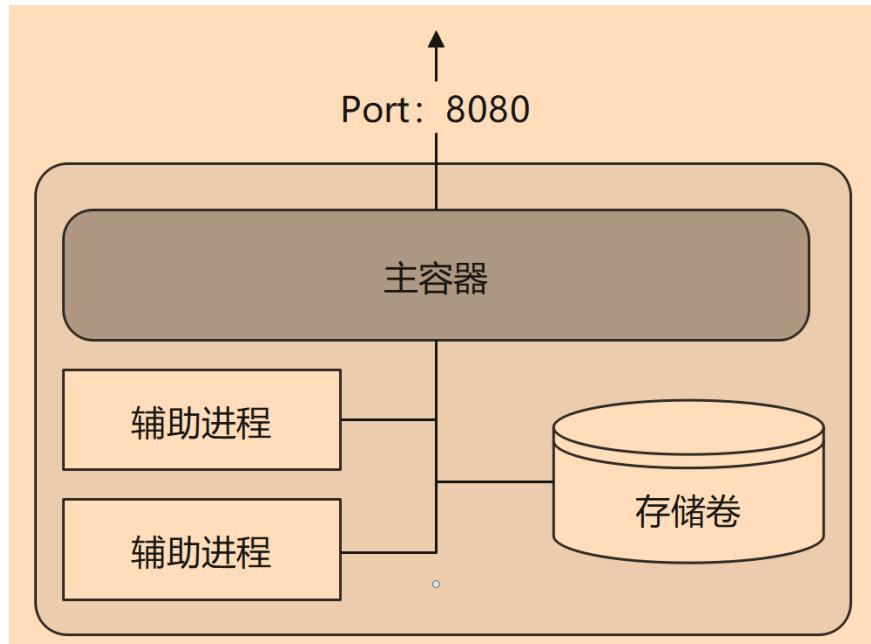
[Info] 提示

在上一章的 Docker 网络中，介绍了 container 网络模式，Pod 正是通过这种网络模式，让 Pod 中的容器共享网络，也就是说，Pod 中的容器，网络是互通的，容器之间不能使用相同的端口。

Pod 是 Kubernetes 集群中最小的执行单位。在 Kubernetes 中，容器不直接在集群节点上运行，而是将一个或多个容器封装在一个 Pod 中，接着将 Pod 调度到节点上运行，这些容器会一起被运行、停止，它们是一个整体。

Pod 中的所有容器共享相同的资源和本地网络，从而简化了 Pod 中应用程序之间的通讯。在 Pod 中，所有容器中的进程共享网络，可以通过 `127.0.0.1`、`localhost` 相互进行访问。详见 3.1 章中 "Pod 共享网络和存储" 一节。

一个简单的 Pod，其结构如下：

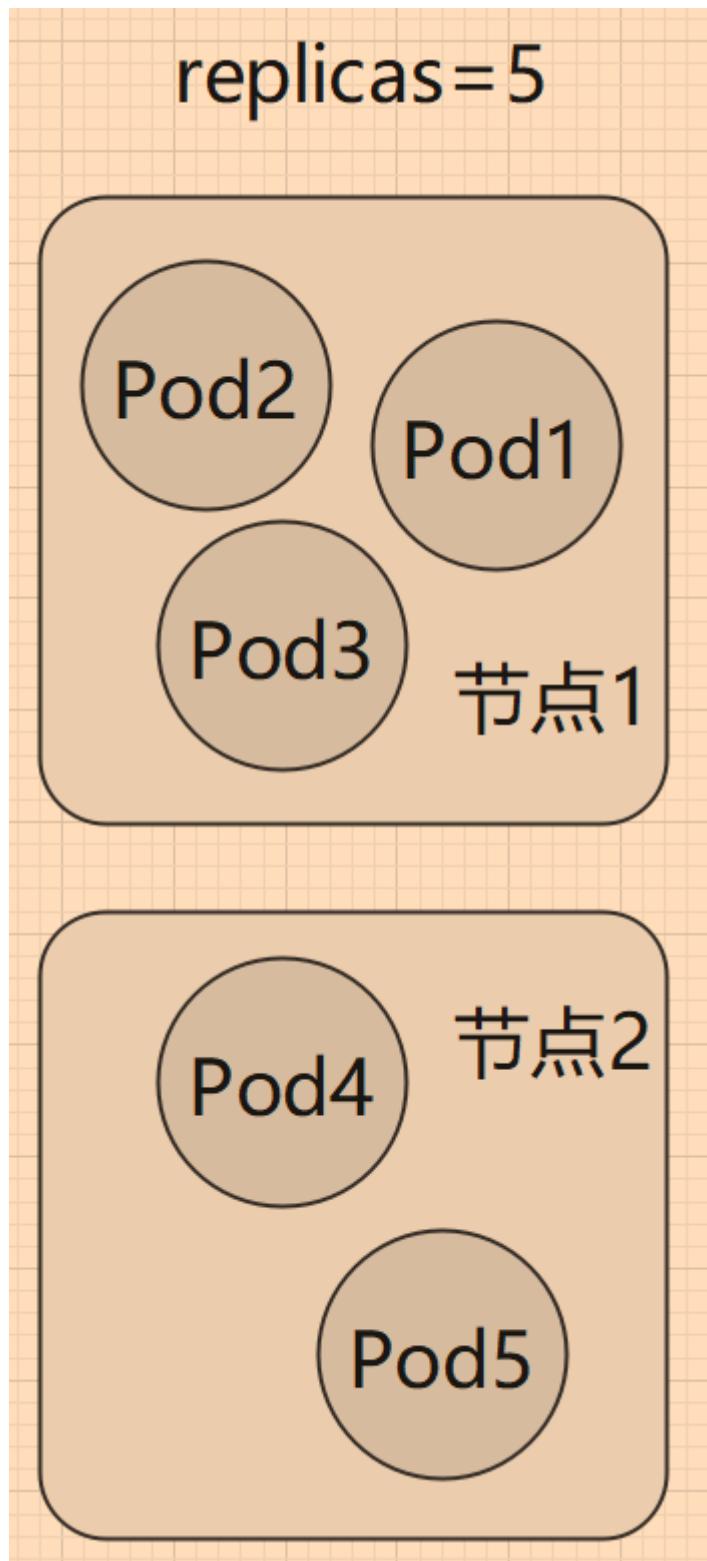


[Info] 提示

Pod 启动时会启动一个容器，K8S 给这个容器分配虚拟 IP，接着，其他容器使用 container 网络模式，连接到这个容器中，此时有容器共享网络。

随着 Pod 负载的增加，Kubernetes 可以自动复制 Pod 以达到预期的可拓展性(部署更多的 Pod 提供相同的服务，负载均衡)。因此，设计一个尽可能精简的 Pod 是很重要的，降低因复制扩容、减少收缩过程中带来的资源损失。

前面提到，容器应当是无状态的，所以拓展 Pod 时，每个实例都提供了一模一样的服务，这些 Pod 分配到不同的节点上，可以利用更多的 CPU、内存资源。

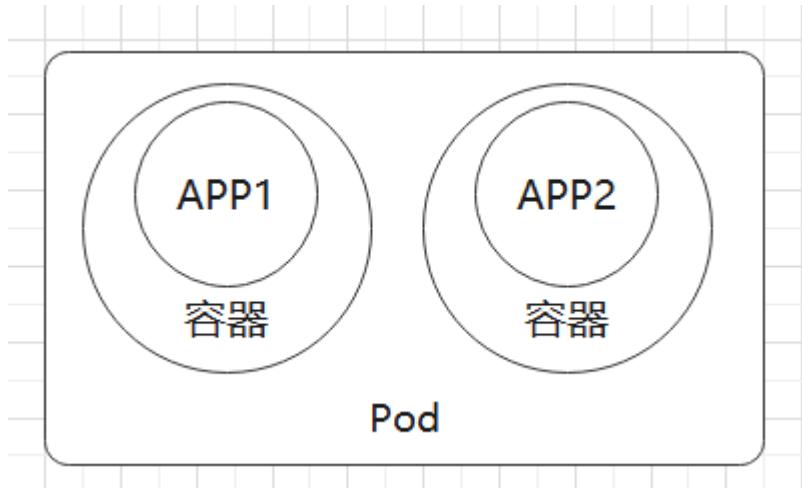


在第三章中，我们会更加详细地学习 Pod，这里就不再赘述。

容器与 Pod 的区别

容器包含执行特定流程或函数所需的代码(编译后的二进制可执行程序)。在 **Kubernetes** 之前，可以直接在物理或虚拟服务器上运行容器，但是缺乏 **Kubernetes** 集群所提供的可伸缩性和灵活性。

Pod 为容器提供了一种抽象，可以将一个或多个应用程序包装到一个 **Pod** 中，而 **Pod** 是 **Kubernetes** 集群中最小的执行单元。例如 **Pod** 可以包含初始化容器，这些容器为其它应用提供了准备环境，然后在应用程序开始执行前终结。**Pod** 是集群中复制的最小单位，**Pod** 中的容器作为整体被扩展或缩小。



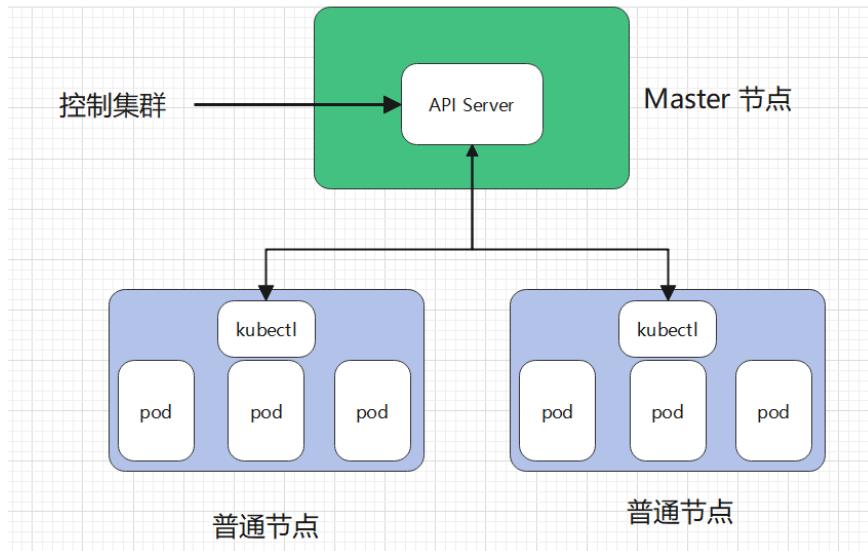
例如对应前端分离的项目，可能不需要把前端文件和后端程序放在一起，而是分别放在两个容器中。然后通过 **Pod**，将这两个容器作为一组服务打包在一起。

节点

Pod 是 **Kubernetes** 中最小的执行单元，而 **Node** 是 **Kubernetes** 中最小的计算硬件单元，节点可以是物理的本地服务器，也可以是虚拟机，节点即使宿主服务器，可以运行 **Docker** 的机器。

与容器一样，**Node** 提供了一个抽象层。多个 **Node** 一起工作形成了 **Kubernetes** 集群，它可以根据需求的变化自动分配工作负载，增加或减少在节点上的 **Pod** 数量。如果 **A** 节点和 **B** 节点的硬件资源是一致的，那么 **A**、**B** 两个节点是等价的，如果 **A** 节点失败，它将自动从集群中移除，由 **B** 节点接管，不会出现问题。

每个节点都运行着一个名为 **kubelet** 的组件，它是节点的主要组件，**Kubernetes** 与集群控制平面组件(**API Server**)通信，所有对节点有影响的操作都会通过 **kubectl** 控制此节点。**kubelet** 也是 **master** 节点跟 **worker** 节点之间直接通讯的唯一组件。

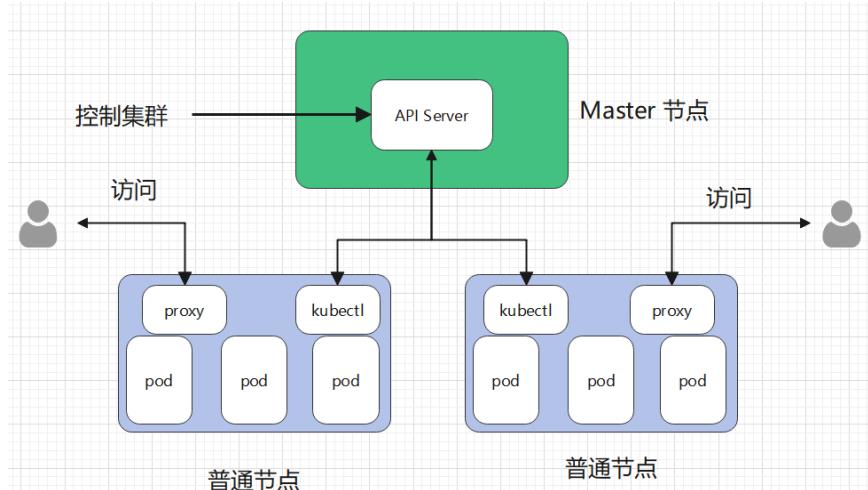


kubelet的一些功能有：

- 在节点上创建、更新、删除容器；
- 参与调度 Pod；
- 为容器创建和挂载卷；
- 使用命令查看 Pod、容器，例如 `exec`、`log` 等时，需要通过 kubelet；

例如，集群有 A、B 两个节点，Pod 部署在哪里了，这不是用户关心的事情，用户在想看到容器的日志，可以随便找集群中的一台主机，执行命令，Kubernetes 会自动寻找容器所在的节点，然后 kubectl 取得需要的内容。

另外节点上还有 proxy，主要是为 Pod 提供代理服务，外界可通过此代理，使用节点的 IP 访问 Pod 中的容器。



Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-10-31 07:48:14

- 1.5 Kubernetes 入门基础
 - Kubernetes 是什么
 - Kubernetes 集群的组成
 - Kubernetes 结构
 - 组件
 - Master 节点
 - kube-apiserver
 - etcd
 - kube-scheduler
 - kube-controller-manager

1.5 Kubernetes 入门基础

我们要学习 Kubernetes，就有首先了解 Kubernetes 的技术范围、基础理论知识库等，要学习 Kubernetes，肯定要有入门过程，在这个过程中，学习要从易到难，先从基础学习。

接下来笔者将为大家讲解 Kubernetes 各方面的知识，让读者了解 Kubernetes 是什么。

Kubernetes 是什么

在 2008 年，**LXC (Linux containers)** 发布第一个版本，这是最初的容器版本；2013 年，Docker 推出了第一个版本；而 Google 则在 2014 年推出了 **LMCTFY**。

为了解决大集群(Cluster)中容器部署、伸缩和管理的各种问题，出现了 Kubernetes、Docker Swarm 等软件，称为 容器编排引擎。

容器的产生解决了很多开发、部署痛点，但随着云原生、微服务的兴起，纯 Docker 出现了一些管理难题。我们先思考一下，运行一个 Docker 容器，只需要使用 `docker run ...` 命令即可，这是相当简单(*relatively simple*)的方法。

但是，要实现以下场景，则是困难的：

- 跨多台主机的容器相互连接(connecting containers across multiple hosts)
- 拓展容器(scaling containers)
- 在不停机的情况下配置应用(deploying applications without downtime)
- 在多个方面进行服务发现(service discovery among several aspects)

Kubernetes 是 Google 基于十多年的生产环境运维经验，开发出的一个生产级别的容器编排系统。在 Kunernetes 文档中，这样描述 Kubernetes：

[Success]

"an open-source system for automating deployment, scaling, and management of containerized applications".

“一个自动化部署、可拓展和管理容器应用的开源系统”

Google 的基础设施在虚拟机(Virtual machines)技术普及之前就已经达到了很大的规模，高效地使用集群和管理分布式应用成为 Google 挑战的核心，而容器技术提供了一种高效打包集群的解决方案。

多年来，Google 一直使用 Borg 来管理集群中的容器，积累了大量的集群管理经验和运维软件开发能力，Google 参考 Borg，开发出了 Kubernetes，即 Borg 是 Kubernetes 的前身。（但是 Google 目前还是主要使用 Borg）。

Kubernetes 从一开始就通过一组基元(primitives)、强大的和可拓展的 API 应对这些挑战，添加新对象和控制器地能力可以很容易地地址各种各样的产品需求(production needs)。

编排管理是通过一系列的监控循环控制或操作的；每个控制器都向询问对象状态，然后修改它，直至达到条件为止。容器编排是管理容器的最主要的技术。Dockers 也有其官方开发的 swarm 这个编排工具，但是在 2017 年的容器编排大战中，swarm 败于 Kubernetes。

Kubernetes 集群的组成

在 Kubernetes 中，运行应用程序的环境处于虚拟化当中，因此我们一般不谈论硬件。

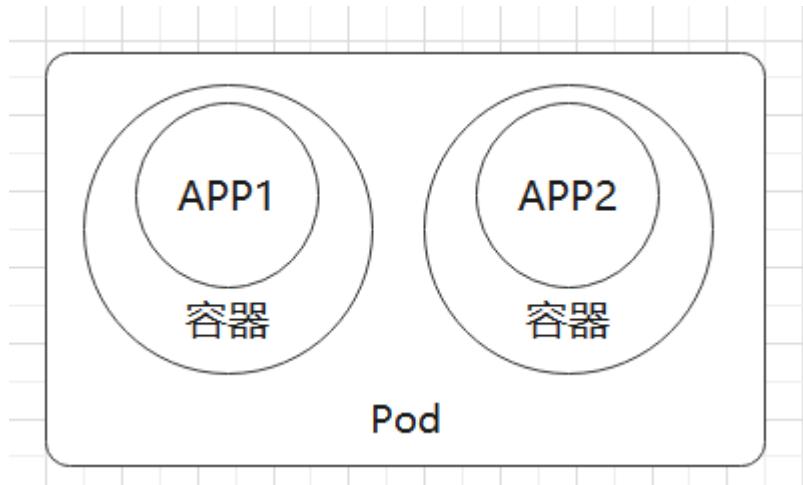
我们谈起 Kubernetes 和应用部署时，往往会涉及到容器、节点、Pods 等概念，它们共同工作来管理容器化(containerized)应用的部署和执行，但是各种各样的术语，令人眼花缭乱。为了更好地摸清 Kubernetes，下面我们将列举这些有边界的对象。

成分	名称
Cluster	集群
Node	节点
Pod	不翻译
Container	容器
Containerized Application	容器化的应用

在 Kubernetes 中，不同的对象其管理的范围、作用范围不同，它们的边界大小也不同。接下来的内容，按将从小到大的粒度介绍这些组成成分。

Pod

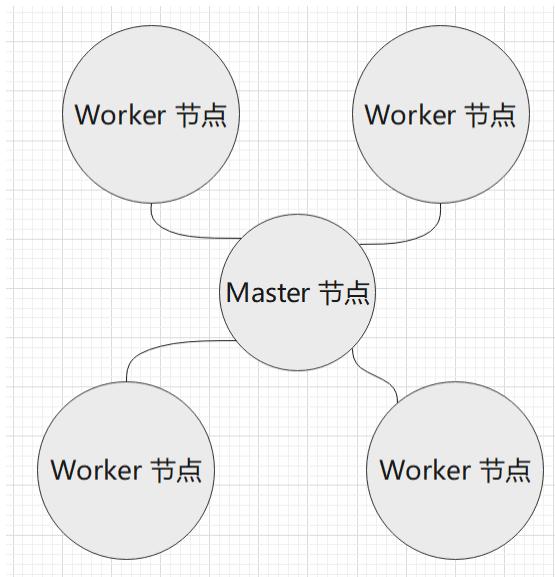
在上一章中已经介绍过，Pod 是 Kubernetes 中管理和调度的最小工作单位，Pod 中可以包含多个容器。这些容器会共享 Pod 中的网络等资源。当部署 Pod 时，会把一组关联性较强的容器部署到同一个节点上。



而节点则是指一台服务器、虚拟机等，运行着一个完整的操作系统，提供了CPU、内存等计算资源，一个节点可以部署多个Pod。



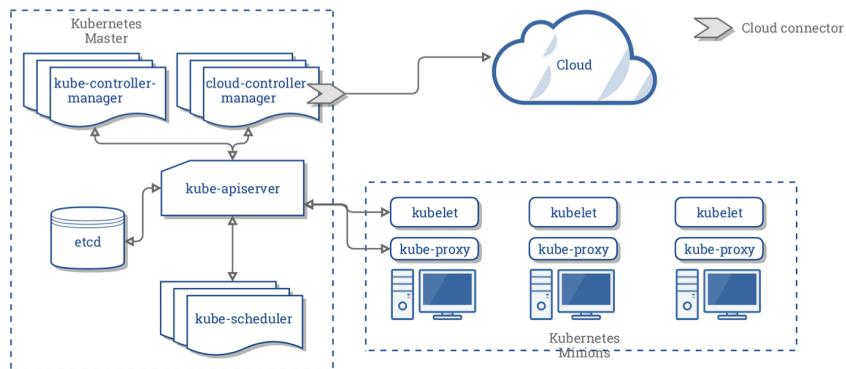
而一个集群(Cluster)之中，运行着N台服务器，即N个节点。这些节点有两种，一种是master节点，一种是worker节点。master节点运行着Kubernetes系统组件，而worker节点负责运行用户的程序。所有节点都归master管，我们通过命令、API的方式管理Kubernetes集群时，是通过发送命令或请求到master节点上的系统组件，然后控制整个集群。



另外，kubernetes 中有命名空间(namespace)的概念，这跟在 1.2 章中学习到的 Linux-namespace 类似，在一个集群中使用命名空间将不同的 Pod 隔离开来。但是 Kubernetes 中，不同 namespace 的 Pod 是可以相互访问的，它们不是完全隔离开的。

Kubernetes 结构

用图来表示体系结构，是阐述 Kubernetes 最快的方式，下面是一张称为 *Kubernetes Architecture graphic*。



上图是简单的 kubernetes 结构，左侧虚线方框中，是 master 节点，运行着各种各样的组件，master 节点负责控制整个集群，当然在很大的集群中也可以有多个 master 节点；而右侧是三个工作节点，负责运行我们的容器应用。这种结构一般称为 master-slave 结构，因为某些原因，在 Kubernetes 中后来改称为 master-minions。工作节点挂了没关系，master 节点会将故障节点上的业务自动在另一个节点上部署。

工作节点比较简单，在工作节点中，我们看到有 kubelet 和 kube-proxy 两个组件，这两个组件在上一章中接触过了，kubelet 和 kube-proxy 都是跟主节点的 kube-apiserver 进行通信的。kube-proxy 全称是 Kubernetes Service Proxy，负责组件之间的负载均衡网络流量。

在上图中，主节点由多个组件构成，结构比较复杂，主节点中记录了整个集群的工作数据，负责控制整个集群的运行。工作节点挂了没关系，但是主节点挂了，整个集群就挂了。因此，有条件的情况下，也应该设置多个主节点。

一个主节点中包含以下访问：

- 一个 API 服务(`kube-apiserver`)
- 一个调度器(`kube-scheduler`)
- 各种各样的控制器(上图有两个控制器)
- 一个存储系统(这个组件称为`etcd`)，存储集群的状态、容器的设置、网络配置等数据。

这张图片中还有很多东西，这里暂时不作讲解，我们在后面的章节再去学习那些 `Kubernetes` 中的术语和关键字。

组件

一个 `Kubernetes` 集群是由一组被称为节点的机器或虚拟机组成，节点有 `master`、`worker` 两种类型。一个集群中至少有一个 `master` 节点，在没有 `worker` 节点的情况下，`Pod` 也可以部署到 `master` 节点上。如果集群中的节点数量非常多，则可考虑扩展 `master` 节点，使用多个 `master` 节点控制集群。

在上一小节中，我们看到主节点中包含了比较多的组件，工作节点也包含了一些组件，这些组件可以分为两种，分别是 **Control Plane Components**(控制平面组件)、**Node Components**(节点组件)。

Control Plane Components 用于对集群做出全局决策，部署在 `master` 节点上；

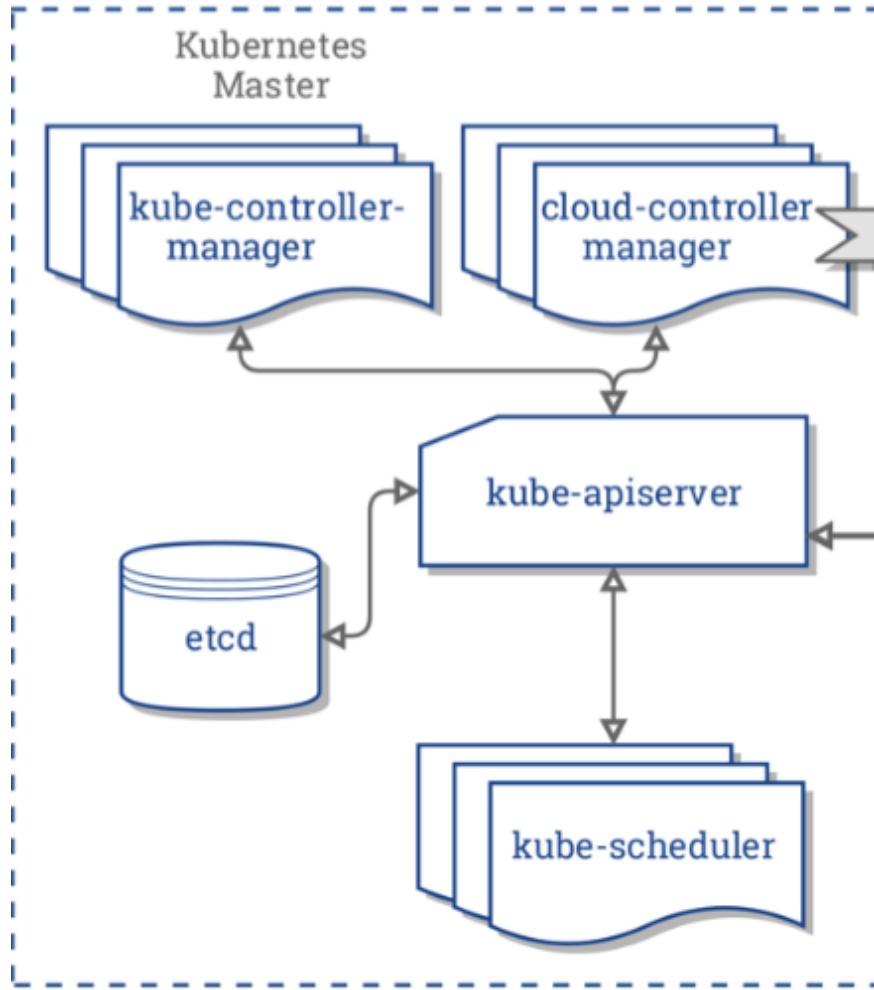
Node Components 在 `worker` 节点中运行，为 `Pod` 提供 `Kubernetes` 环境。

Master 节点

`Master` 是由一组称为控制平面组件组成的，如果你已经根据第二章中，通过 `minikube` 或 `kubeadm` 部署了 `Kubernetes`，那么我们可以打开 `/etc/kubernetes/manifests/` 目录，这里存放了 `k8s` 默认的控制平面组件的 `YAML` 文件。

```
.  
├── etcd.yaml  
├── kube-apiserver.yaml  
├── kube-controller-manager.yaml  
└── kube-scheduler.yaml
```

对于集群来说，这四个组件都是必不可少的。



在结构图中，还有一个 `cloud-controller` 组件，主要由云平台服务商提供，属于第三方组件，这里不再讨论。下面我们来了解 `master` 中的组件。

`master` 节点中各个组件(控制平面组件)需要使用到的端口：

协议	方向	端口范围	作用	使用者
TCP	入站	6443	Kubernetes API 服务器	所有组件
TCP	入站	2379-2380	etcd 服务器客户端 API	<code>kube-apiserver</code> , <code>etcd</code>
TCP	入站	10250	Kubelet API	<code>kubelet</code> 自身、控制平面组件
TCP	入站	10251	<code>kube-scheduler</code>	<code>kube-scheduler</code> 自身
TCP	入站	10252	<code>kube-controller-manager</code>	<code>kube-controller-manager</code> 自身

普通节点中各个组件需要使用到的端口：

协议	方向	端口范围	作用	使用者
TCP	入站	10250	Kubelet API	kubelet 自身、控制平面组件
TCP	入站	30000-32767	NodePort 服务†	所有组件

kube-apiserver

`kube-apiserver` 是 k8s 主要进程之一，`apiserver` 组件公开了 `Kubernetes API` (`HTTP API`)，`apiserver` 是 `Kubernetes` 控制面的前端，我们可以用 `Go`、`C#` 等编程语言写代码，远程调用 `Kubernetes`，控制集群的运行。`apiserver` 暴露的 endpoint 端口是 `6443`。

为了控制集群的运行，`Kubernetes` 官方提供了一个名为 `kubectl` 的二进制命令行工具，正是 `apiserver` 提供了接口服务，`kubectl` 解析用户输入的指令后，向 `apiserver` 发起 `HTTP` 请求，再将结果反馈给用户。

[Info] kubectl

`kubectl` 是 `Kubernetes` 自带的一个非常强大的控制集群的工具，通过命令行操作去管理整个集群。

`Kubernetes` 有很多可视化面板，例如 `Dashboard`，其背后也是调用 `apiserver` 的 `API`，相当于前端调后端。

总之，我们使用的各种管理集群的工具，其后端都是 `apiserver`，通过 `apiserver`，我们还可以定制各种各样的管理集群的工具，例如网格管理工具 `istio`。腾讯云、阿里云等云平台都提供了在线的 `kubernetes` 服务，还有控制台可视化操作，也是利用了 `apiserver`。

etcd

`etcd` 是兼具一致性和高可用性的键值数据库，作为保存 `Kubernetes` 所有集群数据的后台数据库。`apiserver` 的所有操作结果都会存储到 `etcd` 数据库中，`etcd` 主要存储 k8s 的状态、网络配置以及其它持久化数据，`etcd` 是使用 `B+` 树实现的，`etcd` 是非常重要的组件，需要及时备份数据。

kube-scheduler

`scheduler` 负责监视新创建的 `pod`，并把 `pod` 分配到节点上。当要运行容器时，发送的请求会被调度器转发到 `API`；调度器还可以寻找一个合适的节点运行这个容器。

kube-controller-manager

`kube-controller-manager` 中包含了多个控制器，它们都被编译到一个二进制文件中，但是启动后会产生不同的进程。这些控制器有：

- 节点控制器（`Node Controller`）

负责在节点出现故障时进行通知和响应

- 任务控制器 (Job controller)

监测代表一次性任务的 Job 对象，然后创建 Pods 来运行这些任务直至完成

- 端点控制器 (Endpoints Controller)

填充端点(Endpoints)对象(即加入 Service 与 Pod)

- 服务帐户和令牌控制器 (Service Account & Token Controllers)

为新的命名空间创建默认帐户和 API 访问令牌

控制器控制的 Pod、Job、Endpoints、Service 等，都是后面要深入学习的。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-10-31 18:42:23

- 第二章：部署和配置
 - 学习目标
 - 如何创建练习环境

第二章：部署和配置

本章内容主要是介绍、实践部署以及配置 **Kubernetes** 集群，如果读者的服务器环境处于国内，可能会因为网络原因无法部署 **Kubernetes**，建议读者使用国产的容器平台管理工具 **kubesphere**。

但是还是建议读者使用本章中的 **minikube** 和 **kubeadm** 教程部署 **kubernetes**，因为教程中会讲解一些 **kubernetes** 的知识，我们要学习它，就不应该绕过这个部署过程。如果是新手上路，部署失败，则建议使用 **kubesphere** 一键部署，等学习过 **kubernetes** 后，有空再尝试手动部署 **kubernetes**。

学习目标

- 下载安装和配置工具

通过多种方式下载安装工具集，了解每种工具的功能。
- 安装一个 **Kubernetes** 主节点并扩展一个集群

搭建 **Kubernetes Master** 节点，并加入 **Worker** 节点。
- 解决网络问题

解决国内无法拉取 **Kubernetes** 镜像问题。
- 部署和配置

学会部署以及配置启动集群，学会清除集群环境。

对于 **Kubernetes CKAD** 认证来说，在部署的时候需要考虑以下几个问题：

- 配置安全通信的网络方案

例如 **Claio**。
- 讨论高可用性部署注意事项

本章只讨论部署相关的知识，关于网络和其它知识，在其它章节中可以学习到。

如何创建练习环境

可以使用 **minikube** 做单机学习环境，或者购买云服务器、白嫖 3 个月 **Google Cloud**，也可以使用电脑创建多个虚拟机或者多台电脑来搭建节点，或者使用线上学习环境。

对于每个节点或虚拟机，尽量满足最低每台 2 核 2GB 的配置。

在 <https://katacoda.com/> 网站，有很多教程以及能够免费学习，并且可以使用线上的服务器环境。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-11-07 08:16:18

- 2.1 使用 Minikube 部署
 - Minikube
 - 部署
 - 查看集群状态
 - 创建资源
 - 创建 Deployment
 - 创建 Service
 - 清理集群资源

2.1 使用 Minikube 部署

Minikube 是一个创建单机 Kubernetes 集群的工具，它可以在一台服务器上快速创建一个学习环境，单机集群也可以学习到大多数入门的 Kubernetes 知识以及上手练习。CKAD 认证并不要求掌握 Minikube，不过我们可以初步学习练习，后面再使用 kubeadm 部署多节点集群。

本篇内容较为简单，可供读者练手使用，除了会使用 minikube 部署 kubernetes 外，也会部署应用，对于本章的内容，读者可快速练习一遍，后面再详细介绍各方面的知识。

Minikube

Minikube 是一个二进制工具，项目源码或文档等内容可以在 <https://github.com/kubernetes/minikube> 中找到，已经编译好的二进制可执行文件，可以在仓库的 Release 中下载，Minikube 支持 Windows、Linux、MacOS。

直接下载 minikube 最新版本的二进制文件。

可以通过官方 Github 仓库下载，也可以使用国内代理下载。

下载地址(Linux版本):

- <https://kubernetes.oss-cn-hangzhou.aliyuncs.com/minikube/releases/v1.20.0/minikube-linux-amd64>
- <https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64>

注：如果要下载 Win 版本，把 `minikube-linux-amd64` 改成 `minikube-windows-amd64.exe` 即可；如果是 MacOS 则是 `minikube-darwin-amd64`。另外要注意下载的版本号。

阿里云源下载的二进制工具，本身可以使用国内镜像，不需要代理，可以到仓库了解 <https://github.com/AliyunContainerService/minikube>。

你也可以看官方文档，按照文档安装 <https://minikube.sigs.k8s.io/docs/start/>。

```
curl -Lo minikube {下载地址}
```

```
chmod +x minikube
```

```
sudo mv minikube /usr/local/bin
```

部署

直接执行 `minikube start` 命令即可进行部署，但是国内会被墙，可能拉取不了镜像，需要设置代理，可参考 [2.4 章设置镜像代理](#)。

```
# 国外服务器  
minikube start  
# 使用阿里云版本时，指定国内源  
minikube start --image-mirror-country=cn  
# 使用阿里云版本时，指定镜像源  
minikube start --image-mirror=registry.cn-hangzhou.aliyuncs.com/google_containers
```

注：虚拟机安装或其他方式，需要配置安装驱动，请参考官方文档。还是无法拉取镜像的话，打开文档看看

https://minikube.sigs.k8s.io/docs/handbook/vpn_and_proxy/，配置代理试试。

接下来 `minikube` 会拉取各种镜像，需要一些时间。

```
* Pulling base image ...  
* Downloading Kubernetes v1.20.2 preload ...  
  > preloaded-images-k8s-v10-v1...: 491.71 MiB / 491.71 MiB 100.00% 60.04 Mi  
  > gcr.io/k8s-minikube/kicbase...: 357.67 MiB / 357.67 MiB 100.00% 7.41 MiB  
* Creating docker container (CPUs=2, Memory=4000MB) .../
```

通过 `minikube version` 命令可以查看 `minikube` 的版本，接下来我们使用 `minikube start` 命令，可以直接创建一个 `kubernetes` 集群。

问题一

如果启动不起来提示没有 `docker` 用户，这是因为默认不应该使用 `root` 用户执行命令和启动程序，可以创建一个 `docker` 用户，也可以使用 `--driver=none` 指定不用 `docker` 用户。

如果要用 `docker` 用户：

```
groupdel docker  
useradd -m docker  
passwd docker  
# 修改密码后，加入用户组  
gpasswd -a docker docker
```

打开 `/etc/sudoers` 文件，在 `root ALL=(ALL:ALL) ALL` 下增加新的一行：

```
docker ALL=(ALL)ALL
```

然后切换为 `docker` 用户：`su docker`。

如果不使用 `docker` 用户，只需要在初始化集群时加上 `--driver=none`。

```
minikube start --driver=none
```

问题二

PS: 如果报 `X Exiting due to GUEST_MISSING_CONNTRACK: Sorry, Kubernetes 1.20.2 requires conntrack to be installed in root's path`, 则需要安装 `conntrack`, `apt install conntrack`。

如果没有问题, 会自动进行安装。

```
* The requested memory allocation of 1986MB does not leave room for system overhead (total system meminfo: 1986MB)
* Suggestion: Start minikube with less memory allocated: 'minikube start --memory=1986mb'

* Starting control plane node minikube in cluster minikube
* Running on localhost (CPUs=2, Memory=1986MB, Disk=9749MB) ...
* OS release is Ubuntu 18.04.5 LTS
* Preparing Kubernetes v1.20.2 on Docker 20.10.2 ...
  - kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
    > kubeadm.sha256: 64 B / 64 B [-----] 100.00% ? p/s 0s
    > kubectl.sha256: 64 B / 64 B [-----] 100.00% ? p/s 0s
    > kubelet.sha256: 64 B / 64 B [-----] 100.00% ? p/s 0s
    > kubeadm: 37.40 MiB / 37.40 MiB [-----] 100.00% 15.16 MiB p/s 2.7s
    > kubectl: 38.37 MiB / 38.37 MiB [-----] 100.00% 13.79 MiB p/s 3.0s
    > kubelet: 108.73 MiB / 108.73 MiB [-----] 100.00% 20.40 MiB p/s 5.5s
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
* Configuring local host environment ...
*
```

minikube 完成初始化后, 打开新的终端窗口, 执行 `minikube dashboard` 启动面板, 根据 URL 地址, 可以访问面板。

正常的话, 执行 `docker ps` 后是这样的。

```
root@minikube-1:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
1320b23eb082        gcr.io/k8s-minikube/storage-provisioner   "/storage-provisioner"   4 seconds ago       Up 3 seconds          k8s_stora...
ge-provisioner        storage-provisioner_kube-system_80cc6f1f-50b7-41ba-a6f0-4311e5f52c1_0
0e65ee1223b6        bfc3a36ebd25   "/coredns -conf /etc...
minikube             k8s.gcr.io/minikube-system_7d69e62c-d9f4-48e7-aadb-1b05a5f5f...
061c16162298        k8s.gcr.io/pause:3.2      "/pause"
oredns-747f55c5b-4w85  kube-system_7d69e62c-d9f4-48e7-aedb-b4ba2575a5f5...
2a651ee5d47e        43154dd057a8   "/usr/local/bin/kube...
proxy_kube-proxy     pd72_kube-system_42cc5f78-843b-4714-b748-e79e0439876_0
03d015151515        k8s.gcr.io/pause:3.2      "/pause"
storage-provisioner  kube-system_86ce6f1f-50b7-41ba-a6f0-4311e5f52c1_0
ab170fcf2707        k8s.gcr.io/pause:3.2      "/pause"
ube-proxy-pdrp2      kube-system_42cc5f78-843b-4714-b748-e79e0439876_0
f5303e1e0e0e         gcr.io/k8s-dns/dns-sidecar-apiserver-ad...
apiserver_kube-apiserver-1_kube-system_c806bc5e4f845557078b0f5af9bb0brc0_0
8796d89c2183        a2716429d998   "/kube-controller-main..."
controller-manager_kube-controller-manager-instance-1_kube-system_578c22dbe6410e4bd36cf14b0fb0bd...
83d41ed01e0e         099e74483031   "/etc -advertis...
etcd_kube-etcd-1_kube-system_e5951278547105a0a1e0d59579fc0_0
107b060deebd        ed2c44ffdd78   "/kube-scheduler -ou...
scheduler_kube-scheduler-instance-1_kube-system_6b4a0eeb3d15a1c12e47c15d32e6eb0d_0
3277ff701116        k8s.gcr.io/pause:3.2      "/pause"
controller-manager_kube-controller-manager-instance-1_kube-system_57b0c22db6410e4bd36cf14b0fb0bd...
191fa4b6ea09        k8s.gcr.io/pause:3.2      "/pause"
ube-apiserver_kube-apiserver-1_kube-system_c806bc5e4f845557078b0f5af9bb0brc0_0
61d108433acf        k8s.gcr.io/pause:3.2      "/pause"
tcd_instance-1_kube-system_0905b121a47165a0a1f1e0d59579fc0_0
ube-scheduler_kube-scheduler-1_kube-system_e5951278547105a0a1e0d59579fc0_0
ube-scheduler-instance-1_kube-system_eb4a0eeb3d15a1c2e47c15d32e6eb0d_0
root@instance-1:~#
```

查看集群状态

本身 minikube 带有一些简单的 `kubectl` 命令, 可以查看集群状态信息。

获取集群所有节点(机器):

```
minikube kubectl get nodes
```

获取集群所有命名空间:

```
minikube kubectl get namespaces
```

查看集群所有 Pod:

```
minikube kubectl -- get pods -A
```

```

root@instance-1:~# minikube kubectl get nodes
NAME      STATUS   ROLES      AGE      VERSION
instance-1 Ready    control-plane,master  5m23s   v1.20.2
root@instance-1:~# minikube kubectl -- get pods -A
NAMESPACE     NAME          READY   STATUS    RESTARTS   AGE
kube-system   coredns-74ff55c5b-5zf9h   1/1    Running   0          5m39s
kube-system   etcd-instance-1        1/1    Running   0          5m39s
kube-system   kube-apiserver-instance-1  1/1    Running   0          5m39s
kube-system   kube-controller-manager-instance-1  1/1    Running   0          5m39s
kube-system   kube-proxy-9srhg       1/1    Running   0          5m39s
kube-system   kube-scheduler-instance-1  1/1    Running   0          5m39s
kube-system   storage-provisioner    1/1    Running   0          5m51s
root@instance-1:~# minikube kubectl get namespaces
NAME      STATUS   AGE
default  Active   6m5s
kube-node-lease  Active   6m7s
kube-public  Active   6m7s
kube-system  Active   6m7s

```

创建资源

由于 `minikube` 不会自动下载 `kubectl`、`kubelet` 等工具，我们需要手动安装，你可以参考 2.2 章的安装方法，关于 `kubectl`、`kubelet`，后面的章节会详细介绍。最简单的安装方法：

```

# 仅供 ubuntu 参考
snap install kubectl --classic
snap install kubelet --classic

```

本节的内容供简单练习，不会详细说明命令的作用和相关知识，待读者阅读后面的章节时，可以学习到更多内容。

创建 Deployment

`Deployment` 可以部署应用并管理实例数量，它提供了一种故障的自我修复机制，当应用挂了后，`Deployment` 可以自动启动一个新的实例，维护固定数量的 Pod。

`kubectl create deployment` 命令创建管理 Pod 的 Deployment。

```

# 格式 kubectl create deployment {deployment名称} {参数}
kubectl create deployment hello --image=nginx:latest

```

查看 Deployment：

```

kubectl get deployments

```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello	1/1	1	1	6m39s

查看 Pod：

```

kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
hello-b8d95ff4c-x67zr	1/1	Running	0	7m8s

查看集群事件：

```
kubectl get events
```

查看 `kubectl` 配置:

```
kubectl config view
```

创建 Service

Service 为 Pod 提供了一种外网访问能力，默认情况下，Pod 只能在 Kubernetes 集群的同一节点访问，如果要外部网络访问，则需要为 Pod 暴露一个 Kubernetes Service，Service 为 Pod 提供了外网访问能力。这里我们把上一小节的 hello 暴露出去。

nginx 镜像会暴露一个 80 端口，通过 80 端口我们可以访问到 nginx 服务。但是在 Kubernetes 中，则可能有些麻烦。

每个 Pod 在集群中都有一个唯一 IP，我们可以查看详细的 Pod 信息：

```
kubectl get pods -o wide
```

```
root@instance-1:~# kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE      NOMINATED NODE   READINESS GATES
hello-b8d95ff4c-x67zr   1/1     Running   0          15m    172.17.0.3   instance-1   <none>        <none>
```

我们可以直接访问它：

```
curl 172.17.0.3
```

```
root@instance-1:~# curl 172.17.0.3
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

为了能够在外网访问，我们创建 Service:

```
kubectl expose deployment hello --type=LoadBalancer --port=80
```

然后查看刚刚创建的 service:

```
kubectl get service hello  
# 或  
minikube service hello
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hello	LoadBalancer	10.102.73.188	<pending>	80:31286/TCP	5s

NAMESPACE	NAME	TARGET PORT	URL
default	hello	80	http://10.170.0.5:31286

* Opening service default/hello in default browser...
<http://10.170.0.5:31286>

此时，在集群内，通过 `http://10.170.0.5:31286` 可以访问此 Pod，或者在外网访问 31286 端口。

清理集群资源

由于 Minikube 创建的资源只是单机的，同时会产生很多 Docker 容器，我们练习完毕后，就要清除环境，以免影响后续实践环境。

首先清除 service、deployment (可以跳过这个步骤)。

```
kubectl delete service hello  
kubectl delete deployment hello
```

然后停止 Minikube 虚拟机 (VM) :

```
minikube stop
```

接着删除 Minikube 虚拟机 (VM) :

```
minikube delete
```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-10-31 07:49:26

- 2.2 使用 `kubeadm` 部署
 - 命令行工具
 - 安装命令行工具
 - 通过软件仓库安装
 - 不同操作系统
 - 集群管理
 - 创建 `kubernetes` 集群
 - 可能碰到的问题
 - 删减节点
 - 清除环境

2.2 使用 `kubeadm` 部署

上一章中，我们用 `minikube` 去搭建单机集群，并且创建 `Deployment`、`Service`(在三章中讲解)，本篇将介绍利用 `kubeadm` 部署多节点集群，并学会安装以及使用 `kubernetes` 的命令行工具，快速创建集群实例，完成部署 `hello world` 应用的实践。

`Kubeadm` 是 CKAD 认证中要求掌握的部署方式，但是镜像需要国外网络才能下载，读者如果是国内服务器，可以参考 2.4 章的内容，使用国内服务器进行代理。

本章内容主要介绍如何安装 `kubeadm` 以及部署集群、添加节点。

需要提前在服务器安装好 `Docker`。

命令行工具

在 `kubernetes` 中，主要有三个日常使用的工具，这些工具使用 `kube` 前缀命名，这三个工具如下：

- `kubeadm`：用来初始化集群的指令，能够创建集群已经添加新的节点。可用其它部署工具替代。
- `kubelet`：在集群中的每个节点上用来启动 `Pod` 和容器等，每个节点必须有，相对于节点与集群的网络代理。
- `kubectl`：用来与集群通信/交互的命令行工具，与 `kubernetes API-Server` 通讯，是我们操作集群的客户端。

在 1.5 章中介绍过 `kubelet`、`kubectl`，`kubelet` 负责集群中节点间的通讯，`kubectl` 供用户输入命令控制集群，而且 `kubeadm` 则是创建集群、添加减少节点的工具。

安装命令行工具

命令行工具是每个节点都需要安装的，`kubectl`、`kubelet` 两个是必需的组件，而 `kubeadm` 则可以被代替。`Kubeadm` 是 `Kubernetes` 官方推荐的部署工具，但由于网络等各方面原因，中文社区中也开发了一些替代项目，例如 `Kubesphere`(<https://kubesphere.com.cn/>)，可在内部署 `Kubernetes`，省去网络问题。

通过软件仓库安装

下面介绍如何通过 Google 的源下载安装工具包。

更新 `apt` 包索引并安装使用 Kubernetes `apt` 仓库所需要的包:

```
sudo apt-get update  
sudo apt-get install -y apt-transport-https ca-certificates curl
```

下载 Google Cloud 公开签名秘钥:

```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.c
```

添加 Kubernetes `apt` 仓库:

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.k
```

注: 如果是国内服务器, 请忽略以上两步, 使用以下命令解决:

```
apt-get update && apt-get install -y apt-transport-https  
curl https://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg | apt-key add -  
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list  
deb https://mirrors.aliyun.com/kubernetes/apt/ kubernetes-xenial main  
EOF
```

更新 `apt` 包索引, 安装 `kubelet`、`kubeadm` 和 `kubectl`, 并锁定其版本:

```
sudo apt-get update  
sudo apt-get install -y kubelet kubeadm kubectl  
sudo apt-mark hold kubelet kubeadm kubectl
```

执行命令检查是否正常:

```
kubeadm --help
```

不同操作系统

只是这里介绍一下 `ubuntu` 和 `centos` 不同的安装方法, 已经通过前面的安装方法安装好, 则不需要理会这一小节。

`Ubuntu` 和 `Debain` 等系统可以使用以下命令通过软件仓库安装:

```
sudo apt-get update && sudo apt-get install -y apt-transport-https gnupg2 curl  
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sc  
sudo apt-get update  
sudo apt-get install -y kubelet kubeadm kubectl
```

`Centos`、`RHEL` 等系统可以使用以下命令通过软件仓库安装:

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/yum-key.pub
EOF
yum install -y kubelet kubeadm kubectl
```

集群管理

创建 **kubernetes** 集群

Kubeadm 是一个创建管理工具，主要提供了 `kubeadm init` 和 `kubeadm join` 两个命令，作为创建 **Kubernetes** 集群的“快捷途径”的最佳实践。

Kubernetes 集群由 **Master** 和 **Worker** 两种节点组成，**Master** 节点负责控制集群所有的节点。

注意，本教程集群中的节点应当都是内网可互通的服务器，这些服务器之间可以通过内网相互访问。如果是服务器之间通过公网相互通讯的，操作方法请查询其它教程。

1. 创建 **Master**

执行 `hostname -i` 查看此 node 的 ip。

我们初始化一个 **API Server** 服务，绑定地址为 **192.168.0.8**（按照你的ip改）。此步骤创建了一个 **master** 节点。

注：可以直接使用 `kubeadm init`，它会自动使用默认网络ip。

```
kubeadm init
# 或 kubeadm init --apiserver-advertise-address 192.168.0.8
# 或 kubeadm init --apiserver-advertise-address $(hostname -i)
```

部署失败，可以参考下面两个命令，查看失败原因。

```
systemctl status kubelet
journalctl -xeu kubelet
```

常见与 Docker 有关的错误可参考：

<https://kubernetes.io/docs/setup/production-environment/container-runtimes/#docker>

完成后，会提示一些信息，在提示的内容中找到：

```
kubeadm join 192.168.0.8:6443 --token q25z3f.v5uo5bphvgxkjnmz \
--discovery-token-ca-cert-hash sha256:0496adc212112b5485d0ff12796f66b29237d066fbc1
```

保存这段信息下来备用，后面加入节点时需要使用到。

如果有提示 `Alternatively, if you are the root user, you can run:` 则你还需要执行下面的命令。

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

[Info] 提示

`admin.conf` 是连接 **Kubernetes** 的认证文件，通过此文件才能连接到 **kubernetes**，`kubectl` 也需要这个文件；在 Linux 中，使用 `KUBECONFIG` 环境变量知道认证文件的所在。

Linux 中每个用户的环境变量是不同的，如果切换了用户，则也需要设置 `KUBECONFIG` 环境变量；如果要在别的节点上连接集群，则可以把这个文件复制过去。

后面的操作都需要 `admin.conf` 文件，否则会报 `The connection to the server localhost:8080 was refused - did you specify the right host or port?`。

由于 `export` 的环境变量不能持久化，请打开 `~/.bashrc` 文件，把这个命令加到文件最后面。

[Info] 提示

为了保护 `/etc/kubernetes/admin.conf`，避免直接指向，建议每个用户复制一次此文件到用户目录下，其命令如下：

```
mkdir -p $HOME/.kube
cp -f /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

2. 初始化网络

这一步不是必需的，不过一般来说，部署 **Kubernetes** 会配置网络，否则会节点之间不能相互访问，读者可以跟着做一次，在后面的章节中我们在一探究竟。

通过远程配置文件初始化网络，需要从第三方拉取一个 `yaml` 文件。

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | # --namespace=kube-system 表示插件放到 kube-system 命名空间中运行
```

成功的话会提示：

```
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
```

我们也可以手动配置，执行 `kubectl version` 查看版本号，找到 `GitVersion:v1.21.1`，替换 `yaml` 文件的地址 `https://cloud.weave.works/k8s/net?k8s-version=v1.21.1`，然后执行 `kubectl apply -n kube-system -f net.yaml` 即可。

3，加入集群

前面已经创建了 **Master** 节点，接下来将另一个服务器以 **Worker** 节点的方式加入集群中。如果读者只有一台服务器，则可以跳过这个步骤。

当节点加入 `kubeadm` 初始化的集群时，双方需要建立双向信任，分为发现(**Worker**信任**Master**)和 TLS 引导(**Master**信任待加入**Worker**)两部分。目前有两种加入方式，一种是通过令牌加入，一种是通过 `kubeconfig` 文件加入。

格式：

```
kubeadm join --discovery-token abcdef.xxx {IP}:6443 --discovery-token-ca-cert-hash sha256:0496adc212112b5485d0ff12796f66b29237d066fb...
```

在第二个节点中，使用之前**备份好的命令**，直接执行，加入集群，**格式如下命令所示**。

```
kubeadm join 192.168.0.8:6443 --token q25z3f.v5uo5bphvgxkjnmz \
--discovery-token-ca-cert-hash sha256:0496adc212112b5485d0ff12796f66b29237d066fb...
```

复制粘贴时，要注意，可能会由于 \ 换行符，导致粘贴时，多了一个小数点，导致报错。

正确的：
`kubeadm join 10.170.0.5:6443 --token 5x5tyd.rj3souyarsdeu4d7 \
--discovery-token-ca-cert-hash sha256:812bbe22f6625cc58cbb96d4da8b4a88795c0bbc070fbe9b0a1205c72a97d929`
粘贴出错：
`kubeadm join 10.170.0.5:6443 --token 5x5tyd.rj3souyarsdeu4d7 \
> --discovery-token-ca-cert-hash sha256:812bbe22f6625cc58cbb96d4da8b4a88795c0bbc070fbe9b0a1205c72a97d929`


可能碰到的问题

查看 `docker` 版本： `yum list installed | grep docker` 和 `docker version`。

如果部署过程中出现 `failed to parse kernel config: unable to load kernel module`，也说明了 `docker` 版本太高，需要降级。

如果服务器装了 `dnf`，那么降级 `docker` 版本的命令：

```
dnf remove docker \
    docker-client \
    docker-client-latest \
    docker-common \
    docker-latest \
    docker-latest-logrotate \
    docker-logrotate \
    docker-selinux \
    docker-engine-selinux \
    docker-engine
```

```
dnf -y install dnf-plugins-core
```

```
dnf install docker-ce-18.06.3.ce-3.el7 docker-ce-cli containerd.io
```

不行的话就按照 <https://docs.docker.com/engine/install/centos/> 降级，或者自行按照其它方法处理。

注意，`docker version` 会看到 `client` 和 `server` 版本，两者的版本号可能不一致。

删除节点

在生产环境中，由于节点上已经部署着服务，因此直接删除节点，可能会导致严重的故障问题。因此需要移除一个节点时，首先要在此节点上驱逐所有 `Pods`，`Kubernetes` 会自动将此节点上的 `Pod` 转移到其它节点上部署(第三章会讲)。

获取集群中的所有节点，找到需要驱逐的节点名称。

```
kubectl get nodes
```

驱逐此节点上所有的 `Pod`:

```
kubectl drain {node名称}
```

虽然驱逐了节点上所有的服务，但是节点依然在集群中，只是 `Kubernetes` 不会再部署 `Pod` 到此节点上。如果需要恢复此节点，允许继续部署 `Pod`，可使用:

```
kubectl uncordon {节点名称}
```

关于驱逐，后面的章节会学习到。

注：驱逐 `Pod`，并一定能够驱逐所有 `Pod`，有些 `Pod` 可能不会被清除。

最终删除此节点:

```
kubectl delete node {节点名称}
```

集群删除了此节点后，节点上还保留着一些数据，可以继续清除环境。

清除环境

如果步骤做错了想重来，或者移除节点需要清除环境，可以执行 `kubeadm reset [flags]` 命令。

注：只执行 `kubeadm reset` 命令无效。

`[flags]` 有四种类型：

<code>preflight</code>	<code>Run reset pre-flight checks</code>
<code>update-cluster-status</code>	<code>Remove this node from the ClusterStatus object.</code>
<code>remove-etcd-member</code>	<code>Remove a <code>local</code> etcd member.</code>
<code>cleanup-node</code>	<code>Run cleanup node.</code>

我们需要执行：

```
kubeadm reset cleanup-node  
kubeadm reset
```

即可在当前服务器上清除 **Kubernetes** 残留的 容器或者其它数据。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-11-03 21:33:53

- 2.3 CKAD认证中的部署教程
 - 部署
 - 预设网络
 - kubeadm 安装 k8s
 - 配置 Calico
 - 其它
 - 在节点上执行命令
 - 自动补全工具
 - 状态描述

2.3 CKAD认证中的部署教程

在上一章中，我们已经学会了使用 `kubeadm` 创建集群和加入新的节点，在本章中，将按照 CKAD 课程的方法重新部署一遍，实际上官方教程的内容不多，笔者写了两篇类似的部署方式，如果已经部署了 `kubernetes` 集群，则本章的内容可跳过。

部署

预设网络

本节主要是配置 `hosts` 文件，在后续配置中，通过主机名称即可快速连接，而不需要每次都打上 IP 地址。

我们在 Master 节点服务器执行 `ip addr` 命令，找到 `ens4`，把里面提到的 ip 记录下来。

```
ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq state UP group default qlen 1
      link/ether 42:01:0a:aa:00:02 brd ff:ff:ff:ff:ff:ff
      inet 10.170.0.2/32 scope global dynamic ens4
          valid_lft 2645sec preferred_lft 2645sec
      inet6 fe80::4001:aff:fea:2/64 scope link
          valid_lft forever preferred_lft forever
```

如上述 ip 是 10.170.0.2。或者使用 `hostname -i` 查询。方式有很多，目前是获得主机的内网 IP。

然后修改 `/etc/hosts` 文件，加上一行（替换这个ip为你的）：

```
10.170.0.2      k8smaster
```

后面我们访问集群，使用 `k8smaster` 这个主机名称(域名)，而且不是需要 IP 地址，使用主机名称方便记忆，也避免了 IP 强固定。

kubeadm 安装 k8s

这里的部署过程跟上一章中的有所差异，因为上章中，直接使用 `kubeadm init` 进行初始化集群，没有配置更多细节。

执行 `kubectl version` 查看 k8s 版本，找到这段 `GitVersion:"v1.21.0"`，即为 Kubernetes 版本。

创建一个 `kubeadm-config.yaml` 文件，我们使用 `kubeadm init` 时，通过此配置文件出初始化 k8s master。

文件内容为：

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
kubernetesVersion: 1.21.0
controlPlaneEndpoint: "k8smaster:6443"
networking:
  podSubnet: 192.168.0.0/16
```

注意，`:` 后面必须带一个空格。表示 `key: value`。例如 `image: nginx:latest`，不带空格的 `:` 会连在一起。

然后通过配置文件初始化 Master：

```
kubeadm init --config=kubeadm-config.yaml --upload-certs --v=5 | tee kubeadm-init.out
# 可省略为 kubeadm init --config=kubeadm-config.yaml --upload-certs
```

`--v=5` 可以输出更多信息信息，`tee xxx` 可以让信息输出到一个文件中，方便收集日志或者后续检查。

执行初始化命令后，终端或查看 `kubeadm-init.out` 文件，有以下内容：

```
To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of the control-plane node running the following command or

kubeadm join k8smaster:6443 --token 45td1j.xqdscm4k06a4edi2 \
--discovery-token-ca-cert-hash sha256:aeb772c57a35a283716b65d16744a71250bcc25d624e
--control-plane --certificate-key d76287ccc4701db9d34e0c9302fa285be2e9241fc43c9421

Please note that the certificate-key gives access to cluster sensitive data, keep it safe.
As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use
"kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join k8smaster:6443 --token 45td1j.xqdscm4k06a4edi2 \
--discovery-token-ca-cert-hash sha256:aeb772c57a35a283716b65d16744a71250bcc25d624e
```

按照提示，我们逐个执行下面的命令，不要一次性粘贴执行，因为 `cp -i` 表示要你输入 `y/n` 确认更改，一次性粘贴会导致跳过(把 `-i` 改为 `-f` 也行)。

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

然后：

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

笔者注：`KUBECONFIG` 环境变量在下次登录或新建终端窗口会失效，打开用户目录的 `.bashrc` 文件，在最后面加上 `export KUBECONFIG=/etc/kubernetes/admin.conf`，可保证下次登录或切换终端，依然可用。

笔者注：因为涉及到多用户，所以如果切换用户，就不能使用

`kubeadm/kubectl/kubelet` 命令了，如果读者切换了用户，则可以执行上面 `make -p $HOME/.kube` 到 `export xxx` 这两部分的命令，这样别的用户也可以执行命令操作节点。

输入 `kubeadm config print init-default` 可以查看到 master 初始化时配置。

以上便是 CKAD 官方的部署方法。

配置 Calico

什么是 CNI

CNI 意为容器网络接口，是 Kubernetes 的一种标准设计，使用者可以不需要关注使用了何种网络插件，可以在插件或销毁容器时更加容易地配置网络。

Kubernetes 中有 Flannel、Calico、Weave 等主流的插件，在上一篇中，我们部署 Kubernetes 网络时，使用了 Weave，而在本章中，我们将使用 Calico 来部署网络。

对于 CNI，后面的章节会深入学习。

Calico(<https://github.com/projectcalico/calico>) 是针对容器、虚拟机和裸机工作负载的开源网络和安全解决方案，它提供了 Pod 之间的网络连接和网络安全策略实施。

Flannel、Calico、Weave 都是常用的 Kubernetes 网络插件，读者可参考 <https://kubernetes.io/zh/docs/concepts/cluster-administration/networking/> 这里不做过多的说明。

首先下载 Calico 的 yaml 文件。

```
wget https://docs.projectcalico.org/manifests/calico.yaml
```

然后我们需要留意 yaml 文件中的 `CALICO_IPV4POOL_CIDR` 的值，读者直接打开 <https://docs.projectcalico.org/manifests/calico.yaml> 或者使用 `less calico.yaml` 在终端上阅读文件。

找到 `CALICO_IPV4POOL_CIDR` 例如：

```
# - name: CALICO_IPV4POOL_CIDR
#   value: "192.168.0.0/16"
```

这个表示 ip4 池，如果 ip 不存在，则会自动创建，创建的 pod 的网络 ip 会在这个范围。默认是 `192.168.0.0` 我们不需要改，如果你需要定制，则可以删除 `#`，然后改动 ip。

[Error] 提示

请务必根据你集群中的 IP 段，配置此参数。

然后我们启用 Calico 网络插件：

```
kubectl apply -f calico.yaml
```

当网络配置完成后，即可使用 `kubeadm join` 加入节点。

其它

在节点上执行命令

如果我们在 Worker 节点上执行命令，会发现：

```
root@instance-2:~# kubectl describe nodes
The connection to the server localhost:8080 was refused - did you specify the right hc
```

首先在 Master 节点中，下载 `/etc/kubernetes/admin.conf` 文件，或者复制文件内容，到 Worker 节点中。

将文件上传或复制到 Worker 节点的 `/etc/kubernetes/admin.conf` 文件，执行配置即可。

```
mkdir -p $HOME/.kube
sudo cp -f /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

```
echo 'export KUBECONFIG=/etc/kubernetes/admin.conf' >> $HOME/.bashrc
```

自动补全工具

`kubectl` 命令和可选参数非常多，每次都要敲长长的命令，容易出错，我们可以利用 `bash-completion` 为我们快速完成命令的输入。

```
sudo apt-get install bash-completion -y
```

```
source <(kubectl completion bash)
echo "source <(kubectl completion bash)" >> $HOME/.bashrc
```

当我们敲命令时，按下 TAB 键，会自动补全。

输入 `kubectl des`，然后按一下 TAB 键，会发现内容自动补全为 `kubectl describe`。

状态描述

执行 `kubectl describe nodes` 命令，我们可以看到节点详细的信息，其中有个 `Conditions` 字段，描述了所有正在运行中(Running)的节点的状态，它有 5 个字段或类型：

- **Ready**
Node 是否能够接收 pod，如果可以则 `status` 为 True；如果节点不健康，不能接收 pod，则为 False。正常情况下为 True。
- **DiskPressure**
表示节点的空闲空间不足以用于添加新 Pod，如果为 True 则说明不正常。
- **MemoryPressure**
表示节点存在内存压力，即节点内存可用量低，如果为 True 则说明不正常。
- **PIDPressure**
表示节点存在进程压力，即节点上进程过多；如果为 True 则说明不正常。
- **NetworkUnavailable**
表示节点网络配置不正确；如果为 True，则说明不正常。

如果使用 JSON 表示：

```
"conditions": [
  {
    "type": "Ready",
    "status": "True",
    "reason": "KubeletReady",
    "message": "kubelet is posting ready status",
    "lastHeartbeatTime": "2019-06-05T18:38:35Z",
    "lastTransitionTime": "2019-06-05T11:41:27Z"
  }
]
```

读者可参考：<https://kubernetes.io/zh/docs/concepts/architecture/nodes/>

本章内容主要介绍了 CKAD 认证中要求掌握的 `kubeadm` 部署 k8s、配置启动 Calico 网络插件，跟上一篇的内容比较，主要是通过 yaml 文件去控制创建 kubernetes 集群，两章的部署过程一致，只是网络插件有所不同。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-03 21:26:26

- 2.4 国内代理
 - 配置
 - 使用

2.4 国内代理

在本章中，将会学习如果在国内服务器中拉取 Kubernetes 镜像，解决 kubeadm 网络问题。

首先按照 2.2 中的教程，安装好 `kubeadm`、`kubectl`、`kubelet` 三个工具。

配置

当使用 `kubeadm init` 命令时，如果不传递参数，则会使用默认的配置文件初始化集群，我们可以导出默认的配置文件内容。

```
kubeadm config print init-defaults > kubeadm.conf
```

找到大约在 31 行的 `imageRepository: k8s.gcr.io`，修改为镜像地址：

```
imageRepository: registry.aliyuncs.com/google_containers
```

通过配置文件 `kubeadm` 会在阿里云源拉取镜像。

```
kubeadm config images pull --config kubeadm.conf
```

但并不是所有 Kubernetes 镜像阿里云源都有，可能有些镜像会报错。

```
root@instance-r0rzc5pb:~# kubeadm config images pull --config kubeadm.conf
[config/images] Pulled registry.aliyuncs.com/google_containers/kube-apiserver:v1.21.0
[config/images] Pulled registry.aliyuncs.com/google_containers/kube-controller-manager:v1.21.0
[config/images] Pulled registry.aliyuncs.com/google_containers/kube-scheduler:v1.21.0
[config/images] Pulled registry.aliyuncs.com/google_containers/kube-proxy:v1.21.0
[config/images] Pulled registry.aliyuncs.com/google_containers/etcd:3.4.13-0
[config/images] Pulled registry.aliyuncs.com/google_containers/coredns:v1.8.0*
Failed to pull image "registry.aliyuncs.com/google_containers/coredns:v1.8.0": output: Error response from daemon: pull access denied for registry.aliyuncs.com/google_containers/coredns, repository does not exist or may require 'docker login': denied: requested access to the resource is denied
, error: exit status 1
To see the stack trace of this error execute with --v=5 or higher
```

```
root@instance-r0rzc5pb:~# kubeadm init --image-repository registry.aliyuncs.com/google_containers
[init] Using Kubernetes version: v1.21.1
[preflight] Running pre-flight checks
[preflight] Running image refresher, this may take a few seconds depending on the size of your internet connection
[preflight] This may take a minute or two, depending on the size of your internet connection
[preflight] You can also perform this action in beforehand using `kubeadm config images pull`
error execution phase preflight: [preflight] Some fatal errors occurred:
[ERROR ImagePull]: failed to pull image registry.aliyuncs.com/google_containers/coredns:v1.8.0: output: Error response from daemon: pull access denied for registry.aliyuncs.com/google_containers/coredns, repository does not exist or may require 'docker login': denied: requested access to the resource is denied
, error: exit status 1
[preflight] If you know what you are doing, you can make a check non-fatal with `--ignore-preflight-errors=...`
To see the stack trace of this error execute with --v=5 or higher
```

例如 `coredns` 镜像，阿里云源的 `google_containers` 仓库中不能直接拉取到，那么我们可以手动拉取再改 `tag`。

```
docker pull coredns/coredns:latest
```

```
docker tag coredns/coredns:latest registry.aliyuncs.com/google_containers/coredns/core
```

使用

当我们要在国内的服务器创建集群时，在 `kubeadm init` 命令后面指定代理源。

```
kubeadm init --image-repository registry.aliyuncs.com/google_containers
```

耐心等待，即可完成集群的初始化。

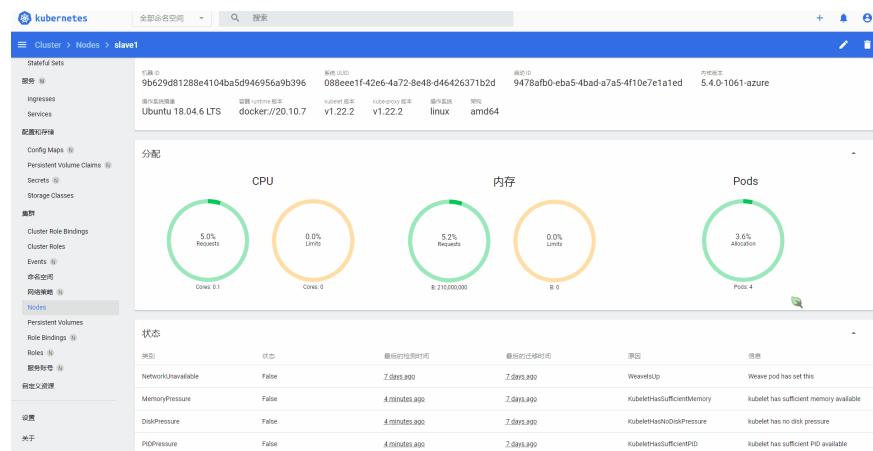
Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-10-31 10:56:41

- 2.5 Dashboard

2.5 Dashboard

Kubernetes-Dashboard 是一个管理 Kubernetes 集群的 Web UI，其界面大气优雅，跟 kubectl 一样，其后端是 API-Server，通过它可以查看集群中每个集群的节点状态，各类资源的调度状况，还可以动态伸缩节点、管理资源。

其界面如下：



dashboard 的 Github 仓库地址为：

<https://github.com/kubernetes/dashboard/tags>，请从中选择最新版本的 yaml。

使用在线的 YAML 文件部署 Kubernetes-Dashboard：

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.2.0/aio/deployment.yaml
```

请替换 url 中的 2.2.0。

dashboard 创建后会出现在 kubernetes-dashboard 命名空间中。

```
root@instance-1:~# kubectl get pods --namespace=kubernetes-dashboard
NAME                               READY   STATUS    RESTARTS   AGE
dashboard-metrics-scraper-856586f554-4nd9v   1/1    Running   0          9d
kubernetes-dashboard-78c79f97b4-288js        1/1    Running   0          9d

root@instance-1:~# kubectl get services --namespace=kubernetes-dashboard
NAME           TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)
dashboard-metrics-scraper   ClusterIP  10.98.50.123  <none>       8000/TCP
kubernetes-dashboard   NodePort    10.111.44.44   <none>       443/TCP
```

[Error] 提示

如果你的集群上有多个节点，那么可能会被放到某个节点上。

接着，查看 dashboard 被放到哪个节点上运行：

```
root@master:~# kubectl get pods --namespace=kubernetes-dashboard -o wide
NAME                               READY   STATUS    RESTARTS   AGE     IP
dashboard-metrics-scraper-c45b7869d-d9s2m   1/1     Running   0          15m   10.32.6
kubernetes-dashboard-576cb95f94-8dzcj       1/1     Running   0          15m   10.32.6
```

可以看到，笔者的 **dashboard** 服务被放到 **slave1** 节点上去了，接着请记录 **slave1** 的服务器 ip。

由于 **dashboard** 网络默认是 **ClusterIP** 方式，因此外网是不能访问的，所以为了能够被外界访问，可以修改其 **service**。

```
kubectl edit service kubernetes-dashboard --namespace=kubernetes-dashboard
```

将 **type: ClusterIP** 改成 **type: NodePort**，然后直接保存即可(会使用 **vi** 编辑器打开文件，修改和保存请使用 **vi** 方式处理)。

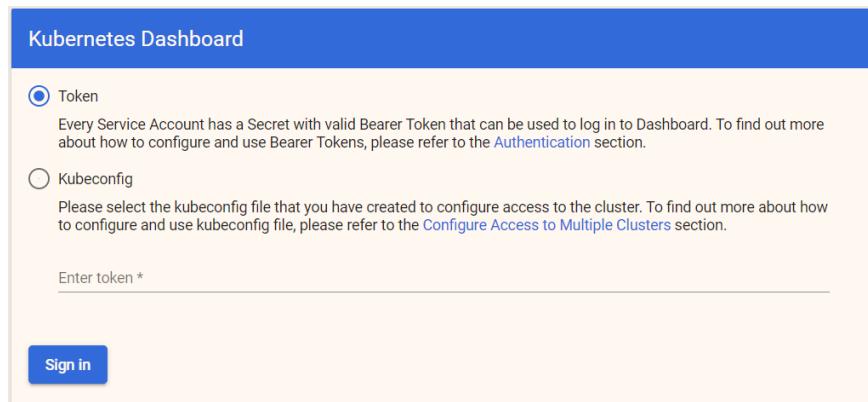
```
ports:
  port: 443
  protocol: TCP
  targetPort: 8443
selector:
  k8s-app: kubernetes-dashboard
sessionAffinity: None
type: NodePort
```

然后执行 `kubectl get services --namespace=kubernetes-dashboard -o wide` 命令，查看随机生成的端口。

```
root@master:~# kubectl get services --namespace=kubernetes-dashboard -o wide
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)
dashboard-metrics-scraper  ClusterIP  10.97.89.174 <none>        8000/TCP
kubernetes-dashboard   NodePort   10.108.209.60  <none>        443:31984/TCP
```

笔者的端口是 **31984**，接着可以在外网访问 <https://{}:31984>。注意，访问的 IP 是部署了 **dashboard** 的节点的 IP。

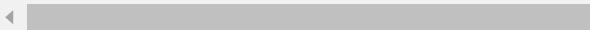
打开网页后，其页面如下：



可以看到，访问方式有 **Token** 和配置文件方式(**kubeconfig**)，这两者后面再讲。

通过下面这条命令我们可以查看 Token:

```
kubectl -n kube-system describe $(kubectl -n kube-system get secret -n kube-system -o
```



复制 token，填写到 Web UI 中，即可进入控制台。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间：2021-10-31 10:55:46

- 第三章：Pod部署和调度
 - 学习目标

第三章：Pod部署和调度

学习目标

部署、配置、标签选择、调度、副本、控制器、有状态的无状态的 Pod

- 讨论部署配置细节
 - 通过 `kubectl create/apply` 创建 Deployment，部署 Pod;
 - 通过 `kubectl edit` 修改对象;
 - 如何查看对象信息，`kubectl get`、`kubectl describe`、`-o wide`、`-o yaml`；
 - 创建 Service，`kubectl expose`；
 - 设置副本集，`replicaset`；
 - 查看对象支持的属性 `kubectl explain pods`，`kubectl explain pod.spec`
- 向上和向下扩展部署。
 - 扩容 Pod，设置副本集，`kubectl scale`、`kubectl edit`；
 - 自动扩容 Pod，水平缩放，比例缩放，根据 CPU、内存缩放，`kubectl autoscale`；
 - DaemonSet
- 实现滚动更新和回滚
 - 更换镜像版本、更新 Pod，`kubectl set image`；
 - 设置滚动更新速度，`maxSurge`、`maxUnavailable`；
 - 暂停和恢复更新，`kubectl rollout pause`、`kubectl rollout resume`；
 - 回滚旧版本，`kubectl rollout undo`；
- 使用标签选择各种对象以及调度
 - 了解 `label`、`selector`、`nodeSelector`；
 - 查询 `label`、选择 `label`、使用选择器调度 Pod;
 - 选择器运算符，等值选择、集合选择；
- 配置污点和容忍度
 - 亲和性和反亲和性
 - 配置污点和容忍度
 - 系统默认污点、`master` 的调度配置
- Job 和 CronJob
 - 了解 Job，完成数(`completions`)、工作队列(`parallelism`)、控制并行性、清理 Job。
 - CronJob 的时间表示
- StatefulSet
 - 有状态的应用

掌握以下命令的使用：

`kubectl`

`kubectl` 原理是请求 `apiserver` 完成某些操作，日常操作中，最常用的就是 `kubectl`。

`kubectl create {对象}` , 创建 deployment、job 等对象。

`kubectl apply -f` 应用 yaml 文件, 完成某些操作。

`kubectl get {对象}` 查询对象。

`kubectl scale {对象}` 伸缩对象数量(ReplicaSet)。

`kubectl expose` 创建 Service。

`kubectl describe` 获取对象详细的信息。

`kubectl exec` 在对象中执行命令, 例如 pod。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-11-07 08:16:26

- 3.1 Pod
 - Pod 基础知识
 - 创建 Pod
 - 了解 Pod
 - Pod 共享网络和存储
 - 划分 Pod 和容器
 - 何时使用多个容器
 - Pod 生命周期
 - 容器重启策略
 - Pod 的部署和管理
 - 创建 Pod
 - 覆盖容器命令
 - Pod 管理
 - 查看日志

3.1 Pod

Pod 是在 Kubernetes 中创建和管理的、最小的可部署的计算单元，是最重要的对象之一。一个 Pod 中包含一个或多个容器，这些容器在 Pod 中能够共享网络、存储等环境。

学习 Kubernetes，Pod 是最重要最基本的知识，本章将介绍什么是 Pod、Pod 的结构等，并练习创建 Pod。

Pod 基础知识

创建 Pod

创建一个 Pod 很简单，示例命令如下：

```
kubectl run nginxtest --image=nginx:latest --port=80
```

nginxtest 为 Pod 名称，并且为其映射端口为 80。

但是一般不会直接创建 Pod，因为手动创建的 Pod 不被“托管”起来，如果 Pod 故障或者其他原因消失了，不会自动恢复，而且 `kubectl run` 提供的参数有限。

Kubernetes Pod 中的所有容器共享一个相同的 Linux 命名空间(network、UTS、IPC)，而不是每个容器一个命名空间，因此 Pod 中的容器，网络、主机名称相同、IP 地址相同。据说在新版本的 Kubernetes 和 Docker 中，PID 命名空间也可以设置为相同的。由于 Mount、User 命名空间不共享，因此在容器中，文件系统和用户是隔离的。

另外我们也可以使用 yaml 方式定义 Pod，然后创建它。使用 YAML 文件定义需要的 Pod 格式示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginxtest
spec:
  containers:
  - image: nginx:latest
    name: nginxtest
    ports:
    - containerPort: 80
      protocol: TCP
```

但是描述 Pod 的 YAML 文件，包含哪些内容？格式如何？每个字段是什么意思？我们不可能记住所有 Kubernetes 对象的 YAML 文件的每个字段吧？

还好，Kubernetes 提供了 `kubectl explain` 命令，可以帮助我们快速了解创建一个对象的 YAML 需要哪几部分内容。例如创建 Pod 的 YAML 定义如下：

```
oot@master:~# kubectl explain pod
KIND:     Pod
VERSION:  v1

DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource is
created by clients and scheduled onto hosts.

FIELDS:
apiVersion  <string>
APIVersion defines the versioned schema of this representation of an
object. Servers should convert recognized schemas to the latest internal
value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.

kind  <string>
Kind is a string value representing the REST resource this object
represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.

metadata  <Object>
Standard object's metadata. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.

spec  <Object>
Specification of the desired behavior of the pod. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.

status  <Object>
Most recently observed status of the pod. This data may not be up to date.
Populated by the system. Read-only. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.
```

如果要查更深一层的字段，则可以：

```

root@master:~# kubectl explain pod.kind
KIND:     Pod
VERSION:  v1

FIELD:    kind <string>

DESCRIPTION:
  Kind is a string value representing the REST resource this object
  represents. Servers may infer this from the endpoint the client submits
  requests to. Cannot be updated. In CamelCase. More info:
  https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.

```

[Info] 提示

此方式查询到的文档，其列举的字段不一定都是 Pod YAML 所需要的，还需要读者多练习多学习，从文档中补充需要的信息。

了解 Pod

Pod 是 Kubernetes 中调度资源的最小单位，一个 Pod 中可以包含多个容器，Pod 中的容器被打包在一起作为一个整体，**Pod** 中的容器不会被分配到不同节点中，它们一定被部署到同一个节点中。

每个 Pod 有且只有一个唯一的 IP 地址，通过 `kubectl get pod {pod名称} -o wide` 可以查询到。

```

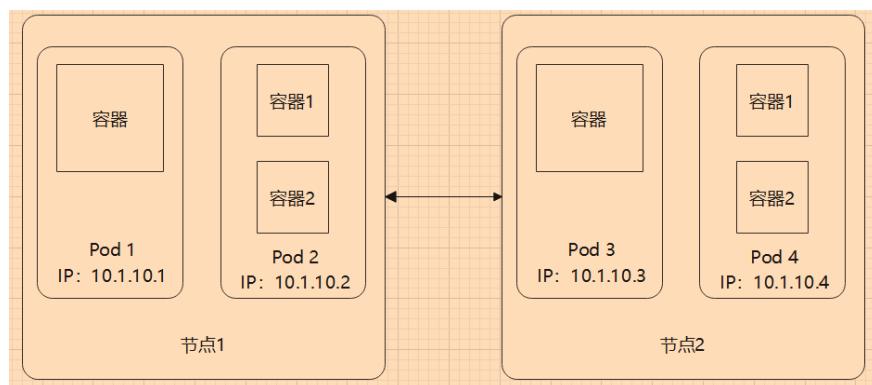
root@master:~# kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE     IP          NODE   NOMINATED NODE   R
nginxtest  1/1     Running   0          12m    10.32.0.2   slave1 <none>

```

[Info] 提示

`-o wide` 可以查看更多对象的信息，不只是 Pod，其他对象也可以加上去。

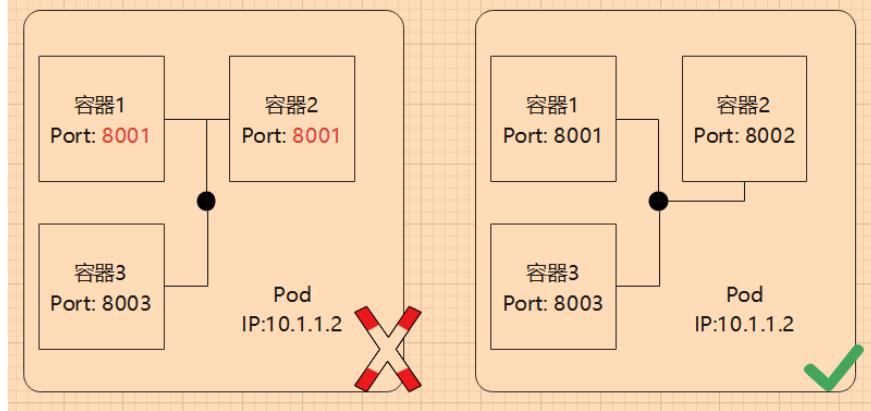
Pod 之间可以通过 IP 访问，这个 IP 可以 Ping 通。



Pod 共享网络和存储

我们可以把一个 Pod 形容为一个虚拟主机。在 Pod 中，所有容器(进程)都在一个主机上，我们知道，同一个主机上的所有进程共享着主机网络，多个进程自然不能同时占用一个端口，否则会冲突。容器共享 Pod 中的网络，所以 Pod 中的容器也

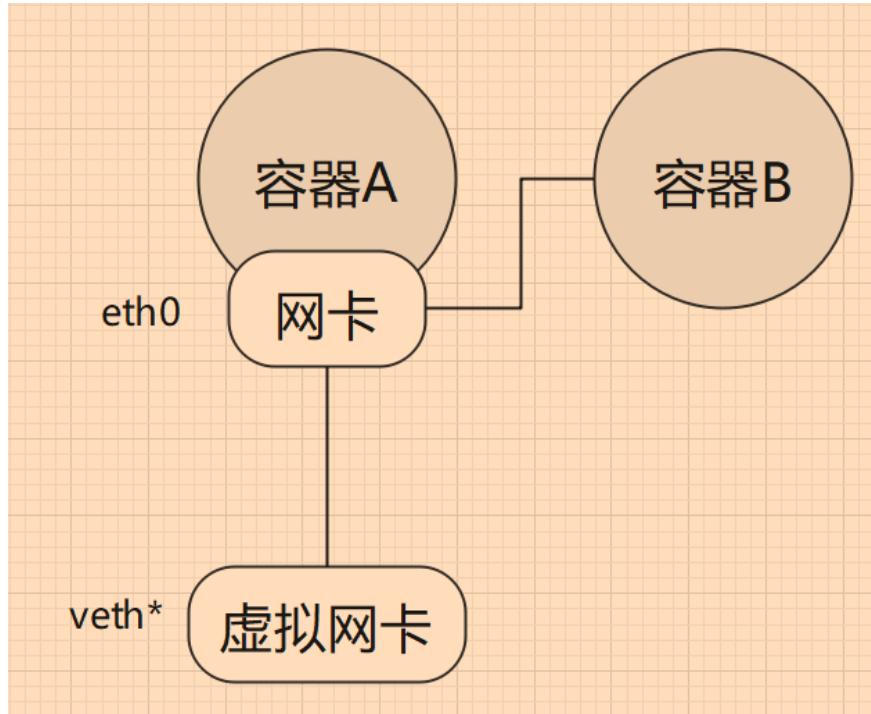
能够通过 `localhost` 相互访问，因此 Pod 中的容器(进程)不能暴露/使用相同的端口。



通过 Pod 和 `localhost`，解决了高度耦合的容器间通信问题。

Docker 有一个 Container 模式，能够让多个容器共享一个 Network Namespace，可以让一个容器和另一个容器共享 IP、端口范围。

假如容器 A 已经创建起来，那么容器 A 会创建一个虚拟网卡；然后指定容器 B 与容器 A 共享网络，那么两者就可以直接通过 `localhost` 进行通讯。



在 Kubernetes 中，当创建 Pod 时，会先启动一个 pause 容器，然后 Pod 中我们定义的容器会以 Container 模式共享 pause 中的网络。

```
docker ps | grep pause
```

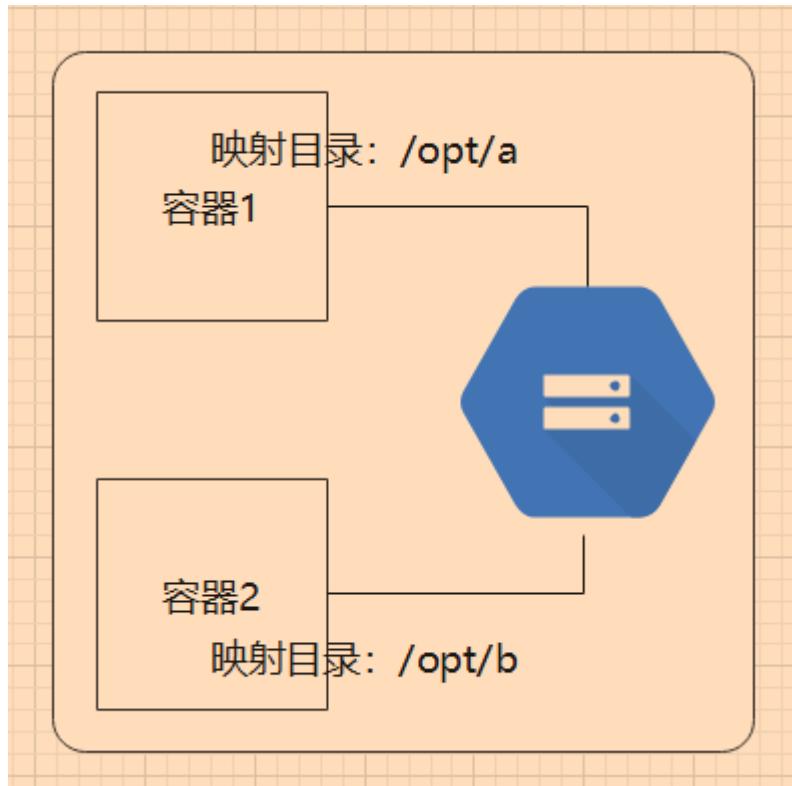
45cba7964aaa	k8s.gcr.io/pause:3.4.1	"/pause"	7 hours ago	Up 7 hours
4fabef02ccfa	k8s.gcr.io/pause:3.4.1	"/pause"	7 hours ago	Up 7 hours
f863176b3225	k8s.gcr.io/pause:3.4.1	"/pause"	24 hours ago	Up 24 hours
5b21446a8822	k8s.gcr.io/pause:3.4.1	"/pause"	2 days ago	Up 2 days
871d8e200364	k8s.gcr.io/pause:3.4.1	"/pause"	2 days ago	Up 2 days

当然，pause 不只是实现容器的网络互通，还有其它功能。

在第一章的 Docker 介绍中，谈到过此类容器，其目的是让其他容器联通起来，`pause` 在 Pod 的网络中相当于交换机。

Pod 中的容器是部分隔离的，每个容器都有自己的文件系统，各自的文件被隔离，容器不能访问或修改其它容器的文件。

为了让多个容器之间能够共享文件，可以使用卷，把同一个卷映射到容器中。



划分 Pod 和容器

容器中应只包含一个进程，或进程和创建的子进程。如果在同一个容器中包含多个进程，那么需要同时管理进程的启动、日志等，一个进程崩溃时，容易影响到另一个进程。由于多个进程都会记录信息到标准输出中(如控制台输出)，容器日志会合在一起，可能会导致出现问题难以排查。

一个容器只应该运行一个进程，但是他们放到一个 Pod 中就行了吗？例如程序和数据库，在设计时应该放到同一个 Pod，还是单独不同的 Pod？接下来我们简单讨论一下这个问题，限于经验和技术水平，笔者的论点可能不到位，读者可以多参考一下别的文章，了解如何设计这些架构。

下面以 Web 程序和数据库举例。

耦合

使用 Pod/容器的原因，是为了让不同服务能够降低耦合，能够隔离环境，如果程序跟数据库放在一起，是否能够有足够的隔离程度？如果 Web 跟数据库放在同一个 Pod，此时 web 跟数据库的实例(容器)数量是 1: 1。对于 Kubernetes 来说，Pod 是最小单位，Kubernetes 不能横向扩容单个容器，因此扩容的最小单位是

Pod, 多个容器必须捆绑在一起。同时 **Pod** 中的所有容器都使用同一机器的资源。在同一个 **Pod** 中的容器，在生命周期、计算机资源(内存、CPU)、实例数量、网络等都会耦合在一起。

请参考 <https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/>

访问压力

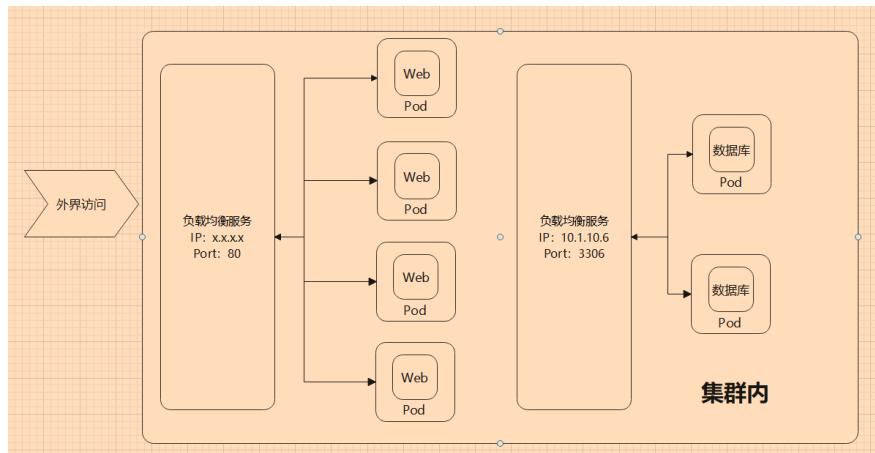
一般来说，**Web** 是要被外界访问的，但是数据库为了安全，应当避免能够公网访问，只有处于集群中的程序或客户端才能访问数据库。同时**Web**的访问是直接面向用户的，访问量肯定比数据库的访问量大得多，而且数据库需要的存储空间比**web**大得多，那么两者使用的计算资源并不相近。

Pod 可以使用服务器资源，当服务器压力过大时，当太多用户访问 **Web** 时，**Web**就要考虑扩容实例，可以在其它节点上部署相同的 **Pod**(扩容)，降低单节点访问压力。而一个数据库实例能够支持多个 **Web** 程序同时访问，那么数据库实例有必要跟 **Web** 放在同一个 **Pod** 中，保持 1: 1 的实例数量？

故障恢复

在 **Kubernetes** 中，容器应当是无状态的，也就是说容器或容器中的进程挂了，**Kubernetes** 可以快速在其它地方再创建一个 **Pod**，启动容器，维持一定数量的 **Pod** 实例。对于 **Web** 来说，只要配置文件和数据库数据在，再启动一个 **Web** 容器，结果是一样的，流水的程序铁打的数据，只要数据在，可以随时启动 **Web** 程序，很容易恢复服务。但是数据库却不一定，数据库的运维比 **Web** 程序复杂得多，我们要考虑数据的安全性和可用性，当容器甚至节点服务器挂了后、磁盘损坏等，如何恢复数据库。数据库的维护不觉得。

两者的维护难度不在同一水平上，此时我们要考虑两者放在不同的 **Pod** 中。(实际上很少将数据库放在容器中，一般都是裸机部署)。



其中负载均衡是通过 **Ingress** 和 **Service** 实现的，后面的章节会学习到。

何时使用多个容器

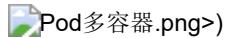
前面提到 **Web** 跟数据库，应当划分在不同的 **Pod** 中，类似地，对于微服务中的不同服务或模块，也应当放在不同的 **Pod** 中。微服务架构、容器化，并不是那么容易，例如，对于前后端分离的项目，前后端文件放在同一个容器中还是同一个 **Pod** 中还是不同 **Pod** 中？在设计中我们要考虑很多问题。

对于单体 Web 来说，一个程序中包含了所有服务，那么 Web 完全可以托管前端静态文件，前端文件跟后端程序打包在一起即可。例如 PHP、ASP.NET Core 等使用 `wwwroot`、`www` 等目录存储静态文件。

如果是一个较大的网站，网站使用了多个微服务，则前端更可能放到一个 Pod 中，用户访问前端页面，然后前端根据访问的模块，自动访问不同的服务。

如果前端和后端文件需要频繁发布，两者的发布版本分开工作，则为了避免一方等待另一方发布，或者从 Devops 角度，前端和后端文件可以放在不同容器中，然后通过存储卷，两个容器共享文件。

如果一个 Pod 中，包含一个主进程和多个辅助进程，则可以使用一个 Pod 部署多个容器，多个容器之间紧密联系。



具体怎么设计，需要根据实际情况考虑。

Pod 生命周期

当 Pod 被分配到某个节点时，Pod 会一直在该节点运行，直到停止或被终止，Pod 在整个生命周期中只会被调度一次。

Pod 的整个生命周期可能有四种状态：

- Pending，尝试启动容器，如果容器正常启动，则进入下一个阶段；
- Running，处于运行状态；
- Succeeded、Failed，正常结束或故障等导致容器结束；
- Unknown，因为某些原因无法取得 Pod 的状态。

我们可以创建一个 Deployment，查看当前正在发生的事件：

```
kubectl create deployment nginx --image=nginx:latest --replicas=3
```

```
kubectl get events
```

```
root@instance-2:~# kubectl get events
LAST SEEN TYPE      REASON          OBJECT                MESSAGE
4s Normal    Scheduled   pod/nginx-55649fd747-4vwq7   Successfully assigned default/nginx-55649fd747-4vwq7 to instance-2
4s Normal    Pulling    pod/nginx-55649fd747-4vwq7   Pulling image "nginx:latest"
4s Normal    Scheduled   pod/nginx-55649fd747-d7c9s   Successfully assigned default/nginx-55649fd747-d7c9s to instance-2
4s Normal    Pulling    pod/nginx-55649fd747-d7c9s   Pulling image "nginx:latest"
4s Normal    Scheduled   pod/nginx-55649fd747-qqszn   Successfully assigned default/nginx-55649fd747-qqszn to instance-2
1s Normal    Killing    pod/nginx-55649fd747-s4824   Stopping container nginx
38s Normal   Killing    pod/nginx-55649fd747-sr402   Stopping container nginx
38s Normal   Killing    pod/nginx-55649fd747-xtmhx   Stopping container nginx
4s Normal   SuccessfulCreate replicaset/nginx-55649fd747   Created pod: nginx-55649fd747-4vwq7
4s Normal   SuccessfulCreate replicaset/nginx-55649fd747   Created pod: nginx-55649fd747-qqszn
4s Normal   SuccessfulCreate replicaset/nginx-55649fd747   Created pod: nginx-55649fd747-d7c9s
4s Normal   ScalingReplicaSet deployment/nginx           Scaled up replica set nginx-55649fd747 to 3
```

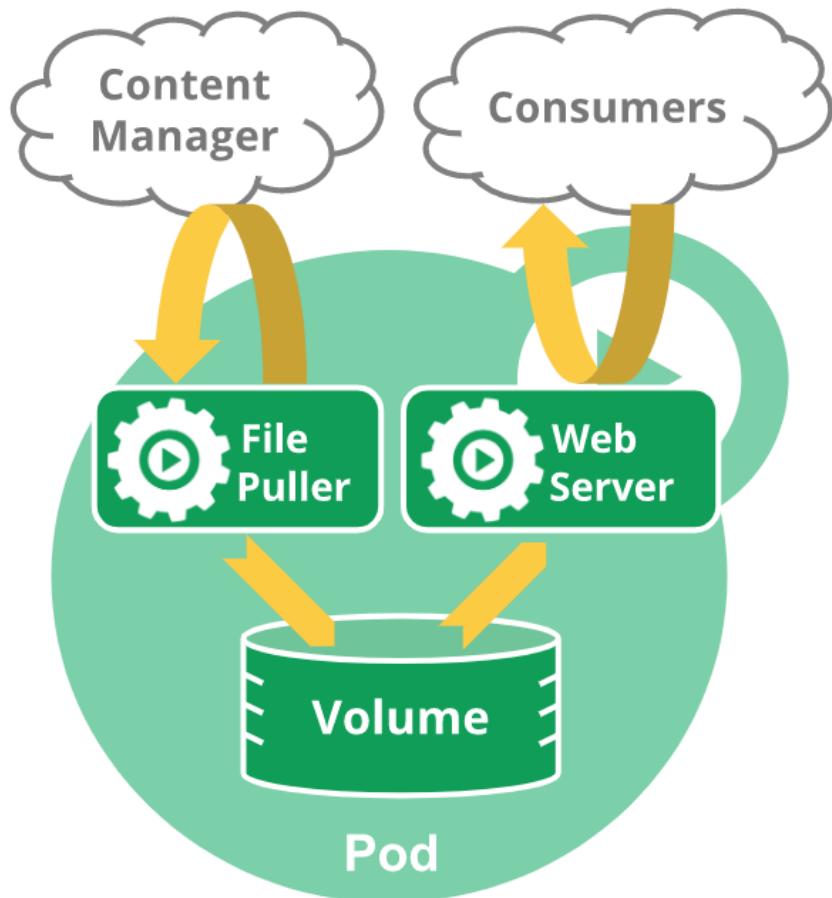
通过 `kubectl describe pod {pod名称}` 查看一个 Pod 的事件：

Type	Reason	Age	From	Message
Normal	Scheduled	3m22s	default-scheduler	Successfully assigned default/nginx-* to instance-2
Normal	Pulling	3m19s	kubelet	Pulling image "nginx:latest"
Normal	Pulled	3m17s	kubelet	Successfully pulled image "nginx:latest"
Normal	Created	3m17s	kubelet	Created container nginx
Normal	Started	3m16s	kubelet	Started container nginx

上面查询到的事件均发生在 Pod 的 Pending 状态，我们可以看到在这个阶段中，Pod 被调度，然后拉取镜像、启动容器，如果容器启动成功，Pod 便会进入 Running 状态。

事件记录保存在 etcd 中。

在 Kubernetes 中，Pod 被认为是相对的临时性实体，而不是长期存在的。由于 Pod 本身不具有治愈能力，如果节点故障或者节点资源耗尽、节点被维护、Pod 被驱逐等，那么 Pod 无法在节点上继续存活。无论 Pod 因为何种原因被删除，在 Pod 中的网络、存储卷等，也会被销毁，新的 Pod 被创建时，相关的网络、存储卷也会被重建。



【图来源：<https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/>】

[Info] 提示

由于 Pod 是临时性的，为了保障服务能够在 Pod 挂了后自动重建，可以使用户 Deployment、Daemon 等对象管理 Pod，这些对象被称为 控制器。

在删除 Pod 时，Kubernetes 会终止 Pod 中的所有容器，会向容器中的进程发生 SIGTERM 信号，等待进程的正常关闭，所以 Pod 可能不会被马上删除，当然如果进程不能正常关闭，Kubernetes 最多等待 30s，不然会使用 SIGKILL 杀死进程。

容器重启策略

这是一个简单的 Pod 的 YAML 定义：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
spec:
  containers:
  - image: nginx:latest
    imagePullPolicy: Always
    name: nginx
    ...
  restartPolicy: Always
```

在这个 YAML 中，有两个 `*Policy`，取值有 `Always`、`OnFailure` 和 `Never` 三种，它们代表了某种生命周期策略。

对于每个容器，都可以设置 `imagePullPolicy`，指示在拉取镜像时如果失败，是否进行重试。

在 `spec` 中，有个 `restartPolicy` 字段，其值默认为 `Always`，指示 Pod 中所有容器的重启动作，但是在 `Deployment`、`StatefulSet`、`DaemonSet` 等控制器中，`restartPolicy` 只支持 `Always`，不支持 `OnFailure` 和 `Never`。

[Info] 提示

如果容器启动失败，重试间隔会越来越长。kubelet 会在 10s 后重试第一次，如果还是失败，第二次 20s 后再重试；按照 10s、20s、40s、80s ... 的间隔重试，但最长不超过 5 分钟。如果容器被成功运行且运行了 10 分钟以上，那么计时器会被重置，下次出现故障时，按照 10s、20s 的间隔时间重试。

如果单独创建 Pod，并且设置了 `restartPolicy: Always`，那么 Pod 会一直停留在此节点上，如果容器故障，Pod 可能会无限重试。

如果我们使用 `Deployment`、`StatefulSet`、`DaemonSet` 等控制器管理 Pod，这些控制器能够处理 Pod/副本的管理、上线，并且在 Pod 失效时提供治愈能力，当，当然，这些东西后面再提。

节点上的 Pod 停止工作时，可以创建替代性的 Pod，Pod 被调度到一个健康的节点执行。

[Info] 提示

为什么要使用控制器管理 Pod 呢？

举个例子，笔者有个朋友，写了个 A 程序，这个程序能够在执行后，关闭计算机(关机)。本来是需要的时候执行一次，但是后面配置了一个守护进程，这个守护进程会自动启动 A 程序。这样出现了无限循环，开机 -> 启动守护进程 -> A 出现 -> 关机，结果这个朋友的电脑无法正常开机，一开机就被关机。

这种情况跟单独的 Pod 部署类似，很容易因为某些原因无限重试，并且一直驻留在节点上。因为重试是在原来的基础上进行重试，使用原来的文件、数据、网络等。控制器则可以将其重置，恢复 `出厂设置`。

如果一个节点上的 CPU、内存不够用了，那么容器有可能因为资源不足，无法启动，导致无限重试；如果使用控制器，控制器会在有充足资源的、健康的节点上重建 Pod。

Pod 的部署和管理

但是一般很少直接创建或管理 Pod，一般使用控制器来管理 Pod。下面列出一些控制器，在后面的学习中我们会一步步深入学习。

- Deployment
- StatefulSet
- DaemonSet

单独创建 Pod，一般用于临时调试的等。

创建 Pod

在 Kubernetes 中，所有对象都可以使用 YAML 表示。我们可以使用 YAML 来定义 Pod 对象，一个 Pod 的基本模板：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

如果要映射网络端口，则 YAML 文件为：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
      protocol: TCP
```

由于 Pod 中的容器共享网络，技术不加上 `ports`，照样能够访问。在 YAML 中多写一些，有助于其他人快速了解此 Pod 的定义信息。

将上面的 YAML 内容复制到 `pod.yaml` 中，然后执行命令应用 YAML：

```
kubectl apply -f pod.yaml
# 或
kubectl create -f pod.yaml
```

使用 `kubectl run` 命令也可以创建 pod，命令示例：

```
kubectl run nginx-pod --image=nginx:latest
```

覆盖容器命令

在 Pod 中可以配置容器的一些信息，也可以替换容器的启动命令，其配置格式如下：

```
spec:
  containers:
  - name: nginx
    image: nginx:latest
    command: ["/bin/command"]
    args: ["arg1","arg2","arg3"]
```

Pod 管理

输入 `kubectl get pods` 可以查看名为 `default` 命名空间的 Pod。输入 `kubectl get pods -o wide` 可以查看多几个字段的 Pod 信息，输入 `kubectl describe pods` 可以查看每个 Pod 的所有详细信息。

```
root@instance-2:~# kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE       IP           NODE      NOMINATED NODE
nginx    1/1     Running   0          11s      192.168.56.3   instance-2 <none>
```

可以看到 Pod 在第二个节点上部署，其 IP 为 `192.168.56.3`。Pod 的 IP 只能在被部署服务的节点上访问，不同节点不能访问其的 Pod。

`nginx` 默认使用了 `80` 端口，因此通过 `192.168.56.3:80`，可以访问到容器中的 `nginx` 服务。

```
root@instance-2:~# curl 192.168.56.3
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

[Error] 注意

只能在 Pod 部署的节点上连接到此 IP。如果要在不同的节点访问 Pod，需要安装 CNI 网络插件，如 flannel、calico、weave 等，在 2.2、2.3 中有介绍。

查看日志

在 Docker 中，我们可以通过 `docker logs {容器id}` 来查看容器中的日志，这些日志是进程打印到控制器的标准输出，例如 C# 的 `Console.WriteLine`、C 语言的 `printf`、Go 语言的 `fmt.Println`，Docker 的本地日志驱动会捕获容器的 `stdout/stderr` 输出记录驱动器。

Docker 日志默认限制 5 个日志文件，每个最大 20 MB，每天会轮替一个日志文件，默认情况下会为每个容器保留 100MB 的日志消息，并使用自动压缩来减少磁盘文件大小。

Docker 日志驱动程序使用基于文件的存储。文件格式和存储机制被设计为由 Docker 守护进程独占访问，不应被外部工具使用，因为在未来的版本中实现可能会发生变化。

笔者没有明确查找到 Docker 的日志具体限制情况，以上内容来自 <https://docs.docker.com/config/containers/logging/local/> 的参考资料。

在 Kubernetes 中，也可以通过命令快速查看 Pod 中的容器的日志。

如果 Pod 中只有一个容器，则直接使用类似命令即可：

```
kubectl logs {pod名称}
```

如果 Pod 中有多个容器，则需要指定容器名称：

```
kubectl logs {pod名称} -c {容器名称}
```

`kubectl logs` 只能获取当前正在运行的 Pod 的日志，如果 Pod 被删除，所有日志记录都会被删除。

查看、维护 Pod 状态，比较常用的命令有：

- **kubectl get** - 列出对象资源，如 `kubectl get pods`；
- **kubectl describe** - 显示有关资源的详细信息，如 `kubectl describe pod nginxtest`；
- **kubectl logs** - 打印 pod 和其中容器的日志；
- **kubectl exec** - 在 pod 中的容器上执行命令，格式为 `kubectl exec {pod 名称} -c {容器名称} -- {要执行的命令}`，`--` 用于分隔命令，其后面的参数均表示要传递进容器的命令。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-08 20:36:47

- 3.2 Deployment部署
 - Deployment
 - 创建 Deployment
 - kubectl apply/create
 - 检查 YAML
 - 查看 Deployment
 - 查看 Pod
 - 实时修改
 - 导出 yaml
 - 多个 容器
 - 总结

3.2 Deployment部署

Deployment 是 Kubernetes 提供的一种自我修复机制来解决机器故障维护的问题。前面提到了单独部署 Pod，但是这种方式只适合临时的 Pod，用于测试调试。如果要用于生产，则需要 Deployment 等控制器管理部署 Pod，维护 Pod 的副本数量以及 Pod 监控和维护。

对于 Kubernetes 对象的部署，例如 Pod、Deployment、Service 等，有三种部署方式：

- Using Generators (Run, Expose)
- Using Imperative way (Create)
- Using Declarative way (Apply)

在 2.1 章中，我们已经学习了 Run 和 apply 等，在本篇以及后面的章节中，我们会一步步深入学习这些部署方式。

本篇包含或需要掌握以下内容：

- 创建 Deployment
- 修改 Deployment
- 查看 Deployment、Pod、Services、副本

在本篇文章中，我们将部署一个 Nginx 实例，并学会部署以及管理 Deployment、Pod。

Deployment

当我们单独使用 docker 部署应用时，为了应用挂了后能够重启，我们可以使用 --restart=always 参数，例如：

```
docker run -itd --restart=always -p 666:80 nginx:latest
```

但是这种方式只能单纯重启容器，并不具备从机器故障中恢复的能力，即当一台服务器挂了后，此服务器上所有的容器全部挂掉。

Kubernetes Deployment 是一种 Pod 管理方式，它可以指挥 Kubernetes 如何创建和更新你部署的应用实例，创建 Deployment 后，Kubernetes master 会将应用程序调度到集群中的各个节点上。Kubernetes Deployment 提供了一种与众不同的应用程序管理方法。

Deployment 的创建，有两种方法，一种是直接使用命令创建(`kubectl create`)，一种是通过 YAML(`kubectl apply`)，后面我们会介绍这两种创建方法。

创建 Deployment

在 Kubernetes 中，Pod 是调度的最小单位，一个 Pod 中包含多个容器，所以我们各种操作都是在 Pod 之上。

我们来使用 deployment 部署一个 Pod，这个 Pod 包含一个 Nginx 容器。

```
kubectl create deployment nginx --image=nginx:latest
```

格式：

```
kubectl create deployment {deployment对象名称} --images={镜像名称和标签}
```

此时，nginx 容器会以 Pod 的方式部署到节点中，但是被部署到哪个节点是随机的，如果你只有一个 worker 节点，则 Pod 必定在这个 Worker 节点上。当然，我们可以获取到具体的调度信息，从中查看 Pod 被调度到哪个节点。

```
root@instance-1:~# kubectl get deployments -o wide
NAME      READY   UP-TO-DATE   AVAILABLE   AGE      CONTAINERS   IMAGES       SELECTOR
nginx     1/1     1           1           52s     nginx        nginx:latest   app=nginx
root@instance-1:~# kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE      IP           NODE
nginx-55649fd747-s4824  1/1     Running   0          61s     192.168.56.4  instance-2
```

可以看到，Pod 在 instance-2 中运行着。

Deployment 会为我们自动创建 Pod，Pod 由 `{deployment名称}-{随机名称}` 组成。

[Info] 提示

还有一个地方也说一下，`kubectl get xxx` 时，带不带 `s` 都没关系，例如 `kubectl get nodes` / `kubectl get node` 都是一样的。

不过，一般从语义上，我们获取全部对象时，可以使用 `kubectl get nodes`，获取具体的对象时，可以使用 `kubectl get node nginx`。类似的，`kubectl describe nodes`、`kubectl describe node nginx`。实际上加不加 `s` 都一样。

kubectl apply/create

当我们创建一个 deployment 时，`kubectl create` 和 `kubectl apply` 效果是一样的，但是 `apply` 还具有更新(update)的功能。

`kubectl apply` 会在以前的配置、提供的输入和资源的当前配置之间 找出三方差异，以确定如何修改资源，`kubectl apply` 命令将会把推送的版本与以前的版本进行比较，并应用你所做的更改，但是不会自动覆盖任何你没有指定更改的属性

另外还有 `kubectl replace`、`kubectl edit`。`kubectl replace` 是破坏性更新/替换，容易导致问题；`kubectl edit` 可以更新 Deployment 等已存在的对象。

根据 Kubernetes 官方的文档说明，应始终使用 `kubectl apply` 或 `kubectl create --save-config` 创建资源。

前面已经学习了 `kubectl create`，这里学习一下 `kubectl apply`。

通过 YAML 文件部署 nginx：

```
kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
```

很多开源软件提供了 YAML 文件，我们通过 YAML 文件可以快速部署服务，如 Redis、Consul 等。

这里再说一下创建 Deployment 的区别。

如果使用 `create` 创建，命令格式：

```
kubectl create deployment {deployment的名字} --image={镜像名称}
```

如果使用 `apply` 命令创建，YAML 中需要指定一些信息，可定制性很高。

```
kind: Deployment
...
metadata:
  name:nginx
...
spec:
  containers:
    - image: nginx:latest
```

然后执行 `kubectl apply -f xxx.yaml` 文件。

一个是 `kubectl create deployment`；另一个是 `kubectl apply -f`，在 yaml 中指定 `kind: Deployment`。

如果我们只需要快速创建，使用命令形式就行；如何生产生产，还是得使用 YAML 文件，并于留存记录。

要删除一个对象，可以使用 `kubectl delete -f {名称}.yaml`，如删除 calico。

```
kubectl delete -f calico.yaml
```

检查 YAML

有时我们不知道我们的创建命令或 yaml 是否正确，可以使用 `--dry-run=client`，`--dry-run=client` 参数来表示当前内容只是预览而不真正提交。

```
kubectl create deployment testnginx --image=nginx:latest --dry-run=client
```

在一些 k8s 认证中，我们没时间一点点写 yaml，但是又需要定制，此时可以使用 `--dry-run=client -o yaml`，既可以不生效 Deployment，又可以导出 yaml 文件。

[Info] 提示

`-o wide` 可以查看对象更多的字段信息；`kubectl describe` 可以查看对象的全部详细信息；`-o yaml` 或 `-o json` 可以查看对象的定义/描述文件。

`--dry-run` 取值必须为 `none`、`server` 或 `client`。如果客户端策略，只打印将要发送的对象，而不发送它。如果是服务器策略，提交服务器端请求而不持久化资源。

命令示例如下：

```
kubectl create deployment testnginx --image=nginx:latest --dry-run=client -o yaml  
# -o json 可以输出 json 格式
```

```
root@master:~# kubectl create deployment testnginx --image=nginx:latest --dry-run=client -o yaml  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  creationTimestamp: null  
  labels:  
    app: testnginx  
    name: testnginx  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: testnginx  
  strategy: {}  
  template:  
    metadata:  
      creationTimestamp: null  
      labels:  
        app: testnginx  
    spec:  
      containers:  
      - image: nginx:latest  
        name: nginx  
        resources: {}  
  status: {}
```

轻松获得 YAML 文件

使用这样的方法，可以快速获得需要的 YAML 模板，然后复制到 YAML 文件，根据需要改动、定制。除了 deployment，其它 kubernetes 对象也可以使用这种方法。

查看 Deployment

我们以 Deployment 的方式部署 Pod，就会创建一个 Deployment 对象，获得 deployment 列表：

```
kubectl get deployments  
kubectl get deployments -o wide
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE			
nginx	1/1	1	1	2m24s			
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
nginx	1/1	1	1	2m42s	nginx	nginx:latest	app=nginx

在 `kubectl get ...` 后面加上 `-o wide` 可以获得更多的标签信息。

使用 `kubectl get events` 可以获得集群中最近发生的事件，如创建 Deployment 到部署容器过程的详细事件记录。

```
Successfully assigned default/nginx-55649fd747-wdrjj to instance-2
Pulling image "nginx:latest"
Successfully pulled image "nginx:latest" in 8.917597859s
Created container nginx
Started container nginx
Created pod: nginx-55649fd747-wdrjj
Scaled up replica set nginx-55649fd747 to 1
```

使用 `kubectl describe deployment nginx` 可以获得更加详细的信息，是各种信息的集合。

```
Name: nginx
Namespace: default
CreationTimestamp: Mon, 03 May 2021 09:22:40 +0000
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision: 1
Selector: app=nginx
Replicas: 1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image: nginx:latest
      Port: <none>
      Host Port: <none>
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Conditions:
    Type Status Reason
    ---- ------
    Available True   MinimumReplicasAvailable
    Progressing True   NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet: nginx-55649fd747 (1/1 replicas created)
Events:
  Type Reason Age From Message
  ---- ---- - - - -
  Normal ScalingReplicaSet 13m deployment-controller Scaled up replica set nginx-55649fd747 to 1
```

查看 Pod

我们没有直接创建 Pod，而是通过 Deployment 创建，接下来我们需要了解如何查看 Pod。

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-55649fd747-msw8g	1/1	Running	0	4h16m

可以看到一个 Pod 名为 `nginx-`，因为我们是利用 Deployment 部署 Pod 的，没有指定这个 Pod 的名称，所以默认 Pod 名称以 Deployment 名称为前缀。

我们查看这个 pods 被部署到了哪个节点上：

```
kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx-55649fd747-msw8g	1/1	Running	0	4h19m	192.168.56.57	instance-2

可以看到，这个 Pod 在 `instances-2` 这个节点上，同时这个 Pod 也有一个 IP，Kubernetes 会为每个 Pod 分配一个唯一的 IP，这个 IP 可以在节点上访问，其它 Pod 也可以通过 IP 访问此 Pod。

由于这个 Pod 里面的容器是 Nginx(80端口)，所以我们可以访问这个 IP 可以打开 Nginx 页面。

```
root@instance-1:~# curl 192.168.56.57
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
... ...
```

对于没有指定的 Deployment、Pod、Service 等对象，会默认分配到 default 命名空间中，我们在执行 `kubectl get {}` 时，会获取 default 命名空间中的对象，后面加上 `--all-namespaces` 可以获得所有命名空间中的对象，使用 `--namespace=` 可以指定获取某个命名空间中的对象。

```
kubectl get pods --all-namespaces
# kubectl get pods --namespace=default
```

NAMESPACE	NAME	READY	STATUS
default	nginx-55649fd747-msw8g	1/1	Running
kube-system	etcd-instance-1	1/1	Running
kube-system	kube-apiserver-instance-1	1/1	Running
kube-system	kube-controller-manager-instance-1	1/1	Running
kube-system	kube-proxy-bhj76	1/1	Running
kube-system	kube-proxy-pm597	1/1	Running
kube-system	kube-scheduler-instance-1	1/1	Running
kubernetes-dashboard	dashboard-metrics-scraper-856586f554-4nd9v	1/1	Running
kubernetes-dashboard	kubernetes-dashboard-78c79f97b4-288js	1/1	Running

在 worker 节点上执行 `docker ps`，可以看到 Nginx 容器：

```
root@instance-2:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
fe7433f906a0 nginx "/docker-entrypoint...." 7 seconds ago Up 6 seconds
```

实时修改

使用 `kubectl edit` 可以实时修改一个对象，这里我们使用前面已经部署好的 Nginx 这个 Deployment，因为创建的 Pod 名称有点长，我们希望直接改成 nginx。

修改 deployment：

```
kubectl edit pod nginx-55649fd747-msw8g
```

在 `metadata` 字段中，找到 `name: nginx-55649fd747-msw8g`，修改为合适的名称。

修改完毕后，会提示：

```
A copy of your changes has been stored to "/tmp/kubectl-edit-w11bx.yaml"
error: At least one of apiVersion, kind and name was changed
```

出于某些原因，直接修改 `name` 字段是不行的，还需要修改其它地方。

这里读者了解，可以通过 `kubectl edit` 实时直接修改对象即可。后面还会介绍另一种修改方法。

导出 yaml

我们可以从已经创建的 `Deployment`、`Pod`、`Service` 等对象导出 yaml 文件，使用 `-o yaml` 即可导出(`-o json` 导出json)。

```
kubectl get deployment nginx -o yaml  
# 保存到文件  
# kubectl get deployment nginx -o yaml > mynginx.yaml
```

然后终端会打印：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  annotations:  
    deployment.kubernetes.io/revision: "1"  
  creationTimestamp: "2021-04-21T00:37:13Z"  
  generation: 1  
  labels:  
    app: nginx  
    name: nginx  
    namespace: default  
  ...
```

虽然我们创建对象时的命令很简单，但是生成的 YAML 很复杂。

我们可以尝试把 yaml 导出到 `mynginx.yaml` 文件中，然后我们删除这个 `Deployment`。

```
kubectl get deployment nginx -o yaml > mynginx.yaml
```

```
kubectl delete deployment nginx
```

然后利用导出的 `mynginx.yaml` 再创建一个 `Deployment`。

```
kubectl apply -f mynginx.yaml
```

多个容器

一个 `Pod` 中，是可以包含多个容器的，我们可以使用命令或 YAML 形式创建包含多个容器的 `Pod` 服务。示例如下：

```
root@master:~# kubectl create deployment testnginx --image=nginx:latest --image=busybox:latest
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: testnginx
    name: testnginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: testnginx
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: testnginx
    spec:
      containers:
        - image: nginx:latest
          name: nginx
          resources: {}
        - image: busybox:latest
          name: busybox
          resources: {}
status: {}
```

总结

本小章中不只是讲解 **Deployment**, 还包含了很多日常管理对象的命令, 读者可多做练习, 加深记忆。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间: 2021-11-08 20:55:17

- 3.3 副本集(ReplicaSet)
 - 副本集
 - Deployment 保护 Pod
 - 扩容 Pod
 - Scale
 - DaemonSet
 - 负载均衡
 - ReplicationController 和 ReplicaSet

3.3 副本集(ReplicaSet)

ReplicaSet 也是一种管理 Pod 的对象，跟上一章提到的 Deployment 一样，可以守护 Pod，保证 Pod 挂了后能够恢复。

副本集

ReplicaSet 为副本集的意思，Deployment 部署的 Pod，可以指定其副本数量，副本数量就是表示部署多少个 Pod，一个 Pod 模板生成 N 个 Pod 实例，而不是指一个原型+多个克隆，数量是 N，而不是 N+1。

Deployment 保护 Pod

前面我们学习到 Deployment，使用 Deployment 部署 Pod，当 Pod 或所在节点出现故障时，Deployment 会自动创建新的 Pod。

我们执行 `kubectl get deployments` 命令，输出：

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	38m

- NAME 列出了集群中 Deployment 的名称。
- READY 显示应用程序的可用的副本数。显示的模式是“就绪个数/期望个数”。
- UP-TO-DATE 显示为了达到期望状态已经更新的副本数。
- AVAILABLE 显示应用可供用户使用的副本数。
- AGE 显示应用程序运行的时间。

如果你使用命令创建 Deployment，在 Pod 没有创建完成前，READY 字段可能显示为 `0/1`。

这是因为 Deployment 中默认只维护一个 Pod 实例，我们可以查看 Deployment 的 YAML 文件，找到：

```
# 查看命令 kubectl get deployment nginx -o yaml
...
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  ...
...
```

`replicas` 字段设置了维持多少个 Pod 实例，所以当 Pod 挂了后，Deployment 会重新创建一个 Pod，而不是两个，三个。这个字段跟 ReplicaSet 有关，创建 Deployment 后，会自动创建 ReplicaSet，ReplicaSet 管理的副本数量跟 YAML `replicas` 字段值一致。

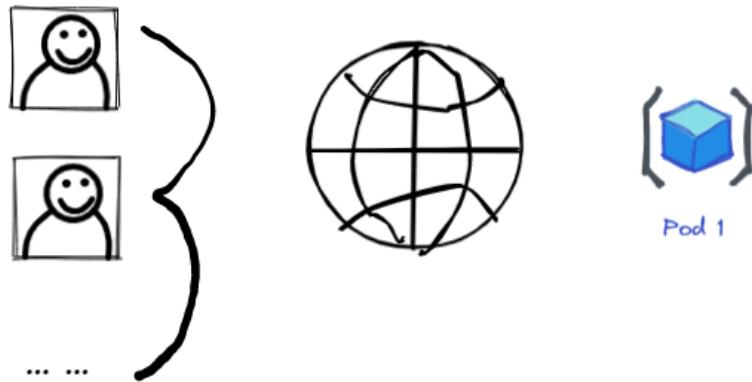
查看 replicaset 实例：

```
root@slave1:~# kubectl get replicaset
NAME          DESIRED   CURRENT   READY   AGE
nginx-55649fd747    1         1         1      22h
```

replicaset 可缩写为 rs，即 `kubectl get rs`。

扩容 Pod

在项目初期，用户人数少的时候，单个应用实例可能就满足了所有用户的访问需求。



但是随着时间发展，用户数量越来越多，单个应用无法撑起高并发，访问速度变慢了，响应时间变长了，内存爆炸了。



这个时候，如果可以提供多个实例，则可以均分用户的访问请求，让一个 Pod 只负载一部分用户，这样就能有效地支持亿级用户并发。



支持亿级并发？有这么简单吗？当然没有。

回归正题，要扩展 Pod 实例，需要保证 Pod 应当是无状态的，即每个 Pod 都是一样的，它们不能表现出差异来，同一个请求无论被哪个 Pod 处理，其响应结果都是一致的。

在 1.4 章中，根据云原生十二因素(<https://12factor.net>)的方法论和核心思想，一个 Processes 应当是无状态的，任何持久化的数据都要存储在后端服务中。A 镜像，启动 N 个 docker 容器，端口为 801、802、803...，他们其实都是一样的，我们访问哪个容器，最终提供的服务都是一致的。

如果我们把这些容器放到不同 Node 中，再通过 k8s，就可以为多个实例之间分配流量，即负载均衡。即，**加机器就能解决问题**。

在 Deployment 中，可以通过指定 YAML 文件的 `.spec.replicas` 字段或者以命令参数 `--replicas=` 设置副本数量。

我们可以动态修改一个控制器只能的 Pod 数量，直接编辑 YAML 文件即可：

```
kubectl edit deployment nginx
```

在 `spec` 字段后面找到 `replicas: 1`，修改为 `replicas: 2`。

```
resourceVersion: 1557542
uid: e4cf6984-7620-46fc-995b-010676d9e4a0
spec:
  progressDeadlineSeconds: 600
  replicas: 2 ←
  revisionHistoryLimit: 10
  selector:
    matchLabels:
```

再次查看 Deployment：

```
root@instance-1:~# kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
nginx    2/2     2           2           24m
```

在创建 Deployment 时，我们也可以使用 `--replicas=` 参数决定 Pod 的副本数量。

```
kubectl create deployment nginx --image=nginx:latest --replicas=2
```

Scale

前面，已经通过修改 Deployment 对象的 YAML 文件实现多副本扩容，但是我们总不能老是直接操作 YAML 文件吧？Kubernetes 提供了 `kubectl scale` 扩容命令，可以为指定对象扩容 Pod 数量。

为 Deployment 对象扩容 Pod 数量：

```
kubectl scale deployment nginx --replicas=3
```

然后等几秒后执行 `kubectl get deployments` 查看结果。

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	3h15m

查看 ReplicaSet：

```
root@slave1:~# kubectl get rs
NAME          DESIRED   CURRENT   READY   AGE
nginx-55649fd747    3         3         3      23h
```

执行 `kubectl get pod -o wide` 可以输出信息的 pod 信息。

NAME	READY	STATUS	ESTARTS	AGE	IP	NODE	NOMINATED NODE
nginx-581	1/1	Running	0	3h11m	192.168.56.24	instance-2	<none>
nginx-582	1/1	Running	0	3m30s	192.168.56.25	instance-2	<none>
nginx-583	1/1	Running	0	3m30s	192.168.56.26	instance-2	<none>

注，笔者删除了Name的部分名称

当我们使用 `kubectl delete xxx` 删除 pod 时，或者 Pod 故障时、手贱改了 Pod 状态时，Deployment 会自动保持三个副本集，会自动启用新的 pod，你可以删除其中一个或多个 Pod，再查看 Pod 数量，会发现一直保持 3 个。

ReplicaSet 可以简单地设置副本数量，而我们后面会学习到更加复杂的 Pod 扩容收缩机制。

`kubectl scale` 命令也可作用于 ReplicaSet 和 ReplicationController，两者可用简化名称 rs、rc 代替。

```
kubectl scale rc {名称} --replicas=10
kubectl scale rs {名称} --replicas=10
```

DaemonSet

在 Kubernetes 中，负载类型有 Deployments、ReplicaSet、DaemonSet、StatefulSets 等(或者说有这几个控制器)。

前面已经介绍过 **Deployments** 和 **ReplicaSet**, 它们都可以通过命令和 YAML 创建, 但是 **ReplicaSet** 一般没必要使用 YAML 创建的。

DaemonSet 可以确保一个节点只运行一个 Pod 副本。

假如有个 Pod, 当新的节点加入集群时, 会自动在这个节点上部署一个 Pod; 当节点从集群中移开时, 这个 Node 上的 Pod 会被回收; 如果 DaemonSet 配置被删除, 则也会删除所有由它创建的 Pod。DaemonSet 无视节点的排斥性, 即节点可以排斥调度器在此节点上部署 Pod, DaemonSet 则会绕过调度器, 强行部署。

Kubernetes 的一些系统服务, 也是 DaemonSet 部署模式。

DaemonSet 的一些典型用法:

- 在每个节点上运行集群守护进程
- 在每个节点上运行日志收集守护进程
- 在每个节点上运行监控守护进程

在 yaml 中, 要配置 Daemon, 可以使用 `tolerations`, 配置示例:

```
kind: DaemonSet
...
...
```

其它地方跟 Deployment 一致, 这里据不多说了, 读者了解即可。

关于 3.10 章会提及。

负载均衡

我们可以通过 Deployment, 或者 DaemonSet, 为 Pod 创建多个副本, 这些 Pod 副本实例, 会自动分到各个节点上。

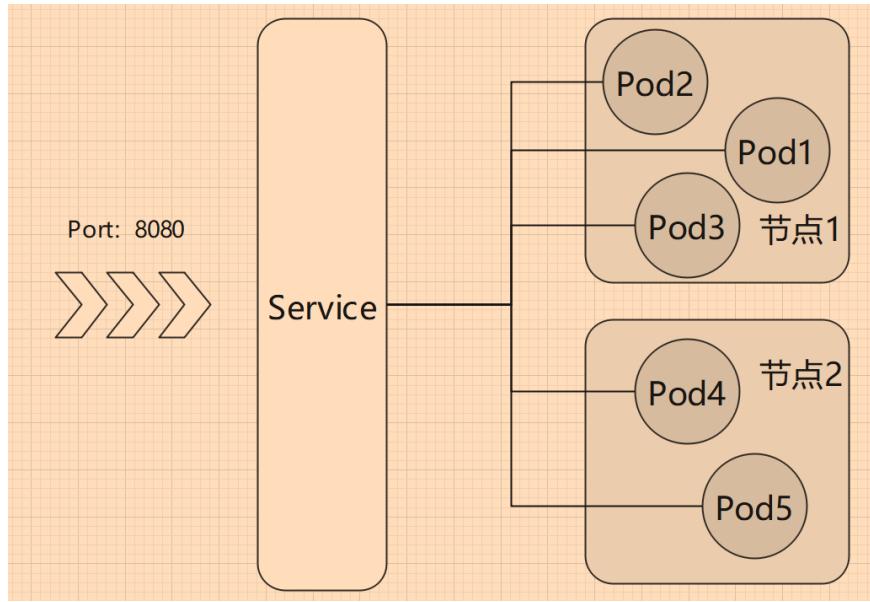


创建多个 Pod 能够实现负载均衡的原因是 Pod 被分配到多个节点中, 利用了多台服务器的计算资源, 本质是分配更多的资源, 如果只有一个服务器, 那么再扩容 Pod, 能够使用的资源是有限的, 不会带来什么效果。当然, 如果 Pod 被设置了资源限制, 只要该服务器上还有剩余资源, 那么分配更多的 Pod, 也是有效的。这点我们后面的章节再讲解。

Pod 实例数量是多了, 也分配到不同的节点上, 但是我们怎么访问 Pod? 难道 2 个节点, 我们要单独访问 IP?

其实我们无需担心太多, 因为 Kubernetes 提供了 Service、Ingress 等网络服务, 我们以 Service 举例。

Pod 提供了一个 8080 端口, 我们可以创建一个 Service, 这个 Service 对外也提供了 8080 端口, 当用户访问公网 IP:8080 时, Service 会随机选择一个 Pod 为用户提供服务, 我们无需关注访问了哪个 Pod。



不过 Deployment 部署的 Pod，假如有 5 个节点，要部署 5 个 Pod，不会每个节点都刚好有一个 Pod。同样，5 个节点，10 个 Pod，不一定每个节点都有两个 Pod。如果我们需要 Pod 能够平均地分配到节点中，可以使用 DaemonSet。

ReplicationController 和 ReplicaSet

ReplicationController、ReplicaSet 都是 Kubernetes 里面的副本管理机制，能够保证最终具有一定数量的 Pod 处于运行状态中，当节点故障或者 Pod 故障时、Pod 被节点逐出时，能够及时在其它节点上恢复一定数量的 Pod 实例。

ReplicationController 出现较早，后面出现了 ReplicaSet 用于替代它。从中文翻译来看，ReplicationController 译为 复制控制器，ReplicaSet 译为 副本集。两者主要区别是选择器支持，在 [3.7 标签](#)、[3.8 调度](#) 两章，你可以了解标签和选择器的使用方法。

ReplicationController 只支持等值选择器，示例：

```
environment = production
```

ReplicaSet 支持基于集合的选择器：

```
environment in (production, qa)
```

另外就是 ReplicaSet 一般不单独使用，而是结合 Deployment 等一起使用。

ReplicaSet 虽然是 ReplicationController 的替代方案，但是两者属于不同的资源。

```
root@instance-2:~# kubectl get replicationcontroller
No resources found in default namespace.
root@instance-2:~# kubectl get replicaset
NAME          DESIRED   CURRENT   READY   AGE
nginxtaint-6c6dd878f9  5         5         5      2d19h
```

既然 ReplicationController 要淘汰了，那么在本章以及后面章节，只会讨论 ReplicaSet。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-04 21:41:29

- 3.4 Pod 端口映射
 - containerPort
 - 网络端口映射
 - 本地端口

3.4 Pod 端口映射

在 3.1, 3.2 中, 我们部署过了 Nginx 容器, 使用了 `--port=8080` 或 `containerPort: 8080` 为 Pod 暴露一个端口, 本章只是简单地为 Pod 创建 Service, 并且介绍 Pod 的一些网络知识, 在第四章中会详细讲解网络方面的知识。

containerPort

这个字段用于规范化声明容器对外暴露的端口, 但这个端口并不是容器映射到主机的端口, 它是一个声明式的字段, 属于容器端口规范。

在很多情况下, 我们不需要设置此 `containerPort` 也可以直接访问 Pod。

读者可以把上一章中创建的 `deployment` 删除, 然后重新创建。

```
kubectl create deployment nginx --image=nginx:latest
```

然后获取 Pod 的 IP:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NON
nginx-55649fd747-zndzx	1/1	Running	0	7m57s	10.32.0.2	slave1	<none>

然后访问 10.32.0.2, 会发现正常打开。

```
root@master:~# curl http://10.32.0.2:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

加不加端口映射都可以访问 80, 它是一个容器端口规范, 跟 Dockerfile 的 `port` 一样, 不具有强制开放的功能, 在创建 Service 时有用处。

[Error] 提示

如果你没有配置过 CNI 网络插件，则创建的 Pod IP，只能在所在的节点中访问，要跨节点访问，请按照 2.2 的初始化网络，或 2.3 中的 Calico 一节，安装好网络插件。

另外，如果 Calico 配置错误，会导致一直无法创建 Pod。

我们在创建 Pod 时，如果指定了 `--port`，那么这个端口便会生成 `containerPort`，可以使用下面的命令查看创建的 Deployment YAML 的定义：

```
kubectl create deployment nginx --image=nginx:latest --port=80 --dry-run=client -o yaml
```



```
containers:
- image: nginx:latest
  name: nginx
  ports:
  - containerPort: 80
resources: {}
```

网络端口映射

对于 docker，我们要映射端口时，可以使用 `docker ... -p 6666:80`，那么对于直接创建或使用 Deployment 等方式部署的 Pod，都有一个 Pod IP 可以在集群中的所有节点中访问，但是这个 IP 是虚拟 IP，不能在集群外中访问，即使都是内网机器，没有加入 Kubernetes 集群，一样不能访问。如果我们要把端口暴露出去，供外网访问，则可以使用 Service。

关于 Service 的知识，在第四章中会详细讲解，这里仅说明如何创建 Service，以及 `containerPort` 的作用。

在上一小节中，我们创建的 Pod 没有声明过 `containerPort`，因此我们创建 Service 的时候，需要指定映射的端口。

查看上一节创建的 Deployment、Pod：

```
kubectl get deployments
kubectl get pods
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	1/1	1	1	5m44s

NAME	READY	STATUS	RESTARTS	AGE
nginx-55649fd747-9vfrx	1/1	Running	0	5m44s

通过 `kubectl expose` 可以快速创建 Service，并为 Deployment 部署的多个 Pod 暴露一个相同的端口。

由于 Pod 中的 nginx 访问端口是 80，我们想在外网中访问时使用 6666 端口，则命令如下：

```
kubectl expose deployment nginx --port=6666 --target-port=80  
# 指定源端口为 80, 要映射到 6666 端口
```

```
kubectl expose deployment nginx --port=80  
# 如果没有指定源端口, 则表示其端口也是 80
```

如果对象直接是 Pod 或者 DaemonSet 等对象, 则 `kubectl expose {对象} ...`, 会自动为此对象中的 Pod 创建端口映射。

查看 Service:

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	13d
nginx	ClusterIP	10.105.13.163	<none>	6666/TCP	6s

由于我们只是简单地创建 Service, 并没有指定 Service 的公开程度, 可以看到 TYPE 中, 其类型为 ClusterIP, 表示只能在 Pod 所在的节点上通过此 IP 和端口访问 Pod(如果安装了 CNI 网络插件, 则任意节点皆可访问)。

可以使用 10.105.13.163 和 6666 端口访问 Pod 中的 Nginx 服务。

这里介绍了如何创建 Service, 暴露端口, 而在后面的章节中, 会详细介绍 Service。

[Info] 提示

如果已经指定过 containerPort, 可以使用 `kubectl expose deployment nginx` 快速映射 containerPort 中指定的端口。

本地端口

每个 Pod 都有自己的唯一 IP 地址, Pod 所在的服务器上, 通过 IP 地址可以直接访问 Pod。

除了 Service, 我们还可以使用 `port-forward` 在服务器上直接映射本地端口到 Pod。

```
root@instance-2:~# kubectl port-forward nginx-55649fd747-s4824 666:80  
Forwarding from 127.0.0.1:666 -> 80  
Forwarding from [::1]:666 -> 80
```

此方式具有很大限制, 首先如果 Pod 在 instance-2 节点中, 那么此命令在 instance-2 节点上运行才能生效, 在其它节点上运行无效。

此方式只能映射本地端口, 如 127.0.0.1, 不能通过外网访问。

原本只能通过具体的 Pod IP 才能访问 Pod, 现在在服务器上通过 127.0.0.1 也可以直接访问服务。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-11-05 19:24:02

- 3.5 Pod 升级、回滚
 - 滚动更新和回滚
 - 部署应用
 - 更新版本
 - 上线
 - 如何滚动更新
 - 查看上线记录
 - 回滚
 - 暂停上线

3.5 Pod 升级、回滚

本篇主要讨论如何实现滚动更新和回滚，任意更换版本并且回滚以前的版本(版本更新)，而下一章会讨论到 Pod 缩放，根据机器资源自动拓展和收缩应用(自动扩容实例)。

滚动更新和回滚

部署应用

首先我们来部署 nginx，使用 nginx 作为练习的镜像。

打开 https://hub.docker.com/_/nginx 可以查询 nginx 的镜像版本，笔者这里选择三个版本：1.19.10、1.20.0、latest，后续我们更新和回滚时，会在这几个版本之间选择。

[Info] 提示

需要读者明确选择nginx 的三个不同版本，我们后面的升级回滚练习会在这三个版本中来回切换。

1.19.10 -> 1.20.0 -> latest

首先，我们创建一个 Nginx 的 Deployment，副本数量为 3，首次部署的时候，跟之前的操作一致，不需要什么特殊的命令。这里我们使用旧一些的版本，笔者使用的是 1.19.0。

```
kubectl create deployment nginx --image=nginx:1.19.0 --replicas=3
# 或者
# kubectl create deployment nginx --image=nginx:1.19.0 --replicas=3 --record
```

注：我们也可以加上 --record 标志将所执行的命令写入资源注解 kubernetes.io/change-cause 中。这对于以后的检查是有用的。例如，要查看针对每个 Deployment 修订版本所执行过的命令，对于这个参数的作用，我们后面再解释。

执行 `kubectl get pods`、`kubectl describe pods` 可以观察到有三个 Pod，每个 Pod 的 nginx 镜像版本都是 1.19.0。

NAME	READY	STATUS	RESTARTS	AGE
nginx-85b45874d9-7j1rv	1/1	Running	0	5s
nginx-85b45874d9-h22xv	1/1	Running	0	5s
nginx-85b45874d9-vthfb	1/1	Running	0	5s

Events:				
Type	Reason	Age	From	Message
Normal	Scheduled	119s	default-scheduler	Successfully assigned default/nginx-85b4
Normal	Pulled	117s	kubelet	Container image "nginx:1.19.0" already p
Normal	Created	117s	kubelet	Created container nginx
Normal	Started	117s	kubelet	Started container nginx

更新版本

其实更新 Pod 是非常简单的，我们不需要控制每个 Pod 的更新，也不需要担心会不会对业务产生影响，K8S 会自动控制这些过程。我们只需要触发镜像版本更新事件，K8S 会自动为我们更新所有 Pod 的。

`kubectl set image` 可以更新现有资源对象的容器镜像，对象包括 Pod、Deployment、DaemonSet、Job、ReplicaSet。在更新版本中，单个容器的 Pod，对于多个容器的 Pod 行为是差不多的，所以我们使用单容器 Pod 练习即可。

更新 Deployment 中的镜像版本，触发 Pod：

```
kubectl set image deployment nginx nginx=nginx:1.20.0
```

格式为：

```
kubectl set image deployment {deployment名称} {镜像名称}:={镜像名称}:{版本}
```

此命令可以任意修改 Pod 中的其中一个容器的版本，只要某个容器的镜像版本变化，整个 Pod 都会重新部署。如果镜像版本没有变化，即使是执行了 `kubectl set image`，也不会产生影响。

```
gantt dateFormat YYYY-MM-DD title Nginx image update excludes weekdays
2021-11-04 section Nginx Nginx 1.19.0 :done , 2021-11-05, 1d Nginx 1.20.0
:active, 2021-11-06, 1d
```

我们可以查看 Pod 的详细信息：

```
kubectl describe pods
```

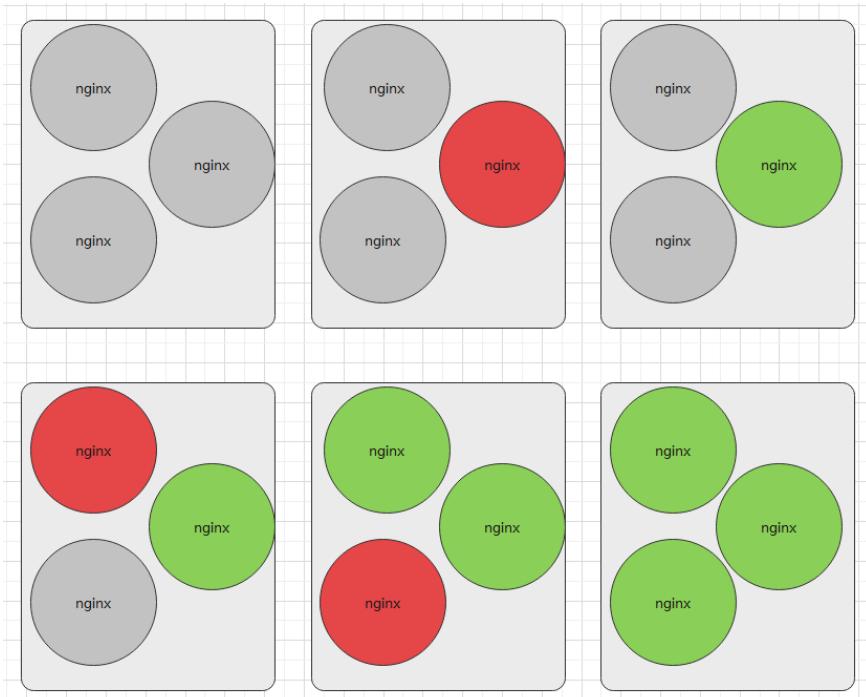
找到 Events 描述：

Type	Reason	Age	From	Message
Normal	Scheduled	66s	default-scheduler	Successfully assigned default/nginx-7b8755d55-5q7t5 to node-1
Normal	Pulled	66s	kubelet	Container image "nginx:1.20.0" already present on machine
Normal	Created	66s	kubelet	Created container nginx
Normal	Started	65s	kubelet	Started container nginx

可以看到，现在创建的 Pod 实例为 1.20.0 版本。

更新过程中，会创建新版本的 Pod，旧的 Pod 会被逐渐移除。

我们在创建 Deployment 时，生成了三个 Pod，而当我们触发镜像版本更新时，Pod 不会一次性更新，而是按照一定规则每次只重新部署一部分 Pod，Pod 更新替换过程类似下图所示(实际上 Pod 数量可能大于 3 个)：



另外，我们还可以通过 `kubectl edit yaml` 的方式更新 Pod。

执行：

```
kubectl edit deployment nginx
```

然后会弹出编辑 YAML 的界面，将 `.spec.template.spec.containers[0].image` 从 `nginx:1.19.0` 更改至 `nginx:1.20.0`，然后保存即可。

为了记录版本更新信息，我们需要在 `kubectl create deployment`、`kubectl set image` 命令后面加上 `--record`。

别忘记了 `kubectl scale` 命令也可以更改副本数量。

上线

仅当 Deployment Pod 模板（即 `.spec.template`）发生改变时，例如模板的标签或容器镜像被更新，才会触发 Deployment 上线。

其他更新（如对 Deployment 执行扩缩容的操作）不会触发上线动作，Deployment 的上线动作可以为我们更新 Pod 的版本（Pod 中的容器版本）。

这里提到的 上线/更新版本 是因为容器版本会发生变化，而更新一般指修改了 YAML 等，不一定会对容器产生影响。

当我们更新 Pod 版本时，K8S 会自动负载均衡，而不是把所有 pod 删除，再重新创建新版本 Pod，它会以稳健的方式逐渐替换 Pod 副本，所以叫滚动更新。

我们可以通过 `kubectl rollout status` 命令，查看 Pod 的上线状态，即 Pod 副本集的更替状态：

```
kubectl rollout status deployment nginx
```

输出结果一般有两种：

```
# 已经完成时:  
deployment "nginx-deployment" successfully rolled out  
# 还在更新时:  
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

我们也可以通过获取 Deployment 信息时，查看已更新的 pod 数量：

```
kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	18m

UP-TO-DATE 字段可以看到成功更新的 pod 数量。

还可以查看 ReplicaSet 和 Pod：

```
kubectl get replicaset  
kubectl get pods
```

输出类型于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-7b87485749	0	0	0	20m
nginx-85b45874d9	3	3	3	21m

NAME	READY	STATUS	RESTARTS	AGE
nginx-85b45874d9-nrbg8	1/1	Running	0	12m
nginx-85b45874d9-qc7f2	1/1	Running	0	12m
nginx-85b45874d9-t48vw	1/1	Running	0	12m

可以看到有两个 ReplicaSet，`nginx-7b87485749` 是 1.19.0 版本，已经被全部更新到 1.20.0 版本了，所以前者的数量为 0，我们也可以看到 Pod 中，所有 Pod 都是以 `nginx-85b45874d9` 作为前缀的。这几个关键信息，我们可以截图，后面再次对

照。

如何滚动更新

我们更新镜像版本时，旧的 Pod 版本会被替换，但是 ReplicaSet 副本记录是不会被删除的。实际上滚动更新就是控制副本数量，原本 1.19.0 的副本数量为 3，现在变成 0，1.20.0 的副本数量变成 3。

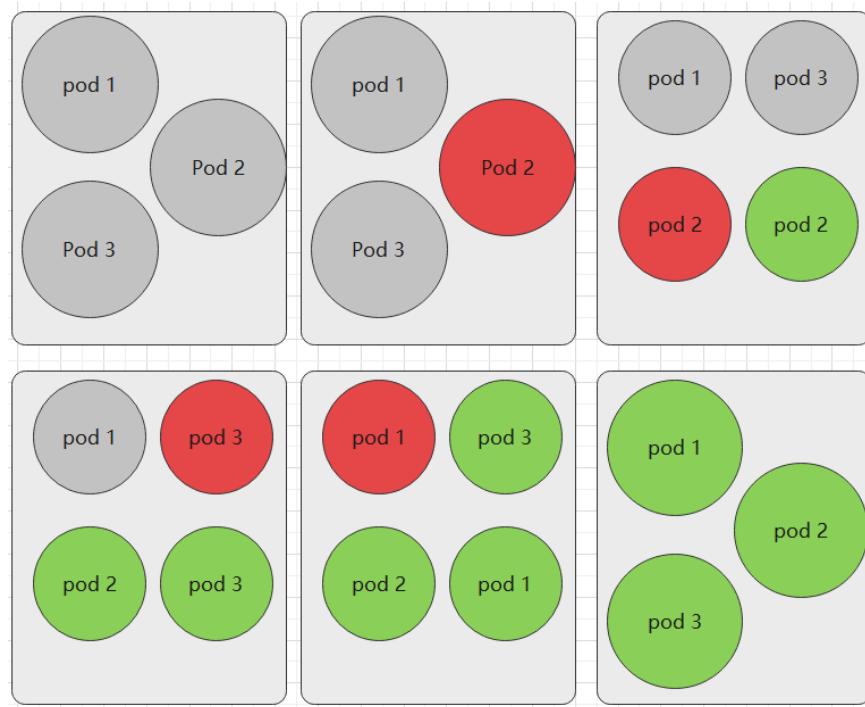
如果我们的项目上线了，我们更新软件版本，如果一次性更新所有容器或者 pod，那么我们的软件会有一段时间处于不可用状态，直到所有 Pod 都完成更新。

Deployment 可确保在更新时仅关闭一定数量的 Pod，默认情况下，它确保至少所需 Pods 75% 处于运行状态，也就是说正在被更新的 Pod 比例不超过 25%。当然，只有两三个 pod 的 Deployment 不会按照这个比例限定。也就是说，Deployment 等处于滚动更新状态时，其始终可以保证有可用的 Pod 提供服务。

如果我们的 Pod 数量足够大，或者在更新 Deployment 时迅速输出上线状态，可以看到新旧的 Pod 数量加起来不一定就是 3 个，因为它不会杀死老 Pods，直到有足够的数量新的 Pods 已经出现。在足够数量的旧 Pods 被杀死前并没有创建新 Pods。当副本数量为 3 个时，它确保至少 2 个 Pod 可用，同时最多总共 4 个 Pod 存在(不同版本)。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
nginx-705f645656-t9kdf	1/1	Running	0	41m	192.168.56.58	instance-2	<none>	<none>
nginx-ingress-6cd6d878f9-g9m0k	1/1	Running	0	15s	192.168.23.143	instance-1	<none>	<none>
nginx-ingress-6cd6d878f9-np2qz	1/1	Running	0	15s	192.168.23.142	instance-1	<none>	<none>
nginx-ingress-6cd6d878f9-vqfhd	1/1	Running	0	15s	192.168.23.144	instance-1	<none>	<none>

滚动更新过程如下图所示：



Deployment 确保仅所创建 Pod 数量只可能比期望 Pods 数高一点点。默认情况下，它可确保启动的 Pod 个数比期望个数最多多出 25%（最大峰值 25%）所以在自动更新 Deployment 时，观察到的 pod 可能为 4 个，这是由 Deployment 的缩放配置决定的(下一章讲解)。另外，在 Deployment 更新时，除了可以更改镜像的版本，也可以更改 ReplicaSet 的数量。

执行 `kubectl describe deployment nginx` 查看 Deployment 详细信息，我们查看 Event 字段，也可以观察到新旧 Pod 的更替过程。

OldReplicaSets: <none>	NewReplicaSet: nginx-85b45874d9 (3/3 replicas created)	Events:	Message
Normal ScalingReplicaSet 32m		deployment-controller	Scaled up replica set nginx-85b45874d9 to 3
Normal ScalingReplicaSet 32m		deployment-controller	Scaled up replica set nginx-7b87485749 to 1
Normal ScalingReplicaSet 32m		deployment-controller	Scaled down replica set nginx-85b45874d9 to 2
Normal ScalingReplicaSet 32m		deployment-controller	Scaled up replica set nginx-7b87485749 to 2
Normal ScalingReplicaSet 32m		deployment-controller	Scaled down replica set nginx-7b87485749 to 1
Normal ScalingReplicaSet 32m		deployment-controller	Scaled up replica set nginx-7b87485749 to 3
Normal ScalingReplicaSet 23m		deployment-controller	Scaled down replica set nginx-85b45874d9 to 0
Normal ScalingReplicaSet 23m		deployment-controller	Scaled up replica set nginx-85b45874d9 to 1
Normal ScalingReplicaSet 22m (x4 over 22m)		deployment-controller	Scaled down replica set nginx-7b87485749 to 0 combined from similar events: Scaled down replica set nginx-7b87485749 to 0

但是这些原理等知识我们都不需要记，也不需要深入，我们记得有这回事就行，有需要的时候也可以直接查看文档的，后面的章节还会详细介绍 ReplicaSet 的规则。

查看上线记录

默认情况下，Deployment 的上线记录都会保留在系统中，以便可以随时回滚，前面我们也提到了查看 `kubectl get replicsets` 时出现的副本记录。

我们查看 Deployment 的上线历史记录：

```
kubectl rollout history deployment nginx
```



```
REVISION  CHANGE-CAUSE
1          <none>
2          <none>

# 带 --record 的话，输出是
REVISION  CHANGE-CAUSE
2          kubectl set image deployment nginx nginx=nginx:1.20.0 --record=true
```

可以看到有两个版本，但是 CHANGE-CAUSE 为 `<none>` 呢？这是因为笔者没有使用 `--record` 参数记录信息，如果没带上 `--record` 的话，我们看着这个历史记录，完全分不出到底是什么版本。

现在我们查看 版本2 的详细信息：

```
kubectl rollout history deployment nginx --revision=2
```

```
deployment.apps/nginx with revision #2
Pod Template:
  Labels:    app=nginx
  pod-template-hash=85b45874d9
  Containers:
    nginx:
      Image:    nginx:1.20.0
      Port:     <none>
      Host Port: <none>
      Environment: <none>
      Mounts:    <none>
      Volumes:   <none>
```

回滚

当部署的新版本程序发现严重 bug 影响平台稳定性时，你可能需要将项目切换为上一个版本。目前介绍了几个查看 Deployment 上线的历史记录的命令，下面介绍如果将 Pod 换到旧的版本。

回滚到上一个版本的命令：

```
root@master:~# kubectl rollout undo deployment nginx  
deployment.apps/nginx rolled back
```

例如当前是 版本2，那么会回滚到 版本1。

再执行 `kubectl rollout history deployment nginx` 会发现 revision 变成 3 了。

```
root@master:~# kubectl rollout history deployment nginx  
deployment.apps/nginx  
REVISION  CHANGE-CAUSE  
2          <none>  
3          <none>
```

revision 记录的是部署记录，与 Pod 的镜像版本无关，每次更新版本或进行回滚等操作时，revision 会自动递增 1。

如果版本数量多了，我们还可以指定回滚到特定的版本。

```
kubectl rollout undo deployment nginx --to-revision=2
```

这里提一下 `--record`，在前面，我们创建和更新 Deployment 时，都没有使用到这个参数，其实这个参数很有用的，接下来我们每次执行滚动更新时都要带上这个参数才行。

更新镜像到指定版本：

```
kubectl set image deployment nginx nginx=nginx:1.19.0 --record
```

```
kubectl rollout history deployment nginx
```

输出：

```
REVISION  CHANGE-CAUSE  
5          <none>  
6          kubectl set image deployment nginx nginx=nginx:1.19.0 --record=true
```

但是我们这里目前来说，只有两个记录，我们明明提交了多次，虽然 revision 会变化，但是这里查询的只有两条记录，这时因为我们操作的时候，只用到了 1.19.0、1.20.0 两个版本，所以也就只有这两个版本的提交记录。多用几个版本，输出结果：

```
REVISION  CHANGE-CAUSE  
7          kubectl set image deployment nginx nginx=nginx:1.19.0 --record=true  
8          kubectl set image deployment nginx nginx=nginx:1.20.0 --record=true  
9          kubectl set image deployment nginx nginx=nginx:latest --record=true
```

REVISION 字段的数字是会递增的，当我们触发上线动作(容器标签、版本等)时，会产生新的上线记录。

暂停上线

本小节需要水平缩放、比例缩放等知识，请先阅读 3.6 章关于缩放的内容。

如果在上线过程中，发现机器不够用了，或者需要调整一些配置等，可以暂停上线过程。

`kubectl rollout pause` 命令可以让我们在 Deployment 的 Pod 版本时，暂停滚动更新。

命令：

```
kubectl rollout pause deployment nginx
```

在滚动更新过程中，会有一些现象需要我们留意。

先创建一个 Deployment 或者更新 Deployment 的 Pod 为 10 个副本。

```
kubectl create deployment --image=nginx:1.19.0 --replicas=10
```

我们执行 `kubectl edit deployment nginx` 修改缩放个数：

```
strategy:  
  rollingUpdate:  
    maxSurge: 3  
    maxUnavailable: 2  
  type: RollingUpdate
```

设置了这个 `maxSurge` 和 `maxUnavailable`，可以让 Deployment 替换 Pod 时慢一些。

之前我们已经使用了 `1.19.0`、`1.20.0` 两个版本进行演示，这里我们使用 `latest` 版本进行实践。

复制以下两条命令快速执行，可以快速卡住上线过程。我们暂停上线后，查看一些状态信息。

```
kubectl set image deployment nginx nginx=nginx:latest  
kubectl rollout pause deployment nginx
```

执行 `kubectl get replicaset` 查看这些版本的数量。

NAME	DESIRED	CURRENT	READY	AGE
nginx-7b87485749	8	8	8	109m
nginx-85b45874d9	0	0	0	109m
nginx-bb957bbb5	5	5	5	52m

可以看到，所有的 Pod 加起来数量大于 10，旧容器以每次 2 个的数量减少；新容器以每次 3 个的数量创建；暂停上线后，多次执行 `kubectl get replicaset`，会发现副本数量不会变化。

前面我们已经暂停了上线，如果我们执行上线命令换成别的版本：

```
kubectl set image deployment nginx nginx=nginx:1.19.0
```

会发现虽然提示更新了，但是实际上没有变化。执行 `kubectl rollout history deployment nginx` 也查不到我们提交的 `1.19.0` 的请求。这是因为在已经暂停上线的控制器对象中，执行新的上线动作是无效的。

暂停的时候，我们可以更新一些配置，例如限制 Pod 中的 nginx 容器使用的 CPU 和资源：

```
kubectl set resources deployment nginx -c=nginx --limits=cpu=200m,memory=512Mi
```

再恢复 Deployment 上线：

```
kubectl rollout resume deployment nginx
```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-11-05 20:27:38

- 3.6 Pod 缩放
 - 缩放 Deployment
 - 设置副本数量
 - 水平自动缩放
 - 比例缩放

3.6 Pod 缩放

在前面我们已经学习到了 Pod 的扩容、滚动更新等知识，我们可以手动为 Deployment 等设置 Pod 副本的数量，而这里会继续学习关于 Pod 扩容、收缩的规则，让 Pod 根据节点服务器的资源自动增加或减少 Pod 数量。

缩放 Deployment

设置副本数量

很简单，使用 `kubectl scale` 命令直接设置：

```
kubectl scale deployment nginx --replicas=10
```

其它方式前面的章节已经提到过了，还有通过修改 YAML 文件的方式。

水平自动缩放

K8S 有个 Pod 水平自动扩缩（Horizontal Pod Autoscaler）可以基于 CPU 利用率自动扩缩 ReplicationController、Deployment、ReplicaSet 和 StatefulSet 中的 Pod 数量。Pod 自动扩缩不适用于无法扩缩的对象，比如 DaemonSet。

除了 CPU 利用率，也可以基于其他应用程序提供的自定义度量指标来执行自动扩缩。

参考资料：<https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale/>

命令：

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

表示目标 CPU 使用率为 80% (期望指标)，副本数量配置应该为 10 到 15 之间，CPU 是动态缩放 pod 的指标，会根据具体的 CPU 使用率计算副本数量，其计算公式如下。

```
期望副本数 = ceil[当前副本数 * (当前指标 / 期望指标)]
```

因为笔者这里只有一个 Worker 节点，不能控制 CPU 使用率模拟场景，所以不方便演示，读者只需要了解这个命令即可。

按照算法计算，加入当前副本数量为 12，且 CPU 使用率达到 90%，则期望副本数为 $12 * (90\% / 80\%) = 13.5$ ，那么理论上会部署 14 个 Pod，但是 CPU 再继续增加的话，最多 15 个副本数量。如果在机器管够的情况下，可以去掉 `min` 和 `max` 参数。

算法细节请查看：<https://kubernetes.io/zh/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details>

比例缩放

比例缩放指的是在上线 Deployment 时，临时运行着应用程序的多个版本(共存)，比例缩放是控制上线时多个 Pod 服务可用数量的方式。

水平缩放只关心最终的期望 Pod 数量，直接修改副本数和水平缩放，决定最终 Pod 数量有多少个。

而比例缩放是控制对象上线过程中，新的 Pod 创建速度和旧的 Pod 销毁速度、Pod 的可用程度，跟上线过程中新旧版本的 Pod 替换数量有关。

查看上一章中创建的 Deployment 的部分 YAML 如下：

```
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

`strategy` 可以配置 Pod 是怎么更新的。

当我们设置 `.spec.strategy.type==RollingUpdate` 时，便会采取滚动更新的方式更新 Pods，此时可以指定 `maxUnavailable` 和 `maxSurge` 来控制滚动更新过程。这个我们之前提到过，就是 Deployment 默认会保证一直有 75% 的 pod 处于可用状态，在完成更新前可能有多个版本的 pod 共存。

- `maxUnavailable`

最大不可用数量或比例，旧的 Pod 会以这个数量或比例逐渐减少。

- `maxSurge`

最大峰值，新的 Pod 会按照这个数量或比例逐渐创建。

3.5 章已经使用到了这两者，这里就不细说了，读者请参考：

<https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#max-unavailable>

我们查看之前的 Deployment，执行命令 `kubectl get deployment nginx -o yaml`：

```
... ...
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
...
...
```

配置表示，每次只有 1/4 的 Pod 被更新、替换。

这个是所有 Deployment 的默认配置，在更新镜像版本时，旧的 Pod 会被新的 Pod 替换，但是不是一下子完成的，每次处理 25% 的 Pod，在更新过程中，我们必须保证我们的服务依然可用，即还有旧版本的 Pod 在运行。这个配置设定了更新过程中至少保证 75% 的 Pod 还可以使用，这个就是比例缩放。

下面我们来进行实验。

首先创建新的 Deployment，设置副本数量为 10：

```
kubectl create deployment nginx --image=nginx:1.19.0 --replicas=10
# kubectl scale deployment nginx --replicas=10
```

我们执行 `kubectl edit deployment nginx` 修改缩放个数：

```
strategy:
  rollingUpdate:
    maxSurge: 3
    maxUnavailable: 2
  type: RollingUpdate
```

除了可用百分比表示，也可以使用个数表示。

旧的 Pod 按照最大 2 个的速度不断减少；新的 Pod 按照最大 3 个的速度不断增加；

比例缩放的配置处理好了，它会在我们上线新版本的时候生效，我们可以观察到这个过程，但是需要快一点执行命令查看状态。

快速执行以下命令：

```
kubectl set image deployment nginx nginx=nginx:1.20.0
kubectl get replicaset
```

```
root@instance-1:~# kubectl set image deployment nginx nginx=nginx:1.20.0
deployment.apps/nginx image updated
root@instance-1:~# kubectl get replicaset
NAME          DESIRED   CURRENT   READY   AGE
nginx-7b87485749   5         5         0      93m
nginx-85b45874d9   0         0         0      93m
nginx-bb957bbbb5   8         8         8      35m
```

因为允许新的 Pod 创建较快(3个)，所以最终可能新的 Pod 数量达到 10 个了，旧的 Pod 还有很多，总数量大于 10。

最终：

NAME	DESIRED	CURRENT	READY	AGE
nginx-7b87485749	10	10	10	99m
nginx-85b45874d9	0	0	0	99m
nginx-bb957bbb5	0	0	0	41m

如果想新版本的 Pod 上线速度更快，则可以把 `maxSurge` 数量或比例设置大一些；为了保证上线过程稳定、服务可用程度高，可以把 `maxUnavailable` 设置小一些。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-05 20:51:15

- 3.8 Pod 调度
 - 亲和性
 - 亲和性和反亲和性
 - 污点和容忍度
 - 污点
 - 系统默认污点
 - 容忍度

3.8 Pod 调度

在上一章中，我们学习了 `Label`，可以为对象设置标签，并且使用选择器筛选对象，本章中我们将学习节点调度的相关知识，在标签之上控制 Pod 在节点中的部署。

亲和性

亲和性和反亲和性

前面我们学习了 `nodeSelector`，`Deployment` 等可以使用 `nodeSelector` 选择合适的节点部署 Pod。

例如选择磁盘空间大的节点部署 Pod：

```
spec:  
  containers:  
    - image: nginx:latest  
      name: nginx  
    nodeSelector:  
      disksize: "big"
```

而节点亲和性类似于 `nodeSelector`，可以根据节点上的标签约束 Pod 可以调度到哪些节点。

Pod 亲和性有两种别为：

- `requiredDuringSchedulingIgnoredDuringExecution`

硬需求，将 Pod 调度到一个节点必须满足的规则。

- `preferredDuringSchedulingIgnoredDuringExecution`。

尽量满足，尝试执行但是不能保证偏好。

这是从 Kubernetes 官方文档找到的一个例子：

```

apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
                  - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: another-node-label-key
                operator: In
                values:
                  - another-node-label-value
  containers:
    - name: with-node-affinity
      image: k8s.gcr.io/pause:2.0

```

这个 YAML 的层级太多，关于亲和性部分的内容抽选如下：

```

affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
          ...
    preferredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
          ...

```

`affinity` 配置亲密关系；亲和性有多种类型，示例中使用了 `nodeAffinity` 设置节点亲密关系，最后是亲和性表达式，它们表示必须满足(`requiredDuring`)和尽量满足(`preferredDuring`)。

`requiredDuring` 部分的规则定义如下：

```

requiredDuringSchedulingIgnoredDuringExecution:
  nodeSelectorTerms:
    - matchExpressions:
        - key: kubernetes.io/e2e-az-name
          operator: In
          values:
            - e2e-az1
            - e2e-az2

```

`requiredDuringSchedulingIgnoredDuringExecution` 亲和性的约束如果使用标签选择器来表示，相当于：

```
... ... -l kubernetes.io/e2e-az-name in (e2e-az1,e2e-az2)
```

如果我们设置了多个 `nodeSelectorTerms` :

```
requiredDuringSchedulingIgnoredDuringExecution:  
  nodeSelectorTerms:  
  ...  
  nodeSelectorTerms:
```

则只需要满足其中一种表达式即可调度 **Pod** 到 节点上。

我们再回忆一下，节点选择器叫 `nodeSelector`，而节点亲和性叫 `nodeAffinity`，它们都可以让 `Deployment` 等对象部署 `Pod` 时选择合适的节点，它们都是使用标签 (`Label`) 来完成选择工作。

如果你同时指定了 `nodeSelector` 和 `nodeAffinity`，则两者必须同时满足条件，才能将 `Pod` 调度到候选节点上。

除了节点亲和性，也有 `Pod` 亲和性。

节点亲和性语法支持下面的操作符：

`In` , `NotIn` , `Exists` , `DoesNotExist` , `Gt` , `Lt` 。

`Pod` 亲和性与反亲和性的合法操作符有 `In` , `NotIn` , `Exists` , `DoesNotExist` 。

通过 `Affinity` 后缀可以设置亲和性，例如节点亲和性 `nodeAffinity`。而设置反亲和性使用 `AntiAffinity` 后缀，例如 `nodeAntiAffinity`。

反亲和性跟亲和性一样，都有 `requiredDuringSchedulingIgnoredDuringExecution` 硬限制和 `preferredDuringSchedulingIgnoredDuringExecution` 软限制，只是反亲和性是相反的表示，如果符合条件则不能调度。

关于亲和性和反亲和性的说明就到这里，这两者的配置比较多和复杂，读者可以参考官方文档，这里不再赘述。

亲和性和反亲和性的 YAML 很复杂，所以手写不出来的，只需要努力了解大概的意思和看懂就行，需要使用时查看文档。

<https://kubernetes.io/zh/docs/concepts/scheduling-eviction/assign-pod-node>

污点和容忍度

污点

前面提到亲和性和反亲和性，我们可以让 `Pod` 选择合适的 `Node`(节点亲和性)，或者 `service`、`Deployment` 选择合适的 `Pod`(`Pod`亲和性)，这些拥有 `Label` 的对象都是被选择的。

这里，我们介绍污点和容忍度，`Node` 和 `Pod` 它们可以排斥“被选择”的命运。

[Info] 提示

节点污点：当节点添加一个污点后，除非 Pod 声明能够容忍这个污点，否则 Pod 不会被调度到这个节点上。

也就是说，节点也可以“丑拒”Pod。

如果节点存在污点，那么 Pod 可能不会被分配到此节点上；如果节点一开始没有设置污点，然后部署了 Pod，后面节点设置了污点，节点可能会删除已部署的 Pod，这种行为称为驱逐。

节点污点(taint)可以排斥一类特定的 Pod，而 容忍度(Tolerations)则表示能够容忍这个对象的污点。

节点说自己又笨又丑，但是 Pod 说我不介意，要跟你谈一场轰轰烈烈的恋爱。

```
Node - taint > 排斥 Pod  
Pod - tolerations -> 我是真爱，我能容忍污点 -> Node
```

污点有强制和尽量两种，前者完全排斥，后者尽可能排斥，另外某些污点可以将已经在这台节点上部署的 Pod 逐出，这个过程称为 effect。

系统会 尽量避免将 Pod 调度到在其不能容忍污点的节点上，但这不是强制的。Kubernetes 处理多个污点和容忍度的过程就像一个过滤器：从一个节点的所有污点开始遍历，过滤掉那些 Pod 中存在与之相匹配的容忍度的污点。

但是如果你只有一个 Worker，那么设置了污点，那 Pod 也只能选择在这个节点上运行。就好像全世界只剩下女(Node)一男(Pod)，不想在一起也得在一起。

添加污点格式：

```
kubectl taint node [node] key=value:[effect]
```

更新污点或覆盖：

```
kubectl taint node [node] key=value:[effect] --overwrite=true
```

我们也可以修改节点的 YAML 文件，修改污点：

```
apiVersion: v1  
kind: Node  
metadata:  
...  
spec:  
  taints:  
  - effect: NoSchedule  
    key: node-role.kubernetes.io/master  
...
```

我们来实际尝试命令，使用 `kubectl taint` 给节点增加一个污点。

```
kubectl taint node instance-2 key1=value1:NoSchedule
```

移除污点：

```
kubectl taint node instance-2 key1=value1:NoSchedule-
```

[Error] 提示

污点生成的键值不是标签，污点生成的是 `taint` 对象，标签生成的是 `label` 对象。

给节点设置的污点，不会在 `Labels` 中出现，而是在 `Taints` 中。

查看节点的污点：

```
kubectl get nodes instance-2 -o json | jq '.items[].spec.taints'
```

使用 `apt install jq` 安装 `json` 筛选工具。

```
[  
 {  
   "effect": "NoSchedule",  
   "key": "key1",  
   "value": "value1"  
 }
```

或者从 `Node` 描述文件中获取污点信息：

```
root@master:~# kubectl describe nodes slave1 | grep Taints  
Taints:           key1=value1:NoSchedule
```

污点的效果称为 `effect`，我们在前面设置了 `{key1,value1}`，其默认效果为 `NoSchedule`。

节点的污点可以设置为以下三种效果：

- `NoSchedule`：不能容忍此污点的 `Pod` 不会被调度到节点上；不会影响已存在的 `pod`。
- `PreferNoSchedule`：`Kubernetes` 会避免将不能容忍此污点的 `Pod` 安排到节点上。
- `NoExecute`：如果 `Pod` 已在节点上运行，则会将该 `Pod` 从节点中逐出；如果尚未在节点上运行，则不会将其调度到此节点上。

当节点设置污点后，无论其效果是哪一种，只要 `Pod` 没有设置相关的容忍度，`Pod` 就不会调度到此节点上。

例如节点声明了 `smallcpu` 污点，只要 `Pod` 没有声明容忍此污点，那么 `Pod` 就不应该被调用到此节点上。除了 `smallcpu` 污点名称外，还有值也属于规则约束，这点我们后面再解释。

系统默认污点

尽管一个节点上的污点完全排斥 Pod，但是某些系统创建的 Pod 可以容忍所有 `NoExecute` 和 `NoSchedule` 污点，因此不会被逐出。

例如 `master` 节点是不会被 Deployment 等分配 Pod 的，因为 `master` 有个污点，表面它只应该运行 `kube-system` 命名空间中的很多系统 Pod，用户 Pod 会被排斥部署到 `master` 节点上。

```
root@master:~# kubectl describe nodes master | grep Taints
Taints:           node-role.kubernetes.io/master:NoSchedule
```

当然我们通过修改污点，可以让用户的 Pod 部署到 `master` 节点中，本小节我们将学习系统默认污点以及相关驱逐的知识。

查询所有节点的污点：

```
kubectl describe nodes | grep Taints
```

```
# master 输出
Taints:           node-role.kubernetes.io/master:NoSchedule
# worker 输出
Taints:           key1=value1:NoSchedule
```

`master` 节点上会有一个 `node-role.kubernetes.io/master:NoSchedule` 的污点，Kubernetes 部署用户的 Pod 时会检查节点是否存在此污点，如果有，则不会在此节点上部署 Pod。

这里我们做一个实践，去掉 `master` 节点的污点，使得用户 Pod 能够在此节点上部署。

我们去除 `master` 和 `worker` 的污点：

```
kubectl taint node instance-1 node-role.kubernetes.io/master:NoSchedule-
kubectl taint node instance-2 key1=value1:NoSchedule-
# instance-1 是笔者的 master 节点名称
# 有污点就删除，没有就不用管
```

然后部署通过 Deployment 部署 Nginx，并设置副本集。

```
kubectl create deployment nginxtaint --image=nginx:latest --replicas=5
```

查看 Pod 都部署到哪些节点上：

```
kubectl get pods -o wide
```

可以看到，`master` 节点也被部署了 Pod。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
nginxtaint-6c6dd0878f9-28vkz	0/1	ContainerCreating	0	5s	<none>	instance-2	<none>	<none>
nginxtaint-6c6dd0878f9-2k8lq	1/1	Terminating	0	3m	192.168.23.136	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-56fdn	0/1	Terminating	0	3m	192.168.23.143	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-6694n	0/1	ContainerCreating	0	5s	<none>	instance-2	<none>	<none>
nginxtaint-6c6dd0878f9-dxj7p	1/1	Terminating	0	3m	192.168.23.138	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-ffskk	0/1	ContainerCreating	0	5s	<none>	instance-2	<none>	<none>
nginxtaint-6c6dd0878f9-g9v9d	0/1	ContainerCreating	0	5s	<none>	instance-2	<none>	<none>
nginxtaint-6c6dd0878f9-ljz98	0/1	Terminating	0	3m	192.168.23.142	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-m4bbg	1/1	Terminating	0	3m	192.168.23.144	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-mrvmr	1/1	Terminating	0	3m	192.168.23.135	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-pcn9q	0/1	Terminating	0	3m	192.168.23.141	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-pjf94	0/1	Terminating	0	3m	192.168.23.137	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-q2x79	0/1	ContainerCreating	0	5s	<none>	instance-2	<none>	<none>
nginxtaint-6c6dd0878f9-rbspu	1/1	Terminating	0	3m	192.168.23.139	instance-1	<none>	<none>
nginxtaint-6c6dd0878f9-tcsvl	1/1	Terminating	0	3m	192.168.23.140	instance-1	<none>	<none>

可以看到，Pod 在笔者的主节点上运行了(instance-1)。

这里只是练手实践，为了保证集群安全，我们需要恢复 master 的污点，驱逐用户 Pod。

```
kubectl taint node instance-1 node-role.kubernetes.io/master:NoSchedule
```

除了上面提及的 `node-role.kubernetes.io/master`，Kubernetes 还会在某些情况下，某种条件为真时，节点控制器会自动给节点添加一个污点。当前内置的污点包括：

- `node.kubernetes.io/not-ready`：节点未准备好。这相当于节点状态 `Ready` 的值为 "False"。
- `node.kubernetes.io/unreachable`：节点控制器访问不到节点。这相当于节点状态 `Ready` 的值为 "Unknown"。
- `node.kubernetes.io/out-of-disk`：节点磁盘耗尽。
- `node.kubernetes.io/memory-pressure`：节点存在内存压力。
- `node.kubernetes.io/disk-pressure`：节点存在磁盘压力。
- `node.kubernetes.io/network-unavailable`：节点网络不可用。
- `node.kubernetes.io/unschedulable`：节点不可调度。
- `node.cloudprovider.kubernetes.io/uninitialized`：如果 `kubelet` 启动时指定了一个“外部”云平台驱动，它将给当前节点添加一个污点将其标志为不可用。在 `cloud-controller-manager` 的一个控制器初始化这个节点后，`kubelet` 将删除这个污点。

当节点上的资源不足时，会添加一个污点，排斥后续 Pod 在此节点上部署，但不会驱逐已存在的 Pod。如果我们的 Pod 对机器资源有要求，可以排斥相关的污点，如果没要求，则需要容忍相关污点。

容忍度

污点和容忍度相互配合，用来避免 Pod 被分配到不合适的节点上；也可以让真正合适的 Pod 部署到有污点的节点上。一个节点可以设置污点，排斥 Pod，但是 Pod 也可以设置容忍度，容忍节点的污点。

YAML 示例：

```
tolerations:  
- key: "key1"  
  operator: "Exists"  
  effect: "NoSchedule"
```

此 Pod 能够容忍带有 `key1` 标签的污点，且无论是什么值。

也可以设置带 `value` 的容忍。

```
tolerations:  
- key: "key1"  
  operator: "Equal"  
  value: "value1"  
  effect: "NoSchedule"
```

```
operator 的默认值是 Equal。
```

只有当污点 key1 的值为 value1 时，才容忍。

可以同时定义多个容忍度：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

一个容忍度和一个污点相“匹配”是指它们有一样的键名(key)和效果(effect)，并且：

- 如果 operator 是 Exists

此时不需要填写 value 字段；如果存在 key 为 key1 的 label，且污点效果为 NoSchedule，无论是什么值都容忍。

- 如果 operator 是 Equal

则它们的 value 应该相等，如果相同的话，则容忍，如果只不同则容忍。

- 如果 effect 留空

则表示只要是 label 为 key1 的节点，都可以容忍。

如果 Pod 的容忍度设置为以下 YAML：

```
tolerations:
  operator: "Exists"
```

则表示此 Pod 能够容忍任意的污点，无论节点怎么设置 key、value、effect，此 Pod 都不会介意。

如果要在 master 上也能部署 Pod，则可以修改 Pod 的容忍度：

```
spec:
  tolerations:
    # this toleration is to have the daemonset runnable on master nodes
    # remove it if your masters can't run pods
    - key: node-role.kubernetes.io/master
      effect: NoSchedule
```

下面是一份官方的 Pod 的带有容忍度的 YAML 文件模板：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "example-key"
    operator: "Exists"
    effect: "NoSchedule"
```

模板文件地址:

<https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/pod-with-toleration.yaml>

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook文档最后更新
时间: 2021-11-06 20:20:32

- 3.9 Jobs、CronJobs
 - Job
 - 控制器异同点
 - Job
 - 完成数
 - 并行
 - 带工作队列的 Job
 - 带类型的 Job
 - 处理 Pod 和容器失效
 - Job 终止和清理
 - CronJob

3.9 Jobs、CronJobs

在 Kubernetes 中，常用的工作负载/控制器，有 Deployment、ReplicaSet、StatefulSet、DaemonSet、Job、Cronjob、ReplicationController 等，在前面，我们已经学习了 Deployment、DaemonSet、ReplicationController，本章将学习 Job 和 Cronjob。

我们知道，Pod 是一种临时性的对象，我们单独创建 Pod，一般用于临时测试调试，在生产中我们使用 Deployment 等对 Pod 进行管理。而 Job、Cronjob 它们主要用于创建一个或多个 Pod，来完成某些任务，它们创建的 Pod 不会长久的运行在节点中。

Job

Job 是用来只运行一次任务的对象，Job 对象以一种可靠的方式运行某 Pod 直到完成，适合用于批处理，例如编译程序、执行运算任务。Job 适合一次到位或者一次完整的流程，完成后即可抛弃的任务。

控制器异同点

前面我们已经学习到了 Deployment、ReplicaSet，Job 跟它们一些异同点，这里列举一些它们各自的特点。

- Pod 生命期

和一个个独立的应用容器一样，Pod 也被认为是相对临时性（而不是长期存在）的实体。Pod 会被创建、赋予一个唯一的 ID，并被调度到节点，并在终止（根据重启策略）或删除之前一直运行在该节点。

- Pod 是临时的

Pod 自身不具有自愈能力。如果 Pod 被调度到某节点而该节点之后失效，或者调度操作本身失效，Pod 会被删除；与此类似，Pod 无法在节点资源耗尽或者节点维护期间继续存活。Deployment、ReplicaSet、Job 都会重新创建新的 Pod 来替代已终止的 Pod。

- Deployment/ReplicaSet、Job

Deployment/ReplicaSet 控制器管理的是那些不希望被终止的 Pod (例如, Web 服务器), **Job** 管理的是那些希望被预期终止的 Pod(例如, 批处理作业)。

- 为什么不用 Pod

Pod 是临时性的, 有 Always、OnFailure、Never 等选项, 设置在容器无法启动时的重试机制; 而 Job 只有 OnFailure、Never , Job 最多只能重试一次。Pod 在自身没有保障, 挂了就挂了, 而且 Pod 本身只有一个。Job 会创建一个或者多个 Pods, 在失败时将继续重试 Pods 的执行, 或者创建新的 Pod, 直到指定数量的 Pods 成功终止。

Job

当 Job 启动时, Job 会跟踪成功完成的 Pod 的个数, 当成功数量达到某个阈值时, Job 会被终结。当 Job 运行过程中, 我们暂停/挂起 Job, Job 会删除正在运行的 Pod, 保留已完成的 Pod 数量, 当恢复 Job 时, 会创建新的 Pod, 继续完成任务。

Job 的结构很简单, 下面是一个示例, 这个 Job 只有一个镜像, 启动后执行 sleep 3 命令, 容器会在3秒后自动退出, Job 标记其已经完成。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox
spec:
  template:
    spec:
      containers:
        - name: busybox
          image: busybox
          command: ["/bin/sleep"]
          args: ["3"]
      restartPolicy: Never
```

对于一个简单的 Job, 其关键在于设定一个可结束的命令, 容器执行命令后会退出。

此时这个容器被标记为完成状态。

```
command: ["/bin/sleep"]
args: ["3"]
restartPolicy: Never
```

restartPolicy 需要标记为 Never 或 OnFailure。

使用 kubectl apply -f job.yaml 命令启动此 Job, 3 秒后查看 Pod:

```
root@instance-1:~# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
busybox-k4wkn  0/1     Completed  0          16s
```

可以发现此 Pod 处于完成状态(Completed), 再查看 job 列表:

```
root@instance-1:~# kubectl get jobs
NAME      COMPLETIONS  DURATION   AGE
busybox   1/1          11s        101s
```

完成后我们可以直接删除它：

```
kubectl delete job busybox
```

对于这种简单的 Job，称为非并行的，它的特点有：

- 只启动一个 Pod，除非该 Pod 失败；
- 当 Pod 成功终止时，立即视 Job 为完成状态；

这里再列举一个 Job 示例，这个 Job 用于计算圆周率，计算完毕后打印圆周率并结束容器。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: Never
  backoffLimit: 4
```

查看 Pod 列表，找到 pi- 开头的 Pod，查看日志：

```
root@instance-1:~# kubectl logs pi-7htxl
3.1415926535897932384626433... ...
```

.spec.backoffLimit 可视为 Job 在失败之前的重试次数。

如果我们设置了 restartPolicy: OnFailure，那么此 Pod 会被重试一次；

如果 restartPolicy: Never，并且设置了 backoffLimit，Pod 会被重试多次。

完成数

使用 .spec.completions 来设置完成数时，Job 控制器所创建的每个 Pod 使用完全相同的 spec 模板。这意味着任务的所有 Pod 都有相同的命令行，都使用相同的镜像和数据卷，甚至连环境变量都（几乎）相同。

如下 YAML，template 中的 spec，是每个 Pod 都带有的相关属性，使得每个 Pod 都保持一致。

```
spec:
  completions: 5
  template:
    spec:
```

Job 会创建一个或者多个 **Pods**，并将继续重试 **Pods** 的执行，直到指定数量的 **Pods** 成功终止。随着 **Pods** 成功结束，**Job** 跟踪记录成功完成的 **Pods** 个数。当数量达到指定的成功个数阈值时，任务（即 **Job**）结束。删除 **Job** 的操作会清除所创建的全部 **Pods**。

我们继续使用上次的 **Job** 模板，这里增加一个 `completions`，完整 **YAML** 内容如下：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox
spec:
  completions: 5
  template:
    spec:
      containers:
        - name: busybox
          image: busybox
          command: ["/bin/sleep"]
          args: ["3"]
      restartPolicy: Never
```

使用 `kubectl apply -f job.yaml` 启动此 **Job**。

查看 **Job** 和 **Pod**：

```
root@instance-1:~# kubectl get jobs
NAME      COMPLETIONS   DURATION   AGE
busybox   5/5           36s        38s
root@instance-2:~# kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
busybox-rfhcj         0/1     Completed  0          9s
busybox-stkbg         0/1     Completed  0          23s
busybox-xk6sb         0/1     Completed  0          30s
busybox-z6h9x         0/1     Completed  0          40s
busybox-zqgcb         0/1     Completed  0          16s
```

Pod 的创建是串行的，每次只运行一个 Pod，当一个 Pod 处于 `Completed` 状态时，创建下一个 Pod。当有 5 个 Pod 处于 `Completed` 状态时，此 **Job** 标记完成。

前面学习到了 `completions`，串行启动 Pod，直到 Pod 完成数量为 5。

如果经常查看 **Job** 的状态，可以观察到：

```

NAME      COMPLETIONS  DURATION  AGE
busybox  1/5          30s       10s

NAME      COMPLETIONS  DURATION  AGE
busybox  2/5          28s       18s

NAME      COMPLETIONS  DURATION  AGE
busybox  3/5          28s       25s

root@instance-1:~# kubectl get jobs
NAME      COMPLETIONS  DURATION  AGE
busybox  4/5          30s       30s

root@instance-1:~# kubectl get jobs
NAME      COMPLETIONS  DURATION  AGE
busybox  5/5          36s       37s

```

flowchart LR Start --> busybox1 --> busybox2 --> busybox3 --> busybox4 --> busybox5 --> Stop

并行

我们查看前面设置 Job 的 YAML :

```
kubectl get job busybox -o yaml
```

```

...
spec:
  backoffLimit: 6
  completions: 5
  parallelism: 1
  selector:
...
...
```

可以看到 `parallelism=1`，这个 `parallelism` 正是控制并行度的字段，由于这个字段默认为 1，所以这个 Job 每次只能运行一个 Pod。

`spec.completions` 和 `spec.parallelism`，这两个属性都不设置时，均取默认值 1。

当我们修改这个 `parallelism` 值大于 1 时，多个 Pod 可以同时执行。

`.spec.parallelism` 可以设置为任何非负整数。如果设置为 0，则 Job 相当于启动之后便被暂停，因为一直没有 Pod 完成，当然我们可以使用 `kubectl edit` 命令修改其值。

另外，可以 Job 运行时，使用 `kubectl scale job xxxx --replicas=n` 修改 Job 的并行数量 `parallelism` 字段的值。

当 `parallelism=2` 时，其启动过程可能如下：

```
graph TD; Start-->busybox1 Start-->busybox2; busybox1-->busybox3; busybox2-->busybox4; busybox3-->busybox5;
```

带工作队列的 Job

带工作队列的 Job，指设置了 `.spec.parallelism`，可以不设置 `.spec.completions` 的 Job。

此时 Job 需要等待所有的 Pod 完成任务，Job 才能终结。

下面创建一个 Job，这里我们启动 5 个 Pod，每个 Pod 不断向 RabbitMQ 请求消费一条消息，当 RabbitMQ 已经没有可消费的消息时，Pod 结束。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  parallelism: 5
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
        - name: c
          image: gcr.io/<project>/job-wq-1
          env:
            - name: BROKER_URL
              value: amqp://guest:guest@rabbitmq-service:5672
            - name: QUEUE
              value: job1
      restartPolicy: OnFailure
```

[Success] 提示

此处使用的 RabbitMQ 是 Kubernetes 官方供大家练习用的，不需要自己部署一个来测试。

此次容器的镜像不是真实存在的，要进行实验，可参考
https://kubernetes.io/docs/tasks/job/_print/#create-an-image 构建镜像测试。

对于没有设置 `.spec.completions` 的 Job，情况复杂的多，这种情况下任务的总数是未知的。

不设置 `.spec.completions` 的带工作队列的 Job 会同时启动多个 Pod，只要有任意一个 Pod 成功退出，Job 控制器就知道任务已经完成了，所有的 Pod 将很快会退出。当所有 Pod 结束，此 Job 才会标记为成功完成，虽然只需要有一个 Pod 完成即可，但 Job 控制器还是会等待其它 Pod 结束。

笔者摘录 Kubernetes 官方文档关于这种 Job 的描述：

- 不设置 `spec.completions`。
- 多个 Pod 之间必须相互协调，或者借助外部服务确定每个 Pod 要处理哪个工作条目。
- 每个 Pod 都可以独立确定是否其它 Pod 都已完成，进而确定 Job 是否完成
- 当 Job 中任何 Pod 成功终止，不再创建新 Pod
- 一旦至少 1 个 Pod 成功完成，并且所有 Pod 都已终止，即可宣告 Job 成功完成
- 一旦任何 Pod 成功退出，任何其它 Pod 都不应再对此任务执行任何操作或生成任何输出。所有 Pod 都应启动退出过程。

对于这种 Job，使用起来比较复杂，这种 Job 一般需要结合外部服务来完成任务，例如前面提到了 RabbitMQ。

以前面的 RabbitMQ 为例，Job 启动了 5 个 Pod 消费消息，当 RabbitMQ 没有消息了，此时 Pod1 把任务完成了，然后请求 RabbitMQ，发现没有消息可处理，Pod1 便结束运行。此时 Job 已经算完成了，因为 Job 的任务是处理完 RabbitMQ，RabbitMQ 没有消息了，此 Job 自然算完成了，但是此时还有四个 Pod 取到了消息但是就没有计算完成，这四个 Pod 继续处理完成任务。当 5 个 Pod 都结束时，此 Job 便功德圆满了。

带类型的 Job

Job 中的 Pod 都是一样的，因此如果要 Job 处理不同的工作任务，则需要外界帮忙。

举个例子，平台是一个电商系统，消息队列中有评论、订单等五类消息，那么应该设计五种程序去处理这些消息，但是 Pod 只有一种。此时可以设置原子性的任务领取中心，Job 启动 Pod 后，Pod 便向任务中心领取任务类型，领取到后，开始工作。

那么 Job 中的 Pod 可能是这样完成工作的：

```
gantt dateFormat YYYY-MM-DD title Message Queue section Job
评论处理 :done, 2021-11-01, 1d
问答处理 :done, 2021-11-02, 1d
差评处理 :done, 2021-11-03, 1d
订单处理 :done, 2021-11-04, 1d
退货处理 :active, 2021-11-05, 1d
```

[Info] 提示

限于条件，这里就不编写具有这种功能的程序了，读者只需要了解即可。

在 Job 创建的 Pod 中，会有个名为 `JOB_COMPLETION_INDEX` 的环境变量，此环境变量标识了 Pod 的索引，Pod 可以通过此索引标识自己的身份。

示例 YAML 如下：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox
spec:
  parallelism: 1
  completions: 5
  completionMode: Indexed
  template:
    spec:
      containers:
        - name: busybox
          image: busybox
          command: ["env"]
      restartPolicy: Never
```

[Info] 提示

`completionMode: Indexed` 表明当前 Pod 是带索引的，如果 `completionMode: NonIndexed` 则不带索引。

索引会按照 0, 1, 2, 3 这样递增。

执行 `kubectl apply -f job.yaml` 启动此 Job，会发现：

NAME	READY	STATUS	RESTARTS	AGE
busybox-0-j8gvz	0/1	Completed	0	29s
busybox-1-k4kfx	0/1	Completed	0	25s
busybox-2-zplxl	0/1	Completed	0	14s
busybox-3-pj4jk	0/1	Completed	0	10s
busybox-4-q5fq9	0/1	Completed	0	6s

如果我们查看 Pod 的环境变量，会发现：

```
root@master:~# kubectl logs busybox-0-j8gvz
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=busybox-0
JOB_COMPLETION_INDEX=0
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
HOME=/root
```

处理 Pod 和容器失效

Job 终止和清理

如果我们不希望 Job 运行太长时间，可以为 Job 的 `.spec.activeDeadlineSeconds` 设置一个秒数值。在 Job 的整个生命期，无论 Job 创建了多少个 Pod。一旦 Job 运行时间达到 `activeDeadlineSeconds` 秒，其所有运行中的 Pod 都会被终止，并且 Job 的状态更新为 `type: Failed` 及 `reason: DeadlineExceeded`。

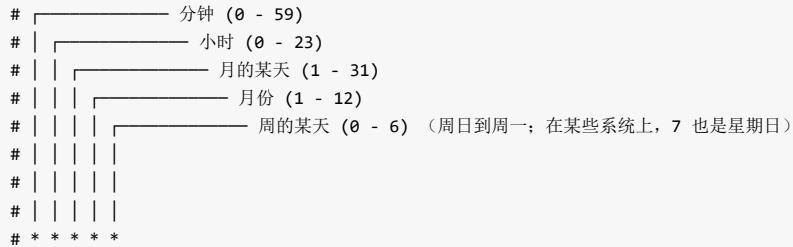
YAML 示例：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox
spec:
  completions: 5
  activeDeadlineSeconds: 2
  template:
    spec:
      containers:
        - name: busybox
          image: busybox
          command: ["/bin/sleep"]
          args: ["3"]
      restartPolicy: Never
```

CronJob

CronJobs 对于创建周期性的、反复重复的任务很有用，例如执行数据备份或者发送邮件。**CronJobs** 也可以用来计划在指定时间时来执行的独立任务，例如计划当集群看起来很空闲时 执行某个 Job。

CronJob 有个 `schedule` 字段，用来配置合适启动此 CronJob，其使用五个时间单位，每一位表示一个时刻表示，如 `"*/1 * * * *"`。



每个小时运行一次，在 0 分时开始运行： `0 * * * *`，或使用 `@hourly` 表示。

每天运行一次，在半夜 0 时 0 分运行： `0 0 * * *`，或使用 `@daily` 表示。

在每周的周日 0 点运行一次，时间在半夜： `0 0 * * 0`，或使用 `@weekly` 表示。

在每月的 1 号运行一次: `0 0 1 * *`, 或使用 `@monthly` 表示。

每年的 1 月 1 日 执行一次: `0 0 1 1 *` , 或使用 `@yearly` 表示。

CronJob 有四种符号，分别是 *、,、-、/。

符号 /，使用格式 */{值}，表示每间隔 1 个时间执行一次，例如 */1 * * * * 表示间隔一分钟执行一次。

符号 `-`，使用格式 `{值1}-{值2}`，表示范围中的每一个时间，例如 `0-2 * * * *` 表示 0 到 20 分钟的每一分钟执行一次，一共 20 次。

符号 `,`，使用格式 `*/{值1},{值2}...`，表示碰到这里的时间都执行一次，例如
1.5.6 * * * *，表示每次在 1, 5, 6 分钟时都执行一次。

组合示例，`23 0-20/2 * * *`，表示在 `0-20` 点时，每间隔 `2` 个小时执行一次，其包括的时间：

00:23:00
02:23:00
04:23:00
...
20:23:00

请参考：

<https://en.wikipedia.org/wiki/Cron>

<https://crontab.guru/>

可供实验的 YAML 示例如下：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

此 CronJob 会每分钟执行一次。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-11-06 22:05:42

- 第四章：Kubernetes 网络
 - 1.学习目标
 - 10.网络策略及资源限制

第四章：Kubernetes 网络

1.学习目标

服务与网络 - 13%

- 了解服务
- 展示对NetworkPolicies的基本了解

Service 部分

解释Kubernetes服务。

应用程序公开。

讨论可用的服务类型。

启动本地代理。

使用集群DNS

Ingress 部分

讨论入口控制器和服务之间的区别。

了解nginx和GCE输入控制器。

部署Ingress Controller。

配置入接口规则。

对于 Ingress 的内容，需要了解那些知识和概念，对于实践方面，需要学会配置规则即可。

高可用(HIGH AVAILABILITY)

讨论Kubernetes中的高可用性。

讨论并置数据库和非并置数据库。

学习Kubernetes中实现高可用性的步骤。

coredns

10.网络策略及资源限制

10.1配置calico网络实现跨节点docker容器通信

10.2网络策略

限制同一命名空间里的pod的访问

允许指定命名空间里的pod访问

允许指定命名空间里特定的pod访问

egress策略的使用

默认策略

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook文档最后更新
时间： 2021-11-07 08:53:16

- 4.1 Kubernetes 网络
 - 导读
 - Kubernetes 网络
 - Kubernetes 网络模型
 - Service
 - 没有 Service 时
 - Service 的定义和创建
 - Service 外部服务类型
 - 配置 ServiceType 公开 Service
 - LoadBalancer
 - Service 如何选择 Pod
 - kube-proxy 三种代理模式
 - Service 暴露多端口

4.1 Kubernetes 网络

导读

本章内容主要讲解 Kubernetes 中的网络和 Service 对象，作为 Kubernetes 的核心内容，Kubernetes 网络涉及到了很多东西，计算机网络原理、Linux 网络、分布式网络理论等，限于个人水平和篇幅，将不会深入讲解这些细节。

Kubernetes 网络

Kubernetes 网络模型

首先，我们要知道 Kubernetes 中的网络可以解决什么问题，在 Kubernetes 官方文档中，清晰地列举了 Kubernetes 网络的功能：

1. 高度耦合的容器间通信：这个已经被 Pods 和 localhost 通信解决了。
2. Pod 间通信；
3. Pod 和服务间通信；
4. 外部和服务间通信；

这里的服务，指的是 Service 对象。

而 Kubernetes 本身的网络服务自带了这些功能：

- NAT: 网络地址转换
- Source NAT: 替换数据包的源 IP, 通常为节点的 IP
- Destination NAT: 替换数据包的目的 IP, 通常为 Pod 的 IP
- VIP: 一个虚拟 IP, 例如分配给每个 Kubernetes Service 的 IP
- Kube-proxy: 一个网络守护程序，在每个节点上协调 Service VIP 管理

可参考 <https://kubernetes.io/zh/docs/tutorials/services/source-ip/>

Pod IP 从哪里来

还记得在 1.3 章 Docker 网络 中介绍的 container 网络么？多个容器共享一个容器的网络接口，实现多个容器共享网络、同一个 IP、同一个 hostname。Pod 内多容器共享网络就是这样创建的，Pod 的 IP 是 Docker 创建和分配的容器 IP，这个 IP 是带虚拟网卡的，因此这个 IP 是可以被 ping 的，与此同时，这个 IP 只能在当前节点中被访问。

首先创建 Pod 时，Pod 会启动一个 pause 容器，这个容器创建了一个虚拟网卡，并被 Docker 分配 IP，接着 Pod 的容器会使用 container 网络模式连接到这个 pause 容器中，pause 容器的生命周期跟 Pod 的生命周期一致。可以在工作节点上使用 docker ps -a | grep pause 命令查看 pause 容器：

```
root@slave1:~# docker ps -a | grep pause
38d0d454d68 k8s.gcr.io/pause:3.5 "/pause"
7b-48ff-87eb-6835e1244de 0
28738f5f66a k8s.gcr.io/pause:3.5 "/pause"
7a-42dc-b8bf-0c0864a36b1 0
38d0d454d68 k8s.gcr.io/pause:3.5 "/pause"
ca-4005-bfff-04ec361583x 0
90f8ce1ee68 k8s.gcr.io/pause:3.5 "/pause"
ec-421e-9768-ad38c4d175a 0
3796838a-48d4-4994-a068-68062d5ea36 0
de-4904-a068-68062d5ea36 0
e3b042b7725 k8s.gcr.io/pause:3.5 "/pause"
29eeb-6499-48ad-ab11-50d425ac1748 0
447471aa82c3 k8s.gcr.io/pause:3.5 "/pause"
3393838a-48d4-4994-a068-68062d5ea36 0
12ec0b29d4 k8s.gcr.io/pause:3.5 "/pause"
ghc_dapr-system fc59a020-8c91-4799-8159-6fc10a3e51a 0
528f132c0f5a k8s.gcr.io/pause:3.5 "/pause"
r-system f2a0b46f-262a-4073-964a-04b019a 0
1171f132c0f5a k8s.gcr.io/pause:3.5 "/pause"
system 556f0792-314d-45dc-bf96-fec4a8055d6 0
63c0465a8079 k8s.gcr.io/pause:3.5 "/pause"
_eec57297-f8c1-4d38-a6fb-979e20b64809 0
16d49908-48d4-4994-a068-68062d5ea36 0
ystrm-75faac6b-4ff1-4643-905a-03ba8ce5486a 0
fe18df77ce47e k8s.gcr.io/pause:3.5 "/pause"
aae-9bba-4639-9098-6d45692436e 0
c7741a-9239-48d4-4994-a068-68062d5ea36 0
ec-4a8-98ff-48d4-4994-a068-68062d5ea36 0
p99d146f25e k8s.gcr.io/pause:3.5 "/pause"
6-7de4-47f2-9d88-1c922913e768 0
k8s_POD_busybox-4-q5fq9_default_957f03e9-ce
k8s_POD_busybox-3-p14jk_default_013557fc-54
k8s_POD_busybox-2-zplxl_default_5ea0e8be-6f
k8s_POD_busybox-1-kkfrx_default_f0f32c9-d3
k8s_POD_busybox-0-j8gvz_default_0e611b05-84
k8s_POD_myapp-854bb8ffff-kccs8_default_da
k8s_POD_envttest_default_a05412f2-5681-4178-
k8s_POD_dapr-sidecar-injector-7879dd599b-jm
k8s_POD_dapr-dashboard-5f88cbf969-77hr9_dap
k8s_POD_dapr-operator-f496f8669-fh26m_dapr-
k8s_POD_dapr-placement-server-0_dapr-system
k8s_POD_dapr-sentry-7f6f56615f-2tf7d_daprs
k8s_POD_nginx-75c71965d8-cbth_default_70b3
k8s_POD_kube-proxy-b14jd_kube-system_052ed7
k8s_POD_weave-net-sp4xj_kube-system_ae895af
```

不过，Docker 中的容器 IP 是 172.17.0.0 地址段，而 Pod IP 的地址段却不是这样的，一般是 10.x.x.x 网络，其中用户自定义 Pod 是 10.32.0.0 地址段。

```
root@master:~# kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME           READY   STATUS    RESTARTS   AGE      IP          NODE   NOMINATED NODE   READINESS GATES
dapr-system   dapr-dashboard-5f88cbf969-77hr9   1/1    Running   0          22h     10.32.0.7   slave1  <none>        <none>
dapr-system   dapr-operator-f496f8669-fh26m   1/1    Running   0          22h     10.32.0.5   slave1  <none>        <none>
dapr-system   dapr-sentry-7f6f56615f-2tf7d   1/1    Running   1 (17 ago)  22h     10.32.0.5   slave1  <none>        <none>
dapr-system   dapr-sidecar-injector-7879dd59b-jmghc 1/1    Running   13 (11 ago)  22h     10.32.0.8   slave1  <none>        <none>
default       busybox-0-j8gvz                 0/1    Completed  0          11h     10.32.0.11  slave1  <none>        <none>
default       busybox-1-k4ktx                 0/1    Completed  0          11h     10.32.0.11  slave1  <none>        <none>
default       busybox-2-p14jk                0/1    Completed  0          11h     10.32.0.11  slave1  <none>        <none>
default       busybox-3-p14jk                0/1    Completed  0          11h     10.32.0.11  slave1  <none>        <none>
default       myapp-546bb8ffff-kccs8            0/2    CrashLoopBackOff  351 (74s ago)  16h    10.32.0.10  slave1  <none>        <none>
default       myapp-5c71965d8-cbth              0/1    Running   0          307m    10.32.0.11  slave1  <none>        <none>
kube-system   coredns-78fc6d9978-q49xr       0/1    Running   14 (37s ago)  3d1h    10.44.0.1   master  <none>        <none>
kube-system   coredns-78fc6d9978-zxcn6       0/1    Running   46 (56s ago)  3d1h    10.44.0.2   master  <none>        <none>
kube-system   etcd-master                     1/1    Running   1          3d1h    10.0.0.4   master  <none>        <none>
kube-system   kube-apiserver-master           1/1    Running   0          3d1h    10.0.0.4   master  <none>        <none>
kube-system   kube-controller-manager-master  1/1    Running   0          3d1h    10.0.0.4   master  <none>        <none>
kube-system   kube-dns-74164                   1/1    Running   1          3d1h    10.0.0.5   slave1  <none>        <none>
kube-system   kube-proxy-hds14               1/1    Running   0          3d1h    10.0.0.4   master  <none>        <none>
kube-system   kube-scheduler-master          1/1    Running   0          3d1h    10.0.0.4   master  <none>        <none>
kube-system   weave-net-8kwxn                2/2    Running   0          3d1h    10.0.0.4   master  <none>        <none>
kube-system   weave-net-sp4xj                2/2    Running   2          3d1h    10.0.0.5   slave1  <none>
```

既然 Pod 的 IP 是 Docker 分配的，为什么其地址不是 172.17.0.0 地址段？

还记得我们在部署 Kubernetes 集群时，部署的 网络插件吗？2.2 章中介绍的 weave 网络插件。

首先，在部署了 Docker 的机器上，都会有一个 docker0 的东西，这个东西叫网桥。

```
root@slave1:~# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
      inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:fa:3e:f5:80  txqueuelen 0  (Ethernet)
          RX packets 0  bytes 0 (0.0 B)
          RX errors 0  dropped 0  overrun 0  frame 0
          TX packets 0  bytes 0 (0.0 B)
          TX errors 0  dropped 0  overrun 0  carrier 0  collisions 0
```

docker 的默认网桥叫 docker0，这个网桥的 IP 是 172.17.0.1，基于这个网桥创建的容器的虚拟网卡自然是 172.17.0.0 地址段。

而如果我们使用 weave 网络插件部署集群，那么使用 ifconfig 命令，可以找到一个 weave 的自定义网桥：

```
weave: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1376
      inet 10.32.0.1 netmask 255.240.0.0 broadcast 10.47.255.255
        inet6 fe80::ac45:ebff:fe0a:31ae prefixlen 64 scopeid 0x20<link>
          ether ae:45:eb:0a:31:ae txqueuelen 1000 (Ethernet)
            RX packets 2905588 bytes 391313728 (391.3 MB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 3179102 bytes 640814125 (640.8 MB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

看，这不就对了，通过 **Kubernetes** 创建的自定义 Pod，会使用这个网桥创建 IP，其 IP 地址跟网络插件创建的网桥有关。

[Info] 提示

读者如果想了解更多的 Docker 网络知识，请访问
<https://docs.docker.com/network/bridge/>

跨节点访问 Pod

既然 Pod 的 IP 是 Docker 创建的，而 Docker 创建的 IP 只能在本地服务器上访问，那么怎么才能在别的节点上访问这个 Pod IP？当然是网络插件啦，就是在部署 **Kubernetes** 时一起部署的 **weave**。

当然，除了 **weave**，还有很多网络插件可以使用，如 **calico**、**flannel**。因为 **Kubernetes** 网络模型中有个叫 **CNI** 的标准接口，只要实现了这个接口，用啥网络插件都没问题，使用者不需要关心插件是怎么实现的。

CNI 的功能大概以下几点：

- 节点上的 Pod 可以不通过 NAT 和其他任何节点上的 Pod 通信(称为扁平化网络)，即节点间 Pod 的互相访问；
- 节点上的代理（比如：系统守护进程、**kubelet**）可以和节点上的所有 Pod 通信，即系统组件访问 Pod；

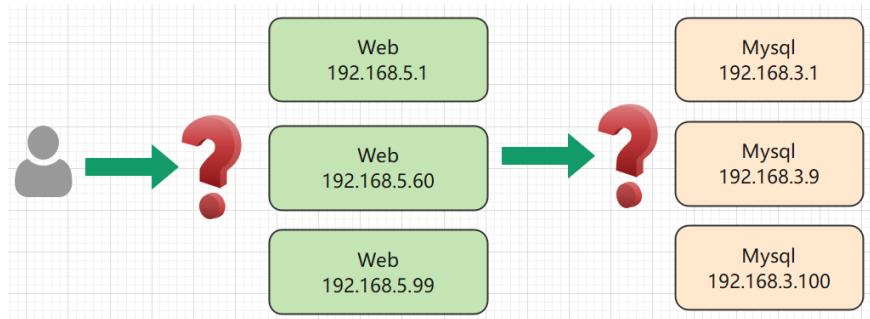
Service

Service 是一种 **Kubernetes** 的对象，它跟网络有关，**Service** 不是服务提供者，也不是应用程序接口。

Service 是将运行在一组 **Pods** 上的应用程序公开为网络服务的抽象方法。如果我们使用 **Deployment**、**Daemon** 等部署 Pod，则可为此控制器创建 **Service**，**Service** 会监控此 **Deployment** 上增加或移除 Pod 的操作，自动为所有 Pod 提供网络服务。当然，**Service** 并不是指向 **Deployment**、**Daemon** 的，而是指向这些控制器上的 Pod，通过 **Label** 指向相关的 Pod。

没有 Service 时

假如有一组 **Web Pod**，如果 **Web** 动态伸缩副本数量或因为某些原因 IP/端口发生改变，那么我们很难追踪这种变化，我们如何在客户端访问这组 **Web** 服务？又假如 **Web** 服务、**Mysql** 分别部署在不同的 Pod 中，那么 **Web** 如何查找并跟踪要连接的 **Mysql** IP 地址？



Service 可以解决这个问题。Kubernetes Service 定义了一种通常称为微服务的抽象，Service 为逻辑上的一组 Pod 提供可以访问它们的策略。当使用 Service 为一组 Pod 创建服务时，无论我们创建了多少个 Pod 副本，这些 Pod 怎么变化，Pod A 不需要关心它们调用了哪个 Pod B 副本，而且不需要知道 Pod B 的状态也不需要跟踪 Pod B。因为Service 把 Pod 的这种关联抽象化，把它们解耦了。

Service 的定义和创建

我们创建一个 Deployment 对象，包含三个 Pod 实例。

```
kubectl create deployment nginx --image=nginx:latest --replicas=3
```

接着，为这些 Pod 创建一个 Service。

```
kubectl expose deployment nginx --type=ClusterIP --port=6666 --target-port=80
```

查看创建的 Service：

```
root@master:~# kubectl get service -o wide
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
kubernetes   ClusterIP    10.96.0.1        <none>           443/TCP
mygoapp-dapr ClusterIP    None            <none>           80/TCP,50001/TCP,50002/TCP,50003/TCP
nginx      ClusterIP    10.107.200.232   <none>           6666/TCP
```

可以看到，Service 会生成一个随机 IP 10.107.200.232，我们为 Pod 映射了一个新的端口为 6666，此端口映射到了 Pod 的 80 端口中，我们可以测试这个 IP 和端口是否可用：

```
curl 10.107.200.232:6666
```

在部署了 Pod 的机器上才能使用此 IP 和端口访问 Pod。

假如有 master、slave 两个节点，Pod 都被部署到 slave 节点上，而 master 节点没有部署此 Pod 的话，master 是访问不了此 Service 的。

为了验证这样情况，我们可以消去 master 的污点，使其能够被部署用户自定义的 Pod。

此时使用 DaemonSet 部署可能更加合适。

```
kubectl taint node instance-1 node-role.kubernetes.io/master:NoSchedule-
```

然后重新部署 Deployment，但是不需要重新部署 Service。

```
kubectl delete deployment nginx
kubectl create deployment nginx --image=nginx:latest --replicas=3
```

查看这些 Pod 都被部署到哪里：

```
root@master:~# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP
nginx-55649fd747-26f5q   1/1     Running   0          11s   10.44.0.3
nginx-55649fd747-5znfc   1/1     Running   0          11s   10.44.0.4
nginx-55649fd747-77k8m   1/1     Running   0          11s   10.32.0.3
```

看来 master、slave 都部署了 Pod，那么我们在 master 节点上访问此 Service：

```
root@master:~# curl 10.107.200.232:6666
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

在 Deployment 对象上创建 Service，会直接关联一个 Deployment 中的所有 Pod，并监控是否有新建或移除 Pod，这样无论 Pod 的数量有多少，Service 都可以代理这些 Pod。

如果我们通过 YAML 定义 Service，其模板如下：

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 6666
      targetPort: 80
  type: ClusterIP
```

[Error] 提示

由于 Service 的 IP 是虚拟的，因此此 IP 是无法 Ping 通的。

Service 外部服务类型

虽然创建了 Service 后，所有的 Pod 可以被一个 IP 地址访问，但是这个 IP 只能在被部署了 Pod 的节点中访问，并且这个 IP 不能被集群外访问，这是因为我们创建 Service 的时候，使用了 ClusterIP 类型，如果是 NodePort 类型，则可以被外界访问到。

Kubernetes Service 有个 `ServiceType`，允许我们指定如何暴露服务，可以将一个 Service 暴露到集群外部，外界可以通过 IP 访问这个 Service。

Type 有四种类型，其取值说明如下：

- ClusterIP

通过集群内部 IP 暴露服务，ClusterIP 是 `ServiceType` 的默认值。

- NodePort

通过每个节点上的 IP 和静态端口（`NodePort`）暴露服务。由于其是节点上的，所以具有通过节点的公网 IP 访问这个服务。

- LoadBalancer

使用负载均衡器向外部暴露服务。外部负载均衡器可以将流量路由到自动创建的 `NodePort` 服务和 `ClusterIP` 服务上。需要云平台服务提供商的支持，分配公网 IP 才能使用。

- ExternalName

通过返回 `CNAME` 和对应值，可以将服务映射到 `externalName` 字段的内容（例如，`foo.bar.example.com`）。

需要使用 `kube-dns` 1.7 及以上版本或者 `CoreDNS` 0.0.8 及以上版本才能使用 `ExternalName` 类型。

ClusterIP、NodePort、LoadBalancer 三者是有关系的，前者是后者的基础。创建一个 `NodePort` 类型的 Service，必定带有一个 `ClusterIP`；创建一个 `LoadBalancer`，必定带有 `ClusterIP`、`NodePort`。

配置 `ServiceType` 公开 Service

我们使用 `kubectl edit service nginx` 将前面创建的 Service 修改为 `NodePort` 类型，然后查看 Service 列表：

```
root@master:~# kubectl get services -o wide
  NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
  kubernetes  ClusterIP  10.96.0.1    <none>          443/TCP
  nginx     NodePort   10.107.200.232  <none>          6666:30291/TCP
```

此时 Service 会创建一个随机端口，这个端口映射到每个部署了 Pod 的节点上，例如笔者的是 30291，此时外界可以通过使用节点 IP 访问此 Service。

每个节点上可以使用 `127.0.0.1:30291` 访问，也可以使用公网 IP 访问。



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

用户发起请求后，请求转发至 Pod 的过程如下：

```
graph TD; 用户 ==> Host(公网IP:30291) Host == NodePort ==>  
Service(Service:6666) Service == ClusterIP ==> Pod1(Pod1:80); Service ==  
ClusterIP ==> Pod2(Pod2:80); Service == ClusterIP ==> Pod3(Pod3:80);
```

LoadBalancer

当我们使用公网 IP 访问 Pod 时，也出现了一个问题，Pod 是负载均衡了，但是总不能只访问一个节点吧？节点的网络也需要负载均衡呀，而且节点 IP 这么多，用户总不能记住这么多 IP 吧？就算使用域名，域名也不能绑定这么多 IP 呀，此时 LoadBalancer 可以帮到你。

当使用 LoadBalancer 暴露服务到集群外部网络时，云基础设施需要时间来创建负载均衡器并获取服务中的 IP 地址。如果使用的是 `minikube`、`kubeadm` 等创建的自定义 Kubernetes 集群，没有集成 LoadBalancer，则会一直处于 `<Pending>` 状态。

```
graph TD; 用户 ==> LoadBalancer(123.123.123.123) LoadBalancer ==>  
Host1(100.100.100.1) LoadBalancer ==> Host2(100.100.100.2) LoadBalancer  
==> Host3(100.100.100.3) LoadBalancer ==> Host4(100.100.100.4)
```

我们删除之前 Deployment 部署 nginx 时，通过 expose 创建的 Service。

```
kubectl delete service nginx
```

然后重新创建 service。

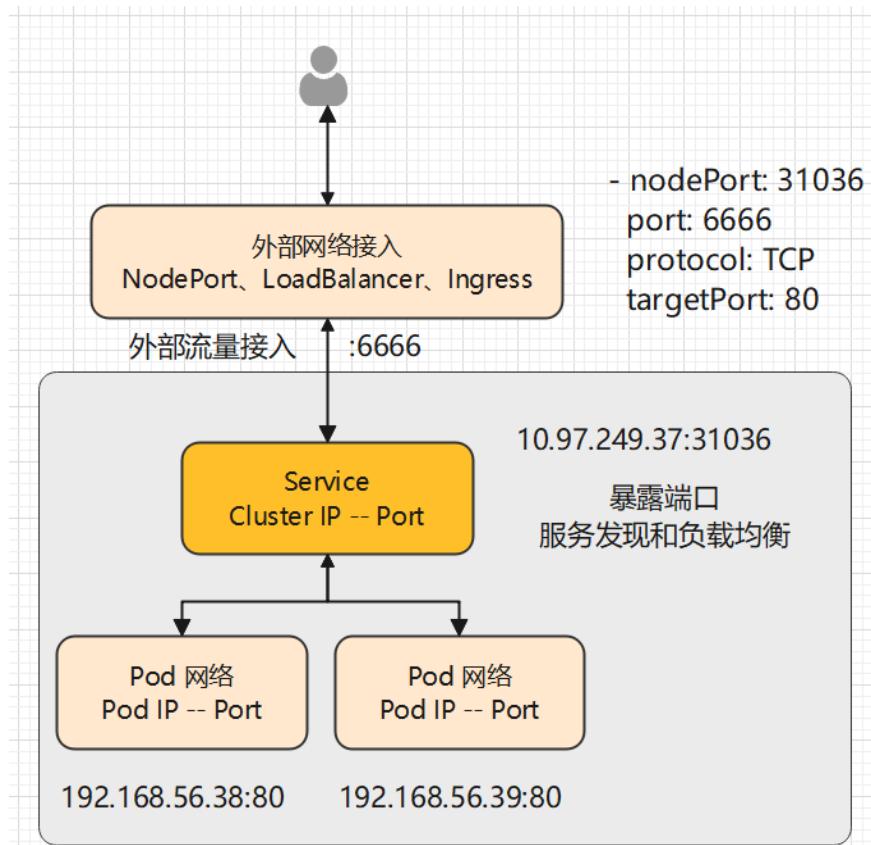
```
kubectl expose deployment nginx --type=LoadBalancer --port=80 --target-port=6666  
# 可以只填写 --port，此时映射的端口跟 Pod 端口一致
```

查询 Service:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	29h
nginx	LoadBalancer	10.97.249.37	<pending>	80:31036/TCP	30s

```
ports:
- nodePort: 31036
  port: 6666
  protocol: TCP
  targetPort: 80
  sessionAffinity: None
```

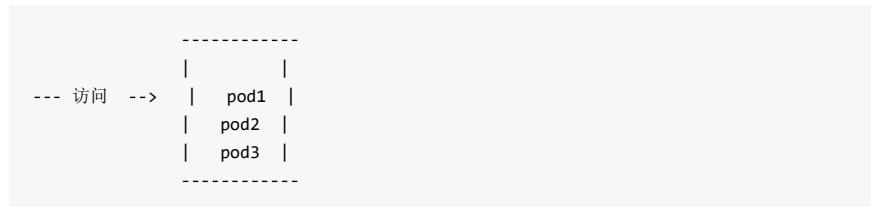
LoadBalancer 需要云服务商支持，而且收费，这里就不做其他实验了，读者知道就行。



前面提到的 LoadBalancer 便是负载均衡器，在 Kubernetes 上创建 LoadBalancer 需要云平台的支持，读者可参考腾讯云的 CLB。

Service 如何选择 Pod

当我们通过外部网络访问时，Service 会自动提供其中一个 Pod 给我们。



我们查看上一个小节创建的 Service 的 YAML 文件：

```

clusterIP: 10.100.66.200
clusterIPs:
- 10.100.66.200
...
ports:
- nodePort: 31672
  port: 6666
  protocol: TCP
  targetPort: 80

```

此 Service 的 IP 是 10.100.66.200，其类型是 ClusterIP，可以在集群内部所有节点上访问，如果集群没有安装网络插件，则 master 节点上是没法访问的。

由于我们使用的是 NodePort 网络类型，所以会生成一个 node 端口，此端口会映射到节点本地网络上。例如可以通过任意能够连接此节点的 IP 进行访问，例如 127.0.0.1:31672，或者访问此节点的内网 IP、公网 IP。

现在知道外界怎么访问此 Service，我们再来看看，Service 怎么选择哪个 Pod 提供服务。

我们查看通过 Deployment 创建的 pod:

```
kubectl get pods -o wide
```

NAME	IP	NODE	NOMINATED NODE	READINESS GATE
nginx-55649fd747-9fzlr	192.168.56.56	instance-2	<none>	<none>
nginx-55649fd747-ckhrw	192.168.56.57	instance-2	<none>	<none>
nginx-55649fd747-1dzkf	192.168.23.58	instance-2	<none>	<none>

然后我们通过命令查看 iptables 配置：

```
iptables-save
```

在终端控制台中查找 random 关键字：

```
#!/bin/sh
# DEPORTS
-A KUBE-SVC-XCMGP7HKOJUN7L6W -m comment --comment "default/nginx" -m statistic --mode random --probability 0.3333333349 -j KUBE-SEP-W2D7CXNMPA74C603
-A KUBE-SVC-ZCMKP7HKGUV3N7L6W -m comment --comment "default/nginx" -m statistic --mode random --probability 0.500000000000 -j KUBE-SEP-S4FWT5R2F4N013N
-A KUBE-SVC-ZCMKP7HKLIVN7L6W -m comment --comment "default/nginx" -j KUBE-SEP-BMMNPJYB51EVVWFK
-A KUBE-SVC-CEZP13SAUFWMSYF0 -m comment --comment "kubernetes-dashboard/kubernetes-dashboard" -j KUBE-SEP-PZ3LT0D03VRGJZT
-A KUBE-SVC-ERIFXISQEP7F7OF4 -m comment --comment "kube-system/kube-dns/dns-tcp" -m statistic --mode random --probability 0.500000000000 -j KUBE-SEP-CMLUDEIS
-DSELUOKU
```

你可以看到有三个 default/nginx，第一个 pod 被访问的机会是 0.3333...。在剩下的 2/3 的概率中，有 0.5 的概率选择第二个 Pod，剩下的 1/3 概率选择第三个 Pod。这种随机选择的模式称为 iptables 代理模式。

kube-proxy 三种代理模式

本节内容了解简单了解即可。

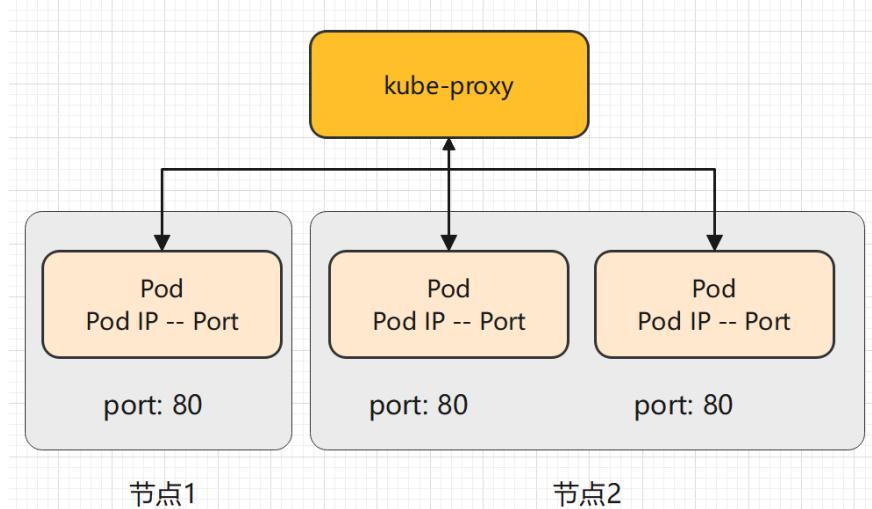
当我们使用命令创建一个 Service 时，可看到每个 Service 都有一个 IP 地址，这是由 kube-proxy 负责为 Service 实现的一种虚拟 IP，即 ClusterIP。

kube-proxy 可以为多个 Pod 创建一个统一的代理，在访问 Service 时，自动选择一个 Pod 提供服务，至于如何选择 Pod，kube-proxy 有三种模式。

- userspace 代理模式

- `iptables` 代理模式(默认)
- IPVS 代理模式 (Kubernetes v1.11 [stable]), 如果要使用 IPVS, 需要修改配置激活)

在这些代理模式中，客户端可以在不了解 Kubernetes 服务或 Pod 的任何信息的情况下，将 Port 代理到适当的后端。

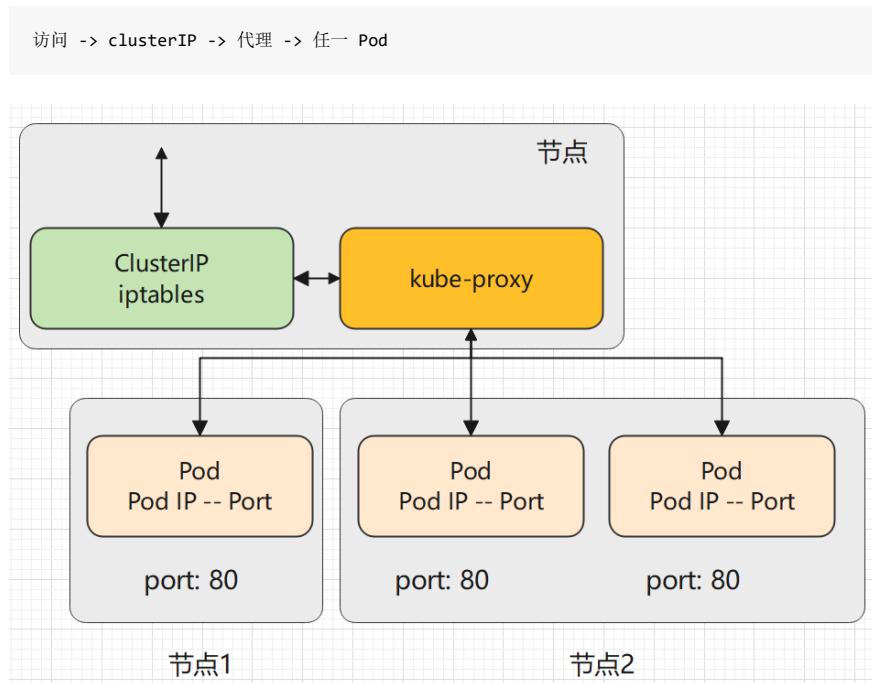


userspace 模式

userspace 模式下，`kube-proxy` 通过轮转算法选择后端。

对每个 Service，它会在本地 Node 上打开一个端口(端口号大于 30000)。任何连接到此端口的请求，都会被代理到 Service 后端的某个 Pod 上。使用哪个后端 Pod，是 `kube-proxy` 基于 YAML 的 `SessionAffinity` 终端来确定的。

最后，它配置 `iptables` 规则，捕获到达该 Service 的 `clusterIP` 和 `Port` 的请求，并重定向到代理端口，代理端口再代理请求到后端 Pod。



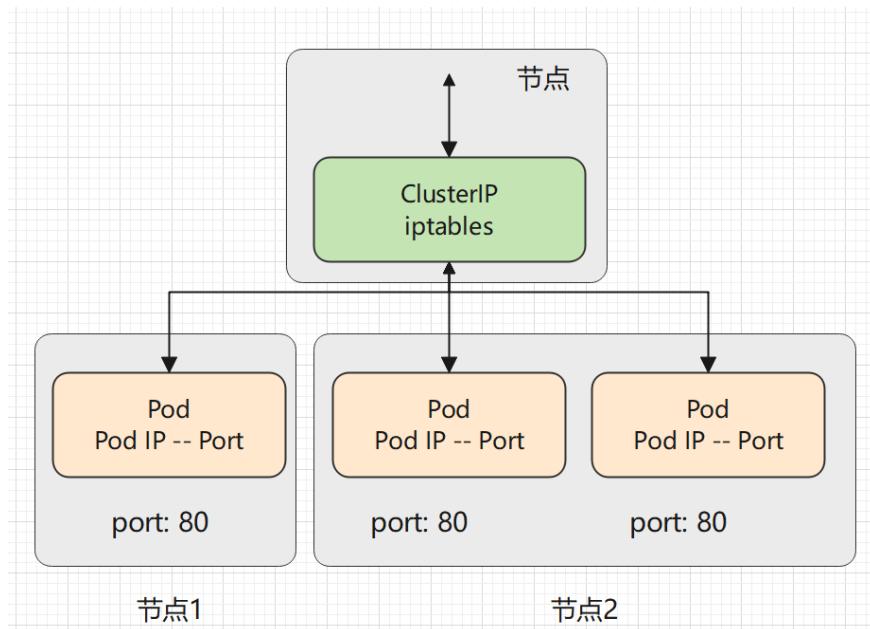
iptables 代理模式

`kube-proxy` 默认模式。`iptables` 代理模式的策略是，`kube-proxy` 在 `iptables` 模式下随机选择一个 `Pod`。

它会为每个 Service 配置 iptables 规则，捕获所有访问此 Service 的 clusterIP 请求，进而将请求重定向到 Service 的一组后端中的某个 Pod 上面。对于每个 Endpoints 对象，它也会配置 iptables 规则，这个规则会选择一个后端组合。

使用 `iptables` 处理流量具有较低的系统开销，因为流量由 Linux `netfilter` 处理，而无需在用户空间和内核空间之间切换，这种方法也可能更可靠。

如果 `kube-proxy` 在 `iptables` 模式下运行，如果随机所选的第一个 Pod 没有响应，则连接会失败，在这种情况下，会自动使用其他后端 Pod 重试。

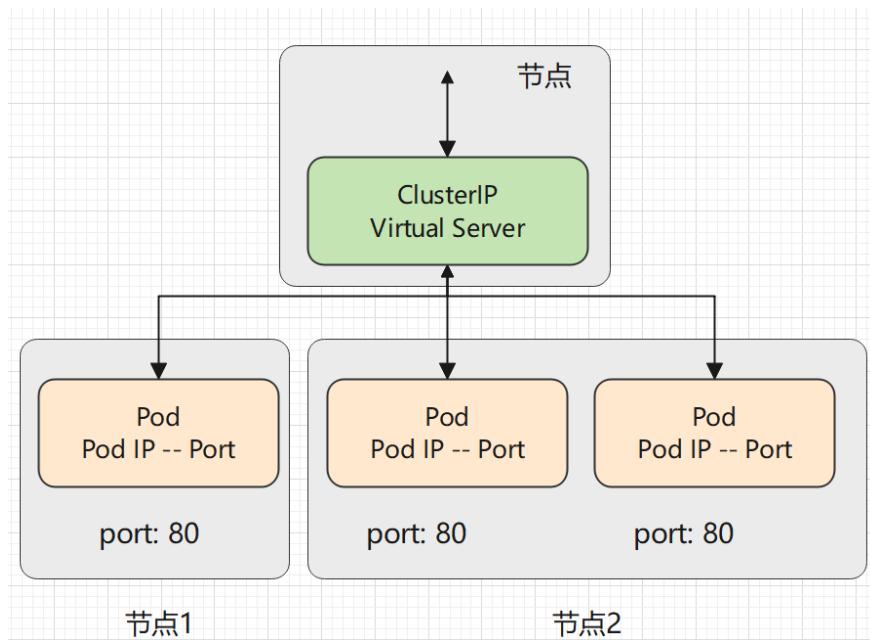


IPVS 代理模式

与其他代理模式相比，IPVS 模式还支持更高的网络流量吞吐量。与 `iptables` 模式下的 `kube-proxy` 相比，IPVS 模式下的 `kube-proxy` 重定向通信的延迟要短，并且在同步代理规则时具有更好的性能。

IPVS 提供了更多选项来平衡后端 Pod 的流量。这些是：

- rr : 轮替 (Round-Robin)
 - lc : 最少链接 (Least Connection), 即打开链接数量最少者优先
 - dh : 目标地址哈希 (Destination Hashing)
 - sh : 源地址哈希 (Source Hashing)
 - sed : 最短预期延迟 (Shortest Expected Delay)
 - nq : 从不排队 (Never Queue)



Service 暴露多端口

如果要在 Service 中暴露多个端口，则每个端口都需要设置一个名字。

```

ports:
- name: p1
  port: 2323
  protocol: TCP
  targetPort: 81
- name: p2
  port: 6666
  protocol: TCP
  targetPort: 82

```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间：2021-11-08 06:45:38

- 4.2 Endpoint
 - 创建 Endpoint、Service
 - 创建 Service
 - 创建应用
 - 创建 Endpoint
 - 连接外部服务

4.2 Endpoint

当我们为一个 Deployment 等对象创建一个 Service 时，Service 会自动创建 Endpoint。

我们查看默认命名空间的 Endpoint:

```
kubectl get endpoints
```

NAME	ENDPOINTS	AGE
kubernetes	10.170.0.2:6443	3d7h
nginx	192.168.56.24:80,192.168.56.25:80,192.168.56.26:80	59m

nginx-55649fd747-67twm	1/1	Running	0	5d21h	192.168.56.24
nginx-55649fd747-vntwb	1/1	Running	0	5d21h	192.168.56.25
nginx-55649fd747-wmr1v	1/1	Running	0	5d21h	192.168.56.26

可以看到，Endpoint 中的多个 IP，对应了每一个 Pod 的 IP，其显示的端口，也正是我们需要暴露的端口。Endpoint 记录了一组 Pod 的 IP 地址，当使用 Service 为 Pod 暴露网络时会从 Endpoint 中获得 Pod 的 IP 列表，以便能够将流量代理到合适的 Pod IP。

假如，公司创建了 A、B 两个开发小组，独立开发一个产品，其中 A 应用依赖 B 应用，需要通过远程访问获取 B 中的数据。由于两个小组都刚刚成立，各方面都还没有确定，我们如何在尽快为两个小组在远程调用上达成一些确定？使用 Consul 等服务注册和发现？这也太复杂了吧，而且 IP 也需要提前确定呀，也没有更简单的方法呢？

创建 Endpoint、Service

本节将介绍如何手动创建 Endpoint、Service。

创建 Service

创建一个没有选择器的 Service，此 Service 没有绑定任何 Pod，service.yaml 文件内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
    - protocol: TCP
      port: 6666
      targetPort: 80
```

Service 跟 Endpoint 不能直接关联，两者通过 `metadata.name` 中定义相同的名称进行匹配。

应用 YAML:

```
kubectl apply -f service.yaml
```

查看 Service 和 Endpoint:

```
root@master:~# kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP
nginx      ClusterIP   10.96.204.132    <none>        6666/TCP
root@master:~# kubectl get endpoints
NAME      ENDPOINTS      AGE
kubernetes   10.0.0.4:6443   3d17h
```

由于此 Service 没有定义选择器，因此 Service 不会自动创建 Endpoint，另外此 Service 没有 Pod 可用，暂时不能提供可用的服务。

但是，此时已经能够确定一个名为 nginx 的 Service 的 IP 地址，实际上此 IP 已经可以确定。我们把各个环节解耦了，我们无需先确定具体的应用，A、B 两个开发组，可以创建一个空的 Service，确定了 IP 和端口，然后 A、B 组继续开发应用，待 B 组开发完成后，直接部署 Pod；而 A 组可以在 B 应用没有开发完成前，就可以确定要使用的 IP。在这里，我们假设 B 组的应用叫 nginx。

创建应用

现在 B 组开发出了应用的雏形，可以使用一部分功能了，那么要先部署到云端，让 A 组连接测试。注意，B 组的应用不需要部署到 Kubernetes 中，只要能够知道此应用的 IP 和端口即可。

我们随便找台 Worker 或者 Master 节点，创建一个 nginx 容器：

```
docker run -itd -p 80:80 nginx:latest
```

为什么不用 Pod，直接创建容器？因为我们处于开发阶段，如果把 nginx 改成 mysql，我们要 Debug 呢？测试自己的数据库呢？要模拟数据呢？如果直接使用 Pod，调试会比较麻烦，我们在生产时再通过 Deployment 创建应用，但是此时我们可以使用 Docker 部署自己的数据库或者本地应用，Docker 调试比较容易，同时也很容易操作。当然，去掉 Docker 也是可以的。

总之，我们创建了 **Service**，可以提供了抽象，至于怎么提供这个服务，我们可以使用 **Pod**，也可以直接使用 **Docker**，也可以在服务器中直接使用命令启动程序而不使用容器。总之，我们目前把多个环节解耦了，**A** 组只需要关心通过 **Service** 中的 IP 和端口能够访问到 **B** 应用即可；对应 **B** 组，其开发的应用还处于迭代中，直接上 **Kubernetes**，不是一个好选择，可以临时使用 **Docker** 快速部署。

在前面，**Docker** 已经部署完成了，接着查询这个容器的 ip，：

```
docker inspect {容器id} | grep IPAddress
```

笔者得到的是结果是 "IPAddress": "172.17.0.3"，我们在当前服务器中使用 `curl 172.17.0.3`，测试是否能够访问 **nginx**，如果没问题我们来进行下一步操作。

创建 Endpoint

现在，我们创建了 **Service**，能够给 **A** 组的应用提供一个 IP 和端口，同时 **B** 组也提供了一个临时测试的后端(什么形式都行)，但是现在 **Service** 还没有关联这个后端，暂时不能通过 **Service** 访问 **B** 应用，接下来我们将这两者关联起来。

创建一个 `endpoint.yaml` 文件，内容如下(注意替换ip为你容器访问ip)：

```
apiVersion: v1
kind: Endpoints
metadata:
  name: nginx
subsets:
- addresses:
  - ip: 172.17.0.3
  ports:
  - port: 80
```

`metadata.name` 名称必须跟 **Service** 一致，否则不会相匹配。

然后应用此 `yaml`：

```
kubectl apply -f endpoint.yaml
```

查看 **Endpoint**：

```
kubectl get endpoints
# 不能填写成 endpoint
```

NAME	ENDPOINTS	AGE
nginx	172.17.0.3:80	7s

然后访问 **Service** 中的 IP：

```
curl 10.96.204.132:6666
```

```

root@instance-2:~# curl 10.96.204.132:6666
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>

```

如果 **Endpoint** 需要跟踪多个 IP (多个 Pod 或者容器或者应用), 可以使用:

```

- addresses:
  - ip: 172.17.0.2
  - ip: 172.17.0.3
  - ip: 172.17.0.4
  ...
  ...

```

我们定义好 **Service** 和 **Endpoint** 后, 后续 **Debug** 应用时, 可以手动添加实例数量, 让 **B** 应用一直能够快速迭代。如果 **B** 应用开发完毕, 则应该使用 **Pod** 部署。

后续使用 **Pod** 时, 修改其 `metadata`, 加上选择器。

```

metadata:
  annotations:
  labels:
    app: nginx
    name: nginx
    namespace: default

```

```

graph TD; 用户 ==> Service(B Service 10.96.204.132) Service ==> Endpoint(B
Endpoint 10.32.0.1,10.32.0.2,10.32.0.3) Endpoint ==> Pod1(Pod1 10.32.0.1)
Endpoint ==> Pod2(Pod2 10.32.0.2) Endpoint ==> Pod3(Pod3 10.32.0.3)

```

连接外部服务

假如, 应用依赖于 **Redis**、**Mysql**, 但是考虑了自建 **Mysql** 的复杂性和宕机修复难度, 决定使用云平台的 **Mysql** 服务。

好的, 现在公司买了云平台的多节点 **Mysql** 服务, 其中有两个主节点, 对外提供服务, 其 IP 分别为:

```

111.111.111.111
222.222.222.222

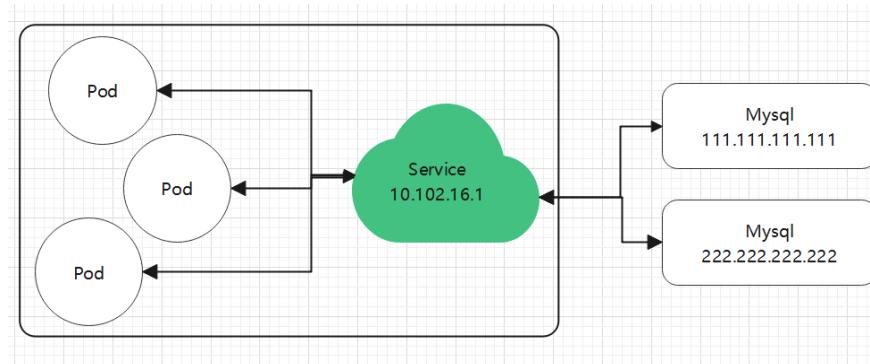
```

但是，**Mysql** 服务在云服务器上，不在自己的 **Kubernetes** 集群上，我们想在集群中创建一个 **Service** 来对这两个 IP 进行负载均衡，那么，我们怎么在 **A** 应用中访问 **Mysql**？写死 IP？但是现在有两个 **Mysql** IP，代码里面只能写一个呀，这样做怎么做负载均衡？

此时，还是可以使用 **Service**、**Endpoint** 的。无非就是在 **Endpoint** 的 IP 列表中，使用公网 IP。

```
- addresses:  
  - ip: 111.111.111.111  
  - ip: 222.222.222.222  
    ... ...  
- ports:  
  - port: 3306
```

这样，在集群中还是可以使用 **Service** 的 IP(内网)，**A** 应用不需要知道有多少个 **Mysql** 实例，也不需要知道 **Mysql** 的 IP 会不会变化或者某一天挂掉了，也不需要关心平衡访问 **Mysql** 的流量，我们只需要知道 **Service** 的 IP 即可。



Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-07 15:07:58

- 负载均衡器
 - 什么是 Ingress
 - Ingress 与 Service
- 安装 Ingress 控制器
 - Apisix Ingress
 - Nginx Ingress
- 创建和使用 Ingress
 - 快速实践
 - 解决小问题
- 实战 Ingress
 - 部署 web
 - 创建 Ingress
 - 公网访问
- Ingress 配置
 - Ingress 结构
 - Ingress 类型

4.3 ingress

在 4.1 章中，介绍了 Kubernetes 网络以及 Service，当我们的服务在多台服务器上时，我们需要提供一个统一的访问点，用户只需要访问这个地址即可，而不需要所有服务器的 IP 地址。使用 LoadBalancer 是个好办法，例如腾讯云的 CLB，可以创建一个 IP，当用户访问这个 IP 时，可以动态将用户请求头部的 IP 地址替换成需要访问的节点服务器地址。

```
flowchart LR
    用户 --> LB(LoadBalancer IP)
    LB --> Node1
    LB --> Node2
    LB --> Node3
    Node1 --> Node2
    Node2 --> Node3
```

这部分跟 TCP 的 IP 头部有关，TCP 是传输层的对象，例如当 TCP IP 为 111.111.111.111 时，用户的 TCP 数据报到底下一个路由时，这个路由可以修改 TCP 中的 IP 部分，改成 222.222.222.1，这样此 TCP 数据报便会寻找 IP 为 222.222.222.1 的服务器。

负载均衡器





负载均衡器有两类，区别在于四层网络和七层网络的支持，传输层在第四层，这层协议有 TCP/UDP/TCP SSL 等，而七层有 HTTP/HTTPS。

一般来说，像腾讯云等云服务商，人家有很多机房，提供 DNS, NAT 这些服务很正常，但是欸，我就是不用，我就要自己做，那你可以用 Kubernetes 中的 Ingress 来做。

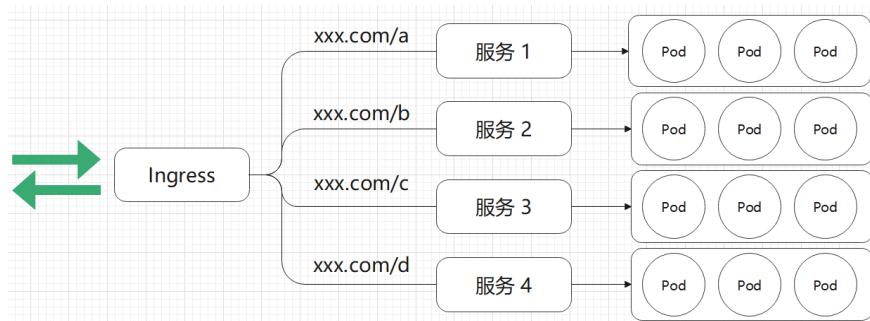
什么是 Ingress

我们做网站时，使用 Nginx 做 Web 服务器，会使用一个子域名绑定一个网站，`a.xxx.com` 绑定 A 网站，`b.xxx.com` 绑定 B 网站，这样在一个域名的不同子域名可以访问不同的站点，对于现在的大多数互联网网站，依然会使用这种方法划分。

在微服务架构中，多个模块部署在不同的服务器上，则但是我们希望都通过 `xxx.com` 这个域名直接访问，就好像所有模块都在一起，让用户感觉只有一个网站。则可能会使用目录路径对模块进行划分，例如如果我们要实现 `xxx.com/a` 访问 A 模块，`xxx.com/b` 访问 B 模块，但对用户来说，一直在访问 `xxx.com` 这个域名。

这种需求，我们可以使用 nginx 进行反向代理，而在 Kubernetes 中，这种需求也是一模一样的。

首先，我们可以为 A、B、C 等应用，创建多个 Service，每个 Service 访问一个应用，然后使用 Ingress 配置路由规则，决定 URL 可以访问哪个 Service。



Ingress 公开了从集群外部到集群内服务的 HTTP 和 HTTPS 路由，Ingress 资源上定义的规则控制了路由。

Ingress 可以让集群中的多个 Service 能够从集群外访问，Ingress 还提供负载均衡、SSL/TLS 和基于名称的虚拟服务器等，Ingress 可以配置边缘路由器或其他前端工具来帮助处理网络流量，但是一般都是通自己的负载均衡器来实现。

Ingress 有两部分，一部分是 LoadBalancer，提供统一入口，代理请求；另一部分是 Ingress 控制器，复制定义路由规则等。

如果不使用公有云平台的 LoadBalancer，那么就自己搭建一个服务器，这台服务器加入到 Kubernetes 集群中，做流量入口，这台服务器网络接口必须够大，抗得住流量。

Ingress 与 Service

在前面，我们已经学习到了 **Service**，通过 **Service** 我们可以暴露一个端口到外网中，通过这个端口可以访问应用。

其中，有两种方法可以暴露 **Service**，可以让其被集群外部访问：

- 使用 `Service.Type=LoadBalancer`
- 使用 `Service.Type=NodePort`

Service 的访问方式是 IP，每次要将服务公开给外界时，都必须创建一个新的 **LoadBalancer** 并向云服务商获取一个公网 IP 地址。或者使用 **NodePort**，但是只能在一台服务器上被访问，而且 **Service** 只能为一种 Pod 服务，暴露一个或多个端口，那么 N 个服务，就需要创建 N 个 **Service**。**Service** 虽然能够公开端口到外部网络中，但是无法将这些服务合并到一个 `example.com/{服务}` 中访问，**Service** 需要通过不同的端口访问。

如果你有一个 `example.com` 域名，你部署了多个 **Web** 服务，其中有两个子模块分别为课程(course)、考试(exam)两个微服务，这些模块构成了一个培训网站。此时我们希望访问 `example.com/api/course` 能够访问课程学习模块，访问 `example.com/api/exam` 能够访问考试模块。显然，**Service** 是无法做到的。

使用 **Ingress**，可以轻松设置路由规则，而且无需创建一堆 **LoadBalancers/Nodes** 公开每个服务，并且 **Ingress** 本身具有很多功能。

Ingress 也需要 Service。

安装 Ingress 控制器

Ingress 控制器有多种实现，其中 **Kubernetes** 官方有一个名为 **Ingress-nginx** 的实现，其它实现还有 **Kong Ingress**、**Traefik**、**HAProxy Ingress** 等，在本章中，我们安装使用 **Apisix Ingress** 或 **Nginx Ingress**，但是只使用 **Nginx Ingress** 做演示，其它控制器请参考官方文档：<https://kubernetes.github.io/ingress-nginx/deploy/#provider-specific-steps>

Apisix Ingress

Helm 是一个 **Kubernetes** 上的打包工具，如果服务器中已经有 **Helm**，那么我们通过 **Helm** 工具安装 **Apisix**：

```
sudo snap install helm --classic

helm repo add apisix https://charts.apiseven.com
helm repo update
kubectl create ns ingress-apisix
helm install apisix apisix/apisix \
--set gateway.type=NodePort \
--set ingress-controller.enabled=true \
--namespace ingress-apisix

kubectl get service --namespace ingress-apisix
```

安装 **dashboard**：

```
helm install apisix-dashboard apisix/apisix-dashboard --namespace ingress-apisix
```

然后会提示你执行一些命令，将命令复制到部署了 `dashboard` 的节点运行。

安装 Apisix Ingress 控制器：

```
helm install apisix-ingress-controller apisix/apisix-ingress-controller --namespace ingre
```

查看这些组件的 Service：

```
root@master:~# kubectl get services --namespace=ingress-apisix
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)
apisix-admin   ClusterIP 10.96.38.188 <none>       9180/TCP
apisix-dashboard   ClusterIP 10.111.179.134 <none>       80/TCP
apisix-etcd    ClusterIP 10.98.124.145 <none>       2379/TCP,2380/T
apisix-etcd-headless ClusterIP None        <none>       2379/TCP,2380/T
apisix-gateway  NodePort   10.100.83.155 <none>       80:30712/TCP
apisix-ingress-controller ClusterIP 10.108.233.236 <none>       80/TCP
```

修改 `apisix-dashboard` 的 Service 类型为 `NodePort`。

```
kubectl edit service apisix-dashboard --namespace ingress-apisix
```

```
root@master:~# kubectl get services --namespace=ingress-apisix
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)
apisix-admin   ClusterIP 10.96.38.188 <none>       9180/TCP
apisix-dashboard   NodePort   10.111.179.134 <none>       80:31429/TCP
apisix-etcd    ClusterIP 10.98.124.145 <none>       2379/TCP,2380/T
apisix-etcd-headless ClusterIP None        <none>       2379/TCP,2380/T
apisix-gateway  NodePort   10.100.83.155 <none>       80:30712/TCP
apisix-ingress-controller ClusterIP 10.108.233.236 <none>       80/TCP
```

查看 `apisix-dashboard` 安装到了哪里：

```
root@master:~# kubectl get pods --namespace ingress-apisix -o wide
NAME                           READY   STATUS      RESTARTS
apisix-dashboard-66b4ddb8b8-zcgs4   0/1     CrashLoopBackOff   4 (66s ago)
```

由于其部署在 `master` 节点上，则可以使用 `master` 的公网 IP 访问它。

查找其映射到节点的端口：

```
root@master:~# kubectl get service --namespace ingress-apisix
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)
apisix-dashboard   NodePort   10.109.51.102 <none>       80:32026/TCP
```

请自行参考配置 pvc, <https://apisix.apache.org/zh/docs/ingress-controller/deployments/minikube>

卸载方法:

```
helm uninstall apisix apisix-dashboard apisix-ingress-controller --namespace=ingress
```

[Info] 提示

由于 Apisix Ingress 配置复杂，笔者已经放弃尝试。

Nginx Ingress

通过 Helm 工具安装 ingress-nginx:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update

helm upgrade --install ingress-nginx ingress-nginx \
--repo https://kubernetes.github.io/ingress-nginx \
--namespace ingress-nginx --create-namespace
```

检测安装的版本:

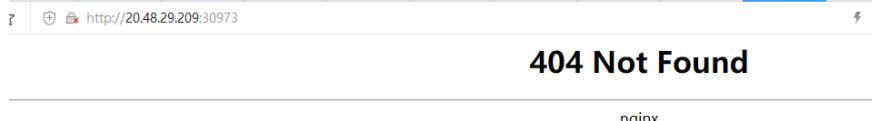
```
POD_NAME=$(kubectl get pods -l app.kubernetes.io/name=ingress-nginx -o jsonpath='{.items[0].metadata.name}')
kubectl exec -it $POD_NAME -- /nginx-ingress-controller --version
```

接下来我们检查是否有成功安装 ingress-nginx 控制器，安装 ingress 控制器完成后，在 kube-system 命名空间会有相关的 Pod 出现。

```
kubectl get services --all-namespaces
```

Namespace	Name	Type	IP	Port	Labels
default	ingress-nginx-controller	LoadBalancer	10.108.196.41	30973	<none>
default	ingress-nginx-controller-admission	ClusterIP	10.99.253.53	443	<none>

其中 ingress-nginx-controller 已被映射到节点的 30973 端口，所以可以通过公网 IP 访问。



创建和使用 Ingress

快速实践

这里我们快速创建一个简单的 Ingress 路由，练习一下。

创建 Pod:

```
kubectl create deployment nginx --image=nginx:latest --replicas=3
```

通过命令创建 Service:

```
kubectl expose deployment nginx --type=NodePort --port=8080 --target-port=80
```

创建 Ingress 服务:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  - host: k1.whuanle.cn
    defaultBackend:
      service:
        name: nginx
        port:
          number: 8080
```

如果执行 `kubectl apply -f` 时出现问题，请参考下一小节。

`k1.whuanle.cn` 是笔者的域名。你也可以将域名解析到部署了 ingress 的服务器上。

获取 ingress:

```
root@instance-2:~# kubectl get ingress
NAME      CLASS      HOSTS      ADDRESS      PORTS      AGE
test-ingress  nginx  k1.whuanle.cn  10.99.63.81  80      3m37s
```

ADDRESS 可能在1分钟后才会生成，如果其没有自动生成 ADDRESS，这里可能暂时不能访问，后面会提及如何解决这个问题。

因为 `10.99.63.81` 不是公网 IP，域名不能解析到此 IP 上。所以此时还是相当于访问 IP:端口，ingress 负载均衡器不起作用。

此地址同时是 ingress 控制器的地址。

按照 NodePort 可以使用域名访问后端服务。

A screenshot of a web browser window. The address bar at the top shows the URL "http://k1.whuanle.cn:30564". The main content area features a large, bold, black "Welcome to nginx!" heading. Below it, a message states, "If you see this page, the nginx web server is successfully installed and working. Further configuration is required." At the bottom, there are links for "online documentation and support" (nginx.org) and "commercial support" (nginx.com). A footer message says, "Thank you for using nginx.".

解决小问题

如果应用 YAML 文件创建 **Ingress** 时，出现下面的提示，我们需要做些修改才能成功创建 **ingress**。或者多试几次。

```
Error from server (InternalError): error when creating "ingress.yaml": Internal error
```

使用下面的命令查看 webhook

```
kubectl get validatingwebhookconfigurations
```

NAME	WEBHOOKS	AGE
ingress-nginx-admission	1	4h5m

删除 ingress-nginx-admission :

```
kubectl delete -A ValidatingWebhookConfiguration ingress-nginx-admission
```

接着重新创建 Ingress 即可。

接下来，我们将实际创建 Ingress，在本小节中，将使用 `hello-world.info` 域名，通过 `/web1` 访问一个网站，`web2` 访问另一个网站。

删除之前创建的对象：

```
kubectl delete deployment nginx  
kubectl delete svc nginx  
kubectl delete ingress test-ingress
```

实战 Ingress

部署 web

使用 Kubernetes 官方的容器创建一个 Hello world 的网站应用。

```
kubectl create deployment web1 --image=gcr.io/google-samples/hello-app:1.0 --replicas=3  
kubectl create deployment web2 --image=gcr.io/google-samples/hello-app:1.0 --replicas=3
```

创建 Service，暴露端口：

```
apiVersion: v1  
kind: Service  
metadata:  
  labels:  
    app: web1  
  name: web1  
spec:  
  ports:  
  - name: http  
    port: 8080  
    protocol: TCP  
    targetPort: 8080  
  selector:  
    app: web1  
  type: NodePort  
---  
apiVersion: v1  
kind: Service  
metadata:  
  labels:  
    app: web2  
  name: web2  
spec:  
  ports:  
  - name: http  
    port: 9090  
    protocol: TCP  
    targetPort: 8080  
  selector:  
    app: web2  
  type: NodePort
```

或使用

```
kubectl expose deployment web1 --type=NodePort --port=8080  
kubectl expose deployment web2 --type=NodePort --port=8080
```

查看 Service 信息：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
web1	NodePort	10.100.137.80	<none>	8080:31901/TCP
web2	NodePort	10.105.184.142	<none>	9090:31631/TCP

通过 ClusterIP 可以访问对于的 nginx 应用，其端口为 8080 或 9090。既然要创建 Ingress，我们就不需要关心 NodePort 端口了，只需要知道 8080、9090 即可。

创建 Ingress

这里我们为 Ingress 配置路由规则，访问 /web1、/web2 时，是在访问不同的应用，其中我们约定，要绑定的域名是 hello-world.info。

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  ingressClassName: nginx
  rules:
    - host: hello-world.info
      http:
        paths:
          - path: /web1
            pathType: Prefix
            backend:
              service:
                name: web1
                port:
                  number: 8080
          - path: /web2
            pathType: Prefix
            backend:
              service:
                name: web2
                port:
                  number: 9090
```

当访问 <http://hello-world.info/web1> 时，会导向 Service Web1，接着由 Service 导向 Pod。

等十来秒钟，然后查看 ingress 列表：

```
root@master:~# kubectl get ingress
NAME      CLASS   HOSTS           ADDRESS      PORTS   AGE
test-ingress  nginx  hello-world.info  10.99.63.81  80      3m37s
```

ADDRESS 中的是负载均衡器的地址，记下此地址，我们要将 hello-world.info 这个域名解析到 10.99.63.81，才能接着访问。

查看 Ingress：

```
kubectl describe ingress test-ingress
```

```

root@master:~# kubectl describe ingress test-ingress
Name:           test-ingress
Namespace:      default
Address:        10.99.63.81
Default backend: default-http-backend:80 (<error: endpoints "default-http-backend" not found>
Rules:
  Host          Path  Backends
  ----          ---   -----
  k1.whuanle.cn
            /     web1:8080 (10.32.0.12:80,10.32.0.13:80,10.32.0.16:80)
            /web2  web2:9090 (10.32.0.11:80,10.32.0.14:80,10.32.0.15:80)
Annotations:    <none>
Events:
  Type  Reason  Age           From           Message
  ----  -----  --           --           --
  Normal Sync   6m53s (x2 over 7m26s)  nginx-ingress-controller  Scheduled for sync
  Normal Sync   5m53s (x2 over 5m53s)   nginx-ingress-controller  Scheduled for sync

```

由于我们没有为 Ingress 提供可以访问的公网 IP，因此没办法绑定域名，但是我们可以修改内网的 DNS，使得在内网可以通过域名访问。

打开 `/etc/hosts` 文件，添加一行：

```
10.99.63.81 hello-world.info
```

[Error] 提示

由于前面使用了 Deployment 部署服务，只有一部分节点部署到了 Pod，未被部署 Pod 的节点，不能通过此 IP 访问 Pod。因此这个配置只能在部署了 Pod 的节点上生效。

之后直接访问 `curl http://hello-world.info/web1`、`curl http://hello-world.info/web2`，便可以访问到具体的 Pod 服务。

```

root@slavel:~# curl http://hello-world.info/web1
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
root@slavel:~# curl http://hello-world.info/web1
Hello, world!
Version: 1.0.0
Hostname: web1-7856799799-xr5xw
root@slavel:~# curl http://hello-world.info/web2
Hello, world!
Version: 1.0.0
Hostname: web2-5b669f8984-7jd7f

```

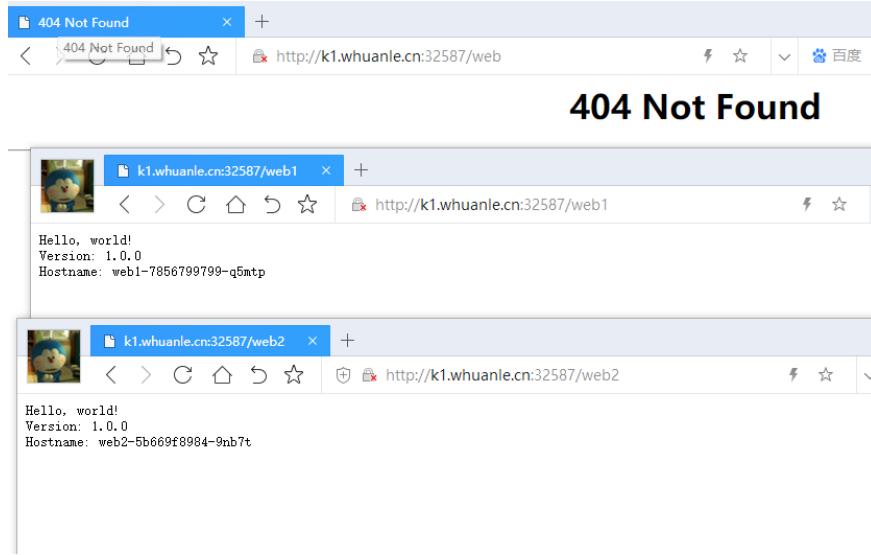
公网访问

由于没有公网负载均衡器，上面笔者的 Ingress 创建的 负载均衡器 IP 是集群内的，地址是 10.99.63.81，这个地址其实就是 Ingress 控制器的 Service IP。

```
root@master:~# kubectl get svc -n ingress-nginx
NAME                      TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)
ingress-nginx-controller   NodePort   10.99.63.81 <none>       80:32587/TCP
ingress-nginx-controller-admission ClusterIP  10.106.108.53 <none>       443/TCP
```

如果在集群中访问，则只需要映射 10.99.63.81 然后访问 80 端口即可。

如果是公网访问，也有办法，因为此时 Service NodePort 是 32587，那么使用域名解析到节点 IP，在使用此端口访问即可。



Ingress 配置

在前面，我们使用一个 Ingress 关联了多个 Service，其简化 YAML 如下：

```
rules:
  ...
    backend:
      service:
        name: web1
    backend:
      service:
        name: web2
```

可以看到，这个 Ingress 关联了 Service，不过，Ingress 并不是通过 Service 转发流量。

我们通过 `kubectl describe ingress` 可以看到：

```
Rules:
Host          Path  Backends
----          ----
hello-world.info
  /web1      web1:8080 (192.168.56.1:80,192.168.56.2:80,192.168.56.63:80)
  /web2      web2:9090 (192.168.56.3:80,192.168.56.4:80,192.168.56.5:80)
```

当用户访问 `hello-world.info/web1` 时，**Ingress** 并不会将用户请求转发给 Service `web1`，而是通过 Service `web1` 选择一个 Pod，这个 Pod 列表和 Pod 的 IP 列表，是 **Endpoint** 提供的。

当用户请求时 `hello-world.info` 时，会首先进行 DNS 查找，获取实际请求的 IP，然后请求会发送到 `ingress-nginx` 这个 Ingress 控制器，接着 Ingress 控制器根据请求的 HOST/URL 确定要访问哪个 Service，最后根据 Service 关联的 Endpoint 对象，查看 Pod 的 IP，最后，Ingress 控制器将请求转发给其中一个 Pod。

Ingress 结构

Ingress 由 Ingress Controller、Ingress API 两部分组成，为了让 Ingress 资源工作，集群必须有一个正在运行的 Ingress 控制器，在前面，我们使用了 `ingress-nginx` 控制器。Ingress 控制器负责满足 Ingress 中所设置的规则，即路由规则，例如 `nginx`，我们脱离 Kubernetes，单独部署使用 `nginx` 也可以完成这个需求。

由于 CKAD 认证中，只要求掌握 Ingress 控制器，而且很多书中都不会深入 ingress，所以这里笔者不再深入讲解，读者可以自行参考官方文档。

我们还可以使用其它 ingress 控制器，例如经常提及到的 Istio，详细可以参考官方完整名单：

<https://kubernetes.io/zh/docs/concepts/services-networking/ingress-controllers/#其他控制器>

<https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/configmap/>

Ingress 类型

在深入 Ingress 之前，我们来了解一下 Ingress 的类型/架构，这也称为调度方式，这可以帮助我们了解如何设计 Ingress。

默认后端

在 Ingress 中，当用户请求的 URL 没有任何匹配的 Service 可用时，会返回 404，但是我们可以指定一个默认后端(Service)，当请求的 URL 不存在时对应路由规则时，此请求会被路由到默认的后端(Service)中。

```
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
      name: static-assets
  rules:
    - http:
        paths:
          - path: /icons
            pathType: ImplementationSpecific
            backend:
              ... ...
```

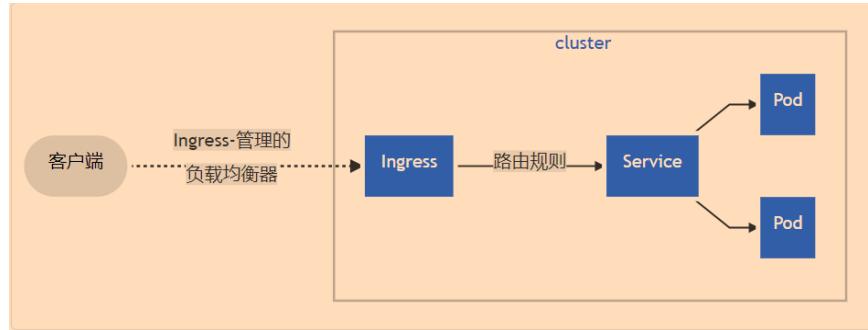
单个 Service

这种 Ingress 只有一个后端，即一个 Service，Ingress 会将所有流量都发送到同一 Service 的简单 Ingress。

由于其只需要一个 Service，我们可以直接使用 `DefaultBackend` 指定一个 Service 即可，如 快速实践一小节中的 Ingress 模板。

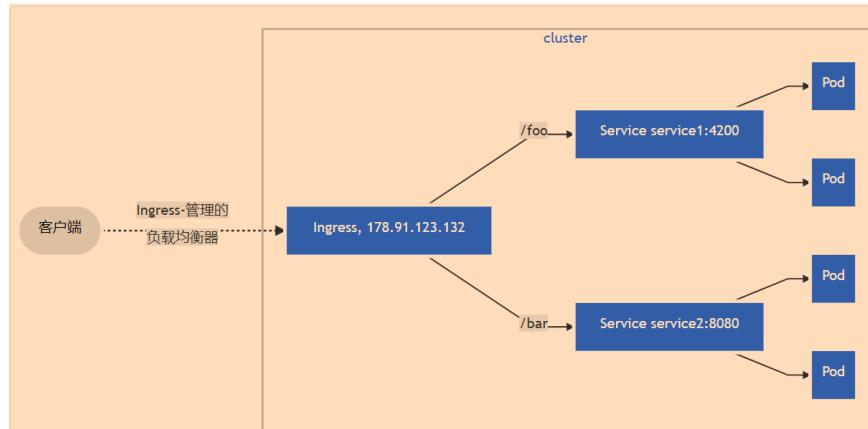
```
spec:  
  defaultBackend:  
    service:  
      name: nginx  
      port:  
        number: 8080
```

```
spec:  
  defaultBackend:  
    resource:  
      apiGroup: my666.com  
      kind: StorageBucket  
      name: static-assets
```



简单扇出

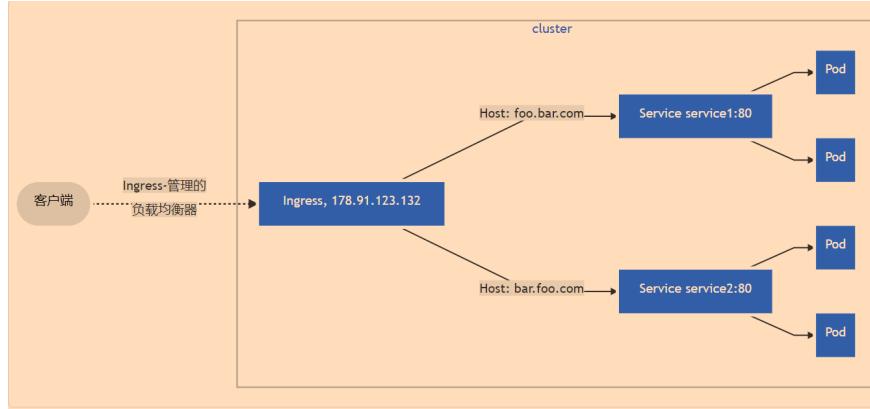
一个扇出配置(带 `rule`)根据请求的 HTTP URL，将来自同一 IP 地址的流量路由到多个 Service 中，这种方式也称为 URL 映射，即我们在前面创建的多 Service。



如果用户请求的 URL，没有 `hosts` 或 `paths` 与 Ingress 对象中的 HTTP 请求匹配，则流量将路由到默认后端。

基于名称的虚拟托管

即域名模式，Ingress 根据域名的不同，将请求转发到不同的 Pod 中，一般会使用不同的子域名访问不同的 Service。



TLS

Ingress 也支持 HTTPS，通过 SSL 证书为 HTTPS 提供安全保障。

当我们在云服务商的域名处，为域名分配 SSL 证书后，获取 `.key`、`.crt` 两个证书文件，TLS Secret 必须包含名为 `tls.crt` 和 `tls.key`，然后使用命令将证书保存到 Kubernetes 的 Secret 对象中。

```
kubectl create secret tls {secret名称} --key {证书名称}.key --cert {证书名称}.crt
```

然后在 Ingress 对象的 YAML 文件中，加上 TLS 证书：

```
spec:
  tls:
    - hosts:
        - mywebsite.com
      secretName: {secret名称}
  rules:
    - http:
      ... ...
```

TLS 连接终止于 Ingress 端，客户端跟 Ingress 之间通过 HTTPS 传输，然后 Ingress 将流量转发到 Pod 中，此时以 HTTP 传输，Pod 中的 Web 应用不需要支持 HTTPS。

我们也可以不使用 secret，TLS 证书直接附加到 Ingress YAML 中，如：

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 编码的 cert
  tls.key: base64 编码的 key
type: kubernetes.io/tls
```

secret 的知识在 5.2 章中讲解。

笔者实验过程如下所示。

申请域名证书:

算法选择 RSA算法 推荐! ECC算法

证书绑定域名 * ✓

即绑定证书的域名, 请填写单个域名。例如tencent.com、ssl.tencent.com。

申请邮箱 * ✓

证书备注名 ✓

私钥密码(选填) 为了保障私钥安全, 目前不支持密码找回功能, 请您牢记私钥密码。
如需部署腾讯云负载均衡、CDN等云服务, 请勿填写私钥密码。 ✓

确认密码 ✓

[更多](#)

名称	类型	压缩大小	密码保护
1_k1.whuanle.cn_bundle.crt	安全证书	3 KB	否
2_k1.whuanle.cn.key	KEY文件	2 KB	否

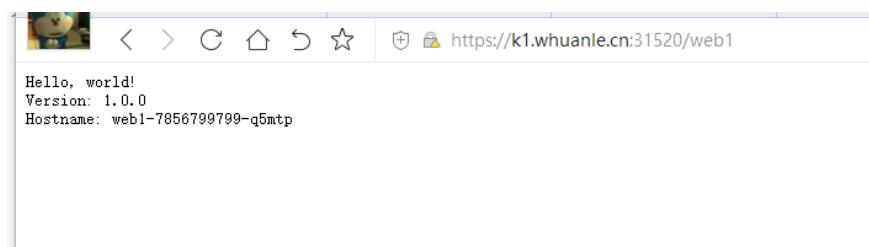
将证书上传到服务器, 创建 secret:

```
kubectl create secret tls tls-secret --cert=1_k1.whuanle.cn_bundle.crt --key=2_k1.whua
```

重新创建 Ingress, 并配置 TLS, Ingress 的 YAML 文件如下:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - k1.whuanle.cn
    secretName: tls-secret
  rules:
  - host: k1.whuanle.cn
    http:
      paths:
      - path: /web1
        pathType: Prefix
        backend:
          service:
            name: web1
            port:
              number: 8080
      - path: /web2
        pathType: Prefix
        backend:
          service:
            name: web2
            port:
              number: 9090
```

通过 <https://> 访问域名：



[Info] 提示

注意，由于没有公网负载均衡器，因此需要使用一个节点做入口，其方式是 Service，所以 80, 443 端口用不了，笔者的 Service 使用

`80:32587/TCP,443:31520/TCP` 代替访问。当然，Service 的端口是可以手动修改的，默认端口范围是 31000-32000，你可以修改配置，使其绑定 80 和 443 端口。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-09 20:46:22

- 4.4 服务发现
 - proxy
 - kubectl proxy
 - 服务发现
 - Service
 - 通过环境变量发现 Service
 - DNS/CoreDNS

4.4 服务发现

服务发现、服务注册是微服务中必不可少的东西，Kubernetes 本身也提供了简单的服务发现，本章介绍如何通过 Service、DNS 等，对外暴露服务。

proxy

kubectl proxy

kubectl proxy 可以帮我们映射一个端口到节点中，主要用处是创建类似 Service NodePort 的访问服务。

例如 kubectl proxy 可在 localhost 和 Kubernetes API server 之间创建代理服务器或应用程序级网关，方便我们访问 API Server。

执行命令：

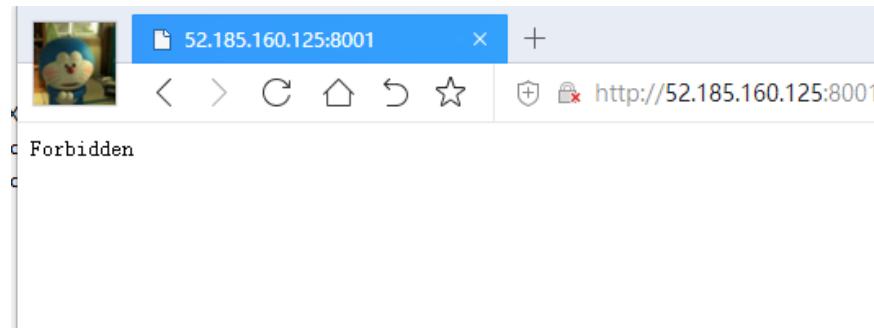
```
kubectl proxy
# 或指定访问端口，端口默认为 8001
kubectl proxy --port=8001
```

在本机上执行访问 `http://127.0.0.1:8001` 即可访问 API Server。

如果我们想在外网访问 API Server，可以使用 `--address`。

```
kubectl proxy --address=0.0.0.0 --port=8001
```

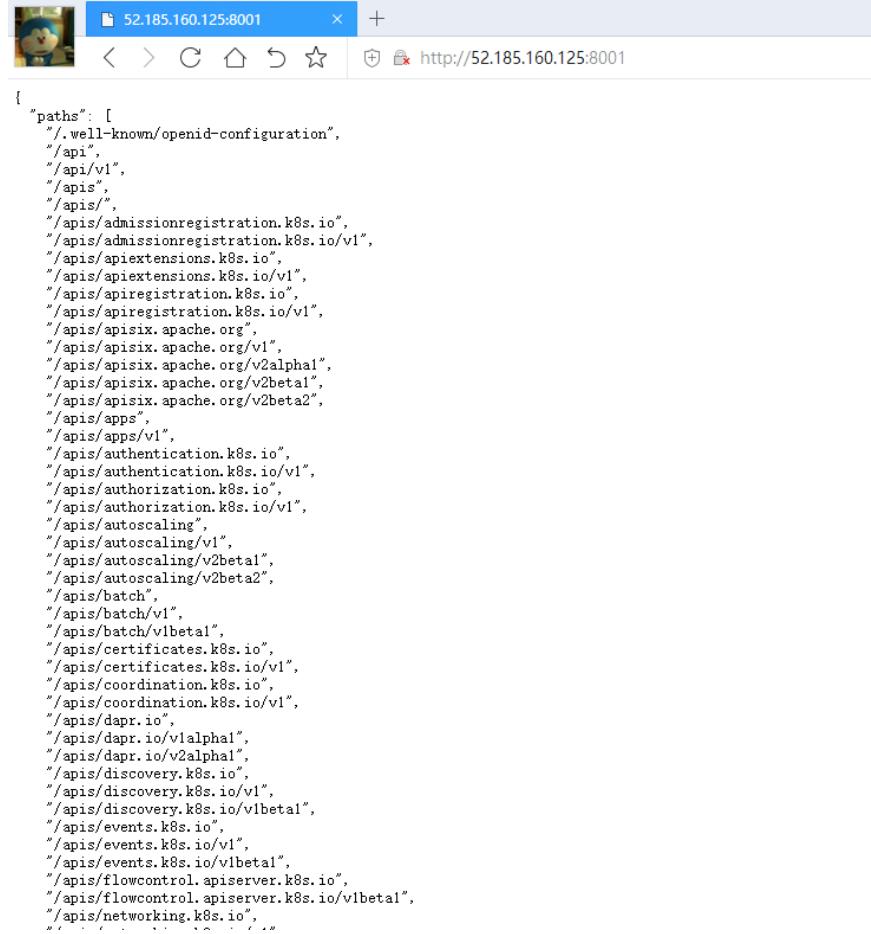
如果要在外网访问，需要添加头部认证信息，否则显示“Forbidden 403”。这跟 API Server 有关，如果要映射别的服务，则不需要这样。



如果我们想允许所有 IP 访问，又不需要认证，则可以使用：

```
kubectl proxy --address='0.0.0.0' --accept-hosts='^*$'
```

可以利用这种方式为 **Api Server** 暴露公网服务，然后使用 `kubeadm join` 通过公网 IP 将节点加入到集群中。



服务发现

Service

利用常规的 **Service**、**Endpoint**，也可以做服务发现，因为我们只需要知道一个固定的 IP 即可，而不需要关心背后是谁在提供服务。通过创建 **Service**，可以获取一个单一的稳定的 IP，只要 **Service** 存在，其 IP 便不会变化，其背后的 Pod 可能增加或减少，但始终可以通过固定的 **Service IP** 进行访问。

一般，我们创建 **Service**，然后获得 **Cluster IP**，接着将 IP 写入程序的配置文件中。在下面这个 **Service** 中，我们能够获得其的 **Cluster IP** 和暴露的 8080 端口。

```
metadata:  
  labels:  
    app: web1  
    name: web1  
spec:  
  clusterIP: 10.104.102.51  
  ports:  
    port: 8080  
  selector:  
    app: web1
```

但是这样有个缺点，需要提前将 IP 写到配置文件中。

在 [1.2章的容器化应用](#) 中，介绍了一个容器化应用应当遵从哪些设计原则，对于需要使用的远程服务，在配置文件中提前配置 IP，不是一个好做法。如果 IP 通过配置文件记录，那么一旦打包好镜像，其配置文件无法再改变，除非更新版本，难道要为了一个配置项，更新程序版本？或者手动进入容器中修改配置文件，这样每个 Pod 都要手工进入容器，修改配置文件(这是一种极其愚蠢的做法，例如我上家公司)。

在 [4.2 Endpoint](#) 一章中，我们学会了如何解耦 Service 跟 Endpoint、Pod，在这里，我们将利用这个解耦，实现简单的服务发现。

通过环境变量发现 Service

我们有一个应用名为 my-app，需要访问 mysql 服务，那么我们该如何为 my-app 配置 mysql 服务的 IP 地址？

创建 Deployment，部署两个应用。

```
kubectl create deployment my-app --image=nginx:latest --replicas=1  
kubectl create deployment mysql --image=mysql:latest --replicas=1
```

创建 Service，为两个应用暴露端口：

```
kubectl expose deployment my-app --port=6666 --target-port=80  
kubectl expose deployment mysql --port=3306 --target-port=3306
```

获取 Service 列表：

```
root@instance-2:~# kubectl get services  
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE  
my-app    ClusterIP  10.109.136.115  <none>          6666/TCP      4m16s  
mysql     ClusterIP  10.97.129.201   <none>          3306/TCP      7s
```

由于在创建 Service 之前，已经存在 Pod，这个 Pod 中不会有 Service 的环境变量，所以我们需要先删除这个旧的 Pod。

获取 Pod 列表，并删除由 my-app 创建的 Pod。

```
root@instance-2:~# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
my-app-d57587579-glh22  1/1     Running   0          37s

root@instance-2:~# kubectl delete pod my-app-d57587579-glh22
pod "my-app-d57587579-glh22" deleted
```

接着，**ReplicaSet** 会为我们重新创建一个 Pod，我们查看这个 Pod 的名称，然后进入 Pod 中的容器执行命令。

```
kubectl exec my-app-d57587579-rdsf7 my-app -- env
```

```
HOSTNAME=my-app-d57587579-wt5s8
MY_APP_SERVICE_HOST=10.109.136.115
MY_APP_SERVICE_PORT=6666

MYSQL_SERVICE_HOST=10.97.129.201
MYSQL_SERVICE_PORT=3306
```

所有环境变量名为大写字母，名称不支持特殊符号，例如 `my-app` 会被转为 `MY_APP`。

在 `*_SERVICE_HOST` 和 `*_SERVICE_PORT` 中，保存有 集群中所有 Service 的 IP，在 `my-app` 的 Pod 中，我们可以通过 `MYSQL_SERVICE_HOST` 环境变量获取 `mysql` 服务的 IP 地址，这便是一种服务发现。

在 Pod 中，可以使用 `env` 命令查看环境变量，里面有当前命名空间的 Service 信息。

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin  
HOSTNAME=web2-5b669f8984-szfzk  
WEB2_SERVICE_HOST=10.105.184.142  
WEB2_PORT_9090_TCP_ADDR=10.105.184.142  
WEB2_PORT_9090_TCP_PROTO=tcp  
WEB1_SERVICE_PORT_HTTP=8080  
WEB3_PORT_8080_TCP_PROTO=tcp  
KUBERNETES_SERVICE_PORT_HTTPS=443  
WEB2_SERVICE_PORT_HTTP=9090  
WEB3_SERVICE_HOST=10.107.23.1  
WEB1_PORT=tcp://10.100.137.80:8080  
WEB1_PORT_8080_TCP_PROTO=tcp  
WEB1_PORT_8080_TCP_ADDR=10.100.137.80  
KUBERNETES_PORT=tcp://10.96.0.1:443  
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1  
WEB3_PORT=tcp://10.107.23.1:8080  
WEB3_PORT_8080_TCP_PORT=8080  
KUBERNETES_PORT_443_TCP_PORT=443  
WEB2_SERVICE_PORT=9090  
KUBERNETES_SERVICE_HOST=10.96.0.1  
WEB1_PORT_8080_TCP=tcp://10.100.137.80:8080  
WEB2_PORT=tcp://10.105.184.142:9090  
WEB1_SERVICE_HOST=10.100.137.80  
WEB1_SERVICE_PORT=8080  
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443  
KUBERNETES_PORT_443_TCP_PROTO=tcp  
WEB2_PORT_9090_TCP=tcp://10.105.184.142:9090  
WEB2_PORT_9090_TCP_PORT=9090  
WEB3_PORT_8080_TCP_ADDR=10.107.23.1  
WEB1_PORT_8080_TCP_PORT=8080  
KUBERNETES_SERVICE_PORT=443  
WEB3_SERVICE_PORT=8080  
WEB3_PORT_8080_TCP=tcp://10.107.23.1:8080  
PORT=8080
```

DNS/CoreDNS

在 kube-system 中，运行着一个 DNS 服务器，其名称为 coredns，在 Kubernetes 中，每个 Service 都会被赋予一个 DNS 名称(就像域名)，每当有新增的 Service 时，coredns 便会为此 Service 添加 DNS 记录。

```
kubectl get deployment -n kube-system
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
calico-kube-controllers	1/1	1	1	13d
coredns	2/2	2	2	13d

继续上一个小节中的内容，我们部署了 my-app 和 mysql，我们查看这两个 Service 的 DNS 名称：

```
nslookup my-app.default  
nslookup mysql.default
```

。就像环境变量一样，每个 Pod 中的 `/etc/resolv.conf` 文件中，也会保存所有 Service 的 DNS 名称。

执行命令查询 DNS 列表：

```
kubectl exec my-app-d57587579-rdsf7 my-app -- cat /etc/resolv.conf
```

不过，Pod 中的 `/etc/resolv.conf` 只能看到相同命名空间下的其它 Service。

按照官网文档，如果正常的话，查询结果类似：

```
Server: 10.0.0.10
Address 1: 10.0.0.10

Name: kubernetes.default
Address 1: 10.0.0.1
```

如果输出跟笔者的一样，提示 `server can't find my-app.default: NXDOMAIN`，说明 coredns 有问题。

```
root@instance-2:~# nslookup my-app.default
Server: 127.0.0.53
Address: 127.0.0.53#53

** server can't find my-app.default: NXDOMAIN

root@instance-2:~# nslookup mysql.default
Server: 127.0.0.53
Address: 127.0.0.53#53

** server can't find mysql.default: NXDOMAIN
```

由于这个问题很麻烦，笔者暂时没能解决这个问题，因此这里就不再讲述了，读者可自行参考资料解决问题。

<https://github.com/kubernetes/dns/blob/master/docs/specification.md>

<https://kubernetes.io/zh/docs/tasks/administer-cluster/dns-debugging-resolution/>

<https://github.com/aws/containers-roadmap/issues/1115>

https://github.com/coredns/deployment/blob/master/kubernetes/Upgrading_CoreDNS.md

通过 Headless 发现服务，也是使用这种方式。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-08 20:00:53

- [5.volumes](#)
 - [Downward API介绍](#)

5.volumes

存储卷、向容器写入内容、访问数据、etcd

了解卷；

学会创建 emptydir、Hostpath、ntfs 卷。

理解并创建持久卷(PV)。

配置持久卷声明(PVC)。

管理卷访问方式。

部署一个能够访问持久存储的应用程序。

讨论存储的动态配置。

配置 Secret 和 ConfigMaps

环境变量和数据传递

5.1 使用secret管理密码

5.2 以卷的方式引用密码，传递配置文件

5.3 以变量的方式引用密码

5.4 使用configmap管理密码

Downward API介绍

作用：让Pod里的容器能够直接获取到这个Pod API对象本身的信息

etcd, etcd 快照与恢复

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新

时间：2021-11-08 20:01:21

- 5.1 卷
 - 卷
 - Docker 卷
 - hostPath 卷
 - emptyDir 卷
 - Git 卷

5.1 卷

由于容器最初设计为临时性和无状态的，因此几乎不需要解决存储持久性问题。然而，随着越来越多需要从持久性存储读写的应用程序被容器化，对持久性存储卷的访问需求也随之出现，于是 Docker 出现了卷。Docker 中的卷，实际上就是将主机目录映射到容器中。

但是，在多节点集群中，Pod/容器 在不同的节点服务器中运行，而 Docker 中的卷只能映射一个服务器中的目录/文件，不同服务器间不能共享文件，那么不同的 Docker 是无法访问同一份文件的。

为了让散落在不同服务器中的同一类容器访问同一份文件，Kubernetes 设计了有持久化的卷。Kubernetes 的卷独特之处在于它们是集群外部的，可以将持久卷挂载到集群，而不需要将它们与特定节点、容器或 Pod 关联。

采用容器存储接口(Container Storage Interface，CSI) 使用于容器编排的行业标准接口的目标能够允许访问任意存储系统，所以 Kubernetes 不仅支持 Docker，还可以支持多种容器引擎。在 Kubernetes 中，支持的卷类型非常多，而且很抽象，不同的容器技术之间的实现有很大差异，但是只要支持 CSI 接口，则 Kubernetes 可以很容易为这些容器引擎建立统一的卷文件系统。

接下来笔者将介绍 Kubernetes 中一些不同类型的卷。

卷

Docker 卷

在 Docker 中，我们可以使用以下命令管理卷。

```
# 创建自定义容器卷
docker volume create {卷名称}
```

```
# 查看所有容器卷
docker volume ls
```

```
# 查看指定容器卷的详细信息
docker volume inspect {卷名称}
```

我们可以在运行容器时，使用 `-v` 映射主机目录，或者映射容器卷到容器中。

```
docker -itd ... -v /var/tmp:/opt/app ...
docker -itd ... -v {卷名}:/opt/app ...
```

例如：

```
docker run -v /opt/test:/opt/test -it ubuntu bash
docker run --read-only -v /opt/test:/opt/test -it ubuntu bash
```

在 Docker 中，卷是一个目录，Docker 可以通过参数指定不同类型的挂载方式：

参数/命令	说明
--volume , -v	绑定挂载卷
--volume-driver	容器的可选卷驱动程序
--volumes-from	从指定的容器挂载卷，容器间可传递共享

Docker 中卷没有生命周期管理。对于纯 Docker 容器来说，只要宿主机的目录存在，那么容器创建和销毁时，不会删除宿主机的文件，但 Docker 的卷只有少量且松散的管理。如果不挂载卷，在 Docker 中，创建新版本的容器后，旧容器的数据会丢失。

hostPath 卷

`hostPath` 卷能将主机节点文件系统上的文件或目录挂载到 Pod 中，类似 Docker 的 `-v` 挂载。`hostPath` 卷依赖于节点上的目录或文件，不同节点的 Pod 无法共享相同的文件内容。

一般考虑 `hostPath` 在以下情况下使用：

- 单个 Pod，部署在一个节点上；
- 多个 Pod，但是都部署在一个节点上，一般会为 Pod 加上节点选择器，确保 Pod 分配到需要的节点上。

... ...

一个 `hostPath` 卷映射主机目录的配置示例如下：

```
volumes:
- name: test-volume
  hostPath:
    # 宿主上目录位置
    path: /data
    # 此字段为可选
    type: Directory
```

这段 YAML 配置了一个名称为 `test-volume` 的卷，映射类型是目录，位置是 `/data`。

`hostPath` 的 `type`，有以下类型：

取值	行为
	空字符串（默认）用于向后兼容，这意味着在安装 <code>hostPath</code> 卷之前不会执行任何检查。
<code>DirectoryOrCreate</code>	如果在给定路径上什么都没有存在，那么将根据需要创建空目录，权限设置为 <code>0755</code> ，具有与 <code>kubelet</code> 相同的组和属主信息。
<code>Directory</code>	在给定路径上必须存在的目录。
<code>FileOrCreate</code>	如果在给定路径上什么都没有存在，那么将在那里根据需要创建空文件，权限设置为 <code>0644</code> ，具有与 <code>kubelet</code> 相同的组和所有权。
<code>File</code>	在给定路径上必须存在的文件。
<code>Socket</code>	在给定路径上必须存在的 UNIX 套接字。
<code>CharDevice</code>	在给定路径上必须存在的字符设备。
<code>BlockDevice</code>	在给定路径上必须存在的块设备。

`hostPath` 的配置比较简单，这里就不做过多介绍。`Pod` 中直接定义 `hostPath` 的方式，其 `YAML` 模板如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
spec:
  containers:
  - image: nginx:latest
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # 宿主上目录位置
      path: /data
      # 此字段为可选
      type: DirectoryOrCreate
```

由于 `hostPath` 是在节点上的，因此使用单个 `Pod` 部署比较好。对于 `Deployment` 等多 `Pod` 的对象，`Pod` 分配在不同节点上，使用 `hostPath`，可能会导致不能正常工作。

emptyDir 卷

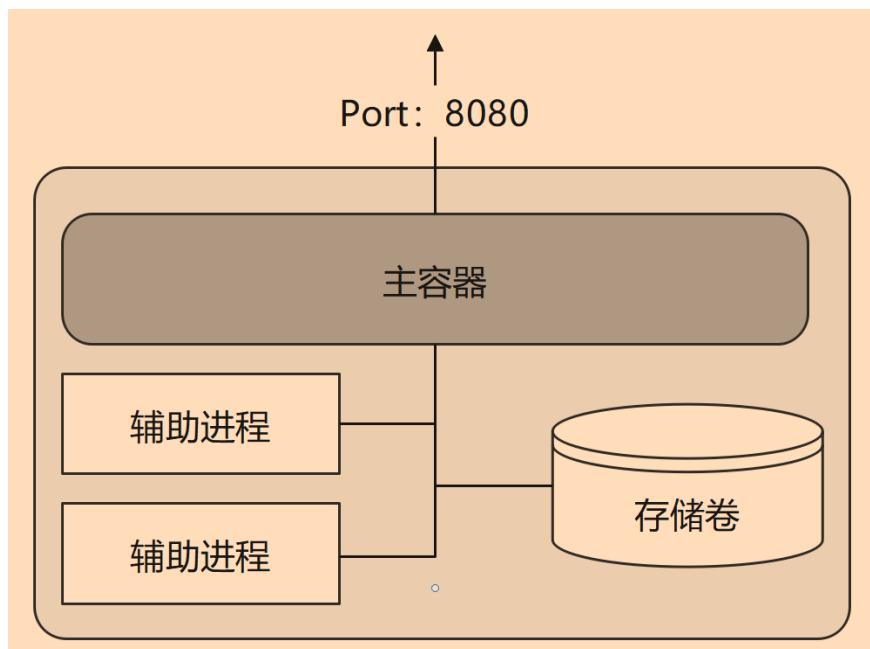
`emptyDir` 卷是最简单的一种卷，首先，它是一个空的卷，意味着什么都没有，并且它是临时的，其生命周期与 `Pod` 绑定。但是可以在同 `Pod` 的不同容器中访问同一个 `emptyDir`。

我们为 `Pod` 配置一个 `emptyDir` 卷并开始创建 `Pod`，当 `Node` 创建 `Pod` 后，会自动创建 `emptyDir`，并且在 `Pod` 运行期间这个 `emptyDir` 卷一直存在，但是如果 `Pod` 被删除，则 `emptyDir` 卷会丢失。

说明：容器崩溃并不会导致 Pod 被从节点上移除，因此容器崩溃期间 `emptyDir` 卷中的数据是安全的。

当 `emptyDir` 卷创建完成后，Pod 中的容器便可以输出文件到卷中。

`emptyDir` 卷适合用于 Pod 中的容器共享文件，例如前后端容器，前端容器是辅助进程，不对外提供任何服务，前端文件通过 `emptyDir` 共享到 Web 的 `wwwroot` 目录中。由 Web 对外提供 8080 端口服务，用户访问 Web 时，可以访问到前端静态文件。这样前端可以分开更新和部署，存放前端文件的辅助容器只负责提供静态文件，最终由 Web 后端程序对外提供静态页面和 API。



Deployment 模板示例如下：

```

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-web
  name: my-web
spec:
  selector:
    matchLabels:
      app: my-web
  strategy:
  template:
    metadata:
      labels:
        app: my-web
    spec:
      containers:
        - image: web:1.e
          name: web
          ports:
            - containerPort: 8080
          volumeMounts:
            - mountPath: /opt/web/root
              name: wwwroot-volume
        - image: frontend:1.2
          name: frontend
          volumeMounts:
            - mountPath: /opt/frontend/root
              name: wwwroot-volume
      volumes:
        - name: wwwroot-volume
      emptyDir: {}

```

使用 Pod 配置 单容器，其简单的示例如下：

```

apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: nginx:latest
      name: test-container
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}

```

`emptyDir` 不需要指定宿主机上对应的目录，Kubernetes 会为其指定临时目录，`emptyDir` 卷存储介质跟主机的存储介质有关，一般是磁盘、固态/机械硬盘、网络存储等，也可以是内存。在 Linux 中有一种 `tmpfs`（基于 RAM 的文件系统）文件系统，其速度比磁盘快，适合用作缓存，将 `emptyDir.medium` 字段设置为 "`Memory`" 即可，所写入的所有文件都会计入容器的内存消耗，受容器内存限制约束，如果机器重启，文件会丢失。

`emptyDir` 的一些用途：

- 缓存空间，例如基于磁盘的归并排序算法。

- 为耗时较长的计算任务提供检查点，以便任务能方便地从崩溃前状态恢复执行。
- 在 Web 服务器容器服务数据时，保存内容管理器容器获取的文件。

Git 卷

Git 卷也是一种 `emptyDir` 卷，顾名思义，Git 卷可以帮助我们拉取 Git 中的代码到 Pod 中。在最初时，映射到 Pod 中的目录是空的，当 Pod 启动起来时，便会从 Git 中拉取源代码到 `emptyDir` 卷中，填充存储空间。

Git 卷适合用来做编译，启动 Pod 克隆仓库后，容器中的程序读取源代码，然后编译项目，编译后的文件可以部署运行，也可以打包发布。

很多人应该都了解过 Jenkins，通过流水线构建自动化发布，其实 Kubernetes 中的 Git 卷也可以做到类似效果，只是定制不太灵活。

不过 Git 卷的缺点是只在 Pod 启动时拉取一次代码，之后仓库中的代码提交多少次，都不会同步到你的卷中。不过每个创建的 Pod，都会拉取最新的源代码。

Git 卷的 `volumes` 部分，其定义模板如下：

```
volumes:  
- name: erp  
  gitRepo:  
    repository: https://github.com/whuanle/CZGL.AOP  
    revision: main  
    directory: .
```

`directory` 配置拉取的源代码放到卷的什么目录中，使用 `.` 表示放到卷的根目录中。

以上四种卷是比较常用的，还有一些卷在后面的章节中会接着讲解。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间：2021-11-08 20:26:37

- 5.2 ConfigMap 和 secret
 - ConfigMap
 - 创建 configMap
 - 在环境变量中使用
 - ConfigMap 卷
 - Secret
 - TLS
 - 基本身份认证 Secret
 - SSH 身份认证 Secret
 - 使用 Secret

5.2 ConfigMap 和 secret

ConfigMap 可以用来存储非机密性的数据到键值对中，这些信息会被存储到 etcd，不会分节点，任何节点都可以使用到。使用时，将其用作环境变量、命令行参数或者存储卷中的配置文件送入到 Pod 中，主要目的是解耦你的应用程序和配置，这样不必维护那些 .json 等配置文件，也可以避免不小心将带有机密信息的配置文件上传到代码仓库中。但是 ConfigMap 中的内容都是非加密的，可以很容易地看到全部信息，如果需要加密，则使用 secret。

ConfigMap

创建 configMap

创建 configMap 的命令格式：

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



```
kubectl create configmap NAME [--from-file=[key=]source] [--from-literal=key1=value1]
```

如果以文件(`--from-file`)形式创建 ConfigMap，则为 key，文件内容为 value，`--from-file` 也可以指定目录。

如果以键值对(`--from-literal`)形式创建 ConfigMap，可直接创建 `key=value`。

可以根据文件内容、目录或键值对创建 ConfigMap，ConfigMap 可以指定多个来源。当基于目录创建一个 ConfigMap 时，每个文件的基名是该目录中的有效密钥，这些文件将被打包到 ConfigMap 中。

这三点都是从哪里生成 k-v 到 ConfigMap，不要纠结于字面。

直接使用键值对创建 ConfigMap：

```
kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=co
```

查看 ConfigMap：

```
root@master:~# kubectl get configmap my-config
Data
=====
key1:
-----
config1
key2:
-----
config2
```

--from-literal 表示，当前使用键值对赋值的形式创建 ConfigMap。

通过文件内容生成 ConfigMap，文件内容示例：

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

设置文件名称为 c.txt。

文件中的内容使用键值对形式。

创建 ConfigMap：

```
kubectl create configmap my-config1 --from-file=c.txt
```

查看 ConfigMap：

```
root@master:~# kubectl describe configmap my-config1
Data
=====
c.txt:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

多个文件：

```
kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --fr
```

通过目录创建 ConfigMap，创建目录和文件，其目录结构如下：

```
|   config
|   |   a.txt
|   |   b.txt
```

```
root@master:~# cat config/a.txt  
a=666  
root@master:~# cat config/b.txt  
b=666
```

创建 ConfigMap:

```
kubectl create configmap my-config2 --from-file=config/
```

查看 ConfigMap:

```
root@master:~# kubectl describe configmap my-config2  
Data  
=====  
a.txt:  
----  
a=666  
  
b.txt:  
----  
b=666
```

注意，通过文件或目录创建的 ConfigMap 中，文件名称为 Key，而文件内容为 Value，所以 `a=666` 是文件的值，里面不能再分为 Key a、Value B，`a=666` 是一个值。

或者改成下面这样会更容易理解：

```
=====  
name.txt:  
----  
痴者工良  
  
url.txt:  
----  
k8s.whuanle.cn
```

下面将介绍怎么使用 ConfigMap。

在环境变量中使用

前面我们已经创建了 `my-config`、`my-config1`、`my-config2` 三个配置，我们希望这些配置，能够以环境变量的形式传递到容器中，那么定义 Pod 时其 YAML 模板如下：

```
env:  
- name:letter  
  valueFrom:  
    configMapKeyRef:  
      name: my-config2  
      key: a.txt
```

注：key 为 | 之前的名称，例如 `a.txt`，文件名；

或者：

```
envFrom:  
- configMapRef:  
  name: my-config2
```

如果要整个 configMap 的内容全部导入，则使用 `envFrom`，如果要只使用一部分，可以用 `valueFrom`。

那么我们来真实启动一个 Nginx Pod：

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx  
spec:  
  containers:  
    - name: nginx  
      image: nginx:latest  
      env:  
        - name: myconfig_a  
          valueFrom:  
            configMapKeyRef:  
              name: my-config2  
              key: a.txt  
        - name: myconfig_b  
          valueFrom:  
            configMapKeyRef:  
              name: my-config2  
              key: b.txt
```

[Info] 提示

如果 ConfigMap 通过文件或目录创建的，那么每个文件的环境变量是打包一起的。所以：

```
configMapKeyRef:  
  name: my-config2  
  key: a.txt
```

a.txt 中的环境变量映射到 myconfig_a。

```
kubectl apply -f configMapNginx.yaml
```

打印 Pod 中的环境变量：

```
root@master:~# kubectl exec nginx -- printenv  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
HOSTNAME=nginx  
myconfig_a=a=666  
  
myconfig_b=b=666
```

不过这样映射会有的奇怪，因此使用环境变量使用 ConfigMap 时，都是使用 K/V 的数据，而不使用通过文件、目录的数据。

ConfigMap 卷

`configMap` 卷提供了向 Pod 注入配置数据的方法。其主要用途是给 Pod 中的容器传递配置，在 Pod 中显示为文件。

Pod 的 YAML 文件示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: configmap-pod
      image: busybox
      command: ["ls"]
      args: ["/etc/config"]
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: my-config
```

查看此 Pod 的日志：

```
root@master:~# kubectl logs configmap-pod
key1
key2
```

key1 文件中存储了值，ConfigMap my-config 中有 key1=config1、key2=config2 两个配置，则在 key1 文件中，其值为 config1。

但是这样好像没啥意思，既然要映射为文件，应该创建有意思的内容。

下面创建一个 config.json 文件：

```
{
  "name": "痴者工良",
  "url": "k8s.whuanle.cn"
}
```

创建 ConfigMap：

```
kubectl create configmap my-config3 --from-file=config.json
```

```
root@master:~# kubectl describe configmap my-config3
Name:           my-config3
config.json:
-----
{
  "name": "痴者工良",
  "url": "k8s.whuanle.cn"
}
```

映射为 Pod 中的文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: configmap-pod
      image: busybox
      command: ["cat"]
      args: ["/etc/config/config.json"]
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: my-config3
```

查看容器中被映射的文件：

```
root@master:~# kubectl logs configmap-pod
{
  "name": "痴者工良",
  "url": "k8s.whuanle.cn"
}
# cat /etc/config/config.json
```

[Success] 提示

通过 ConfigMap 卷映射到容器中，如果使用的是 `mountPath` 挂载到目录，则 ConfigMap 更新内容时，容器中的配置文件内容也会被更新，但是可能需要几分钟。如果挂载的目录是容器中已经存在的，则可能无法同步更新，因此挂载 ConfigMap 到容器中的时，需要使用一个新的目录。

Secret

Secret 卷用来给 Pod 传递敏感信息，例如密码、密钥等。**Secret**卷实际上不是用于存储的，**Secret** 中存储的信息，会以环境变量、文件等形式显示在 Pod 中。在 4.3 章的 Ingress 中，就讲解过使用 Secret 为 Ingress 增加 TLS 加密访问(HTTPS)。

Secret 的信息存储在 etcd 中，但是 **Secret** 一般情况下也不是加密存储的，关于如何加密 **Secret** 后面的章节会介绍到。

说明： 使用前你必须在 Kubernetes API 中创建 Secret。

Secret 的类型很多，这里将官方文档中列举的 **secret** 类型复制一份：

内置类型	用法
Opaque	用户定义的任意数据
kubernetes.io/service-account-token	服务账号令牌
kubernetes.io/dockercfg	~/.dockercfg 文件的序列化形式
kubernetes.io/dockerconfigjson	~/.docker/config.json 文件的序列化形式
kubernetes.io/basic-auth	用于基本身份认证的凭据
kubernetes.io/ssh-auth	用于 SSH 身份认证的凭据
kubernetes.io/tls	用于 TLS 客户端或者服务器端的数据
bootstrap.kubernetes.io/token	启动引导令牌数据

这里只讲解一部分常用的 `Secret` 类型，如需了解更多详细知识，请参考官方文档：
<https://kubernetes.io/zh/docs/concepts/configuration/secret/>

TLS

通过证书创建 `Secret` 的命令如下所示：

```
kubectl create secret tls tls-secret --cert=1_k1.whuanle.cn_bundle.crt --key=2_k1.whuanle.cn.key
```

使用 YAML 表示：

```
apiVersion: v1
data:
  tls.crt: ...
  tls.key: ...
kind: Secret
metadata:
  name: tls-secret
type: kubernetes.io/tls
```

基本身份认证 `Secret`

其类型为 `kubernetes.io/basic-auth`，它是用来存储账号密码的，`spec.data` 中必须包含 `username` 和 `password` 两个字段，其 YAML 文件示例如下所示：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
```

如果我们查看数据：

```
root@master:~# kubectl get secret secret-basic-auth -o yaml
apiVersion: v1
data:
password: dDBwLVNlY3J1dA==
username: YWRtaW4=
kind: Secret
... ...
```

`username` 和 `password` 等会被使用 `base64` 编码，但是并不是加密屏蔽，这些信息可以被直接用 `base64` 还原。

SSH 身份认证 Secret

其类型是 `kubernetes.io/ssh-auth`，用来存储一段 `ssh` 密钥。

其 YAML 示例如下：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # 此例中的实际数据被截断
  ssh-privatekey: | 
    MIIEpQIBAAKCAQEAv1qb/Y ...
```

使用 Secret

前面已经介绍过 `Ingress` 使用 `Secret`，这里就不再复述。

在 `Pod` 中使用 `Secret` 有环境变量和文件两种形式，将 `Secret` 挂载到 `Pod` 的目录下，

```
apiVersion: v1
kind: Pod
metadata:
  name: secret1
spec:
  containers:
  - name: secret1
    image: nginx
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: secret-basic-auth
```

这里使用之前创建的基于身份认证的 `Secret`，有 `username`、`password` 两个字段。

创建 `Pod`，然后查看映射到 `Pod` 的文件：

```
root@master:~# kubectl exec secret1 -- ls /etc/foo
password
username

root@master:~# kubectl exec secret1 -- cat /etc/foo/username
adminroot
@master:~# kubectl exec secret1 -- cat /etc/foo/password
t0p-Secret
```

可以看到，**Secret** 中每一个 **字段/Key**，会生成一个文件名，**Value** 是文件内容。

如果以环境变量的方式使用 **Secret**，其 **YAML** 跟 **ConfigMap** 类似，文件示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: mysecret2
spec:
  containers:
    - name: mysecret
      image: nginx
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: secret-basic-auth
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret-basic-auth
              key: password
  restartPolicy: Never
```

会将 **secret-basic-auth** 中的 **username** 和 **password** 两个字段映射到 Pod 的 **SECRET_USERNAME** 、 **SECRET_PASSWORD** 两个环境变量中。

查看 Pod 中的环境变量：

```
root@master:~# kubectl exec mysecret2 -- env
SECRET_USERNAME=admin
SECRET_PASSWORD=t0p-Secret
```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新时间： 2021-11-09 21:20:17

- 5.3 NFS卷
 - 搭建 NFS 服务
 - nfs 服务
 - nfs 客户端
 - 挂载到 Pod

5.3 NFS卷

NFS 卷，可以把它想象成云硬盘，随时可以挂载到服务器的任意目录，而且多个服务器可以同时使用这个卷。



在前面，我们学习了 `emptyDir`、`hostPath` 等卷，这些卷都有各自的应用场景，创建也很方便。但是这些卷都不能为多节点中的 Pod 提供存储空间，当 Pod 的副本在不同节点中运行时，无法通过这些卷为所有的 Pod 提供一致的文件内容；后续如果程序修改了卷中的文件，也无法同步操作到所有 Pod 中。

NFS 是一种网络文件系统，英文 Network File System(NFS)，能使使用者访问网络上别处的文件就像在使用自己的计算机一样，NFS 可以独立于集群中的节点，其存储空间可以在集群之外，然后通过网络为多个节点共享文件。不同的节点可以通过远程存取、操作文件，同时文件的状态为所有节点共享，保证每个节点上访问到的文件内容、状态都是一致的、实时的。

NFS 是基于 UDP/IP 协议的应用，其实现是采用远程过程调用 RPC 机制，NFS 提供了一组与机器、操作系统以及低层传送协议无关的存取远程文件的操作。

以某云的云硬盘为例，了解一下它。

什么是腾讯云云硬盘

云硬盘 (Cloud Block Storage, CBS) 是一种高可用、高可靠、低成本、可定制化的块存储设备，可以作为云服务器的独立可扩展硬盘使用，为云服务器实例提供高效可靠的 存储 设备。云硬盘提供数据块级别的持久性存储，通常用作需要频繁更新、细粒度更新的数据（如文件系统、数据库等）的主存储设备，具有高可用、高可靠和高性能的特点。云硬盘采用三副本的分布式机制，将您的数据备份在不同的物理机上，避免单点故障引起的数据丢失等问题，提高数据的可靠性。

您可通过控制台轻松购买、调整以及管理您的云硬盘设备，并通过构建文件系统创建出高于单块云硬盘容量的存储空间。根据生命周期的不同，云硬盘可分为：

- 非弹性云硬盘的生命周期完全跟随云服务器，随云服务器一起购买并作为系统盘使用，不支持挂载与卸载。
- 弹性云硬盘的生命周期独立于云服务器，可单独购买然后手动挂载至云服务器，也可随云服务器一起购买并自动挂载至该云服务器，作为数据盘使用。弹性云硬盘支持随时在同一可用区内的云服务器上挂载或卸载。您可以将多块弹性云硬盘挂载至同一个云服务器，也可以将弹性云硬盘从云服务器 A 中卸载然后挂载到云服务器 B。

腾讯云对用户的云硬盘配额有相应的限制，详情请参考 [使用限制](#)。

典型使用场景

- 云服务器在使用过程中发现硬盘空间不够，可以通过购买一块或多块云硬盘挂载至云服务器上满足存储容量需求。
- 购买云服务器时不需要额外的存储空间，有存储需求时再通过购买云硬盘扩展云服务器的存储容量。
- 在多个云服务器之间存在数据交换的诉求时，可以通过卸载云硬盘（数据盘）并重新挂载到其他云服务器上实现。
- 可以通过购买多块云硬盘并配置 LVM (Logical Volume Manager) 逻辑卷来突破单块云硬盘存储容量上限。
- 可以通过购买多块云硬盘并配置 RAID (Redundant Array of Independent Disks) 策略来突破单块云硬盘 I/O 能力上限。

搭建 NFS 服务

nfs 服务

我们首先在一台服务器上，部署 NFS 服务，由于这个服务器对所有节点提供了文件系统，所以这台服务器的硬盘需要能够自动备份、宕机保护等，保证数据安全。

笔者建议是 master 节点创建 nfs 服务，worker 节点使用 nfs 服务。

在 Ubuntu 中安装 nfs 库，启动 nfs 服务。

```
sudo apt-get update && sudo apt-get install -y nfs-kernel-server
```

在服务器上创建要共享的目录。

```
mkdir /nfs-share
chmod 777 /nfs-share
# 随便填充一些文件
echo env > /nfs-share/env.txt
```

检查 nfs 服务是否已经正常安装：

```
rpcinfo -p localhost
```

导出配置：

```
echo "/nfs-share *(rw,no_root_squash,sync)" >> /etc/exports
```

使配置生效：

```
exportfs -r
```

检查是否已经生效:

```
exportfs
```

启动 rpcbind、nfs 服务:

```
systemctl restart rpcbind
systemctl enable rpcbind
systemctl restart nfs
systemctl enable nfs
```

接下来可以在本服务器上测试:

```
showmount -e $(hostname -i)
# 或者 showmount -e 127.0.0.1
```

```
Export list for 10.170.0.2:
/nfs-share *
```

nfs 客户端

在另一台服务器上安装 nfs 客户端。

```
apt install nfs-common
```

挂载 nfs 服务器的目录到本地中:

```
mount {部署了nfs服务器的ip}:/nfs-share /mnt
```

查看挂载的目录:

```
ls -lah /mnt
total 12K
drwxrwxrwx  2 root root 4.0K May  1 02:36 .
drwxr-xr-x 24 root root 4.0K May  1 02:35 ..
-rw-r--r--  1 root root    4 May  1 02:36 env.txt
```

说明我们的配置没有问题，现在是挂载到主机的目录。接下来我们将 NFS 卷挂载到 Pod 中。

挂载到 Pod

模板如下:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest
      volumeMounts:
        - name: mynfs
          mountPath: /mnt/mynfs
  volumes:
    - name: mynfs
      nfs:
        server: 10.170.0.2
        path: /nfs-share
```

```
kubectl apply -f nginx.yaml
```

查看容器中的 `/mnt/mynfs` 目录:

```
kubectl exec nginx -- ls /mnt/mynfs
```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-11-09 21:37:38

- 5.3 持久卷
 - 持久卷和持久卷声明
 - 创建远程存储空间
 - 创建卷
 - 创建 PVC
 - 在 Pod 中使用卷
 - StorageClass

5.3 持久卷

开门见山，PV、PVC 将存储卷分为实际卷和声明卷两部分，面向对象语言中将抽象部分写成接口，将实现部分写成类。大概过程是这样的，运维人员可以将实际存储空间创建为卷，即 PV，每个卷都有其空间容量、访问权限，这些卷创建后会提交给 Kubernetes，并记录下来。开发人员可以在程序中使用卷，但是开发人员不知道、也不能指定哪个卷可以使用，部署应用时，编写 PVC，定义了此程序需要多少容量的卷，接着会向 Kubernetes 发出请求要使用卷，例如需要 2GB 的存储空间，然后 Kubernetes 查找符合要求的 PV，并将 PVC 和 PV 绑定起来。PV 和 PVC 两个称为 持久卷和持久卷声明。

在本章中，需要根据上一章安装好的 NFS Server 和 Client 来做实验，请读者提前搭建好环境。

持久卷和持久卷声明

如果 Pod 本身没有什么存储要求，那么 Pod 做成无状态是很容易的，但是实际上 Pod 需要使用 Redis、Mysql 等存储数据，这是日常开发中必不可少的，如果有 N 个节点，1 个 Mysql 实例不够用了，需要部署 M 个 Mysql 服务，此时这 M 个 Mysql 实例，必须保证数据完全一致，否则短时间请求 Mysql 服务得到的数据可能不一样。不过问题也不大，因为 Pod 们使用了同一个 Mysql 集群，只要解决 Mysql 多节点的一致性问题，对 Pod 来说就可以保持一致性。

但是对于存储来说，如果没有方法像 Mysql 一样存储 Pod 的文件，那么 Pod 只能在节点的存储器中读写文件内容，而每个节点的文件系统和磁盘是不能共用，它们存储的内容不会自动合在一起，那么每个 Pod 就会产生差异性。例如用户上传文件到应用中，如果直接存储到本地时，这个文件在其他节点不存在，如果用户取文件时，访问的 Pod 不是上次上传文件的 Pod，那么便取不到文件。当然，可以用对象存储解决这个问题，也可以使用 NFS 或者各大云厂商的云硬盘产品。

持久卷的类型很多，这里笔者就不一一举例了，笔者从官方文档中截图如下：

持久卷的类型

PV 持久卷是用插件的形式来实现的。Kubernetes 目前支持以下插件：

- [awsElasticBlockStore](#) - AWS 弹性块存储 (EBS)
- [azureDisk](#) - Azure Disk
- [azureFile](#) - Azure File
- [cephfs](#) - CephFS volume
- [csi](#) - 容器存储接口 (CSI)
- [fc](#) - Fibre Channel (FC) 存储
- [flexVolume](#) - FlexVolume
- [gcePersistentDisk](#) - GCE 持久化盘
- [glusterfs](#) - Glusterfs 卷
- [hostPath](#) - HostPath 卷 (仅供单节点测试使用; 不适用于多节点集群; 请尝试使用 `local` 卷作为替代)
- [iscsi](#) - iSCSI (SCSI over IP) 存储
- [local](#) - 节点上挂载的本地存储设备
- [nfs](#) - 网络文件系统 (NFS) 存储
- [portworxVolume](#) - Portworx 卷
- [rbd](#) - Rados 块设备 (RBD) 卷
- [vsphereVolume](#) - vSphere VMDK 卷

以下的持久卷已被弃用。这意味着当前仍是支持的，但是 Kubernetes 将来的发行版会将其移除。

- [cinder](#) - Cinder (OpenStack 块存储) (于 v1.18 弃用)
- [flocker](#) - Flocker 存储 (于 v1.22 弃用)
- [quobyte](#) - Quobyte 卷 (于 v1.22 弃用)
- [storageos](#) - StorageOS 卷 (于 v1.22 弃用)

旧版本的 Kubernetes 仍支持这些“树内 (In-Tree) ”持久卷类型：

- [photonPersistentDisk](#) - Photon 控制器持久化盘。 (v1.15 之后不可用)
- [scaleIO](#) - ScaleIO 卷 (v1.21 之后不可用)

本章将介绍如果使用 NFS 搭建自己的分布式存储系统。

创建远程存储空间

这一步目的是创建云硬盘，能够为所有节点提供存储服务，即是分布式存储，所有节点所有 Pod 挂载同一个存储服务，所有 Pod 向同一个存储空间写入文件，与此同时读取的是最新的文件内容。

在上一章中的 NFS Server 服务器，创建一个 `/data/columns` 目录，这些目录存储空间由于创建卷。

创建 5 个共享目录：

```
mkdir /data
mkdir /data/columns
mkdir /data/columns/1
mkdir /data/columns/2
mkdir /data/columns/3
mkdir /data/columns/4
mkdir /data/columns/5
```

```
echo "/data/columns/1 *(rw,no_root_squash,sync)" >> /etc/exports
echo "/data/columns/2 *(rw,no_root_squash,sync)" >> /etc/exports
echo "/data/columns/3 *(rw,no_root_squash,sync)" >> /etc/exports
echo "/data/columns/4 *(rw,no_root_squash,sync)" >> /etc/exports
echo "/data/columns/5 *(rw,no_root_squash,sync)" >> /etc/exports
```

刷新 NFS 服务器共享目录信息：

```
exportfs -r
```

查看共享目录:

```
root@master:~# exportfs  
/nfs-share <world>  
/data/columns/1  
    <world>  
/data/columns/2  
    <world>  
/data/columns/3  
    <world>  
/data/columns/4  
    <world>  
/data/columns/5  
    <world>
```

创建卷

这一步将会创建持久卷 PersistentVolume (PV), 其 YAML 示例如下:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /data/volumns/1
    server: 10.0.0.4
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0002
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /data/volumns/2
    server: 10.0.0.4
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 4Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /data/volumns/3
    server: 10.0.0.4
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0004
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle

```

```
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /data/volumns/4
  server: 10.0.0.4
```

执行 `hostname -i` 命令获取 NFS Server 服务器的 IP。这里暂时不创建 pv 5。

创建 PV:

```
root@master:~# kubectl apply -f pv.yaml
persistentvolume/pv0001 created
persistentvolume/pv0002 created
persistentvolume/pv0003 created
persistentvolume/pv0004 created
```

查看 PV:

```
root@master:~# kubectl get pv
NAME      CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS      CLAIM      STORAGECLASS
pv0001    1Gi        RWO          Recycle        Available
pv0002    2Gi        RWO          Recycle        Available
pv0003    4Gi        RWO          Recycle        Available
pv0004    5Gi        RWO          Recycle        Available
```

每个 PV 可以设定三种 ACCESS MODES，即访问模式：

- `ReadWriteOnce` , `RWO`

卷可以被一个节点以读写方式挂载，即只允许被一个节点挂载。

- `ReadOnlyMany` , `ROX`

卷可以被多个节点以只读方式挂载。

- `ReadWriteMany` , `RWX`

卷可以被多个节点以读写方式挂载。

- `ReadWriteOncePod` , `RWOP`

卷可以被单个 Pod 以读写方式挂载。即只能有一个节点一个 Pod 挂载访问。

创建 PVC

PVC 的 YAML 示例如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc3
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

创建并查看 PVC 状态：

```
root@master:~# kubectl apply -f pvc1.yaml
persistentvolumeclaim/pvc3 created
root@master:~# kubectl get pvc
NAME      STATUS      VOLUME      CAPACITY      ACCESS MODES      STORAGECLASS      AGE
pvc3      Pending      pv0003      4Gi          RWO            manual           1s
```

```
root@master:~# kubectl get pvc
NAME      STATUS      VOLUME      CAPACITY      ACCESS MODES      STORAGECLASS      AGE
pvc3      Bound       pv0003      4Gi          RWO            default          5s
```

在刚创建 PVC 时，尚未绑定到 PV，此时 PVC 处于 Pending 状态，一旦找到符合要求的 PV，则 PVC 变成 Bound 状态。

刚开始我们创建 PVC 时，要求使用的是 3GB 存储空间，如果查找发现 PV 中没有刚好的容量，则会使用稍大的 PV 卷。

此时我们再查看 PV：

```
root@master:~# kubectl get pv pv0003
NAME      CAPACITY      ACCESS MODES      RECLAIM POLICY      STATUS      CLAIM      STORAGECLASS
pv0003   4Gi          RWO              Recycle           Bound      default/pvc3
```

可以看到，它正在被 pvc3 使用。

PV 有以下四种状态：

- **Available**（可用）尚未绑定到任何 PVC；
- **Bound**（已绑定）已被某 PVC 绑定；
- **Released**（已释放）PVC 已被删除，但 PV 还没有被回收；
- **Failed**（失败）卷的自动回收操作失败。

在 Pod 中使用卷

例如要在 Nginx Pod 中使用卷存储静态页面，将 pvc3 映射到 Pod 中，YAML 文件示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: pvcpod
spec:
  containers:
    - name: pvctest
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc3
```

笔者测试发现笔者使用的 Nginx 镜像默认文件位置是
`/usr/share/nginx/html`，有的可能是 `/var/www/html`。

读者部署 Pod 后，可以使用 `kubectl exec pvcpod -- cat /etc/nginx/conf.d/default.conf` 命令查看静态文件放在哪个位置。

创建 Pod 后，打开 NFS Server 的 `/data/columns/3` 目录，生成一个 `index.html` 文件：

```
echo '<h1 style="color:blue">痴者工良</h1>' > index.html
```

查看 Pod 的 IP：

```
root@master:/data/columns/3# kubectl get pod -o wide
NAME        READY   STATUS    RESTARTS   AGE     IP
pvcpod      1/1     Running   0          3m11s   10.32.
```

查看 Pod 中的文件：

```
root@master:/data/columns/3# kubectl exec pvcpod -- ls /var/www/html
index.html
root@master:/data/columns/3# kubectl exec pvcpod -- cat /var/www/html/index.html
<h1 style=color:blue>痴者工良</h1>
```

访问此 Pod：

```
root@master:~# curl 10.32.0.21
<h1 style=color:blue>痴者工良</h1>
```

StorageClass

PV 和 PVC 都是可以分类的，只有具有相同名称的 `storageClassName` 字段，PV 和 PVC 才能绑定起来。具有 `storageClassName` 字段的 PV 和 PVC 示例如下：

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0004
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: shop
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /data/columns/4
    server: 10.0.0.4

```

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc5
spec:
  storageClassName: shop
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi

```

```

root@master:~# kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM     STORAGECLASS
pv0004    5Gi        RWO          Recycle         Available   slow

```

当 PVC 设置了 `storageClassName` 时，如果没有相同 `storageClassName` 名称的 PV，那么 PVC 会一直处于 Pending 状态，其实早已失败：

```

Events:
  Type    Reason           Age           From            Message
  ----  -----   ----   -----
  Warning  ProvisioningFailed  4m37s (x42 over 14m)  persistentvolume-controller  stor

```

在 PV 的 `persistentVolumeReclaimPolicy` 字段可以设置当 PVC 被删除后，PV 如何处理，`persistentVolumeReclaimPolicy` 可以设置 `Retaine`(保留)、`Recycle`(回收)或 `Delete`(删除)三种类型。如果使用了 `Recycle` 类型，那么当 PVC 被删除后，此 PV 马上可被别的 PVC 使用。如果是 `Retaine` 类型，PV 会被保留下，但是不能被 PVC 绑定，此对象已经不可用，可以手动删除 PV 对象，当然，存储空间文件会保留。如果使用了 `Delete` 模式，则要尤其注意，因为删除了 PVC，PV 会被自动删除，并且，还可能会删除存储空间的文件，会不会删除存储空间的文件，要看 PV 的存储空间是什么类型的。

使用 `kubectl edit pv pv0003` 修改上面的 PV3 的 YAML 文件：

```
nfs:  
  path: /data/volumns/3  
  server: 10.0.0.4  
  persistentVolumeReclaimPolicy: Delete  
  volumeMode: Filesystem
```

删除 PVC:

```
root@master:~# kubectl delete pvc pvc3  
persistentvolumeclaim "pvc3" deleted
```

查看 PV、PVC:

```
root@master:~# kubectl get pvc  
NAME      STATUS      VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE  
pvc3     Terminating   pv0003    4Gi        RWO            
                    43m  
  
root@master:~# kubectl get pv  
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM           STORAGE  
pv0001    1Gi        RWO          Recycle         Available  
pv0002    2Gi        RWO          Recycle         Available  
pv0003    4Gi        RWO          Delete          Bound       default/pvc3
```

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-11-10 22:32:05

- 6. 安全

6. 安全

待写内容

安全

解释API请求的流程。

配置授权规则。

检查认证政策。

使用网络策略限制网络流量。

认证授权鉴权

管理基于角色的访问控制(RBAC)

Pod 和容器操作权限安全策略

network policy

上下文，命名空间隔离网络

集群切换和上下文

11.1 验证管理

token的认证方式

kubeconfig的认证方式

11.2 RBAC 鉴权

了解kubernetes的鉴权方式

配置RBAC 鉴权

11.3 资源限制

LimitRange

ResourceQuota

将内容分为 3 篇比较合适。

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新

时间： 2021-11-11 07:16:08

- 7.API与对象

7.API与对象

APIS AND ACCESS

<https://linux.thoughtindustries.com/learn/course/kubernetes-fundamentals-lfs258/apis-and-access/introduction?client=linux-foundation-open-source-software-university>

API OBJECTS

<https://linux.thoughtindustries.com/learn/course/kubernetes-fundamentals-lfs258/api-objects/introduction?client=linux-foundation-open-source-software-university>

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间: 2021-11-02 21:47:52

- 8.集群维护与日志和故障排除

8.集群维护与日志和故障排除

日常管理和维护:

节点维护命令 **cordon/uncordon**

kubectl cluster-info dump 查看集群日志

管理和插件整个集群的节点的日志。

管理容器标准输出和错误日志。包括了使用某些平台托管监控 **pod** 日志。

健康检查:

就绪探针， 健康探针

容器管理工具

7.1**pod**的默认检查策略

7.2通过**liveness**对**pod**健康性检查

7.3使用**readiness**对**pod**健康性检查

7.4健康性检查在各种环境中的应用

Pod 存活探针

84 页

死锁避免，死锁预防，死锁检测，死锁解除，通过存活探针检测状态，也是进而被 **kubernetes** 杀死并重启。

探针的CPU时间，计入容器的CPU配额。

helm:

12.1**Helm**工具的架构和安装使用

12.2**helm**源管理

12.3搭建**helm**私有仓库

实战：用**helm3**部署**EFK**日志

实战：用**helm3**部署**prometheus**监控

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook文档最后更新

时间： 2021-11-11 07:23:49

- [Readme](#)

Readme

痴者工良-Kubernetes 教程-CKAD教程

Copyright © 痴者工良 2021 all right reserved, powered by Gitbook 文档最后更新
时间： 2021-05-20 21:52:38