SOFT130091.01

云原生软件技术

5. 容器编排 Kubernetes

复旦大学软件学院

沈立炜

shenliwei@fudan.edu.cn

编排

- 编排（Orchestration）
  - ✓编排是一个广泛的概念，涵盖了整个应用程序生命周期的管理，是诸如Kubernetes等平台对应用程序整体的自动化管理和协调过程
  - ✓编排涉及定义应用程序的部署方式、扩展策略、更新机制、故障恢复、服务发现、网络策略、存储卷管理等多个方面

Docker Swarm与Kubernetes之争

- 容器编排领域的竞争主要聚焦在Docker Swarm和Kubernetes项目上
- 除了容器，用户还希望平台还以提供路由网关、水平扩展、监控、备份、灾难恢复等一系列运维能力，而Docker公司在PaaS层上的能力与Google、RedHat等公司有一定差距
- Google和RedHat等公司牵头发起CNCF，期望以Kubernetes为基础，建立一个由开源基础设施领域的厂商主导、按照独立基金会运营的平台及社区

Docker Swarm与Kubernetes之争

- Kubernetes凭借其先进的设计理念和号召力，构建出与众不同的容器编排和管理的生态体系，在GitHub上各项指标远超Swarm项目
- Docker公司于2016年宣布放弃Swarm项目，将容器编排、集群管理和负载均衡功能全部内置到Docker项目中，与Docker的无缝集成可以实现优势最大化，但同时也增加了技术复杂度和维护难度
- 2017年10月，Docker公司宣布在其主打产品Docker企业版中内置Kubernetes，宣告Kubernetes胜出

什么是K8s

- Kubernetes源自于希腊语，意为**舵手**或**飞行员**
- K8s是通过将8个字母“ubernete”替换为8而导出的缩写

Photo by Julius Silver from Pexels

什么是K8s

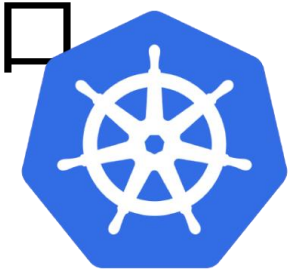
- Kubernetes 或 K8s 是一个从 Google 分离出来的项目，作为开源的下一代容器调度程序，其设计吸取了开发和管理 Borg 和 Omega 的经验教训
- 声明式配置: 允许用户描述他们希望系统处于的状态，而不是指定系统如何进行操作。这种模型使得系统更加可靠，因为它可以自动处理状态和配置的变化，而不需要用户手动介入
- 自动化: 通过自动化任务调度、资源分配和故障恢复等方面来简化操作，提高了系统的可用性和可靠性。自动化使得系统更具弹性，并且可以更快速地适应变化的条件

什么是K8s

- 灵活性和可扩展性: Kubernetes设计了一个灵活的插件架构，使得用户可以根据自己的需求进行扩展和定制，以适应不断增长和变化的需求。
- 服务发现和负载均衡: Kubernetes提供了内置的服务发现机制和负载均衡器，使得用户可以轻松地管理和连接其应用程序中的各个服务，并且可以有效地分配流量以实现高可用性和性能。

K8s做什么

- Kubernetes 是从头开始设计的，是一个松散耦合的组件集合，以部署、维护和扩展应用程序为中心
- Kubernetes可被视为一种分布式系统的操作系统内核
- 它抽象了节点的底层硬件，并为要部署和消耗共享资源池的应用程序提供了统一的接口



K8s的核心功能

- Kubernetes是自动化的容器编排平台
  - ✓部署
  - ✓弹性
  - ✓管理

- 核心功能
  - ✓服务发现与负载均衡
  - ✓容器调度
  - ✓存储编排
  - ✓自动容器恢复
  - ✓自动发布与回滚
  - ✓配置与密文管理
  - ✓批量执行
  - ✓水平伸缩

K8s - 调度

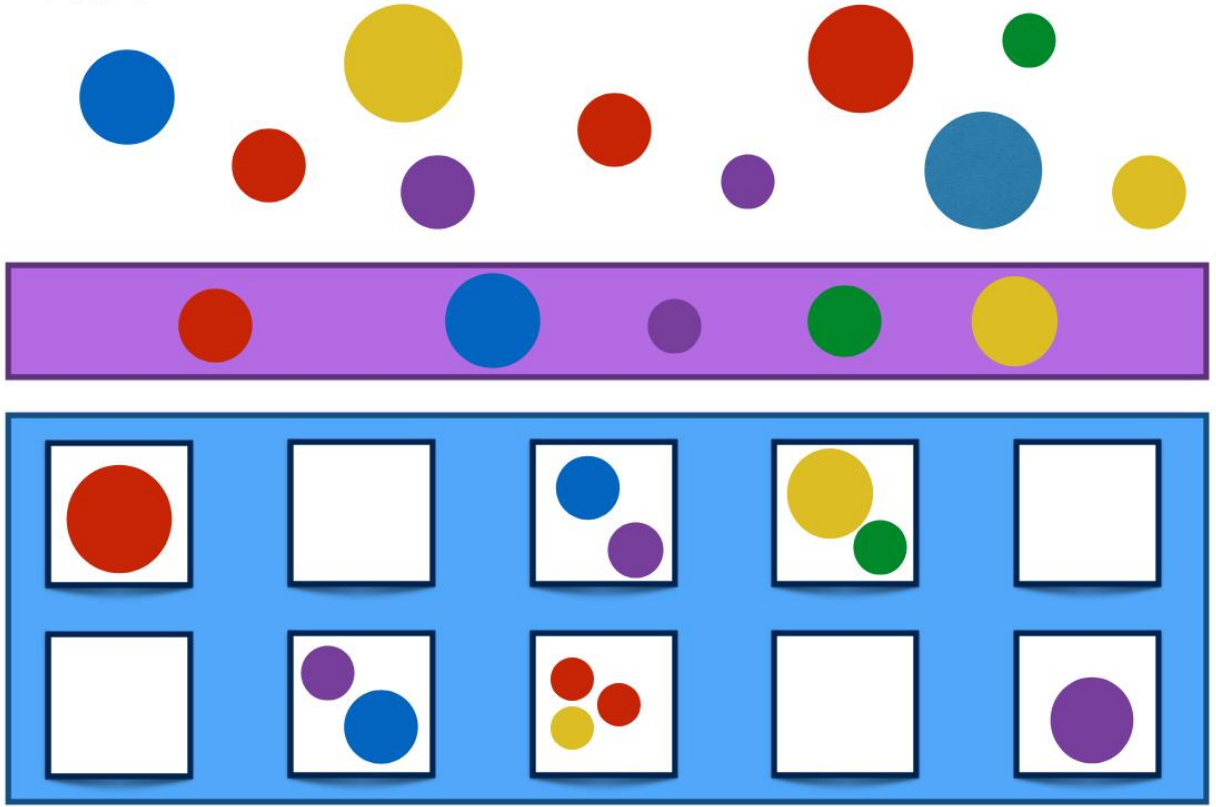
- 调度（Scheduling）
  - ✓调度是 Kubernetes 中的一个过程，负责将应用程序的 Pods（容器组）分配到集群中的合适节点上运行
  - ✓调度器（Scheduler）根据节点的资源可用性、Pod 的资源需求、节点的选择器约束以及其他调度策略来决定每个 Pod 应该部署在哪个节点上
  - ✓编排包括调度，但不仅限于调度

K8s - 调度

等待调度

正在调度

集群状态



K8s - 自动恢复

- 自动恢复是指系统自动检测并修复应用程序运行过程中可能出现的各种故障，以确保应用程序的高可用性和稳定性
- 自动恢复机制是 k8s 的核心特性之一，它能够在无需人工干预的情况下，快速响应并处理故障，减少服务中断时间，提高系统的整体可靠性

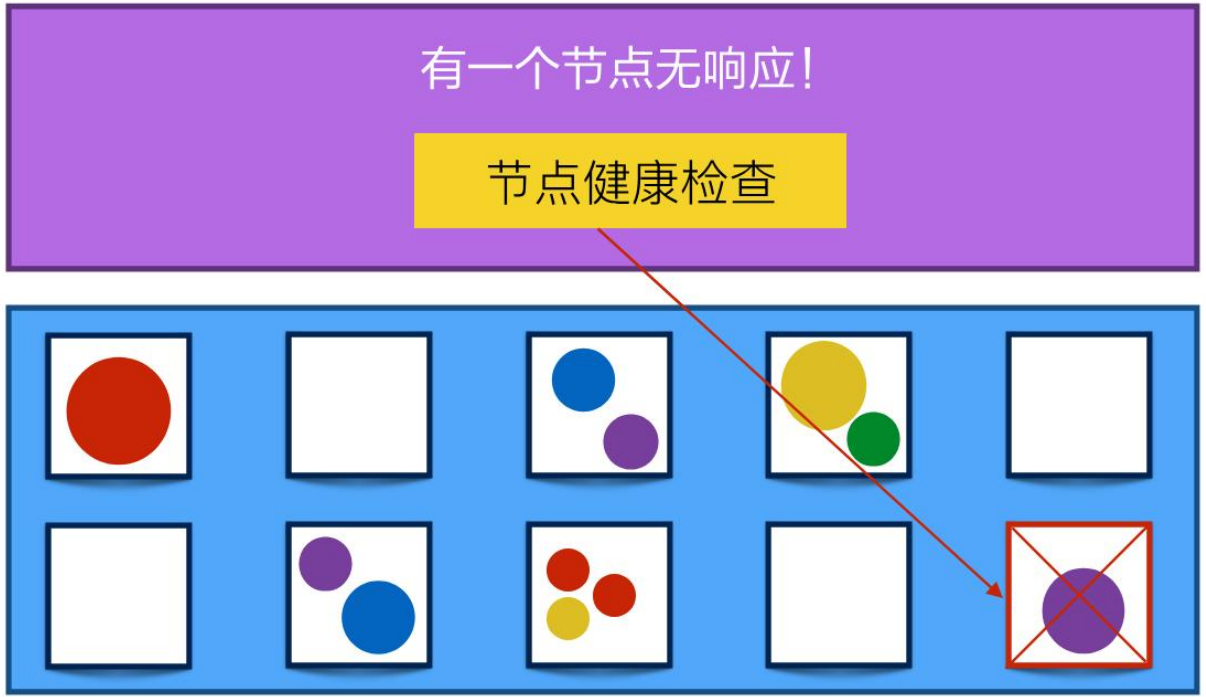
K8s - 自动恢复

- 自动恢复的实现机制
  - ✓容器重启：当容器出现故障时，如因程序错误或资源耗尽导致容器崩溃，k8s 会根据 Pod 的 restartPolicy 设置，自动重启该容器。restartPolicy 有三种可选值：Always（无论容器退出状态如何都重启）、OnFailure（仅在容器异常退出时重启）和 Never（从不重启）
  - ✓Pod 重调度：如果节点出现故障，如宕机或网络故障，k8s 会将该节点上运行的 Pod 自动重新调度到其他健康的节点上运行，以确保应用程序的连续性
  - ✓健康检查：通过 Liveness 和 Readiness 探针，k8s 能够检测容器的运行状态和是否准备好接收流量。如果 Liveness 探针检测到容器不健康，k8s 会自动重启该容器；如果 Readiness 探针检测到容器尚未准备好，k8s 会暂时停止向该容器发送流量，直到它准备好为止

K8s - 自动恢复

状态检查与修复

集群状态



K8s - 水平伸缩

- 水平伸缩是指根据应用的负载情况，自动增加或减少Pod的数量，以保持应用的性能和可用性
- 这种机制允许集群中的工作负载（如 Deployments、ReplicaSets 和 StatefulSets）根据实际的负载情况自动调整 Pod 的数量，从而优化资源的使用和提高服务的响应能力

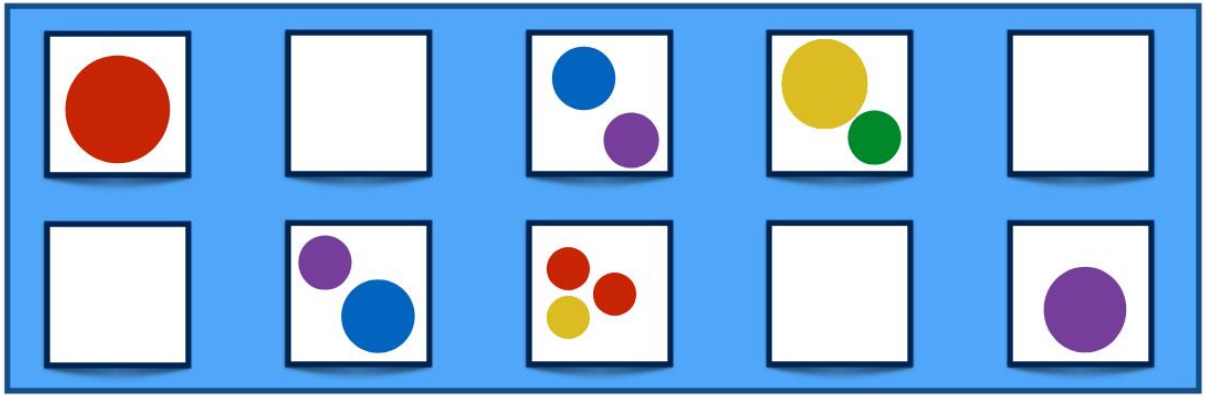
K8s - 水平伸缩

- 监控指标
  - ✓水平伸缩通过 Kubernetes Metrics Server 或其他自定义的指标（如 Prometheus），定期获取当前的负载情况。常见的指标包括 CPU 使用率、内存使用率、自定义的应用程序指标等
- 调整策略
  - ✓根据定义的策略，水平伸缩会判断是否需要增加或减少 Pod 的数量。例如，如果 CPU 使用率超过预设的阈值，水平伸缩会增加 Pod 的数量；如果 CPU 使用率低于预设的阈值，水平伸缩会减少 Pod 的数量

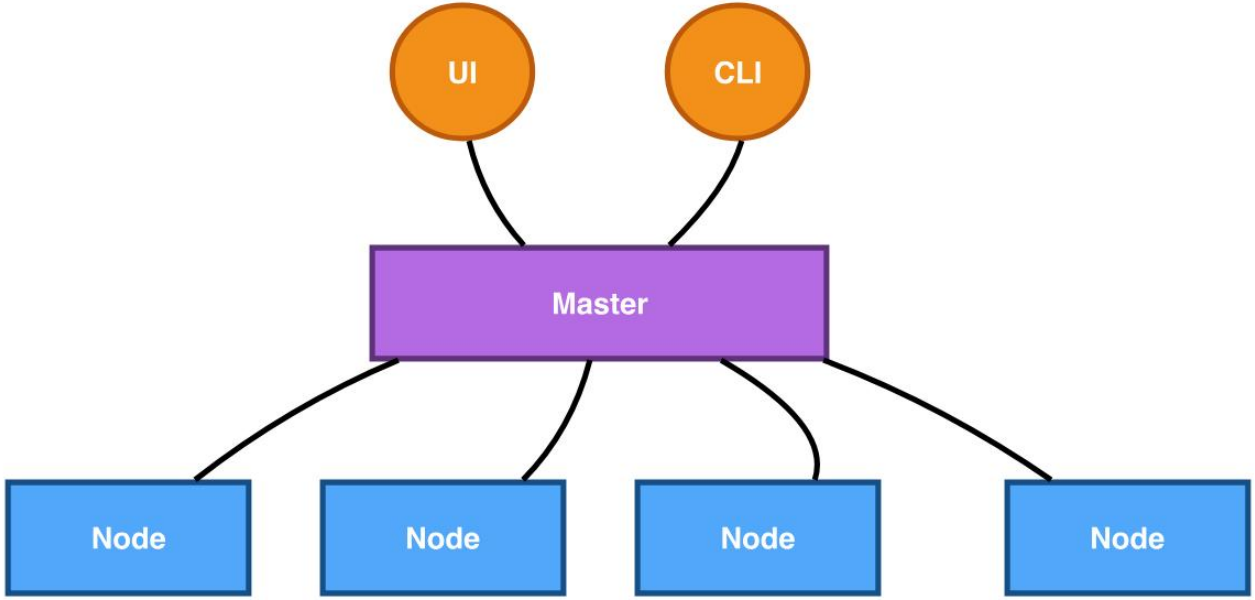
K8s - 水平伸缩

状态检查与修复

集群状态



K8s的架构



- Kubernetes 架构是一个比较典型的二层架构和 server-client 架构
- Master 作为中央的管控节点，与 Node 进行连接
- 所有UI、clients等user侧组件只与Master进行连接，把希望的状态或者想执行的命令下发给Master，Master把这些命令或者状态下发给相应的节点，进行最终执行

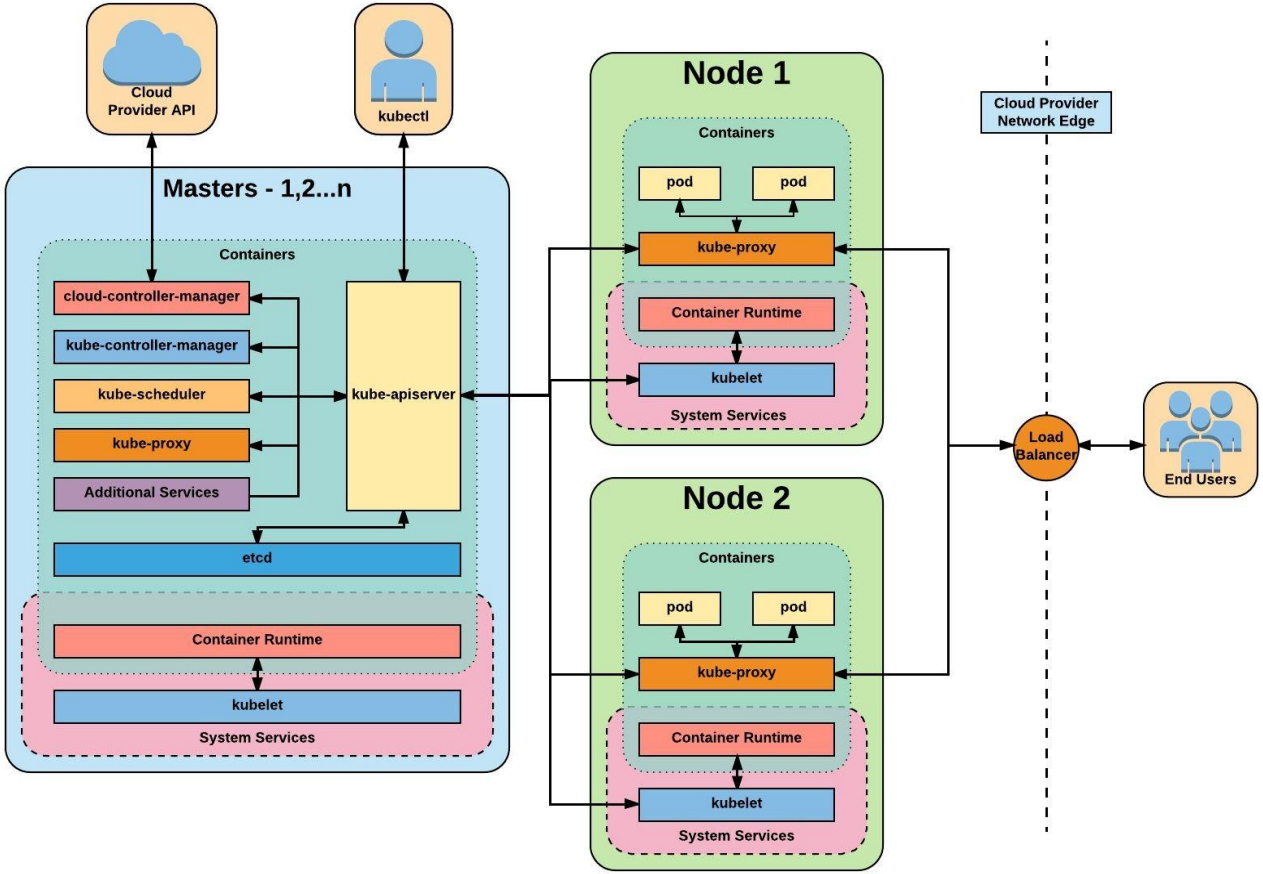
控制面与数据面

- 控制面（Control Plane）
  - ✓控制面是云原生架构的管理和决策中心，负责整个系统的协调和控制。它包含一系列的组件，用于处理用户的请求、配置系统的状态、调度资源、执行策略以及监控系统的运行状况等。控制面的主要任务是确保系统按照预期的方式运行，并在出现异常时进行调整和恢复
- 数据面（Data Plane）
  - ✓数据面是云原生架构中实际执行数据处理和业务逻辑的部分。它负责接收、处理和转发数据，执行具体的业务操作，以及与外部系统进行交互。数据面通常由多个分布式组件构成，这些组件在控制面的指导下协同工作，以实现高效的数据处理和传输

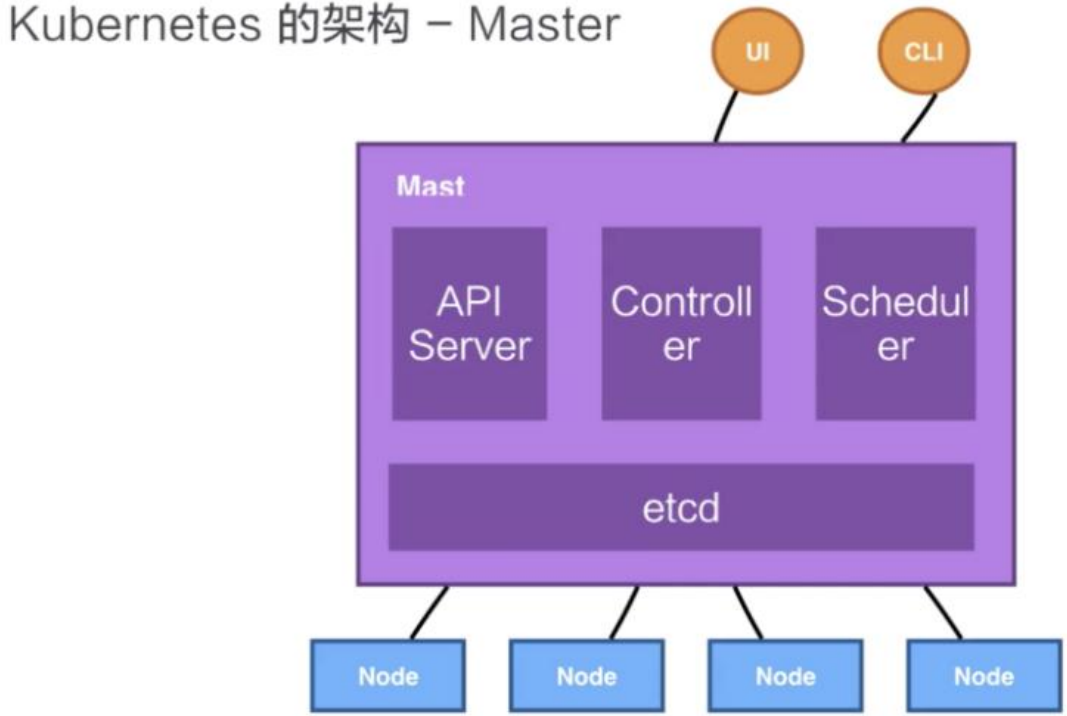
控制面与数据面

方面	控制面	数据面
功能职责	负责管理和决策，如配置、调度、监控等	负责数据处理和业务执行，如接收、处理、转发数据等
组件构成	包含API Server、etcd、Controller Manager、Scheduler等	包含Kubelet、Kube-proxy、容器运行时等
运行位置	通常集中部署在少数几个节点上，作为系统的管理核心	分布式部署在多个工作节点上，直接与用户和业务系统交互
稳定性要求	对稳定性和可靠性要求极高，因为其故障会影响整个系统	相对而言稳定性要求稍低，但需要具备良好的弹性和可扩展性
与业务的关系	与具体业务相对独立，主要关注系统的整体运行和管理	直接与业务逻辑相关，负责执行具体的业务操作
性能关注点	更关注响应时间和决策的准确性，以确保系统的高效管理和稳定运行	更关注数据处理的吞吐量、延迟和并发能力，以满足业务的高性能需求

K8s的架构 - 组件

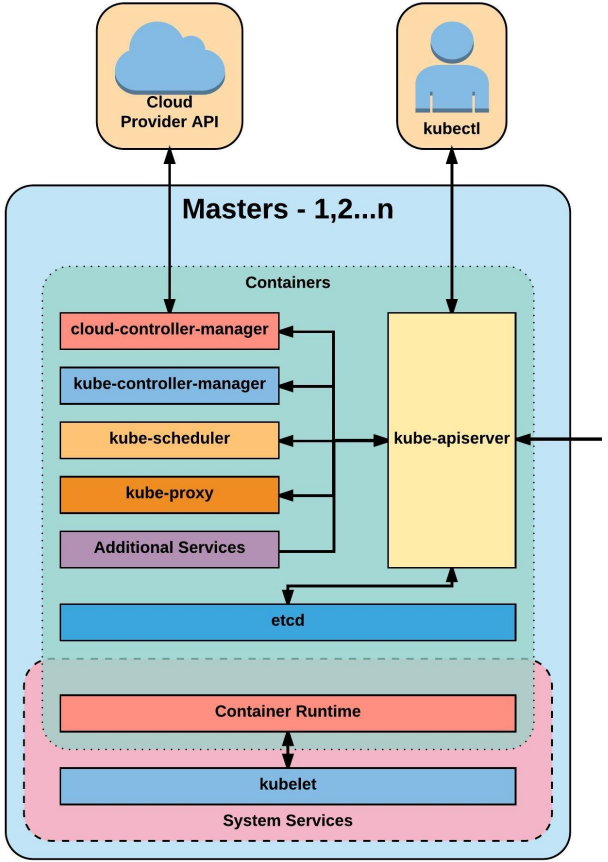

Cloud Provider API
kubectl
Masters - 1,2...n
Containers
cloud-controller-manager
kube-controller-manager
kube-scheduler
kube-proxy
Additional Services
kube-apiserver
etcd
Container Runtime
kubelet
System Services
Node 1
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Node 2
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Cloud Provider Network Edge
Load Balancer
End Users

K8s的架构 - Master



K8s的架构 - Master - 组件

- Kube-apiserver
- Etcd
- kube-controller-manager
- cloud-controller-manager
- kube-scheduler



API Server

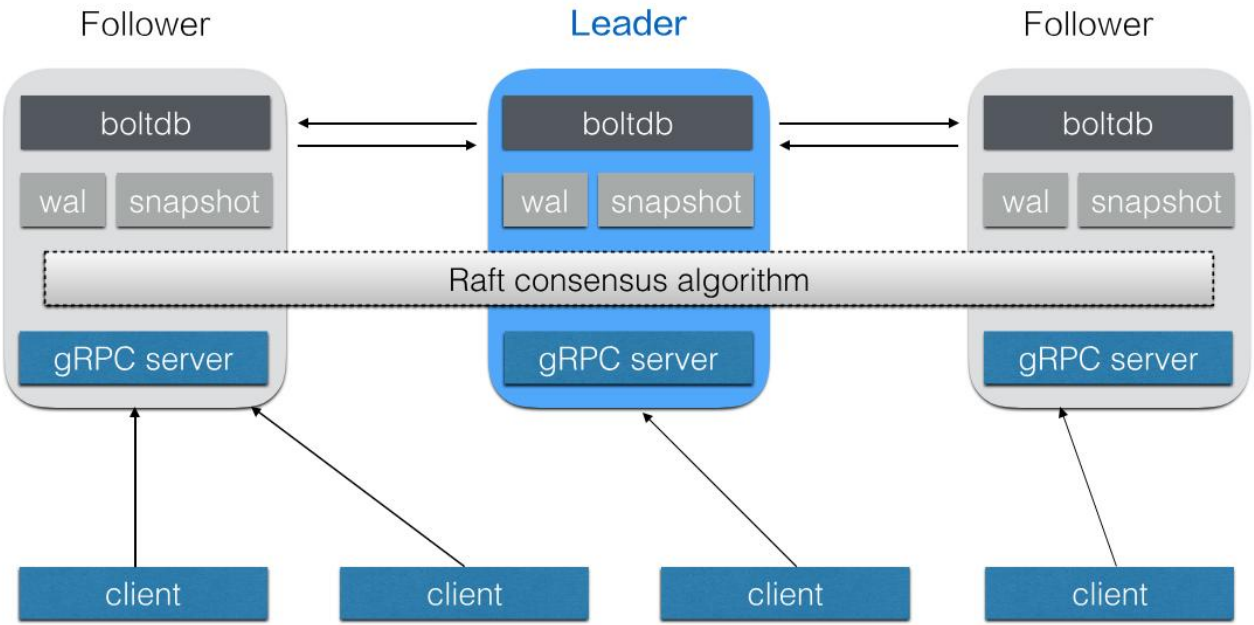
- API Server为Kubernetes控制平面和数据存储提供了一个面向前端的REST接口
- 所有客户端，包括节点、用户和其他应用程序，严格地通过API Server与Kubernetes进行交互
- 是Kubernetes的真正核心，通过处理身份验证和授权、请求验证、变更和准入控制，以及作为后端数据存储的前端，充当着集群门卫的角色
- Kubernetes中所有组件都会和 API Server 进行连接，组件与组件之间一般不进行独立的连接，都依赖于 API Server 进行消息的传送

Etcd

- Etcd是一个分布式的存储系统，充当集群数据存储库
- 提供一个强大、一致且高可用的键值存储，用于持久化集群状态，API Server 中所需要的信息都被放置在 etcd 中
- Etcd 本身是一个高可用系统，通过 etcd 保证整个Kubernetes的Master组件的高可用性

Etcd

A distributed, reliable key-value store for the most critical data of a distributed system



两个 quorum
一定存在交集

quorum = (n+1) / 2

3个节点容忍1个故障

5个节点容忍2个故障

Controller 控制器

- Controller（控制器）
  - ✓控制器是Kubernetes中的一种抽象概念，代表一类负责监控集群状态并对其进行调节的控制器，用于完成对集群状态的管理
  - ✓支持：自动对容器进行修复、自动进行水平扩张等
  - ✓具体控制器：ReplicaSet控制器负责确保在集群中运行指定数量的 Pod副本，Deployment控制器负责管理应用程序的部署和更新等

kube-controller-manager

- kube-controller-manager是Kubernetes控制平面的核心组件之一，它负责运行多种控制器，这些控制器用于管理集群中的各种资源对象。同时，这些控制器共同协作，确保集群的状态与用户期望的状态一致
  - ✓节点控制器：负责管理节点的生命周期，包括节点的注册、健康检查和删除等操作
  - ✓副本控制器：确保指定数量的Pod副本在集群中运行，以满足应用的扩展需求
  - ✓端点控制器：负责维护Service和Pod之间的映射关系，使得外部可以通过Service访问到对应的Pod
- kube-controller-manager通过 API Server监视集群状态，并引导集群朝着期望的状态发展

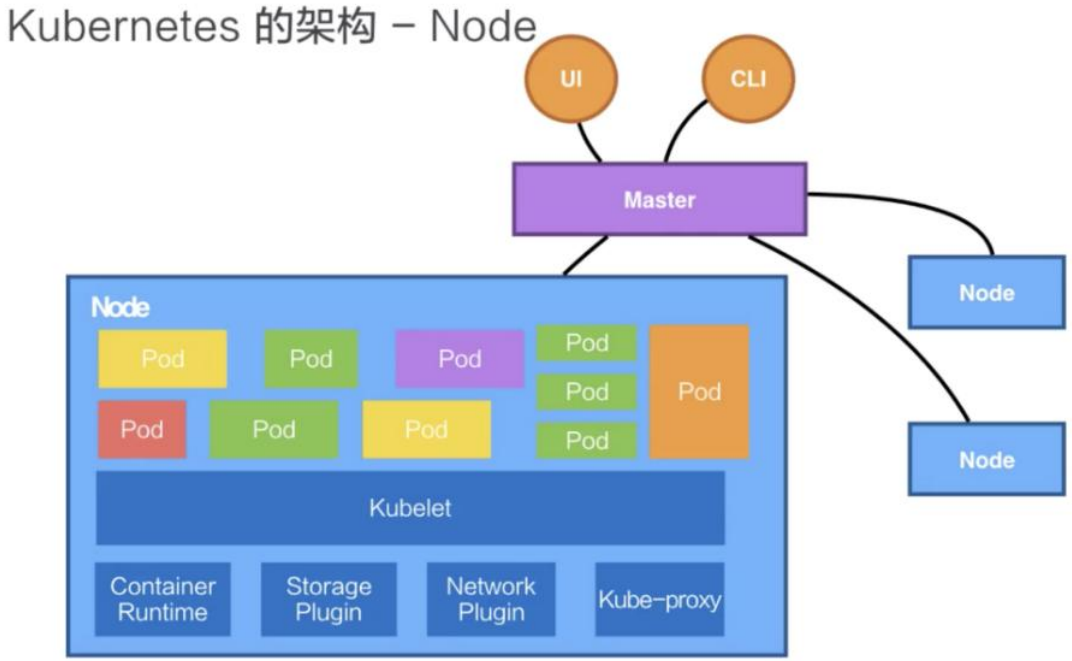
cloud-controller-manager

- cloud-controller-manager是一个可选的组件，它嵌入了云特定的控制逻辑，允许Kubernetes集群与云提供商的API进行交互
- cloud-controller-manager处理与云平台相关的功能，如管理云负载均衡器、云路由和云存储等资源
- cloud-controller-manager的设计目的是将与云平台交互的逻辑与K8s核心组件解耦，使得云提供商可以以不同的速度发布功能，而不受K8s主项目发布周期的限制

kube-scheduler（调度器）

- kube-scheduler是一个详细的、策略丰富的引擎，它评估工作负载的需求，并尝试将其放置在匹配的资源上
- 这些需求可以包括一般的硬件(CPU、memory)需求、亲和性、反亲和性，以及其他自定义资源需求
  - ✓亲和性是指将相关联的容器或服务放置在同一个节点或彼此靠近的位置
  - ✓反亲和性是指将容器或服务分散到不同的节点上，以避免它们集中在同一个节点上

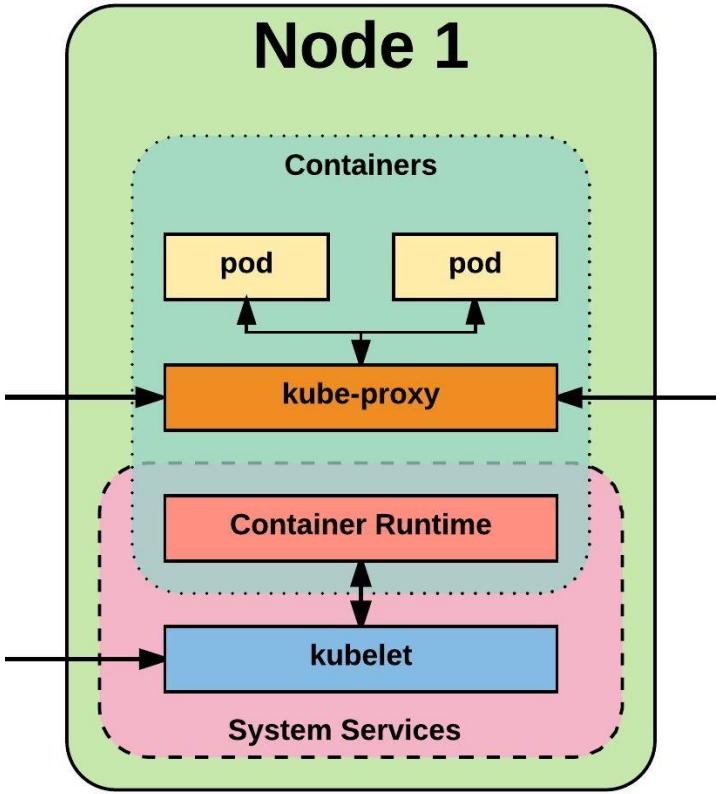
K8s的架构 - Node



Kubernetes的Node是真正运行业务负载的节点，每个业务负载会以Pod的形式运行，一个Pod中运行一个或者多个容器

K8s的架构 - Node - 组件

- kubelet
- kube-proxy
- container runtime



kubelet

- kubelet是真正运行Pod的组件，也是Node上最为关键的组件，它通过API Server接收到所需要Pod运行的状态，提交到container Runtime 组件
- kubelet充当节点代理，负责在其主机上管理Pod的生命周期
- kubelet解析YAML容器清单文件，可以从多个来源读取这些清单
  - ✓文件路径（File Path）
  - ✓HTTP终端（HTTP Endpoint）
  - ✓对Etcd中更改的监听（Etcd watch acting on any changes）
  - ✓HTTP 服务器模式，通过简单的 API 接收容器清单

kubelet

- 文件路径（File Path）
  - ✓kubelet从本地文件系统中指定的路径读取YAML容器清单文件。这种方式通常用于静态Pod的配置，静态Pod的清单通常位于/etc/kubernetes/manifests目录。
  - ✓适用于在节点上直接运行一些固定的、不需要动态调度的Pod，如节点本地的监控代理、日志收集器等。
- HTTP终端（HTTP Endpoint）
  - ✓kubelet从一个HTTP终端获取YAML容器清单文件。这个终端可以是一个简单的HTTP服务器，返回包含容器配置的YAML文件。
  - ✓适用于需要动态更新容器配置的场景，例如通过一个配置管理工具或服务来动态生成和提供容器配置文件，kubelet可以定期从该终端拉取最新的配置。

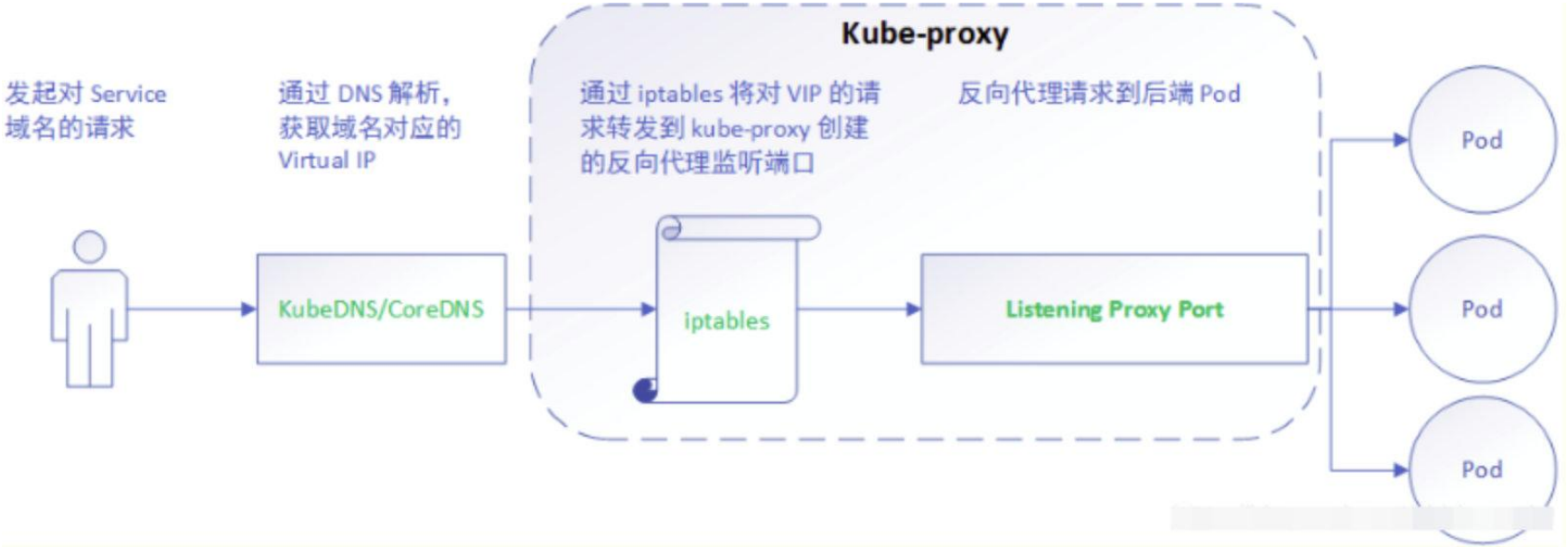
kubelet

- 对Etcd中更改的监听（Etcd watch acting on any changes）
  - ✓kubelet监听Etcd中的更改，当Etcd中的容器配置发生变化时，kubelet会检测到这些变化并相应地更新容器。
  - ✓在Kubernetes集群中，控制平面组件（如API Server）通常会将容器的配置信息存储在Etcd中。kubelet通过监听Etcd中的变化，可以实时获取最新的容器配置，确保节点上的容器状态与集群期望的状态一致。
- HTTP 服务器模式
  - ✓kubelet运行一个HTTP服务器，通过简单的API接收容器清单。其他组件或服务可以通过向该服务器发送HTTP请求来提交容器配置。
  - ✓适用于需要通过外部系统或工具动态创建和管理容器的场景，例如通过一个CI/CD流水线或自动化部署工具来提交容器配置，kubelet接收这些配置并创建相应的容器。

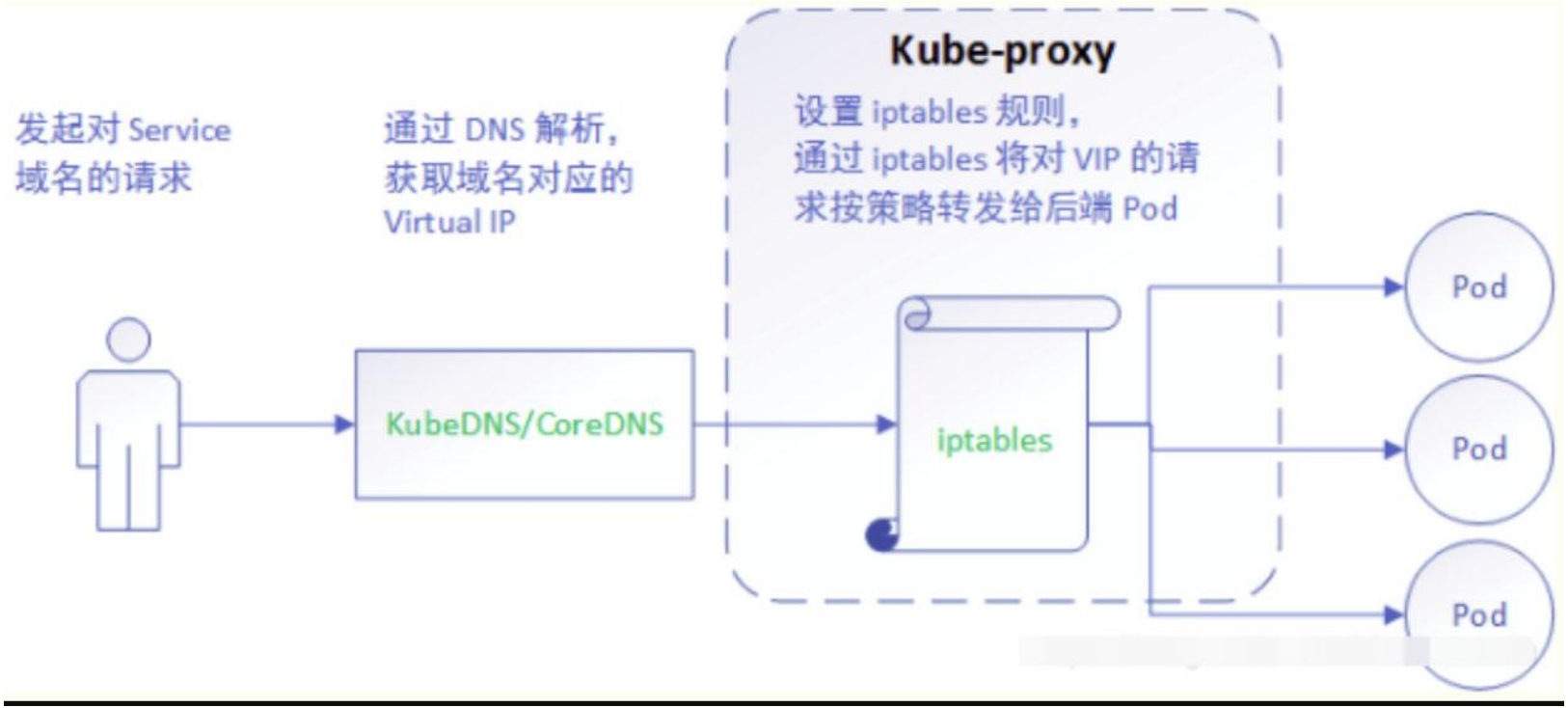
kube-proxy

- 在每个节点上管理网络规则，并为 Kubernetes 集群服务执行连接转发或负载均衡
- 可用的代理模式
  - ✓用户空间（Userspace）
  - ✓iptables
  - ✓ipvs

kube-proxy userspace模式



kube-proxy iptables模式



kube-proxy iptables模式

Allow incoming traffic on port 30000 for a NodePort service
允许流量通过 NodePort（这里端口号为30000，在实际情况中是K8s中定义的 NodePort）
iptables -A INPUT -p tcp --dport 30000 -j ACCEPT

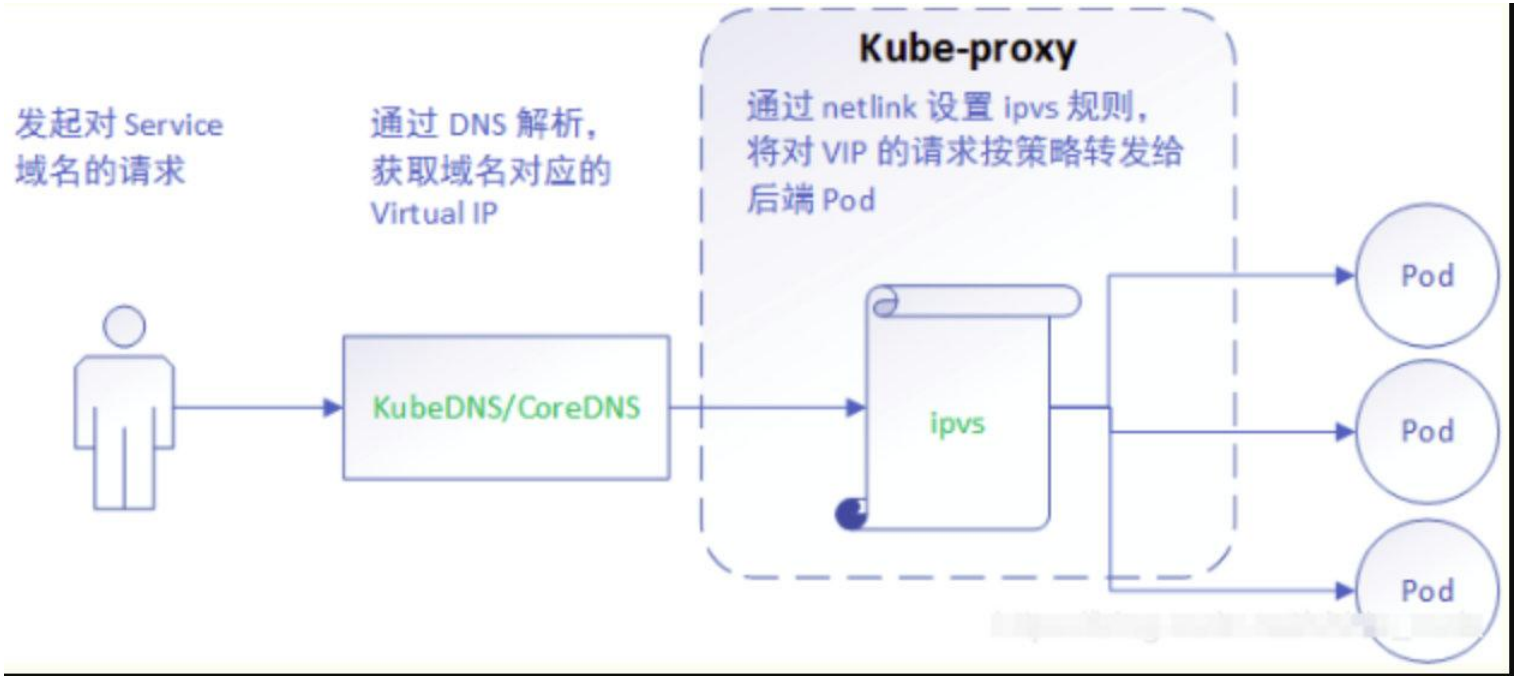
DNAT traffic to Service Cluster IP address (example: 10.0.0.1)
PREROUTING链的规则，将目标地址为服务的 Cluster IP（<service_cluster_ip>）且目标端口为 80 的流量进行目标地址转换（DNAT），将其转发到后端 Pod 的 IP 地址（<backend_pod_ip>）上的端口 80
iptables -t nat -A PREROUTING -d <service_cluster_ip> -p tcp --dport 80 -j DNAT --to-destination <backend_pod_ip>:80

kube-proxy iptables模式

Allow forwarded traffic to backend Pod (example: 192.168.0.1)
FORWARD 链的规则，它允许转发到后端 Pod IP 地址的流量
iptables -t filter -A FORWARD -d <backend_pod_ip> -p tcp --dport 80 -j ACCEPT

SNAT traffic back to client with source IP of Service Cluster IP address
POSTROUTING 链的规则，它将源自后端 Pod 的 IP 地址，目标端口为 80 的流量进行源地址转换（SNAT），将源 IP 地址修改为服务的 Cluster IP 地址（<service_cluster_ip>）
iptables -t nat -A POSTROUTING -s <backend_pod_ip> -p tcp --sport 80 -j SNAT --to-source <service_cluster_ip>

kube-proxy ipvs模式



container runtime

- 就 Kubernetes 而言，容器运行时（container runtime）是一个符合 CRI（容器运行时接口）的应用程序，用于执行和管理容器
- 可支持的容器类型
  - ✓Containerd (docker)
  - ✓Cri-o
  - ✓Rkt
  - ✓Kata
  - ✓Virtlet

Storage Plugin & Network Plugin

- 在OS上创建容器所需要运行的环境，最终把容器或者Pod运行起来，也需要对存储和网络进行管理
- Kubernetes依靠Storage Plugin、Network Plugin实现以上目标
- 用户或者云厂商开发相应的Storage Plugin或者Network Plugin完成存储操作或网络操作

其他服务

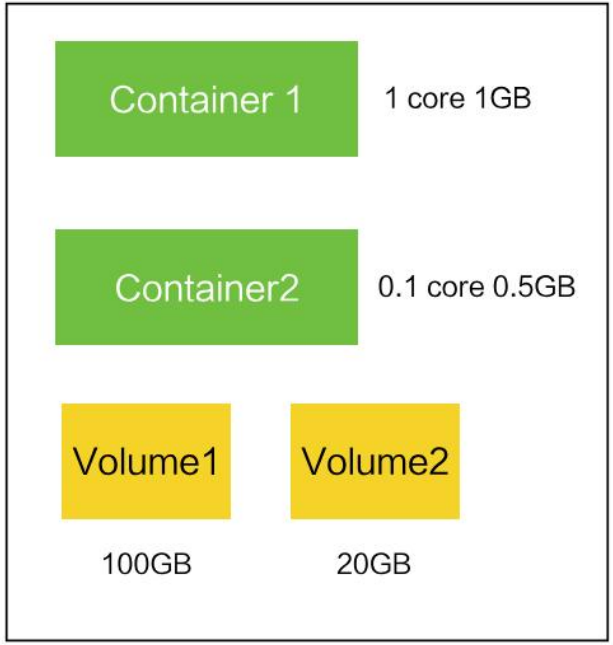
- kube-dns
  - ✓提供集群范围的 DNS 服务 （Domain Name System，域名系统）
  - ✓服务可解析为 <service>.<namespace>.svc.cluster.local.
- Heapster
  - ✓Kubernetes集群的指标收集器，被某些资源使用，例如水平Pod自动伸缩器
- kube-dashboard
  - ✓一个通用的基于 Web 的 Kubernetes 用户界面

Workload

- Workload是在集群中运行的应用程序或工作负载的抽象
- 包括了容器化的应用、批处理作业、持续运行的服务
- Workload可以是长期运行的服务（比如Web服务器），也可以是一次性任务（例如数据处理作业）
- K8S中的workload通常由以下几种类型组成
  - ✓Pods（容器组）
  - ✓Deployment（部署）
  - ✓StatefulSet（有状态集）
  - ✓DaemonSet（守护进程集）
  - ✓Job和CronJob（作业和定时任务）

Pod

- Pod是最小的调度以及资源单元
  - ✓用户可通过Kubernetes的Pod API生产一个Pod，让Kubernetes对这个Pod进行调度，即把它放在某一个 Kubernetes管理的节点上运行起来
- Pod是对一组容器的抽象，由一个或者多个容器组成，通常用于运行一个特定应用程序的实例
- Pod定义容器运行的方式（Command、环境变量等）



Pod的重要概念和特点

- 多个容器
  - ✓Pod可以包含一个或多个相关联的容器。这些容器通常共享相同的网络命名空间和存储卷，并在同一主机上运行。这种设计使得多个容器能够共享资源和相互通信，例如应用程序容器和辅助容器（如日志收集器）可以一起运行在同一个Pod中。
- 共享网络命名空间
  - ✓Pod中的所有容器共享相同的网络命名空间，它们可以使用localhost互相通信
  - ✓简化了容器之间的通信，并支持在Pod内部通过本地主机地址进行通信
- 共享存储卷
  - ✓Pod中的容器可以访问共享的存储卷，这些存储卷可以用于在容器之间共享数据
  - ✓Pod级别的存储卷提供了在容器之间共享数据的一种方式，例如，一个容器可以生成数据，另一个容器可以读取该数据进行处理

Pod的重要概念和特点

- 生命周期管理
  - ✓Pod有一个独立的生命周期，Kubernetes负责管理Pod的创建、调度、重新启动和销毁
  - ✓当Pod被创建时，Kubernetes会分配一个唯一的标识符给Pod，并确保Pod中的所有容器在同一节点上运行
  - ✓如果Pod失败或被删除，Kubernetes会重新创建Pod，并确保Pod中的容器按照定义重新启动
- 资源调度
  - ✓Pod可以指定资源请求和限制，例如CPU和内存
  - ✓Kubernetes使用这些信息来调度Pod，并确保Pod在具有足够资源的节点上运行
  - ✓Pod还可以指定节点选择策略，以指定Pod应该在哪些节点上运行
- 标签和选择器
  - ✓Pod可以使用标签进行分类和组织，并且可以使用标签选择器来指定Pod应该被哪些其他资源（如服务或部署）访问

Pod的资源清单

```
apiVersion: v1                    //必选，版本号
kind: Pod                         //必选，创建的资源类型
metadata:                         //必选，元数据
  name: my-app-pod                //必选，pod名称，此处命名为my-app-pod
  labels:                         //自定义标签列表
    app: my-app                   //标签键值对
spec:                             //pod中容器的详细定义
  containers:                     //pod中的容器列表，可以有多个容器
  - name: my-app-container        //容器名称
    image: my-app-image           //容器中的镜像名
    ports:                        //容器需要暴露的端口号列表
    - containerPort: 80           //容器要暴露的端口，此处暴露了端口80
```

ReplicaSet

- ReplicaSet是用于维护一组Pod副本的资源对象，确保在集群中运行指定数量的 Pod 副本，以满足用户定义的副本数量
- 如果由于节点故障或其他原因导致某些 Pod 副本不可用，ReplicaSet 将启动新的 Pod 副本以替换它们，确保维护所需的副本数量
- ReplicaSet的特点和用途
  - ✓自动扩展和缩减
  - ✓声明式配置
  - ✓故障恢复
  - ✓标签选择器

ReplicaSet

- 自动扩展和缩减
  - ✓通过定义副本数量，用户可以告诉 ReplicaSet 需要运行多少个 Pod 副本。当需要更多副本时，ReplicaSet 会自动启动新的 Pod 副本，当需要减少副本时，它会自动删除不再需要的 Pod 副本
- 声明式配置
  - ✓用户可以使用 YAML 或 JSON 文件来定义 ReplicaSet 的配置。这使得配置管理变得简单，用户只需指定所需的副本数量和其他参数即可，而不必担心底层实现细节

ReplicaSet

- 故障恢复
  - ✓如果某个节点上的 Pod 副本出现故障或不可用，ReplicaSet 将监测并自动替换这些不可用的 Pod 副本，以确保所需的副本数量得以维持
- 标签选择器
  - ✓ReplicaSet 使用标签选择器来标识它所管理的 Pod。这使得用户可以使用标签选择器来选择并操作与 ReplicaSet 关联的 Pod

ReplicaSet的资源清单

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

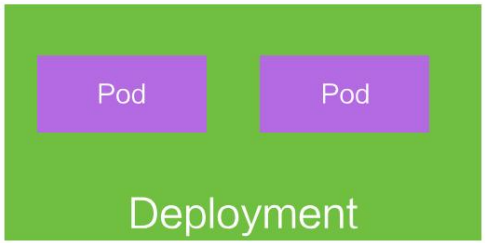
- 使用ReplicaSet来管理Pods的副本数量，确保服务的高可用性
- 该YAML文件定义了一个名为"frontend"的ReplicaSet资源，它将确保3个拥有"tier: frontend"标签的Pods正常运行
- 标签选择器app: my-app表示所有具有该标签的Pod都属于这个ReplicaSet。
- template定义了工作负载创建的Pod的模板。这里的labels必须与 selector.matchLabels匹配，否则 Kubernetes 会报错。
- 这些Pods将运行一个Nginx容器，基于"nginx:1.14.2"镜像，并监听端口80

ReplicaSet 和 Deployment

- ReplicaSet 是一种低级别的资源，用于确保一定数量的 Pod 副本始终存在。它不支持滚动更新和回滚。
- Deployment 是一种高级资源，用于管理 ReplicaSet 和 Pod。它不仅确保 Pod 副本的数量，还支持滚动更新、回滚、版本控制等功能。它通过管理 ReplicaSet 来间接管理 Pod。
- 通常不直接使用 ReplicaSet，而是通过 Deployment 来管理。
- 在需要更精细控制副本集时，可能会直接使用 ReplicaSet。
- Deployment是最常用的工作负载资源之一，适用于大多数需要管理 Pod 副本的场景。特别适合需要频繁更新和回滚的场景。

Deployment

- Deployment是在Pod抽象上更为上层的一个抽象
- Deployment可以定义一组Pod的副本数目、以及该Pod的版本；一般用Deployment实现应用真正的管理，而Pod是组成 Deployment的最小单元
- Kubernetes通过Controller维护Deployment中Pod的数目，并帮助Deployment自动恢复失败的Pod



Deployment

- Kubernetes通过Controller以指定的策略控制版本
  - ✓滚动升级
  - ✓重新生成的升级
  - ✓版本的回滚

Deployment - 滚动升级

- 滚动升级是一种平滑的升级策略，它逐步替换旧版本的 Pod 为新版本的 Pod，同时保持服务的高可用性。
- 过程
  - ✓创建一个新的 ReplicaSet，使用新的镜像或配置。
  - ✓逐步减少旧 ReplicaSet 中的 Pod 数量，同时增加新 ReplicaSet 中的 Pod 数量。
  - ✓在这个过程中，旧的 Pod 在新 Pod 准备好之前不会被删除，确保服务始终可用。
- 优势：
  - ✓服务不会中断，确保高可用性。
  - ✓可以逐步验证新版本的稳定性。
- 劣势
  - ✓回滚复杂、版本不一致、资源使用增加等
- 适用场景
  - ✓适用于需要频繁更新且对服务中断容忍度低的场景。

Deployment - 重新生成的升级

- 重新生成的升级是一种简单的升级策略，它会先删除所有旧版本的 Pod，然后创建新版本的 Pod。
- 过程：
  - ✓删除所有旧版本的 Pod。
  - ✓创建新版本的 Pod。
- 优势：
  - ✓简单直接，适用于对服务中断容忍度较高的场景。
- 劣势：
  - ✓在升级过程中，服务会短暂中断。
- 适用场景
  - ✓适用于对服务中断容忍度较高的场景，如测试环境或非关键服务。

Deployment - 版本的回滚

- 版本的回滚是指将服务恢复到之前的某个稳定版本，通常用于新版本出现问题时。
- 过程
  - ✓回滚到之前保存的配置版本。
  - ✓重新创建旧版本的 ReplicaSet 和 Pod。
- 优势
  - ✓快速恢复服务到稳定状态。
  - ✓降低新版本引入的风险。
- 适用场景
  - ✓适用于新版本出现问题时，需要快速恢复服务的场景。

Deployment的资源清单

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-nginx-deployment
  annotations:
    decription:   “nginx test”
  labels:
    app: nginx
    desc: test
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
    matchExpressions:
      - {key: desc, operator: In, values: [test]}
  ……
```

基于集合的选择器:

- In
- NotIn
- Exists
- DoesNotExist

支持基于集合选择器的对象:

- ReplicaSet
- Deployment
- Job
- DaemonSet
- PersistentVolumeClaims

StatefulSet

- StatefulSets是一种用于管理有状态应用程序的资源对象，它确保Pods具有稳定的标识符和持久的网络标识
- 与ReplicaSet和Deployment不同，StatefulSet提供了一种更加稳定和持久的方式来管理有状态应用程序的部署，例如数据库、缓存系统等
- StatefulSets的特点和用途
  - ✓稳定的网络标识符
  - ✓有序部署和扩展
  - ✓持久化存储
  - ✓有状态服务发现
  - ✓有限制的滚动更新

StatefulSet

- 稳定的网络标识符
  - ✓StatefulSet 为每个 Pod 分配稳定的网络标识符（通常是一个序列编号），这个标识符与 Pod 的生命周期绑定
  - ✓使得有状态应用程序可以通过网络标识符进行可靠的识别和访问
- 有序部署和扩展
  - ✓StatefulSet 支持有序的部署和扩展，即每个 Pod 都按照预定义的顺序依次启动或终止
  - ✓这对于有状态应用程序非常重要，因为它们可能依赖于特定的启动顺序或依赖关系
- 持久化存储
  - ✓StatefulSet 通常与持久卷存储（Persistent Volume）结合使用，以确保每个 Pod 的数据持久化存储
  - ✓使得有状态应用程序能够在 Pod 重新启动或迁移时保持数据的持久性和一致性

StatefulSet

- 有状态服务发现
  - ✓StatefulSet 与 Kubernetes 的服务发现机制集成，可以为每个 Pod 提供唯一的 DNS
  - ✓其他应用程序可以通过 DNS 来发现和访问特定的有状态应用程序副本
- 有限制的滚动更新
  - ✓与 ReplicaSet 和 Deployment 不同，StatefulSet 的滚动更新是有限制的
  - ✓它只能对一个 Pod 进行更新，确保每次更新都不会破坏应用程序的状态或导致服务中断

StatefulSet的资源清单

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql
  replicas: 3
  selector:
    matchLabels:
      app: mysql
```

```
template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:latest
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: password
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
```

```
volumeClaimTemplates:
  - metadata:
      name: mysql-persistent-storage
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

- Pod 中定义了一个卷挂载 mysql-persistent-storage，用于持久化存储 MySQL 数据
- volumeClaimTemplates 字段定义了持久卷存储的模板。在该示例中，创建了一个名为 mysql-persistent-storage 的持久卷存储，容量为 1Gi，访问模式为 ReadWriteOnce（单路读写）

DaemonSet

- DaemonSet用于在集群的每个节点上运行一个Pod的副本
- 当新的节点加入集群时，DaemonSet会自动在新节点上启动 Pod，确保所有节点都具有相同的副本
- DaemonSet可以配置为只在具有特定标签的节点上运行Pod，这使得用户可以选择在集群的特定子集节点上运行守护进程
- 与Deployment不同，DaemonSet的Pod通常是持续运行的，即它们不会因为没有任务或工作而终止
- DaemonSet常用于部署日志收集器、监控代理和其他与运维相关的服务
  - ✓通过在每个节点上运行这些守护进程，可以确保及时收集节点级别的日志和监控数据，并将其发送到中央集群或服务以进行分析和处理

DaemonSet的资源清单

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
```

```yaml
    spec:
      containers:
      - name: fluentd
        image:
fluent/fluentd:v1.12.0
        resources:
          limits:
            memory: "200Mi"
            cpu: "100m"
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

- 该示例将在集群的每个节点上创建一个 Fluentd Pod，用于收集节点上的日志并发送到中央日志收集器或存储中
- Pod 中定义了资源限制，以限制 Fluentd 容器的内存和 CPU 使用量
- Pod 中定义了一个卷挂载 varlog，用于挂载节点的 /var/log 目录，以便 Fluentd 可以收集日志
- volumes 字段定义了 Pod 中使用的卷。在该示例中，创建了一个名为 varlog 的卷，类型为 hostPath，它将节点的 /var/log 目录挂载到 Fluentd 容器中

Job

- Job确保在集群中运行指定数量的Pods
- Job用于运行一次性任务，这些任务通常是一些需要在集群中执行的离线或批处理作业，例如数据处理、备份任务、定时任务等
  - ✓每个Job都会创建一个或多个独立的Pod实例来运行作业。这些Pod实例与Job相关联，它们共同负责执行作业中定义的任务
  - ✓一旦作业完成，Job将终止其创建的所有Pod实例
  - ✓如果作业中的某些任务失败，Job可以配置为根据用户定义的策略执行相应的操作。例如，可以指定重试策略、指定失败后的延迟时间、设置失败后保留的 Pod 实例数量等
- Job可以配置为并行执行任务，也可以配置为按顺序执行任务

Job的资源清单

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  template:
    metadata:
      name: my-pod
    spec:
      containers:
      - name: my-container
        image: busybox:latest
        command: ["echo", "Hello, Kubernetes!"]
      restartPolicy: Never
```

- 该Job示例创建一个名为 my-job 的 Job 实例，该 Job 会创建一个临时的 Pod，运行一个 echo 命令输出文本 Hello, Kubernetes!，然后在任务完成后自动终止
- Pod 的容器使用了 busybox:latest 镜像，其中包含了基本的 Linux 命令工具
- Pod 中的容器定义了一个命令 ["echo", "Hello, Kubernetes!"]，用于在容器内部执行 echo 命令输出文本 Hello, Kubernetes!
- restartPolicy: Never 表示 Pod 的重启策略为永远不重启，表示当容器执行完成后，Pod 将会被终止，而不会重新启动

重启策略

- Always
  - ✓默认的重启策略
  - ✓当容器退出时，无论是正常退出还是异常退出，Kubernetes 都会自动重新启动容器
  - ✓容器会在退出后立即重新启动，直到 Pod 被删除或终止
  - ✓Job/CronJob中不能设置为Always
- OnFailure
  - ✓当容器因为失败而退出（即退出代码不为 0）时，Kubernetes 会自动重新启动容器
  - ✓如果容器正常退出（退出代码为 0），则不会触发重新启动
  - ✓适用于一些失败时可以自动恢复的场景。
- Never
  - ✓当容器退出时，Kubernetes 不会自动重新启动容器
  - ✓容器只会在 Pod 第一次创建时启动，并在退出后不会重新启动
  - ✓通常用于一次性任务或者需要手动干预的场景，例如 Job 或者 CronJob

CronJob

- CronJob允许按照预定时间间隔执行作业，适用于运行定时任务，如数据备份、定时清理、报表生成
- CronJob允许用户根据Cron表达式来定义定期执行的作业
- CronJob提供了灵活的调度选项，允许用户定义作业的启动时间、重复间隔、截止日期等参数
- 每当CronJob触发时，会自动创建一个新的Job实例来运行指定的任务，作业由 CronJob 自动创建和管理
- CronJob支持定义任务失败时的处理策略，用户可以指定重试策略、失败后的延迟时间、设置失败后保留的作业实例数量等
- 当CronJob触发时，它可以并行执行多个作业实例

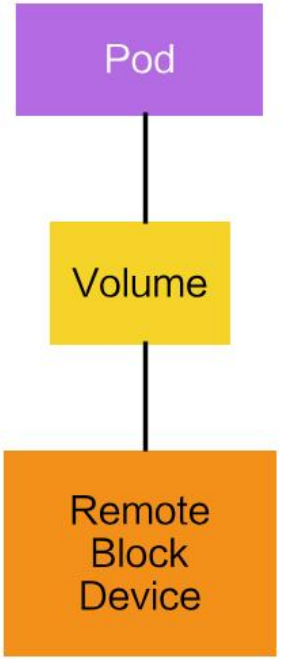
CronJob的资源清单

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/1 * * * *" # 每分钟执行一次
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: my-container
            image: busybox:latest
            command: ["echo", "Hello, Kubernetes!"]
          restartPolicy: Never
```

- 该CronJob 示例创建一个名为 my-cronjob 的 CronJob 实例，该 CronJob 每分钟执行一次任务，任务是在一个临时的 Pod 内部执行 echo 命令输出文本 Hello, Kubernetes!，然后在任务完成后自动终止
- spec.schedule 字段定义了任务的执行计划，这里是每分钟执行一次。该字段遵循标准的 Cron 表达式格式
- spec.jobTemplate 字段定义了在每个执行计划触发时要创建的 Job 的模板
- Job 模板中的容器使用了 busybox:latest 镜像，其中包含了基本的 Linux 命令工具
- Job 模板中的容器定义了一个命令 ["echo", "Hello, Kubernetes!"]，用于在容器内部执行 echo 命令输出文本 Hello, Kubernetes!

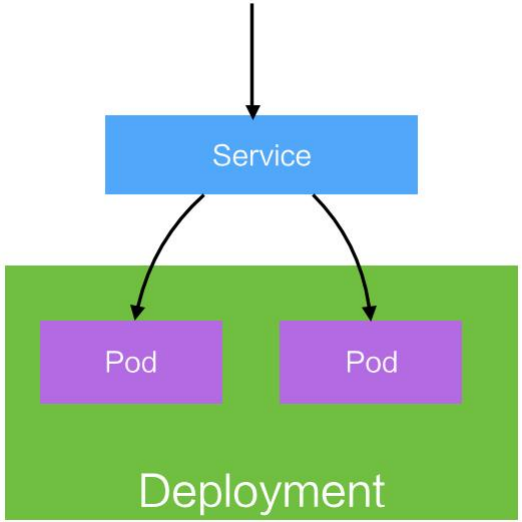
Volume

- Volume(卷)管理Kubernetes存储，用于声明在Pod中的容器可以访问的文件目录
  - ✓一个卷可以被挂载在Pod中一个或者多个容器的指定路径下面
- Volume是一个抽象的概念，一个Volume可以支持多种后端存储
  - ✓本地存储
  - ✓分布式的存储：ceph，GlusterFS
  - ✓云存储：阿里云上的云盘、AWS上的云盘、Google上的云盘



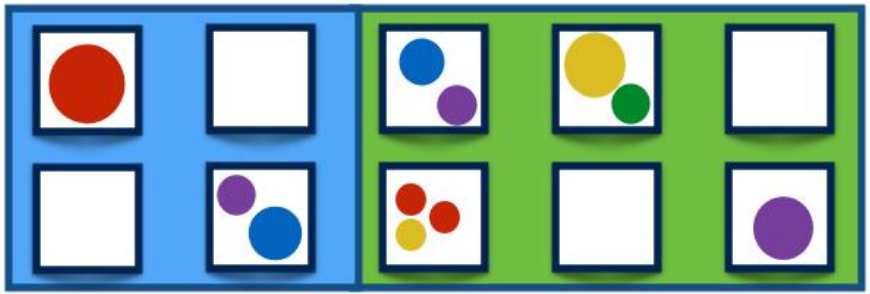
Service

- Service提供一个或多个Pod实例的稳定访问地址
  - ✓一个Deployment可能有两个甚至更多个完全相同的Pod
  - ✓对于一个外部用户，访问哪个Pod其实都是一样的，所以希望做一次负载均衡，在做负载均衡的同时，只想访问某一个固定的VIP（Virtual IP地址），而不希望得知每一个具体Pod的IP地址
- 实现Service的多种方式
  - ✓Cluster IP
  - ✓NodePort
  - ✓LoadBalancer



Namespace

- Namespaces实现一个集群内部的逻辑隔离，包括鉴权、资源管理等
- Kubernetes的每个资源（Pod、Deployment、Service）都属于一个Namespace，同一个Namespace中的资源需要命名的唯一性，不同的Namespace中的资源可以重名



声明式API

- Kubernetes 的声明式 API 是一种设计理念，它使得用户能够通过定义所需的状态，而非实现操作的具体步骤，来管理集群中的资源；这种 API 设计风格使得 Kubernetes 更加易用、可扩展和自动化
- 关键特点
  - ✓通过 YAML 或 JSON 文件定义资源配置
  - ✓不指定操作步骤
  - ✓K8S自动管理资源状态
  - ✓自动恢复
  - ✓幂等性操作
  - ✓可追踪和审计

声明式API

- 通过 YAML 或 JSON 文件定义资源配置
  - ✓K8S的声明式 API 允许用户通过 YAML 或 JSON 文件定义所需资源的配置
  - ✓这些文件描述了资源的期望状态，包括容器镜像、标签、资源请求、副本数量等信息
- 不指定操作步骤
  - ✓与传统的命令式 API 不同，声明式 API 不需要用户指定执行操作的具体步骤
  - ✓用户只需要描述资源的期望状态，而不需要告诉 K8S 如何实现这个状态
- K8S自动管理资源状态
  - ✓一旦用户提交了资源配置文件，K8S将自动比较当前状态与所需状态，并采取必要的操作来使资源达到期望状态
  - ✓操作包括创建、更新、删除资源，以及调度、启动和终止容器等

声明式API

- 自动恢复
  - ✓如果资源的当前状态与期望状态不一致（例如由于节点故障导致 Pod 不可用），K8S 将自动采取恢复措施，以确保资源恢复到期望的状态
- 幂等性操作
  - ✓声明式 API 的操作是幂等的，即无论操作执行多少次，最终结果都是相同的
  - ✓用户可以多次提交相同的配置文件，而不必担心造成不一致的状态
- 可追踪和审计
  - ✓由于用户通过提交 YAML 或 JSON 文件来定义资源，因此可以轻松地追踪和审计资源的更改历史
  - ✓使得在多人协作或复杂环境中更容易跟踪和理解系统的变化

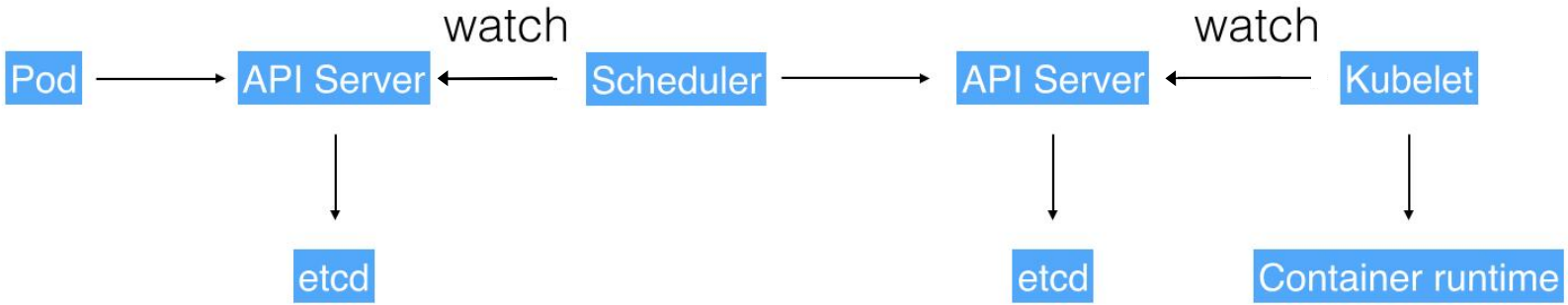
List-Watch机制

- 在 K8S 中，List-Watch 机制是一种用于获取资源对象的变化通知的机制
- 它允许客户端持续监视 Kubernetes API Server中的资源，并在资源对象的状态发生变化时接收相应的通知
- List-Watch 机制通常与长连接（长轮询）结合使用，以实现实时获取资源对象变更的能力
- List-Watch 机制包含List操作和Watch操作
- 通过 List-Watch 机制，客户端可以实现实时地获取资源对象的变更通知，从而及时地响应集群中资源的状态变化，有利于需要实时监控、自动伸缩、事件驱动等场景

List-Watch机制

- List操作
  - ✓List 操作用于获取指定类型的资源对象的当前状态
  - ✓当客户端发起 List 请求时，API Server将返回当前集群中所有符合指定条件的资源对象的列表
  - ✓使得客户端可以获取资源对象的初始状态，并与本地存储的状态进行比较
- Watch操作
  - ✓Watch 操作用于持续监视资源对象的状态变化
  - ✓一旦客户端发起 Watch 请求，API Server将保持连接打开，并将在资源对象的状态发生变化时向客户端发送通知
  - ✓这些通知可以包括新增、修改、删除等事件，客户端可以根据这些事件进行相应的处理

pod调度过程中的组件交互



1. 用户通过UI或者CLI提交一个Pod给Kubernetes进行部署，该Pod请求首先提交给Kubernetes API Server，API Server会把这个信息写入存储系统etcd，之后Scheduler会通过 API Server 的watch（notification）机制得到这个信息，即有一个Pod 需要被调度
2. Scheduler根据内存状态进行一次调度决策，在完成这次调度之后，会向 API Server报告，说明该Pod需要被调度到某一个节点上
3. API Server接收到这次操作之后，把结果再次写到etcd中，然后API Server通知相应的节点进行该Pod真正的执行启动。相应节点的kubelet会得到这个通知，调用Container runtime真正启动配置这个容器和这个容器的运行环境，调度 Storage Plugin配置存储，network Plugin配置网络

SOFT130091.01
云原生软件技术

End

5. 容器编排 Kubernetes