云计算实验二：k8s 实验指南

姜婧妍、蔡元哲 2022-10-22

K8s简介

Kubernetes 是一个可移植、可扩展的开源平台，用于管理容器化的工作负载和服务，可促进声明式配置和自动化。 Kubernetes 拥有一个庞大且快速增长的生态，其服务、支持和工具的使用范围相当广泛。

Kubernetes 这个名字源于希腊语，意为“舵手”或“飞行员”。k8s 这个缩写是因为 k 和 s 之间有八个字符的关系。 Google 在 2014 年开源了 Kubernetes 项目。 Kubernetes 建立在 Google 大规模运行生产工作负载十几年经验的基础上， 结合了社区中最优秀的想法和实践。

为什么需要 Kubernetes，它能做什么?

容器是打包和运行应用程序的好方式。在生产环境中， 你需要管理运行着应用程序的容器，并确保服务不会下线。 例如，如果一个容器发生故障，则你需要启动另一个容器。 如果此行为交由给系统处理，是不是会更容易一些?

这就是 Kubernetes 要来做的事情！ Kubernetes 为你提供了一个可弹性运行分布式系统的框架。 Kubernetes 会满足你的扩展要求、故障转移你的应用、提供部署模式等。 例如，Kubernetes 可以轻松管理系统的 Canary 部署。

Kubernetes 为你提供：

- **服务发现和负载均衡**

  Kubernetes 可以使用 DNS 名称或自己的 IP 地址来曝露容器。 如果进入容器的流量很大， Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。

- **存储编排**

  Kubernetes 允许你自动挂载你选择的存储系统，例如本地存储、公共云提供商等。

- **自动部署和回滚**

  你可以使用 Kubernetes 描述已部署容器的所需状态， 它可以以受控的速率将实际状态更改为

期望状态。 例如，你可以自动化 Kubernetes 来为你的部署创建新容器， 删除现有容器并将它们的所有资源用于新容器。

- **自动完成装箱计算**

  你为 Kubernetes 提供许多节点组成的集群，在这个集群上运行容器化的任务。 你告诉 Kubernetes 每个容器需要多少 CPU 和内存 (RAM)。 Kubernetes 可以将这些容器按实际情况调度到你的节点上，以最佳方式利用你的资源。

- **自我修复**

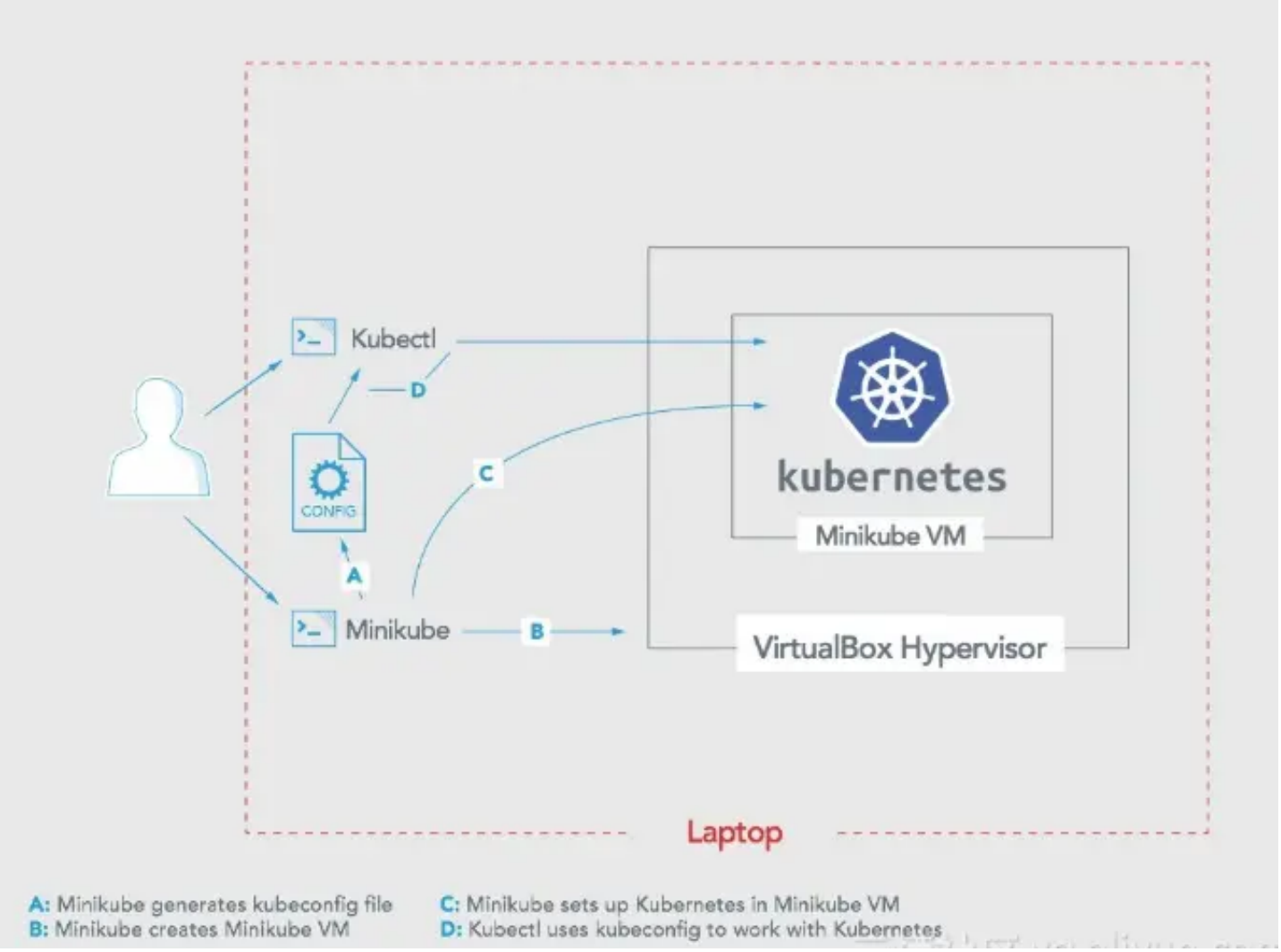
  Kubernetes 将重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器， 并且在准备好服务之前不将其通告给客户端。

- **密钥与配置管理**

  Kubernetes 允许你存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。 你可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

minikube 介绍

官网：https://minikube.sigs.k8s.io/docs/
Minikube是由Kubernetes社区维护的单机版的Kubernetes集群，支持macOS, Linux, and Windows等多种操作系统平台，使用最新的官方stable版本，并支持Kubernetes的大部分功能，从基础的容器编排管理，到高级特性如负载均衡、Ingress，权限控制等。非常适合作为Kubernetes入门，或开发测试环境使用。



搭建部署单机 Kubernetes

在上节课提供的ubuntu镜像上面安装minikube，需要先把cpu调整至2： 先关闭虚拟机然后调整cpu个数至4（至少是2，可以多调整）

安装步骤（ubuntu）

推荐：

- 如果可以流畅访问https://kubernetes.io/zh-cn/docs/tutorials/kubernetes-basics/，则也可以直接在交互网页上执行
- 安装k8s
  参考官网，可以自行安装在windows或者linux等，如果安装在上节课提供的镜像里，可以参考如下的教程（需要自行处理各种镜像下载问题）：

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

- ref：

https://developer.aliyun.com/article/886463

实验内容

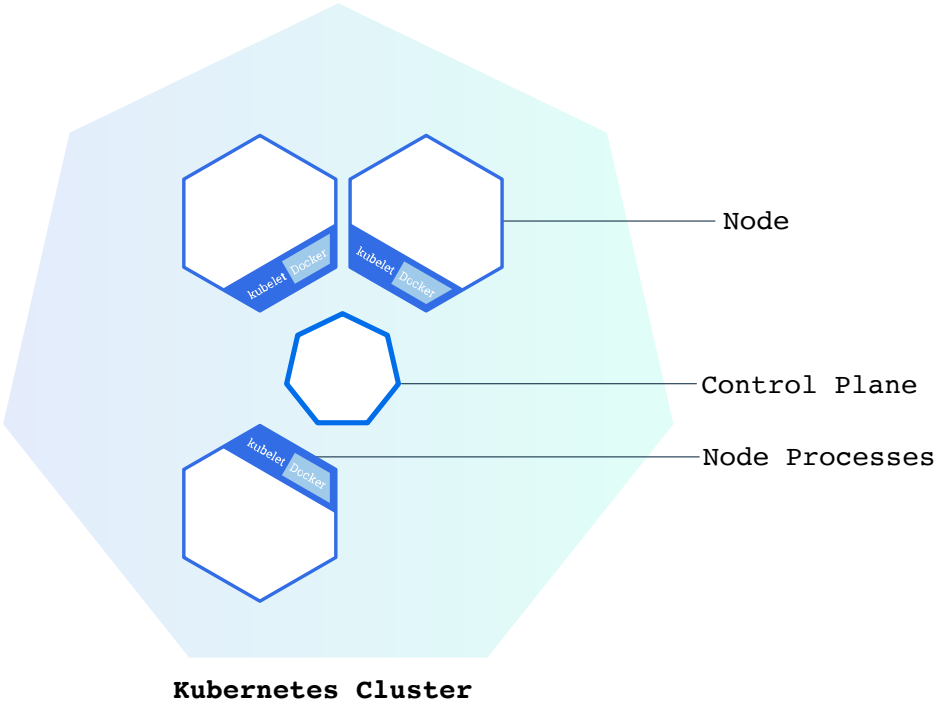
在minikube环境并按照官网指南（https://kubernetes.io/zh-cn/docs/tutorials/kubernetes-basics/）中的交互程序（推荐）完成6个小实验（或者在本地搭建minikube环境（进阶选手））：

1. 创建集群

Kubernetes 协调一个高可用计算机集群，每个计算机作为独立单元互相连接工作。 Kubernetes 中的抽象允许你将容器化的应用部署到集群，而无需将它们绑定到某个特定的独立计算机。为了使用这种新的部署模型，应用需要以将应用与单个主机分离的方式打包：它们需要被容器化。与过去的那种应用直接以包的方式深度与主机集成的部署模型相比，容器化应用更灵活、更可用。
Kubernetes 以更高效的方式跨集群自动分发和调度应用容器。 Kubernetes 是一个开源平台，并且可应用于生产环境。

一个 Kubernetes 集群包含两种类型的资源:

- **Master** 调度整个集群
- **Nodes** 负责运行应用



Master 负责管理整个集群。 Master 协调集群中的所有活动，例如调度应用、维护应用的所需状态、应用扩容以及推出新的更新。

Node 是一个虚拟机或者物理机，它在 Kubernetes 集群中充当工作机器的角色 每个Node都有 Kubelet , 它管理 Node 而且是 Node 与 Master 通信的代理。 Node 还应该具有用于处理容器操作的工具，例如 Docker 或 rkt 。处理生产级流量的 Kubernetes 集群至少应具有三个 Node，因为如果一个 Node 出现故障其对应的 etcd 成员和控制平面实例都会丢失，并且冗余会受到影响。 你可以通过添加更多控制平面节点来降低这种风险 。

Master 管理集群，Node 用于托管正在运行的应用。

在 Kubernetes 上部署应用时，你告诉 Master 启动应用容器。 Master 就编排容器在集群的 Node 上运行。 **Node 使用 Master 暴露的 Kubernetes API 与 Master 通信。**终端用户也可以使用 Kubernetes API 与集群交互。

Kubernetes 既可以部署在物理机上也可以部署在虚拟机上。你可以使用 Minikube 开始部署 Kubernetes 集群。 Minikube 是一种轻量级的 Kubernetes 实现，可在本地计算机上创建 VM 并部署仅包含一个节点的简单集群。 Minikube 可用于 Linux， macOS 和 Windows 系统。Minikube CLI 提供了用于引导集群工作的多种操作，包括启动、停止、查看状态和删除。在本教程里，你可以使用预装有 Minikube 的在线终端进行体验。

既然你已经知道 Kubernetes 是什么，让我们转到在线教程并启动我们的第一个 Kubernetes 集

群！

实验1：

通过在终端中执行 minikube start 来创建一个单节点的K8S集群：

确认一下minikube的版本：

```
minikube version
```

开启集群，运行：

```
sudo usermod -aG docker $USER && newgrp docker
minikube start --driver=docker --image-mirror-country='cn'
```

太棒了现在，您的终端中有一个正在运行的Kubernetes集群。Minikube为您启动了一个虚拟机，并且Kubernetes集群现在正在该VM中运行。

为了在这个实验中与Kubernetes交互，我们将使用命令行界面kubectl。我们将在下一个模块中详细解释kubectl，但现在，我们只想了解一些集群信息。要检查是否安装了kubectl，可以运行kubectl-version命令：

```
kubectl version
```

好了，kubectl已经配置好了，我们可以看到客户端和服务器的版本。客户端版本是kubectl版本；服务器版本是安装在主机上的Kubernetes版本。您还可以查看有关构建的详细信息。

让我们查看集群详细信息。我们将通过运行kubectl cluster info:

```
kubectl cluster-info
```

在本教程中，我们将关注用于部署和探索应用程序的命令行。要查看集群中的节点，请运行kubectl get nodes命令：

```
kubectl get nodes
```

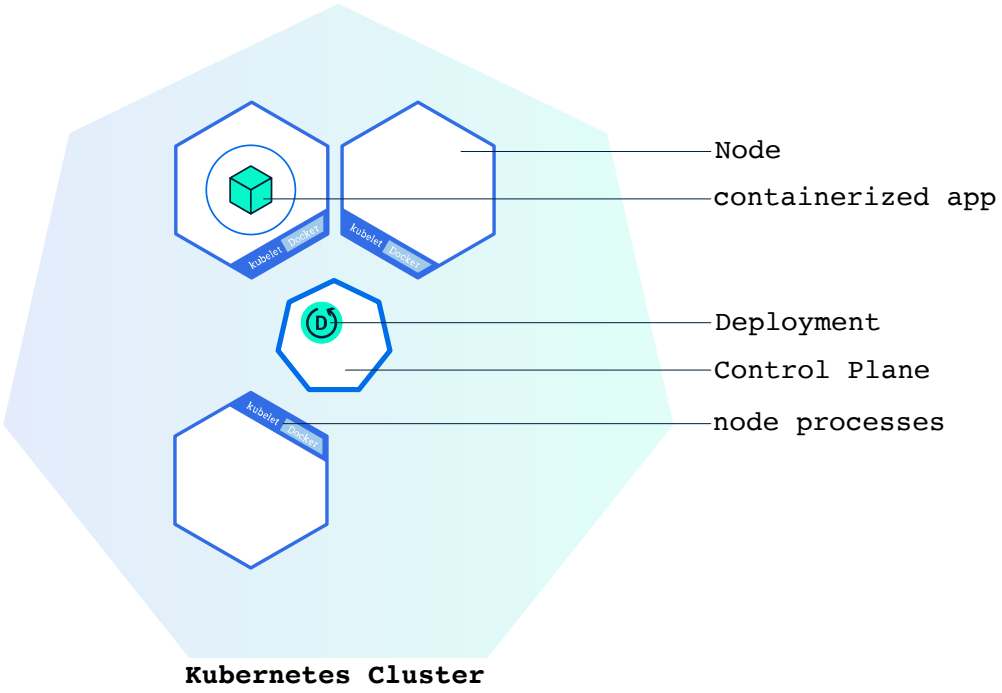
此命令显示可用于承载应用程序的所有节点。现在我们只有一个节点，我们可以看到它的状态已就绪（它已准备好接受部署应用程序）。

2.部署应用

一旦运行了 Kubernetes 集群，就可以在其上部署容器化应用程序。 为此，你需要创建 Kubernetes **Deployment** 配置。Deployment 指挥 Kubernetes 如何创建和更新应用程序的实例。创建 Deployment 后，Kubernetes master 将应用程序实例调度到集群中的各个节点上。

创建应用程序实例后，Kubernetes Deployment 控制器会持续监视这些实例。 如果托管实例的节点关闭或被删除，则 Deployment 控制器会将该实例替换为集群中另一个节点上的实例。 **这提供了一种自我修复机制来解决机器故障维护问题。**

在没有 Kubernetes 这种编排系统之前，安装脚本通常用于启动应用程序，但它们不允许从机器故障中恢复。通过创建应用程序实例并使它们在节点之间运行， Kubernetes Deployments 提供了一种与众不同的应用程序管理方法。



你可以使用 Kubernetes 命令行界面 **Kubectl** 创建和管理 Deployment。Kubectl 使用 Kubernetes API 与集群进行交互。在本单元中，你将学习创建在 Kubernetes 集群上运行应用程序的 Deployment 所需的最常见的 Kubectl 命令。

创建 Deployment 时，你需要指定应用程序的容器镜像以及要运行的副本数。你可以稍后通过更新 Deployment 来更改该信息; 模块 5 和 6 讨论了如何扩展和更新 Deployments。

应用程序需要打包成一种受支持的容器格式，以便部署在 Kubernetes 上

对于我们的第一次部署，我们将使用打包在 Docker 容器中的 Node.js 应用程序。 要创建 Node.js 应用程序并部署 Docker 容器，请按照 你好 Minikube 教程.

现在你已经了解了 Deployment 的内容，让我们转到在线教程并部署我们的第一个应用程序！

实验2：

与minikube一样，kubectl安装在在线终端中。在终端中键入kubectl以查看其用法。kubectl命令的常见格式是：kubectl-action resource。这将对指定的资源（如节点、容器）执行指定的操作（如创建、描述）。您可以在命令之后使用--help来获取有关可能参数的其他信息（kubectl get nodes--help）。

通过运行kubectl version命令，检查kubect是否配置为与集群对话：

```
kubectl version
```

好的，安装了kubectl，您可以看到客户端和服务器版本。

要查看集群中的节点，请运行kubectl get nodes命令：

```
kubectl get nodes
```

这里我们看到了可用节点（在我们的例子中为1）。Kubernetes将根据节点可用资源选择部署应用程序的位置。

让我们使用kubectl create deployment命令在Kubernetes上部署我们的第一个应用程序。我们需要提供部署名称和应用程序映像位置（包括Docker hub外部托管映像的完整存储库url）。

```
kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-
samples/kubernetes-bootcamp:v1
```

太棒了您刚刚通过创建部署部署了第一个应用程序。这为您执行了一些操作：

搜索可以运行应用程序实例的合适节点（我们只有1个可用节点）计划应用程序在该节点上运行，将群集配置为在需要时在新节点上重新安排实例

要列出部署，请使用get deployments命令：

```
kubectl get deployments
```

我们看到有1个部署运行您的应用程序的单个实例。实例正在节点上的Docker容器中运行。

运行在Kubernetes内部的Pod运行在一个私有的、隔离的网络上。默认情况下，它们在同一kubernetes集群内的其他pod和服务中可见，但在该网络之外不可见。当我们使用kubectl时，我们通过API端点与应用程序进行交互。

我们将在模块4中介绍如何在kubernetes集群之外公开应用程序的其他选项。

kubectl命令可以创建一个代理，将通信转发到集群范围的专用网络。按control-C可以终止代理，并且在运行时不会显示任何输出。

我们将打开第二个终端窗口来运行代理。

```
echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a
response. Please click the first Terminal Tab\n";  kubectl proxy
```

kubectl代理

现在，我们的主机（在线终端）和Kubernetes集群之间建立了连接。代理允许从这些终端直接访问API。

您可以看到通过代理端点托管的所有API。例如，我们可以使用curl命令直接通过API查询版本：

```
curl http://localhost:8001/version
```

注意：检查终端顶部。代理在新选项卡（终端2）中运行，最近的命令在原始选项卡（终端1）中执行。代理仍然在第二个选项卡中运行，这使得我们的curl命令可以使用localhost:8001运行。

如果端口8001不可访问，请确保上面启动的kubectl代理正在运行。

API服务器将根据pod名称自动为每个pod创建一个端点，该端点也可以通过代理访问。

首先，我们需要获取Pod名称，并将其存储在环境变量Pod_name中：

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}
{{.metadata.name}}{{"\n"}}{{end}}')
echo Name of the Pod: $POD_NAME
```

您可以通过运行API访问Pod：

```
curl http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/
```

为了在不使用代理的情况下访问新部署，需要一个服务，这将在下一个模块中解释。

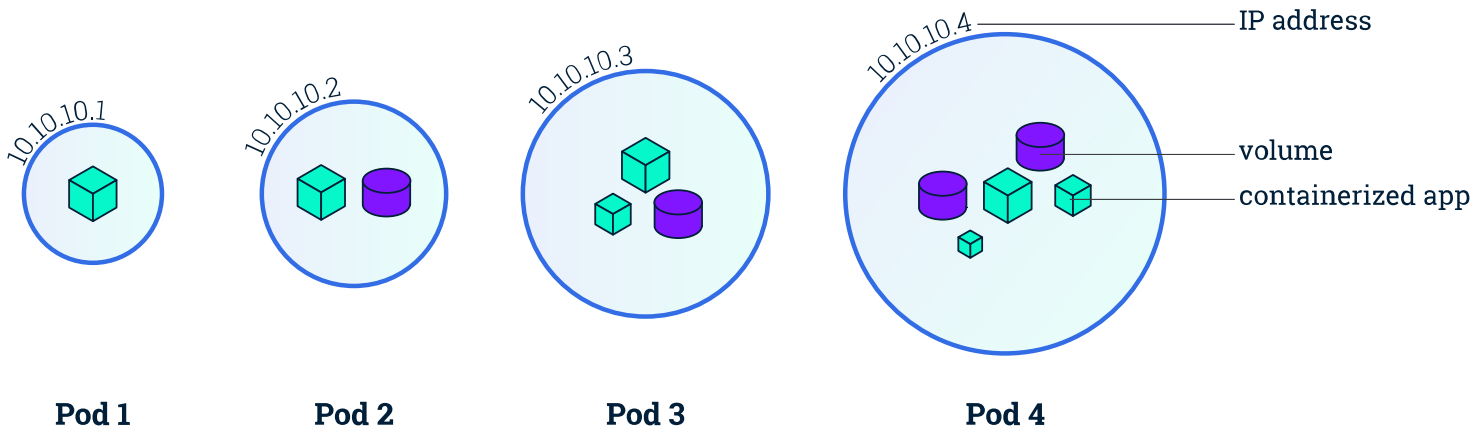
3.了解应用

在模块 2创建 Deployment 时, Kubernetes 添加了一个 **Pod** 来托管你的应用实例。Pod 是 Kubernetes 抽象出来的，表示一组一个或多个应用程序容器（如 Docker），以及这些容器的一些共享资源。这些资源包括:

- 共享存储，当作卷
- 网络，作为唯一的集群 IP 地址
- 有关每个容器如何运行的信息，例如容器镜像版本或要使用的特定端口。

Pod 为特定于应用程序的“逻辑主机”建模，并且可以包含相对紧耦合的不同应用容器。例如，Pod 可能既包含带有 Node.js 应用的容器，也包含另一个不同的容器，用于提供 Node.js 网络服务器要发布的数据。Pod 中的容器共享 IP 地址和端口，始终位于同一位置并且共同调度，并在同一工作节点上的共享上下文中运行。

Pod是 Kubernetes 平台上的原子单元。 当我们在 Kubernetes 上创建 Deployment 时，该 Deployment 会在其中创建包含容器的 Pod （而不是直接创建容器）。每个 Pod 都与调度它的工作节点绑定，并保持在那里直到终止（根据重启策略）或删除。 如果工作节点发生故障，则会在集群中的其他可用工作节点上调度相同的 Pod。

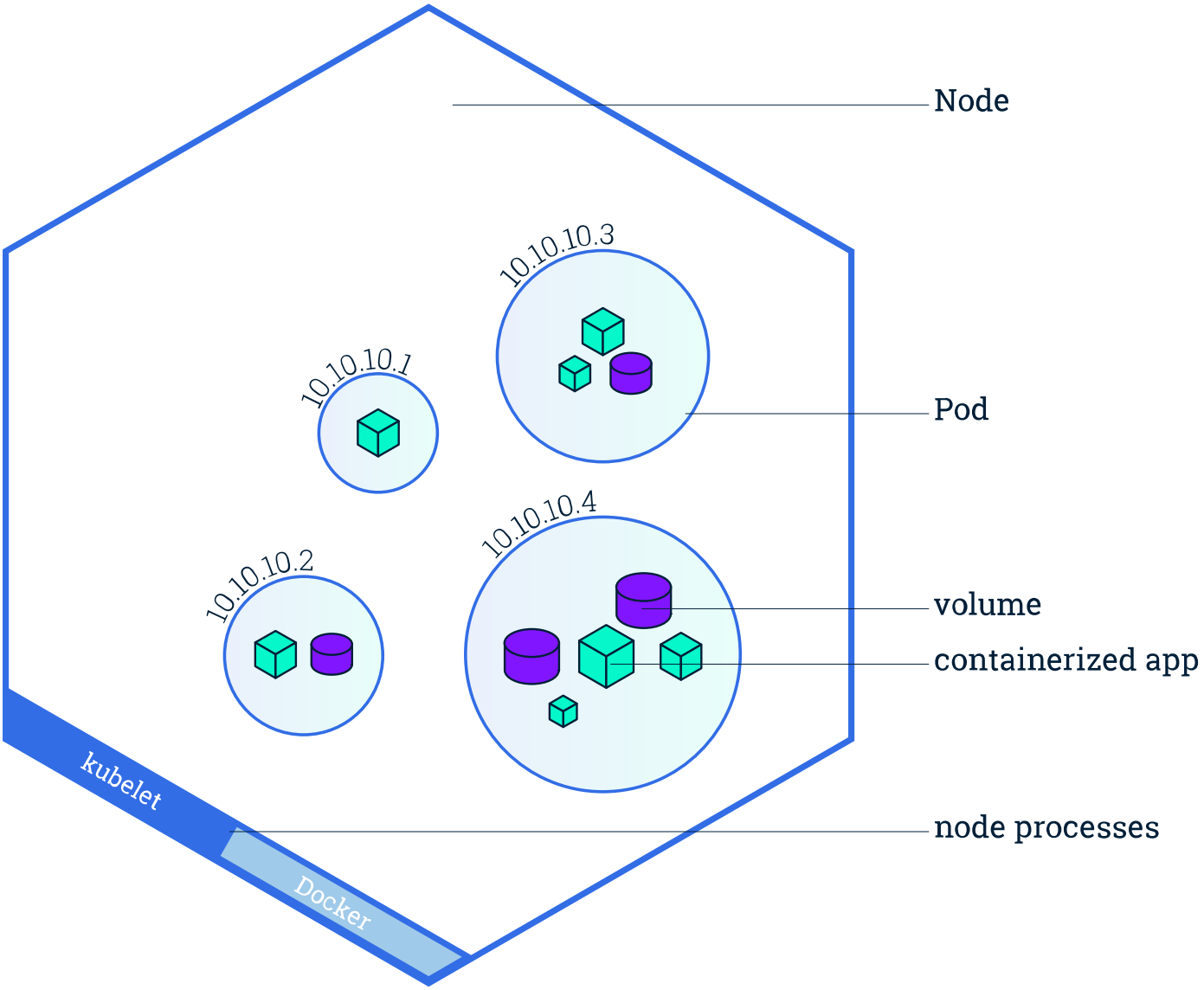


工作节点

一个 pod 总是运行在 **工作节点**。工作节点是 Kubernetes 中的参与计算的机器，可以是虚拟机或物理计算机，具体取决于集群。每个工作节点由主节点管理。工作节点可以有多个 pod ，Kubernetes 主节点会自动处理在集群中的工作节点上调度 pod 。 主节点的自动调度考量了每个工作节点上的可用资源。

每个 Kubernetes 工作节点至少运行:

- Kubelet，负责 Kubernetes 主节点和工作节点之间通信的过程; 它管理 Pod 和机器上运行的容器。
- 容器运行时（如 Docker）负责从仓库中提取容器镜像，解压缩容器以及运行应用程序。



使用 kubectl 进行故障排除

在模块 2,你使用了 Kubectl 命令行界面。 你将继续在第3单元中使用它来获取有关已部署的应用程序及其环境的信息。 最常见的操作可以使用以下 kubectl 命令完成：

- **kubectl get** - 列出资源

- **kubectl describe** - 显示有关资源的详细信息
- **kubectl logs** - 打印 pod 和其中容器的日志
- **kubectl exec** - 在 pod 中的容器上执行命令

你可以使用这些命令查看应用程序的部署时间，当前状态，运行位置以及配置。

现在我们了解了有关集群组件和命令行的更多信息，让我们来探索一下我们的应用程序。

实验3：

步骤1检查应用程序配置

让我们验证我们在前一个场景中部署的应用程序是否正在运行。我们将使用kubectl get命令查找现有Pod：

```
kubectl get pods
```

如果没有pod正在运行，则意味着交互环境仍在重新加载其先前的状态。请稍等几秒钟，然后再次列出Pod。您可以在看到一个Pod运行后继续。

接下来，要查看Pod中的容器以及用于构建这些容器的image，我们运行describe pods命令：

```
kubectl describe pods
```

我们在这里看到关于Pod容器的详细信息：IP地址、使用的端口以及与Pod生命周期相关的事件列表。

describe命令的输出非常广泛，涵盖了一些我们还没有解释的概念，但不用担心，到训练营结束时，它们会变得熟悉。

注意：describe命令可用于获取有关大多数kubernetes原语的详细信息：节点、pod和部署。description输出设计为人类可读，而不是编写脚本。

步骤2在终端中显示应用程序

回想一下Pod是在一个隔离的私有网络中运行的，所以我们需要代理对它们的访问，这样我们就可以调试它们并与它们交互。为此，我们将使用kubectl proxy命令在第二个终端窗口中运行代理。单击下面的命令自动打开新终端并运行代理：

```
echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a
response. Please click the first Terminal Tab\n"; kubectl proxy
```

现在，我们将再次获得Pod名称并直接通过代理查询该Pod。要获取Pod名称并将其存储在Pod_name环境变量中：

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}
{{.metadata.name}}{{"\n"}}{{end}}')
echo Name of the Pod: $POD_NAME
```

要查看应用程序的输出，请运行curl请求。

```
curl http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/proxy/
```

url是指向Pod的API的路径。

步骤3查看容器日志

应用程序通常发送给STDOUT的任何内容都将成为Pod中容器的日志。我们可以使用kubectl logs命令检索这些日志：

```
kubectl logs $POD_NAME
```

注意：我们不需要指定容器名称，因为pod中只有一个容器。

步骤4在容器上执行命令

一旦Pod启动并运行，我们可以直接在容器上执行命令。为此，我们使用exec命令并使用Pod的名称作为参数。让我们列出环境变量：

```
kubectl exec $POD_NAME -- env
```

同样值得一提的是，容器本身的名称可以省略，因为Pod中只有一个容器。

接下来，让我们在Pod的容器中启动一个bash会话：

```
kubectl exec -ti $POD_NAME -- bash
```

现在，我们在运行NodeJS应用程序的容器上有一个打开的控制台。应用程序的源代码在服务器中。js文件：

```
cat server.js
```

您可以通过运行curl命令来检查应用程序是否启动：

```
curl localhost:8080
```

注意：这里我们使用localhost，因为我们在NodeJS Pod中执行了命令。如果您无法连接到localhost:8080，请检查以确保您已经运行了kubectl exec命令并正在Pod中启动该命令

要关闭容器连接，请键入exit。

4. 公开的暴露你的应用

Kubernetes Service 总览

Kubernetes Pod 是转瞬即逝的。 Pod 实际上拥有 生命周期。 当一个工作 Node 挂掉后, 在 Node 上运行的 Pod 也会消亡。 ReplicaSet 会自动地通过创建新的 Pod 驱动集群回到目标状态，以保证应用程序正常运行。 换一个例子，考虑一个具有3个副本数的用作image处理的后端程序。这些副本是可替换的; 前端系统不应该关心后端副本，即使 Pod 丢失或重新创建。也就是说，Kubernetes 集群中的每个 Pod (即使是在同一个 Node 上的 Pod)都有一个唯一的 IP 地址，因此需要一种方法自动协调 Pod 之间的变更，以便应用程序保持运行。

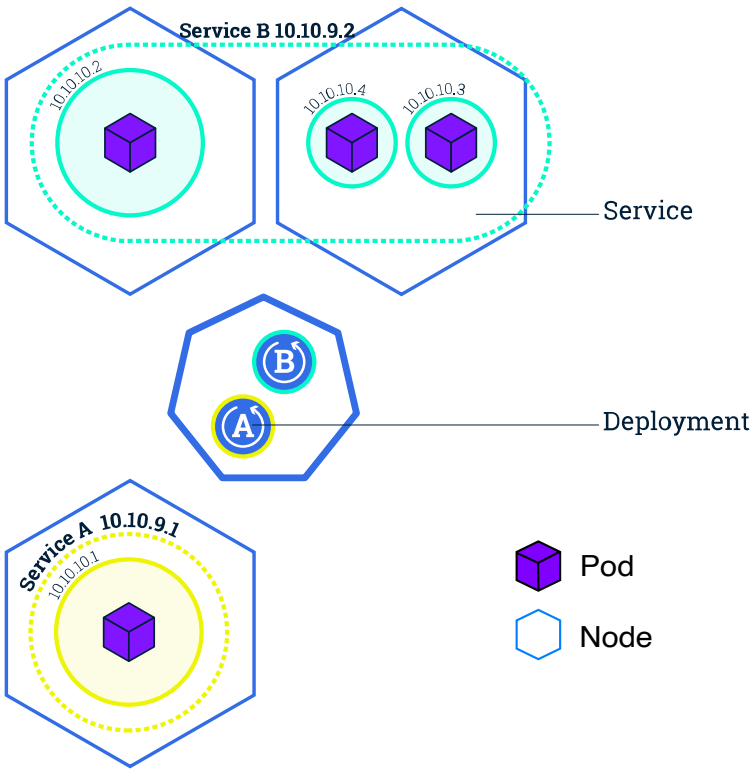
Kubernetes 中的服务(Service)是一种抽象概念，它定义了 Pod 的逻辑集和访问 Pod 的协议。Service 使从属 Pod 之间的松耦合成为可能。 和其他 Kubernetes 对象一样, Service 用 YAML (更推荐) 或者 JSON 来定义. Service 下的一组 Pod 通常由 *LabelSelector* (请参阅下面的说明为什么你可能想要一个 spec 中不包含 `selector` 的服务)来标记。

尽管每个 Pod 都有一个唯一的 IP 地址，但是如果没有 Service ， 这些 IP 不会暴露在集群外部。Service 允许你的应用程序接收流量。Service 也可以用在 ServiceSpec 标记 `type` 的方式暴露

- *ClusterIP* (默认) - 在集群的内部 IP 上公开 Service 。这种类型使得 Service 只能从集群内访问。
- *NodePort* - 使用 NAT 在集群中每个选定 Node 的相同端口上公开 Service 。使用 `<NodeIP>:<NodePort>` 从集群外部访问 Service。是 ClusterIP 的超集。
- *LoadBalancer* - 在当前云中创建一个外部负载均衡器(如果支持的话)，并为 Service 分配一个固定的外部IP。是 NodePort 的超集。
- *ExternalName* - 通过返回带有该名称的 CNAME 记录，使用任意名称(由 spec 中的 `externalName` 指定)公开 Service。不使用代理。这种类型需要 `kube-dns` 的v1.7或更高版本。

更多关于不同 Service 类型的信息可以在使用源 IP 教程。 也请参阅 连接应用程序和 Service 。

另外，需要注意的是有一些 Service 的用例没有在 spec 中定义 `selector`。 一个没有 `selector` 创建的 Service 也不会创建相应的端点对象。这允许用户手动将服务映射到特定的端点。没有 selector 的另一种可能是你严格使用 `type: ExternalName` 来标记。

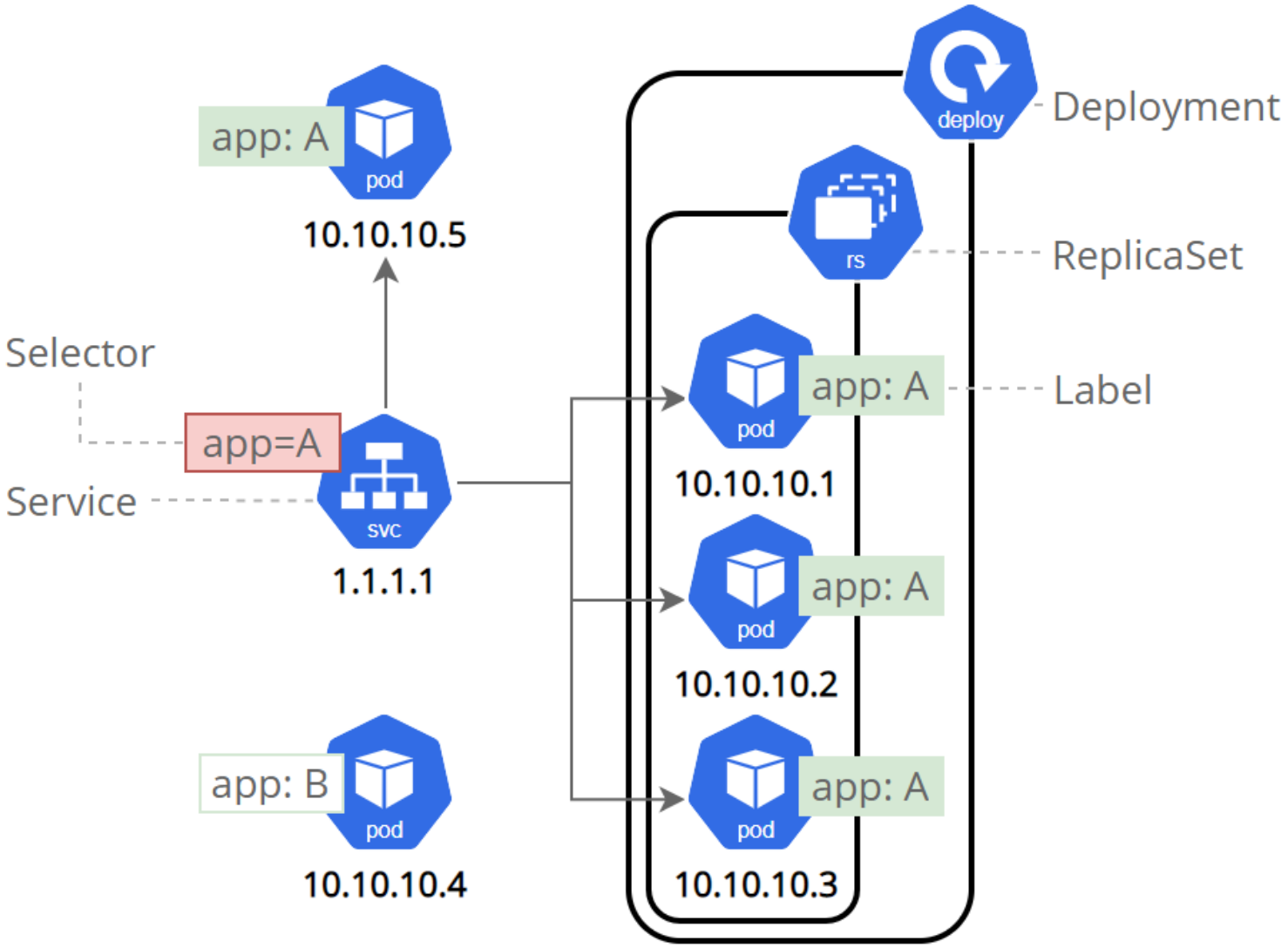


Service 通过一组 Pod 路由通信。Service 是一种抽象，它允许 Pod 死亡并在 Kubernetes 中复制，而不会影响应用程序。在依赖的 Pod (如应用程序中的前端和后端组件)之间进行发现和路由是由Kubernetes Service 处理的。

Service 匹配一组 Pod 是使用 标签(Label)和选择器(Selector), 它们是允许对 Kubernetes 中的对象进行逻辑操作的一种分组原语。标签(Label)是附加在对象上的键/值对，可以以多种方式使用:

- 指定用于开发，测试和生产的对象
- 嵌入版本标签
- 使用 Label 将对象进行分类

标签(Label)可以在创建时或之后附加到对象上。他们可以随时被修改。现在使用 Service 发布我们的应用程序并添加一些 Label 。



实验4：

1.创建新服务

让我们验证应用程序是否正在运行。我们将使用kubectl get命令查找现有Pod：

```
kubectl get pods
```

如果没有pod正在运行，则意味着交互环境仍在重新加载其先前的状态。请稍等几秒钟，然后再次列出Pod。您可以在看到一个Pod运行后继续。

接下来，让我们列出集群中的当前服务：

```
kubectl get services
```

我们有一个名为kubernetes的服务，它是在minikube启动集群时默认创建的。为了创建一个新服务

并将其暴露给外部流量，我们将使用带有NodePort作为参数的expose命令（minikube还不支持LoadBalancer选项）。

```
kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080
```

让我们再次运行get services命令：

```
kubectl get services
```

我们现在有一个名为kubernetes的训练营。这里我们看到服务收到了一个唯一的集群IP、一个内部端口和一个外部IP（节点的IP）。

要了解外部（通过NodePort选项）打开了哪个端口，我们将运行description service命令：

```
kubectl describe services/kubernetes-bootcamp
```

创建一个名为NODE_PORT的环境变量，该环境变量指定了节点端口的值：

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-
template='{{(index .spec.ports 0).nodePort}}')
echo NODE_PORT=$NODE_PORT
```

现在我们可以使用curl、节点的IP和外部暴露端口测试应用程序是否暴露在集群外部：

```
curl $(minikube ip):$NODE_PORT
```

我们从服务器得到了响应。服务已暴露。

步骤2：使用标签

Deployment自动为我们的Pod创建了一个标签。使用describe deployment命令，您可以看到标签的名称：

```
kubectl describe deployment
```

让我们使用这个标签来查询我们的Pod列表。我们将使用kubectl get pods命令和-l作为参数，后跟标签值：

```
kubectl get pods -l app=kubernetes-bootcamp
```

您可以使用相同的方法列出现有服务：

```
kubectl get services -l app=kubernetes-bootcamp
```

获取Pod的名称并将其存储在Pod_name环境变量中：

```
export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}
{{.metadata.name}}{{"\n"}}{{end}}')
echo Name of the Pod: $POD_NAME
```

要应用新标签，我们使用label命令，后跟对象类型、对象名称和新标签：
在这个实验中，要把version的编号列为自己的学号

```
kubectl label pods $POD_NAME version=v1
```

这将为我们的Pod应用一个新标签（我们将应用程序版本固定在Pod上），我们可以使用describe Pod命令进行检查：

```
kubectl describe pods $POD_NAME
```

我们在这里看到，标签现在已附加到我们的Pod。现在我们可以使用新标签查询pod列表：

```
kubectl get pods -l version=v1
```

我们看到了Pod。

3.删除服务

要删除服务，可以使用delete service命令。标签也可以在这里使用：

```
kubectl delete service -l app=kubernetes-bootcamp
```

确认服务已停止：

```
kubectl get services
```

这确认我们的服务已被删除。要确认路由不再暴露，可以curl先前暴露的IP和端口：

```
curl $(minikube ip):$NODE_PORT
```

这证明应用程序不再能够从集群外部访问。您可以确认应用程序仍在运行，并在pod内进行curl：

```
kubectl exec -ti $POD_NAME -- curl localhost:8080
```

我们在这里看到应用程序启动了。这是因为部署正在管理应用程序。要关闭应用程序，还需要删除Deployment。

5. 缩放你的应用

扩缩应用程序

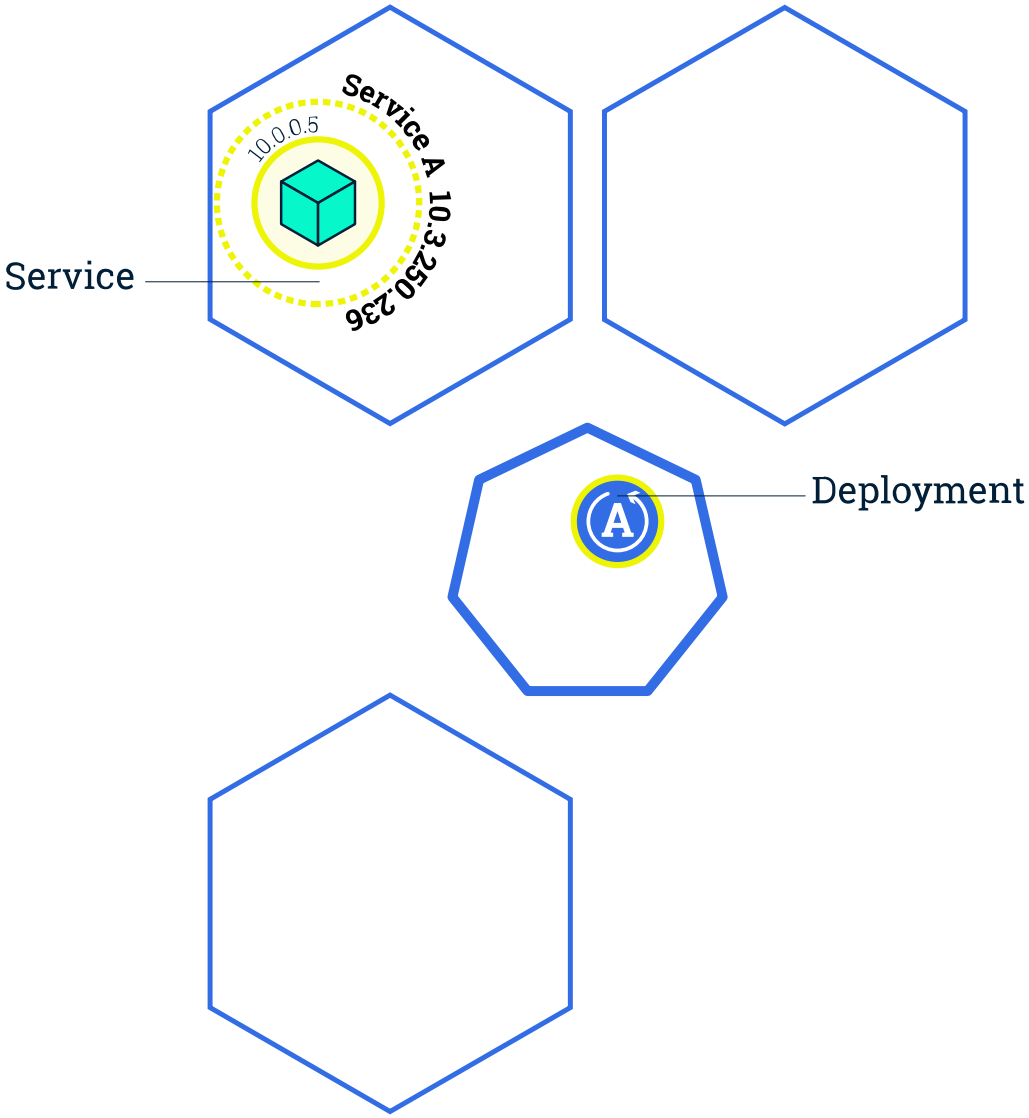
在之前的模块中，我们创建了一个 Deployment，然后通过 Service让其可以开放访问。Deployment 仅为跑这个应用程序创建了一个 Pod。 当流量增加时，我们需要扩容应用程序满足用户需求。

扩缩 是通过改变 Deployment 中的副本数量来实现的。

扩展 Deployment 将创建新的 Pods，并将资源调度请求分配到有可用资源的节点上，收缩 会将 Pods 数量减少至所需的状态。Kubernetes 还支持 Pods 的自动缩放，但这并不在本教程的讨论范围内。将 Pods 数量收缩到0也是可以的，但这会终止 Deployment 上所有已经部署的 Pods。

运行应用程序的多个实例需要在它们之间分配流量。服务 (Service)有一种负载均衡器类型，可以将网络流量均衡分配到外部可访问的 Pods 上。服务将会一直通过端点来监视 Pods 的运行，保证流量只分配到可用的 Pods 上。

扩缩是通过改变 Deployment 中的副本数量来实现的。



一旦有了多个应用实例，就可以没有宕机地滚动更新。我们将会在下面的模块中介绍这些。现在让我们使用在线终端来体验一下应用程序的扩缩过程。

实验5：

步骤1：扩展部署

要列出部署，请使用get deployments命令：

```
kubectl get deplosyments
```

输出应类似于：

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE kubernetes-bootcamp
1/1     1            1           11m
```

我们应该有一个Pod。如果没有，请再次运行该命令。这表明：

NAME列出了群集中部署的名称。

READY显示CURRENT/DESIRED复制副本的比率

UP-TO-DATE显示已更新以达到所需状态的副本数。

AVAILABLE显示用户可以使用的应用程序副本数量。

AGE显示应用程序运行的时间量。

要查看Deployment创建的ReplicaSet，请运行

```
kubectl get-rs
```

请注意，ReplicaSet的名称始终格式为[DEPLOYMENT-name]-[RANDOM-STRING]。随机字符串是随机生成的，并使用pod模板散列作为种子。

此命令的两个重要列是：

DESIRED显示应用程序的所需副本数，您在创建部署时定义这些副本。这是所需的状态。

CURRENT显示当前正在运行的副本数。

接下来，让我们将部署扩展到4个副本。我们将使用kubectl scale命令，后跟部署类型、名称和所需实例数：

```
kubectl scale deployments/kubernetes-bootcamp --replicas=4
```

要再次列出部署，请使用get Deployments：

```
kubectl get deployments
```

更改已应用，我们有4个应用程序实例可用。接下来，让我们检查Pod的数量是否更改：

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

现在有4个Pod，具有不同的IP地址。更改已在部署事件日志中注册。要进行检查，请使用description命令：

```
kubectl describe deployments/kubernetes-bootcamp
```

您还可以在该命令的输出中看到现在有4个副本。

步骤2：负载平衡

让我们检查服务是否正在负载平衡流量。为了找出暴露的IP和端口，我们可以使用我们在上一模块中学习的描述服务：

```
kubectl describe services/kubernetes-bootcamp
```

创建一个名为NODE_PORT的环境变量，该变量的值为节点端口：

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-
template='{{(index .spec.ports 0).nodePort}}')
echo NODE_PORT=$NODE_PORT
```

接下来，我们将对暴露的IP和端口进行curl。多次执行命令：

```
curl $(minikube ip):$NODE_POR
```

我们每一个请求都有不同的Pod。这表明负载平衡正在工作。

步骤3：缩小

要将服务缩减到2个副本，请再次运行scale命令：

```
kubectl scale deployments/kubernetes-bootcamp --replicas=2
```

列出Deployments以检查是否使用get Deployments命令应用了更改：

```
kubectl get deployments
```

副本的数量减少到2。列出Pod的数量，并使用get pod：

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

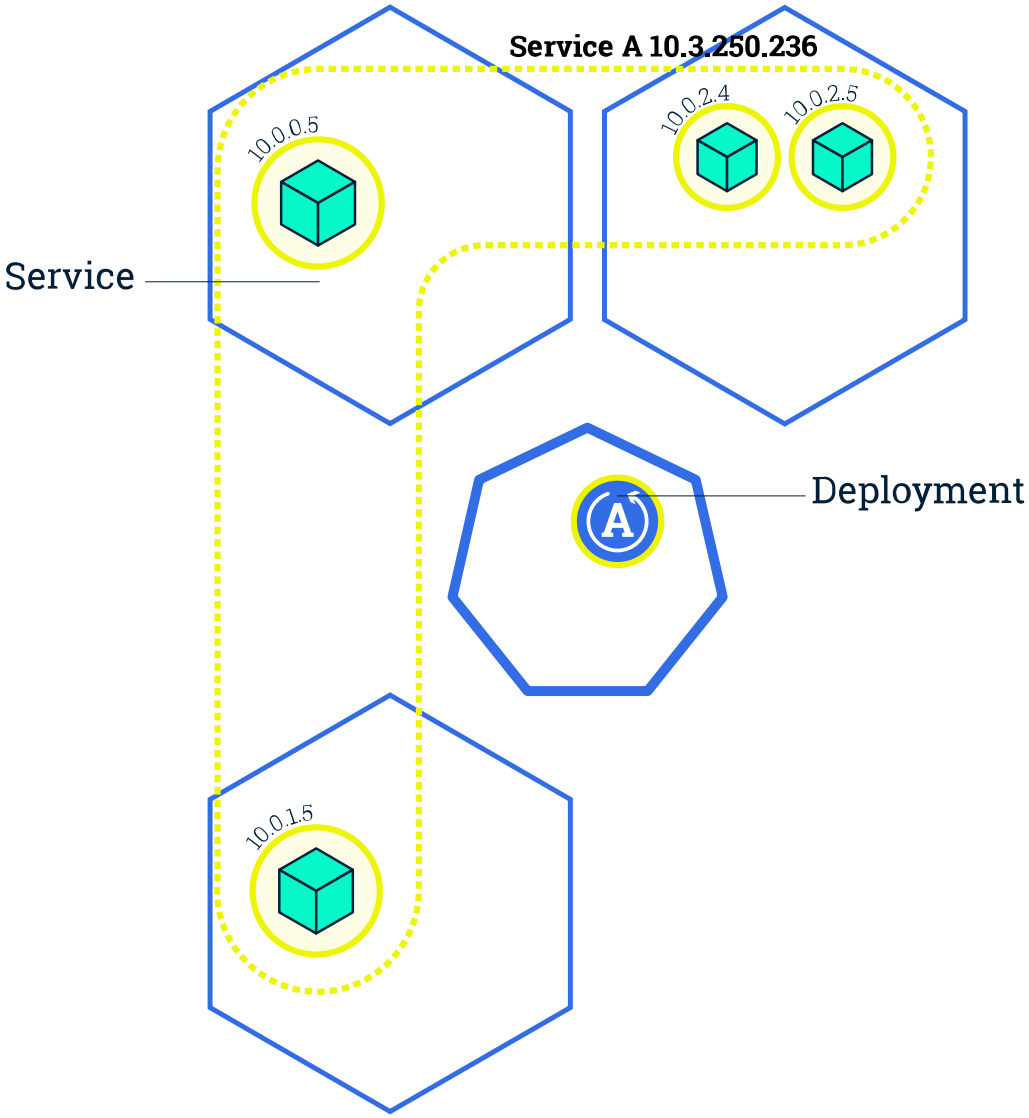
这确认2个Pod已终止。

6.更新你的应用

更新应用程序

用户希望应用程序始终可用，而开发人员则需要每天多次部署它们的新版本。在 Kubernetes 中，这些是通过滚动更新（Rolling Updates）完成的。 **滚动更新** 允许通过使用新的实例逐步更新 Pod 实例，零停机进行 Deployment 更新。新的 Pod 将在具有可用资源的节点上进行调度。

在前面的模块中，我们将应用程序扩展为运行多个实例。这是在不影响应用程序可用性的情况下执行更新的要求。默认情况下，更新期间不可用的 pod 的最大值和可以创建的新 pod 数都是 1。这两个选项都可以配置为（pod）数字或百分比。 在 Kubernetes 中，更新是经过版本控制的，任何 Deployment 更新都可以恢复到以前的（稳定）版本。



与应用程序扩展类似，如果 Deployment 是公开的，服务将在更新期间仅对可用的 pod 进行负载均衡。可用 Pod 是应用程序用户可用的实例。

滚动更新允许以下操作：

- 将应用程序从一个环境提升到另一个环境（通过容器镜像更新）
- 回滚到以前的版本
- 持续集成和持续交付应用程序，无需停机

如果 Deployment 是公开的，则服务将仅在更新期间对可用的 pod 进行负载均衡。

在下面的交互式教程中，我们将应用程序更新为新版本，并执行回滚。

实验6：

在这个实验中，稳定版本为自己的学号，并更新为V10版本，再进行回滚

步骤1：更新应用程序的版本

要列出部署，请运行get deployments命令：

```
kubectl get deployments
```

要列出正在运行的Pods，请运行get Pods命令：

```
kubectl get pods
```

要查看应用程序的当前image版本，请运行describe pods命令并查找image字段：

```
kubectl describe pods
```

要将应用程序的映像更新为版本2，请使用set image命令，后跟部署名称和新映像版本：

```
kubectl set image deployments/kubernetes-bootcamp kubernetes-
bootcamp=jocatalin/kubernetes-bootcamp:v2
```

该命令通知Deployment为您的应用程序使用不同的映像，并启动了滚动更新。检查新Pod的状态，并使用getpods命令查看旧Pod的终止：

```
kubectl get pods
```

步骤2：验证更新

首先，检查应用程序是否正在运行。要查找暴露的IP和端口，请运行description service命令：

```
kubectl describe services/kubernetes-bootcamp
```

创建一个名为NODE_PORT的环境变量，该环境变量指定了节点端口的值：

```
export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-
template='{{(index .spec.ports 0).nodePort}}')
```

```
echo NODE_PORT=$NODE_PORT
```

接下来，curl暴露的IP和端口：

```
curl $(minikube ip):$NODE_PORT
```

每次运行curl命令时，都会碰到不同的Pod。请注意，所有Pod都运行最新版本（v2）。

您还可以通过运行rollout状态命令来确认更新：

```
kubectl rollout status deployments/kubernetes-bootcamp
```

要查看应用程序的当前image版本，请运行describe pods命令：

```
kubectl describe pods
```

在输出的Image（image）字段中，验证您正在运行最新的image版本（v2）。

步骤3：回滚更新

让我们执行另一个更新，并部署一个标有v10的映像：

```
kubectl set image deployments/kubernetes-bootcamp kubernetes-
bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10
```

使用get deployment查看部署的状态：

```
kubectl get deployments
```

请注意，输出没有列出所需的可用Pod数量。运行get pods命令以列出所有Pod：

```
kubectl get pods
```

请注意，一些Pod的状态为ImagePullBackOff。

要深入了解问题，请运行describe pods命令：

```
kubectl describe pods
```

在受影响Pod输出的Events部分中，请注意存储库中不存在v10映像版本。

要将展开回滚到上一个工作版本，请使用rollout撤消命令：

```
kubectl rollout undo deployments/kubernetes-bootcamp
```

rollout撤消命令将展开恢复到先前的已知状态（image的v2）。更新是版本化的，您可以恢复到部署的任何先前已知状态。

使用get pods命令再次列出pods：

```
kubectl get pods
```

四个吊舱正在运行。要检查部署在这些Pod上的映像，请使用describe Pods命令：

```
kubectl describe pods
```

部署再次使用稳定版本的应用程序（v2）回滚成功。