第 1 章 Kubernetes 组件概念

1.1 云计算概念

云计算技术其实是将硬件设备、操作系统、软件服务、网络带宽、流量、计费系统等资源组成一个大的资源池（动态扩容、弹性伸缩），然后可以再将所有的资源池分配给租户使用，租户可以根据自身的需求，按需购买资源。

云计算技术强调的是资源池，是租户的概念。虚拟化技术是云计算技术框架中的一个小模块、组件技术。云计算技术最终的产物包括硬件设备、操作系统、软件服务、网络带宽等。每个产物都可以租给用户使用，用户可以自行购买。

对于云计算技术的资源池，租户不需要了解云计算底层框架、架构，只要清楚自身对资源池的需求，需要多少台服务器、多少云主机、多大网络带宽，最终按需付费即可。

1.2 云计算技术的分类

云计算技术按照实现方式分为 3 种类型：基础设施即服务（Infrastructure as a Service，IaaS）、平台即服务（Platform as a Service，PaaS）、软件即服务（Software as a Service，SaaS），如图 1-1 所示。每种云的特点如下。

（1）基础设施即服务。

① 租户无须管理底层硬件设备、网络、服务器、存储、虚拟化技术。

② 租户只需对操作系统、中间件、数据、应用做维护即可。

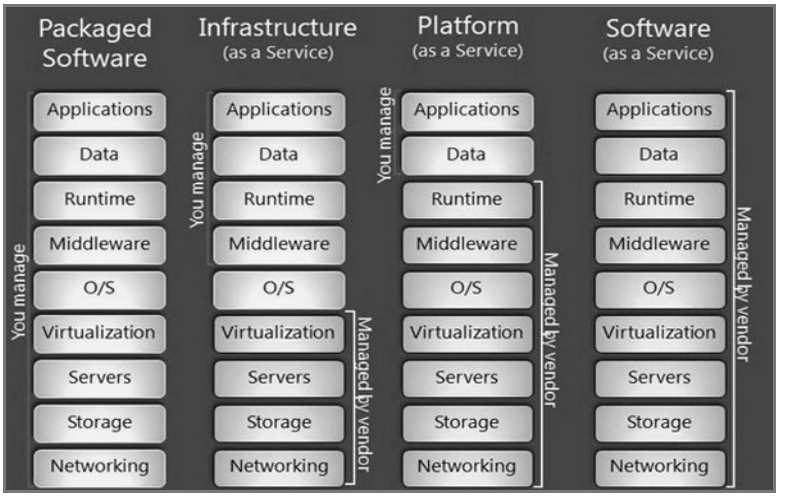


图 1-1　云计算技术分类

（2）平台即服务。

① 租户无须管理底层硬件设备、网络、服务器、存储、虚拟化技术、操作系统、中间件。

② 租户只需对应用服务、软件程序做维护，无须操作系统和底层设施。

（3）软件即服务。

① 租户无须管理底层硬件设备、网络、服务器、存储、虚拟化技术、操作系统、中间件、应用服务、软件程序等。

② 租户只需花钱、付费，提交业务需求，运营商将满足租户所有需求。

1.3　Kubernetes 入门及概念介绍

Kubernetes 又称为 K8S（首字母为 K，首字母与尾字母之间有 8 个字符。尾字母为 S，所以简称 K8S）或者简称为 kube，是一种可以自动实施 Linux 容器操作的开源平台。

Kubernetes 可以帮助用户省去应用容器化过程中的许多手动部署和扩展操作。我们可以将运行 Linux 容器的多组主机聚集在一起，由 Kubernetes 帮助用户轻松高效地管理这些集群，而且这些集群可跨公共云、私有云或混合云部署主机。因此，对于要求快速扩展的云原生应用而言（例如，借助 Apache Kafka 进行的实时数据流处理），Kubernetes 是理想的托管平台。

Kubernetes 最初由 Google 公司的工程师开发和设计。Google 是最早研发 Linux 容器技术的企业之一（组建了 cgroups），曾公开分享介绍 Google 如何将一切都运行于容器之中（这是 Google

云服务背后的技术）。Google 每周会启用超过 20 亿个容器——全都由内部平台 Borg 支撑。Borg 是 Kubernetes 的前身，多年来开发 Borg 的经验教训成了影响 Kubernetes 中许多技术的主要因素。

在企业生产环境中，应用会涉及多个容器（主机），这些容器必须跨多个服务器主机进行部署。容器安全性需要多层部署，因此可能比较复杂。但 Kubernetes 有助于解决这一问题。Kubernetes 可以提供所需的编排和管理功能，以便针对这些工作负载大规模部署容器。借助 Kubernetes 编排功能，可以构建跨多个容器的应用服务、跨集群调度、扩展这些容器，并长期持续管理这些容器的健康状况。

Kubernetes（K8S）是自动化容器操作的开源平台，这些操作包括部署、调度和节点集群间扩展。如果用户曾经用过 Docker 容器技术部署容器，可以将 Docker 看成 Kubernetes 内部使用的低级别组件。Kubernetes 不仅支持 Docker，还支持 Rocket，这是另一种容器技术。使用 Kubernetes 可以实现如下功能：

（1）自动化容器的部署和复制。

（2）跨多台主机进行容器编排和管理。

（3）有效管控应用部署和更新，并实现自动化操作。

（4）挂载和增加存储，用于运行有状态的应用。

（5）能够快速、按需扩展容器化应用及其资源。

（6）对服务进行声明式管理，保证部署的应用始终按照部署的方式运行。

（7）更加充分地利用硬件，最大程度获取运行企业应用所需的资源。

（8）利用自动布局、自动重启、自动复制及自动扩展功能，对应用实施状况进行检查和自我修复。

1.4　Kubernetes 平台组件概念

Kubernetes 集群中主要存在两种类型的节点：master 节点和 node 节点。

1）master 节点

master 节点主要负责对外提供一系列管理集群的 API 接口，并且通过与 node 节点交互实现对集群的操作管理。以下为 master 节点的组件和用途。

（1）Apiserver：用户和 Kubernetes 集群交互的入口，封装了核心对象的增、删、改、查操作，提供了 RESTFul 风格的 API 接口，通过 etcd 实现持久化并维护对象的一致性。

（2）Scheduler：负责集群资源的调度和管理。例如，当有 Pod 异常退出需要重新分配机器

时，Scheduler 通过一定的调度算法找到最合适的节点。

（3）controller-manager：主要用于保证 replication controller 定义的复制数量和实际运行的 Pod（容器）数量一致，并保证从 service 到 Pod 的映射关系总是最新的。

2）node 节点

node 节点主要负责接收 master 发送的指令，同时与本地 Docker 引擎交互等操作。以下为 node 节点的组件和用途。

（1）kubelet：运行在 node 节点，负责与节点上的 Docker 交互。例如，启停容器、监控运行状态等。

（2）kube-proxy：运行在 node 节点上，负责为 Pod 提供代理功能，会定期从 etcd 获取 service 信息，并根据 service 信息通过修改 iptables 实现流量转发（最初的版本是直接通过程序提供转发功能，效率较低），将流量转发到要访问的 Pod 所在的节点上。

Kubernetes 云计算平台除了 master 和 node 节点上相应的组件模块之外，还需要 etcd、Flannel、Docker 等插件。以下为相关插件的用途。

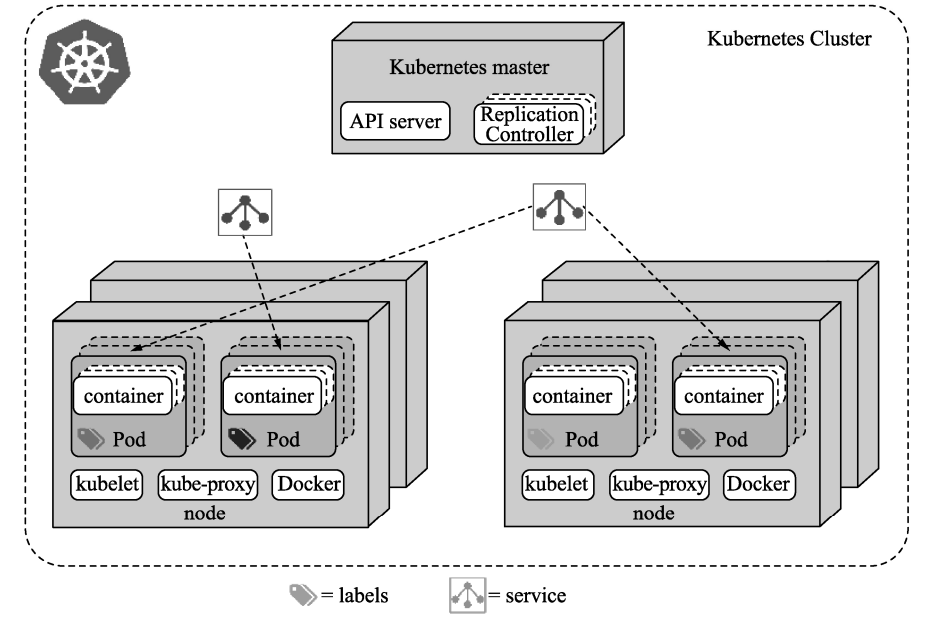
（1）etcd：etcd 是一个分布式一致性 key-value 存储系统数据库，可用于注册与发现配置共享和服务，用来存储 Kubernetes 的信息。etcd 组件作为一个高可用、强一致性的服务发现存储仓库，渐渐为开发人员所关注。在云计算时代，如何让服务快速透明地接入计算集群，如何让共享配置信息快速被集群中的所有机器发现，更为重要的是，如何构建这样一套高可用、安全、易于部署以及响应快速的服务集群？etcd 的诞生就是为了解决该问题。

（2）Flannel：Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个覆盖网络（Overlay Network）工具，目的是为集群中的所有节点重新制定 IP 地址的使用规则，从而使不同节点上的容器能够获得同属一个内网且不重复的 IP 地址，并让属于不同节点上的容器能够直接通过内网 IP 通信。

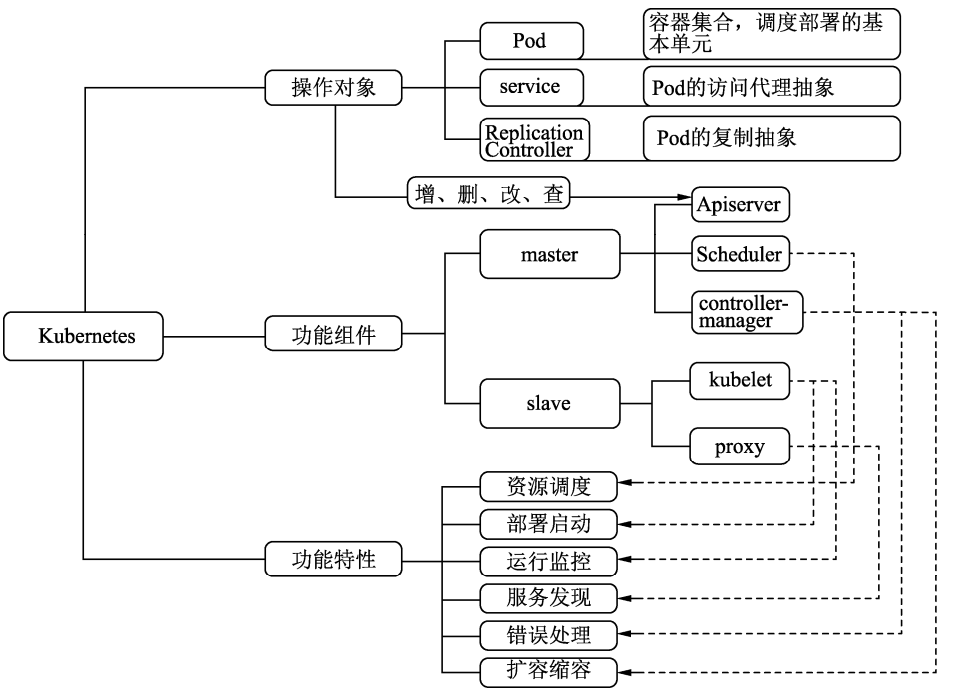
（3）Docker：Docker 是一款轻量级的虚拟化软件，主要是为了解决企业轻量级服务器操作系统和应用容器而诞生的，在 Kubernetes 集群中，Docker 被看成 Kubernetes 集群中的低级别组件，负责接收 node 节点上 kubelet 组件进行交互操作，如启动、停止、删除 Docker 容器，监控 Docker 容器运行状态等。

1.5 Kubernetes 工作原理剖析

Kubernetes 集群是一组节点，这些节点可以是物理服务器或者虚拟机，在其上安装 Kubernetes 平台。Kubernetes 云计算架构如图 1-2 所示，图中为了强调核心概念有所简化。

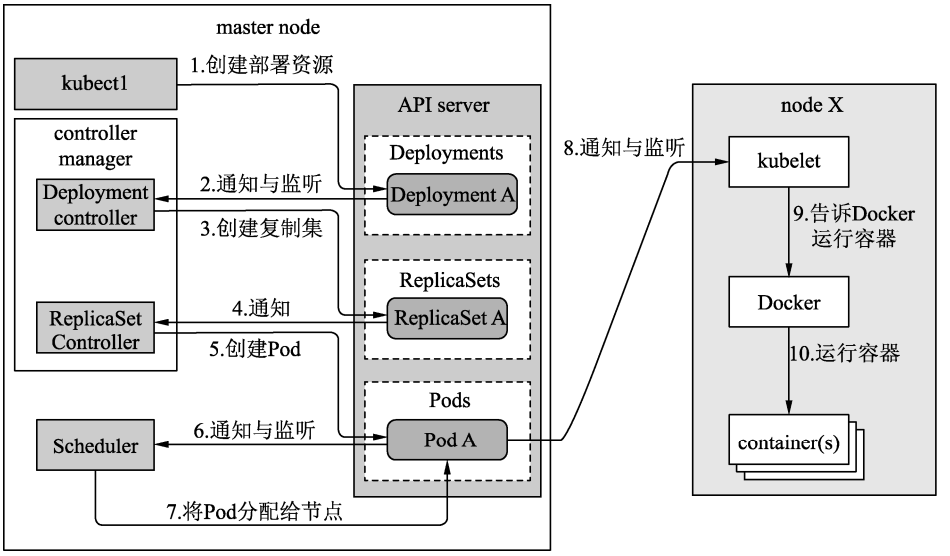


（a）



（b）

图 1-2　Kubernetes 云计算架构



（c）

图 1-2 （续）

图 1-2（a）中相关模块和组件的名称如下：

（1）Pod（容器组）。

（2）container（容器）。

（3）labels()（标签）。

（4）service（）（服务）。

（5）node（Kubernetes 计算节点）。

（6）Replication Controller（副本控制器）。

（7）Kubernetes master（Kubernetes 主节点）。

从图 1-2 可以看到 Kubernetes 组件和逻辑关系：Kubernetes 集群主要由 master 和 node 两类节点组成；master 的组件包括 API server、controller-manager、Scheduler 和 etcd 等，其中 API server 是整个集群的网关。

Kubernetes node 主要由 kubelet、kube-proxy、Docker 引擎等组成。kubelet 是 Kubernetes 集群的工作与节点上的代理组件。

在企业生产环境中，完整的 Kubernetes 集群还包括 CoreDNS、Prometheus（或 HeapSter）、Dashboard、Ingress Controller、cAdvisor 等几个附加组件。其中 cAdvisor 组件作用于各个节点（master 和 node 节点），用于收集容器和节点的 CPU、内存以及磁盘资源的利用率指标数据，这些统计

数据由 Heapster 聚合后，可以通过 API server 访问。

Kubernetes 集群中创建一个资源（Pod 容器）的工作流程和步骤如下。

（1）客户端提交创建（Deployment、Namespace、Pod）请求，可以通过 API server 的 Restful API 提交，也可以使用 kubectl 命令行工具提交。

（2）通过 API server 处理用户的请求，并将相关数据（Deployment、Namespace 和 Pod）存储到 etcd 配置数据库中。

（3）Kubernetes Scheduler 调度器通过 API server 查看未绑定的 Pod，并尝试为该 Pod 分配 node 主机资源。

（4）过滤主机（调度预选）：调度器用一组规则过滤掉不符合要求的主机。例如，Pod 指定了所需要的资源量，那么可用资源比 Pod 需要的资源量少的主机就会被过滤掉。

（5）主机打分（调度优选）：对第一步筛选出的符合要求的主机进行打分。在主机打分阶段，调度器会考虑一些整体优化策略，例如，把 Replication Controller 的副本分布到不同的主机上，使用最低负载的主机等。

（6）选择主机：选择打分最高的主机，进行 binding 操作，将结果存储到 etcd 中。

（7）node 节点上的 kubelet 根据调度结果，调用主机上的 Docker 引擎执行 Pod 创建操作，绑定成功后，Scheduler 会调用 API server 的 API 在 etcd 中创建一个 boundpod 对象，描述在一个工作节点上绑定运行的所有 Pod 信息。

（8）同时运行在每个工作节点上的 kubelet 也会定期与 etcd 同步 boundpod 信息，一旦发现应该在该工作节点上运行的 boundpod 对象没有更新，将调用 Docker API 创建并启动 Pod 内的容器。

1.6　Pod 概念剖析

Pod（容器组）是 Kubernetes 中最小的可布署单元了，一般部署在 node 节点上，包含一组容器和卷。同一个 Pod 中的容器共享同一个网络命名空间，可以使用 localhost 互相通信，Pod 是短暂的，不是持续性实体。关于 Pod 的常见问题和解答如下：

（1）Pod 一般是短暂的，如何才能持久化容器数据，使其能够跨重启而存在呢？Kubernetes 支持卷的概念，因此可以使用持久化的卷类型。

（2）如果要创建同一个容器的多份副本，需要逐个创建出来吗？可以手动创建单个 Pod，也可以使用 Replication Controller 中的 Pod 模板创建出多份副本。

（3）Pod 是短暂的，重启 Pod 时 IP 地址可能会改变，那么怎样才能从前端容器正确可靠地指向后台容器呢？答案是可以使用 service。

1.7 label 概念剖析

Kubernetes label 是标记到 Pod 的一个键/值对，用来传递用户定义的属性。例如，用户可能创建了一个 tier 和 app 标签，可通过 label（tier=frontend, app=jfedu-app）标记前端 Pod 容器，使用 label（tier=backend, app=jfedu-app）标记后台 Pod。

可以使用 Selectors 选择带有特定 label（标签）的 Pod，并且将 service 或者 Replication Controller 应用于这些 Pod。

1.8 Replication Controller 概念剖析

Replication Controller 确保任意时间都有指定数量的 Pod“副本”在运行。如果为某个 Pod 创建了 Replication Controller 并指定 3 个副本，它会创建 3 个 Pod，并持续监控它们。如果某个 Pod 不响应，那么 Replication Controller 会替换它，并保持总数为 3，如图 1-3 所示。

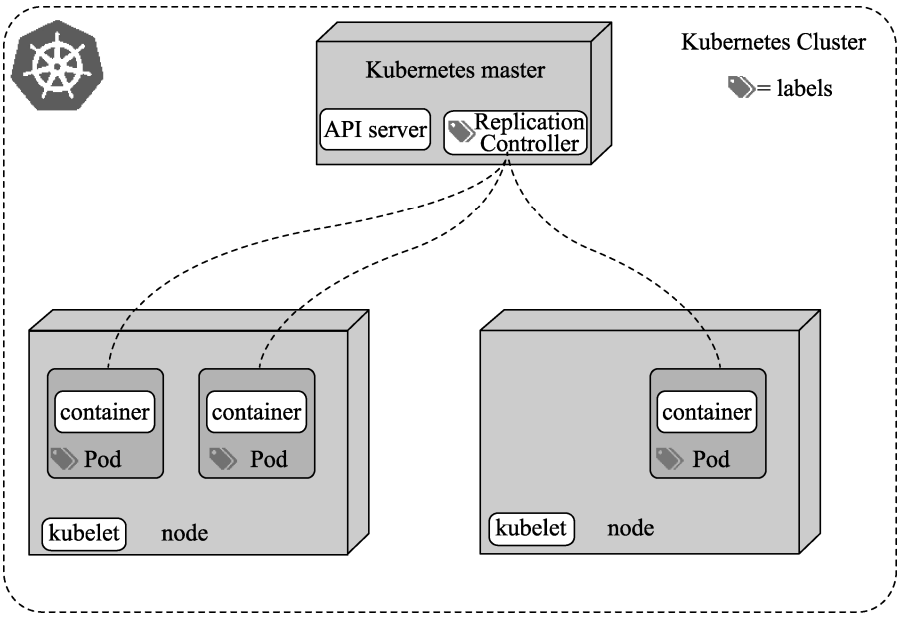


图 1-3　Replication Controller 副本结构

如果之前不响应的 Pod 恢复，就有了 4 个 Pod，那么 Replication Controller 会将其中一个 Pod 终止，以保持总数为 3。如果在运行中将副本总数改为 5，Replication Controller 会立刻启动 2 个新 Pod，保证总数为 5。还可以按照这样的方式减少 Pod 副本，这个特性在执行滚动升级时很有用。当创建 Replication Controller 时，需要指定以下两个内容。

（1）Pod 模板：用来创建 Pod 副本的模板。

（2）label：Replication Controller 需要监控的 Pod 的标签。

1.9　service 概念剖析

service 是定义一系列 Pod 以及访问这些 Pod 的策略的一层抽象。service 通过 label 找到 Pod 组。因为 service 是抽象的，所以通常看不到它们的存在，这就让这一概念更难以理解。Kubernetes service 内部结构如图 1-4 所示。

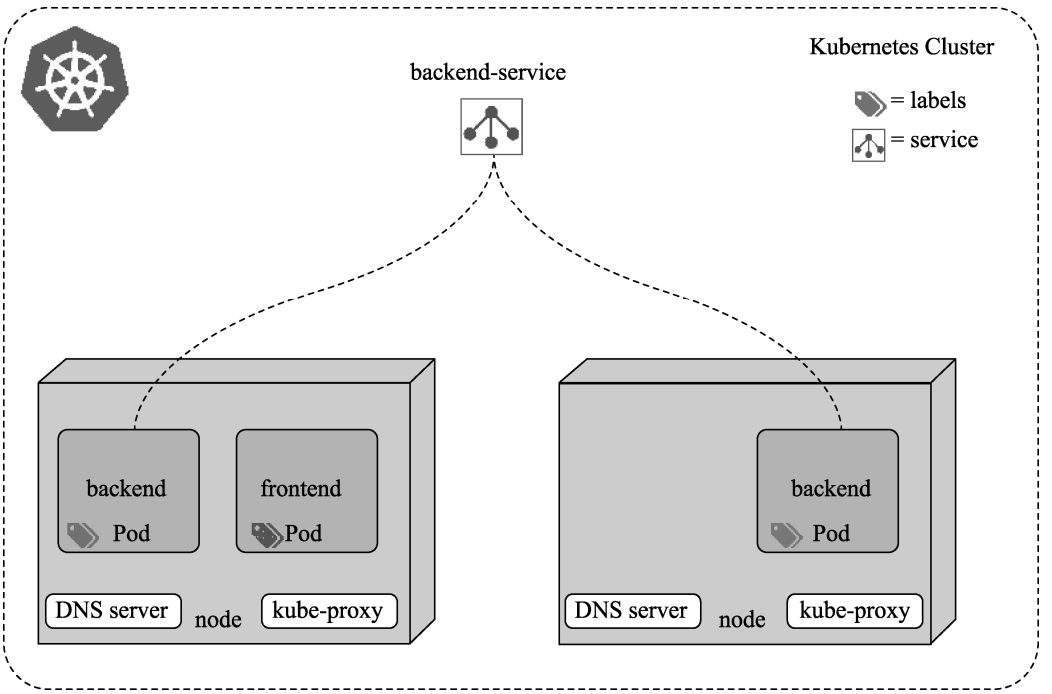


图 1-4　Kubernetes service 内部结构

假设创建了 2 个 Pod 容器，并定义后台 service 的名称为 backend-service，label 选择器为

（tier=backend, app=jfedu-app），backend-service 的 service 会完成如下操作：

（1）会为 service 创建一个本地集群的 DNS 入口，因此前端 Pod 只需要 DNS 查找主机名为 backend-service，就能够解析出前端应用程序可用的 IP 地址。

（2）service 为创建的 2 个 Pod 容器提供透明的负载均衡，并将请求分发给其中的任意一个，通过每个 node 上运行的代理（kube-proxy）完成。

1.10 node 概念剖析

node（节点）是物理或虚拟机器，作为 Kubernetes worker，通常称为计算节点。每个节点都运行以下 Kubernetes 关键组件。

（1）kubelet：是 node 节点上的主程序，负责接收 master 发送的指令。

（2）kube-proxy：service 使用 kube-proxy 将请求转发至 Pod。

（3）Docker 或 Rocket：Kubernetes 使用 Docker 或 Rocket 容器技术创建容器。

1.11 Kubernetes volume 概念剖析

在 Docker 中有 volume 这个概念，volume 只是磁盘上或其他容器中的简单目录，生命周期不受管理，并且直到最近都是基于本地后端存储的。Docker 现在也提供了 volume driver，但是功能较弱。

Kubernetes 的 volume 有着明显的生命周期——和使用它的 Pod 生命周期一致。因此，volume 生命周期就比运行在 Pod 中的容器要长久，即使容器重启，volume 上的数据依然保存着。当 Pod 不再存在时，volume 也就消失了。更重要的是，Kubernetes 支持多种类型的 volume，且 Pod 可以同时使用多种类型的 volume。

在 Kubernetes 内部实现中，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。这个目录是如何产生的？它后端基于什么存储介质？其中的数据内容是什么？这些都由使用的特定 volume 类型决定。

要使用 volume，Pod 需要指定 volume 的类型和内容（spec.volumes 字段），以及映射到容器的位置（spec.containers.volumeMounts 字段）。

容器中的进程可以看到 Docker image 和 volumes 组成的文件系统。Docker image 处于文件系

统架构的 root，任何 volume 都映射在镜像的特定路径上。volume 不能映射到其他 volume 上，也不能硬链接到其他 volume。容器中的每个容器必须指定它们要映射的 volume。

Kubernetes 支持多种类型的 volume，包括 emptyDir、hostPath、gcePersistentDisk、awsElastic-BlockStore、nfs、iscsi、flocker、glusterfs、rbd、cephfs、gitRepo、secret、persistentVolumeClaim 等。

1.12　Deployment 概念剖析

Deployment 为 Pod 和 ReplicaSet 提供声明式更新和部署，只需要在 Deployment 中描述想要的目标状态，Deployment controller 就会将 Pod 和 ReplicaSet 的实际状态改变到目标状态。

可以定义一个全新的 Deployment 创建 ReplicaSet，或者删除已有的 Deployment 并创建一个新的替换。

注意：无须手动管理由 Deployment 创建的 ReplicaSet，否则就越俎代庖了。

1.13　DaemonSet 概念剖析

DaemonSet 对象能确保其创建的 Pod 在集群中的每一台（或指定）node 上都运行一个副本。如果集群中动态加入了新的 node，DaemonSet 中的 Pod 也会被添加到新加入的 node 上。删除一个 DaemonSet 也会级联删除所有其创建的 Pod。DaemonSet 和 Deployment 的区别如下。

（1）Deployment 部署 Pod 会分布在各个 node 上，node 可能运行好几个副本。

（2）DaemonSet 部署 Pod 会分布在各个 node 上，每个 node 只能运行一个 Pod。

以下为 DaemonSet 的使用场景：

（1）在每台节点上运行一个集群存储服务，如运行 glusterd、ceph。

（2）在每台节点上运行一个日志收集服务，如 fluentd、logstash。

（3）在每台节点上运行一个节点监控服务，如 Prometheus node exporter、collectd、Datadog agent、New Relic agent 或 Ganglia gmond。

1.14　StatefulSet 概念剖析

Kubernetes RC、Deployment 和 DaemonSet 都是面向无状态的服务，它们所管理的 Pod 的 IP、

名字、启停顺序等都是随机的，而 StatefulSet 是什么？顾名思义，StatefulSet 是有状态的集合，管理所有有状态的服务，如 MySQL、MongoDB 集群等。

StatefulSet 本质上是 Deployment 的一种变体，在 v1.9 版本中已成为 GA 版本。为了解决有状态服务的问题，它所管理的 Pod 拥有固定的 Pod 名称、启停顺序，在 StatefulSet 中，Pod 名字称为网络标识（hostname），还必须用到共享存储。

Deployment 中的服务是 service，而 StatefulSet 中的服务是 headless service，即无头服务，与 service 的区别就是后者没有 Cluster IP，解析其名称时将返回该 headless service 对应的全部 Pod 的 Endpoint 列表。

1.15 ConfigMap 概念剖析

在企业生产环境中，经常会遇到需要修改配置文件的情况。例如，Web 网站连接数据库的配置、业务系统之间相互调用配置、Kubernetes Pod 配置信息等，如果使用传统手动方式修改，则不仅会影响到服务的正常运行，操作步骤也很烦琐。

对于传统的应用服务而言，每个服务都有自己的配置文件，各自配置文件存储在服务所在的节点。对于单体应用，这种存储没有任何问题，但是随着用户数量的激增，一个节点不能满足线上用户的使用，故服务可能从一个节点扩展到十个节点，这就导致，如果有一个配置出现变更，就需要对应修改十次配置文件。

这种人工处理方式显然不能满足线上部署要求，故引入了各种类似于 ZooKeeper 中间件实现的配置中心，但配置中心属于“侵入式”设计，需要修改引入第三方类库，它要求每个业务都调用特定的配置接口，破坏了系统本身的完整性。

Kubernetes 利用 volume 功能完整设计了一套配置中心，其核心对象就是 ConfigMap，使用过程中不用修改任何原有设计，即可无缝对接 ConfigMap。

Kubernetes 项目从 1.2.0 版本开始引入了 ConfigMap 功能，主要用于将应用的配置信息与程序分离。这种方式不仅可以实现应用程序被复用，还可以通过不同的配置实现更灵活的功能。

在创建容器时，用户可以将应用程序打包为容器镜像，通过环境变量或外接挂载文件的方式进行配置注入。ConfigMap 非常灵活，相当于把配置文件信息单独存储在某处，需要时直接引用、挂载即可。

ConfigMap 以 key:value（K-V）的形式保存配置项，既可以表示一个变量的值（例如，

config=info），也可以表示一个完整配置文件的内容（例如，server.xml=<?xml…>…）。ConfigMap 在容器中使用的典型用法如下：

（1）将配置项设置为容器内的环境变量。

（2）将启动参数设置为环境变量。

（3）以 volume 的形式挂载到容器内部的文件或目录。

可以基于 kubectl create 指令创建 ConfigMap。例如，使用 ConfigMap 存储 MySQL 数据库的 IP 地址和端口信息，如图 1-5 所示。

```
kubectl create configmap mysql-config --from-literal=db.host=192.168.1.111
--from-literal=db.port=3306
kubectl get configmap mysql-config -o yaml
```

```
[root@node1 ~]# kubectl get configmap mysql-config -o yaml
apiVersion: v1
data:
  db.host: 192.168.1.111
  db.port: "3306"
kind: ConfigMap
metadata:
  creationTimestamp: "2021-03-31T06:38:50Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:db.host: {}
        f:db.port: {}
    manager: kubectl-create
    operation: Update
    time: "2021-03-31T06:38:50Z"
  name: mysql-config
  namespace: default
  resourceVersion: "75563"
  uid: 0118d3de-7558-42ac-9b59-f35edb02b29c
```

图 1-5　Kubernetes ConfigMap 配置界面

1.16　Secrets 概念剖析

在 Kubernetes 集群中，Secrets 通常用于存储和管理一些敏感数据，如密码、token、密钥等敏感信息。它把 Pod 想要访问的加密数据存放到 etcd 中。

用户可以通过在 Pod 容器里挂载 volume 或环境变量的方式访问这些 Secret 里保存的信息。Secret 有以下 3 种类型。

（1）Opaque：用来存储密码、密钥等，采用 base64 编码格式。但数据也可以通过 base64–decode 解码得到原始数据，加密性很弱。

（2）Service Account：用来访问 Kubernetes API，由 Kubernetes 自动创建，会自动挂载到 Pod

的 /run/secrets/kubernetes.io/serviceaccount 目录中。

（3）kubernetes.io/dockerconfigjson：用来存储私有 Docker registry 的认证信息。

1.17 CronJob 概念剖析

CronJob 用于创建基于时间调度的任务（Jobs）。一个 CronJob 对象就像 Crontab (Cron table) 文件中的一行。它用 Cron 格式进行编写，并周期性地在给定的调度时间执行。所有 CronJob 的 schedule 都是基于 kube-controller-manager 的时区。

如果 Kubernetes 在 Pod 或裸容器中运行了 kube-controller-manager，那么为该容器所设置的时区将会决定 CronJob 的控制器所使用的时区。

为 CronJob 资源创建清单时，请确保所提供的名称是一个合法的 DNS 子域名，名称不能超过 52 个字符。这是因为 CronJob 控制器将自动在提供的任务名称后附加 11 个字符，且存在一个限制，即任务名称的最大长度不能超过 63 个字符。

CronJob 对于创建周期性的、多次重复的任务很有用。例如，执行数据备份或者发送邮件。CronJob 也可以计划在指定时间执行的独立任务。例如，计划当集群看起来很空闲时执行某个任务。

（1）CronJob 案例会在每分钟打印出当前时间和问候消息，代码如下，如图 1-6 所示。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
```

```
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

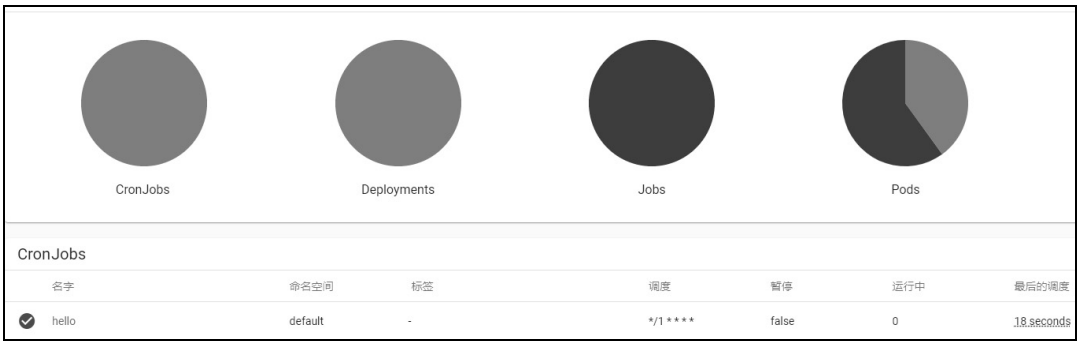


图 1-6　Kubernetes CronJob 任务创建

（2）CronJob 时间表语法如下所示。

```
# ┌──────────────────────── 分 (0 ~ 59)
# │ ┌────────────────────── 时 (0 ~23)
# │ │ ┌──────────────────── 日 (1 ~31)
# │ │ │ ┌────────────────── 月 (1 ~12)
# │ │ │ │ ┌──────────────── 星期 (0 ~6) (星期日到星期一;在某些系统上,
# 7 也是星期日)
# │ │ │ │ │
# │ │ │ │ │
# │ │ │ │ │
# * * * * *
```

1.18　Kubernetes 证书剖析和制作实战

Kubernetes 需要公钥基础设施（Public Key Infrastructure，PKI）证书才能进行基于安全传输层协议（Transport Layer Security，TLS，主要用于在两个通信应用程序之间提供保密性和数据完整性）的身份验证。如果用户是通过 kubeadm 安装的 Kubernetes，则会自动生成集群所需的证书。用户还可以生成自己的证书。例如，不将私钥存储在 API 服务器上，可以让私钥更加安全。如果是通过 kubeadm 安装的 Kubernetes，则所有证书都存放在 /etc/kubernetes/pki 目录下。本文所有相关的路径都是基于该路径的相对路径。

PKI 采用证书进行公钥管理，通过第三方的可信任机构（认证中心，即 CA），把用户的公钥和其他标识信息捆绑在一起，其中包括用户名和电子邮件地址等信息，以在互联网上验证用

户的身份。PKI 把公钥密码和对称密码结合起来，在互联网上实现密钥的自动管理，保证网上数据的安全传输。

Kubernetes 集群中需要的证书环节如下：

（1）etcd 对外提供服务，需要一套 etcd server 证书。

（2）etcd 各节点之间进行通信，需要一套 etcd peer 证书。

（3）kube-apiserver 访问 etcd，需要一套 etcd client 证书。

（4）kube-apiserver 对外提供服务，需要一套 kube-apiserver server 证书。

（5）kube-scheduler、kube-controller-manager、kube-proxy、kubelet 和其他可能用到的组件，需要访问 Kube-apiserver，需要一套 kube-apiserver client 证书。

（6）kube-controller-manager 要生成服务的 service account，需要一套用来签署 service account 的证书（CA 证书）。

（7）kubelet 对外提供服务，需要一套 kubelet server 证书。

（8）kube-apiserver 需要访问 kubelet，需要一套 kubelet client 证书。

同一个套里的证书必须是用同一个 CA 签署的，签署不同套里的证书的 CA 可以相同，也可以不同。例如，所有 etcd server 证书需要是同一个 CA 签署的，所有的 etcd peer 证书也是同一个 CA 签署的，而一个 etcd server 证书和一个 etcd peer 证书完全可以是两个 CA 签署的，彼此没有任何关系，这里就算是两套证书。

虽然可以用多套证书，但是维护多套 CA 实在过于繁杂，这里还是用一个 CA 签署所有证书。

（1）需要准备的证书如下：

```
admin.pem
ca.-key.pem
ca.pem
admin-key.pem
kube-scheduler-key.pem
kube-scheduler.pem
kube-controller-manager-key.pem
kube-controller-manager.pem
kube-proxy-key.pem
kube-proxy.pem
kubernetes-key.pem
kubernetes.pem
```

（2）使用证书的组件如下：

① etcd：ca.pem、kubernetes-key.pem、kubernetes.pem。

② Kube-apiserver：ca.pem、ca-key.pem、kubernetes-key.pem、kubernetes.pem。

③ Kubelet：ca.pem。

④ kube-proxy：ca.pem、kube-proxy-key.pem、kube-proxy.pem。

⑤ kubectl：ca.pem、admin-key.pem、admin.pem。

⑥ kube-controller-manager：ca-key.pem、ca.pem、kube-controller-manager-key.pem、kube-controller-manager.pem。

⑦ kube-scheduler：kube-scheduler-key.pem、kube-scheduler.pem。

此处使用 CFSSL 制作证书，CFSSL 是 Cloudflare 开发的一个开源的 PKI 工具，是一个完备的 CA 服务系统，可以签署、撤销证书等，覆盖了一个证书的整个生命周期。后面只用到了它的命令行工具。

注：一般情况下，Kubernetes 中的证书只需要创建一次，以后在向集群中添加新节点时只要将/etc/kubernetes/ssl 目录下的证书复制到新节点上即可。

```
wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
wget https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64
chmod +x cfssl_linux-amd64 cfssljson_linux-amd64 cfssl-certinfo_linux-
amd64
mv cfssl_linux-amd64 /usr/local/bin/cfssl
mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
mv cfssl-certinfo_linux-amd64 /usr/bin/cfssl-certinfo
```

（3）创建 CA 证书，配置文件，操作指令如下：

```
cat>ca-config.json<<EOF
{
  "signing": {
    "default": {
      "expiry": "87600h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
            "signing",
            "key encipherment",
            "server auth",
```

```
            "client auth"
          ],
          "expiry": "87600h"
        }
      }
    }
}
EOF
```

（4）Kubernetes 证书配置文件参数剖析。

① ca-config.json：可以定义多个 profiles，指定不同的过期时间、使用场景等参数。后续在签名证书时使用某个 profile。

② signing：可以签名其他证书，生成的 ca.pem 证书中 CA=TRUE。

③ server auth：client 可以用该 CA 对 server 提供的证书进行验证。

④ client auth：server 可以用该 CA 对 client 提供的证书进行验证。

⑤ expiry：设置证书过期时间。

（5）创建 CA 证书签名请求文件，操作指令如下：

```
cat>ca-csr.json<<EOF
{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "Kubernetes",
      "OU": "System"
    }
  ],
    "ca": {
       "expiry": "87600h"
    }
}
EOF
```

（6）CA 证书签名请求文件配置参数剖析。

① CN：Common Name，Kube-apiserver 从证书中提取该字段作为请求的用户名（User Name）。浏览器使用该字段验证网站是否合法。

② O：Organization，Kube-apiserver 从证书中提取该字段作为请求用户所属的组（Group）。

（7）生成 CA 证书和私钥，操作指令如下：

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca
ls | grep ca
ca-config.json
ca.csr
ca-csr.json
ca-key.pem
ca.pem
```

其中，ca-key.pem 是 CA 的私钥；ca.csr 是一个签署请求；ca.pem 是 CA 证书，是后面 Kubernetes 组件会用到的 RootCA。

（8）创建 Kubernetes 证书，并创建 Kubernetes 证书签名请求文件，操作指令如下：

```
cat>kubernetes-csr.json<<EOF
{
    "CN": "kubernetes",
    "hosts": [
      "127.0.0.1",
      "192.168.1.145",
      "192.168.1.146",
      "etcd01",
      "kubernetes",
      "kube-api.jd.com",
      "kubernetes.default",
      "kubernetes.default.svc",
      "kubernetes.default.svc.cluster",
      "kubernetes.default.svc.cluster.local"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
```

```
            "ST": "BeiJing",
            "L": "BeiJing",
            "O": "Kubernetes",
            "OU": "System"
        }
    ]
}
EOF
```

如果 hosts 字段不为空，则需要指定授权使用该证书的 IP 地址或域名列表。由于该证书后续被 etcd 集群和 Kubernetes master 集群使用，将 etcd、master 节点的 IP 地址都填上，同时还有 service 网络的首 IP 地址(一般是 Kube-apiserver 指定的 service-cluster-ip-range 网段的第一个 IP 地址，如 10.0.0.1)。以上物理节点的 IP 地址也可以更换为主机名。

（9）生成 Kubernetes 证书和私钥，操作指令如下：

```
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes
ls |grep kubernetes
kubernetes.csr
kubernetes-csr.json
kubernetes-key.pem
kubernetes.pem
```

（10）创建 admin 证书，并创建 admin 证书签名请求文件，操作指令如下：

```
cat>admin-csr.json<<EOF
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "system:masters",
      "OU": "System"
    }
  ]
```

```
}
EOF
```

Kube-apiserver 使用 RBAC 对客户端（如 kubelet、kube-proxy、Pod）请求进行授权。Kube-apiserver 预定义了一些 RBAC 使用的 RoleBindings。例如，cluster-admin 将 Group system:masters 与 Role cluster-admin 绑定，该 Role 授予了调用 Kube-apiserver 的所有 API 的权限。

O 指定该证书的 Group 为 system:masters，Kubelet 使用该证书访问 Kube-apiserver 时，由于证书被 CA 签名，所以认证通过，同时由于证书用户组为经过预授权的 system:masters，所以被授予访问所有 API 的权限。

注：这个 admin 证书用于生成管理员用的 kube config 配置文件，一般建议使用 RBAC 对 Kubernetes 进行角色权限控制，Kubernetes 将证书中的 CN 字段作为 User，O 字段作为 Group。

（11）生成 admin 证书和私钥，操作指令如下：

```
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes admin-csr.json | cfssljson -bare admin
ls | grep admin
admin.csr
admin-csr.json
admin-key.pem
admin.pem
```

（12）创建 kube-proxy 证书及 kube-proxy 证书签名请求文件，操作指令如下：

```
cat>kube-proxy-csr.json<<EOF
{
  "CN": "system:kube-proxy",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "Kubernetes",
      "OU": "System"
    }
```

```
    ]
}
EOF
```

Kube-apiserver 预定义的 RoleBinding system:node-proxier 将 User system:kube-proxy 与 Role system:node-proxier 绑定，该 Role 授予了调用 Kube-apiserver Proxy 相关 API 的权限。CN 指定证书的 User 为 system:kube-proxy。

因为该证书只会被 kubectl 当作 client 证书使用，所以 hosts 字段为空，生成 kube-proxy 证书和私钥。

```
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes  kube-proxy-csr.json | cfssljson -bare kube-proxy
ls |grep kube-proxy
kube-proxy.csr
kube-proxy-csr.json
kube-proxy-key.pem
kube-proxy.pem
```

（13）创建 kube-controller-manager 证书，并创建其证书签名请求文件，操作指令如下：

```
cat>kube-controller-manager-csr.json<<EOF
{
    "CN": "system:kube-controller-manager",
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "hosts": [
      "127.0.0.1",
      "192.168.1.145",
      "192.168.1.146",
      "Kubernetes-master1",
    ],
    "names": [
      {
        "C": "CN",
        "ST": "BeiJing",
        "L": "BeiJing",
        "O": "system:kube-controller-manager",
        "OU": "system"
      }
```

```
    ]
}
EOF
```

其中，hosts 列表包含所有 kube-controller-manager 节点 IP；CN 为 system:kube-controller-manager；O 为 system:kube-controller-manager。

（14）生成 kube-controller-manager 证书和私钥，操作指令如下：

```
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes kube-controller-manager-csr.json | cfssljson -bare
kube-controller-manager
```

（15）创建 kube-scheduler 证书签名请求文件，操作指令如下：

```
cat>kube-scheduler-csr.json<<EOF
{
    "CN": "system:kube-scheduler",
    "hosts": [
      "127.0.0.1",
      "192.168.1.145",
      "192.168.1.146",
      "Kubernetes-master1",

    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
      {
        "C": "CN",
        "ST": "BeiJing",
        "L": "BeiJing",
        "O": "system:kube-scheduler",
        "OU": "4Paradigm"
      }
    ]
}
EOF
```

（16）根据以上创建的 CA 证书和 JSON 文件，接下来创建 Kubernetes 各个服务所需证书，操作指令如下：

```
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -
profile=kubernetes kube-scheduler-csr.json| cfssljson -bare kube-scheduler
ls | grep pem
admin-key.pem
admin.pem
ca-key.pem
ca.pem
kube-proxy-key.pem
kube-proxy.pem
kubernetes-key.pem
kubernetes.pem
kube-controller-manager-key.pem
kube-controller-manager.pem
kube-scheduler-key.pem
kube-scheduler.pem
```

（17）查看证书信息，操作指令如下：

```
cfssl-certinfo -cert kubernetes.pem
```

部署 Kubernetes 云计算平台时，将这些证书文件分发至此集群中其他节点机器对应目录即可。至此，TLS 证书创建完毕。