Kubernetes 零基础入门文档

Kubernetes 是什么

Kubernetes 是容器集群管理系统，是一个开源的平台，可以实现容器集群的自动化部署、自动扩缩容、维护等功能。

通过 Kubernetes 你可以：

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用功能
- 节省资源，优化硬件资源的使用
- 我们的目标是促进完善组件和工具的生态系统，以减轻应用程序在公有云或私有云中运行的负担。

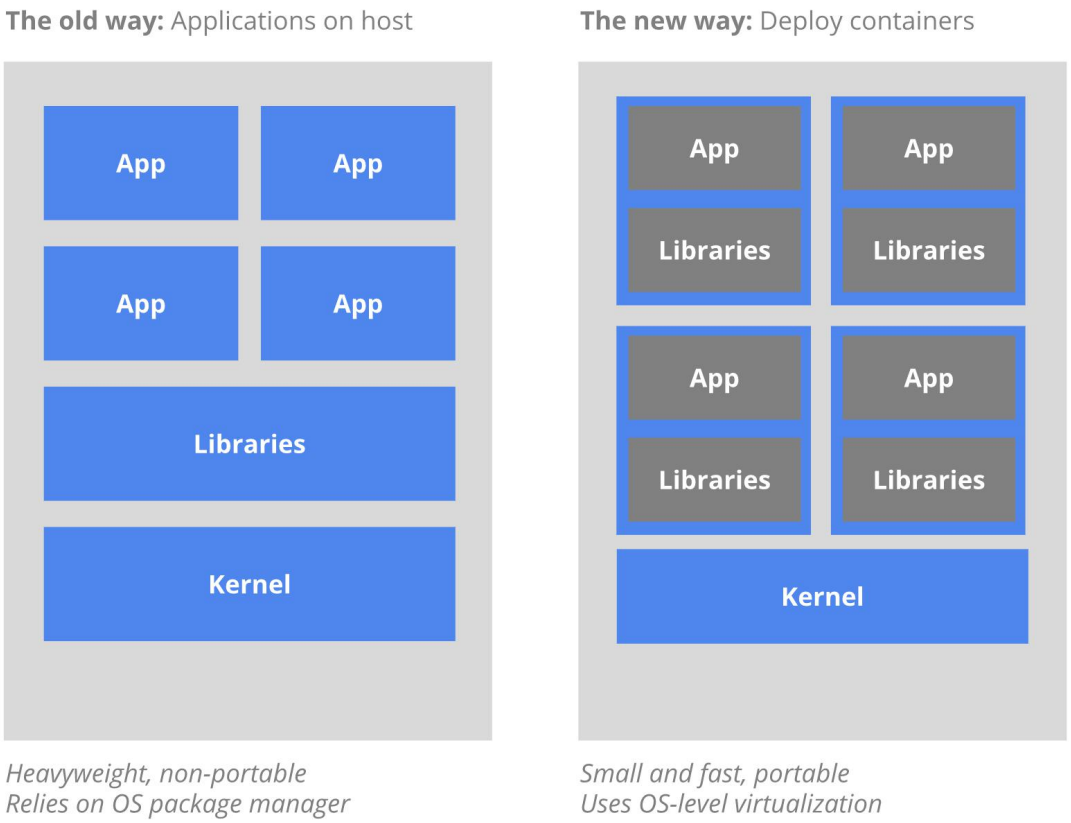
Kubernetes 特点

- 可移植：支持公有云，私有云，混合云，多重云（multi-cloud）
- 可扩展：模块化，插件化，可挂载，可组合
- 自动化：自动部署，自动重启，自动复制，自动伸缩/扩展

Kubernetes 是 Google 2014 年创建管理的，是 Google 10 多年大规模容器管理技术 Borg 的开源版本。

Why contain ers?

为什么要使用容器？通过以下两个图对比：



传统的应用部署方式是通过插件或脚本来安装应用。这样做的缺点是应用的运行、配置、管理、所有生存周期将与当前操作系统绑定，这样做并不利于应用的升级更新/回滚等操作，当然也可以通过创建虚机的方式来实现某些功能，但是虚拟机非常重，并不利于可移植性。

新的方式是通过部署容器方式实现，每个容器之间互相隔离，每个容器有自己的文件系统 ，容器之间进程不会相互影响，能区分计算资源。相对于虚拟机，容器能快速部署，由于容器与底层设施、机器文件系统解耦的，所以它能在不同云、不同版本操作系统间进行迁移。

容器占用资源少、部署快，每个应用可以被打包成一个容器镜像，每个应用与容器间成一对一关系也使容器有更大优势，使用容器可以在 build 或 release 的阶段，为应用创建容器镜像，因为每个应用不需要与其余的应用堆栈组合，也不依赖于生产环境基础结构，这使得从研发到测试、生产能提供一致环境。类似地，容器比虚机轻量、更“透明”，这更便于监控和管理。最后，

容器优势总结：

- 快速创建/部署应用：与 VM 虚拟机相比，容器镜像的创建更加容易。
- 持续开发、集成和部署：提供可靠且频繁的容器镜像构建/部署，并使用快速和简单的回滚(由于镜像不可变性)。
- 开发和运行相分离：在 build 或者 release 阶段创建容器镜像，使得应用和基础设施解耦。
- 开发，测试和生产环境一致性：在本地或外网（生产环境）运行的一致性。

- 云平台或其他操作系统：可以在 Ubuntu、RHEL、 CoreOS、on-prem、Google Container Engine 或其它任何环境中运行。
- Loosely coupled，分布式，弹性，微服务化：应用程序分为更小的、独立的部件，可以动态部署和管理。
- 资源隔离
- 资源利用：更高效

使用 Kubernetes 能做什么？

可以在物理或虚拟机的 Kubernetes 集群上运行容器化应用，Kubernetes 能提供一个以“容器为中心的基础架构”，满足在生产环境中运行应用的一些常见需求，如：

- 多个进程（作为容器运行）协同工作。（Pod）
- 存储系统挂载
- Distributing secrets
- 应用健康检测
- 应用实例的复制
- Pod 自动伸缩/扩展
- Naming and discovering
- 负载均衡
- 滚动更新
- 资源监控
- 日志访问
- 调试应用程序
- 提供认证和授权

Kubernetes 不是什么？

Kubernetes 并不是传统的 PaaS（平台即服务）系统。

Kubernetes 不限制支持应用的类型，不限制应用框架。不限制受支持的语言 runtimes (例如，Java, Python, Ruby)，满足 12-factor applications 。不区分 “apps” 或者“services”。Kubernetes 支持不同负载应用，包括有状态、无状态、数据处理类型的应用。只要这个应用可以在容器里运行，那么就能很好的运行在 Kubernetes 上。
Kubernetes 不提供中间件（如 message buses）、数据处理框架（如 Spark）、数据库(如 Mysql)或者集群存储系统(如 Ceph)作为内置服务。但这些应用都可以运行在 Kubernetes 上面。
Kubernetes 不部署源码不编译应用。持续集成的 (CI)工作流方面，不同的用户有不同的需求和偏好的区域，因此，我们提供分层的 CI 工作流，但并不定义它应该如何工作。
Kubernetes 允许用户选择自己的日志、监控和报警系统。
Kubernetes 不提供或授权一个全面的应用程序配置 语言/系统（例如，jsonnet）。
Kubernetes 不提供任何机器配置、维护、管理或者自修复系统。
另一方面，大量的 Paas 系统都可以运行在 Kubernetes 上，比如 Openshift、Deis、Gondor。

可以构建自己的 Paas 平台，与自己选择的 CI 系统集成。

由于 Kubernetes 运行在应用级别而不是硬件级，因此提供了普通的 Paas 平台提供的一些通用功能，比如部署，扩展，负载均衡，日志，监控等。这些默认功能是可选的。

另外，Kubernetes 不仅仅是一个“编排系统”；它消除了编排的需要。“编排”的定义是指执行一个预定的工作流：先执行 A，之 B，然 C。相反，Kubernetes 由一组独立的可组合控制进程组成。怎么样从 A 到 C 并不重要，达到目的就好。当然集中控制也是必不可少，方法更像排舞的过程。这使得系统更加易用、强大、弹性和可扩展。

Kubernetes 是什么意思？K8S?

Kubernetes 的名字来自希腊语, 意思是“舵手”或“领航员”。K8s 是将 8 个字母“ubernete”替换为“8”的缩写。

设计文档

Kubernetes 最初源于谷歌内部的 Borg，提供了面向应用的容器集群部署和管理系统。Kubernetes 的目标旨在消除编排物理/虚拟计算，网络和存储基础设施的负担，并使应用程序运营商和开发人员完全将重点放在以容器为中心的原语上进行自助运营。Kubernetes 也提供稳定、兼容的基础（平台），用于构建定制化的 workflows 和更高级的自动化任务。

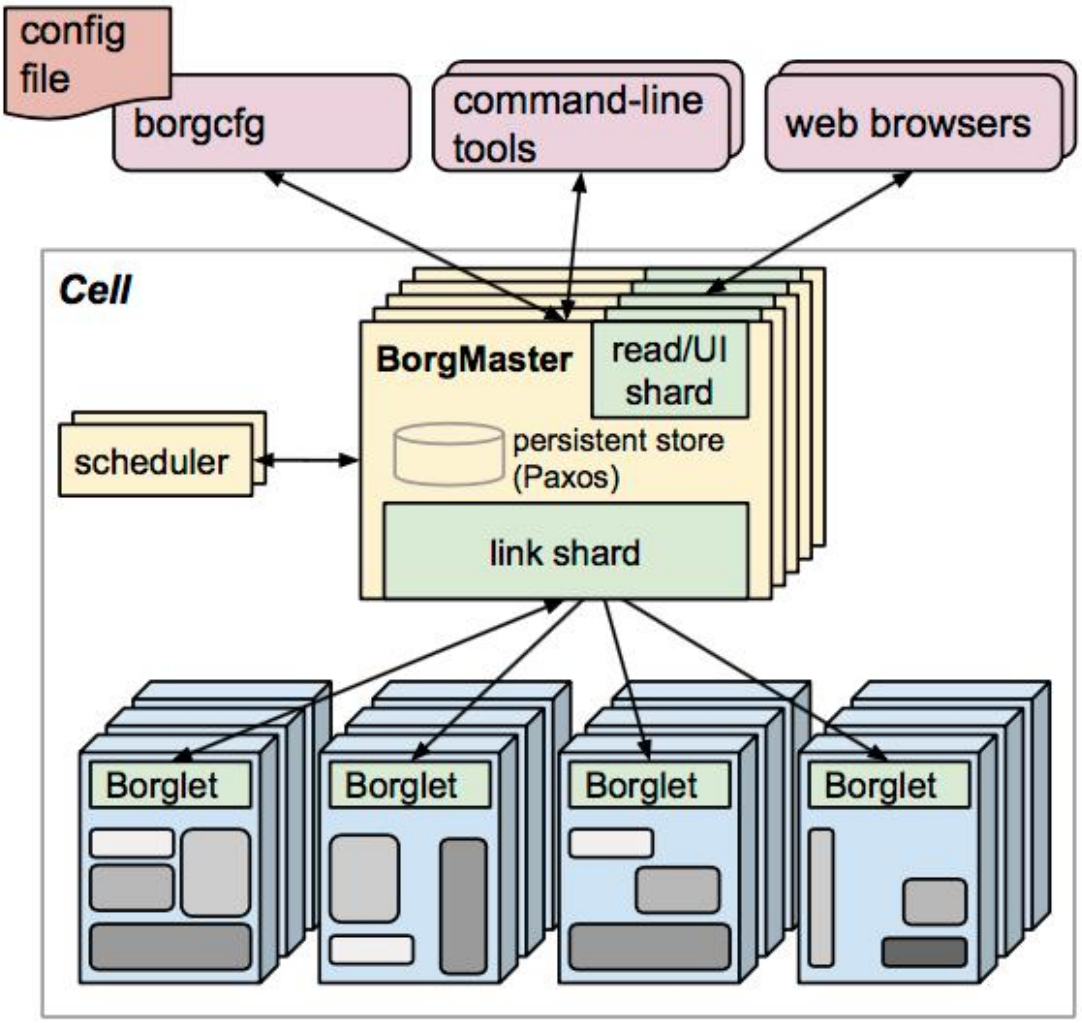
Kubernetes 具备完善的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和在线扩容、可扩展的资源自动调度机制、多粒度的资源配额管理能力。

Kubernetes 还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

Borg 简介

Borg 是谷歌内部的大规模集群管理系统, 负责对谷歌内部很多核心服务的调度和管理。Borg 的目的是让用户能够不必操心资源管理的问题，让他们专注于自己的核心业务，并且做到跨多个数据中心的资源利用率最大化。

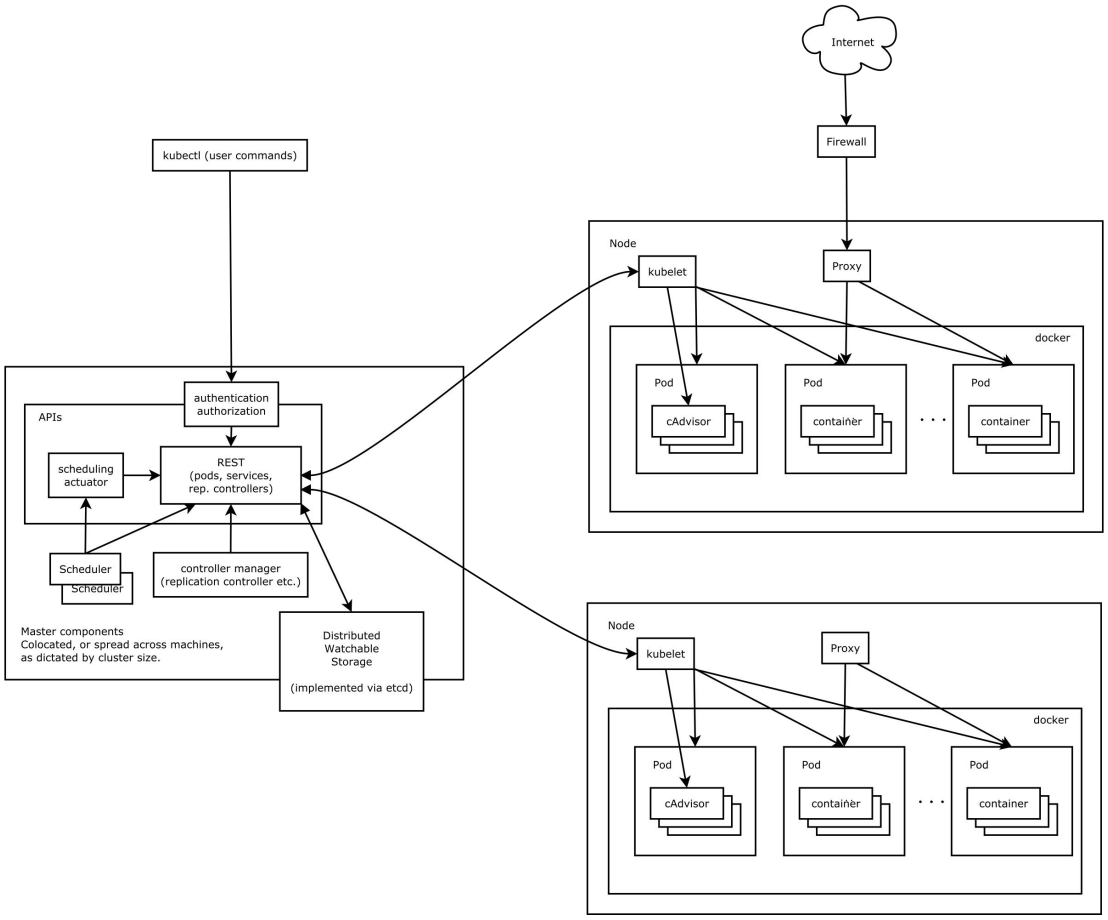
Borg 主要由 BorgMaster、Borglet、borgcfg 和 Scheduler 组成，如下图所示



- BorgMaster 是整个集群的大脑，负责维护整个集群的状态，并将数据持久化到 Paxos 存储中；
- Scheduer 负责任务的调度，根据应用的特点将其调度到具体的机器上去；
- Borglet 负责真正运行任务（在容器中）；
- borgcfg 是 Borg 的命令行工具，用于跟 Borg 系统交互，一般通过一个配置文件来提交任务。

Kubernetes 架构

Kubernetes 借鉴了 Borg 的设计理念，比如 Pod、Service、Labels 和单 Pod 单 IP 等。Kubernetes 的整体架构跟 Borg 非常像，如下图所示



Kubernetes 主要由以下几个核心组件组成：

- etcd 保存了整个集群的状态；
- apiserver 提供了资源操作的唯一入口，并提供认证、授权、访问控制、API 注册和发现等机制；
- controller manager 负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上；
- kubelet 负责维护容器的生命周期，同时也负责 Volume（CVI）和网络（CNI）的管理；
- Container runtime 负责镜像管理以及 Pod 和容器的真正运行（CRI）；
- kube-proxy 负责为 Service 提供 cluster 内部的服务发现和负载均衡；

Pod

什么是 Pod?

Pod 就像是豌豆荚一样，它由一个或者多个容器组成（例如 Docker 容器），它们共享容器存储、网络和容器运行配置项。Pod 中的容器总是被同时调度，有共同的运行环境。你可以把单个 Pod 想象成是运行独立应用的“逻辑主机”——其中运行着一个或者多个紧密耦合的应用容器——在有容器之前，这些应用都是运行在几个相同的物理机或者虚拟机上。

尽管 kubernetes 支持多种容器运行时，但是 Docker 依然是最常用的运行时环境，我们可以使用 Docker 的术语和规则来定义 Pod。

Pod 中共享的环境包括 Linux 的 namespace、cgroup 和其他可能的隔绝环境，这一点跟 Docker 容器一致。在 Pod 的环境中，每个容器中可能还有更小的子隔离环境。

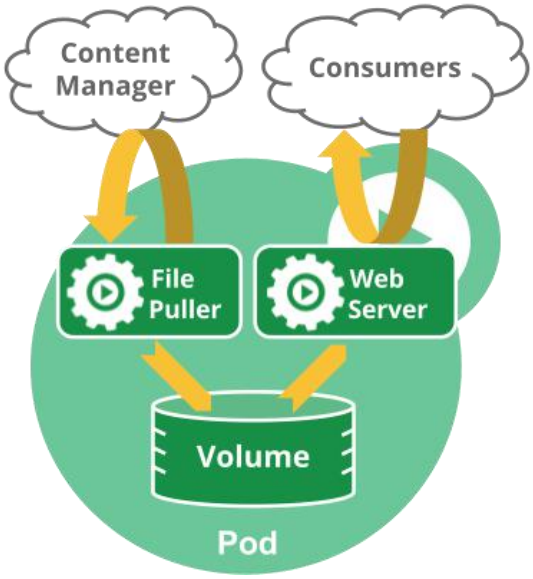
Pod 中的容器共享 IP 地址和端口号，它们之间可以通过 localhost 互相发现。它们之间可以通过进程间通信，例如 SystemV 信号或者 POSIX 共享内存。不同 Pod 之间的容器具有不同的 IP 地址，不能直接通过 IPC 通信。

Pod 中的容器也有访问共享 volume 的权限，这些 volume 会被定义成 pod 的一部分并挂载到应用容器的文件系统中。

根据 Docker 的结构，Pod 中的容器共享 namespace 和 volume，不支持共享 PID 的 namespace。

就像每个应用容器，pod 被认为是临时（非持久的）实体。在 Pod 的生命周期中讨论过，pod 被创建后，被分配一个唯一的 ID（UID），调度到节点上，并一致维持期望的状态直到被终结（根据重启策略）或者被删除。如果 node 死掉了，分配到了这个 node 上的 pod，在经过一个超时时间后会被重新调度到其他 node 节点上。一个给定的 pod（如 UID 定义的）不会被“重新调度”到新的节点上，而是被一个同样的 pod 取代，如果期望的话甚至可以是相同的名字，但是会有一个新的 UID。

Volume 跟 pod 有相同的生命周期（当其 UID 存在的时候）。当 Pod 因为某种原因被删除或者被新创建的相同的 Pod 取代，它相关的东西（例如 volume）也会被销毁和再创建一个新的 volume。



一个多容器 Pod，包含文件提取程序和 Web 服务器，该服务器使用持久卷在容器之间共享存储。

Pod 安全策略

PodSecurityPolicy 类型的对象能够控制，是否可以向 Pod 发送请求，该 Pod 能够影响被应用到 Pod 和容器的 SecurityContext。 查看 Pod 安全策略建议 获取更多信息。

什么是 Pod 安全策略?

Pod 安全策略 是集群级别的资源，它能够控制 Pod 运行的行为，以及它具有访问什么的能力。 PodSecurityPolicy 对象定义了一组条件，指示 Pod 必须按系统所能接受的顺序运行。 它们允许管理员控制如下方面：

控制面	字段名称
已授权容器的运行	privileged
为容器添加默认的一组能力	defaultAddCapabilities
为容器去掉某些能力	requiredDropCapabilities
容器能够请求添加某些能力	allowedCapabilities
控制卷类型的使用	volumes
主机网络的使用	hostNetwork
主机端口的使用	hostPorts
主机 PID namespace 的使用	hostPID
主机 IPC namespace 的使用	hostIPC
主机路径的使用	allowedHostPaths
容器的 SELinux 上下文	seLinux
用户 ID	runAsUser
配置允许的补充组	supplementalGroups
分配拥有 Pod 数据卷的 FSGroup	fsGroup
必须使用一个只读的 root 文件系统	readOnlyRootFilesystem

Pod 安全策略 由设置和策略组成，它们能够控制 Pod 访问的安全特征。这些设置分为如下三类：

- 基于布尔值控制：这种类型的字段默认为最严格限制的值。
- 基于被允许的值集合控制：这种类型的字段会与这组值进行对比，以确认值被允许。
- 基于策略控制：设置项通过一种策略提供的机制来生成该值，这种机制能够确保指定的值落在被允许的这组值中。

RunAsUser

- MustRunAs - 必须配置一个 range。使用该范围内的第一个值作为默认值。验证是否不在配置的该范围内。
- MustRunAsNonRoot - 要求提交的 Pod 具有非零 runAsUser 值，或在镜像中定义了 USER 环境变量。不提供默认值。
- RunAsAny - 没有提供默认值。允许指定任何 runAsUser 。

SELinux

- MustRunAs - 如果没有使用预分配的值，必须配置 seLinuxOptions。默认使用 seLinuxOptions。验证 seLinuxOptions。
- RunAsAny - 没有提供默认值。允许任意指定的 seLinuxOptions ID。

SupplementalGroups

- MustRunAs - 至少需要指定一个范围。默认使用第一个范围的最小值。验证所有范围的值。
- RunAsAny - 没有提供默认值。允许任意指定的 supplementalGroups ID。

FSGroup

- MustRunAs - 至少需要指定一个范围。默认使用第一个范围的最小值。验证在第一个范围内的第一个 ID。
- RunAsAny - 没有提供默认值。允许任意指定的 fsGroup ID。

控制卷

通过设置 PSP 卷字段，能够控制具体卷类型的使用。当创建一个卷的时候，与该字段相关的已定义卷可以允许设置如下值：

- azureFile
- azureDisk
- flocker
- flexVolume

- hostPath
- emptyDir
- gcePersistentDisk
- awsElasticBlockStore
- gitRepo
- secret
- nfs
- iscsi
- glusterfs
- persistentVolumeClaim
- rbd
- cinder
- cephFS
- downwardAPI
- fc
- configMap
- vsphereVolume
- quobyte
- photonPersistentDisk
- projected
- portworxVolume
- scaleIO
- storageos

* (allow all volumes)

对新的 PSP，推荐允许的卷的最小集合包括：configMap、downwardAPI、emptyDir、persistentVolumeClaim、secret 和 projected。

主机网络

HostPorts， 默认为 empty。HostPortRange 列表通过 min(包含) and max(包含) 来定义，指定了被允许的主机端口。

允许的主机路径

AllowedHostPaths 是一个被允许的主机路径前缀的白名单。空值表示所有的主机路径都可以使用。

许可

包含 PodSecurityPolicy 的 许可控制，允许控制集群资源的创建和修改，基于这些资源在集群范围内被许可的能力。

许可使用如下的方式为 Pod 创建最终的安全上下文：

1. 检索所有可用的 PSP。
2. 生成在请求中没有指定的安全上下文设置的字段值。
3. 基于可用的策略，验证最终的设置。

如果某个策略能够匹配上，该 Pod 就被接受。如果请求与 PSP 不匹配，则 Pod 被拒绝。

Pod 必须基于 PSP 验证每个字段。

创建 Pod 安全策略

下面是一个 Pod 安全策略的例子，所有字段的设置都被允许：

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - min: 8000
    max: 8080
  volumes:
  - '*'
```

下载示例文件可以创建该策略，然后执行如下命令：

```
$ kubectl create -f ./psp.yaml
podsecuritypolicy "permissive" created
```

获取 Pod 安全策略列表

获取已存在策略列表，使用 kubectl get：

```
$ kubectl get psp
NAME          PRIV    CAPS   SELINUX    RUNASUSER          FSGROUP     SUPGROUP    READONLYROOTFS   VOLUMES
permissive    false   []     RunAsAny   RunAsAny           RunAsAny    RunAsAny    false            [*]
privileged    true    []     RunAsAny   RunAsAny           RunAsAny    RunAsAny    false            [*]
restricted    false   []     RunAsAny   MustRunAsNonRoot   RunAsAny    RunAsAny    false            [emptyDir secret downwardAPI
configMap persistentVolumeClaim projected]
```

修改 Pod 安全策略

通过交互方式修改策略，使用 kubectl edit：

```
$ kubectl edit psp permissive
```

该命令将打开一个默认文本编辑器，在这里能够修改策略。

删除 Pod 安全策略

一旦不再需要一个策略，很容易通过 kubectl 删除它：

```
$ kubectl delete psp permissive
podsecuritypolicy "permissive" deleted
```

启用 Pod 安全策略

为了能够在集群中使用 Pod 安全策略，必须确保如下：

1、启用 API 类型 extensions/v1beta1/podsecuritypolicy（仅对 1.6 之前的版本）
2、启用许可控制器 PodSecurityPolicy
3、定义自己的策略

使用 RBAC

在 Kubernetes 1.5 或更新版本，可以使用 PodSecurityPolicy 来控制，对基于用户角色和组的已授权容器的访问。访问不同的 PodSecurityPolicy 对象，可以基于认证来控制。基于 Deployment、ReplicaSet 等创建的 Pod，限制访问 PodSecurityPolicy 对象，Controller Manager 必须基于安全 API 端口运行，并且不能够具有超级用户权限。

PodSecurityPolicy 认证使用所有可用的策略，包括创建 Pod 的用户，Pod 上指定的服务账户（service acount）。当 Pod 基于 Deployment、ReplicaSet 创建时，它是创建 Pod

的 Controller Manager，所以如果基于非安全 API 端口运行，允许所有的 PodSecurityPolicy 对象，并且不能够有效地实现细分权限。用户访问给定的 PSP 策略有效，仅当是直接部署 Pod 的情况。更多详情，查看 PodSecurityPolicy RBAC 示例，当直接部署 Pod 时，应用 PodSecurityPolicy 控制基于角色和组的已授权容器的访问 。

原文地址：https://k8smeetup.github.io/docs/concepts/policy/pod-security-policy/
译者：shirdrn

Pod 生命周期

Pod phase

Pod 的 status 定义在 PodStatus 对象中，其中有一个 phase 字段。

Pod 的相位（phase）是 Pod 在其生命周期中的简单宏观概述。该阶段并不是对容器或 Pod 的综合汇总，也不是为了做为综合状态机。

Pod 相位的数量和含义是严格指定的。除了本文档中列举的内容外，不应该再假定 Pod 有其他的 phase 值。

下面是 phase 可能的值：

- 挂起（Pending）：Pod 已被 Kubernetes 系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括调度 Pod 的时间和通过网络下载镜像的时间，这可能需要花点时间。
- 运行中（Running）：该 Pod 已经绑定到了一个节点上，Pod 中所有的容器都已被创建。至少有一个容器正在运行，或者正处于启动或重启状态。
- 成功（Succeeded）：Pod 中的所有容器都被成功终止，并且不会再重启。
- 失败（Failed）：Pod 中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非 0 状态退出或者被系统终止。
- 未知（Unknown）：因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败。

Pod 状态

Pod 有一个 PodStatus 对象，其中包含一个 PodCondition 数组。 PodCondition 数组的每个元素都有一个 type 字段和一个 status 字段。type 字段是字符串，可能的值有 PodScheduled、Ready、Initialized 和 Unschedulable。status 字段是一个字符串，可能的值有 True、False 和 Unknown。

容器探针

探针 是由 kubelet 对容器执行的定期诊断。要执行诊断，kubelet 调用由容器实现的 Handler。有三种类型的处理程序：

- ExecAction：在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。
- TCPSocketAction：对指定端口上的容器的 IP 地址进行 TCP 检查。如果端口打开，则诊断被认为是成功的。
- HTTPGetAction：对指定的端口和路径上的容器的 IP 地址执行 HTTP Get 请求。如果响应的状态码大于等于 200 且小于 400，则诊断被认为是成功的。

每次探测都将获得以下三种结果之一：

- 成功：容器通过了诊断。
- 失败：容器未通过诊断。
- 未知：诊断失败，因此不会采取任何行动。

Kubelet 可以选择是否执行在容器上运行的两种探针执行和做出反应：

- livenessProbe：指示容器是否正在运行。如果存活探测失败，则 kubelet 会杀死容器，并且容器将受到其 重启策略 的影响。如果容器不提供存活探针，则默认状态为 Success。
- readinessProbe：指示容器是否准备好服务请求。如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service 的端点中删除该 Pod 的 IP 地址。初始延迟之前的就绪状态默认为 Failure。如果容器不提供就绪探针，则默认状态为 Success。

该什么时候使用存活（liveness）和就绪（readiness）探针?

如果容器中的进程能够在遇到问题或不健康的情况下自行崩溃，则不一定需要存活探针；kubelet 将根据 Pod 的 restartPolicy 自动执行正确的操作。

如果您希望容器在探测失败时被杀死并重新启动，那么请指定一个存活探针，并指定 restartPolicy 为 Always 或 OnFailure。

如果要仅在探测成功时才开始向 Pod 发送流量，请指定就绪探针。在这种情况下，就绪探针可能与存活探针相同，但是 spec 中的就绪探针的存在意味着 Pod 将在没有接收到任何流量的情况下启动，并且只有在探针探测成功后才开始接收流量。

如果您希望容器能够自行维护，您可以指定一个就绪探针，该探针检查与存活探针不同的端点。

请注意，如果您只想在 Pod 被删除时能够排除请求，则不一定需要使用就绪探针；在删除 Pod 时，Pod 会自动将自身置于未完成状态，无论就绪探针是否存在。当等待 Pod 中的容器停止时，Pod 仍处于未完成状态。

Pod 和容器状态

有关 Pod 容器状态的详细信息，请参阅 PodStatus 和 ContainerStatus。请注意，报告

的 Pod 状态信息取决于当前的 ContainerState。

重启策略

PodSpec 中有一个 restartPolicy 字段，可能的值为 Always、OnFailure 和 Never。默认为 Always。 restartPolicy 适用于 Pod 中的所有容器。restartPolicy 仅指通过同一节点上的 kubelet 重新启动容器。失败的容器由 kubelet 以五分钟为上限的指数退避延迟（10 秒，20 秒，40 秒…）重新启动，并在成功执行十分钟后重置。如 Pod 文档 中所述，一旦绑定到一个节点，Pod 将永远不会重新绑定到另一个节点。

Pod 的生命

一般来说，Pod 不会消失，直到人为销毁他们。这可能是一个人或控制器。这个规则的唯一例外是成功或失败的 phase 超过一段时间（由 master 确定）的 Pod 将过期并被自动销毁。

有三种可用的控制器：

- 使用 Job 运行预期会终止的 Pod，例如批量计算。Job 仅适用于重启策略为 OnFailure 或 Never 的 Pod。
- 对预期不会终止的 Pod 使用 ReplicationController、ReplicaSet 和 Deployment ，例如 Web 服务器。 ReplicationController 仅适用于具有 restartPolicy 为 Always 的 Pod。
- 提供特定于机器的系统服务，使用 DaemonSet 为每台机器运行一个 Pod 。

所有这三种类型的控制器都包含一个 PodTemplate。建议创建适当的控制器，让它们来创建 Pod，而不是直接自己创建 Pod。这是因为单独的 Pod 在机器故障的情况下没有办法自动复原，而控制器却可以。

如果节点死亡或与集群的其余部分断开连接，则 Kubernetes 将应用一个策略将丢失节点上的所有 Pod 的 phase 设置为 Failed。

示例

高级 liveness 探针示例
存活探针由 kubelet 来执行，因此所有的请求都在 kubelet 的网络命名空间中进行。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
```

```
    name: liveness-http
spec:
    containers:
    - args:
        - /server
        image: gcr.io/google_containers/liveness
        livenessProbe:
            httpGet:
                # when "host" is not defined, "PodIP" will be used
                # host: my-host
                # when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and
"HTTPS" are allowed
                # scheme: HTTPS
                path: /healthz
                port: 8080
                httpHeaders:
                    - name: X-Custom-Header
                        value: Awesome
            initialDelaySeconds: 15
            timeoutSeconds: 1
        name: liveness
```

状态示例

- Pod 中只有一个容器并且正在运行。容器成功退出。
  记录完成事件。
  如果 restartPolicy 为：
    Always：重启容器；Pod phase 仍为 Running。
    OnFailure：Pod phase 变成 Succeeded。
    Never：Pod phase 变成 Succeeded。

- Pod 中只有一个容器并且正在运行。容器退出失败。
    记录失败事件。
    如果 restartPolicy 为：
        Always：重启容器；Pod phase 仍为 Running。
        OnFailure：重启容器；Pod phase 仍为 Running。
        Never：Pod phase 变成 Failed。

- Pod 中有两个容器并且正在运行。有一个容器退出失败。
    记录失败事件。
    如果 restartPolicy 为：
        Always：重启容器；Pod phase 仍为 Running。
        OnFailure：重启容器；Pod phase 仍为 Running。

Never：不重启容器；Pod phase 仍为 Running。

- 如果有一个容器没有处于运行状态，并且两个容器退出：
  记录失败事件。
  如果 restartPolicy 为：
  Always：重启容器；Pod phase 仍为 Running。
  OnFailure：重启容器；Pod phase 仍为 Running。
  Never：Pod phase 变成 Failed。
- Pod 中只有一个容器并处于运行状态。容器运行时内存超出限制：
  容器以失败状态终止。
  记录 OOM 事件。
  如果 restartPolicy 为：
  Always：重启容器；Pod phase 仍为 Running。
  OnFailure：重启容器；Pod phase 仍为 Running。
  Never: 记录失败事件；Pod phase 仍为 Failed。
- Pod 正在运行，磁盘故障：
  杀掉所有容器。
  记录适当事件。
  Pod phase 变成 Failed。
  如果使用控制器来运行，Pod 将在别处重建。
- Pod 正在运行，其节点被分段。
  节点控制器等待直到超时。
  节点控制器将 Pod phase 设置为 Failed。
  如果是用控制器来运行，Pod 将在别处重建。

译者：jimmysong 原文：
https://k8smeetup.github.io/docs/concepts/workloads/pods/pod-lifecycle/

Init 容器

该特性在 1.6 版本已经退出 beta 版本。Init 容器可以在 PodSpec 中同应用程序的 containers 数组一起来指定。 beta 注解的值将仍需保留，并覆盖 PodSpec 字段值。

本文讲解 Init 容器的基本概念，它是一种专用的容器，在应用程序容器启动之前运行，并包括一些应用镜像中不存在的实用工具和安装脚本。

理解 Init 容器

Pod 能够具有多个容器，应用运行在容器里面，但是它也可能有一个或多个先于应用容器启动的 Init 容器。

Init 容器与普通的容器非常像，除了如下两点：

- Init 容器总是运行到成功完成为止。

- 每个 Init 容器都必须在下一个 Init 容器启动之前成功完成。

如果 Pod 的 Init 容器失败，Kubernetes 会不断地重启该 Pod，直到 Init 容器成功为止。然而，如果 Pod 对应的 restartPolicy 为 Never，它不会重新启动。

指定容器为 Init 容器，在 PodSpec 中添加 initContainers 字段，以 v1.Container 类型对象的 JSON 数组的形式，还有 app 的 containers 数组。 Init 容器的状态在 status.initContainerStatuses 字段中以容器状态数组的格式返回（类似 status.containerStatuses 字段）。

与普通容器的不同之处

Init 容器支持应用容器的全部字段和特性，包括资源限制、数据卷和安全设置。 然而，Init 容器对资源请求和限制的处理稍有不同，在下面 资源 处有说明。 而且 Init 容器不支持 Readiness Probe，因为它们必须在 Pod 就绪之前运行完成。

如果为一个 Pod 指定了多个 Init 容器，那些容器会按顺序一次运行一个。 每个 Init 容器必须运行成功，下一个才能够运行。 当所有的 Init 容器运行完成时，Kubernetes 初始化 Pod 并像平常一样运行应用容器。

Init 容器能做什么？

因为 Init 容器具有与应用程序容器分离的单独镜像，所以它们的启动相关代码具有如下优势：

- 它们可以包含并运行实用工具，但是出于安全考虑，是不建议在应用程序容器镜像中包含这些实用工具的。
- 它们可以包含使用工具和定制化代码来安装，但是不能出现在应用程序镜像中。例如，创建镜像没必要 FROM 另一个镜像，只需要在安装过程中使用类似 sed、 awk、 python 或 dig 这样的工具。
- 应用程序镜像可以分离出创建和部署的角色，而没有必要联合它们构建一个单独的镜像。
- Init 容器使用 Linux Namespace，所以相对应用程序容器来说具有不同的文件系统视图。因此，它们能够具有访问 Secret 的权限，而应用程序容器则不能。
- 它们必须在应用程序容器启动之前运行完成，而应用程序容器是并行运行的，所以 Init 容器能够提供了一种简单的阻塞或延迟应用容器的启动的方法，直到满足了一组先决条件。

示例

下面是一些如何使用 Init 容器的想法：

等待一个 Service 创建完成，通过类似如下 shell 命令：

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; exit 1
```

将 Pod 注册到远程服务器，通过在命令中调用 API，类似如下：

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d
'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
```

在启动应用容器之前等一段时间，使用类似 sleep 60 的命令。
克隆 Git 仓库到数据卷。

将配置值放到配置文件中，运行模板工具为主应用容器动态地生成配置文件。例如，在配置文件中存放 POD_IP 值，并使用 Jinja 生成主应用配置文件。

更多详细用法示例，可以在 StatefulSet 文档 和 生产环境 Pod 指南 中找到。

使用 Init 容器

下面是 Kubernetes 1.5 版本 yaml 文件，展示了一个具有 2 个 Init 容器的简单 Pod。第一个等待 myservice 启动, 第二个等待 mydb 启动。一旦这两个 Service 都启动完成，Pod 将开始启动。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; do echo waiting for
myservice; sleep 2; done;"]
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb;
sleep 2; done;"]
        }
```

```
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

这是 Kubernetes 1.6 版本的新语法，尽管老的 annotation 语法仍然可以使用。我们已经把 Init 容器的声明移到 spec 中：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

1.5 版本的语法在 1.6 版本仍然可以使用，但是我们推荐使用 1.6 版本的新语法。 在 Kubernetes 1.6 版本中，Init 容器在 API 中新建了一个字段。 虽然期望使用 beta 版本的 annotation，但在未来发行版将会被废弃掉。

下面的 yaml 文件展示了 mydb 和 myservice 两个 Service：

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
```

```
      targetPort: 9376
---
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

这个 Pod 可以使用下面的命令进行启动和调试：

```
$ kubectl create -f myapp.yaml
pod "myapp-pod" created
$ kubectl get -f myapp.yaml
NAME         READY     STATUS     RESTARTS   AGE
myapp-pod    0/1       Init:0/2   0          6m
$ kubectl describe -f myapp.yaml
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app=myapp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:     Running
[...]
  init-mydb:
[...]
    State:     Waiting
      Reason:  PodInitializing
    Ready:     False
[...]
Containers:
  myapp-container:
[...]
    State:     Waiting
      Reason:  PodInitializing
    Ready:     False
[...]
Events:
```

```
  FirstSeen       LastSeen       Count       From                                SubObjectPath
Type              Reason          Message
  ---------       --------       -----       ----                                -------------
--------          ------          -------
  16s                 16s                 1                 {default-scheduler }
Normal            Scheduled       Successfully assigned myapp-pod to 172.17.4.201
  16s                 16s                 1                 {kubelet 172.17.4.201}
spec.initContainers{init-myservice}       Normal            Pulling           pulling image
"busybox"
  13s                 13s                 1                 {kubelet 172.17.4.201}
spec.initContainers{init-myservice}       Normal            Pulled            Successfully
pulled image "busybox"
  13s                 13s                 1                 {kubelet 172.17.4.201}
spec.initContainers{init-myservice}       Normal            Created           Created
container with docker id 5ced34a04634; Security:[seccomp=unconfined]
  13s                 13s                 1                 {kubelet 172.17.4.201}
spec.initContainers{init-myservice}       Normal            Started           Started
container with docker id 5ced34a04634
$ kubectl logs myapp-pod -c init-myservice # Inspect the first init container
$ kubectl logs myapp-pod -c init-mydb      # Inspect the second init container
```

一旦我们启动了 mydb 和 myservice 这两个 Service，我们能够看到 Init 容器完成，并且 myapp-pod 被创建：

```
$ kubectl create -f services.yaml
service "myservice" created
service "mydb" created
$ kubectl get -f myapp.yaml
NAME          READY     STATUS    RESTARTS   AGE
myapp-pod     1/1       Running   0          9m
```

这个例子非常简单，但是应该能够为我们创建自己的 Init 容器提供一些启发。

具体行为

在 Pod 启动过程中，Init 容器会按顺序在网络和数据卷初始化之后启动。 每个容器必须在下一个容器启动之前成功退出。 如果由于运行时或失败退出，导致容器启动失败，它会根据 Pod 的 restartPolicy 指定的策略进行重试。 然而，如果 Pod 的 restartPolicy 设置为 Always，Init 容器失败时会使用 RestartPolicy 策略。

在所有的 Init 容器没有成功之前，Pod 将不会变成 Ready 状态。 Init 容器的端口将不会在 Service 中进行聚集。 正在初始化中的 Pod 处于 Pending 状态，但应该会将条件 Initializing 设置为 true。

如果 Pod 重启，所有 Init 容器必须重新执行。

对 Init 容器 spec 的修改，被限制在容器 image 字段中。更改 Init 容器的 image 字段，等价于重启该 Pod。

因为 Init 容器可能会被重启、重试或者重新执行，所以 Init 容器的代码应该是幂等的。特别地，被写到 EmptyDirs 中文件的代码，应该对输出文件可能已经存在做好准备。

Init 容器具有应用容器的所有字段。除了 readinessProbe，因为 Init 容器无法定义不同于完成（completion）的就绪（readiness）的之外的其他状态。这会在验证过程中强制执行。

在 Pod 上使用 activeDeadlineSeconds，在容器上使用 livenessProbe，这样能够避免 Init 容器一直失败。这就为 Init 容器活跃设置了一个期限。

在 Pod 中的每个 app 和 Init 容器的名称必须唯一；与任何其它容器共享同一个名称，会在验证时抛出错误。

资源

为 Init 容器指定顺序和执行逻辑，下面对资源使用的规则将被应用：

- 在所有 Init 容器上定义的，任何特殊资源请求或限制的最大值，是 有效初始请求/限制
- Pod 对资源的有效请求/限制要高于：
  所有应用容器对某个资源的请求/限制之和
  对某个资源的有效初始请求/限制
- 基于有效请求/限制完成调度，这意味着 Init 容器能够为初始化预留资源，这些资源在 Pod 生命周期过程中并没有被使用。
- Pod 的 有效 QoS 层，是 Init 容器和应用容器相同的 QoS 层。

基于有效 Pod 请求和限制来应用配额和限制。Pod 级别的 cgroups 是基于有效 Pod 请求和限制，和调度器相同。

Pod 重启的原因

Pod 能够重启，会导致 Init 容器重新执行，主要有如下几个原因：

- 用户更新 PodSpec 导致 Init 容器镜像发生改变。应用容器镜像的变更只会重启应用容器。
- Pod 基础设施容器被重启。这不多见，但某些具有 root 权限可访问 Node 的人可能会这样做。
- 当 restartPolicy 设置为 Always，Pod 中所有容器会终止，强制重启，由于垃圾收集

导致 Init 容器完成的记录丢失。

支持与兼容性

Apiserver 版本为 1.6 或更高版本的集群，通过使用 spec.initContainers 字段来支持 Init 容器。之前的版本可以使用 alpha 和 beta 注解支持 Init 容器。spec.initContainers 字段也被加入到 alpha 和 beta 注解中，所以 Kubernetes 1.3.0 版本或更高版本可以执行 Init 容器，并且 1.6 版本的 apiserver 能够安全的回退到 1.5.x 版本，而不会使存在的已创建 Pod 失去 Init 容器的功能。

原文地址：https://k8smeetup.github.io/docs/concepts/workloads/pods/init-containers/
译者：shirdrn

Pod 优先级和抢占

FEATURE STATE: Kubernetes v1.8 alpha

Kubernetes 1.8 及其以后的版本中可以指定 Pod 的优先级。优先级表明了一个 Pod 相对于其它 Pod 的重要性。当 Pod 无法被调度时，scheduler 会尝试抢占（驱逐）低优先级的 Pod，使得这些挂起的 pod 可以被调度。在 Kubernetes 未来的发布版本中，优先级也会影响节点上资源回收的排序。

注： 抢占不遵循 PodDisruptionBudget；更多详细的信息，请查看 限制部分。

怎么样使用优先级和抢占

想要在 Kubernetes 1.8 版本中使用优先级和抢占，请参考如下步骤：

1. 启用功能。
2. 增加一个或者多个 PriorityClass。
3. 创建拥有字段 PriorityClassName 的 Pod，该字段的值选取上面增加的 PriorityClass。当然，您没有必要直接创建 pod，通常您可以把 PriorityClassName 增加到类似 Deployment 这样的集合对象的 Pod 模板中。

以下章节提供了有关这些步骤的详细信息。

启用优先级和抢占

Kubernetes 1.8 版本默认没有开启 Pod 优先级和抢占。为了启用该功能，需要在 API server 和 scheduler 的启动参数中设置：

--feature-gates=PodPriority=true

在 API server 中还需要设置如下启动参数：

--runtime-config=scheduling.k8s.io/v1alpha1=true

功能启用后，您能创建 PriorityClass，也能创建使用 PriorityClassName 集的 Pod。

如果在尝试该功能后想要关闭它，那么您可以把 PodPriority 这个命令行标识从启动参数中移除，或者将它的值设置为 false，然后再重启 API server 和 scheduler。功能关闭后，原来的 Pod 会保留它们的优先级字段，但是优先级字段的内容会被忽略，抢占不会生效，在新的 pod 创建时，您也不能设置 PriorityClassName。

PriorityClass

PriorityClass 是一个不受命名空间约束的对象，它定义了优先级类名跟优先级整数值的映射。它的名称通过 PriorityClass 对象 metadata 中的 name 字段指定。值在必选的 value 字段中指定。值越大，优先级越高。

PriorityClass 对象的值可以是小于或者等于 10 亿的 32 位任意整数值。更大的数值被保留给那些通常不应该取代或者驱逐的关键的系统级 Pod 使用。集群管理员应该为它们想要的每个此类映射创建一个 PriorityClass 对象。

PriorityClass 还有两个可选的字段：globalDefault 和 description。globalDefault 表示 PriorityClass 的值应该给那些没有设置 PriorityClassName 的 Pod 使用。整个系统只能存在一个 globalDefault 设置为 true 的 PriorityClass。如果没有任何 globalDefault 为 true 的 PriorityClass 存在，那么，那些没有设置 PriorityClassName 的 Pod 的优先级将为 0。

description 字段的值可以是任意的字符串。它向所有集群用户描述应该在什么时候使用这个 PriorityClass。

注 1：如果您升级已经存在的集群环境，并且启用了该功能，那么，那些已经存在系统里面的 Pod 的优先级将会设置为 0。

注 2：此外，将一个 PriorityClass 的 globalDefault 设置为 true，不会改变系统中已经存在的 Pod 的优先级。也就是说，PriorityClass 的值只能用于在 PriorityClass 添加之后创建的那些 Pod 当中。

注 3：如果您删除一个 PriorityClass，那些使用了该 PriorityClass 的 Pod 将会保持不变，但是，该 PriorityClass 的名称不能在新创建的 Pod 里面使用。

PriorityClass 示例

```
apiVersion: v1
kind: PriorityClass
metadata:
   name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

Pod priority

有了一个或者多个 PriorityClass 之后，您在创建 Pod 时就能在模板文件中指定需要使用的 PriorityClass 的名称。优先级准入控制器通过 priorityClassName 字段查找优先级数值并且填入 Pod 中。如果没有找到相应的 PriorityClass，Pod 将会被拒绝创建。

下面的 YAML 是一个使用了前面创建的 PriorityClass 对 Pod 进行配置的示例。优先级准入控制器会检测配置文件，并将该 Pod 的优先级解析为 1000000。

```
apiVersion: v1
kind: Pod
metadata:
   name: nginx
   labels:
      env: test
spec:
   containers:
   - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
   priorityClassName: high-priority
```

抢占

Pod 生成后，会进入一个队列等待调度。scheduler 从队列中选择一个 Pod，然后尝试将其调度到某个节点上。如果没有任何节点能够满足 Pod 指定的所有要求，对于这个挂起的 Pod，抢占逻辑就会被触发。当前假设我们把挂起的 Pod 称之为 P。抢占逻辑会尝试查找一个节点，在该节点上移除一个或多个比 P 优先级低的 Pod 后， P 能够调度到这个节点上。如果节点找到了，部分优先级低的 Pod 就会从该节点删除。Pod 消失后，P 就能被调度到这个节点上了。

跨节点抢占

假定节点 N 启用了抢占功能，以便我们能够把挂起的 Pod P 调度到节点 N 上。只有其它节点的 Pod 被抢占时，P 才有可能被调度到节点 N 上面。下面是一个示例：

- Pod P 正在考虑节点 N。
- Pod Q 正运行在跟节点 N 同区的另外一个节点上。
- Pod P 跟 Pod Q 之间有反亲和性。
- 在这个区域内没有跟 Pod P 具备反亲和性的其它 Pod。
- 为了将 Pod P 调度到节点 N 上，Pod Q 需要被抢占掉，但是 scheduler 不能执行跨节点的抢占。因此，节点 N 将被视为不可调度节点。

如果将 Pod Q 从它的节点移除，反亲和性随之消失，那么 Pod P 就有可能被调度到节点 N 上。

如果找到一个性能合理的算法，我们可以考虑在将来的版本中增加跨节点抢占。在这一点上，我们不能承诺任何东西，beta 或者 GA 版本也不会因为跨节点抢占功能而有所阻滞。

参考：https://k8smeetup.github.io/docs/concepts/configuration/pod-priority-preemption/

Pod 分配内存资源

这篇教程指导如何给容器分配申请的内存和内存限制。我们保证让容器获得足够的内存 资源，但是不允许它使用超过限制的资源。

配置内存申请和限制

给容器配置内存申请，只要在容器的配置文件里添加 resources:requests 就可以了。配置限制的话， 则是添加 resources:limits。

本实验，我们创建包含一个容器的 Pod，这个容器申请 100M 的内存，并且内存限制设置为 200M，下面 是配置文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
spec:
  containers:
  - name: memory-demo-ctr
```

```
    image: vish/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    args:
    - -mem-total
    - 150Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

在这个配置文件里，args 代码段提供了容器所需的参数。-mem-total 150Mi 告诉容器尝试申请 150M 的内存。

创建 Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit.yaml --namespace=mem-example
```

验证 Pod 的容器是否正常运行:

```
kubectl get pod memory-demo --namespace=mem-example
```

查看 Pod 的详细信息:

```
kubectl get pod memory-demo --output=yaml --namespace=mem-example
```

这个输出显示了 Pod 里的容器申请了 100M 的内存和 200M 的内存限制。

```
...
resources:
  limits:
    memory: 200Mi
  requests:
    memory: 100Mi
...
```

启动 proxy 以便我们可以访问 Heapster 服务：

```
kubectl proxy
```

在另外一个命令行窗口，从 Heapster 服务获取内存使用情况：

curl
http://localhost:8001/api/v1/proxy/namespaces/kube-system/services/heapster/api/v1/model/namespaces/mem-example/pods/memory-demo/metrics/memory/usage

这个输出显示了 Pod 正在使用 162,900,000 字节的内存，大概就是 150M。这很明显超过了申请 的 100M,但是还没达到 200M 的限制。


```
{
  "timestamp": "2017-06-20T18:54:00Z",
  "value": 162856960
}
```


删除 Pod:

kubectl delete pod memory-demo --namespace=mem-example

超出容器的内存限制

只要节点有足够的内存资源，那容器就可以使用超过其申请的内存，但是不允许容器使用超过其限制的 资源。如果容器分配了超过限制的内存，这个容器将会被优先结束。如果容器持续使用超过限制的内存， 这个容器就会被终结。如果一个结束的容器允许重启，kubelet 就会重启他，但是会出现其他类型的运行错误。

本实验，我们创建一个 Pod 尝试分配超过其限制的内存，下面的这个 Pod 的配置文档，它申请 50M 的内存， 内存限制设置为 100M。

```
apiVersion: v1
kind: Pod
metadata:
   name: memory-demo-2
spec:
   containers:
   - name: memory-demo-2-ctr
      image: vish/stress
      resources:
         requests:
            memory: 50Mi
         limits:
            memory: "100Mi"
      args:
      - -mem-total
```

```
    - 250Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

在配置文件里的 args 段里，可以看到容器尝试分配 250M 的内存，超过了限制的 100M。

创建 Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-2.yaml
--namespace=mem-example
```

查看 Pod 的详细信息:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

这时候，容器可能会运行，也可能会被杀掉。如果容器还没被杀掉，重复之前的命令直至 你看到这个容器被杀掉：

```
NAME                     READY        STATUS           RESTARTS      AGE
memory-demo-2      0/1            OOMKilled      1                 24s
```

查看容器更详细的信息:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

这个输出显示了容器被杀掉因为超出了内存限制。

```
lastState:
    terminated:
      containerID:
docker://65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f
      exitCode: 137
      finishedAt: 2017-06-20T20:52:19Z
      reason: OOMKilled
      startedAt: null
```

本实验里的容器可以自动重启，因此 kubelet 会再去启动它。输入多几次这个命令看看它是怎么 被杀掉又被启动的：

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

这个输出显示了容器被杀掉，被启动，又被杀掉，又被启动的过程：

```
stevepe@sperry-1:~/steveperry-53.github.io$ kubectl get pod memory-demo-2
--namespace=mem-example
NAME                READY     STATUS      RESTARTS   AGE
memory-demo-2       0/1       OOMKilled   1          37s
stevepe@sperry-1:~/steveperry-53.github.io$ kubectl get pod memory-demo-2
--namespace=mem-example
NAME                READY     STATUS    RESTARTS   AGE
memory-demo-2       1/1       Running   2          40s
```

查看 Pod 的历史详细信息:

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

这个输出显示了 Pod 一直重复着被杀掉又被启动的过程:

```
... Normal  Created     Created container with id
66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... Warning BackOff     Back-off restarting failed container
```

查看集群里节点的详细信息:

```
kubectl describe nodes
```

输出里面记录了容器被杀掉是因为一个超出内存的状况出现:

```
Warning OOMKilling  Memory cgroup out of memory: Kill process 4481 (stress) score
1994 or sacrifice child
```

删除 Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

配置超出节点能力范围的内存申请

内存的申请和限制是针对容器本身的，但是认为 Pod 也有容器的申请和限制是一个很有帮助的想法。 Pod 申请的内存就是 Pod 里容器申请的内存总和，类似的，Pod 的内存限制就是 Pod 里所有容器的 内存限制的总和。

Pod 的调度策略是基于请求的，只有当节点满足 Pod 的内存申请时，才会将 Pod 调度到合适的节点上。

在这个实验里，我们创建一个申请超大内存的 Pod，超过了集群里任何一个节点的可用内存资源。 这个容器申请了 1000G 的内存，这个应该会超过你集群里能提供的数量。

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
spec:
  containers:
  - name: memory-demo-3-ctr
    image: vish/stress
    resources:
      limits:
        memory: "1000Gi"
      requests:
        memory: "1000Gi"
    args:
    - -mem-total
    - 150Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

创建 Pod:

```
kubectl create -f
https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-3.yaml
--namespace=mem-example
```

查看 Pod 的状态:

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

输出显示 Pod 的状态是 Pending，因为 Pod 不会被调度到任何节点，所有它会一直保持在 Pending 状态下。

```
kubectl get pod memory-demo-3 --namespace=mem-example
NAME                READY       STATUS       RESTARTS     AGE
memory-demo-3     0/1           Pending     0                25s
```

查看 Pod 的详细信息包括事件记录

```
kubectl describe pod memory-demo-3 --namespace=mem-example
```

这个输出显示容器不会被调度因为节点上没有足够的内存：

Events:
  ... Reason   Message
         ------   -------
  ... FailedScheduling No nodes are available that match all of the following predicates:: Insufficient memory (3).

内存单位

内存资源是以字节为单位的, 可以表示为纯整数或者固定的十进制数字, 后缀可以是 E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki.比如，下面几种写法表示相同的数值：alue:

128974848, 129e6, 129M , 123Mi
删除 Pod:

kubectl delete pod memory-demo-3 --namespace=mem-example

如果不配置内存限制

如果不给容器配置内存限制，那下面的任意一种情况可能会出现：

- 容器使用内存资源没有上限，容器可以使用当前节点上所有可用的内存资源。
- 容器所运行的命名空间有默认内存限制，容器会自动继承默认的限制。集群管理员可以使用这个文档 LimitRange 来配置默认的内存限制。

内存申请和限制的原因

通过配置容器的内存申请和限制，你可以更加有效充分的使用集群里内存资源。配置较少的内存申请， 可以让 Pod 跟任意被调度。设置超过内存申请的限制，可以达到以下效果：

- Pod 可以在负载高峰时更加充分利用内存。
- 可以将 Pod 的内存使用限制在比较合理的范围。

清理

删除命名空间，这会顺便删除命名空间里的 Pod。

kubectl delete namespace mem-example

参考：
https://k8smeetup.github.io/docs/tasks/configure-pod-container/assign-memory-resource/

Pod 分配 CPU 资源

这个教程指导如何给容器分配请求的 CPU 资源和配置 CPU 资源限制，我们保证容器可以拥有 所申请的 CPU 资源，但是并不允许它使用超过限制的 CPU 资源。

集群里的每个节点至少需要 1 个 CPU。

这篇教程里的少数步骤可能要求你的集群运行着 Heapster 如果你没有 Heapster，也可以完成大部分步骤，就算跳过 Heapster 的那些步骤，也不见得会有什么问题。

判断 Heapster 服务是否正常运行，执行以下命令：

kubectl get services --namespace=kube-system

如果 heapster 正常运行，命令的输出应该类似下面这样:

NAMESPACE NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kube-system heapster 10.11.240.9 <none> 80/TCP 6d

创建一个命名空间

创建一个命名空间，可以确保你在这个实验里所创建的资源都会被有效隔离， 不会影响你的集群。

kubectl create namespace cpu-example

声明一个 CPU 申请和限制

给容器声明一个 CPU 请求，只要在容器的配置文件里包含这么一句 resources:requests 就可以， 声明一个 CPU 限制，则是这么一句 resources:limits.

在这个实验里, 我们会创建一个只有一个容器的 Pod, 这个容器申请 0.5 个 CPU, 并且 CPU 限制设置为 1. 下面是配置文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
spec:
  containers:
```

```
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
    args:
    - -cpus
    - "2"
```

在这个配置文件里，整个 args 段提供了容器所需的参数。 -cpus "2"代码告诉容器尝试使用 2 个 CPU 资源。

创建 Pod:

```
kubectl create -f
https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit.yaml
--namespace=cpu-example
```

验证 Pod 的容器是否正常运行:

```
kubectl get pod cpu-demo --namespace=cpu-example
```

查看 Pod 的详细信息:

```
kubectl get pod cpu-demo --output=yaml --namespace=cpu-example
```

输出显示了这个 Pod 里的容器申请了 500m 的 cpu，同时 CPU 用量限制为 1.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 500m
```

启用 proxy 以便访问 heapster 服务：

```
kubectl proxy
```

在另外一个命令窗口里，从 heapster 服务读取 CPU 使用率。

```
curl
http://localhost:8001/api/v1/proxy/namespaces/kube-system/services/heapster/api/v1/
```

model/namespaces/cpu-example/pods/cpu-demo/metrics/cpu/usage_rate
输出显示 Pod 目前使用 974m 的 cpu，这个刚好比配置文件里限制的 1 小一点点。


```
{
 "timestamp": "2017-06-22T18:48:00Z",
 "value": 974
}
```


还记得我吗设置了-cpu "2"，这样让容器尝试去使用 2 个 CPU，但是容器却只被运行使用 1 一个， 因为容器的 CPU 使用被限制了，因为容器尝试去使用超过其限制的 CPU 资源。

注意: 有另外一个可能的解释为什么 CPU 会被限制。因为这个节点可能没有足够的 CPU 资源，还记得我们 这个实验的前提条件是每个节点都有至少一个 CPU，如果你的容器所运行的节点只有 1 个 CPU，那容器就无法 使用超过 1 个的 CPU 资源，这跟 CPU 配置上的限制以及没关系了。

CPU 单位

CPU 资源是以 CPU 单位来计算的，一个 CPU，对于 Kubernetes 而言，相当于:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

小数值也是允许的，一个容器申请 0.5 个 CPU，就相当于其他容器申请 1 个 CPU 的一半，你也可以加个后缀 m 表示千分之一的概念。比如说 100m 的 CPU，100 豪的 CPU 和 0.1 个 CPU 都是一样的。但是不支持精度超过 1M 的。

CPU 通常都是以绝对值来申请的，绝对不能是一个相对的数值；0.1 对于单核，双核，48 核的 CPU 都是一样的。

删除 Pod:

kubectl delete pod cpu-demo --namespace=cpu-example

请求的 CPU 超出了节点的能力范围

CPU 资源的请求和限制是用于容器上面的，但是认为 POD 也有 CPU 资源的申请和限制，这种思想会很有帮助。 Pod 的 CPU 申请可以看作 Pod 里的所有容器的 CPU 资源申请的总和，类似的，Pod 的 CPU 限制就可以看出 Pod 里 所有容器的 CPU 资源限制的总和。

Pod 调度是基于请求的，只有当 Node 的 CPU 资源可以满足 Pod 的需求的时候，Pod 才会被调度到这个 Node 上面。

在这个实验当中，我们创建一个 Pod 请求超大的 CPU 资源，超过了集群里任何一个 node 所能提供的资源。 下面这个配置文件，创建一个包含一个容器的 Pod。这个容器申请了 100 个 CPU，这应该会超出你集群里 任何一个节点的 CPU 资源。

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
spec:
  containers:
  - name: cpu-demo-ctr-2
    image: vish/stress
    resources:
      limits:
        cpu: "100"
      requests:
        cpu: "100"
    args:
    - -cpus
    - "2"
```

创建 Pod:

```
kubectl create -f
https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit-2.yaml
--namespace=cpu-example
```

查看 Pod 的状态:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

这个输出显示 Pod 正处在 Pending 状态，那是因为这个 Pod 并不会被调度到任何节点上，所以它会 一直保持这种状态。

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
NAME              READY       STATUS       RESTARTS     AGE
cpu-demo-2     0/1            Pending     0                7m
```

查看 Pod 的详细信息，包括记录的事件：

```
kubectl describe pod cpu-demo-2 --namespace=cpu-example
```

这个输出显示了容器无法被调度因为节点上没有足够的 CPU 资源：

```
Events:
  Reason            Message
  ------            -------
  FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient cpu (3).
```

删除 Pod:

```
kubectl delete pod cpu-demo-2 --namespace=cpu-example
```

如果不指定 CPU 限额呢

如果你不指定容器的 CPU 限额，那下面所描述的其中一种情况会出现：

- 容器使用 CPU 资源没有上限，它可以使用它运行的 Node 上所有的 CPU 资源。
- 容器所运行的命名空间有默认的 CPU 限制，这个容器就自动继承了这个限制。集群管理可以使用 限额范围 来指定一个默认的 CPU 限额。

设置 CPU 申请和限制的动机

通过配置集群里的容器的 CPU 资源申请和限制，我们可以更好的利用集群中各个节点的 CPU 资源。 保持 Pod 的 CPU 请求不太高，这样才能更好的被调度。设置一个大于 CPU 请求的限制，可以获得以下 两点优势：

- Pod 在业务高峰期能获取到足够的 CPU 资源。
- 能将 Pod 在需求高峰期能使用的 CPU 资源限制在合理范围。

清理

删除你的命名空间:

```
kubectl delete namespace cpu-example
```

参考：
https://k8smeetup.github.io/docs/tasks/configure-pod-container/assign-cpu-resource/

Pod 配置服务质量等级

这篇教程指导如何给 Pod 配置特定的服务质量（QoS）等级。Kubernetes 使用 QoS 等级来确定何时调度和终结 Pod 。

QoS 等级

当 Kubernetes 创建一个 Pod 时，它就会给这个 Pod 分配一个 QoS 等级：

- Guaranteed
- Burstable
- BestEffort

创建一个命名空间

创建一个命名空间，以便将我们实验需求的资源与集群其他资源隔离开。

kubectl create namespace qos-example

创建一个 Pod 并分配 QoS 等级为 Guaranteed

想要给 Pod 分配 QoS 等级为 Guaranteed:

- Pod 里的每个容器都必须有内存限制和请求，而且必须是一样的。
- Pod 里的每个容器都必须有 CPU 限制和请求，而且必须是一样的。

这是一个含有一个容器的 Pod 的配置文件。这个容器配置了内存限制和请求，都是 200MB。它还有 CPU 限制和请求，都是 700 millicpu:

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

创建 Pod:

kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod.yaml --namespace=qos-example

查看 Pod 的详细信息:

kubectl get pod qos-demo --namespace=qos-example --output=yaml

输出显示了 Kubernetes 给 Pod 配置的 QoS 等级为 Guaranteed 。也验证了容器的内存和 CPU 的限制都满足了它的请求。

```
spec:
  containers:
    ...
    resources:
      limits:
        cpu: 700m
        memory: 200Mi
      requests:
        cpu: 700m
        memory: 200Mi
...
  qosClass: Guaranteed
```

注意: 如果一个容器配置了内存限制，但是没有配置内存申请，那 Kubernetes 会自动给容器分配一个符合内存限制的请求。 类似的，如果容器有 CPU 限制，但是没有 CPU 申请，Kubernetes 也会自动分配一个符合限制的请求。

删除你的 Pod:

kubectl delete pod qos-demo --namespace=qos-example

创建一个 Pod 并分配 QoS 等级为 Burstable

当出现下面的情况时，则是一个 Pod 被分配了 QoS 等级为 Burstable :

- 该 Pod 不满足 QoS 等级 Guaranteed 的要求。
- Pod 里至少有一个容器有内存或者 CPU 请求。

这是 Pod 的配置文件，里面有一个容器。这个容器配置了 200MB 的内存限制和 100MB 的内存申请。

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-2
spec:
  containers:
  - name: qos-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

创建 Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-2.yaml --namespace=qos-example
```

查看 Pod 的详细信息:

```
kubectl get pod qos-demo-2 --namespace=qos-example --output=yaml
```

输出显示了 Kubernetes 给这个 Pod 配置了 QoS 等级为 Burstable.

```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: qos-demo-2-ctr
    resources:
      limits:
        memory: 200Mi
      requests:
        memory: 100Mi
...
  qosClass: Burstable
```

删除你的 Pod:

```
kubectl delete pod qos-demo-2 --namespace=qos-example
```

创建一个 Pod 并分配 QoS 等级为 BestEffort

要给一个 Pod 配置 BestEffort 的 QoS 等级，Pod 里的容器必须没有任何内存或者 CPU 的限制或请求。

下面是一个 Pod 的配置文件，包含一个容器。这个容器没有内存或者 CPU 的限制或者请求：

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-3
spec:
  containers:
  - name: qos-demo-3-ctr
    image: nginx
```

创建 Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-3.yaml --namespace=qos-example
```

查看 Pod 的详细信息:

```
kubectl get pod qos-demo-3 --namespace=qos-example --output=yaml
```

输出显示了 Kubernetes 给 Pod 配置的 QoS 等级是 BestEffort.

```
spec:
  containers:
    ...
    resources: {}
  ...
  qosClass: BestEffort
```

删除你的 Pod:

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

创建一个拥有两个容器的 Pod

这是一个含有两个容器的 Pod 的配置文件，其中一个容器指定了内存申请为 200MB ，另外一个没有任何申请或限制。

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: qos-demo-4
spec:
  containers:

  - name: qos-demo-4-ctr-1
    image: nginx
    resources:
      requests:
        memory: "200Mi"

  - name: qos-demo-4-ctr-2
    image: redis
```

注意到这个 Pod 满足了 QoS 等级 Burstable 的要求. 就是说，它不满足 Guaranteed 的要求，而且其中一个容器有内存请求。

创建 Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-4.yaml --namespace=qos-example
```

查看 Pod 的详细信息:

```
kubectl get pod qos-demo-4 --namespace=qos-example --output=yaml
```

输出显示了 Kubernetes 给 Pod 配置的 QoS 等级是 Burstable:

```
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
    ...
  qosClass: Burstable
```

删除你的 Pod:

```
kubectl delete pod qos-demo-4 --namespace=qos-example
```

清理

删除你的 namespace:

kubectl delete namespace qos-example

参考：
https://k8smeetup.github.io/docs/tasks/configure-pod-container/quality-service-pod/

Kubernetes 常用命令

kubectl 概述

kubectl 是用于运行 Kubernetes 集群命令的管理工具。

kubectl 与 Docker 命令关系

在本文中，我们将介绍 Kubernetes 命令行与 docker-cli 之间的一些关系。kubectl 工具，对于使用过 docker-cli 用户来说，kubectl 的设计感觉会很熟悉，但是也有一些明显的区别。本文的每个小结都讲解了一个 docker 子命令，并且列出与 kubectl 相对应命令。

docker run

如果运行 nginx 并且暴露？使用 kubectl run。

使用 docker：

```
$ docker run -d --restart=always -e DOMAIN=cluster --name nginx-app -p 80:80 nginx
a9ec34d9878748d2f33dc20cb25c714ff21da8d40558b45bfaec9955859075d0
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                         NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of     2 seconds ago
Up 2 seconds        0.0.0.0:80->80/tcp, 443/tcp   nginx-app
```

使用 kubectl：

```
# start the pod running nginx
$ kubectl run --image=nginx nginx-app --port=80 --env="DOMAIN=cluster"
deployment "nginx-app" created
```

kubectl run 在 Kubernetes 集群> = v1.2 上将创建名是 “nginx-app”的 Deployment。如果运行的是低的版本，则会创建一个 replication controllers，如果要了解低版本方式，请使用--generator=run/v1，将会创建 replication controllers。查看 kubectl run 更多详情。现在，我们可以使用上面创建的 Deployment 来暴露一个新的服务：

```
# expose a port through with a service
$ kubectl expose deployment nginx-app --port=80 --name=nginx-http
service "nginx-http" exposed
```

使用 kubectl 创建一个 Deployment，他能保证任何情况下有 N 个运行 nginx 的 pods（其中 N 是默认定义声明的副本数，默认为 1 个）。它同时会创建一个 Services，使用选择器匹配 Deployment’s selector。有关详细信息，请参阅快速入门。

默认情况下镜像在后台运行，类似于 docker run -d ...如果要在前台运行，请使用：

```
kubectl run [-i] [--tty] --attach <name> --image=<image>
```

要删除 Deployment （及其 pod），使用 kubectl delete deployment <name>

docker ps

如何列出当前运行的内容？参考 kubectl get。

使用 docker：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of     About an hour
ago     Up About an hour       0.0.0.0:80->80/tcp, 443/tcp     nginx-app
```

与 kubectl：

```
$ kubectl get po
```

```
NAME                    READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm         1/1       Running   0          1h
```

docker attach

如何连接已经运行在容器中的进程？参考 kubectl 附件

使用 docker：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of     8 minutes ago
Up 8 minutes        0.0.0.0:80->80/tcp, 443/tcp   nginx-app
$ docker attach a9ec34d98787
...
```

使用 kubectl：

```
$ kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm         1/1       Running   0          10m
$ kubectl attach -it nginx-app-5jyvm
...
```

docker exec

如何在容器中执行命令？参考 kubectl exec。

使用 docker：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of     8 minutes ago
Up 8 minutes        0.0.0.0:80->80/tcp, 443/tcp   nginx-app
$ docker exec a9ec34d98787 cat /etc/hostname
```

a9ec34d98787

使用 kubectl：

```
$ kubectl get po
NAME                    READY     STATUS    RESTARTS   AGE
nginx-app-5jyvm         1/1       Running   0          10m
$ kubectl exec nginx-app-5jyvm -- cat /etc/hostname
nginx-app-5jyvm
```

交互式命令呢？

使用 docker：

```
$ docker exec -ti a9ec34d98787 /bin/sh
# exit
```

使用 kubectl：

```
$ kubectl exec -ti nginx-app-5jyvm -- /bin/sh
# exit
```

有关更多信息，请参阅获取运行容器中的 Shell。

docker logs

如何查看进程打印 stdout / stderr？参考 kubectl logs。

使用 docker：

```
$ docker logs -f a9e
192.168.9.1 - - [14/Jul/2015:01:04:02 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.35.0" "-"
192.168.9.1 - - [14/Jul/2015:01:04:03 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.35.0" "-"
```

使用 kubectl：

```
$ kubectl logs -f nginx-app-zibvs
10.240.63.110 - - [14/Jul/2015:01:09:01 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.26.0" "-"
```

10.240.63.110 - - [14/Jul/2015:01:09:02 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.26.0" "-"

感觉是时候普及下 pods 和 containers 之间的微小差异了；默认情况下, 如果进程退出, pods 是不会终止，相反，它会重新启动该进程。这与 docker run 配置--restart=always 选项有一个主要区别。要查看以前在 Kubernetes 中运行的输出，请运行如下：

```
$ kubectl logs --previous nginx-app-zibvs
10.240.63.110 - - [14/Jul/2015:01:09:01 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.26.0" "-"
10.240.63.110 - - [14/Jul/2015:01:09:02 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.26.0" "-"
```

有关详细信息，请参阅日志和群集监视。

docker stop 和 docker rm

如何停止和删除正在运行的进程？参考 kubectl delete。

使用 docker

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                         NAMES
a9ec34d98787        nginx               "nginx -g 'daemon of     22 hours ago
Up 22 hours         0.0.0.0:80->80/tcp, 443/tcp   nginx-app
$ docker stop a9ec34d98787
a9ec34d98787
$ docker rm a9ec34d98787
a9ec34d98787
```

使用 kubectl：

```
$ kubectl get deployment nginx-app
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-app   1         1         1            1           2m
$ kubectl get po -l run=nginx-app
NAME                         READY     STATUS    RESTARTS   AGE
nginx-app-2883164633-aklf7   1/1       Running   0          2m
$ kubectl delete deployment nginx-app
```

```
deployment "nginx-app" deleted
$ kubectl get po -l run=nginx-app
# Return nothing
```

注意，不要直接删除 pod，使用 kubectl 请删除拥有该 pod 的 Deployment。如果直接删除 pod，则 Deployment 将会重新创建该 pod。

docker login

与 docker login 相比，在 kubectl 中没有直接相识的命令。如果对于 Kubernetes 在私有 registry 中使用感兴趣，请参阅使用私有 Registry。

docker version

如何获客户端和服务器的版本号？参考 kubectl version。

使用 docker：

```
$ docker version
Client version: 1.7.0
Client API version: 1.19
Go version (client): go1.4.2
Git commit (client): 0baf609
OS/Arch (client): linux/amd64
Server version: 1.7.0
Server API version: 1.19
Go version (server): go1.4.2
Git commit (server): 0baf609
OS/Arch (server): linux/amd64
```

使用 kubectl：

```
$ kubectl version
Client Version: version.Info{Major:"0", Minor:"20.1", GitVersion:"v0.20.1", GitCommit:"",
GitTreeState:"not a git tree"}
Server Version: version.Info{Major:"0", Minor:"21+",
GitVersion:"v0.21.1-411-g32699e873ae1ca-dirty",
GitCommit:"32699e873ae1caa01812e41de7eab28df4358ee4", GitTreeState:"dirty"}
```

docker info

如何获取有关配置和环境信息？参考 kubectl cluster-info。

使用 docker：

```
$ docker info
Containers: 40
Images: 168
Storage Driver: aufs
 Root Dir: /usr/local/google/docker/aufs
 Backing Filesystem: extfs
 Dirs: 248
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.13.0-53-generic
Operating System: Ubuntu 14.04.2 LTS
CPUs: 12
Total Memory: 31.32 GiB
Name: k8s-is-fun.mtv.corp.google.com
ID: ADUV:GCYR:B3VJ:HMPO:LNPQ:KD5S:YKFQ:76VN:IANZ:7TFV:ZBF4:BYJO
WARNING: No swap limit support
```

使用 kubectl：

```
$ kubectl cluster-info
Kubernetes master is running at https://108.59.85.141
KubeDNS is running at https://108.59.85.141/api/v1/namespaces/kube-system/services/kube-dns/proxy
KubeUI is running at https://108.59.85.141/api/v1/namespaces/kube-system/services/kube-ui/proxy
Grafana is running at https://108.59.85.141/api/v1/namespaces/kube-system/services/monitoring-grafana/proxy
Heapster is running at https://108.59.85.141/api/v1/namespaces/kube-system/services/monitoring-heapster/proxy
InfluxDB is running at https://108.59.85.141/api/v1/namespaces/kube-system/services/monitoring-influxdb/proxy
```

参考：http://docs.kubernetes.org.cn/70.html

常用命令

1、查看集群状态

kubectl version --short=true 查看客户端及服务端程序版本信息
kubectl cluster-info 查看集群信息

2、创建资源对象

kubectl run name --image=(镜像名) --replicas=(备份数) --port=(容器要暴露的端口) --labels=(设定自定义标签)
kubectl create -f **.yaml 陈述式对象配置管理方式
kubectl apply -f **.yaml 声明式对象配置管理方式（也适用于更新等）

3、查看资源对象

kubectl get namespace 查看命名空间
kubectl get pods,services -o wide (-o 输出格式 wide 表示 plain-text)
kubectl get pod -l "key=value,key=value" -n kube-system (-l 标签选择器(多个的话是与逻辑)，-n 指定命名空间，不指定默认 default)
kubectl get pod -l "key1 in (val1,val2),!key2" -L key (-l 基于集合的标签选择器，-L 查询结果显示标签) 注意：为了避免和 shell 解释器解析!,必须要为此类表达式使用单引号
kubectl get pod -w(-w 监视资源变动信息)

4、打印容器中日志信息

kubectl logs name -f -c container_name -n kube-system (-f 持续监控，-c 如果 pod 中只有一个容器不用加)
5、在容器中执行命令
kubectl exec name -c container_name -n kube-system -- 具体命令
kubectl exec -it pod_name /bin/sh 进入容器的交互式 shell

6、删除资源对象

kubectl delete [pods/services/deployments/...] name 删除指定资源对象
kubectl delete [pods/services/deployments/...] -l key=value -n kube-system 删除

kube-system 下指定标签的资源对象
kubectl delete [pods/services/deployments/...] --all -n kube-system 删除 kube-system 下所有资源对象
kubectl delete [pods/services/deployments/...] source_name --force --grace-period=0 -n kube-system 强制删除 Terminating 的资源对象
kubectl delete -f xx.yaml
kubectl apply -f xx.yaml --prune -l <labels>(一般不用这种方式删除)
kubectl delete rs rs_name --cascade=fale(默认删除控制器会同时删除其管控的所有 Pod 对象，加上 cascade=false 就只删除 rs)

7、更新资源对象

kubectl replace -f xx.yaml --force(--force 如果需要基于此前的配置文件进行替换，需要加上 force)

8、将服务暴露出去(创建 Service)

kubectl expose deployments/deployment_name --type="NodePort" --port=(要暴露的容器端口) --name=(Service 对象名字)

9、扩容和缩容

kubectl scale deployment/deployment_name --replicas=N
kubectl scale deployment/deployment_name --replicas=N --current-replicas=M 只有当前副本数等于 M 时才会执行扩容或者缩容

10、查看 API 版本

kubectl api-versions

11、在本地主机上为 API Server 启动一个代理网关

kubectl proxy --port=8080
之后就可以通过 curl 来对此套字节发起访问请求
curl localhost:8080/api/v1/namespaces/ | jq .items[].metadata.name (jq 可以对 json 进行过滤)

12、当定义资源配置文件时，不知道怎么定义的时候，可以查看某类型资源的配置字段解释

kubectl explain pods/deployments/...(二级对象可用类似于 pods.spec 这种方式查看)

13、查看某资源对象的配置文件

kubectl get source_type source_name -o yaml --export(--export 表示省略由系统生成的信息) 后面加 > file.yaml 就可以快速生成一个配置文件了

14、标签管理相关命令

kubectl label pods/pod_name key=value 添加标签,如果是修改的话需要后面添加 --overwrite
kubectl label nodes node_name key=value 给工作节点添加标签，后续可以使用 nodeSelector 来指定 pod 被调度到指定的工作节点上运行

15、注解管理相关命令

kubectl annotate pods pod_name key=value

16、patch 修改 Deployment 控制器进行控制器升级

kubectl patch deployment deployment-demo -p '{"spec": {"minReadySeconds": 5}}'(-p 以补丁形式更新补丁形式默认是 json)
kubectl set image deployments deployment-demo myapp=ikubernetes/myapp:v2 修改 depolyment 中的镜像文件
kubectl rollout status deployment deployment-demo 打印滚动更新过程中的状态信息
kubectl get deployments deployment-demo --watch 监控 deployment 的更新过程
kubectl kubectl rollout pause deployments deployment-demo 暂停更新
kubectl rollout resume deployments deployment-demo 继续更新
kubectl rollout history deployments deployment-demo 查看历史版本(能查到具体的历史需要在 apply 的时候加上--record 参数)
kubectl rollout undo deployments deployment-demo --to-revision=2 回滚到指定版本，不加--to-version 则回滚到上一个版本

17、查看所有 pod 列表， -n 后跟 namespace，查看指定的命名空间

kubectl get pod
kubectl get pod -n kube

kubectl get pod -o wide

18、查看 RC 和 service 列表， -o wide 查看详细信息

kubectl get rc,svc
kubectl get pod,svc -o wide
kubectl get pod <pod-name> -o yaml

19、显示 Node 的详细信息

kubectl describe node 192.168.0.212

20、显示 Pod 的详细信息，特别是查看 pod 无法创建的时候的日志

kubectl describe pod <pod-name>
eg:
kubectl describe pod redis-master-tqds9

21、根据 yaml 创建资源，apply 可以重复执行，create 不行

kubectl create -f pod.yaml
kubectl apply -f pod.yaml

22、基于 pod.yaml 定义的名称删除 pod

kubectl delete -f pod.yaml

23、删除所有包含某个 label 的 pod 和 service

kubectl delete pod,svc -l name=<label-name>

24、删除所有 Pod

kubectl delete pod --all

25、查看 endpoint 列表

kubectl get endpoints

26、执行 pod 的 date 命令

kubectl exec <pod-name> -- date
kubectl exec <pod-name> -- bash
kubectl exec <pod-name> -- ping 10.24.51.9

27、通过 bash 获得 pod 中某个容器的 TTY，相当于登录容器

kubectl exec -it <pod-name> -c <container-name> -- bash
eg:
kubectl exec -it redis-master-cln81 -- bash

28、查看容器的日志

kubectl logs <pod-name>
kubectl logs -f <pod-name> # 实时查看日志
kubectl log <pod-name> -c <container_name> # 若 pod 只有一个容器，可以不加 -c

29、查看注释

kubectl explain pod
kubectl explain pod.apiVersion

30、查看节点 labels

kubectl get node --show-labels

API

API 概述

REST API 是 Kubernetes 系统的重要部分，组件之间的所有操作和通信均由 API Server 处理的 REST API 调用，大多数情况下，API 定义和实现都符合标准的 HTTP REST 格式，可以通过 kubectl 命令管理工具或其他命令行工具来执行。

API 版本

为了在兼容旧版本的同时不断升级新的 API，Kubernetes 支持多种 API 版本，每种 API 版本都有不同的 API 路径，例如/api/v1 或 /apis/extensions/v1beta1。

API 版本规则是通过基于 API level 选择版本，而不是基于资源和域级别选择，是为了确保 API 能够描述一个清晰的连续的系统资源和行为的视图，能够控制访问的整个过程和控制实验性 API 的访问。

JSON 和 Protobuf 序列化模式遵循相同的模式变化原则，以下所有描述都涵盖了这两种模式。
需要注意，API 版本和软件的版本没有直接关系，不同 API 版本有不同程度稳定性，API 文档中详细描述了每个级别的标准。

Alpha 级别：

- 包含 alpha 名称的版本（例如 v1alpha1）。
- 该软件可能包含错误。启用一个功能可能会导致 bug。默认情况下，功能可能会被禁用。
- 随时可能会丢弃对该功能的支持，恕不另行通知。
- API 可能在以后的软件版本中以不兼容的方式更改，恕不另行通知。
- 该软件建议仅在短期测试集群中使用，因为错误的风险增加和缺乏长期支持。

Beta 级别：

- 包含 beta 名称的版本（例如 v2beta3）。
- 该软件经过很好的测试。启用功能被认为是安全的。默认情况下功能是开启的。
- 细节可能会改变，但功能在后续版本不会被删除
- 对象的模式或语义在随后的 beta 版本或 Stable 版本中可能以不兼容的方式发生变化。如果这种情况发生时，官方会提供迁移操作指南。这可能需要删除、编辑和重新创建 API 对象。
- 该版本在后续可能会更改一些不兼容地方，所以建议用于非关键业务，如果你有多个可以独立升级的集群，你也可以放宽此限制。

- 大家使用过的 Beta 版本后，可以多给社区反馈，如果此版本在后续更新后将不会有太大变化。

Stable 级别：

- 该版本名称命名方式：vX 这里 X 是一个整数。
- Stable 版本的功能特性，将出现在后续发布的软件版本中。

API groups

API groups 使得 Kubernetes API 的扩展更加方便。API groups 是在 REST 路径和序列化对象的 apiVersion 字段中被指定。

目前，有几个 API groups 在使用:

- 核心(also called legacy)组，REST 路径在/api/v1,但此路径不是固定的，v1 是当前的版本。与之相对应的代码里面的 apiVersion 字段的值为 v1。
- Named Groups，REST 路径被指定在/apis/$GROUP_NAME/$VERSION 中，并使用 apiVersion: $GROUP_NAME/$VERSION（例如 apiVersion: batch/v1）。在 Kubernetes API 参考引用中可以看到 API Groups 的完整列表。

使用自定义资源扩展 API 的两种方法：

1. CustomResourceDefinition 为有基本 CRUD 需求用户提供。
2. 即将推出：需要有完整的 Kubernetes API 语义的用户，可以实现自定义的 api server，并使用聚合器来无缝连接客户端。

启用 API Groups

可以使用--runtime-config 在 api server 上设置来启用或禁用某些资源和 API Groups。
--runtime-config 可以使用逗号分隔值。例如，要禁用 batch / v1，set
--runtime-config=batch/v1=false, to enable batch/v2alpha1, set
--runtime-config=batch/v2alpha1。该标签接受逗号分隔的一组 key = value 对，描述了运行时的 api server 配置。

提示：启用和禁用 Groups 或资源需要重新启动 apiserver 和 controller-manager 确保 --runtime-config 更改生效。

启用组中的资源

DaemonSets，Deployments，Horizo ntalPodAutoscalers，Ingress，Jobs 和 ReplicaSets，都是默认启用的。可以通过--runtime-config 在 api server 上设置来启用其他扩展资源。--runtime-config 接受逗号来分隔值。例如，要禁用 deployments 和 jobs，请设置
--runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/jobs=false

使用自定义资源扩展 API

注意：TPR 已经停止维护，kubernetes 1.7 及以上版本请使用 CRD。

自定义资源是对 Kubernetes API 的扩展，kubernetes 中的每个资源都是一个 API 对象的集合，例如我们在 YAML 文件里定义的那些 spec 都是对 kubernetes 中的资源对象的定义，所有的自定义资源可以跟 kubernetes 中内建的资源一样使用 kubectl 操作。

自定义资源

Kubernetes1.6 版本中包含一个内建的资源叫做 TPR（ThirdPartyResource），可以用它来创建自定义资源, 但该资源在 kubernetes1.7 中版本已被 CRD（CustomResourceDefinition）取代。

扩展 API

自定义资源实际上是为了扩展 kubernetes 的 API，向 kubenetes API 中增加新类型，可以使用以下三种方式：

- 修改 kubenetes 的源码，显然难度比较高，也不太合适
- 创建自定义 API server 并聚合到 API 中
- 1.7 以下版本编写 TPR，kubernetes1.7 及以上版本用 CRD

编写自定义资源是扩展 kubernetes API 的最简单的方式，是否编写自定义资源来扩展 API 请参考 Should I add a custom resource to my Kubernetes Cluster?，行动前请先考虑以下几点：

- 你的 API 是否属于声明式的
- 是否想使用 kubectl 命令来管理
- 是否要作为 kubenretes 中的对象类型来管理，同时显示在 kubernetes dashboard 上
- 是否可以遵守 kubernetes 的 API 规则限制，例如 URL 和 API group、namespace 限制
- 是否可以接受该 API 只能作用于集群或者 namespace 范围
- 想要复用 kubernetes API 的公共功能，比如 CRUD、watch、内置的认证和授权等

如果这些都不是你想要的，那么你可以开发一个独立的 API。

TPR

注意：TPR 已经停止维护，kubernetes 1.7 及以上版本请使用 CRD。

假如我们要创建一个名为 cron-tab.stable.example.com 的 TPR，yaml 文件定义如下：

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: cron-tab.stable.example.com
description: "A specification of a Pod to run on a cron style schedule"
versions:
- name: v1
```

然后使用 kubectl create 命令创建该资源，这样就可以创建出一个 API 端点 /apis/stable.example.com/v1/namespaces/<namespace>/crontabs/...。

下面是在 Linkerd 中的一个实际应用，Linkerd 中的一个名为 namerd 的组件使用了 TPR，定义如下：

```
---
kind: ThirdPartyResource
apiVersion: extensions/v1beta1
metadata:
  name: d-tab.l5d.io
description: stores dtabs used by namerd
versions:
- name: v1alpha1
```

CRD

参考下面的 CRD，resourcedefinition.yaml：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # 名称必须符合下面的格式：<plural>.<group>
  name: crontabs.stable.example.com
spec:
  # REST API 使用的组名称：/apis/<group>/<version>
  group: stable.example.com
```

```
  # REST API 使用的版本号：/apis/<group>/<version>
  version: v1
  # Namespaced 或 Cluster
  scope: Namespaced
  names:
    # URL 中使用的复数名称: /apis/<group>/<version>/<plural>
    plural: crontabs
    # CLI 中使用的单数名称
    singular: crontab
    # CamelCased 格式的单数类型。在清单文件中使用
    kind: CronTab
    # CLI 中使用的资源简称
    shortNames:
    - ct
```

创建该 CRD：

```
kubectl create -f resourcedefinition.yaml
```

访问 RESTful API 端点如 http://172.20.0.113:8080 将看到如下 API 端点已创建：

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

创建自定义对象

如下所示：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

引用该自定义资源的 API 创建对象。

终止器

可以为自定义对象添加一个终止器，如下所示：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
```

```
finalizers:
- finalizer.stable.example.com
```

删除自定义对象前，异步执行的钩子。对于具有终止器的一个对象，删除请求仅仅是为 metadata.deletionTimestamp 字段设置一个值，而不是删除它，这将触发监控该对象的控制器执行他们所能处理的任意终止器。

详情参考：Extend the Kubernetes API with CustomResourceDefinitions

自定义控制器

单纯设置了自定义资源，并没有什么用，只有跟自定义控制器结合起来，才能将资源对象中的声明式 API 翻译成用户所期望的状态。自定义控制器可以用来管理任何资源类型，但是一般是跟自定义资源结合使用。

请参考使用 Operator 模式，该模式可以让开发者将自己的领域知识转换成特定的 kubenretes API 扩展。

API server 聚合

Aggregated（聚合的）API server 是为了将原来的 API server 这个巨石（monolithic）应用给拆分成，为了方便用户开发自己的 API server 集成进来，而不用直接修改 kubernetes 官方仓库的代码，这样一来也能将 API server 解耦，方便用户使用实验特性。这些 API server 可以跟 core API server 无缝衔接，使用 kubectl 也可以管理它们。

详情参考 Aggregated API Server。

参考：https://jimmysong.io/kubernetes-handbook/concepts/custom-resource.html

Deployment

Deployment 为 Pod 和 Replica Set（升级版的 Replication Controller）提供声明式更新。

你只需要在 Deployment 中描述您想要的目标状态是什么，Deployment controller 就会帮您将 Pod 和 ReplicaSet 的实际状态改变到您的目标状态。您可以定义一个全新的 Deployment 来创建 ReplicaSet 或者删除已有的 Deployment 并创建一个新的来替换。

注意：您不该手动管理由 Deployment 创建的 Replica Set，否则您就篡越了 Deployment controller 的职责！下文罗列了 Deployment 对象中已经覆盖了所有的用例。如果未有覆盖您所有需要的用例，请直接在 Kubernetes 的代码库中提 issue。

典型的用例如下：

- 使用 Deployment 来创建 ReplicaSet。ReplicaSet 在后台创建 pod。检查启动状态，看它是成功还是失败。
- 然后，通过更新 Deployment 的 PodTemplateSpec 字段来声明 Pod 的新状态。这会创建一个新的 ReplicaSet，Deployment 会按照控制的速率将 pod 从旧的 ReplicaSet 移动到新的 ReplicaSet 中。
- 如果当前状态不稳定，回滚到之前的 Deployment revision。每次回滚都会更新 Deployment 的 revision。
- 扩容 Deployment 以足更高的负载。
- 暂停 Deployment 来应用 PodTemplateSpec 的多个修复，然后恢复上线。
- 根据 Deployment 的状态判断上线是否 hang 住了。
- 清除旧的不必要的 ReplicaSet。

创建 Deployment

下面是一个 Deployment 示例，它创建了一个 ReplicaSet 来启动 3 个 nginx pod。

下载示例文件并执行命令：

```
$ kubectl create -f https://kubernetes.io/docs/user-guide/nginx-deployment.yaml --record
deployment "nginx-deployment" created
```

将 kubectl 的 --record 的 flag 设置为 true 可以在 annotation 中记录当前命令创建或者升级了该资源。这在未来会很有用，例如，查看在每个 Deployment revision 中执行了哪些命令。

然后立即执行 get 将获得如下结果：

```
$ kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    3         0         0            0           1s
```

输出结果表明我们希望的 repalica 数是 3（根据 deployment 中的.spec.replicas 配置）当前 replica 数（ .status.replicas）是 0，最新的 replica 数（.status.updatedReplicas）是 0，可用的 replica 数（.status.availableReplicas）是 0。

过几秒后再执行 get 命令，将获得如下输出：

```
$ kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
```

nginx-deployment 3 3 3 3 18s
我们可以看到 Deployment 已经创建了 3 个 replica, 所有的 replica 都已经是最新的了(包含最新的 pod template），可用的（根据 Deployment 中的.spec.minReadySeconds 声明，处于已就绪状态的 pod 的最少个数）。执行 kubectl get rs 和 kubectl get pods 会显示 Replica Set（RS）和 Pod 已创建。

$ kubectl get rs
NAME DESIRED CURRENT READY AGE
nginx-deployment-2035384211 3 3 0 18s
您可能会注意到 ReplicaSet 的名字总是<Deployment 的名字>-<pod template 的 hash 值>。

$ kubectl get pods --show-labels
NAME READY STATUS RESTARTS AGE
LABELS
nginx-deployment-2035384211-7ci7o 1/1 Running 0 18s
app=nginx,pod-template-hash=2035384211
nginx-deployment-2035384211-kzszj 1/1 Running 0 18s
app=nginx,pod-template-hash=2035384211
nginx-deployment-2035384211-qqcnn 1/1 Running 0 18s
app=nginx,pod-template-hash=2035384211
刚创建的 Replica Set 将保证总是有 3 个 nginx 的 pod 存在。

注意： 您必须在 Deployment 中的 selector 指定正确的 pod template label（在该示例中是 app = nginx），不要跟其他的 controller 的 selector 中指定的 pod template label 搞混了（包括 Deployment、Replica Set、Replication Controller 等）。Kubernetes 本身并不会阻止您任意指定 pod template label ，但是如果您真的这么做了，这些 controller 之间会相互打架，并可能导致不正确的行为。

Pod-template-hash label

注意：这个 label 不是用户指定的!

注意上面示例输出中的 pod label 里的 pod-template-hash label。当 Deployment 创建或者接管 ReplicaSet 时，Deployment controller 会自动为 Pod 添加 pod-template-hash label。这样做的目的是防止 Deployment 的子 ReplicaSet 的 pod 名字重复。通过将 ReplicaSet 的 PodTemplate 进行哈希散列，使用生成的哈希值作为 label 的值，并添加到 ReplicaSet selector 里、 pod template label 和 ReplicaSet 管理中的 Pod 上。

更新 Deployment

注意： Deployment 的 rollout 当且仅当 Deployment 的 pod template （例如.spec.template）中的 label 更新或者镜像更改时被触发。其他更新，例如扩容 Deployment 不会触发 rollout。

假如我们现在想要让 nginx pod 使用 nginx:1.9.1 的镜像来代替原来的 nginx:1.7.9 的镜像。

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

我们可以使用 edit 命令来编辑 Deployment，修改 .spec.template.spec.containers[0].image ，将 nginx:1.7.9 改写成 nginx:1.9.1。

```
$ kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
```

查看 rollout 的状态，只要执行：

```
$ kubectl rollout status deployment/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

Rollout 成功后，get Deployment：

```
$ kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    3         3         3            3           36s
```

UP-TO-DATE 的 replica 的数目已经达到了配置中要求的数目。

CURRENT 的 replica 数表示 Deployment 管理的 replica 数量，AVAILABLE 的 replica 数是当前可用的 replica 数量。

我们通过执行 kubectl get rs 可以看到 Deployment 更新了 Pod，通过创建一个新的 ReplicaSet 并扩容了 3 个 replica，同时将原来的 ReplicaSet 缩容到了 0 个 replica。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         0       6s
nginx-deployment-2035384211   0         0         0       36s
```

执行 get pods 只会看到当前的新的 pod:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1564180365-khku8   1/1       Running   0          14s
nginx-deployment-1564180365-nacti   1/1       Running   0          14s
nginx-deployment-1564180365-z9gth   1/1       Running   0          14s
```

下次更新这些 pod 的时候，只需要更新 Deployment 中的 pod 的 template 即可。

Deployment 可以保证在升级时只有一定数量的 Pod 是 down 的。默认的，它会确保至少有比期望的 Pod 数量少一个是 up 状态（最多一个不可用）。

Deployment 同时也可以确保只创建出超过期望数量的一定数量的 Pod。默认的，它会确保最多比期望的 Pod 数量多一个的 Pod 是 up 的（最多 1 个 surge ）。

在未来的 Kuberentes 版本中，将从 1-1 变成 25%-25%。

例如，如果您自己看下上面的 Deployment，您会发现，开始创建一个新的 Pod，然后删除一些旧的 Pod 再创建一个新的。当新的 Pod 创建出来之前不会杀掉旧的 Pod。这样能够确保可用的 Pod 数量至少有 2 个，Pod 的总数最多 4 个。

```
$ kubectl describe deployments
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 15 Mar 2016 12:01:06 -0700
Labels:                 app=nginx
Selector:               app=nginx
Replicas:               3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                            SubobjectPath   Type        Reason              Message
  --------- --------    -----   ----                            -------------   --------    ------              -------
  36s       36s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-2035384211 to 3
  23s       23s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 1
  23s       23s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 2
  23s       23s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 2
  21s       21s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 3
```

我们可以看到当我们刚开始创建这个 Deployment 的时候，创建了一个 ReplicaSet（nginx-deployment-2035384211），并直接扩容到了 3 个 replica。

当我们更新这个 Deployment 的时候，它会创建一个新的 ReplicaSet（nginx-deployment-1564180365），将它扩容到 1 个 replica，然后缩容原先的 ReplicaSet 到 2 个 replica，此时满足至少 2 个 Pod 是可用状态，同一时刻最多有 4 个 Pod 处于创建的状态。

接着继续使用相同的 rolling update 策略扩容新的 ReplicaSet 和缩容旧的 ReplicaSet。最终，将会在新的 ReplicaSet 中有 3 个可用的 replica，旧的 ReplicaSet 的 replica 数目变成 0。

Rollover（多个 rollout 并行）

每当 Deployment controller 观测到有新的 deployment 被创建时，如果没有已存在的 ReplicaSet 来创建期望个数的 Pod 的话，就会创建出一个新的 ReplicaSet 来做这件事。已存在的 ReplicaSet 控制 label 与.spec.selector 匹配但是 template 跟.spec.template 不匹配的 Pod 缩容。最终，新的 ReplicaSet 将会扩容出.spec.replicas 指定数目的 Pod，旧的 ReplicaSet 会缩容到 0。

如果您更新了一个的已存在并正在进行中的 Deployment，每次更新 Deployment 都会创建一个新的 ReplicaSet 并扩容它，同时回滚之前扩容的 ReplicaSet ——将它添加到旧的 ReplicaSet 列表中，开始缩容。

例如，假如您创建了一个有 5 个 niginx:1.7.9 replica 的 Deployment，但是当还只有 3 个 nginx:1.7.9 的 replica 创建出来的时候您就开始更新含有 5 个 nginx:1.9.1 replica 的 Deployment。在这种情况下，Deployment 会立即杀掉已创建的 3 个 nginx:1.7.9 的 Pod，并开始创建 nginx:1.9.1 的 Pod。它不会等到所有的 5 个 nginx:1.7.9 的 Pod 都创建完成后才开始改变航道。

Label selector 更新

我们通常不鼓励更新 label selector，我们建议实现规划好您的 selector。

任何情况下，只要您想要执行 label selector 的更新，请一定要谨慎并确认您已经预料到所有可能因此导致的后果。

增添 selector 需要同时在 Deployment 的 spec 中更新新的 label，否则将返回校验错误。此更改是不可覆盖的，这意味着新的 selector 不会选择使用旧 selector 创建的 ReplicaSet 和 Pod，从而导致所有旧版本的 ReplicaSet 都被丢弃，并创建新的 ReplicaSet。
更新 selector，即更改 selector key 的当前值，将导致跟增添 selector 同样的后果。
删除 selector，即删除 Deployment selector 中的已有的 key，不需要对 Pod template label 做任何更改，现有的 ReplicaSet 也不会成为孤儿，但是请注意，删除的 label 仍然

存在于现有的 Pod 和 ReplicaSet 中。

回退 Deployment

有时候您可能想回退一个 Deployment，例如，当 Deployment 不稳定时，比如一直 crash looping。

默认情况下，kubernetes 会在系统中保存前两次的 Deployment 的 rollout 历史记录，以便您可以随时回退（您可以修改 revision history limit 来更改保存的 revision 数）。

注意： 只要 Deployment 的 rollout 被触发就会创建一个 revision。也就是说当且仅当 Deployment 的 Pod template（如.spec.template）被更改，例如更新 template 中的 label 和容器镜像时，就会创建出一个新的 revision。

其他的更新，比如扩容 Deployment 不会创建 revision——因此我们可以很方便的手动或者自动扩容。这意味着当您回退到历史 revision 是，直有 Deployment 中的 Pod template 部分才会回退。

假设我们在更新 Deployment 的时候犯了一个拼写错误，将镜像的名字写成了 nginx:1.91，而正确的名字应该是 nginx:1.9.1：

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
deployment "nginx-deployment" image updated
```

Rollout 将会卡住。

```
$ kubectl rollout status deployments nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

按住 Ctrl-C 停止上面的 rollout 状态监控。

您会看到旧的 replica（nginx-deployment-1564180365 和 nginx-deployment-2035384211）和新的 replica（nginx-deployment-3066724191）数目都是 2 个。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   2         2         0       25s
nginx-deployment-2035384211   0         0         0       36s
nginx-deployment-3066724191   2         2         2       6s
```

看下创建 Pod，您会看到有两个新的 ReplicaSet 创建的 Pod 处于 ImagePullBackOff 状态，循环拉取镜像。

```
$ kubectl get pods
NAME                                          READY     STATUS
RESTARTS   AGE
```

```
nginx-deployment-1564180365-70iae  1/1  Running  0
25s
nginx-deployment-1564180365-jbqqo  1/1  Running  0
25s
nginx-deployment-3066724191-08mng  0/1  ImagePullBackOff  0
6s
nginx-deployment-3066724191-eocby  0/1  ImagePullBackOff  0
6s
```

注意，Deployment controller 会自动停止坏的 rollout，并停止扩容新的 ReplicaSet。

```
$ kubectl describe deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 15 Mar 2016 14:48:04 -0700
Labels:                 app=nginx
Selector:               app=nginx
Replicas:               2 updated | 3 total | 2 available | 2 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         nginx-deployment-1564180365 (2/2 replicas created)
NewReplicaSet:          nginx-deployment-3066724191 (2/2 replicas created)
Events:
  FirstSeen LastSeen    Count   From                    SubobjectPath   Type
Reason                  Message
  --------- --------    -----   ----                    -------------   --------
------
  1m        1m          1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled up replica set nginx-deployment-2035384211 to 3
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled up replica set nginx-deployment-1564180365 to 1
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled down replica set nginx-deployment-2035384211 to 2
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled up replica set nginx-deployment-1564180365 to 2
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled down replica set nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled up replica set nginx-deployment-1564180365 to 3
  13s       13s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled up replica set nginx-deployment-3066724191 to 1
  13s       13s         1       {deployment-controller }                Normal
ScalingReplicaSet       Scaled down replica set nginx-deployment-1564180365 to 2
  13s       13s         1       {deployment-controller }                Normal
```

ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 2
为了修复这个问题，我们需要回退到稳定的 Deployment revision。

检查 Deployment 升级的历史记录

首先，检查下 Deployment 的 revision：

```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment":
REVISION CHANGE-CAUSE
1 kubectl create -f https://kubernetes.io/docs/user-guide/nginx-deployment.yaml--record
2 kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
3 kubectl set image deployment/nginx-deployment nginx=nginx:1.91
```

因为我们创建 Deployment 的时候使用了--recored 参数可以记录命令，我们可以很方便的查看每次 revision 的变化。

查看单个 revision 的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels: app=nginx
          pod-template-hash=1159050644
  Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
  Containers:
   nginx:
    Image: nginx:1.9.1
    Port: 80/TCP
     QoS Tier:
        cpu: BestEffort
        memory: BestEffort
    Environment Variables: <none>
  No volumes.
```

回退到历史版本

现在，我们可以决定回退当前的 rollout 到之前的版本：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
```

也可以使用 --revision 参数指定某个历史版本：

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=2
deployment "nginx-deployment" rolled back
```

与 rollout 相关的命令详细文档见 kubectl rollout。

该 Deployment 现在已经回退到了先前的稳定版本。如您所见，Deployment controller 产生了一个回退到 revison 2 的 DeploymentRollback 的 event。

```
$ kubectl get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    3         3         3            3           30m

$ kubectl describe deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 15 Mar 2016 14:48:04 -0700
Labels:                 app=nginx
Selector:               app=nginx
Replicas:               3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:         <none>
NewReplicaSet:          nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                            SubobjectPath   Type        Reason              Message
  --------- --------    -----   ----                            -------------   --------    ------              -------
  30m       30m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-2035384211 to 3
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 1
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 2
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 2
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 0
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 2
  29m       29m         1       {deployment-controller }                        Normal      ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 1
  29m       29m         1       {deployment-controller }                        Normal
```

ScalingReplicaSet Scaled down replica set nginx-deployment-1564180365 to 2
2m 2m 1 {deployment-controller } Normal
ScalingReplicaSet Scaled down replica set nginx-deployment-3066724191 to 0
2m 2m 1 {deployment-controller } Normal
DeploymentRollback Rolled back deployment "nginx-deployment" to revision 2
29m 2m 2 {deployment-controller } Normal
ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

清理 Policy

您可以通过设置.spec.revisonHistoryLimit 项来指定 deployment 最多保留多少 revision 历史记录。默认的会保留所有的 revision；如果将该项设置为 0，Deployment 就不允许回退了。

Deployment 扩容

您可以使用以下命令扩容 Deployment：

$ kubectl scale deployment nginx-deployment --replicas 10
deployment "nginx-deployment" scaled
假设您的集群中启用了 horizontal pod autoscaling，您可以给 Deployment 设置一个 autoscaler，基于当前 Pod 的 CPU 利用率选择最少和最多的 Pod 数。

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

比例扩容

RollingUpdate Deployment 支持同时运行一个应用的多个版本。或者 autoscaler 扩容 RollingUpdate Deployment 的时候，正在中途的 rollout（进行中或者已经暂停的），为了降低风险，Deployment controller 将会平衡已存在的活动中的 ReplicaSet（有 Pod 的 ReplicaSet）和新加入的 replica。这被称为比例扩容。

例如, 您正在运行中含有 10 个 replica 的 Deployment。maxSurge=3, maxUnavailable=2。

$ kubectl get deploy
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
nginx-deployment 10 10 10 10 50s
您更新了一个镜像，而在集群内部无法解析。

```
$ kubectl set image deploy/nginx-deployment nginx=nginx:sometag
deployment "nginx-deployment" image updated
```

镜像更新启动了一个包含 ReplicaSet nginx-deployment-1989198191 的新的 rollout，但是它被阻塞了，因为我们上面提到的 maxUnavailable。

```
$ kubectl get rs
NAME                          DESIRED     CURRENT     READY       AGE
nginx-deployment-1989198191   5           5           0           9s
nginx-deployment-618515232    8           8           8           1m
```

然后发起了一个新的 Deployment 扩容请求。autoscaler 将 Deployment 的 repllica 数目增加到了 15 个。Deployment controller 需要判断在哪里增加这 5 个新的 replica。如果我们没有谁用比例扩容, 所有的 5 个 replica 都会加到一个新的 ReplicaSet 中。如果使用比例扩容, 新添加的 replica 将传播到所有的 ReplicaSet 中。大的部分加入 replica 数最多的 ReplicaSet 中，小的部分加入到 replica 数少的 ReplciaSet 中。0 个 replica 的 ReplicaSet 不会被扩容。

在我们上面的例子中，3 个 replica 将添加到旧的 ReplicaSet 中，2 个 replica 将添加到新的 ReplicaSet 中。rollout 进程最终会将所有的 replica 移动到新的 ReplicaSet 中，假设新的 replica 成为健康状态。

```
$ kubectl get deploy
NAME                DESIRED     CURRENT     UP-TO-DATE     AVAILABLE
AGE
nginx-deployment    15          18          7              8              7m
$ kubectl get rs
NAME                          DESIRED     CURRENT     READY       AGE
nginx-deployment-1989198191   7           7           0           7m
nginx-deployment-618515232    11          11          11          7m
```

暂停和恢复 Deployment

您可以在发出一次或多次更新前暂停一个 Deployment，然后再恢复它。这样您就能多次暂停和恢复 Deployment，在此期间进行一些修复工作，而不会发出不必要的 rollout。

例如使用刚刚创建 Deployment：

```
$ kubectl get deploy
NAME        DESIRED     CURRENT     UP-TO-DATE     AVAILABLE     AGE
nginx       3           3           3              3             1m
[mkargaki@dhcp129-211 kubernetes]$ kubectl get rs
NAME                    DESIRED     CURRENT     READY       AGE
nginx-2142116321        3           3           3           1m
```

使用以下命令暂停 Deployment：

```
$ kubectl rollout pause deployment/nginx-deployment
deployment "nginx-deployment" paused
```
然后更新 Deplyment 中的镜像：

```
$ kubectl set image deploy/nginx nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```
注意新的 rollout 启动了：

```
$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION    CHANGE-CAUSE
1     <none>

$ kubectl get rs
NAME                 DESIRED     CURRENT     READY       AGE
nginx-2142116321     3           3           3           2m
```
您可以进行任意多次更新，例如更新使用的资源：

```
$ kubectl set resources deployment nginx -c=nginx --limits=cpu=200m,memory=512Mi
deployment "nginx" resource requirements updated
```
Deployment 暂停前的初始状态将继续它的功能，而不会对 Deployment 的更新产生任何影响，只要 Deployment 是暂停的。

最后，恢复这个 Deployment，观察完成更新的 ReplicaSet 已经创建出来了：

```
$ kubectl rollout resume deploy nginx
deployment "nginx" resumed
$ KUBECTL get rs -w
NAME                 DESIRED     CURRENT     READY       AGE
nginx-2142116321     2           2           2           2m
nginx-3926361531     2           2           0           6s
nginx-3926361531     2           2           1           18s
nginx-2142116321     1           2           2           2m
nginx-2142116321     1           2           2           2m
nginx-3926361531     3           2           1           18s
nginx-3926361531     3           2           1           18s
nginx-2142116321     1           1           1           2m
nginx-3926361531     3           3           1           18s
nginx-3926361531     3           3           2           19s
nginx-2142116321     0           1           1           2m
nginx-2142116321     0           1           1           2m
nginx-2142116321     0           0           0           2m
nginx-3926361531     3           3           3           20s
^C
```

$ KUBECTL get rs

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	0	0	0	2m
nginx-3926361531	3	3	3	28s

注意： 在恢复 Deployment 之前您无法回退一个已经暂停的 Deployment。

Deployment 状态

Deployment 在生命周期中有多种状态。在创建一个新的 ReplicaSet 的时候它可以是 progressing 状态， complete 状态，或者 fail to progress 状态。

进行中的 Deployment

Kubernetes 将执行过下列任务之一的 Deployment 标记为 progressing 状态：

- Deployment 正在创建新的 ReplicaSet 过程中。
- Deployment 正在扩容一个已有的 ReplicaSet。
- Deployment 正在缩容一个已有的 ReplicaSet。
- 有新的可用的 pod 出现。

您可以使用 kubectl rollout status 命令监控 Deployment 的进度。

完成的 Deployment

Kubernetes 将包括以下特性的 Deployment 标记为 complete 状态：

- Deployment 最小可用。最小可用意味着 Deployment 的可用 replica 个数等于或者超过 Deployment 策略中的期望个数。
- 所有与该 Deployment 相关的 replica 都被更新到了您指定版本，也就说更新完成。
- 该 Deployment 中没有旧的 Pod 存在。

您可以用 kubectl rollout status 命令查看 Deployment 是否完成。如果 rollout 成功完成，kubectl rollout status 将返回一个 0 值的 Exit Code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx" successfully rolled out
$ echo $?
0
```

失败的 Deployment

您的 Deployment 在尝试部署新的 ReplicaSet 的时候可能卡住，用于也不会完成。这可能是因为以下几个因素引起的：

- 无效的引用
- 不可读的 probe failure
- 镜像拉取错误
- 权限不够
- 范围限制
- 程序运行时配置错误

探测这种情况的一种方式是，在您的 Deployment spec 中指定 spec.progressDeadlineSeconds。spec.progressDeadlineSeconds 表示 Deployment controller 等待多少秒才能确定（通过 Deployment status）Deployment 进程是卡住的。

下面的 kubectl 命令设置 progressDeadlineSeconds 使 controller 在 Deployment 在进度卡住 10 分钟后报告：

```
$ kubectl patch deployment/nginx-deployment -p
'{"spec":{"progressDeadlineSeconds":600}}'
"nginx-deployment" patched
```

当超过截止时间后，Deployment controller 会在 Deployment 的 status.conditions 中增加一条 DeploymentCondition，它包括如下属性：

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

浏览 Kubernetes API conventions 查看关于 status conditions 的更多信息。

注意： kubernetes 除了报告 Reason=ProgressDeadlineExceeded 状态信息外不会对卡住的 Deployment 做任何操作。更高层次的协调器可以利用它并采取相应行动，例如，回滚 Deployment 到之前的版本。

注意： 如果您暂停了一个 Deployment，在暂停的这段时间内 kubernetnes 不会检查您指定的 deadline。您可以在 Deployment 的 rollout 途中安全的暂停它，然后再恢复它，这不会触发超过 deadline 的状态。

您可能在使用 Deployment 的时候遇到一些短暂的错误，这些可能是由于您设置了太短的 timeout，也有可能是因为各种其他错误导致的短暂错误。例如，假设您使用了无效的引用。当您 Describe Deployment 的时候可能会注意到如下信息：

```
$ kubectl describe deployment nginx-deployment
<...>
Conditions:
  Type                 Status    Reason
  ----                 ------    ------
  Available            True      MinimumReplicasAvailable
  Progressing          True      ReplicaSetUpdated
  ReplicaFailure       True      FailedCreate
<...>
```

执行 kubectl get deployment nginx-deployment -o yaml，Deployement 的状态可能看起来像这个样子：

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progressing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden:
exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

最终，一旦超过 Deployment 进程的 deadline，kuberentes 会更新状态和导致 Progressing 状态的原因：

```
Conditions:
  Type                 Status   Reason
  ----                 ------   ------
  Available            True     MinimumReplicasAvailable
  Progressing          False    ProgressDeadlineExceeded
  ReplicaFailure       True     FailedCreate
```

您可以通过缩容 Deployment 的方式解决配额不足的问题，或者增加您的 namespace 的配额。如果您满足了配额条件后，Deployment controller 就会完成您的 Deployment rollout，您将看到 Deployment 的状态更新为成功状态（Status=True 并且 Reason=NewReplicaSetAvailable）。

```
Conditions:
  Type                 Status   Reason
  ----                 ------   ------
  Available            True     MinimumReplicasAvailable
  Progressing          True     NewReplicaSetAvailable
```

Type=Available、 Status=True 以为这您的 Deployment 有最小可用性。 最小可用性是在 Deployment 策略中指定的参数。Type=Progressing 、 Status=True 意味着您的 Deployment 或者在部署过程中，或者已经成功部署，达到了期望的最少的可用 replica 数量（查看特定状态的 Reason——在我们的例子中 Reason=NewReplicaSetAvailable 意味着 Deployment 已经完成）。

您可以使用 kubectl rollout status 命令查看 Deployment 进程是否失败。当 Deployment 过程超过了 deadline，kubectl rollout status 将返回非 0 的 exit code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
$ echo $?
1
```

操作失败的 Deployment

所有对完成的 Deployment 的操作都适用于失败的 Deployment。您可以对它扩/缩容，回退到历史版本，您甚至可以多次暂停它来应用 Deployment pod template。

清理 Policy

您可以设置 Deployment 中的 .spec.revisionHistoryLimit 项来指定保留多少旧的 ReplicaSet。 余下的将在后台被当作垃圾收集。默认的，所有的 revision 历史就都会被保留。在未来的版本中，将会更改为 2。

注意：将该值设置为 0，将导致所有的 Deployment 历史记录都会被清除，该 Deployment 就无法再回退了。

用例

金丝雀 Deployment

如果您想要使用 Deployment 对部分用户或服务器发布 release，您可以创建多个 Deployment，每个 Deployment 对应一个 release，参照 managing resources 中对金丝雀模式的描述。

编写 Deployment Spec

在所有的 Kubernetes 配置中，Deployment 也需要 apiVersion，kind 和 metadata 这些配置项。配置文件的通用使用说明查看 部署应用，配置容器，和 使用 kubectl 管理资源 文档。

Deployment 也需要 .spec section.

Pod Template

.spec.template 是 .spec 中唯一要求的字段。

.spec.template 是 pod template. 它跟 Pod 有一模一样的 schema，除了它是嵌套的并且不需要 apiVersion 和 kind 字段。

另外为了划分 Pod 的范围，Deployment 中的 pod template 必须指定适当的 label（不要跟其他 controller 重复了，参考 selector）和适当的重启策略。

.spec.template.spec.restartPolicy 可以设置为 Always，如果不指定的话这就是默认配置。

Replicas

.spec.replicas 是可以选字段，指定期望的 pod 数量，默认是 1。

Selector

.spec.selector 是可选字段，用来指定 label selector ，圈定 Deployment 管理的 pod 范围。

如果被指定， .spec.selector 必须匹配 .spec.template.metadata.labels，否则它将被 API 拒绝。如果 .spec.selector 没有被指定， .spec.selector.matchLabels 默认是 .spec.template.metadata.labels。

在 Pod 的 template 跟.spec.template 不同或者数量超过了.spec.replicas 规定的数量的情况下，Deployment 会杀掉 label 跟 selector 不同的 Pod。

注意：您不应该再创建其他 label 跟这个 selector 匹配的 pod，或者通过其他 Deployment，或者通过其他 Controller，例如 ReplicaSet 和 ReplicationController。否则该 Deployment 会被把它们当成都是自己创建的。Kubernetes 不会阻止您这么做。

如果您有多个 controller 使用了重复的 selector，controller 们就会互相打架并导致不正确的行为。

策略

.spec.strategy 指定新的 Pod 替换旧的 Pod 的策略。 .spec.strategy.type 可以是 "Recreate"或者是 "RollingUpdate"。"RollingUpdate"是默认值。

Recreate Deployment

.spec.strategy.type==Recreate 时，在创建出新的 Pod 之前会先杀掉所有已存在的 Pod。

Rolling Update Deployment

.spec.strategy.type==RollingUpdate 时， Deployment 使用 rolling update 的方式更新 Pod 。您可以指定 maxUnavailable 和 maxSurge 来控制 rolling update 进程。

MAX UNAVAILABLE

.spec.strategy.rollingUpdate.maxUnavailable 是可选配置项，用来指定在升级过程中不可用 Pod 的最大数量。该值可以是一个绝对值（例如 5），也可以是期望 Pod 数量的百分比（例如 10%）。通过计算百分比的绝对值向下取整。如果.spec.strategy.rollingUpdate.maxSurge 为 0 时，这个值不可以为 0。默认值是 1。

例如, 该值设置成 30%, 启动 rolling update 后旧的 ReplicatSet 将会立即缩容到期望的 Pod 数量的 70%。新的 Pod ready 后，随着新的 ReplicaSet 的扩容，旧的 ReplicaSet 会进一步缩容，确保在升级的所有时刻可以用的 Pod 数量至少是期望 Pod 数量的 70%。

MAX SURGE

.spec.strategy.rollingUpdate.maxSurge 是可选配置项，用来指定可以超过期望的 Pod 数量的最大个数。该值可以是一个绝对值（例如 5）或者是期望的 Pod 数量的百分比（例如 10%）。当 MaxUnavailable 为 0 时该值不可以为 0。通过百分比计算的绝对值向上取整。默认值是 1。

例如，该值设置成 30%，启动 rolling update 后新的 ReplicatSet 将会立即扩容，新老 Pod 的总数不能超过期望的 Pod 数量的 130%。旧的 Pod 被杀掉后，新的 ReplicaSet 将继续扩容，旧的 ReplicaSet 会进一步缩容，确保在升级的所有时刻所有的 Pod 数量和不会超过期望 Pod 数量的 130%。

Progress Deadline Seconds

.spec.progressDeadlineSeconds 是可选配置项，用来指定在系统报告 Deployment 的 failed progressing——表现为 resource 的状态中 type=Progressing、Status=False、Reason=ProgressDeadlineExceeded 前可以等待的 Deployment 进行的秒数。Deployment controller 会继续重试该 Deployment。未来，在实现了自动回滚后， deployment controller 在观察到这种状态时就会自动回滚。

如果设置该参数，该值必须大于 .spec.minReadySeconds。

Min Ready Seconds

.spec.minReadySeconds 是一个可选配置项，用来指定没有任何容器 crash 的 Pod 并被认为是可用状态的最小秒数。默认是 0（Pod 在 ready 后就会被认为是可用状态）。进一步了解什么什么后 Pod 会被认为是 ready 状态，参阅 Container Probes。

Rollback To

.spec.rollbackTo 是一个可以选配置项，用来配置 Deployment 回退的配置。设置该参数将触发回退操作，每次回退完成后，该值就会被清除。

Revision

.spec.rollbackTo.revision 是一个可选配置项，用来指定回退到的 revision。默认是 0，意味着回退到历史中最老的 revision。

Revision History Limit

Deployment revision history 存储在它控制的 ReplicaSets 中。

.spec.revisionHistoryLimit 是一个可选配置项，用来指定可以保留的旧的 ReplicaSet 数量。该理想值取决于心 Deployment 的频率和稳定性。如果该值没有设置的话，默认所有旧的 Replicaset 或会被保留，将资源存储在 etcd 中，是用 kubectl get rs 查看输出。每个 Deployment 的该配置都保存在 ReplicaSet 中，然而，一旦您删除的旧的 RepelicaSet，您的 Deployment 就无法再回退到那个 revison 了。

如果您将该值设置为 0，所有具有 0 个 replica 的 ReplicaSet 都会被删除。在这种情况下，新的 Deployment rollout 无法撤销，因为 revision history 都被清理掉了。

Paused

.spec.paused 是可以可选配置项，boolean 值。用来指定暂停和恢复 Deployment。Paused 和没有 paused 的 Deployment 之间的唯一区别就是，所有对 paused deployment 中的 PodTemplateSpec 的修改都不会触发新的 rollout。Deployment 被创建之后默认是非 paused。

Deployment 的替代选择

kubectl rolling update

Kubectl rolling update 虽然使用类似的方式更新 Pod 和 ReplicationController。但是我们推荐使用 Deployment，因为它是声明式的，客户端侧，具有附加特性，例如即使滚动升级结束后也可以回滚到任何历史版本。

参考：
https://rootsongjc.gitbooks.io/kubernetes-handbook/content/concepts/deployment.html

Replica Sets

ReplicaSet（RS）是 Replication Controller（RC）的升级版本。ReplicaSet 和 Replication Controller 之间的唯一区别是对选择器的支持。ReplicaSet 支持 labels user guide 中描述的 set-based 选择器要求， 而 Replication Controller 仅支持 equality-based 的选择器要求。

如何使用 ReplicaSet

大多数 kubectl 支持 Replication Controller 命令的也支持 ReplicaSets。rolling-update 命令除外，如果要使用 rolling-update，请使用 Deployments 来实现。

虽然 ReplicaSets 可以独立使用，但它主要被 Deployments 用作 pod 机制的创建、删除和更新。当使用 Deployment 时，你不必担心创建 pod 的 ReplicaSets，因为可以通过 Deployment 实现管理 ReplicaSets。

何时使用 ReplicaSet

ReplicaSet 能确保运行指定数量的 pod。然而，Deployment 是一个更高层次的概念，它能管理 ReplicaSets，并提供对 pod 的更新等功能。因此，我们建议你使用 Deployment 来管理 ReplicaSets，除非你需要自定义更新编排。

这意味着你可能永远不需要操作 ReplicaSet 对象，而是使用 Deployment 替代管理 。

示例

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
    # app: guestbook
    # tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
```

```
  # selector can be applied automatically
  # from the labels in the pod template if not set,
  # but we are specifying the selector here to
  # demonstrate its usage.
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

将此配置保存到（frontend.yaml）并提交到 Kubernetes 集群时，将创建定义的 ReplicaSet 及其管理的 pod。

```
$ kubectl create -f frontend.yaml
replicaset "frontend" created
$ kubectl describe rs/frontend
Name:                frontend
Namespace:           default
Image(s):            gcr.io/google_samples/gb-frontend:v3
Selector:            tier=frontend,tier in (frontend)
Labels:              app=guestbook,tier=frontend
```

```
Replicas:         3 current / 3 desired
Pods Status:      3 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen       LastSeen       Count       From                        SubobjectPath       Type
Reason                  Message
  ---------       --------       -----       ----                        -------------       --------
------                  -------
  1m                1m                1           {replicaset-controller }                    Normal
SuccessfulCreate    Created pod: frontend-qhloh
  1m                1m                1           {replicaset-controller }                    Normal
SuccessfulCreate    Created pod: frontend-dnjpy
  1m                1m                1           {replicaset-controller }                    Normal
SuccessfulCreate    Created pod: frontend-9si5l
$ kubectl get pods
NAME                READY       STATUS      RESTARTS     AGE
frontend-9si5l      1/1         Running     0            1m
frontend-dnjpy      1/1         Running     0            1m
frontend-qhloh      1/1         Running     0            1m
```

ReplicaSet as an Horizontal Pod Autoscaler target

ReplicaSet 也可以作为 Horizontal Pod Autoscalers (HPA)的目标 。也就是说，一个 ReplicaSet 可以由一个 HPA 来自动伸缩。以下是针对我们在上一个示例中创建的 ReplicaSet 的 HPA 示例。

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
kubectl create -f hpa-rs.yaml
```

Nodes

Node 是什么?

Node 是 Kubernetes 中的工作节点，最开始被称为 minion。一个 Node 可以是 VM 或物理机。每个 Node（节点）具有运行 pod 的一些必要服务，并由 Master 组件进行管理，Node 节点上的服务包括 Docker、kubelet 和 kube-proxy。有关更多详细信息，请参考架构设计文档中的“Kubernetes Node”部分。

Node Status

节点的状态信息包含：

- Addresses
- Phase (已弃用)
- Condition
- Capacity
- Info

下面详细描述每个部分。

Addresses

这些字段的使用取决于云提供商或裸机配置。

HostName：可以通过 kubelet 中 --hostname-override 参数覆盖。
ExternalIP：可以被集群外部路由到的 IP。
InternalIP：只能在集群内进行路由的节点的 IP 地址。

Phase

不推荐使用，已弃用。

Condition

conditions 字段描述所有 Running 节点的状态。

Node Condition	Description
OutOfDisk	True：如果节点上没有足够的可用空间来添加新的 pod；否则为：

	False
Ready	True：如果节点是健康的并准备好接收 pod；False：如果节点不健康并且不接受 pod；Unknown：如果节点控制器在过去 40 秒内没有收到 node 的状态报告。
MemoryPressure	True：如果节点存储器上内存过低; 否则为：False。
DiskPressure	True：如果磁盘容量存在压力 - 也就是说磁盘容量低；否则为：False。

node condition 被表示为一个 JSON 对象。例如，下面的响应描述了一个健康的节点。

```
"conditions": [
  {
    "kind": "Ready",
    "status": "True"
  }
]
```

如果 Ready condition 的 Status 是“Unknown” 或 “False”，比“pod-eviction-timeout”的时间长，则传递给“ kube-controller-manager”的参数，该节点上的所有 Pod 都将被节点控制器删除。默认的 eviction timeout 时间为 5 分钟。在某些情况下，当节点无法访问时，apiserver 将无法与 kubelet 通信, 删除 Pod 的需求不会传递到 kubelet, 直到重新与 apiserver 建立通信，这种情况下，计划删除的 Pod 会继续在划分的节点上运行。

在 Kubernetes 1.5 之前的版本中，节点控制器将强制从 apiserver 中删除这些不可达（上述情况）的 pod。但是，在 1.5 及更高版本中，节点控制器在确认它们已经停止在集群中运行之前, 不会强制删除 Pod。可以看到这些可能在不可达节点上运行的 pod 处于"Terminating"或 “Unknown” 。如果节点永久退出集群，Kubernetes 是无法从底层基础架构辨别出来，则集群管理员需要手动删除节点对象，从 Kubernetes 删除节点对象会导致运行在上面的所有 Pod 对象从 apiserver 中删除，最终将会释放 names。

Capacity

描述节点上可用的资源：CPU、内存和可以调度到节点上的最大 pod 数。

Info

关于节点的一些基础信息，如内核版本、Kubernetes 版本（kubelet 和 kube-proxy 版本）、Docker 版本（如果有使用）、OS 名称等。信息由 Kubelet 从节点收集。

Management

与 pods 和 services 不同，节点不是由 Kubernetes 系统创建，它是由 Google Compute Engine 等云提供商在外部创建的，或使用物理和虚拟机。这意味着当 Kubernetes 创建一个节点时，它只是创建一个代表节点的对象，创建后，Kubernetes 将检查节点是否有效。例如，如果使用以下内容创建一个节点：

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes 将在内部创建一个节点对象，并通过基于 metadata.name 字段的健康检查来验证节点，如果节点有效，即所有必需的服务会同步运行，则才能在上面运行 pod。请注意，Kubernetes 将保留无效节点的对象（除非客户端有明确删除它）并且它将继续检查它是否变为有效。

目前，有三个组件与 Kubernetes 节点接口进行交互：节点控制器（node controller）、kubelet 和 kubectl。

Node Controller

节点控制器（Node Controller）是管理节点的 Kubernetes master 组件。

节点控制器在节点的生命周期中具有多个角色。第一个是在注册时将 CIDR 块分配给节点。

第二个是使节点控制器的内部列表与云提供商的可用机器列表保持最新。当在云环境中运行时，每当节点不健康时，节点控制器将询问云提供程序是否该节点的 VM 仍然可用，如果不可用，节点控制器会从其节点列表中删除该节点。

第三是监测节点的健康状况。当节点变为不可访问时，节点控制器负责将 NodeStatus 的 NodeReady 条件更新为 ConditionUnknown，随后从节点中卸载所有 pod ，如果节点继续无法访问，（默认超时时间为 40 --node-monitor-period 秒，开始报告 ConditionUnknown，之后为 5m 开始卸载）。节点控制器按每秒来检查每个节点的状态。

在 Kubernetes 1.4 中，我们更新了节点控制器的逻辑，以更好地处理大量节点到达主节点

的一些问题（例如，主节某些网络问题）。从 1.4 开始，节点控制器将在决定关于 pod 卸载的过程中会查看集群中所有节点的状态。

在大多数情况下，节点控制器将逐出速率限制为 --node-eviction-rate（默认为 0.1）/秒，这意味着它不会每 10 秒从多于 1 个节点驱逐 Pod。

当给定可用性的区域中的节点变得不健康时，节点逐出行为发生变化，节点控制器同时检查区域中节点的不健康百分比（NodeReady 条件为 ConditionUnknown 或 ConditionFalse）。如果不健康节点的比例为 --unhealthy-zone-threshold（默认为 0.55），那么驱逐速度就会降低：如果集群很小（即小于或等于--large-cluster-size-threshold 节点 - 默认值为 50），则停止驱逐，否则， --secondary-node-eviction-rate（默认为 0.01）每秒。这些策略在可用性区域内实现的原因是，一个可用性区域可能会从主分区中被分区，而其他可用区域则保持连接。如果集群没有跨多个云提供商可用性区域，那么只有一个可用区域(整个集群)。

在可用区域之间传播节点的一个主要原因是，当整个区域停止时，工作负载可以转移到健康区域。因此，如果区域中的所有节点都不健康，则节点控制器以正常速率逐出 --node-eviction-rate。如所有的区域都是完全不健康的（即群集中没有健康的节点），在这种情况下，节点控制器会假设主连接有一些问题，并停止所有驱逐，直到某些连接恢复。

从 Kubernetes 1.6 开始，节点控制器还负责驱逐在节点上运行的 NoExecutepod。

Self-Registration of Nodes

当 kubelet flag --register-node 为 true（默认值）时，kubelet 将向 API 服务器注册自身。这是大多数发行版使用的首选模式。

对于 self-registration，kubelet 从以下选项开始：

- --api-servers - Location of the apiservers.
- --kubeconfig - Path to credentials to authenticate itself to the apiserver.
- --cloud-provider - How to talk to a cloud provider to read metadata about itself.
- --register-node - Automatically register with the API server.
- --register-with-taints - Register the node with the given list of taints (comma separated <key>=<value>:<effect>). No-op if register-node is false.
- --node-ip IP address of the node.
- --node-labels - Labels to add when registering the node in the cluster.
- --node-status-update-frequency - Specifies how often kubelet posts node status to master.

手动管理节点

集群管理员可以创建和修改节点对象。

如果管理员希望手动创建节点对象，请设置 kubelet flag　--register-node=false。

管理员可以修改节点资源 (不管--register-node 设置如何) , 修改包括在节点上设置的 labels 标签，并将其标记为不可调度的。

节点上的标签可以与 pod 上的节点选择器一起使用，以控制调度，例如将一个 pod 限制为只能在节点的子集上运行。

将节点标记为不可调度将防止新的 pod 被调度到该节点，但不会影响节点上的任何现有的 pod，这在节点重新启动之前是有用的。例如，要标记节点不可调度，请运行以下命令：

kubectl cordon $NODENAME

注意，由 daemonSet 控制器创建的 pod 可以绕过 Kubernetes 调度程序，并且不遵循节点上无法调度的属性。

Node 容量

节点的容量(cpu 数量和内存数量)是节点对象的一部分。通常，节点在创建节点对象时注册并通知其容量。如果是手动管理节点，则需要在添加节点时设置节点容量。

Kubernetes 调度程序可确保节点上的所有 pod 都有足够的资源。它会检查节点上容器的请求的总和不大于节点容量。

如果要明确保留非 pod 过程的资源，可以创建一个占位符 pod。使用以下模板：

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: sleep-forever
    image: gcr.io/google_containers/pause:0.8.0
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
```

将 cpu 和内存值设置为你想要保留的资源量，将文件放置在 manifest 目录中(--config=DIR flag of kubelet)。在要预留资源的每个 kubelet 上执行此操作。

API 对象

Node 是 Kubernetes REST API 中的最高级别资源。有关 API 对象的更多详细信息, 请参考: Node API 对象。

Service

Kubernetes Pod 是有生命周期的，它们可以被创建，也可以被销毁，然而一旦被销毁生命就永远结束。 通过 ReplicationController 能够动态地创建和销毁 Pod（例如，需要进行扩缩容，或者执行 滚动升级）。 每个 Pod 都会获取它自己的 IP 地址，即使这些 IP 地址不总是稳定可依赖的。 这会导致一个问题：在 Kubernetes 集群中，如果一组 Pod（称为 backend）为其它 Pod （称为 frontend）提供服务，那么那些 frontend 该如何发现，并连接到这组 Pod 中的哪些 backend 呢?

关于 Service

Kubernetes Service 定义了这样一种抽象：一个 Pod 的逻辑分组，一种可以访问它们的策略 —— 通常称为微服务。 这一组 Pod 能够被 Service 访问到，通常是通过 Label Selector（查看下面了解，为什么可能需要没有 selector 的 Service）实现的。

举个例子，考虑一个图片处理 backend，它运行了 3 个副本。这些副本是可互换的 —— frontend 不需要关心它们调用了哪个 backend 副本。 然而组成这一组 backend 程序的 Pod 实际上可能会发生变化，frontend 客户端不应该也没必要知道，而且也不需要跟踪这一组 backend 的状态。 Service 定义的抽象能够解耦这种关联。

对 Kubernetes 集群中的应用，Kubernetes 提供了简单的 Endpoints API，只要 Service 中的一组 Pod 发生变更，应用程序就会被更新。 对非 Kubernetes 集群中的应用，Kubernetes 提供了基于 VIP 的网桥的方式访问 Service，再由 Service 重定向到 backend Pod。

定义 Service

一个 Service 在 Kubernetes 中是一个 REST 对象，和 Pod 类似。 像所有的 REST 对象一样， Service 定义可以基于 POST 方式，请求 apiserver 创建新的实例。 例如，假定有一组 Pod，它们对外暴露了 9376 端口，同时还被打上 "app=MyApp" 标签。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
```

```
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

上述配置将创建一个名称为 “my-service” 的 Service 对象, 它会将请求代理到使用 TCP 端口 9376，并且具有标签 "app=MyApp" 的 Pod 上。 这个 Service 将被指派一个 IP 地址 (通常称为 “Cluster IP”) , 它会被服务的代理使用 (见下面) 。该 Service 的 selector 将会持续评估，处理结果将被 POST 到一个名称为 “my-service” 的 Endpoints 对象上。

需要注意的是， Service 能够将一个接收端口映射到任意的 targetPort。 默认情况下，targetPort 将被设置为与 port 字段相同的值。 可能更有趣的是，targetPort 可以是一个字符串，引用了 backend Pod 的一个端口的名称。 但是，实际指派给该端口名称的端口号，在每个 backend Pod 中可能并不相同。 对于部署和设计 Service ，这种方式会提供更大的灵活性。 例如，可以在 backend 软件下一个版本中，修改 Pod 暴露的端口，并不会中断客户端的调用。

Kubernetes Service 能够支持 TCP 和 UDP 协议，默认 TCP 协议。

没有 selector 的 Service

Servcie 抽象了该如何访问 Kubernetes Pod，但也能够抽象其它类型的 backend，例如：

- 希望在生产环境中使用外部的数据库集群，但测试环境使用自己的数据库。
- 希望服务指向另一个 Namespace 中或其它集群中的服务。
- 正在将工作负载转移到 Kubernetes 集群，和运行在 Kubernetes 集群之外的 backend。

在任何这些场景中，都能够定义没有 selector 的 Service ：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

由于这个 Service 没有 selector，就不会创建相关的 Endpoints 对象。可以手动将 Service 映射到指定的 Endpoints：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

注意：Endpoint IP 地址不能是 loopback（127.0.0.0/8）、 link-local（169.254.0.0/16）、或者 link-local 多播（224.0.0.0/24）。

访问没有 selector 的 Service，与有 selector 的 Service 的原理相同。请求将被路由到用户定义的 Endpoint（该示例中为 1.2.3.4:9376）。

ExternalName Service 是 Service 的特例，它没有 selector，也没有定义任何的端口和 Endpoint。 相反地，对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式来提供服务。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

当查询主机 my-service.prod.svc.CLUSTER 时，集群的 DNS 服务将返回一个值为 my.database.example.com 的 CNAME 记录。 访问这个服务的工作方式与其它的相同，唯一不同的是重定向发生在 DNS 层，而且不会进行代理或转发。 如果后续决定要将数据库迁移到 Kubernetes 集群中，可以启动对应的 Pod，增加合适的 Selector 或 Endpoint，修改 Service 的 type。

VIP 和 Service 代理

在 Kubernetes 集群中，每个 Node 运行一个 kube-proxy 进程。kube-proxy 负责为 Service 实现了一种 VIP（虚拟 IP）的形式，而不是 ExternalName 的形式。 在

Kubernetes v1.0 版本，代理完全在 userspace。在 Kubernetes v1.1 版本，新增了 iptables 代理，但并不是默认的运行模式。从 Kubernetes v1.2 起，默认就是 iptables 代理。

在 Kubernetes v1.0 版本，Service 是 “4 层”（TCP/UDP over IP）概念。在 Kubernetes v1.1 版本，新增了 Ingress API（beta 版），用来表示 “7 层”（HTTP）服务。

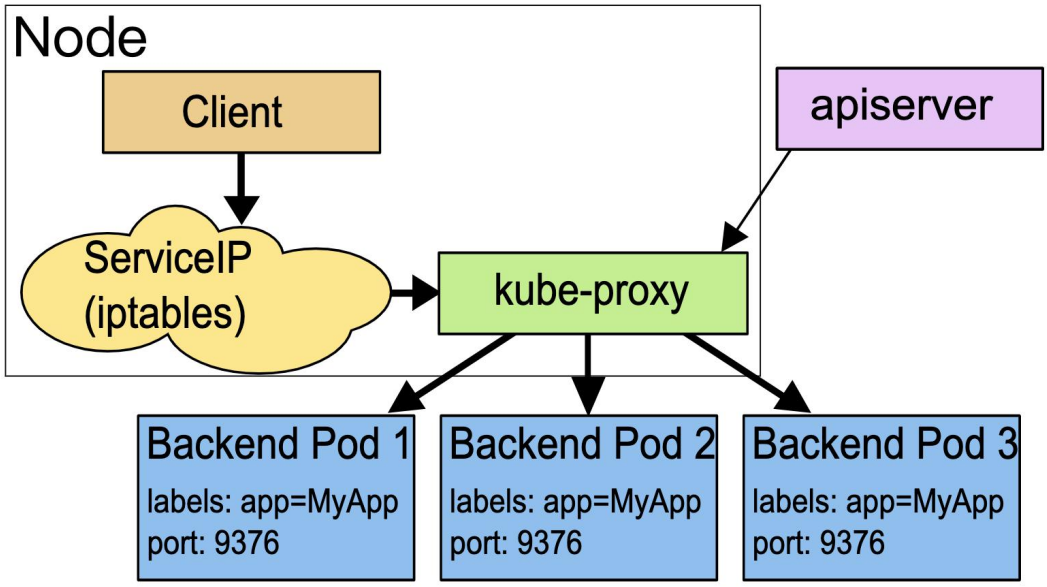
userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。对每个 Service，它会在本地 Node 上打开一个端口（随机选择）。任何连接到“代理端口”的请求，都会被代理到 Service 的 backend Pods 中的某个上面（如 Endpoints 所报告的一样）。使用哪个 backend Pod，是基于 Service 的 SessionAffinity 来确定的。最后，它安装 iptables 规则，捕获到达该 Service 的 clusterIP（是虚拟 IP）和 Port 的请求，并重定向到代理端口，代理端口再代理请求到 backend Pod。

网络返回的结果是，任何到达 Service 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、Service、或 Pod 的任何信息。

默认的策略是，通过 round-robin 算法来选择 backend Pod。实现基于客户端 IP 的会话亲和性，可以通过设置 service.spec.sessionAffinity 的值为 "ClientIP"（默认值为 "None"）。

userspace 代理模式下 Service 概览图



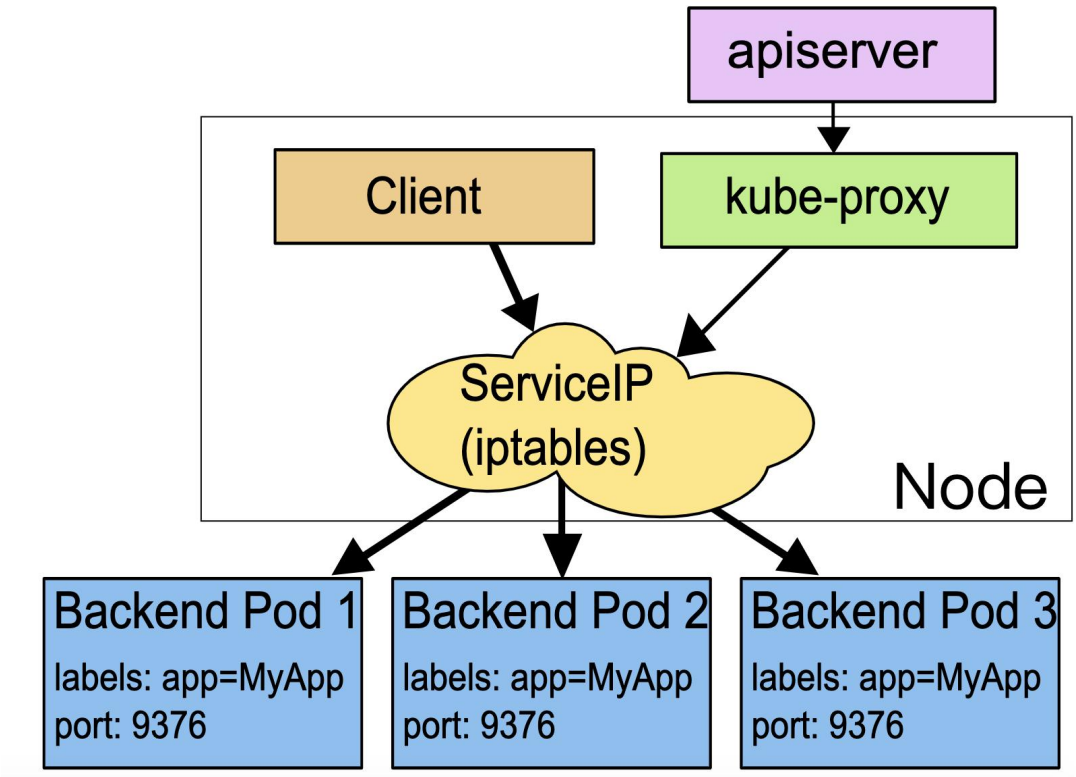
iptables 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会安装 iptables 规则，从而捕获到达该 Service 的 clusterIP（虚拟 IP）和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某个上面。 对于每个 Endpoints 对象，它也会安装 iptables 规则，这个规则会选择一个 backend Pod。

默认的策略是，随机选择一个 backend。 实现基于客户端 IP 的会话亲和性，可以将 service.spec.sessionAffinity 的值设置为 "ClientIP" （默认值为 "None"）。

和 userspace 代理类似，网络返回的结果是，任何到达 Service 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、Service、或 Pod 的任何信息。 这应该比 userspace 代理更快、更可靠。然而，不像 userspace 代理，如果初始选择的 Pod 没有响应，iptables 代理能够自动地重试另一个 Pod，所以它需要依赖 readiness probes。

iptables 代理模式下 Service 概览图



多端口 Service

很多 Service 需要暴露多个端口。对于这种情况，Kubernetes 支持在 Service 对象中定义多个端口。 当使用多个端口时，必须给出所有的端口的名称，这样 Endpoint 就不会产生歧义，例如：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
    selector:
      app: MyApp
    ports:
      - name: http
        protocol: TCP
        port: 80
        targetPort: 9376
      - name: https
        protocol: TCP
        port: 443
        targetPort: 9377
```

选择自己的 IP 地址

在 Service 创建的请求中，可以通过设置 spec.clusterIP 字段来指定自己的集群 IP 地址。 比如，希望替换一个已经已存在的 DNS 条目，或者遗留系统已经配置了一个固定的 IP 且很难重新配置。 用户选择的 IP 地址必须合法，并且这个 IP 地址在 service-cluster-ip-range CIDR 范围内，这对 API Server 来说是通过一个标识来指定的。 如果 IP 地址不合法，API Server 会返回 HTTP 状态码 422，表示值不合法。

为何不使用 round-robin DNS?

一个不时出现的问题是，为什么我们都使用 VIP 的方式，而不使用标准的 round-robin DNS，有如下几个原因：

- 长久以来，DNS 库都没能认真对待 DNS TTL、缓存域名查询结果
- 很多应用只查询一次 DNS 并缓存了结果

就算应用和库能够正确查询解析, 每个客户端反复重解析造成的负载也是非常难以管理的

我们尽力阻止用户做那些对他们没有好处的事情，如果很多人都来问这个问题，我们可能会选择实现它。

服务发现

Kubernetes 支持 2 种基本的服务发现模式 —— 环境变量和 DNS。

环境变量

当 Pod 运行在 Node 上，kubelet 会为每个活跃的 Service 添加一组环境变量。 它同时支持 Docker links 兼容 变量（查看 makeLinkVariables）、简单的 {SVCNAME}_SERVICE_HOST 和 {SVCNAME}_SERVICE_PORT 变量，这里 Service 的名称需大写，横线被转换成下划线。

举个例子，一个名称为 "redis-master" 的 Service 暴露了 TCP 端口 6379，同时给它分配了 Cluster IP 地址 10.0.0.11，这个 Service 生成了如下环境变量：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

这意味着需要有顺序的要求 —— Pod 想要访问的任何 Service 必须在 Pod 自己之前被创建，否则这些环境变量就不会被赋值。DNS 并没有这个限制。

DNS

一个可选 (尽管强烈推荐) 集群插件 是 DNS 服务器。 DNS 服务器监视着创建新 Service 的 Kubernetes API，从而为每一个 Service 创建一组 DNS 记录。 如果整个集群的 DNS 一直被启用，那么所有的 Pod 应该能够自动对 Service 进行名称解析。

例如，有一个名称为 "my-service" 的 Service，它在 Kubernetes 集群中名为 "my-ns" 的 Namespace 中，为 "my-service.my-ns" 创建了一条 DNS 记录。 在名称为 "my-ns" 的 Namespace 中的 Pod 应该能够简单地通过名称查询找到 "my-service"。 在另一个 Namespace 中的 Pod 必须限定名称为 "my-service.my-ns"。 这些名称查询的结果是 Cluster IP。

Kubernetes 也支持对端口名称的 DNS SRV（Service）记录。 如果名称为 "my-service.my-ns" 的 Service 有一个名为 "http" 的 TCP 端口，可以对 "_http._tcp.my-service.my-ns" 执行 DNS SRV 查询，得到 "http" 的端口号。

Kubernetes DNS 服务器是唯一的一种能够访问 ExternalName 类型的 Service 的方式。更多信息可以查看 DNS Pod 和 Service。

Headless Service

有时不需要或不想要负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建 Headless Service。

这个选项允许开发人员自由寻找他们自己的方式，从而降低与 Kubernetes 系统的耦合性。应用仍然可以使用一种自注册的模式和适配器，对其它需要发现机制的系统能够很容易地基于这个 API 来构建。

对这类 Service 并不会分配 Cluster IP，kube-proxy 不会处理它们，而且平台也不会为它们进行负载均衡和路由。 DNS 如何实现自动配置，依赖于 Service 是否定义了 selector。

配置 Selector

对定义了 selector 的 Headless Service，Endpoint 控制器在 API 中创建了 Endpoints 记录，并且修改 DNS 配置返回 A 记录（地址），通过这个地址直接到达 Service 的后端 Pod 上。

不配置 Selector

对没有定义 selector 的 Headless Service，Endpoint 控制器不会创建 Endpoints 记录。然而 DNS 系统会查找和配置，无论是：

- ExternalName 类型 Service 的 CNAME 记录
  记录：与 Service 共享一个名称的任何 Endpoints，以及所有其它类型

发布服务 —— 服务类型

对一些应用（如 Frontend）的某些部分，可能希望通过外部（Kubernetes 集群外部）IP 地址暴露 Service。

Kubernetes ServiceTypes 允许指定一个需要的类型的 Service，默认是 ClusterIP 类型。

Type 的取值以及行为如下：

- ClusterIP：通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。
- NodePort：通过每个 Node 上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求 <NodeIP>:<NodePort>，可以从集群的外部访问一个 NodePort 服务。
- LoadBalancer：使用云提供商的负载局衡器，可以向外部暴露服务。外部的负载均衡器可以路由到 NodePort 服务和 ClusterIP 服务。

- ExternalName：通过返回 CNAME 和它的值，可以将服务映射到 externalName 字段的内容（例如， foo.bar.example.com）。 没有任何类型代理被创建，这只有 Kubernetes 1.7 或更高版本的 kube-dns 才支持。

NodePort 类型

如果设置 type 的值为 "NodePort"，Kubernetes master 将从给定的配置范围内（默认：30000-32767）分配端口，每个 Node 将从该端口（每个 Node 上的同一端口）代理到 Service。该端口将通过 Service 的 spec.ports[*].nodePort 字段被指定。

如果需要指定的端口号, 可以配置 nodePort 的值, 系统将分配这个端口, 否则调用 API 将会失败（比如，需要关心端口冲突的可能性）。

这可以让开发人员自由地安装他们自己的负载均衡器，并配置 Kubernetes 不能完全支持的环境参数，或者直接暴露一个或多个 Node 的 IP 地址。

需要注意的是，Service 将能够通过 <NodeIP>:spec.ports[*].nodePort 和 spec.clusterIp:spec.ports[*].port 而对外可见。

LoadBalancer 类型

使用支持外部负载均衡器的云提供商的服务，设置 type 的值为 "LoadBalancer"，将为 Service 提供负载均衡器。 负载均衡器是异步创建的，关于被提供的负载均衡器的信息将会通过 Service 的 status.loadBalancer 字段被发布出去。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
      nodePort: 30061
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
```

```
    - ip: 146.148.47.155
```

来自外部负载均衡器的流量将直接打到 backend Pod 上，不过实际它们是如何工作的，这要依赖于云提供商。 在这些情况下，将根据用户设置的 loadBalancerIP 来创建负载均衡器。 某些云提供商允许设置 loadBalancerIP。如果没有设置 loadBalancerIP，将会给负载均衡器指派一个临时 IP。 如果设置了 loadBalancerIP，但云提供商并不支持这种特性，那么设置的 loadBalancerIP 值将会被忽略掉。

AWS 内部负载均衡器

在混合云环境中，有时从虚拟私有云（VPC）环境中的服务路由流量是非常有必要的。 可以通过在 Service 中增加 annotation 来实现，如下所示：

```
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0
[...]
```

在水平分割的 DNS 环境中，需要两个 Service 来将外部和内部的流量路由到 Endpoint 上。

AWS SSL 支持

对运行在 AWS 上部分支持 SSL 的集群，从 1.3 版本开始，可以为 LoadBalancer 类型的 Service 增加两个 annotation：

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-ssl-cert:
arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012
```

第一个 annotation 指定了使用的证书。它可以是第三方发行商发行的证书，这个证书或者被上传到 IAM，或者由 AWS 的证书管理器创建。

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-backend-protocol:
(https|http|ssl|tcp)
```

第二个 annotation 指定了 Pod 使用的协议。对于 HTTPS 和 SSL, ELB 将期望该 Pod 基于加密的连接来认证自身。

HTTP 和 HTTPS 将选择 7 层代理：ELB 将中断与用户的连接，当转发请求时，会解析 Header 信息并添加上用户的 IP 地址（Pod 将只能在连接的另一端看到该 IP 地址）。

TCP 和 SSL 将选择 4 层代理：ELB 将转发流量，并不修改 Header 信息。

外部 IP

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes Service 会被暴露给这些 externalIPs。 通过外部 IP（作为目的 IP 地址）进入到集群，打到 Service 的端口上的流量，将会被路由到 Service 的 Endpoint 上。 externalIPs 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

根据 Service 的规定，externalIPs 可以同任意的 ServiceType 来一起指定。 在上面的例子中，my-service 可以在 80.11.12.10:80（外部 IP:端口）上被客户端访问。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
  externalIPs:
    - 80.11.12.10
```

不足之处

为 VIP 使用 userspace 代理，将只适合小型到中型规模的集群，不能够扩展到上千 Service 的大型集群。 查看 最初设计方案 获取更多细节。

使用 userspace 代理，隐藏了访问 Service 的数据包的源 IP 地址。 这使得一些类型的防火墙无法起作用。 iptables 代理不会隐藏 Kubernetes 集群内部的 IP 地址，但却要求客户端请求必须通过一个负载均衡器或 Node 端口。

Type 字段支持嵌套功能 —— 每一层需要添加到上一层里面。 不会严格要求所有云提供商（例如，GCE 就没必要为了使一个 LoadBalancer 能工作而分配一个 NodePort，但是 AWS 需要 ），但当前 API 是强制要求的。

未来工作

未来我们能预见到，代理策略可能会变得比简单的 round-robin 均衡策略有更多细微的差别，比如 master 选举或分片。 我们也能想到，某些 Service 将具有 “真正” 的负载均衡器，这种情况下 VIP 将简化数据包的传输。

我们打算为 L7（HTTP）Service 改进我们对它的支持。

我们打算为 Service 实现更加灵活的请求进入模式，这些 Service 包含当前 ClusterIP、NodePort 和 LoadBalancer 模式，或者更多。

VIP 的那些骇人听闻的细节

对很多想使用 Service 的人来说，前面的信息应该足够了。 然而，有很多内部原理性的内容，还是值去理解的。

避免冲突

Kubernetes 最主要的哲学之一，是用户不应该暴露那些能够导致他们操作失败、但又不是他们的过错的场景。 这种场景下，让我们来看一下网络端口 —— 用户不应该必须选择一个端口号，而且该端口还有可能与其他用户的冲突。 这就是说，在彼此隔离状态下仍然会出现失败。

为了使用户能够为他们的 Service 选择一个端口号，我们必须确保不能有 2 个 Service 发生冲突。 我们可以通过为每个 Service 分配它们自己的 IP 地址来实现。

为了保证每个 Service 被分配到一个唯一的 IP，需要一个内部的分配器能够原子地更新 etcd 中的一个全局分配映射表，这个更新操作要先于创建每一个 Service。 为了使 Service 能够获取到 IP，这个映射表对象必须在注册中心存在，否则创建 Service 将会失败，指示一个 IP 不能被分配。 一个后台 Controller 的职责是创建映射表（从 Kubernetes 的旧版本迁移过来，旧版本中是通过在内存中加锁的方式实现），并检查由于管理员干预和清除任意 IP 造成的不合理分配，这些 IP 被分配了但当前没有 Service 使用它们。

IP 和 VIP

不像 Pod 的 IP 地址，它实际路由到一个固定的目的地，Service 的 IP 实际上不能通过单个主机来进行应答。 相反，我们使用 iptables（Linux 中的数据包处理逻辑）来定义一个虚拟 IP 地址（VIP），它可以根据需要透明地进行重定向。 当客户端连接到 VIP 时，它们的流量会自动地传输到一个合适的 Endpoint。环境变量和 DNS, 实际上会根据 Service 的 VIP 和端口来进行填充。

Userspace

作为一个例子，考虑前面提到的图片处理应用程序。 当创建 backend Service 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 Service 的端口是 1234，该 Service 会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会打开一个新的端口，建立一个从该 VIP 重定向到新端口的 iptables，并开始接收请求连接。

当一个客户端连接到一个 VIP，iptables 规则开始起作用，它会重定向该数据包到 Service 代理 的端口。 Service 代理 选择一个 backend，并将客户端的流量代理到 backend 上。

这意味着 Service 的所有者能够选择任何他们想使用的端口，而不存在冲突的风险。 客户端可以简单地连接到一个 IP 和端口，而不需要知道实际访问了哪些 Pod。

Iptables

再次考虑前面提到的图片处理应用程序。当创建 backend Service 时, Kubernetes master 会给它指派一个虚拟 IP 地址, 比如 10.0.0.1。 假设 Service 的端口是 1234, 该 Service 会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会安装一系列的 iptables 规则，从 VIP 重定向到 per-Service 规则。 该 per-Service 规则连接到 per-Endpoint 规则，该 per-Endpoint 规则会重定向（目标 NAT）到 backend。

当一个客户端连接到一个 VIP，iptables 规则开始起作用。一个 backend 会被选择（或者根据会话亲和性，或者随机），数据包被重定向到这个 backend。 不像 userspace 代理，数据包从来不拷贝到用户空间, kube-proxy 不是必须为该 VIP 工作而运行, 并且客户端 IP 是不可更改的。 当流量打到 Node 的端口上, 或通过负载均衡器, 会执行相同的基本流程，但是在那些案例中客户端 IP 是可以更改的。

API 对象

在 Kubernetes REST API 中，Service 是 top-level 资源。关于 API 对象的更多细节可以查看：Service API 对象。

Ingress

术语

在本篇文章中你将会看到一些在其他地方被交叉使用的术语，为了防止产生歧义，我们首先来澄清下。

- 节点：Kubernetes 集群中的一台物理机或者虚拟机。
- 集群：位于 Internet 防火墙后的节点，这是 kubernetes 管理的主要计算资源。
- 边界路由器：为集群强制执行防火墙策略的路由器。 这可能是由云提供商或物理硬件管理的网关。
- 集群网络：一组逻辑或物理链接，可根据 Kubernetes 网络模型实现群集内的通信。 集群网络的实现包括 Overlay 模型的 flannel 和基于 SDN 的 OVS。
- 服务：使用标签选择器标识一组 pod 成为的 Kubernetes 服务。 除非另有说明，否则服务假定在集群网络内仅可通过虚拟 IP 访问。

什么是 Ingress?

通常情况下，service 和 pod 仅可在集群内部网络中通过 IP 地址访问。所有到达边界路由器的流量或被丢弃或被转发到其他地方。从概念上讲，可能像下面这样：

```
    internet
        |
  ------------
  [ Services ]
```

Ingress 是授权入站连接到达集群服务的规则集合。

```
    internet
        |
  [ Ingress ]
  --|-----|--
  [ Services ]
```

你可以给 Ingress 配置提供外部可访问的 URL、负载均衡、SSL、基于名称的虚拟主机等。用户通过 POST Ingress 资源到 API server 的方式来请求 ingress。 Ingress controller 负责实现 Ingress，通常使用负载平衡器，它还可以配置边界路由和其他前端，这有助于以 HA 方式处理流量。

先决条件

在使用 Ingress resource 之前，有必要先了解下面几件事情。Ingress 是 beta 版本的 resource，在 kubernetes1.1 之前还没有。你需要一个 Ingress Controller 来实现 Ingress，单纯的创建一个 Ingress 没有任何意义。

GCE/GKE 会在 master 节点上部署一个 ingress controller。你可以在一个 pod 中部署任意个自定义的 ingress controller。你必须正确地 annotate 每个 ingress，比如 运行多个 ingress controller 和 关闭 glbc.

确定你已经阅读了 Ingress controller 的 beta 版本限制。在非 GCE/GKE 的环境中，你需要在 pod 中部署一个 controller。

Ingress Resource

最简化的 Ingress 配置：

```
1: apiVersion: extensions/v1beta1
2: kind: Ingress
3: metadata:
4:     name: test-ingress
5: spec:
6:     rules:
7:     - http:
8:         paths:
9:         - path: /testpath
10:           backend:
11:               serviceName: test
12:               servicePort: 80
```

如果你没有配置 Ingress controller 就将其 POST 到 API server 不会有任何用处

配置说明

1-4 行：跟 Kubernetes 的其他配置一样，ingress 的配置也需要 apiVersion，kind 和 metadata 字段。配置文件的详细说明请查看部署应用，配置容器和 使用 resources.

5-7 行：Ingress spec 中包含配置一个 loadbalancer 或 proxy server 的所有信息。最重要的是，它包含了一个匹配所有入站请求的规则列表。目前 ingress 只支持 http 规则。

8-9 行：每条 http 规则包含以下信息：一个 host 配置项（比如 for.bar.com，在这个例子中默认是*），path 列表（比如：/testpath），每个 path 都关联一个 backend(比如 test:80)。在 loadbalancer 将流量转发到 backend 之前，所有的入站请求都要先匹配 host 和 path。

10-12 行：正如 services doc 中描述的那样，backend 是一个 service:port 的组合。Ingress 的流量被转发到它所匹配的 backend。

全局参数：为了简单起见，Ingress 示例中没有全局参数，请参阅资源完整定义的 api 参考。在所有请求都不能跟 spec 中的 path 匹配的情况下，请求被发送到 Ingress controller 的默认后端，可以指定全局缺省 backend。

Ingress Controllers

为了使 Ingress 正常工作，集群中必须运行 Ingress controller。 这与其他类型的控制器不同，其他类型的控制器通常作为 kube-controller-manager 二进制文件的一部分运行，在集群启动时自动启动。 你需要选择最适合自己集群的 Ingress controller 或者自己实现一个。示例和说明可以在这里找到。

在你开始前

以下文档描述了 Ingress 资源中公开的一组跨平台功能。 理想情况下，所有的 Ingress controller 都应该符合这个规范，但是我们还没有实现。 GCE 和 nginx 控制器的文档分别在这里和这里。确保您查看控制器特定的文档，以便您了解每个文档的注意事项。

Ingress 类型

单 Service Ingress
Kubernetes 中已经存在一些概念可以暴露单个 service（查看替代方案），但是你仍然可以通过 Ingress 来实现，通过指定一个没有 rule 的默认 backend 的方式。

ingress.yaml 定义文件：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

使用 kubectl create -f 命令创建，然后查看 ingress：

```
$ kubectl get ing
```

```
NAME                    RULE          BACKEND          ADDRESS
test-ingress            -             testsvc:80       107.178.254.228
```

107.178.254.228 就是 Ingress controller 为了实现 Ingress 而分配的 IP 地址。RULE 列表示所有发送给该 IP 的流量都被转发到了 BACKEND 所列的 Kubernetes service 上。

简单展开

如前面描述的那样，kubernete pod 中的 IP 只在集群网络内部可见，我们需要在边界设置一个东西，让它能够接收 ingress 的流量并将它们转发到正确的端点上。这个东西一般是高可用的 loadbalancer。使用 Ingress 能够允许你将 loadbalancer 的个数降低到最少，例如，嫁入你想要创建这样的一个设置：

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

你需要一个这样的 ingress：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

使用 kubectl create -f 创建完 ingress 后：

```
$ kubectl get ing
NAME      RULE          BACKEND     ADDRESS
test      -
          foo.bar.com
          /foo          s1:80
          /bar          s2:80
```

只要服务（s1，s2）存在，Ingress controller 就会将提供一个满足该 Ingress 的特定 loadbalancer 实现。 这一步完成后，您将在 Ingress 的最后一列看到 loadbalancer 的地址。

基于名称的虚拟主机

Name-based 的虚拟主机在同一个 IP 地址下拥有多个主机名。

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

下面这个 ingress 说明基于 Host header 的后端 loadbalancer 的路由请求：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

默认 backend：一个没有 rule 的 ingress，如前面章节中所示，所有流量都将发送到一个默认 backend。你可以用该技巧通知 loadbalancer 如何找到你网站的 404 页面，通过制定一些列 rule 和一个默认 backend 的方式。如果请求 header 中的 host 不能跟 ingress 中的 host 匹配，并且/或请求的 URL 不能与任何一个 path 匹配，则流量将路由到你的默认 backend。

TLS

你可以通过指定包含 TLS 私钥和证书的 secret 来加密 Ingress。 目前，Ingress 仅支持单个 TLS 端口 443，并假定 TLS termination。 如果 Ingress 中的 TLS 配置部分指定了不同的主机，则它们将根据通过 SNI TLS 扩展指定的主机名（假如 Ingress controller 支持 SNI）在多个相同端口上进行复用。 TLS secret 中必须包含名为 tls.crt 和 tls.key 的密钥，这里面包含了用于 TLS 的证书和私钥，例如：

```
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret
  namespace: default
type: Opaque
```

在 Ingress 中引用这个 secret 将通知 Ingress controller 使用 TLS 加密从将客户端到 loadbalancer 的 channel：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
    - secretName: testsecret
  backend:
    serviceName: s1
    servicePort: 80
```

请注意，各种 Ingress controller 支持的 TLS 功能之间存在差距。 请参阅有关 nginx，GCE 或任何其他平台特定 Ingress controller 的文档，以了解 TLS 在你的环境中的工作原理。

Ingress controller 启动时附带一些适用于所有 Ingress 的负载平衡策略设置，例如负载均衡算法，后端权重方案等。更高级的负载平衡概念（例如持久会话，动态权重）尚未在 Ingress 中公开。 你仍然可以通过 service loadbalancer 获取这些功能。 随着时间的推移，我们计划将适用于跨平台的负载平衡模式加入到 Ingress 资源中。

还值得注意的是，尽管健康检查不直接通过 Ingress 公开，但 Kubernetes 中存在并行概念，例如准备探查，可以使你达成相同的最终结果。 请查看特定控制器的文档，以了解他们如何处理健康检查（nginx，GCE）。

更新 Ingress

假如你想要向已有的 ingress 中增加一个新的 Host，你可以编辑和更新该 ingress：

```
$ kubectl get ing
NAME        RULE          BACKEND     ADDRESS
test        -                         178.91.123.132
```

```
              foo.bar.com
              /foo              s1:80
$ kubectl edit ing test
```

这会弹出一个包含已有的 yaml 文件的编辑器，修改它，增加新的 Host 配置。

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
        path: /foo
  - host: bar.baz.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
        path: /foo
..
```

保存它会更新 API server 中的资源，这会触发 ingress controller 重新配置 loadbalancer。

```
$ kubectl get ing
NAME        RULE          BACKEND     ADDRESS
test        -                         178.91.123.132
              foo.bar.com
              /foo              s1:80
              bar.baz.com
              /foo              s2:80
```

在一个修改过的 ingress yaml 文件上调用 kubectl replace -f 命令一样可以达到同样的效果。

跨可用域故障

在不通云供应商之间，跨故障域的流量传播技术有所不同。 有关详细信息，请查看相关 Ingress controller 的文档。 有关在 federation 集群中部署 Ingress 的详细信息，请参阅 [federation 文档]()。

未来计划

- 多样化的 HTTPS/TLS 模型支持（如 SNI，re-encryption）
- 通过声明来请求 IP 或者主机名
- 结合 L4 和 L7 Ingress
- 更多的 Ingress controller

请跟踪 L7 和 Ingress 的 proposal，了解有关资源演进的更多细节，以及 Ingress repository，了解有关各种 Ingress controller 演进的更多详细信息。

替代方案

你可以通过很多种方式暴露 service 而不必直接使用 ingress：

- 使用 Service.Type=LoadBalancer
- 使用 Service.Type=NodePort
- 使用 Port Proxy
- 部署一个 Service loadbalancer 这允许你在多个 service 之间共享单个 IP，并通过 Service Annotations 实现更高级的负载平衡。

Job

Job 负责批处理任务，即仅执行一次的任务，它保证批处理任务的一个或多个 Pod 成功结束。

- Job Spec 格式
- spec.template 格式同 Pod
- RestartPolicy 仅支持 Never 或 OnFailure
- 单个 Pod 时，默认 Pod 成功运行后 Job 即结束
- .spec.completions 标志 Job 结束需要成功运行的 Pod 个数，默认为 1
- .spec.parallelism 标志并行运行的 Pod 的个数，默认为 1
- spec.activeDeadlineSeconds 标志失败 Pod 的重试最大时间，超过这个时间不会继续重试

一个简单的例子：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
```

```
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

```
$ kubectl create -f ./job.yaml
job "pi" created
$ pods=$(kubectl get pods --selector=job-name=pi
--output=jsonpath={.items..metadata.name})
$ kubectl logs $pods -c pi3.14159265358979323846264338327950 2...
```

所谓 Bare Pods 是指直接用 PodSpec 来创建的 Pod（即不在 ReplicaSets 或者 ReplicationController 的管理之下的 Pods）。这些 Pod 在 Node 重启后不会自动重启，但 Job 则会创建新的 Pod 继续任务。所以，推荐使用 Job 来替代 Bare Pods，即便是应用只需要一个 Pod。

Annotation

Annotation，顾名思义，就是注解。Annotation 可以将 Kubernetes 资源对象关联到任意的非标识性元数据。使用客户端（如工具和库）可以检索到这些元数据。

关联元数据到对象

Label 和 Annotation 都可以将元数据关联到 Kubernetes 资源对象。Label 主要用于选择对象，可以挑选出满足特定条件的对象。相比之下，annotation 不能用于标识及选择对象。annotation 中的元数据可多可少，可以是结构化的或非结构化的，也可以包含 label 中不允许出现的字符。

annotation 和 label 一样都是 key/value 键值对映射结构：

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

以下列出了一些可以记录在 annotation 中的对象信息：

- 声明配置层管理的字段。使用 annotation 关联这类字段可以用于区分以下几种配置来源：客户端或服务器设置的默认值，自动生成的字段或自动生成的 auto-scaling 和 auto-sizing 系统配置的字段。
- 创建信息、版本信息或镜像信息。例如时间戳、版本号、git 分支、PR 序号、镜像哈希值以及仓库地址。
- 记录日志、监控、分析或审计存储仓库的指针
- 可以用于 debug 的客户端（库或工具）信息，例如名称、版本和创建信息。
- 用户信息，以及工具或系统来源信息、例如来自非 Kubernetes 生态的相关对象的 URL 信息。
- 轻量级部署工具元数据，例如配置或检查点。
- 负责人的电话或联系方式，或能找到相关信息的目录条目信息，例如团队网站。

如果不使用 annotation，您也可以将以上类型的信息存放在外部数据库或目录中，但这样做不利于创建用于部署、管理、内部检查的共享工具和客户端库。

示例

如 Istio 的 Deployment 配置中就使用到了 annotation：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: istio-manager
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        alpha.istio.io/sidecar: ignore
      labels:
        istio: manager
    spec:
      serviceAccountName: istio-manager-service-account
      containers:
      - name: discovery
        image: harbor-001.jimmysong.io/library/manager:0.1.5
        imagePullPolicy: Always
        args: ["discovery", "-v", "2"]
        ports:
        - containerPort: 8080
        env:
        - name: POD_NAMESPACE
          valueFrom:
```

```
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
      - name: apiserver
        image: harbor-001.jimmysong.io/library/manager:0.1.5
        imagePullPolicy: Always
        args: ["apiserver", "-v", "2"]
        ports:
        - containerPort: 8081
        env:
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
```

alpha.istio.io/sidecar 注解就是用来控制是否自动向 pod 中注入 sidecar 的。

Labels 和 Selectors

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

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

我们最终会索引并且反向索引（reverse-index）labels，以获得更高效的查询和监视，把他们用到 UI 或者 CLI 中用来排序或者分组等等。我们不想用那些不具有指认效果的 label 来污染 label，特别是那些体积较大和结构型的的数据。不具有指认效果的信息应该使用 annotation 来记录。

Motivation

Labels 可以让用户将他们自己的有组织目的的结构以一种松耦合的方式应用到系统的对象上，且不需要客户端存放这些对应关系（mappings）。

服务部署和批处理管道通常是多维的实体（例如多个分区或者部署，多个发布轨道，多层，

每层多微服务）。管理通常需要跨越式的切割操作，这会打破有严格层级展示关系的封装，特别对那些是由基础设施而非用户决定的很死板的层级关系。

示例标签：

- "release" : "stable"， "release" : "canary"
- "environment" : "dev"， "environment" : "qa"， "environment" : "production"
- "tier" : "frontend"， "tier" : "backend"， "tier" : "cache"
- "partition" : "customerA"， "partition" : "customerB"
- "track" : "daily"， "track" : "weekly"

这些只是常用 Labels 的例子，你可以按自己习惯来定义，需要注意，每个对象的标签 key 具有唯一性。

语法和字符集

Label 其实是一对 key/value。有效的标签键有两个段：可选的前缀和名称，用斜杠（/）分隔，名称段是必需的，最多 63 个字符，以[a-z0-9A-Z]带有虚线（-）、下划线（_）、点（.）和开头和结尾必须是字母或数字（都是字符串形式）的形式组成。前缀是可选的。如果指定了前缀，那么必须是 DNS 子域：一系列的 DNSlabel 通过”.”来划分，不超过 253 个字符，以斜杠（/）结尾。如果前缀被省略了，这个 Label 的 key 被假定为对用户私有的。自动化系统组件有（例如 kube-scheduler，kube-controller-manager，kube-apiserver，kubectl，或其他第三方自动化），这些添加标签终端用户对象都必须指定一个前缀。Kuberentes.io 前缀是为 Kubernetes 内核部分保留的。

有效的标签值最长为 63 个字符。要么为空，要么使用[a-z0-9A-Z]带有虚线（-）、下划线（_）、点（.）和开头和结尾必须是字母或数字（都是字符串形式）的形式组成。

Labels 选择器

与 Name 和 UID 不同，标签不需要有唯一性。一般来说，我们期望许多对象具有相同的标签。

通过标签选择器（Labels Selectors），客户端/用户 能方便辨识出一组对象。标签选择器是 kubernetes 中核心的组成部分。

API 目前支持两种选择器：equality-based（基于平等）和 set-based（基于集合）的。标签选择器可以由逗号分隔的多个 requirements 组成。在多重需求的情况下，必须满足所有要求，因此逗号分隔符作为 AND 逻辑运算符。

一个为空的标签选择器（即有 0 个必须条件的选择器）会选择集合中的每一个对象。

一个 null 型标签选择器（仅对于可选的选择器字段才可能）不会返回任何对象。

注意：两个控制器的标签选择器不能在命名空间中重叠。

Equality-based requirement 基于相等的要求

基于相等的或者不相等的条件允许用标签的 keys 和 values 进行过滤。匹配的对象必须满足所有指定的标签约束，尽管他们可能也有额外的标签。有三种运算符是允许的，“=”，“==”和“!=”。前两种代表相等性（他们是同义运算符），后一种代表非相等性。例如：

environment = production
tier != frontend
第一个选择所有 key 等于 environment 值为 production 的资源。后一种选择所有 key 为 tier 值不等于 frontend 的资源，和那些没有 key 为 tier 的 label 的资源。要过滤所有处于 production 但不是 frontend 的资源，可以使用逗号操作符，

frontend：environment=production,tier!=frontend

Set-based requirement

Set-based 的标签条件允许用一组 value 来过滤 key。支持三种操作符: in ， notin 和 exists(仅针对于 key 符号) 。例如：

environment in (production, qa)
tier notin (frontend, backend)
partition
!partition

第一个例子，选择所有 key 等于 environment ，且 value 等于 production 或者 qa 的资源。 第二个例子，选择所有 key 等于 tier 且值是除了 frontend 和 backend 之外的资源，和那些没有标签的 key 是 tier 的资源。第三个例子，选择所有有一个标签的 key 为 partition 的资源；value 是什么不会被检查。第四个例子，选择所有的没有 lable 的 key 名为 partition 的资源；value 是什么不会被检查。

类似的，逗号操作符相当于一个 AND 操作符。因而要使用一个 partition 键（不管 value 是什么），并且 environment 不是 qa 过滤资源可以用 partition,environment notin (qa) 。

Set-based 的选择器是一个相等性的宽泛的形式，因为 environment=production 相当于 environment in (production) ，与 != and notin 类似。

Set-based 的条件可以与 Equality-based 的条件结合。例如， partition in

(customerA,customerB),environment!=qa 。

API

LIST 和 WATCH 过滤

LIST 和 WATCH 操作可以指定标签选择器来过滤使用查询参数返回的对象集。这两个要求都是允许的（在这里给出，它们会出现在 URL 查询字符串中）：

LIST 和 WATCH 操作，可以使用 query 参数来指定 label 选择器来过滤返回对象的集合。两种条件都可以使用：

- Set-based 的要求：?labelSelector=environment%3Dproduction,tier%3Dfrontend
- Equality-based 的要求：?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29

两个标签选择器样式都可用于通过 REST 客户端列出或观看资源。例如，apiserver 使用 kubectl 和使用基于平等的人可以写：

两种标签选择器样式，都可以通过 REST 客户端来 list 或 watch 资源。比如使用 kubectl 来针对 apiserver ，并且使用 Equality-based 的条件，可以用：

```
$ kubectl get pods -l environment=production,tier=frontend
```

或使用 Set-based 要求：

```
$ kubectl get pods -l 'environment in (production),tier in (frontend)'
```

如已经提到的 Set-based 要求更具表现力。例如，它们可以对 value 执行 OR 运算：

```
$ kubectl get pods -l 'environment in (production, qa)'
```

或者通过 exists 操作符进行否定限制匹配：

```
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

API 对象中引用

一些 Kubernetes 对象，例如 services 和 replicationcontrollers，也使用标签选择器来指定其他资源的集合，如 pod。

Service 和 ReplicationController

一个 service 针对的 pods 的集合是用标签选择器来定义的。类似的，一个 replicationcontroller 管理的 pods 的群体也是用标签选择器来定义的。

对于这两种对象的 Label 选择器是用 map 定义在 json 或者 yaml 文件中的，并且只支持 Equality-based 的条件：

```
"selector": {
    "component" : "redis",
}
```

要么

```
selector:
    component: redis
```

此选择器（分别为 json 或 yaml 格式）等同于 component=redis 或 component in (redis)。

支持 set-based 要求的资源

Job，Deployment，Replica Set，和 Daemon Set，支持 set-based 要求。

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

matchLabels 是一个{key,value}的映射。一个单独的 {key,value} 相当于 matchExpressions 的一个元素，它的 key 字段是”key”,操作符是 In ，并且 value 数组 value 包含”value”。 matchExpressions 是一个 pod 的选择器条件的 list 。有效运算符包含 In, NotIn, Exists，和 DoesNotExist。在 In 和 NotIn 的情况下，value 的组必须不能为空。所有的条件，包含 matchLabels andmatchExpressions 中的，会用 AND 符号连接，他们必须都被满足以完成匹配。

ConfigMap

其实 ConfigMap 功能在 Kubernetes1.2 版本的时候就有了，许多应用程序会从配置文件、命令行参数或环境变量中读取配置信息。这些配置信息需要与 docker image 解耦，你总不能每修改一个配置就重做一个 image 吧？ConfigMap API 给我们提供了向容器中注入配置信息的机制，ConfigMap 可以被用来保存单个属性，也可以用来保存整个配置文件或者 JSON 二进制大对象。

ConfigMap 概览

ConfigMap API 资源用来保存 key-value pair 配置数据，这个数据可以在 pods 里使用，或者被用来为像 controller 一样的系统组件存储配置数据。虽然 ConfigMap 跟 Secrets 类似，但是 ConfigMap 更方便的处理不含敏感信息的字符串。 注意：ConfigMaps 不是属性配置文件的替代品。ConfigMaps 只是作为多个 properties 文件的引用。你可以把它理解为 Linux 系统中的/etc 目录，专门用来存储配置文件的目录。下面举个例子，使用 ConfigMap 配置来创建 Kubernetes Volumes，ConfigMap 中的每个 data 项都会成为一个新文件。

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

data 一栏包括了配置数据，ConfigMap 可以被用来保存单个属性，也可以用来保存一个配置文件。 配置数据可以通过很多种方式在 Pods 里被使用。ConfigMaps 可以被用来：

1. 设置环境变量的值
2. 在容器里设置命令行参数
3. 在数据卷里面创建 config 文件

用户和系统组件两者都可以在 ConfigMap 里面存储配置数据。

其实不用看下面的文章，直接从 kubectl create configmap -h 的帮助信息中就可以对 ConfigMap 究竟如何创建略知一二了。

```
Examples:
  # Create a new configmap named my-config based on folder bar
  kubectl create configmap my-config --from-file=path/to/bar

  # Create a new configmap named my-config with specified keys instead of file
basenames on disk
  kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt
--from-file=key2=/path/to/bar/file2.txt
```

```
  # Create a new configmap named my-config with key1=config1 and key2=config2
  kubectl create configmap my-config --from-literal=key1=config1
--from-literal=key2=config2
```

创建 ConfigMaps

可以使用该命令，用给定值、文件或目录来创建 ConfigMap。

kubectl create configmap

使用目录创建

比如我们已经有了一些配置文件，其中包含了我们想要设置的 ConfigMap 的值：

```
$ ls docs/user-guide/configmap/kubectl/
game.properties
ui.properties

$ cat docs/user-guide/configmap/kubectl/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat docs/user-guide/configmap/kubectl/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

使用下面的命令可以创建一个包含目录中所有文件的 ConfigMap。

```
$ kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl
```

—from-file 指定在目录下的所有文件都会被用在 ConfigMap 里面创建一个键值对，键的名字就是文件名，值就是文件的内容。

让我们来看一下这个命令创建的 ConfigMap：

```
$ kubectl describe configmaps game-config
Name:                   game-config
Namespace:          default
Labels:               <none>
Annotations:       <none>

Data
====
game.properties:              158 bytes
ui.properties:                83 bytes
```

我们可以看到那两个 key 是从 kubectl 指定的目录中的文件名。这些 key 的内容可能会很大，所以在 kubectl describe 的输出中，只能够看到键的名字和他们的大小。 如果想要看到键的值的话，可以使用 kubectl get：

```
$ kubectl get configmaps game-config -o yaml
```

我们以 yaml 格式输出配置。

```
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selfLink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

使用文件创建

刚才使用目录创建的时候我们—from-file 指定的是一个目录, 只要指定为一个文件就可以从单个文件中创建 ConfigMap。

```
$ kubectl create configmap game-config-2
--from-file=docs/user-guide/configmap/kubectl/game.properties

$ kubectl get configmaps game-config-2 -o yaml

apiVersion: v1
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

—from-file 这个参数可以使用多次，你可以使用两次分别指定上个实例中的那两个配置文件，效果就跟指定整个目录是一样的。

使用字面值创建

使用文字值创建，利用—from-literal 参数传递配置信息，该参数可以使用多次，格式如下；

```
$ kubectl create configmap special-config --from-literal=special.how=very
--from-literal=special.type=charm
$ kubectl get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
```

```
    special.type: charm
kind: ConfigMap
metadata:
    creationTimestamp: 2016-02-18T19:14:38Z
    name: special-config
    namespace: default
    resourceVersion: "651"
    selfLink: /api/v1/namespaces/default/configmaps/special-config
    uid: dadce046-d673-11e5-8cd0-68f728db1985
```

Pod 中使用 ConfigMap

使用 ConfigMap 来替代环境变量

ConfigMap 可以被用来填入环境变量。看下下面的 ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: special-config
    namespace: default
data:
    special.how: very
    special.type: charm

apiVersion: v1
kind: ConfigMap
metadata:
    name: env-config
    namespace: default
data:
    log_level: INFO
```

我们可以在 Pod 中这样使用 ConfigMap：

```
apiVersion: v1
kind: Pod
metadata:
    name: dapi-test-pod
spec:
    containers:
        - name: test-container
            image: gcr.io/google_containers/busybox
```

```
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

这个 Pod 运行后会输出如下几行：

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
log_level=INFO
```

用 ConfigMap 设置命令行参数

ConfigMap 也可以被使用来设置容器中的命令或者参数值。它使用的是 Kubernetes 的 $(VAR_NAME)替换语法。我们看下下面这个 ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

为了将 ConfigMap 中的值注入到命令行的参数里面，我们还要像前面那个例子一样使用环境变量替换语法${VAR_NAME)。（其实这个东西就是给 Docker 容器设置环境变量，以前我创建镜像的时候经常这么玩，通过 docker run 的时候指定-e 参数修改镜像里的环境变量，然后 docker 的 CMD 命令再利用该$(VAR_NAME)通过 sed 来修改配置文件或者作为命令行启动参数。）

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

运行这个 Pod 后会输出：

```
very charm
```

通过数据卷插件使用 ConfigMap

ConfigMap 也可以在数据卷里面被使用。还是这个 ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

在数据卷里面使用这个 ConfigMap，有不同的选项。最基本的就是将文件填入数据卷，在这个文件中，键就是文件名，键值就是文件内容：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

运行这个 Pod 的输出是 very。

我们也可以在 ConfigMap 值被映射的数据卷里控制路径。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh","-c","cat /etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never
```

运行这个 Pod 后的结果是 very。

参考：https://jimmysong.io/kubernetes-handbook/concepts/configmap.html

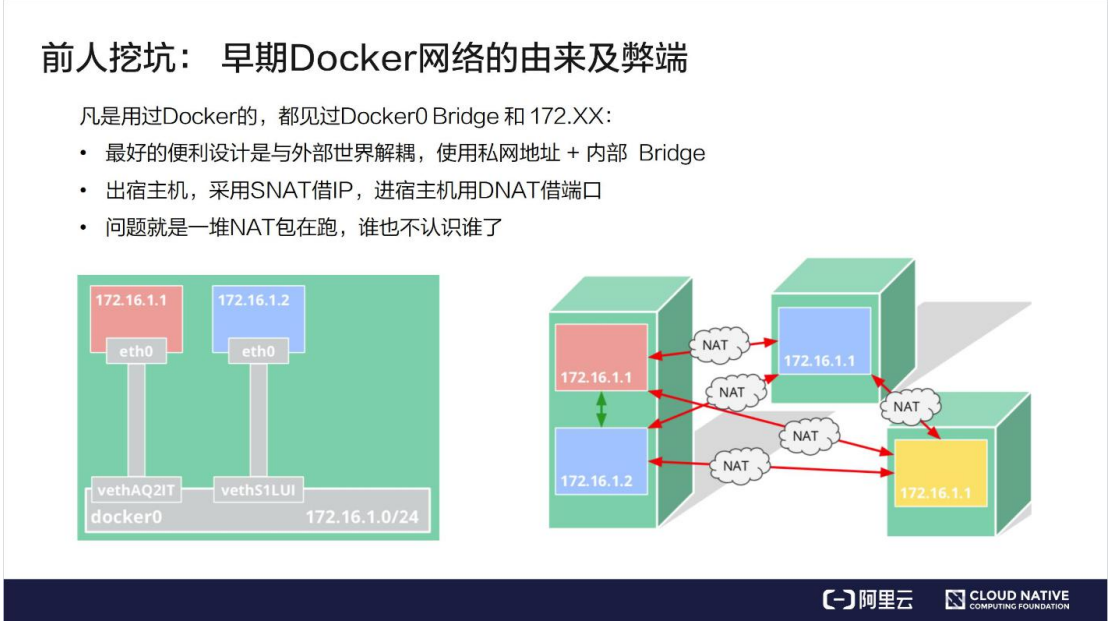
网络

Kubernetes 网络模型进阶

作者 | 叶磊（稻农）阿里巴巴高级技术专家

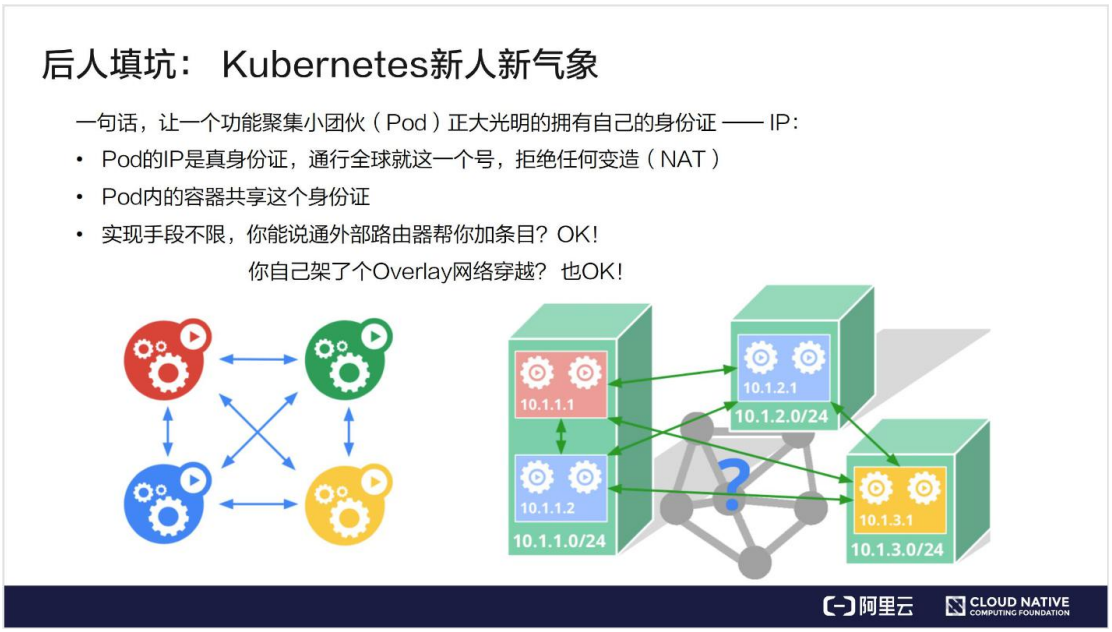
导读：本文将基于之前介绍的基本网络模型，进行更深入的一些了解，希望给予读者一个更广更深的认知。首先简单回顾一下容器网络的历史沿革，剖析一下 Kubernetes 网络模型的由来；其次会剖析一个实际的实现（Flannel Hostgw），展现了数据包从容器到宿主机的变换过程；最后对于和网络息息相关的 Servcie 做了比较深入的机制和使用介绍，通过一个简单的例子说明了 Service 的工作原理。

Kubernetes 网络模型来龙去脉



容器网络发端于 Docker 的网络。Docker 使用了一个比较简单的网络模型，即内部的网桥加内部的保留 IP。这种设计的好处在于容器的网络和外部世界是解耦的，无需占用宿主机的 IP 或者宿主机的资源，完全是虚拟的。它的设计初衷是：当需要访问外部世界时，会采用 SNAT 这种方法来借用 Node 的 IP 去访问外面的服务。比如容器需要对外提供服务的时候，所用的是 DNAT 技术，也就是在 Node 上开一个端口，然后通过 iptable 或者别的某些机制，把流导入到容器的进程上以达到目的。

该模型的问题在于，外部网络无法区分哪些是容器的网络与流量、哪些是宿主机的网络与流量。比如，如果要做一个高可用的时候，172.16.1.1 和 172.16.1.2 是拥有同样功能的两个容器，此时我们需要将两者绑成一个 Group 对外提供服务，而这个时候我们发现从外部看来两者没有相同之处，它们的 IP 都是借用宿主机的端口，因此很难将两者归拢到一起。



在此基础上，Kubernetes 提出了这样一种机制：即每一个 Pod，也就是一个功能聚集小团伙应有自己的“身份证”，或者说 ID。在 TCP 协议栈上，这个 ID 就是 IP。

这个 IP 是真正属于该 Pod 的，外部世界不管通过什么方法一定要给它。对这个 Pod IP 的访问就是真正对它的服务的访问，中间拒绝任何的变造。比如以 10.1.1.1 的 IP 去访问 10.1.2.1 的 Pod，结果到了 10.1.2.1 上发现，它实际上借用的是宿主机的 IP，而不是源 IP，这样是不被允许的。Pod 内部会要求共享这个 IP，从而解决了一些功能内聚的容器如何变成一个部署的原子的问题。

剩下的问题是我们的部署手段。Kubernetes 对怎么实现这个模型其实是没有什么限制的，用 underlay 网络来控制外部路由器进行导流是可以的；如果希望解耦，用 overlay 网络在底层网络之上再加一层叠加网，这样也是可以的。总之，只要达到模型所要求的目的即可。

Pod 究竟如何上网

容器网络的网络包究竟是怎么传送的？

我们可以从以下两个维度来看：

- 协议层次
- 网络拓扑

第一个维度是协议层次。

它和 TCP 协议栈的概念是相同的，需要从两层、三层、四层一层层地摞上去，发包的时候从右往左，即先有应用数据，然后发到了 TCP 或者 UDP 的四层协议，继续向下传送，加上 IP 头，再加上 MAC 头就可以送出去了。收包的时候则按照相反的顺序，首先剥离 MAC 的头，再剥离 IP 的头，最后通过协议号在端口找到需要接收的进程。

第二维度是网络拓扑。

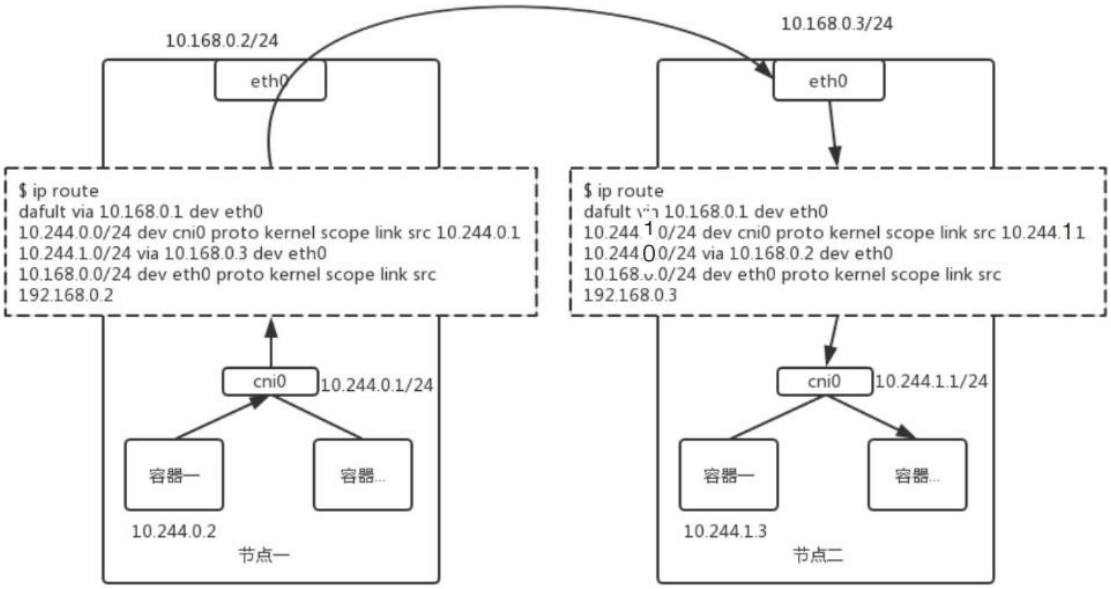
一个容器的包所要解决的问题分为两步：第一步，如何从容器的空间 (c1) 跳到宿主机的空间 (infra)；第二步，如何从宿主机空间到达远端。

我个人的理解是，容器网络的方案可以通过接入、流控、通道这三个层面来考虑。

- 第一个是接入，就是说我们的容器和宿主机之间是使用哪一种机制做连接，比如 Veth + bridge、Veth + pair 这样的经典方式，也有利用高版本内核的新机制等其他方式（如 mac/IPvlan 等），来把包送入到宿主机空间；
- 第二个是流控，就是说我的这个方案要不要支持 Network Policy，如果支持的话又要用何种方式去实现。这里需要注意的是，我们的实现方式一定需要在数据路径必经的一个关节点上。如果数据路径不通过该 Hook 点，那就不会起作用；
- 第三个是通道，即两个主机之间通过什么方式完成包的传输。我们有很多种方式，比如以路由的方式，具体又可分为 BGP 路由或者直接路由。还有各种各样的隧道技术等等。最终我们实现的目的就是一个容器内的包通过容器，经过接入层传到宿主机，再穿越宿主机的流控模块（如果有）到达通道送到对端。

一个最简单的路由方案：Flannel-host-gw

这个方案采用的是每个 Node 独占网段，每个 Subnet 会绑定在一个 Node 上，网关也设置在本地，或者说直接设在 cni0 这个网桥的内部端口上。该方案的好处是管理简单，坏处就是无法跨 Node 迁移 Pod。就是说这个 IP、网段已经是属于这个 Node 之后就无法迁移到别的 Node 上。



这个方案的精髓在于 route 表的设置，如上图所示。接下来为大家一一解读一下。

- 第一条很简单，我们在设置网卡的时候都会加上这一行。就是指定我的默认路由是通过哪个 IP 走掉，默认设备又是什么；
- 第二条是对 Subnet 的一个规则反馈。就是说我的这个网段是 10.244.0.0，掩码是 24 位，它的网关地址就在网桥上，也就是 10.244.0.1。这就是说这个网段的每一个包都发到这个网桥的 IP 上；
- 第三条是对对端的一个反馈。如果你的网段是 10.244.1.0（上图右边的 Subnet），我们就把它的 Host 的网卡上的 IP (10.168.0.3) 作为网关。也就是说，如果数据包是往 10.244.1.0 这个网段发的，就请以 10.168.0.3 作为网关。

再来看一下这个数据包到底是如何跑起来的？

假设容器 (10.244.0.2) 想要发一个包给 10.244.1.3，那么它在本地产生了 TCP 或者 UDP 包之后，再依次填好对端 IP 地址、本地以太网的 MAC 地址作为源 MAC 以及对端 MAC。一般来说本地会设定一条默认路由，默认路由会把 cni0 上的 IP 作为它的默认网关，对端的 MAC 就是这个网关的 MAC 地址。然后这个包就可以发到桥上去了。如果网段在本桥上，那么通过 MAC 层的交换即可解决。

这个例子中我们的 IP 并不属于本网段，因此网桥会将其上送到主机的协议栈去处理。主机协议栈恰好找到了对端的 MAC 地址。使用 10.168.0.3 作为它的网关，通过本地 ARP 探

查后，我们得到了 10.168.0.3 的 MAC 地址。即通过协议栈层层组装，我们达到了目的，将 Dst-MAC 填为右图主机网卡的 MAC 地址，从而将包从主机的 eth0 发到对端的 eth0 上去。

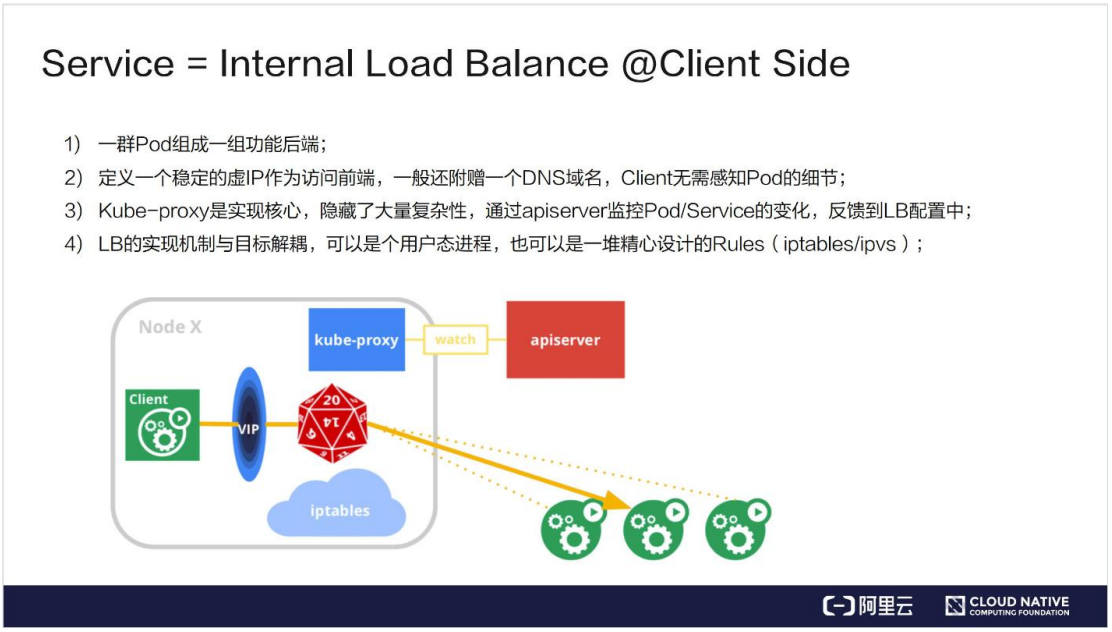
所以大家可以发现，这里有一个隐含的限制，上图中的 MAC 地址填好之后一定是能到达对端的，但如果这两个宿主机之间不是二层连接的，中间经过了一些网关、一些复杂的路由，那么这个 MAC 就不能直达，这种方案就是不能用的。当包到达了对端的 MAC 地址之后，发现这个包确实是给它的，但是 IP 又不是它自己的，就开始 Forward 流程，包上送到协议栈，之后再走一遍路由，刚好会发现 10.244.1.0/24 需要发到 10.244.1.1 这个网关上，从而到达了 cni0 网桥，它会找到 10.244.1.3 对应的 MAC 地址，再通过桥接机制，这个包就到达了对端容器。

大家可以看到，整个过程总是二层、三层，发的时候又变成二层，再做路由，就是一个大环套小环。这是一个比较简单的方案，如果中间要走隧道，则可能会有一条 vxlan tunnel 的设备，此时就不填直接的路由，而填成对端的隧道号。

Service 究竟如何工作

Service 其实是一种负载均衡 (Load Balance) 的机制。

我们认为它是一种用户侧(Client Side) 的负载均衡，也就是说 VIP 到 RIP 的转换在用户侧就已经完成了，并不需要集中式地到达某一个 NGINX 或者是一个 ELB 这样的组件来进行决策。

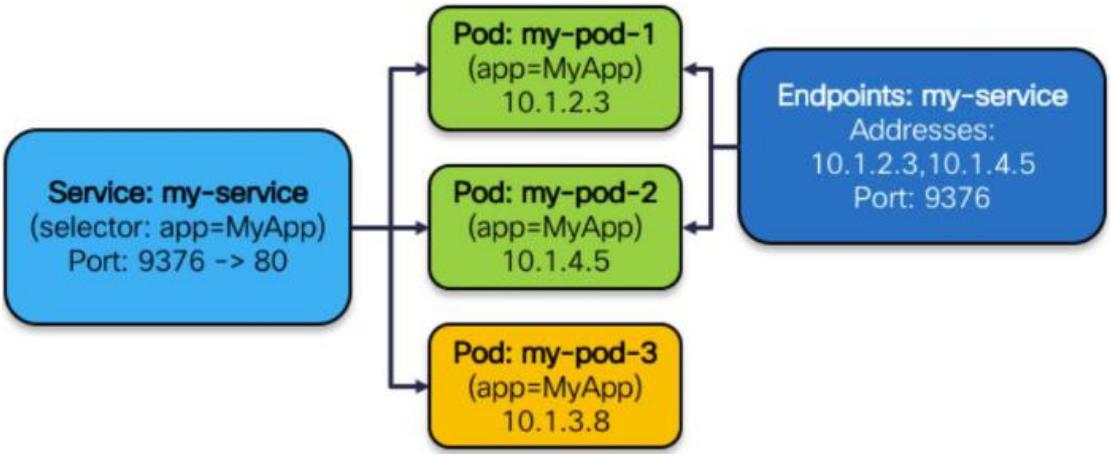


它的实现是这样的：首先是由一群 Pod 组成一组功能后端，再在前端上定义一个虚 IP 作为访问入口。一般来说，由于 IP 不太好记，我们还会附赠一个 DNS 的域名，Client 先访问域名得到虚 IP 之后再转成实 IP。Kube-proxy 则是整个机制的实现核心，它隐藏了大量的复杂性。它的工作机制是通过 apiserver 监控 Pod/Service 的变化（比如是不是新增了 Service、Pod）并将其反馈到本地的规则或者是用户态进程。

一个 LVS 版的 Service

我们来实际做一个 LVS 版的 Service。LVS 是一个专门用于负载均衡的内核机制。它工作在第四层，性能会比用 iptable 实现好一些。

假设我们是一个 Kube-proxy, 拿到了一个 Service 的配置, 如下图所示：它有一个 Cluster IP, 在该 IP 上的端口是 9376, 需要反馈到容器上的是 80 端口, 还有三个可工作的 Pod, 它们的 IP 分别是 10.1.2.3, 10.1.14.5, 10.1.3.8。



它要做的事情就是：

第1步，绑定VIP到本地（欺骗内核）

```
# ip route add to local 192.168.60.200/32 dev eth0proto kernel
```

第2步，为这个虚 IP 创建一个 IPVS 的 virtual server

```
# ipvsadm -A -t 192.168.60.200:9376 -s rr -p 600
```

第3步，为这个 IPVS service 创建相应的 real server

```
# ipvsadm -a -t 192.168.60.200:9376 -r 10.1.2.3:80 -m
# ipvsadm -a -t 192.168.60.200:9376 -r 10.1.4.5:80 -m
# ipvsadm -a -t 192.168.60.200:9376 -r 10.1.3.8:80 -m
```

第 1 步，绑定 VIP 到本地（欺骗内核）；
首先需要让内核相信它拥有这样的一个虚 IP，这是 LVS 的工作机制所决定的，因为它工作在第四层，并不关心 IP 转发，只有它认为这个 IP 是自己的才会拆到 TCP 或 UDP 这

一层。在第一步中，我们将该 IP 设到内核中，告诉内核它确实有这么一个 IP。实现的方法有很多，我们这里用的是 ip route 直接加 local 的方式，用 Dummy 哑设备上加 IP 的方式也是可以的。

第 2 步，为这个虚 IP 创建一个 IPVS 的 virtual server；
告诉它我需要为这个 IP 进行负载均衡分发，后面的参数就是一些分发策略等等。virtual server 的 IP 其实就是我们的 Cluster IP。

第 3 步，为这个 IPVS service 创建相应的 real server。
我们需要为 virtual server 配置相应的 real server，就是真正提供服务的后端是什么。比如说我们刚才看到有三个 Pod，于是就把这三个的 IP 配到 virtual server 上，完全一一对应过来就可以了。Kube-proxy 工作跟这个也是类似的。只是它还需要去监控一些 Pod 的变化，比如 Pod 的数量变成 5 个了，那么规则就应变成 5 条。如果这里面某一个 Pod 死掉了或者被杀死了，那么就要相应地减掉一条。又或者整个 Service 被撤销了，那么这些规则就要全部删掉。所以它其实做的是一些管理层面的工作。

啥？负载均衡还分内部外部

最后我们介绍一下 Service 的类型，可以分为以下 4 类。

ClusterIP

集群内部的一个虚拟 IP，这个 IP 会绑定到一堆服务的 Group Pod 上面，这也是默认的服务方式。它的缺点是这种方式只能在 Node 内部也就是集群内部使用。

NodePort

供集群外部调用。将 Service 承载在 Node 的静态端口上，端口号和 Service 一一对应，那么集群外的用户就可以通过 ：的方式调用到 Service。

LoadBalancer

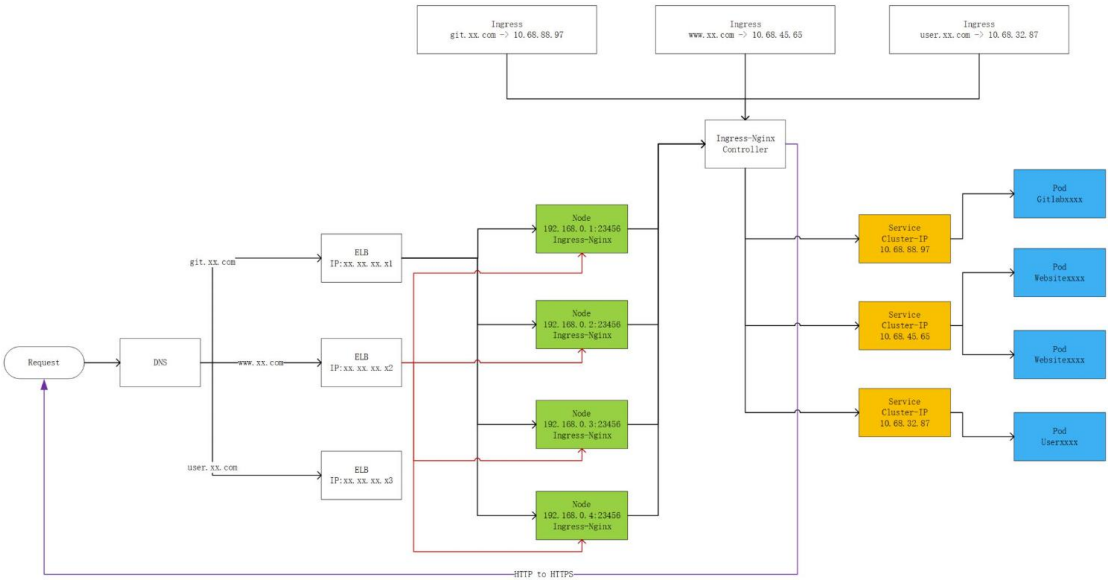
给云厂商的扩展接口。像阿里云、亚马逊这样的云厂商都是有成熟的 LB 机制的，这些机制可能是由一个很大的集群实现的，为了不浪费这种能力，云厂商可通过这个接口进行扩展。它首先会自动创建 NodePort 和 ClusterIP 这两种机制，云厂商可以选择直接将 LB 挂到这两种机制上，或者两种都不用，直接把 Pod 的 RIP 挂到云厂商的 ELB 的后端也是可以的。

ExternalName

摈弃内部机制，依赖外部设施，比如某个用户特别强，他觉得我们提供的都没什么用，就是要自己实现，此时一个 Service 会和一个域名一一对应起来，整个负载均衡的工作都是外

部实现的。

下图是一个实例。它灵活地应用了 ClusterIP、NodePort 等多种服务方式，又结合了云厂商的 ELB，变成了一个很灵活、极度伸缩、生产上真正可用的一套系统。



首先我们用 ClusterIP 来做功能 Pod 的服务入口。大家可以看到，如果有三种 Pod 的话，就有三个 Service Cluster IP 作为它们的服务入口。这些方式都是 Client 端的，如何在 Server 端做一些控制呢？

首先会起一些 Ingress 的 Pod（Ingress 是 K8s 后来新增的一种服务，本质上还是一堆同质的 Pod），然后将这些 Pod 组织起来，暴露到一个 NodePort 的 IP，K8s 的工作到此就结束了。

任何一个用户访问 23456 端口的 Pod 就会访问到 Ingress 的服务，它的后面有一个 Controller，会把 Service IP 和 Ingress 的后端进行管理，最后会调到 ClusterIP，再调到我们的功能 Pod。前面提到我们去对接云厂商的 ELB，我们可以让 ELB 去监听所有集群节点上的 23456 端口，只要在 23456 端口上有服务的，就认为有一个 Ingress 的实例在跑。

整个的流量经过外部域名的一个解析跟分流到达了云厂商的 ELB，ELB 经过负载均衡并通过 NodePort 的方式到达 Ingress，Ingress 再通过 ClusterIP 调用到后台真正的 Pod。这种系统看起来比较丰富，健壮性也比较好。任何一个环节都不存在单点的问题，任何一个环节也都有管理与反馈。

本文总结

本文的主要内容就到此为止了，这里为大家简单总结一下：

大家要从根本上理解 Kubernetes 网络模型的演化来历，理解 PerPodPerIP 的用心在哪里；

网络的事情万变不离其宗，按照模型从 4 层向下就是发包过程，反正层层剥离就是收包过程，容器网络也是如此；

Ingress 等机制是在更高的层次上（服务<->端口）方便大家部署集群对外服务，通过一个真正可用的部署实例，希望大家把 Ingress+Cluster IP + PodIP 等概念联合来看，理解社区出台新机制、新资源对象的思考。

Flannel 网络

Flannel 是 CoreOS 开源的 CNI 网络插件，下图 flannel 官网提供的一个数据包经过封包、传输以及拆包的示意图，从这个图片里面里面可以看出两台机器的 docker0 分别处于不同的段：10.1.20.1/24 和 10.1.15.1/24 ，如果从 Web App Frontend1 pod（10.1.15.2）去连接另一台主机上的 Backend Service2 pod（10.1.20.3），网络包从宿主机 192.168.0.100 发往 192.168.0.200，内层容器的数据包被封装到宿主机的 UDP 里面，并且在外层包装了宿主机的 IP 和 mac 地址。这就是一个经典的 overlay 网络，因为容器的 IP 是一个内部 IP，无法从跨宿主机通信，所以容器的网络互通，需要承载到宿主机的网络之上。

flannel 的支持多种网络模式，常用用都是 vxlan、UDP、hostgw、ipip 以及 gce 和阿里云等，vxlan 和 UDP 的区别是 vxlan 是内核封包，而 UDP 是 flanneld 用户态程序封包，所以 UDP 的方式性能会稍差；hostgw 模式是一种主机网关模式，容器到另外一个主机上容器的网关设置成所在主机的网卡地址，这个和 calico 非常相似，只不过 calico 是通过 BGP 声明，而 hostgw 是通过中心的 etcd 分发，所以 hostgw 是直连模式，不需要通过 overlay 封包和拆包，性能比较高，但 hostgw 模式最大的缺点是必须是在一个二层网络中，毕竟下一跳的路由需要在邻居表中，否则无法通行。

在实际的生产环境总，最常用的还是 vxlan 模式，我们先看工作原理，然后通过源码解析实现过程。

安装的过程非常简单，主要分为两步：

第一步安装 flannel，

yum install flannel 或者通过 kubernetes 的 daemonset 方式启动,配置 flannel 用的 etcd 地址

第二步是配置集群网络，

curl -L http://etcdurl:2379/v2/keys/flannel/network/config -XPUT -d value="{\"Network\":\"172.16.0.0/16\",\"SubnetLen\":24,\"Backend\":{\"Type\":\"vxlan\",\"VNI\":1}}"

然后启动每个节点的 flanned 程序。

工作原理:

1、容器的地址如何分配:

Docker 容器启动时候通过 docker0 分配 IP 地址，flannel 为每个机器分配一个 IP 段，配置在 docker0 上面，容器启动后就在本段内选择一个未占用的 IP，那么 flannel 如何修改 docker0 网段的呢?

先看一下 flannel 的启动文件 /usr/lib/systemd/system/flanneld.service

```
[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
ExecStart=/usr/bin/flanneld-start $FLANNEL_OPTIONS
ExecStartPost=/opt/flannel/mk-docker-opts.sh -k DOCKER_NETWORK_OPTIONS -d /run/flannel/docker
```

文件里面指定了 flannel 环境变量和启动脚本和启动后执行脚本 ExecStartPost 设置的 mk-docker-opts.sh，这个脚本的作用是生成/run/flannel/docker，文件内容如下:

```
DOCKER_OPT_BIP="--bip=10.251.81.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=false"
DOCKER_OPT_MTU="--mtu=1450"
DOCKER_NETWORK_OPTIONS=" --bip=10.251.81.1/24 --ip-masq=false --mtu=1450"
```

而这个文件又被 docker 启动文件/usr/lib/systemd/system/docker.service 所关联,

```
[Service]
Type=notify
NotifyAccess=all
EnvironmentFile=-/run/flannel/docker
EnvironmentFile=-/etc/sysconfig/docker
```

这样便可以设置 docker0 的网桥了。
在开发环境中，有三台机器，分别分配了如下网段:

```
host-139.245 10.254.44.1/24
host-139.246 10.254.60.1/24
host-139.247 10.254.50.1/24
```

2、容器如何通信

上面介绍了为每个容器分配 IP, 那么不同主机上面的容器如何通信呢, 我们用最常见的 vxlan 举例，这里有三个关键点，一个路由，一个 arp，一个 FDB。我们按照容器发包的过程，逐一分析一下上面三个元素的作用，首先容器出来的数据包会经过 docker0，那么下面是直接从主机网络出去，还是通过 vxlan 封包转发呢？这是每个机器上面路由设定的，

```
 #ip route    show dev flannel.1
10.254.50.0/24 via 10.254.50.0 onlink
10.254.60.0/24 via 10.254.60.0 onlink
```

可以看到每个主机上面都有到另外两台机器的路由，这个路由是 onlink 路由，onlink 参数表明强制此网关是“在链路上”的(虽然并没有链路层路由), 否则 linux 上面是没法添加不同网段的路由。这样数据包就能知道，如果是容器直接的访问则交给 flannel.1 设备处理。

flannel.1 这个虚拟网络设备将会对数据封包，但下面一个问题又来了，这个网关的 mac 地址是多少呢？因为这个网关是通过 onlink 设置的，flannel 会下发这个 mac 地址，查看一下 arp 表

```
# ip neig show dev flannel.1
10.254.50.0 lladdr ba:10:0e:7b:74:89 PERMANENT
10.254.60.0 lladdr 92:f3:c8:b2:6e:f0 PERMANENT
```

可以看到这个网关对应的 mac 地址，这样内层的数据包就封装好了

还是最后一个问题，外出的数据包的目的 IP 是多少呢？换句话说，这个封装后的数据包应该发往那一台机器呢？难不成每个数据包都广播。vxlan 默认实现第一次确实是通过广播的方式，但 flannel 再次采用一种 hack 方式直接下发了这个转发表 FDB

```
# bridge fdb show dev flannel.1
92:f3:c8:b2:6e:f0 dst 10.100.139.246 self permanent
ba:10:0e:7b:74:89 dst 10.100.139.247 self permanent
```

这样对应 mac 地址转发目标 IP 便可以获取到了。

这里还有个地方需要注意，无论是 arp 表还是 FDB 表都是 permanent，它表明写记录是手动维护的，传统的 arp 获取邻居的方式是通过广播获取，如果收到对端的 arp 相应则会标记对端为 reachable，在超过 reachable 设定时间后，如果发现对端失效会标记为 stale，之后会转入的 delay 以及 probe 进入探测的状态，如果探测失败会标记为 Failed 状态。之所以介绍 arp 的基础内容，是因为老版本的 flannel 并非使用我上面的方式，而是采用一种临时的 arp 方案，此时下发的 arp 表示 reachable 状态，这就意味着，如果在 flannel 宕机超过 reachable 超时时间的话，那么这台机器上面的容器的网络将会中断，我们简单回顾试一下之前(0.7.x)版本的做法, 容器为了为了能够获取到对端 arp 地址，内核会首先发送 arp 征询，如果尝试

```
/proc/sys/net/ipv4/neigh/$NIC/ucast_solicit
```

此时后会向用户空间发送 arp 征询

/proc/sys/net/ipv4/neigh/$NIC/app_solicit
之前版本的 flannel 正是利用这个特性，设定

cat /proc/sys/net/ipv4/neigh/flannel.1/app_solicit

从而 flanneld 便可以获取到内核发送到用户空间的 L3MISS,并且配合 etcd 返回这个 IP 地址对应的 mac 地址，设置为 reachable。从分析可以看出，如果 flanneld 程序如果退出后，容器之间的通信将会中断，这里需要注意。Flannel 的启动流程如下图所示：

Flannel 启动执行 newSubnetManager，通过他创建后台数据存储，当前有支持两种后端，默认是 etcd 存储, 如果 flannel 启动指定“kube-subnet-mgr”参数则使用 kubernetes 的接口存储数据。

具体代码如下：

```
func newSubnetManager() (subnet.Manager, error) {
    if opts.kubeSubnetMgr {
        return kube.NewSubnetManager(opts.kubeApiUrl, opts.kubeConfigFile)
    }
    cfg := &etcdv2.EtcdConfig{
        Endpoints: strings.Split(opts.etcdEndpoints, ","),
        Keyfile:    opts.etcdKeyfile,
        Certfile:   opts.etcdCertfile,
        CAFile:     opts.etcdCAFile,
        Prefix:     opts.etcdPrefix,
        Username:   opts.etcdUsername,
        Password:   opts.etcdPassword,
    }
    // Attempt to renew the lease for the subnet specified in the subnetFile
    prevSubnet := ReadCIDRFromSubnetFile(opts.subnetFile, "FLANNEL_SUBNET")
    return etcdv2.NewLocalManager(cfg, prevSubnet)
 }
```

通过 SubnetManager，结合上面介绍部署的时候配置的 etcd 的数据，可以获得网络配置信息，主要指 backend 和网段信息，如果是 vxlan，通过 NewManager 创建对应的网络管理器，这里用到简单工程模式，首先每种网络模式管理器都会通过 init 初始化注册，

如 vxlan

```
func init() {
    backend.Register("vxlan", New)
```

如果是 udp

```
func init() {
    backend.Register("udp", New)
 }
```

其它也是类似，将构建方法都注册到一个 map 里面，从而根据 etcd 配置的网络模式，设定启用对应的网络管理器。

第三步是注册网络

RegisterNetwork，首先会创建 flannel.vxlanID 的网卡，默认 vxlanID 是 1.然后就是向 etcd 注册租约并且获取相应的网段信息，这样有个细节，老版的 flannel 每次启动都是去获取新的网段，新版的 flannel 会遍历 etcd 里面已经注册的 etcd 信息，从而获取之前分配的网段，继续使用。

最后通过 WriteSubnetFile 写本地子网文件，

```
    # cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.254.0.0/16
FLANNEL_SUBNET=10.254.44.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=true
```

通过这个文件设定 docker 的网络。细心的读者可能发现这里的 MTU 并不是以太网规定的 1500，这是因为外层的 vxlan 封包还要占据 50 Byte。

当然 flannel 启动后还需要持续的 watch etcd 里面的数据，这是当有新的 flannel 节点加入，或者变更的时候，其他 flannel 节点能够动态更新的那三张表。主要的处理方法都在 handleSubnetEvents 里面

```
    func (nw *network) handleSubnetEvents(batch []subnet.Event) {
 . . .
        switch event.Type {//如果是有新的网段加入（新的主机加入）
        case subnet.EventAdded:
  . . .//更新路由表
if err := netlink.RouteReplace(&directRoute); err != nil {
    log.Errorf("Error adding route to %v via %v: %v", sn, attrs.PublicIP, err)
    continue
 }

//添加 arp 表
log.V(2).Infof("adding subnet: %s PublicIP: %s VtepMAC: %s", sn, attrs.PublicIP,
net.HardwareAddr(vxlanAttrs.VtepMAC))
                if err := nw.dev.AddARP(neighbor{IP: sn.IP, MAC:
net.HardwareAddr(vxlanAttrs.VtepMAC)}); err != nil {
                    log.Error("AddARP failed: ", err)
```

```
                    continue
                }
 //添加 FDB 表
                if err := nw.dev.AddFDB(neighbor{IP: attrs.PublicIP, MAC: net.HardwareAddr(vxlanAttrs.VtepMAC)}); err != nil {
                    log.Error("AddFDB failed: ", err)
                                    if err := nw.dev.DelARP(neighbor{IP: event.Lease.Subnet.IP, MAC: net.HardwareAddr(vxlanAttrs.VtepMAC)}); err != nil {
                        log.Error("DelARP failed: ", err)
                    }
                    continue
                }//如果是删除实践
        case subnet.EventRemoved:
//删除路由
                if err := netlink.RouteDel(&directRoute); err != nil {
                    log.Errorf("Error deleting route to %v via %v: %v", sn, attrs.PublicIP, err)
            } else {
                log.V(2).Infof("removing subnet: %s PublicIP: %s VtepMAC: %s", sn, attrs.PublicIP, net.HardwareAddr(vxlanAttrs.VtepMAC))

                //删除 arp                    if err := nw.dev.DelARP(neighbor{IP: sn.IP, MAC: net.HardwareAddr(vxlanAttrs.VtepMAC)}); err != nil {

                    log.Error("DelARP failed: ", err)
                }
 //删除 FDB
                if err := nw.dev.DelFDB(neighbor{IP: attrs.PublicIP, MAC: net.HardwareAddr(vxlanAttrs.VtepMAC)}); err != nil {
                    log.Error("DelFDB failed: ", err)
                }
                if err := netlink.RouteDel(&vxlanRoute); err != nil {
                    log.Errorf("failed to delete vxlanRoute (%s -> %s): %v", vxlanRoute.Dst, vxlanRoute.Gw, err)
                }
            }
        default:
            log.Error("internal error: unknown event type: ", int(event.Type))
        }
    }
 }
```

这样 flannel 里面任何主机的添加和删除都可以被其它节点所感知到，从而更新本地内核转发表。

作者：陈晓宇
原文：http://college.creditease.cn/#/detail/15/176

calico 网络

calico 是一个安全的 L3 网络和网络策略提供者。
calico 使用 bgp 的原因：why bgp not ospf
有关 BGP rr 的介绍

安装方式

标准托管安装 (ETCD 存储)

需要提前安装 etcd 集群

```
# 创建 calico 连接 etcd 的 secret
kubectl create secret generic calico-etcd-secrets \
--from-file=etcd-key=/etc/kubernetes/ssl/kubernetes-key.pem \
--from-file=etcd-cert=/etc/kubernetes/ssl/kubernetes.pem \
--from-file=etcd-ca=/etc/kubernetes/ssl/ca.pem

# 部署
kubectl create -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/calico.
yaml

# rbac
kubectl apply -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/rbac.yaml
```

kubeadm 托管部署

依赖

- k8s1.7+
- 没有其他 cni 插件(华为开源的 CNI-Genie 可以同时运行多个 CNI)
- –pod-network-cidr 参数需要和 calico ip pool 保持一致
- –service-cidr 不能和 calico ip pool 重叠

部署

kubectl apply -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/kubeadm/1.7/calico.yaml

Kubernetes 数据存储托管安装(不需要 etcd)

依赖

- 暂时不支持 ipam,推荐使用 host-local ipam 与 pod cidr 结合使用
- 默认使用 node-to-node mesh 模式
- k8s1.7+
- 配置使用 CNI
- controller-manager 配置 cluster-cidr

部署

rbac
kubectl create -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/rbac-kdd.yaml

部署

kubectl create -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml

仅使用网络策略

kubectl create -f
https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/kubernetes-datastore/policy-only/1.7/calico.yaml

canal

canal 旨在让用户能够轻松地将 Calico 和 flannel 网络作为一个统一的网络解决方案进行部署.

kubectl apply -f
https://raw.githubusercontent.com/projectcalico/canal/master/k8s-install/1.7/rbac.yaml
kubectl apply -f

https://raw.githubusercontent.com/projectcalico/canal/master/k8s-install/1.7/canal.yaml

配置

环境设置

```
# etcd 数据存储
export ETCD_ENDPOINTS=http://xxx:2379
# k8s 数据存储
export DATASTORE_TYPE=kubernetes KUBECONFIG=~/.kube/config
```

typha 模式

k8s 数据存储模式超过 50 各节点推荐启用 typha,Typha 组件可以帮助 Calico 扩展到大量的节点，而不会对 Kubernetes API 服务器造成过度的影响。
修改 typha_service_name "none"改为"calico-typha"。

禁用 snat

```
calicoctl get ipPool -o yaml | sed 's/natOutgoing: true/natOutgoing: false/g' | calicoctl apply -f -
```

关闭 node-to-node mesh (节点网络全互联)

```
cat << EOF
apiVersion: projectcalico.org/v3
kind: BGPConfiguration
metadata:
  name: default
spec:
  logSeverityScreen: Info
  nodeToNodeMeshEnabled: false
  asNumber: 64512
EOF | calicoctl apply -f -

calicoctl node status
```

创建 IP Pool

```
calicoctl get ippool default-ipv4-ippool -o yaml
```

配置 bird 服务

```
yum install bird
service bird start
```

```
cat >> /etc/bird.conf < EOF
log syslog { debug, trace, info, remote, warning, error, auth, fatal, bug };
log stderr all;

# Override router ID
router id 172.26.6.1;


filter import_kernel {
if ( net != 0.0.0.0/0 ) then {
     accept;
     }
reject;
}

# Turn on global debugging of all protocols
debug protocols all;

# This pseudo-protocol watches all interface up/down events.
protocol device {
   scan time 2;       # Scan interfaces every 2 seconds
}

protocol bgp {
   description "172.26.6.2";
   local as 64512;
   neighbor 172.26.6.2 as 64512;
   multihop;
   rr client;
   graceful restart;
   import all;
   export all;
}
protocol bgp {
   description "172.26.6.3";
   local as 64512;
   neighbor 172.26.6.3 as 64512;
   multihop;
   rr client;
   graceful restart;
   import all;
   export all;
}
```

EOF

IP-IN-IP

calicoctl get ippool default-ipv4-ippool -o yaml > pool.yaml
修改 Off/Always/CrossSubnet
calicoctl apply -f pool.yaml
例：
所有工作负载

```
$ calicoctl apply -f - << EOF
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: ippool-ipip-1
spec:
  cidr: 192.168.0.0/16
  ipipMode: Always
  natOutgoing: true
EOF

# CrossSubnet

$ calicoctl apply -f - << EOF
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: ippool-cs-1
spec:
  cidr: 192.168.0.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
EOF
```

#通过修改配置文件环境变量
CALICO_IPV4POOL_IPIP 参数值 Off, Always, CrossSubnet
如果您的网络结构执行源/目标地址检查，并在未识别这些地址时丢弃流量，则可能需要启用工作负载间流量的 IP-in-IP 封装

bgp peer

查看状态
calicoctl node status

配置全局 bgp peer(rr)

```
cat << EOF | calicoctl create -f -
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-3040
spec:
  peerIP: 172.26.6.1
  asNumber: 64567
EOF

# 删除
$ calicoctl delete bgpPeer 172.26.6.1
特定 BGP peer

$ cat << EOF | calicoctl create -f -
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-node-aabbff
spec:
  peerIP: aa:bb::ff
  node: node1
  asNumber: 64514
EOF

calicoctl delete bgpPeer aa:bb::ff --scope=node --node=node1
calicoctl get bgpPeer
```

原文：
https://rocdu.io/2018/01/k8s%E4%BD%BF%E7%94%A8calico%E7%BD%91%E7%BB%9C/

Volume

默认情况下容器中的磁盘文件是非持久化的，对于运行在容器中的应用来说面临两个问题，第一：当容器挂掉 kubelet 将重启启动它时，文件将会丢失；第二：当 Pod 中同时运行多个容器，容器之间需要共享文件时。Kubernetes 的 Volume 解决了这两个问题。

建议熟悉 Pod。

背景

在 Docker 中也有一个 docker Volume 的概念 ，Docker 的 Volume 只是磁盘中的一个目录，

生命周期不受管理。当然 Docker 现在也提供 Volume 将数据持久化存储，但支持功能比较少（例如，对于 Docker 1.7，每个容器只允许挂载一个 Volume，并且不能将参数传递给 Volume）。

另一方面，Kubernetes Volume 具有明确的生命周期 - 与 pod 相同。因此，Volume 的生命周期比 Pod 中运行的任何容器要持久，在容器重新启动时能可以保留数据，当然，当 Pod 被删除不存在时，Volume 也将消失。注意，Kubernetes 支持许多类型的 Volume，Pod 可以同时使用任意类型/数量的 Volume。

内部实现中，一个 Volume 只是一个目录，目录中可能有一些数据，pod 的容器可以访问这些数据。至于这个目录是如何产生的、支持它的介质、其中的数据内容是什么，这些都由使用的特定 Volume 类型来决定。

要使用 Volume，pod 需要指定 Volume 的类型和内容（spec.volumes 字段），和映射到容器的位置（spec.containers.volumeMounts 字段）。

Volume 类型

Kubernetes 支持 Volume 类型有：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- fc (fibre channel)
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo
- secret
- persistentVolumeClaim
- downwardAPI
- projected
- azureFileVolume
- azureDisk
- vsphereVolume
- Quobyte
- PortworxVolume
- ScaleIO
- StorageOS

- local

emptyDir

使用 emptyDir，当 Pod 分配到 Node 上时，将会创建 emptyDir，并且只要 Node 上的 Pod 一直运行，Volume 就会一直存。当 Pod（不管任何原因）从 Node 上被删除时，emptyDir 也同时会删除，存储的数据也将永久删除。注：删除容器不影响 emptyDir。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath

hostPath 允许挂载 Node 上的文件系统到 Pod 里面去。如果 Pod 需要使用 Node 上的文件，可以使用 hostPath。

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
```

```
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
```

gcePersistentDisk

gcePersistentDisk 可以挂载 GCE 上的永久磁盘到容器，需要 Kubernetes 运行在 GCE 的 VM 中。与 emptyDir 不同，Pod 删除时，gcePersistentDisk 被删除，但 Persistent Disk 的内容任然存在。这就意味着 gcePersistentDisk 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

提示：使用 gcePersistentDisk，必须用 gcloud 或使用 GCE API 或 UI 创建 PD

创建 PD

使用 GCE PD 与 pod 之前，需要创建它

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

awsElasticBlockStore

awsElasticBlockStore 可以挂载 AWS 上的 EBS 盘到容器，需要 Kubernetes 运行在 AWS 的 EC2 上。与 emptyDir Pod 被删除情况不同，Volume 仅被卸载，内容将被保留。这就意味着 awsElasticBlockStore 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

提示：必须使用 aws ec2 create-volumeAWS API 创建 EBS Volume，然后才能使用。

创建 EBS Volume

在使用 EBS Volume 与 pod 之前，需要创建它。

aws ec2 create-volume --availability-zone eu-west-1a --size 10 --volume-type gp2
AWS EBS 配置示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

NFS

NFS 是 Network File System 的缩写，即网络文件系统。Kubernetes 中通过简单地配置就可以挂载 NFS 到 Pod 中，而 NFS 中的数据是可以永久保存的，同时 NFS 支持同时写操作。Pod 被删除时，Volume 被卸载，内容被保留。这就意味着 NFS 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间相互传递。

详细信息，请参阅 NFS 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/nfs

iSCSI

iscsi 允许将现有的 iscsi 磁盘挂载到我们的 pod 中，和 emptyDir 不同的是，删除 Pod 时会被删除，但 Volume 只是被卸载，内容被保留，这就意味着 iscsi 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

详细信息，请参阅 iSCSI 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/iscsi

flocker

Flocker 是一个开源的容器集群数据卷管理器。它提供各种存储后端支持的数据卷的管理和编排。

详细信息，请参阅 Flocker 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/flocker

glusterfs

glusterfs，允许将 Glusterfs（一个开源网络文件系统）Volume 安装到 pod 中。不同于 emptyDir，Pod 被删除时，Volume 只是被卸载，内容被保留。味着 glusterfs 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

详细信息，请参阅 GlusterFS 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/glusterfs

RBD

RBD 允许 Rados Block Device 格式的磁盘挂载到 Pod 中，同样的，当 pod 被删除的时候，rbd 也仅仅是被卸载，内容保留，rbd 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

详细信息，请参阅 RBD 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/rbd

cephfs

cephfs Volume 可以将已经存在的 CephFS Volume 挂载到 pod 中，与 emptyDir 特点不同，pod 被删除的时，cephfs 仅被被卸载，内容保留。cephfs 能够允许我们提前对数据进行处理，而且这些数据可以在 Pod 之间“切换”。

提示：可以使用自己的 Ceph 服务器运行导出，然后在使用 cephfs。

详细信息，请参阅 CephFS 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/cephfs/

gitRepo

gitRepo volume 将 git 代码下拉到指定的容器路径中。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /mypath
      name: git-volume
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

secret

secret volume 用于将敏感信息（如密码）传递给 pod。可以将 secrets 存储在 Kubernetes API 中，使用的时候以文件的形式挂载到 pod 中，而不用连接 api。 secret volume 由 tmpfs（RAM 支持的文件系统）支持。

详细了解 secret
https://kubernetes.io/docs/user-guide/secrets

persistentVolumeClaim

persistentVolumeClaim 用来挂载持久化磁盘的。PersistentVolumes 是用户在不知道特定云环境的细节的情况下，实现持久化存储（如 GCE PersistentDisk 或 iSCSI 卷）的一种方式。

更多详细信息，请参阅 PersistentVolumes 示例。

downwardAPI

通过环境变量的方式告诉容器 Pod 的信息

更多详细信息，请参见 downwardAPI 卷示例。

projected

Projected volume 将多个 Volume 源映射到同一个目录

目前，可以支持以下类型的卷源：

- secret
- downwardAPI
- configMap

所有卷源都要求与 pod 在同一命名空间中。更详细信息，请参阅 all-in-one volume design document。

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
```

```
      projected:
        sources:
        - secret:
            name: mysecret
            items:
              - key: username
                path: my-group/my-username
        - downwardAPI:
            items:
              - path: "labels"
                fieldRef:
                  fieldPath: metadata.labels
              - path: "cpu_limit"
                resourceFieldRef:
                  containerName: container-test
                  resource: limits.cpu
        - configMap:
            name: myconfigmap
            items:
              - key: config
                path: my-group/my-config
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
```

```
      items:
        - key: password
          path: my-group/my-password
          mode: 511
```

FlexVolume

alpha 功能

更多细节在这里
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/flexvolume/README.md

AzureFileVolume

AzureFileVolume 用于将 Microsoft Azure 文件卷（SMB 2.1 和 3.0）挂载到 Pod 中。

更多细节在这里
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/azure_file/README.md

AzureDiskVolume

Azure 是微软提供的公有云服务，如果使用 Azure 上面的虚拟机来作为 Kubernetes 集群使用时，那么可以通过 AzureDisk 这种类型的卷插件来挂载 Azure 提供的数据磁盘。

更多细节在这里
https://www.kubernetes.org.cn/198.html

vsphereVolume

需要条件：配置了 vSphere Cloud Provider 的 Kubernetes。有关 cloudprovider 配置，请参阅 vSphere 入门指南。
https://kubernetes.io/docs/getting-started-guides/vsphere/

vsphereVolume 用于将 vSphere VMDK Volume 挂载到 Pod 中。卸载卷后，内容将被保留。它同时支持 VMFS 和 VSAN 数据存储。

重要提示：使用 POD 之前，必须使用以下方法创建 VMDK。

创建一个 VMDK 卷

使用 vmkfstools 创建。先将 ssh 接入 ESX，然后使用以下命令创建 vmdk
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk

使用 vmware-vdiskmanager 创建
shell vmware-vdiskmanager -c -t 0 -s 40GB -a lsilogic myDisk.vmdk

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-vmdk
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-vmdk
      name: test-volume
  volumes:
  - name: test-volume
    # This VMDK volume must already exist.
    vsphereVolume:
      volumePath: "[DatastoreName] volumes/myDisk"
      fsType: ext4
```

更多例子在这里。
https://git.k8s.io/kubernetes/examples/volumes/vsphere

Quobyte

在 kubernetes 中使用 Quobyte 存储，需要提前部署 Quobyte 软件，要求必须是 1.3 以及更高版本，并且在 kubernetes 管理的节点上面部署 Quobyte 客户端。

详细信息，请参阅这里。
https://www.kubernetes.org.cn/198.html

PortworxVolume

Portworx 能把你的服务器容量进行蓄积（pool），将你的服务器或者云实例变成一个聚合的高可用的计算和存储节点。

PortworxVolume 可以通过 Kubernetes 动态创建, 也可以在 Kubernetes pod 中预先配置和引用。示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /mnt
      name: pxvol
  volumes:
  - name: pxvol
    # This Portworx volume must already exist.
    portworxVolume:
      volumeID: "pxvol"
      fsType: "<fs-type>"
```

更多细节和例子可以在这里找到
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/portworx/README.md

ScaleIO

ScaleIO 是一种基于软件的存储平台（虚拟 SAN），可以使用现有硬件来创建可扩展共享块网络存储的集群。ScaleIO 卷插件允许部署的 pod 访问现有的 ScaleIO 卷（或者可以为持久卷声明动态配置新卷，请参阅 Scaleio Persistent Volumes）。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-0
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: pod-0
```

```
    volumeMounts:
    - mountPath: /test-pd
      name: vol-0
  volumes:
  - name: vol-0
    scaleIO:
      gateway: https://localhost:443/api
      system: scaleio
      volumeName: vol-0
      secretRef:
        name: sio-secret
      fsType: xfs
```

详细信息，请参阅 ScaleIO 示例。
https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/scaleio

StorageOS

StorageOS 是一家英国的初创公司，给无状态容器提供简单的自动块存储、状态来运行数据库和其他需要企业级存储功能，但避免随之而来的复杂性、刚性以及成本。

核心：是 StorageOS 向容器提供块存储，可通过文件系统访问。

StorageOS 容器需要 64 位 Linux，没有额外的依赖关系，提供免费开发许可证。

安装说明，请参阅 StorageOS 文档
https://docs.storageos.com/

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: redis
    role: master
  name: test-storageos-redis
spec:
  containers:
    - name: master
      image: kubernetes/redis:v1
      env:
        - name: MASTER
          value: "true"
      ports:
        - containerPort: 6379
```

```
        volumeMounts:
          - mountPath: /redis-master-data
            name: redis-data
  volumes:
    - name: redis-data
      storageos:
        # The `redis-vol01` volume must already exist within StorageOS in the `default`
namespace.
        volumeName: redis-vol01
        fsType: ext4
```

有关动态配置和持久卷声明的更多信息，请参阅 StorageOS 示例。

Local

目前处于 Kubernetes 1.7 中的 alpha 级别。

Local 是 Kubernetes 集群中每个节点的本地存储 (如磁盘, 分区或目录) , 在 Kubernetes1.7 中 kubelet 可以支持对 kube-reserved 和 system-reserved 指定本地存储资源。

通过上面的这个新特性可以看出来，Local Storage 同 HostPath 的区别在于对 Pod 的调度上，使用 Local Storage 可以由 Kubernetes 自动的对 Pod 进行调度，而是用 HostPath 只能人工手动调度 Pod，因为 Kubernetes 已经知道了每个节点上 kube-reserved 和 system-reserved 设置的本地存储限制。

示例：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
  annotations:
        "volume.alpha.kubernetes.io/node-affinity": '{
            "requiredDuringSchedulingIgnoredDuringExecution": {
                "nodeSelectorTerms": [
                    { "matchExpressions": [
                        { "key": "kubernetes.io/hostname",
                          "operator": "In",
                          "values": ["example-node"]
                        }
                    ]}
                ]}
            }'
spec:
```

```
    capacity:
      storage: 100Gi
    accessModes:
    - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: local-storage
    local:
      path: /mnt/disks/ssd1
```

请注意，本地 PersistentVolume 需要手动清理和删除。

有关 local 卷类型的详细信息，请参阅 Local Persistent Storage user guide

Using subPath

有时，可以在一个 pod 中，将同一个卷共享，使其有多个用处。volumeMounts.subPath 特性可以用来指定卷中的一个子目录，而不是直接使用卷的根目录。

以下是使用单个共享卷的 LAMP 堆栈（Linux Apache Mysql PHP）的 pod 的示例。HTML 内容映射到其 html 文件夹，数据库将存储在 mysql 文件夹中：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

Resources

emptyDir Volume 的存储介质（Disk，SSD 等）取决于 kubelet 根目录（如/var/lib/kubelet）所处文件系统的存储介质。不限制 emptyDir 或 hostPath Volume 使用的空间大小，不对容器或 Pod 的资源隔离。
参考：https://feisky.gitbooks.io/kubernetes/concepts/volume.html

Secret

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

Secret 有三种类型：

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

Opaque Secret

Opaque 类型的数据是一个 map 类型，要求 value 是 base64 编码格式：

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
```

接着，就可以创建 secret 了：kubectl create -f secrets.yml。

创建好 secret 之后，有两种方式来使用它：

- 以 Volume 方式
- 以环境变量方式

将 Secret 挂载到 Volume 中

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: db
  name: db
spec:
  volumes:
  - name: secrets
    secret:
      secretName: mysecret
  containers:
  - image: gcr.io/my_project_id/pg:v1
    name: db
    volumeMounts:
    - name: secrets
      mountPath: "/etc/secrets"
      readOnly: true
    ports:
    - name: cp
      containerPort: 5432
      hostPort: 5432
```

将 Secret 导出到环境变量中

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 2
  strategy:
      type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
```

```
        visualize: "true"
    spec:
      containers:
      - name: "wordpress"
        image: "wordpress"
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

kubernetes.io/dockerconfigjson

可以直接用 kubectl 命令来创建用于 docker registry 认证的 secret：

```
$ kubectl create secret docker-registry myregistrykey
--docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER
--docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
secret "myregistrykey" created.
```

也可以直接读取~/.docker/config.json 的内容来创建：

```
$ cat ~/.docker/config.json | base64
$ cat > myregistrykey.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockerconfigjson:
UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWFhYWFhYWFhYWFhYWFhYWFhYWFhYWxsbGxsbGxsbGxsbGxsbGxsbGxsbGxsbGxsbGx5eXl5eXl5eXl5eXl5eXl5eSBsbGxsbGxsbGxsbG9vb29vb29vb29vb29vb29vb29vb29vb25ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cgYXV0aCBrZXlzCg==
type: kubernetes.io/dockerconfigjson
EOF
$ kubectl create -f myregistrykey.yaml
```

在创建 Pod 的时候，通过 imagePullSecrets 来引用刚创建的 myregistrykey:

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

Service Account

Service Account 用来访问 Kubernetes API，由 Kubernetes 自动创建，并且会自动挂载到 Pod 的/run/secrets/kubernetes.io/serviceaccount 目录中。

```
$ kubectl run nginx --image nginx
deployment "nginx" created
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3137573019-md1u2   1/1       Running   0          13s
$ kubectl exec nginx-3137573019-md1u2 ls /run/secrets/kubernetes.io/serviceaccount
ca.crt
namespace
token
```

参考：https://jimmysong.io/kubernetes-handbook/concepts/secret.html

Kubernetes 安装

1.1 常见部署方式

- 源码部署
  从 github 上拉取 kubernetes 组件的源码，自行编译，生成证书，配置启动参数，安装。
  优点：能快速熟悉 kubernetes 架构和组件构成和组件与组件之前的耦合关系。
  缺点：部署起来繁琐、复杂，容易出错。
- rke 部署
  RKE (Rancher Kubernetes Engine)是 RancherLabs 提供的一个工具，可以在裸机、虚拟机、公私有云上快速安装 Kubernetes 集群。整个集群的部署只需要一个命令、一个

配置文件，解决了如何轻松部署 Kubernetes 的问题。
优点：使用起来方便快捷，一条命令，一个配置文件，并且能自动给 kubernetes 组件做 HA。
缺点：全有组件都做成了 docker image，部署后想定制化修改参数，需要重新改 yaml 文件。

- kubeadm 部署 kubeadm 为 kubernetes 官方推荐的自动化部署工具, 他将 kubernetes 的组件以 pod 的形式部署在 master 和 node 节点上，并自动完成证书认证等操作。因为 kubeadm 默认要从 google 的镜像仓库下载镜像，但目前国内无法访问 google 镜像仓库。需要提前将镜像下好，然后导入。
  优点：kubernetes 社区推荐部署工具，能紧根社区版本，部署起来方便快捷。
  缺点：镜像在 google 镜像仓库，国内使用非常麻烦，没有自动的 HA 需要手动配置。

建议：新手入门先用自动化部署工具 kuebadm 或 rke 先部署一个 kubernetes 出来，熟悉下基本架构然后在在源码部署方式多部署几遍，熟悉组件与组件之间耦合，出问题后更好的 trouble shooting。

使用 Kubeadm 部署 Kubernetes 集群

机器准备

配置	数量	操作系统
4c+8G	x3	centos7.6

1.2 部署过程

1.2.1 安装 docker

kubernetes 1.11 支持的 docker 版本如下
1.13.1, 17.03, 17.06, 17.09, 18.06, 18.09.
这里我们安装 docker 18.09.9-3

安装依赖
yum install -y yum-utils device-mapper-persistent-data lvm2

安装依赖

yum install -y yum-utils device-mapper-persistent-data lvm2 conntrack ebtables socat

导入 docker 官方 repo

yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo

安装指定版本 docker 列出软件版本

yum list docker-ce.x86_64 --showduplicates |sort -r

安装 docker-engine

yum install -y docker-ce-18.09.9-3.el7

启动 docker

systemctl enable docker && systemctl start docker

配置 docker 加速器

tee /etc/docker/daemon.json << EOF
{
"registry-mirrors": ["https://vqgjby9l.mirror.aliyuncs.com"]
}EOF
sudo systemctl daemon-reload sudo systemctl restart docker

1.2.2 使用 kubeadm 部署 kubernetes 集群

关闭 swap

swapoff -a

永久关闭
将 fstab 内 swap 行注释掉
规范主机名
标准 FQDN 格式

hostname xxx-xxx-xxx
vim /etc/hostname

每台主机配置 hosts

vim /etc/hosts172.31.48.86 k8s-master172.31.48.87 k8s-worker-01172.31.48.88
k8s-worker-02

解压离线包

```
tar -xvf   k8s_v1.15_offine_centos.tar.bz
```

配置内核参数

```
echo "net.bridge.bridge-nf-call-ip6tables = 1net.bridge.bridge-nf-call-iptables = 1
" >> /etc/sysctl.conf
sysctl -p
```

导入镜像

```
docker load --input k8s_v1.15_image.tar.gz
```

安装软件包

```
cd rpms/
rpm -ivh rpms/kubeadm-1.15.5-0.x86_64.rpm kubelet-1.15.5-0.x86_64.rpm
cri-tools-1.13.0-0.x86_64.rpm kubectl-1.15.5-0.x86_64.rpm
kubernetes-cni-0.7.5-0.x86_64.rpm
```

master 节点操作
启动 kubelet

```
systemctl enable kubelet && sudo systemctl start kubelet
```

初始化 master 节点

```
kubeadm init --kubernetes-version=v1.15.5 --pod-network-cidr=10.244.0.0/16
```

kubernetes 默认支持多重网络插件如 flannel、weave、calico，这里使用 flanne，就必须要设置–pod-network-cidr 参数，10.244.0.0/16 是 kube-flannel.yml 里面配置的默认网段，如果需要修改的话, 需要把 kubeadm init 的–pod-network-cidr 参数和后面的 kube-flannel.yml 里面修改成一样的网段就可以了。

将 kubeadm join xxx 保存下来，等下 node 节点需要使用 如果忘记了，可以在 master 上通过 kubeadmin token list 得到

部署 dashboard

```
kubectl apply -f dashboar/kubernetes-dashboard.yaml
```

获取连接的 token 生成 token

kubectl apply -f dashboar/admin-role.yaml

获取 secret 名

kubectl -n kube-system get secret|grep admin-token
admin-token-dw2zb
kubernetes.io/service-account-token 3 6m17s

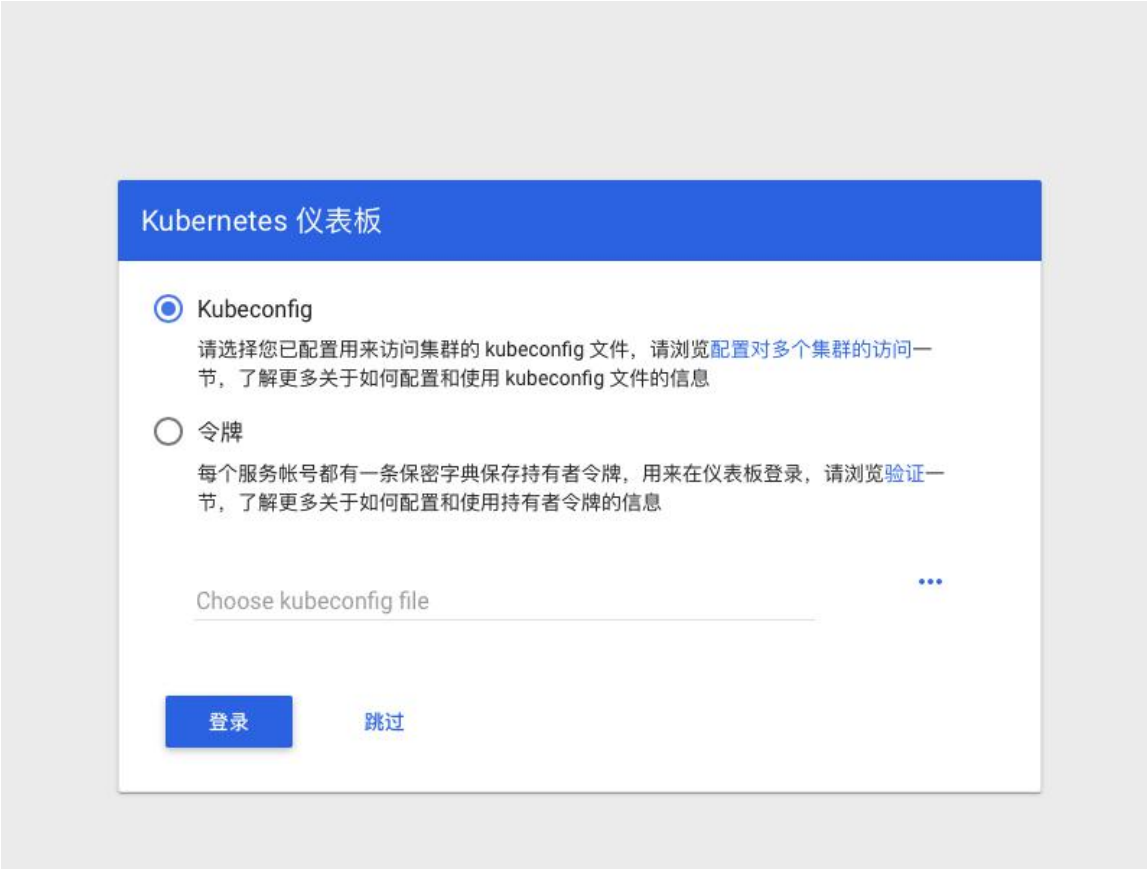
获取 token

kubectl -n kube-system describe secret admin-token-xxx

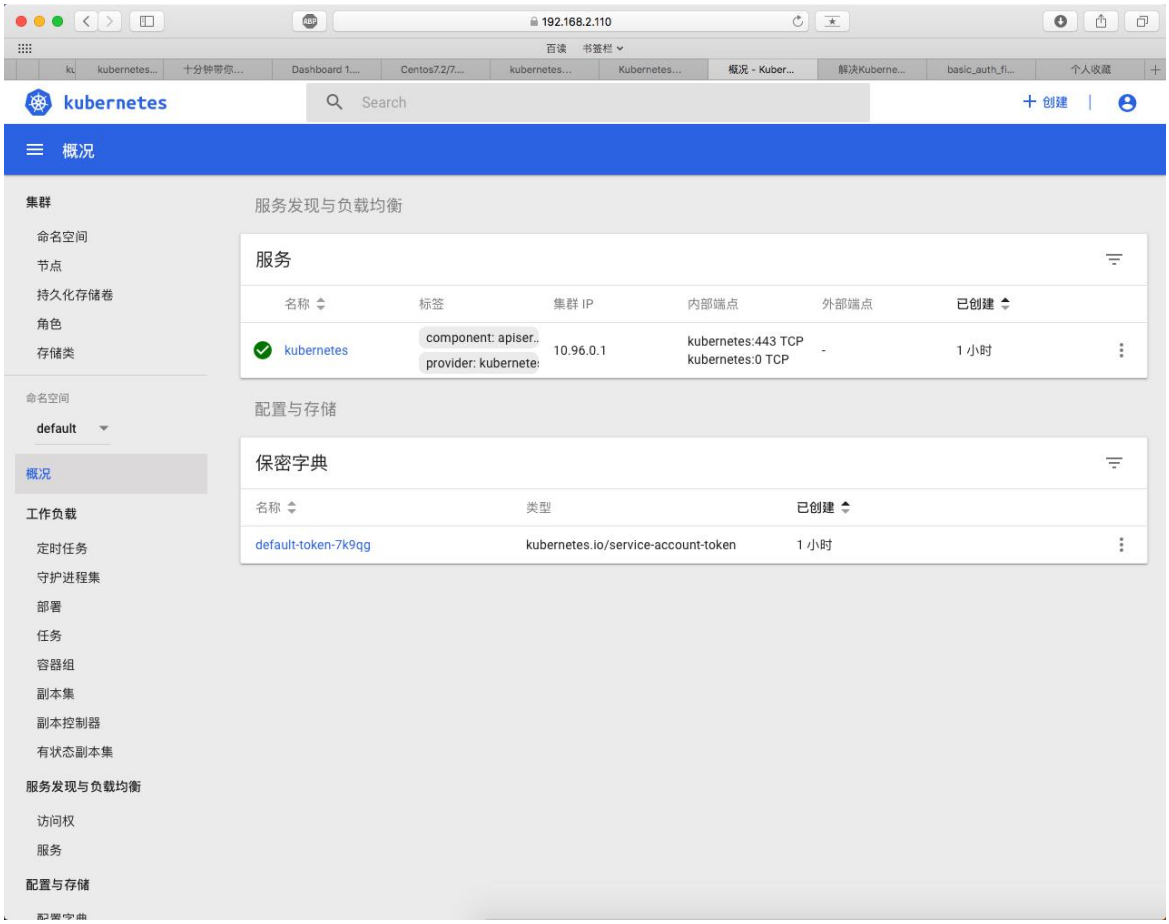
Name: admin-token-dw2zb
Namespace: kube-system
Labels: <none>
Annotations: kubernetes.io/service-account.name: admin
kubernetes.io/service-account.uid:
2f49cd71-afac-4d94-988b-1351d733b312
Type: kubernetes.io/service-account-token
Data
====
ca.crt: 1025 bytes
namespace: 11 bytes
token: xxxx

访问 dashboard

https://master_ip:30443


Kubernetes 仪表板
Kubeconfig
请选择您已配置用来访问集群的 kubeconfig 文件，请浏览配置对多个集群的访问一节，了解更多关于如何配置和使用 kubeconfig 文件的信息
令牌
每个服务帐号都有一条保密字典保存持有者令牌，用来在仪表板登录，请浏览验证一节，了解更多关于如何配置和使用持有者令牌的信息
Choose kubeconfig file
登录
跳过

输入 token



节入 worker 节点
在其他 slave 节点执行

kubeadm join　xxx

拷贝 kubeconfig 文件用于操作 kubernetes 集群

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

安装网络，可以使用 flannel、calico、weave、macvlan 这里我们用 flannel。

kubectl apply -f kube-flannel.yml

检查集群状态
查看组件是否都正常运行

kubectl get pod -n kube-system
NAME READY STATUS RESTARTS AGE
coredns-78fcdf6894-22gd8 1/1 Running 0 6m

```
coredns-78fcdf6894-5q7l4                  1/1     Running   0          6m
etcd-rke-node5                            1/1     Running   0          5m
kube-apiserver-rke-node5                  1/1     Running   0          5m
kube-controller-manager-rke-node5         1/1     Running   0          5m
kube-flannel-ds-mwhv4                     1/1     Running   0          24s
kube-proxy-krhjk                          1/1     Running   0          6m
kube-scheduler-rke-node5                  1/1     Running   0          5m
```

查看节点是否 redy

```
kubectl get node
NAME            STATUS   ROLES    AGE     VERSION
k8s-master      Ready    master   1d      v1.15.5
k8s-worker-01   Ready    <none>   1d      v1.15.5
k8s-worker-02   Ready    <none>   1d      v1.15.5
```