容器编排

容器编排平台简介

集群化、弹性化和敏捷化是容器应用的显著特点，如何有效地对容器集群进行管理，是容器技术落地应用的一个重要方面。集群管理工具（编排工具）能够帮助用户以集群的方式在主机上启动容器，并能够实现相应的网络互联，同时提供负载均衡、可扩展、容错和高可用等保障。当前关注度和使用率比较高的几种容器编排平台主要包括 Kubernetes、Apache Mesos、Docker Swarm、Openshift、Rancher 等，其中还不乏有许多公有云厂商均各自建立了 Kubernetes 服务托管云平台。

Kubernetes

Kubernetesl 是由 Google 基于其内部大规模集群管理系统 Borg[1] 发布的开源分布式容器管理平台， 简称 K8S。Kubernetes 为用户提供了集群管理能力、多租户应用支撑能力、透明的服务注册和服务发现机制、负载均衡能力以及故障发现和自修复等能力。

Kubernetes 架构如图 0.1 所示，主要由一个或多个 Master 节点和一个或多个 Node 节点组成。

[1] https://en.wikipedia.org/wiki/Borg_(cluster_manager)

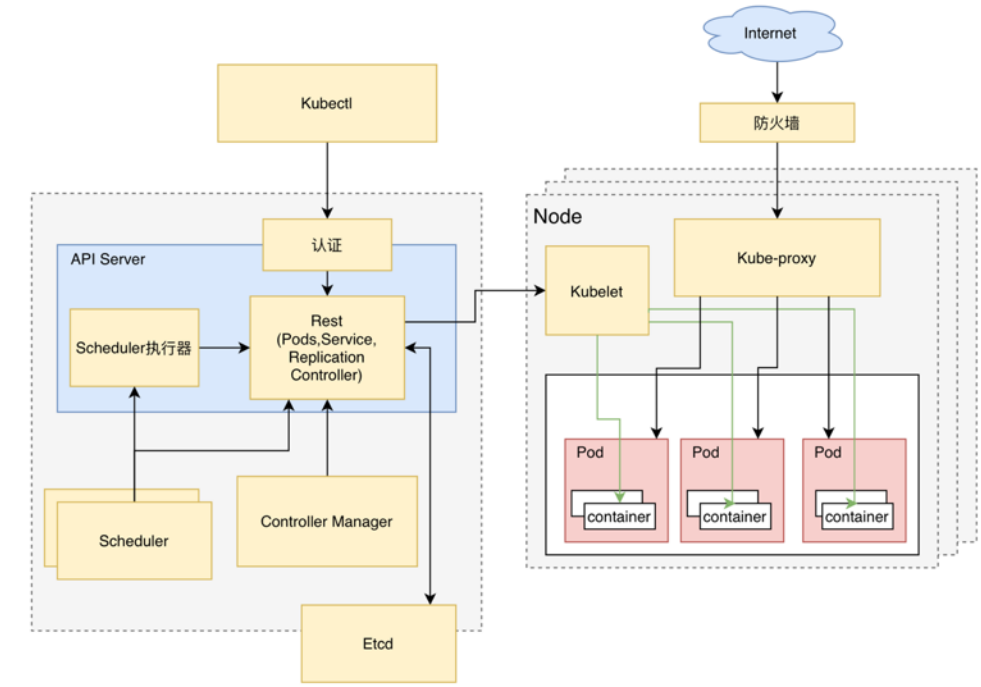

图 0.1 Kubernetes 架构图

Master 节点包含多个重要组件，例如和存储相关的 Etcd 组件、和调度相关的 Scheduler 组件、集群控制高可用组件 Controller Manager、与外部通讯协调整个集群的组件 API Server 、与 Kubernetes 集群 DNS 解析相关的 CoreDNS 组件等。

Node 节点含有两个重要的组件，分别为管理集群中 Pod 生命周期的 Kubelet 组件 和负责 Kubernetes 中网络配置的 Kube-proxy 组件。

Kubernetes 中有一些重要单元，本书后续会涉及：

Pod：Kubernetes 中运行应用或服务的最小单元，其设计理念是支持多个容器在一个 Pod 中共享网络地址和文件系统。

Service：访问 Pod 的代理抽象服务，主要用于集群内部的服务发现和负载均衡。

Deployment：用于管理 Pod 对象， 集成了上线部署、滚动升级、创建副本、暂停 / 恢复上线任务、回滚等功能。

Volumes：即目录，其中存有数据，在容器启动时指定参数会自动挂载至容器内部，Volumes 分静态和动态，其具有不同的生命周期。

StatefulSet：通常创建的 Pod 是无状态的， 这样会导致一旦 Pod 挂掉再重启后会找不到之前挂载的 Volume，所以可以通过 StatefulSet 来保留 Pod 的状态。

DaemonSet：DaemonSet 确保所有节点均运行 Pod 副本。当集群中添加节点时，节点自动进行 Pod 添加。当集群中删除节点时，Pod 自动删除。

综合以上 Kubernetes 单元的解释，Kubernetes Pod 工作流程可描述如下：

1. 提交请求

用户通常提交一个 yaml 文件，向 API Server 发送请求创建一个 Pod，yaml 文件含有此 Pod 的详细信息，包 含此 Pod 运行副本数、镜像、Labels、名称，端口暴露情况等。API Server 接收到请求后将 yaml 文件中的 spec 数据存入 Etcd 中。

2. 资源分配

Scheduler 通过 API Server 的 watch 接口定时监听 Etcd 数据库中资源的变化（此处指上一步待分配的 Pod），当监测到了 Pod 后，通过 Scheduler 的调度策略选择出具有运行 Pod 能力的 Node 节点并将 Pod 与目标 Node 节点进行绑定，同时更新 Etcd 数据库中 Pod 的分配情况。

3. 新建容器

此时目标 Node 节点上的 Kubelet 通过 API Server 的 Watch 接口监测到 Etcd 中 Pod 的分配信息， 同时将 Pod 的相关数据传递给容器运行时以负责此 Pod 的整个生命周期，之后 Kubelet 还会通过容器运行时获取 Pod 的 状态信息并通过 API Server 更新至 Etcd。

4. 资源状态同步

为了保证此 Pod 在 Node 节点中运行正常（Pod 可能会因为某些原因被杀死），Controller Manager 中的 Replication Set 组件通过 API Server 定时监听 Etcd 以获得 Pod 的最新状态并最终对 Pod 进行数量上的同步，从而保证了 Pod 运行副本数与用户指定副本数相同。

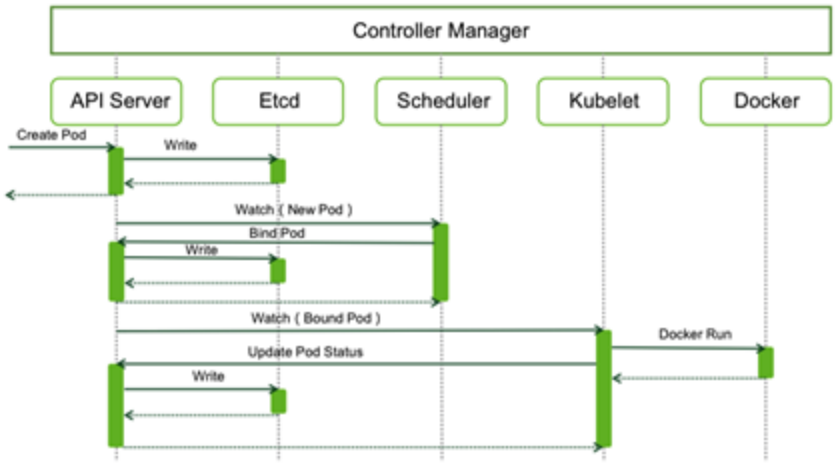

图 0.2 Kubernetes 创建 Pod 流程示意图

除了在本地部署 Kubernetes，许多公有云厂商也推出了各自的 Kubernetes 托管云平台，国外公有云厂商主要以 Google、Amazon、Microsoft Azure 为主，国内则以阿里、腾讯、华为为主。

我们在 Google 趋势图2（Google Trends）中按关键词 “Google GKE” [3]，“Amazon EKS” [4]，“Microsoft AKS” [5]查看了全球近五年的热度趋势，如 **Error! Reference source not found.**所示：

图 0.3. GKE EKS AKS 近五年热度趋势

可以看出，在 2017 年 10 月之前，由于 Amazon EKS 和 Microsoft AKS 基本处在早期萌芽阶段，因此早在 2015 年 8 月就发布了的 Google GKE 一直位于领先地位。2017 年底至

2 https://trends.google.com/trends/?geo=US
3 https://aws.amazon.com/cn/eks/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc&eks-blogs.sort-by=item.additionalFields.createdDate&eks-blogs.sort-order=desc
4 https://azure.microsoft.com/en-us/services/kubernetes-service/
5 https://cloud.google.com/kubernetes-engine

2018 年 6 月，Amazon EKS 经过了 3 年的沉淀与发展，其热度呈指数增长，一度远超 Google GKE 和 Microsoft AKS。2018 年 6 月至今 Google GKE 与 Amazon EKS 热度平稳上升，基本为持平状态，而 Microsoft AKS 热度虽然也承上升趋势，但相比其它两家还存在着较大差距。

国内 Kubernetes 托管云平台发展相对较晚，目前主要以阿里的容器服务 Kubernetes 版 ACK6（Alibaba Cloud Container Service for Kubernetes）、腾讯的 TKE 7(Tencent Kubernetes Engine)、华为的 CCE8 (Huawei Cloud Container Engine)为代表。

Docker Swarm

Swarm 是 Docker 公司在 2014 年 12 月初发布的一种容器编排解决方案，并在 Docker Engine v.1.12 版本中作为一种模式被引入。由于 Swarm 使用标准的 Docker API 接口，因此可以说 Swarm 是 Docker 原生的容器编排系统，其易用性及便利性也在早期获得了一批拥簇者。

像许多容器编排解决方案一样，Swarm 架构由 Manager 节点和 Worker 节点组成，其中 Manager 节点负责容器编排和集群管理，Worker 节点从 Manager 节点接收并执行任务，除此之外，Swarm 设计还包含 Task、Service、Node 的概念：

Task：Swarm 的最小原子调度单元，一个 Service 包含一个或多个 Task，Task 可通常视为一个 Docker 容器；

Service：用户可以通过指定容器镜像和副本数量创建一个 Service，Swarm 中包含两类 Service，分别为 replicated services 和 global services, 其中 replicated services 根据用户设定期望运行的副本数在节点间进行 Task 分配，global services 则在集群中的每个可用节点上运行一个 Task；

Node：Node 可以理解为 Swarm 的一个 Docker Engine 实例，可部署在本地或公有云中；

下图为 Docker Swarm 的架构图：

6 https://www.alibabacloud.com/product/kubernetes
7 https://intl.cloud.tencent.com/product/tke
8 https://www.huaweicloud.com/en-us/product/cce.html

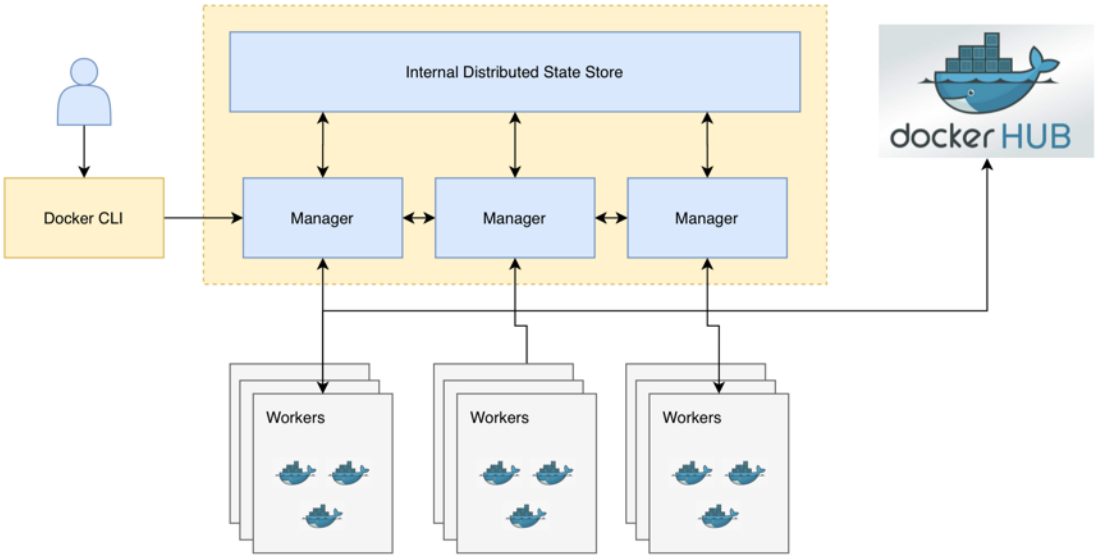

图 0.4 Swarm 架构设计

由图 0.4 可以看出，集群中可以有多个 Manager 节点，每个 Manager 节点可对任意 Worker 节点进行服务调度，同时，所有节点都具有基本的 Docker 运行组件：Docker Daemon、负载均衡。它们为容器的运行提供了基础环境。以下介绍 Swarm 的几个重要组件：

Manager 节点：主要由 Manager CLI（Swarm 命令行工具）、Scheduler（调度器）、Orchestrator（编排模块）、Allocater（分配 IP 模块）、Dispatcher（Task 分发模块）、Discovery Service（服务发现）组件构成；

Worker 节点：主要组件为 Swarm Agent；

服务发现组件：集群中的物理和通讯环境复杂，服务中断无法准确预测，服务发现组件为集群提供了服务保障。当某个 Worker 节点宕机之后，服务发现组件会快速发现，通过调度器在另一个节点恢复已经失效的服务。

Apache Mesos/Marathon

Apache Mesos 源自 UC Berkeley 的一个对集群资源进行抽象和管理的开源项目，后在 Twitter 得到广泛使用。 Apache Mesos 将整个数据中心的资源进行抽象和管理调度，有类似主机操作系统的功能，使得多个应用同时运 行在集群中分享资源。

Apache Mesos 的架构如图 0.5 所示， 由一个或多个 Master 节点和一个或多个 Slave 节点， 以及若干个 Framework 应用程序组成。

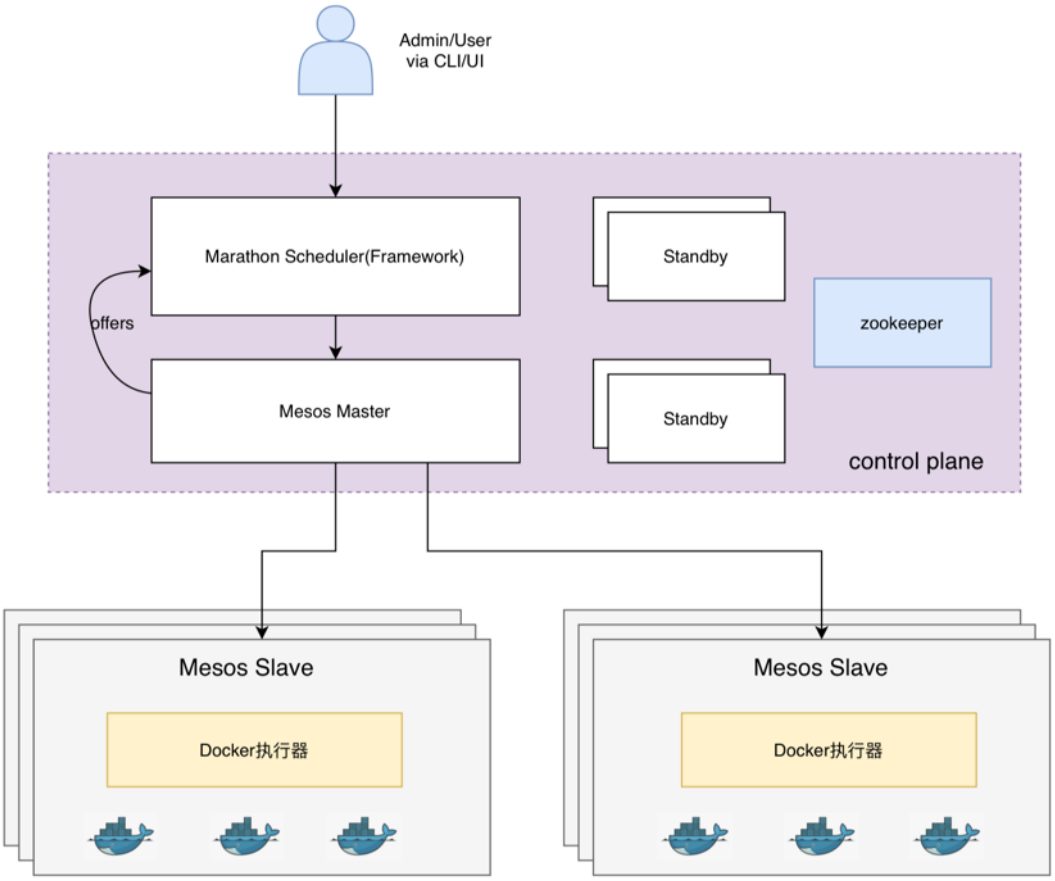

图 0.5 Apache Mesos 架构图

Master 主要负责管理各个 Framework 和 Slave 节点，并将 Slave 节点上的资源分配给相关的 Framework。 Slave 负责管理当前节点上的任务分配。Framework 是外部计算框架，可插拔，常用的有 Marathon、Hadoop、MPI 等， Framework 可通过注册的方式接入 Mesos，以便 Mesos 进行统一管理和资源分配。

Framework 主要由两个组件组成，分别是调度器和执行器。调度器与 Master 节点交互，根据可用资源，将任务调度至 Slave 中加载；执行器从 Framework 中获取变量，在 Slave 节点中运行任务。

OpenShift Container Platform

OpenShift Container Platform 是一个开发和运行容器化应用的平台，为 Redhat Openshift 旗下的一款核心产品。OpenShift Container Platform 围绕着以 CRI-O9驱动的容器核心进行构建，并由 Kubernetes 提供编排及管理，可支持在本地、公有云、混合云、多云环境中部署，其在 2015 年 6 月被首次推出，是市场中最早出现的企业级 Kuberentes 解决方案之一。

下图展示了 OpenShift Container Platform 架构：

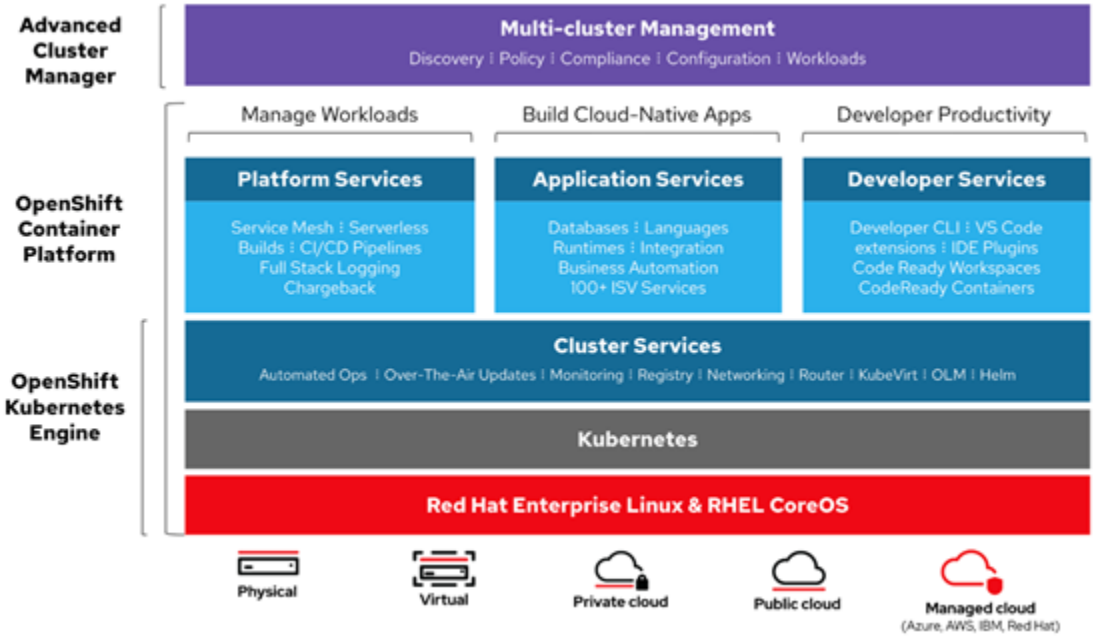

图 0.6 OpenShift Container Platform 架构10

图 0.6 由下至上可以看出，OpenShift Container Platform 可部署在物理机、虚拟机、公有云和私有云中，运行在专为容器设计的轻量级 Linux 发行版 CoreOS 之上。OpenShift Container Platform 的核心组件为 Openshift Kubernetes Engine，其主要提供容器编排以及集群管理能力，再往上一层 OpenShift Container Platform 还支持服务网格、Serverless、数

9 CRI-O 是一个开源的、社区驱动的容器引擎。它的主要目标是取代 Docker 服务作为 Kubernetes 实现的容器引擎，例如 OpenShift 容器平台。

10 https://docs.openshift.com/container-platform/4.6/welcome/oke_about.html

据库、云原生应用等部署。 最上一层 Openshift 使用 Advanced Cluster Manager 对多 Kubernetes 集群进行管理。

OpenShift Container Platform 为开发者提供了许多便捷的功能，主要体现为以下几点：

1. 兼容原生 Kubernetes，包括对应用的 Kubernetes Deployment 资源部署、kubectl 命令行、Pod 安全策略（Pod Security Policie PSP）、Kubernetes Ingress、容器网络接口（Container Network Interface CNI）、容器存储接口（Container Storage Interface CSI）等；
2. 集成第三方云原生工具以实现增值服务，包括监控、日志管理、网络、存储、容器镜像构建、持续集成部署(CI/CD)、Helm Charts 部署等；
3. 支持服务网格部署，Red Hat OpenShift Service Mesh 为基于 Istio 设计的服务网格；
4. 支持 Serverless 部署，Red Hat OpenShift Serverless 为基于 Knative 设计的 Serverless。

Rancher

Rancher 成立于 2014 年，是一家专注于企业级 Kubernetes 解决方案的提供商。2016 年，Rancher 为支持容器编排引擎发布了 Rancher 1.0 版本，其中包括 Rancher 自己的容器编排引擎 Cattle11。随着 Kubernetes 在市场上的兴起，Rancher 改用 Kubernetes 作为其核心编排引擎，并随后发布了 Rancher 2.x 版本，Rancher 2.x 不仅简化了 Kubernetes 操作流程，还使开发者可在任意处部署和管理多个 Kubernetes 集群，从而提供了 Kubernetes 即服务的能力（Kubernetes-as-a-Service）。

Rancher 2.x 可以创建来自 Kubernetes 托管服务提供商的集群，例如 GKE12（Google Kubernetes Engine）、AKS13(Azure Kubernetes Service)、EKS14（Elastic Kubernetes Service），也可以自动创建节点安装 Kubernetes 集群，或者导入任何已经存在的 Kubernetes 集群15。

---

11 https://github.com/rancher/cattle
12 https://cloud.google.com/kubernetes-engine
13 https://azure.microsoft.com/en-us/services/kubernetes-service/
14 https://aws.amazon.com/cn/eks/
15 https://docs.rancher.cn/docs/rancher2/overview/_index

由于 Rancher 1.x 版本目前受众面较窄且不支持 Kubernetes，因此开发者基本已转向使用 Rancher 2.x 版本，以下内容均基于 Rancher 2.x 版本展开。

Rancher 2.x 的核心为 Rancher Server，其地位类似于 Kubernetes 中的 Master 节点，提供核心的容器管理及 API 服务，Rancher Server 的主要组件包括认证代理、Rancher API Server、集群控制器（Cluster Controller）、Etcd 节点和集群 Agent(Cluster Agent)，如下图所示，除了集群 Agent，其它组件均部署在 Rancher Server 中16，Rancher Server 组件的基本结构与 Kubernetes 组件类似。

除了以上介绍的几款主流容器编排工具之外，还有一些市场上使用率较低的容器编排工具，例如 VMware 与 Pivotal 联合开发的企业级 Kubernetes 平台 PKS17 (Pivotal Container Service)、HashiCorp 公司开源的 nomad18等。

Kubernetes

Kubernetes 组件介绍

众所周知，Kubernetes 目前已成为业界容器编排工具的事实上标准，其众多追随者必然是有原因的。首先，Kubernetes 非常轻量级，我们知道通常 Kubernetes 都是以容器作为载体，而容器本来就具有轻量级秒级部署的特点；其次，Kubernetes 有火热的开源社区，自从 Kubernetes 加入 CNCF 组织后，来自世界各地的许多容器开发者参与其中，其中不乏有像 Redhat、IBM、华为等大厂的开发人员，因此良好的生态圈成为了备受关注的主要原因。

由于 Kubernetes 有着弹性伸缩，横向扩展的优势并同时提供负载均衡能力以及良好的自愈性（自动部署、自动重启、自动复制、自动扩展等），因此开发者可在 Kubernetes 中轻松地部署应用。

Kubernetes 由五种主要组件构成，它们之间的协同工作完成了整个集群的管理，这五种组件分别为 API Server、Controller Manager、Scheduler、Kubelet、Etcd。

根据 0 小节中介绍的 kubernetes 工作流可以看出，从客户端发送创建 Pod 请求到 Pod 最终部署至节点期间参与的主要组件有 API Server、 Controller-Manager、 Scheduler、

16 https://docs.rancher.cn/docs/rancher2/overview/architecture/_index/#rancher-server-%E6%9E%B6%E6%9E%84
17 https://docs.pivotal.io/tkgi/1-8/index.html
18 https://github.com/hashicorp/nomad

Kubelet、Etcd。Etcd 组件在整个工作流中主要为其余四个组件提供组件状态存储，本书将不做解释，以下我们主要对其它四个组件的工作机制予以具体说明：

API Server

API Server 组件为各类 Kubernetes 资源对象的增删改查提供了 REST 接口，为贯穿整个 Kubernetes 系统的数据总线。

API Server 组件在如上介绍的 Pod 工作流中体现的功能可总结为以下几点：

1. API Server 为整个 Pod 工作流提供了资源对象（Pod, Deployment, Service 等）的增、删、改、查以及用于集群管理的 Rest API 接口，集群管理主要包括认证授权、集群状态管理和数据校验等。
2. API Server 提供集群中各组件的通信以及交互的功能。
3. API Server 提供资源配额控制入口功能。
4. API Server 提供访问控制功能。

Kubernetes 集群中，API Server 运行在 Master 节点上，默认开放两个端口，分别为本地端口 8080 和安全端口 6443，其中，非认证或授权的 http 请求通过 8080 端口访问 API Server；而 6443 端口用于接收 HTTPS 请求。Kubernetes 中默认不启动 HTTPS 安全访问控制。

API Server 在整个 Pod 工作流中主要负责各个组件间的通信，Scheduler, Controller Manager, Kubelet 通过 API Server 将资源对象信息存入 Etcd 中，当各组件需要这些数据时又通过 API Server 的 REST 接口来实现信息交互，API Server 主要与以下组件进行信息同步：

Kubelet 与 API Server

Pod 工作流中，位于集群中的每个 Node 节点上的 Kubelet 会定期调用 API Server 的 REST 接口以便向 Controller Manager 告知当前状态，API Sever 接收到状态信息后，将其更新至 Etcd 中，Kubelet 也同时通过 API Server 的 Watch 接口去监听 Pod 信息，从而对各个 Node 节点上的 pod 进行管理，监听信息与对应的 Kubelet 操作可汇总为表 0.1 监听信息表：

表 0.1 监听信息表

监听信息	Kubelet 执行
是否有新的 Pod 被绑定到 Node 节点	执行 Pod 对应容器的创建和启动
是否有 Pod 对象被删除	删除 Node 上相应 Pod 容器
是否修改了 Pod 信息	修改 Node 上 Pod 的信息

Controller-Manager 与 API Server

Controller Manager 包含许多控制器，例如 Endpoint Controller、Replication Controller、Service Account Controller 等，这些控制器通过 API Server 提供的接口实时监控当前集群中每个资源对象的状态变化并将最新的信息保存在 Etcd 中，当集群中发生各种故障导致系统发生变化时，各个控制器会从 Etcd 中获取资源对象信息并尝试将系统状态修复至理想状态。

Scheduler 与 API Server

Scheduler 通过 API Server 的 Watch 接口监听 Master 节点新建的 Pod 副本信息并检索所有符合该 Pod 要求的 Node 列表同时执行调度逻辑，成功后将 Pod 绑定在目标节点处。

由于在集群中各组件频繁对 API Server 进行访问，各组件采用了缓存机制来缓解请求量，各组件定时从 API Server 获取资源对象信息并将其保存在本地缓存中，所以组件大部分时间是通过访问缓存数据来获得信息的。

Controller Manager

Controller Manager 在整个 Pod 工作流起着管理和控制整个集群内部的作用，主要对资源对象进行管理，当 Node 节点中运行的 Pod 对象或是 Node 自身发生意外或故障时，Controller Manager 会及时发现并处理以确保整个集群处以理想工作状态。

Controller Manager 组成如图 0.7 所示：

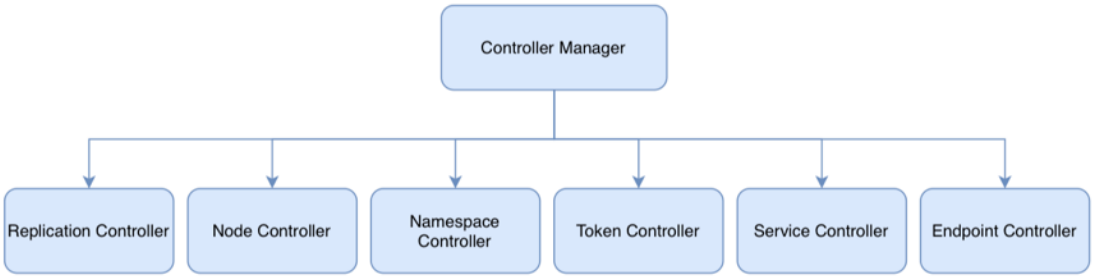

图 0.7 Controller Manager 组成图

由上图我们可以看出 Controller Manager 包含五个不同的控制器：

1. Replication Controller

Replication Controller 称为副本控制器，在 Pod 工作流中主要用于保证集群中的 Pod 副本数始终保持在预期值，若节点发生故障导致 Pod 被意外杀死，Replication Controller 会重新调度保证集群仍然运行指定副本数，另外还可通过调整 Replication Controller 中 spec.replicas 属性值来实现扩容或缩容。

2. Endpoints Controller

Endpoints 用来表示 Service 对应后端 Pod 副本的访问地址，Endpoints Controller 则是用来生成和维护 Endpoints 对象的控制器，其主要负责监听 Service 和对应 Pod 副本变化。若监测到 Service 被删除，则删除和该 Service 同名的 Endpoints 对象；若监测到新的 Service 被创建或是被修改，则根据该 Service 信息获得相关的 Pod 列表，然后创建或更新对应的 Endpoints 对象；若监测到 Pod 的事件，则更新其对应的 Service 的 Endpoints 对象。图 0.8 展示了 Service、Endpoint、Pod 的关系：

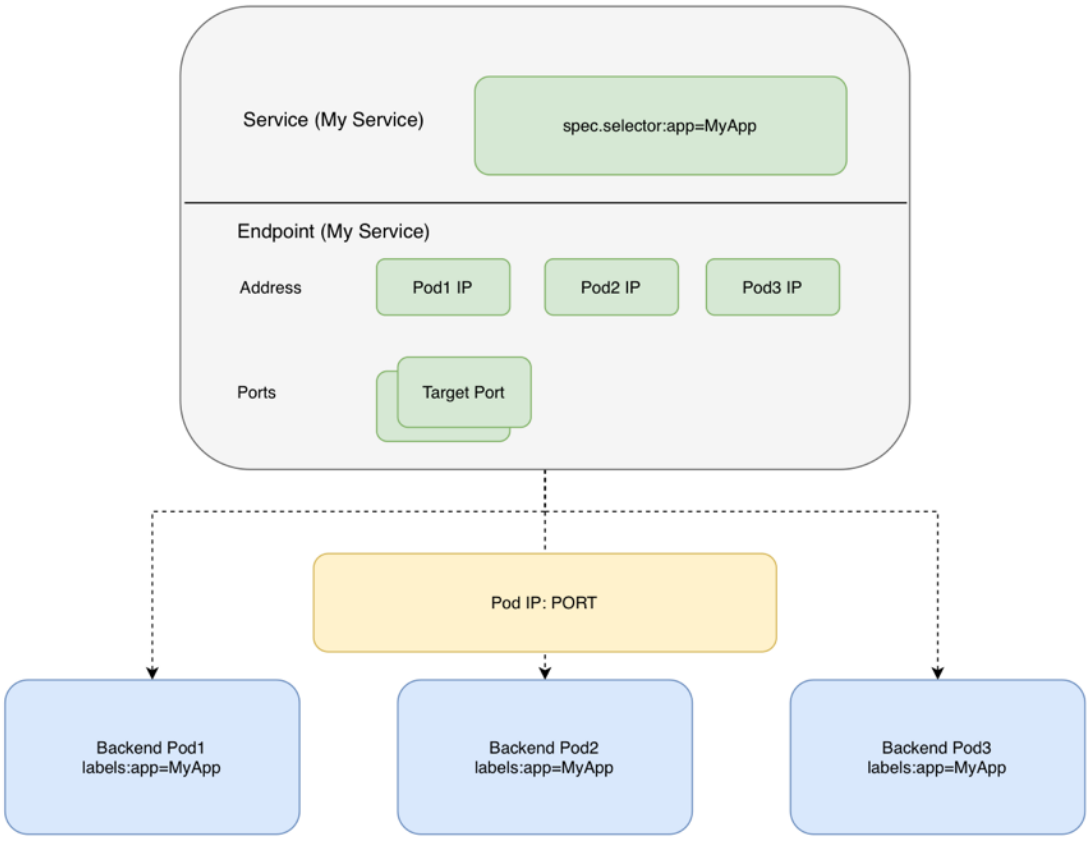

图 0.8 Service Endopoints Pod 关系图

3. Node Controller

主要负责集群中某节点宕机时对 API Server 进行通知和响应。

4. Service Account 和 Token Controller

主要负责在新的命名空间被部署时为其创建默认的 Service Account 账户及对应的 API 访问令牌。

Scheduler

Scheduler 在整个 Pod 工作流中负责将 Pod 调度至具体的 Node 节点，Scheduler 通过 API Server 监听 Pod 状态，若有待调度的 Pod 则根据 Scheduler Controller 中的预选策略和优选策略给各个预备 Node 节点打分排序，最后将 Pod 调度到分数最高的 Node 上，由 Node 中的 Kubelet 组件负责 Pod 的整个生命周期管理。

Kubelet

Kubelet 保证了其所在节点 Pod 的正常工作，当节点的 Pod 配置发生变化时，Kubelet 组件根据最新的配置执行相应的操作，从而保证 Pod 可在理想的预期状态。当 Pod 进行启停更新操作时，容器运行时负责对 Pod 中的容器进行生命周期管理。此外，Kubelet 还负责 Volume 插件（CVI）和网络插件（CNI）地管理。

Kubernetes 简明实践

本小节将以实验的方式为各位读者带来 Kubernetes 简明实践，由于 Kubernetes 通常使用 Deployment 资源来部署应用，因此本实验的操作过程将围绕 Deployment 资源进行展开。

实验环境

Kubernetes 版本：v1.16.2

Docker 版本：v19.03.3

主机操作系统：Ubuntu 18.04

单节点集群

Deployment 资源部署

本实验我们将部署一个简单的 Nginx 应用，由于后续实验需要对应用的版本进行操作，因此 Nginx 版本设定为 v1.7.9，我们通过 kubectl 可直接进行部署，如下所示：

```
root@k8s:~# kubectl run nginx --image=nginx:1.7.9
deployment.apps/nginx created
```

由于部署应用时未指定命名空间，因而 Nginx 应用被部署至默认的 default 命名空间，查看部署的 Nginx 应用：

```
root@k8s:~# kubectl get deployments.apps nginx
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   1/1     1            1           5m2s
```

如需查看部署的详细信息及过程，可以通过以下命令查看：

```
root@k8s:~# kubectl describe deploy nginx
Name:                   nginx
Namespace:              default
```

```
CreationTimestamp:      Mon, 25 Jan 2021 10:20:46 +0800
Labels:                 run=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               run=nginx
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=nginx
  Containers:
    nginx:
    Image:        nginx:1.7.9
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-b8dcdd49d (1/1 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  7m25s  deployment-controller  Scaled up replica set nginx-
b8dcdd49d to 1
```

查看 Nginx 的 ReplicaSet 资源

```
root@k8s:~# kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
nginx-b8dcdd49d   1         1         1       10m
```

以上输出可以看出 Nginx 副本数期望值（DESIRED）为 1，当前副本数（CURRENT）为 1，状态就绪（READY）的副本数为 1, Nginx 已启动时间（AGE）为 10 秒。

查看 Nginx 的 Pod 资源

```
root@k8s:~# kubectl get pod -o wide
NAME  READY  STATUS  RESTARTS  AGE  IP  NODE  NOMINATED NODE   READINESS GATES
nginx-b8dcdd49d-xkgqw   1/1     Running   0          17m   10.244.0.26   xian107   <none>
<none>
```

通过指定”-o wide”参数可以查看该 Pod 运行所处的节点及 IP。

Deployment 日志查看

通过 kubectl 可对 Nginx 应用日志进行查看，如下所示：

```
root@k8s:~# kubectl logs nginx-b8dcdd49d-xkgqw
```

此处由于 Nginx 应用没有标准输出文档，因此没有日志文件输出

Deployment 资源执行

与 Docker 类似，我们可通过 kubectl 进入 Nginx 应用对应的容器，如下所示：

```
root@k8s:~# kubectl exec -it nginx-b8dcdd49d-xkgqw bash
root@nginx-b8dcdd49d-xkgqw:/# nginx -v
nginx version: nginx/1.7.9
```

Service 资源部署

Nginx 应用部署完成后，我们可以通过部署 Service 资源让外部能够访问它，Service 的 yaml 配置如下：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx #为 nginx 应用打标签
spec:
    port:
    - name: http
      port: 8888 #集群内部访问端口
      nodePort: 30001#集群外部访问端口
      targetPort: 80 # 容器端口
    selector:    #选取 label 为 “run: nginx” 的对象进行识别
      run: nginx
    type: NodePort #通过指定 NodePort 端口类型使外部可对应用进行访问
```

接着部署 Service 资源，如下所示：

```
root@k8s:~# kubectl apply -f /tmp/nginx.svc.yml
service/nginx created
```

查看部署的 Service 资源：

```
root@k8s:~# kubectl get svc
NAME         TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
```

```
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP          46d
nginx        NodePort    10.105.36.12   <none>        8888:30001/TCP   30s
```

可以看出 Nginx 应用在集群内部可通过 8888 端口访问，外部可通过 30001 进行访问，我们尝试在浏览器中进行访问，如图 0.9 所示：

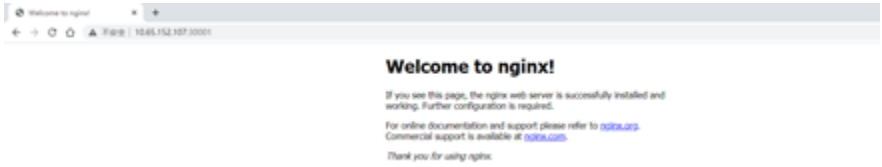

图 0.9 浏览器访问 Nginx

我们通过 kubectl 可以进一步得到 Service 和 Pod 的映射关系，如下所示：

```
root@k8s:~# kubectl get endpoints
NAME         ENDPOINTS           AGE
kubernetes   10.65.152.107:6443   46d
nginx        10.244.0.26:80       6m23s
```

可以看出当 Nginx 副本数为 1 时，对应的 endpoint 地址为 10.244.0.26：80。

Deployment 资源扩展

我们通过 kubectl 对 Nginx 应用的副本数进行扩展，此处假设副本数为 3，具体命令如下所示：

```
root@k8s:~# kubectl scale deployment nginx --replicas=3
deployment.apps/nginx scaled
```

扩展完成后，我们对 Nginx 应用的 Deployment、ReplicaSet、Pod、Endpoints 资源分别进行查看，如下所示：

```
root@k8s:~# kubectl get deployments.apps nginx
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   3/3     3            3           59m
```

```
root@k8s:~# kubectl get rs nginx-b8dcdd49d
NAME              DESIRED   CURRENT   READY   AGE
nginx-b8dcdd49d   3         3         3       59m
```

Nginx Deployment 资源的副本数由之前的 1 变为 3，并且均扩展成功，目前为就绪状态。

查看具体的三个 Pod 副本

```
root@k8s:~# kubectl get pod
NAME                READY   STATUS    RESTARTS   AGE
nginx-b8dcdd49d-bwbrr   1/1     Running   0          31s
nginx-b8dcdd49d-tmgxl   1/1     Running   0          31s
nginx-b8dcdd49d-xkgqw   1/1     Running   0          59m
```

查看 Service 与三个 Pod 副本间的映射关系

```
root@k8s:~# kubectl get ep
NAME         ENDPOINTS                                      AGE
kubernetes   10.65.152.107:6443                             46d
nginx        10.244.0.26:80,10.244.0.27:80,10.244.0.28:80   13m
```

Deployment 资源升级

我们可通过替换镜像版本进行 Nginx 应用的升级，此处我们将 Nginx 镜像版本由 v1.7.9 升级为 v1.9.1 并查看升级状态，如下所示：

```
root@k8s:~# kubectl rollout status deployment nginx
Waiting for deployment "nginx" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "nginx" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "nginx" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx" rollout to finish: 1 old replicas are pending termination...
deployment "nginx" successfully rolled out
```

以上输出说明版本已升级完毕，在升级过程中，若未对镜像仓库进行配置，Kubernetes 则默认从 Dockerhub 仓库中拉取镜像，拉取完成后 Kubernetes 自动启动 3 个新镜像版本的 Pod 资源，同时旧镜像版本的 Pod 资源将会被逐个删除，从而实现应用的平滑升级。

此外，我们可对应用的版本升级历史进行查看，如下所示：

```
root@k8s:~# kubectl rollout history deployment.apps nginx
deployment.apps/nginx

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

Deployment 资源回滚

我们在实际操作中很可能有因配置失误而导致升级失败的情况，例如我们再次升级 Nginx 的镜像版本至 v1.9.5，在输入命令行错输成了 v1.95，并进行了版本升级，这会导致以下输出：

```
root@k8s:~# kubectl set image deploy nginx nginx=nginx:1.95
deployment.apps/nginx image updated
```

```
root@k8s:~# kubectl rollout status deployment nginx
Waiting for deployment "nginx" rollout to finish: 1 out of 3 new replicas have been updated...
```

等待数分钟之后，依然输出 “Waiting for deployment "nginx" rollout to finish: 1 out of 3 new replicas have been updated...”，我们可通过 kubectl 对 pod 信息进行查看，如下所示：

```
root@k8s:~# kubectl get pods
NAME                    READY   STATUS             RESTARTS   AGE
nginx-797b47466d-4c768   0/1     ImagePullBackOff   0          5m25s
nginx-7d94864c84-f8r9p   1/1     Running            0          18m
nginx-7d94864c84-hgqmq   1/1     Running            0          24m
nginx-7d94864c84-njxrz   1/1     Running            0          18m
```

可以看名为 “nginx-797b47466d-4c768” 的 Pod 未正常启动，状态为 “ImagePullBackOff”，这是因为版本号的拼写错误使镜像无法成功拉取，进而导致旧的版本无法终止，新的版本也无法进行更新。

此时，需要注意的是，虽然新版本无法正常启动，但我们通过 curl 命令发现旧版本的 Nginx 依然能够访问，因此即使升级失败，但对于用户层面而言对此次更新失败是无感知的。

```
root@k8s:~# curl 10.65.152.107:30001
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

```
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

下面，我们通过以下操作进行版本回滚：

```
root@k8s:~# kubectl rollout undo deploy nginx
deployment.apps/nginx rolled back
root@k8s:~# kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
nginx-7d94864c84-f8r9p   1/1     Running   0          26m
nginx-7d94864c84-hgqmq   1/1     Running   0          32m
nginx-7d94864c84-njxrz   1/1     Running   0          25m
```

很快的，Nginx 恢复到了之前的状态。

资源删除

Kubernetes 可通过 kubectl delete <resource>或 kubectl delete –f <xxx.yaml>这两种方式删除已部署的资源，本实践中，由于我们使用 kubcetl run 的方式启动 Nginx 应用，因此可通过 kubectl delete <resource>的方式删除对应的 Deployment 资源，如下所示：

```
root@k8s:~# kubectl delete deployments.apps nginx
deployment.apps "nginx" deleted
```

```
root@k8s:~# kubectl get pod
No resources found in default namespace.
```

Service 资源是通过 kubectl apply 的方式启动，因此可通过 kubectl delete –f <xxx.yaml>的方式删除，如下所示：

```
root@k8s:~# kubectl delete -f /tmp/nginx.svc.yml
service "nginx" deleted
```

```
root@k8s:~# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   47d
```

资源自愈

如果我们删除 Nginx 副本中某的一个 Pod，Kubernetes 会自动为我们重启一个 Nginx Pod，以恢复到前面我们声明的 Pod 数量状态，如下所示：

```
root@k8s:~# kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
nginx-7d94864c84-f8r9p   1/1     Running   0          74m
nginx-7d94864c84-hgqmq   1/1     Running   0          81m
nginx-7d94864c84-njxrz   1/1     Running   0          74m

root@k8s:~# kubectl delete pod nginx-7d94864c84-f8r9p
pod "nginx-7d94864c84-f8r9p" deleted

root@k8s:~# kubectl get pods
NAME                READY   STATUS              RESTARTS   AGE
nginx-7d94864c84-f8r9p   1/1     Terminating         0          75m
nginx-7d94864c84-hgqmq   1/1     Running             0          81m
nginx-7d94864c84-njxrz   1/1     Running             0          74m
nginx-7d94864c84-rl25f   0/1     ContainerCreating   0          3s
```

故障信息排查

通常我们可通过 kubectl logs <container_id>或 kubectl describe <pod name>的方式对故障信息进行排查，例如上述 0 小节中，因错写了 Nginx 版本号导致 Pod 无法正常启动，此时我们可通过 kubectl describe 进行故障查看，如下所示：

```
root@k8s:~# kubectl describe pod nginx-797b47466d-gl9qk
Name:         nginx-797b47466d-gl9qk
Namespace:    default
... ...
Events:
  Type     Reason     Age              From               Message
  ----     ------     ----             ----               -------
  Normal   Scheduled  <unknown>        default-scheduler  Successfully assigned default/nginx-797b47466d-gl9qk to xian107
  Normal   Pulling    53s (x3 over 111s)  kubelet, xian107   Pulling image "nginx:1.95"
  Warning  Failed     42s (x3 over 106s)  kubelet, xian107   Failed to pull image "nginx:1.95": rpc error: code = Unknown desc = Error response from daemon: manifest for nginx:1.95 not found: manifest unknown: manifest unknown
  Warning  Failed     42s (x3 over 106s)  kubelet, xian107   Error: ErrImagePull
  Normal   BackOff    13s (x4 over 105s)  kubelet, xian107   Back-off pulling image "nginx:1.95"
  Warning  Failed     13s (x4 over 105s)  kubelet, xian107   Error: ImagePullBackOff
```

可以看出，因 Dockerhub 中不存在 manifest 为 nginx:1.95 的镜像，因而无法被正常下载。

小结

本附录首先为各位读者介绍了各类容器编排平台，进而对业界容器编排标准 Kubernetes 组件进行了相应解读，最后通过简明实践部分介绍了 Kubernetes 基本的使用方法，容器编排平台为下层容器的运行提供了编排能力，为上层微服务、服务网格、Serverless 的实现提供了基础设施，在云原生技术中起着承上启下的重要作用。