

Kubernetes 基础

王天青 | 2024-07

目录 >

CONTENTS

- 1 简介
- 2 架构
- 3 资源对象



1 chapter

简介

- ✓ 什么是 Kubernetes
- ✓ Kubernetes 优势

➤ Kubernetes 起源

- Kubernetes 是开源的容器集群管理项目，诞生于 2014 年，由 Google 公司发起。
- 前身 Borg 系统在 Google 内部应用了十几年，积累了大量来自生产环境的实践经验。
- 试图为基于容器的应用部署和管理打造一套强大并且易用的管理平台。
- 该项目基于 Go 语言实现。



➤ 什么是 Kubernetes

- 一个基于容器技术的分布式架构领先方案
- 一个生产级容器编排工具
- 一个完备的分布式系统支撑平台
- 竞品有 Mesos、Docker Swarm 等

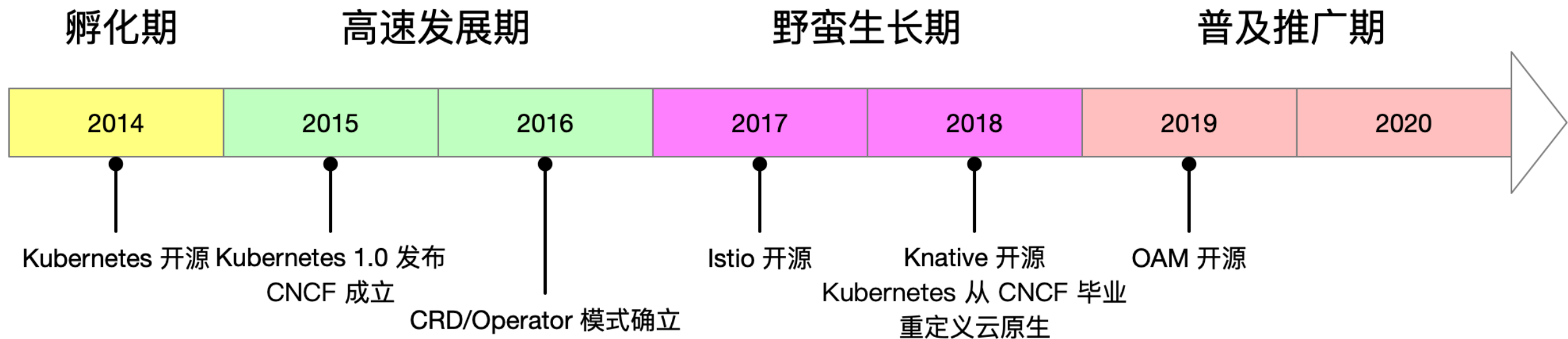



➤ Kubernetes 优势

- 强大的容器编排能力
- 轻量级
- 开放开源
- 优秀的API设计
- 基于微服务模式的多层资源抽象模型
- 可扩展性好
- 自动化程度高
- 部署支持多种环境

➤ Kubernetes 用途

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用
- 优化资源的使用





2 chapter

Kubernetes 资源对象

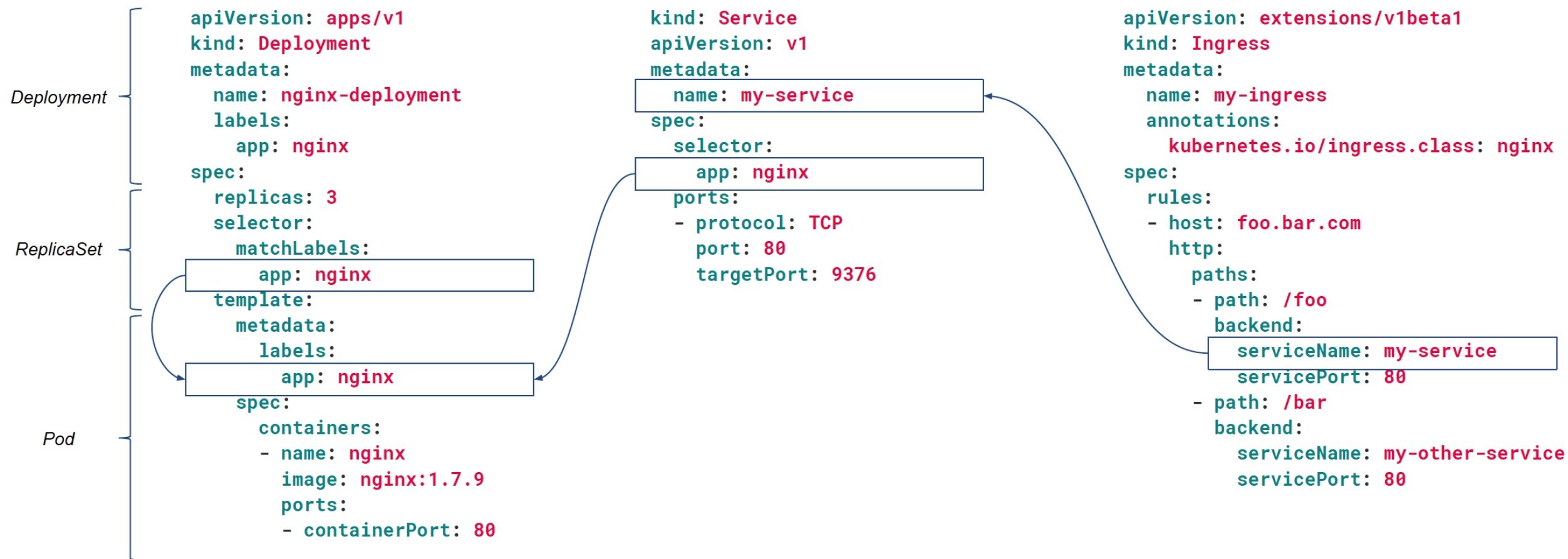
✓ Kubernetes 资源对象

一个例子

Deployment configuration:

Service configuration:

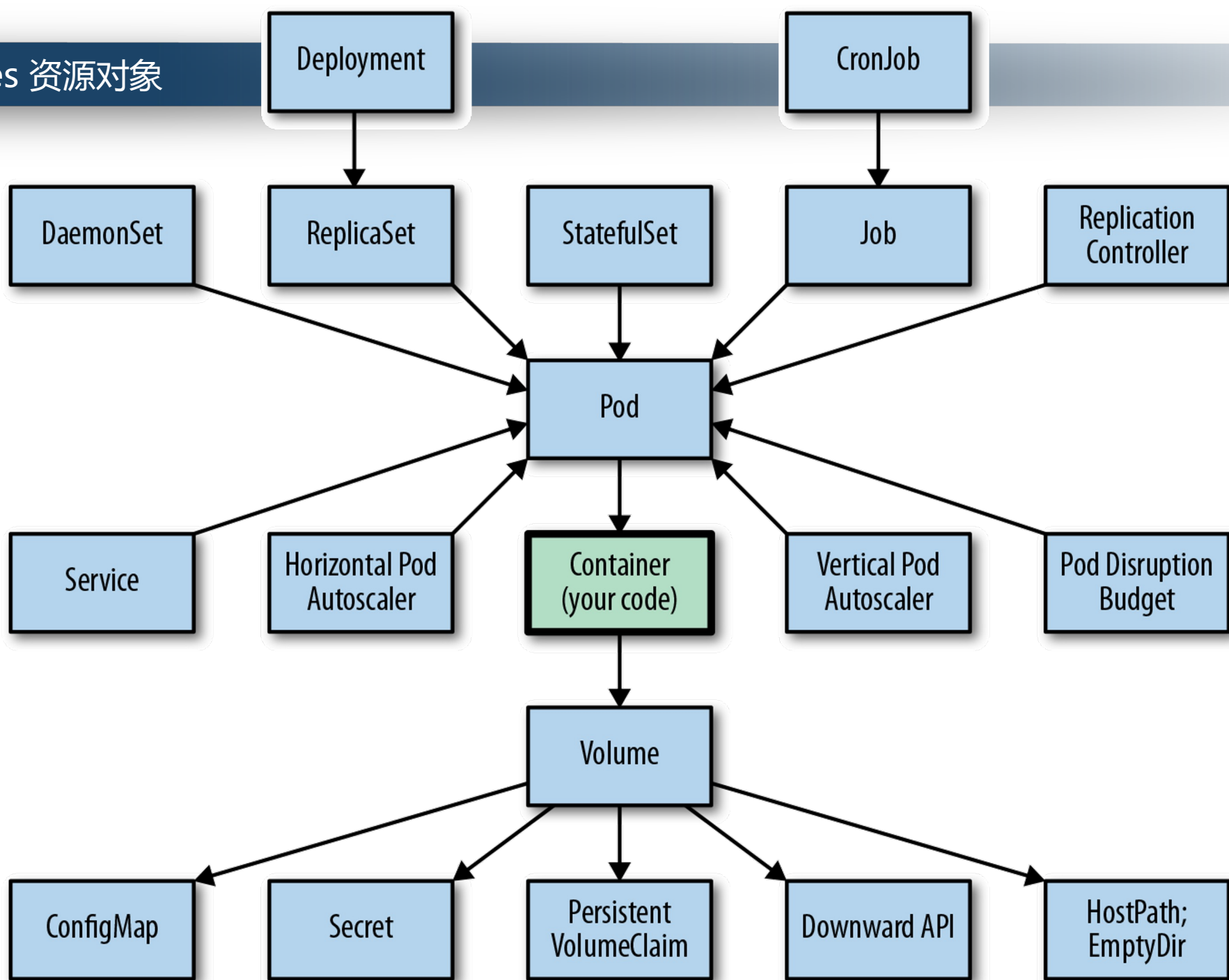
Ingress configuration:

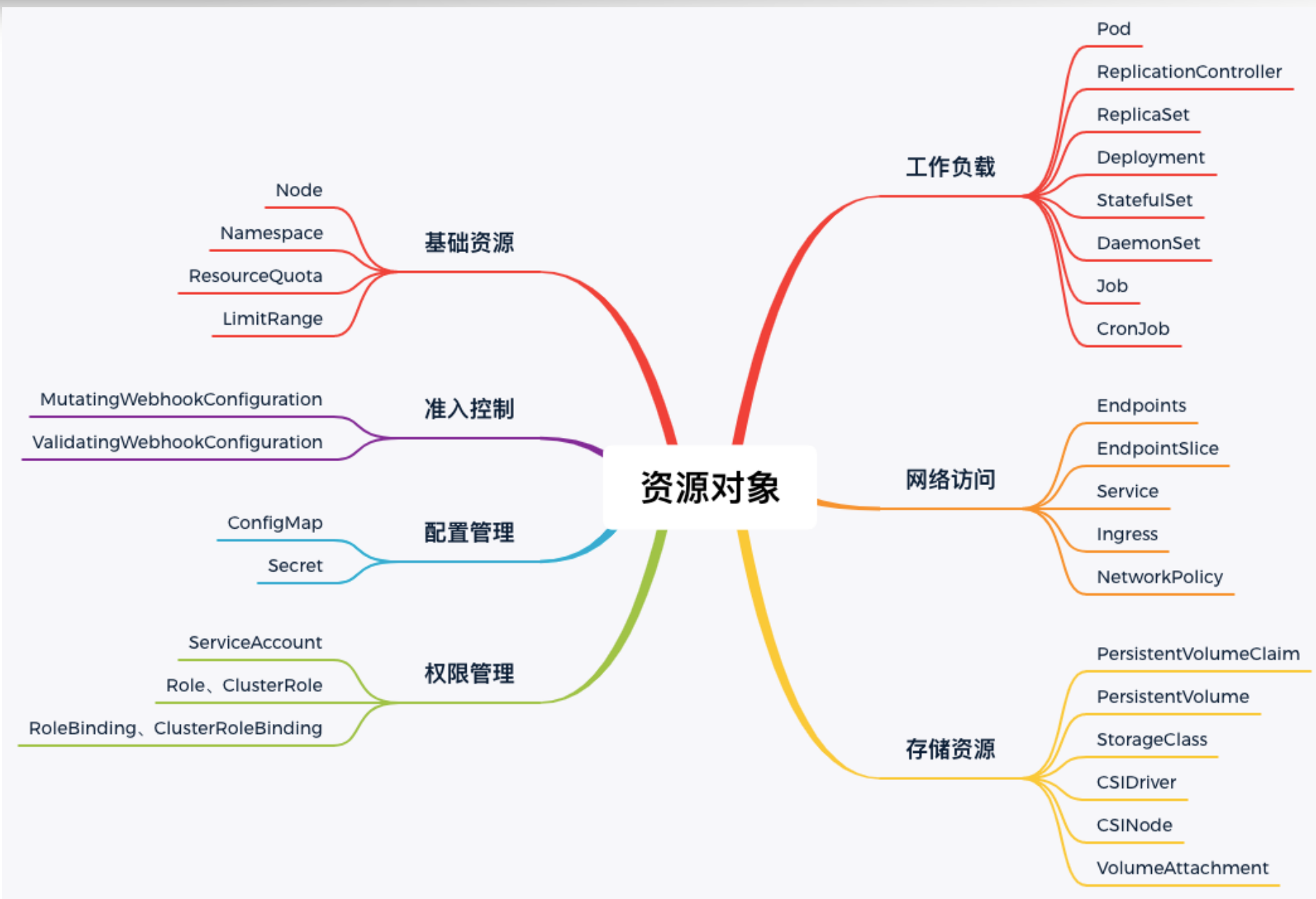


kubectl apply -f <https://k8s.io/examples/application/deployment.yaml>
kubectl apply -f <https://k8s.io/examples/application/deployment-update.yaml>

3.1 Kubernetes 资源对象

Kubernetes 资源对象





➤ Node

- Node即节点，对应一个物理节点（物理机或者虚拟机），运行和管理Pod的生命周期，以及其他属于节点上的资源对象。

➤ Namespace

- Namespace即命名空间，为一个逻辑的资源对象，并不对应具体的物理节点，其是一些逻辑资源定义的集合。不同Namespace之间的资源具有一定的隔离性，无法跨Namespace访问。
- Kubernetes默认创建了两个Namespace
 - default: 创建资源时如果不指定Namespace，则被放到这个Namespace中
 - kube-system: Kubernetes自己创建的系统资源以及核心组件都将放到这个Namespace中

➤ ResourceQuota

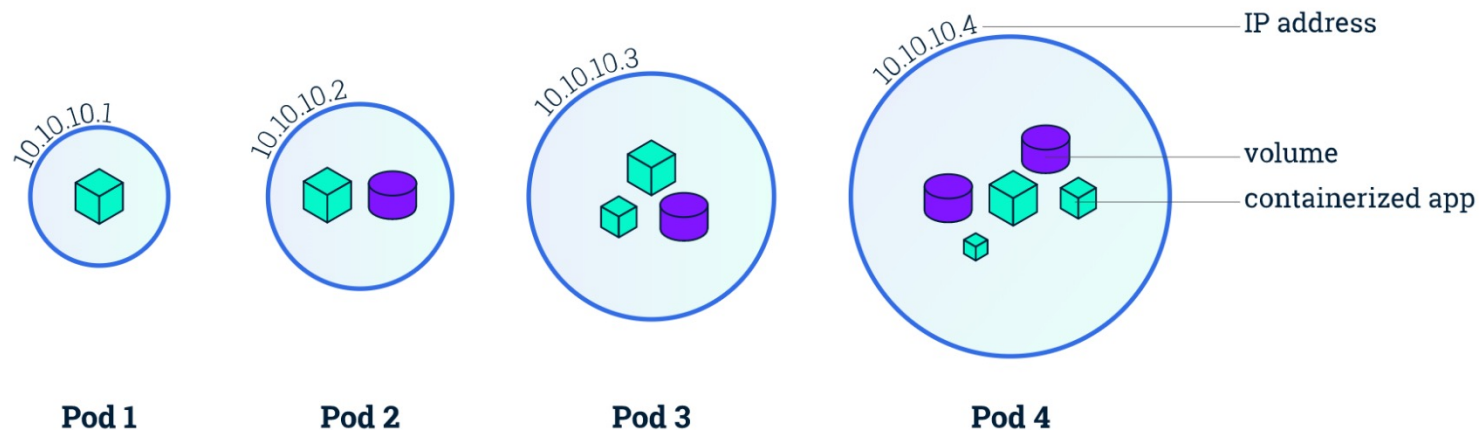
- 为Namespace的资源集合设置配额管理，超出配额上限时，创建工作负载将失败

➤ Pod

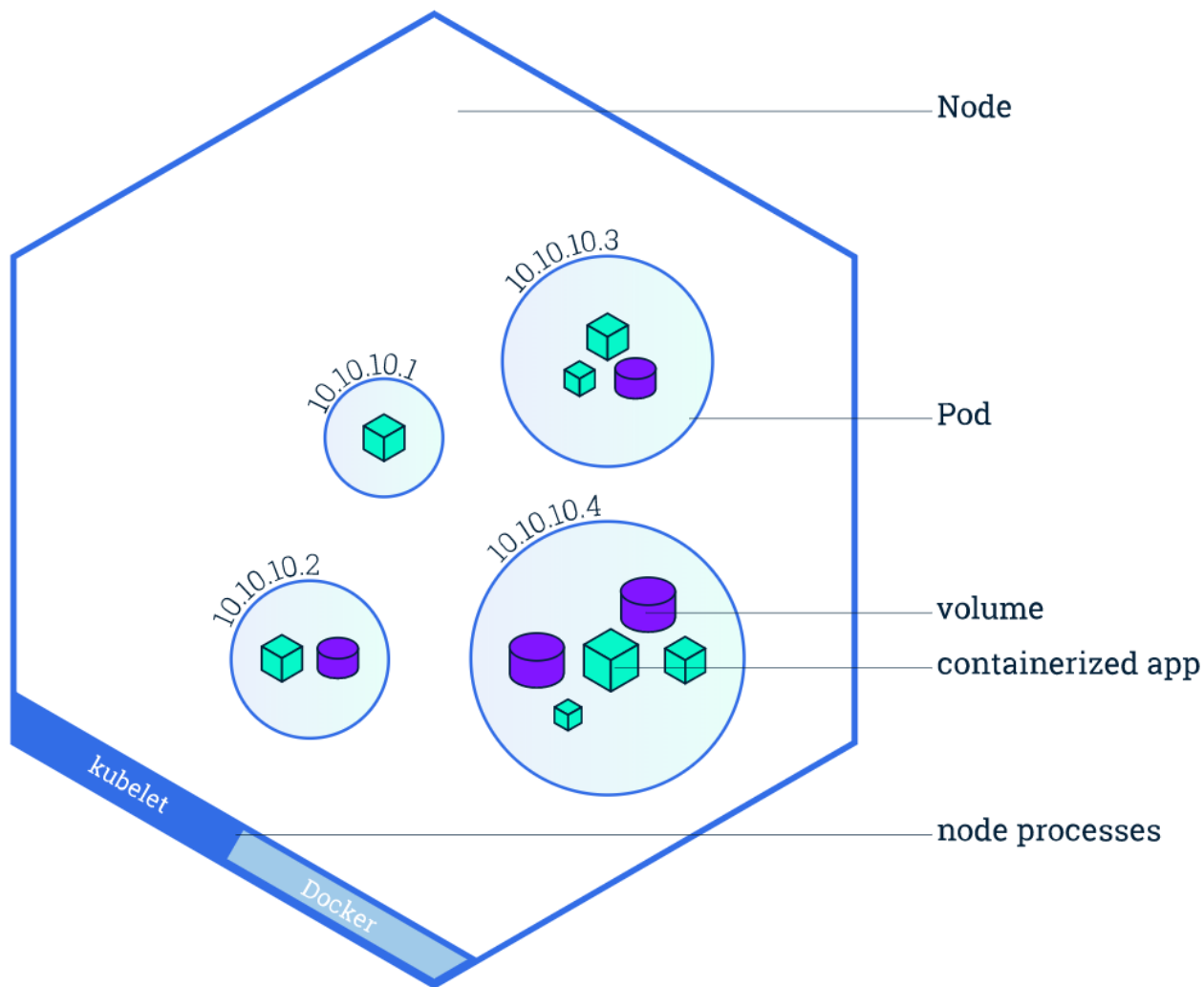
- 即容器组，是 Kubernetes 集群中最小的操作单元，如创建、调度、管理、销毁等
- 一个 Pod 包含一个或多个紧密相关的容器，并共享命名空间、网络 and 存储卷等
- 可以简单将一个 Pod 类比一个抽象的“虚拟机”，里面运行的容器类比成进程

➤ 特点

- 一个 Pod 所有容器在同一个 Node 上
- 每个 Pod 都有一个独立IP



➤ Pod 运行位置

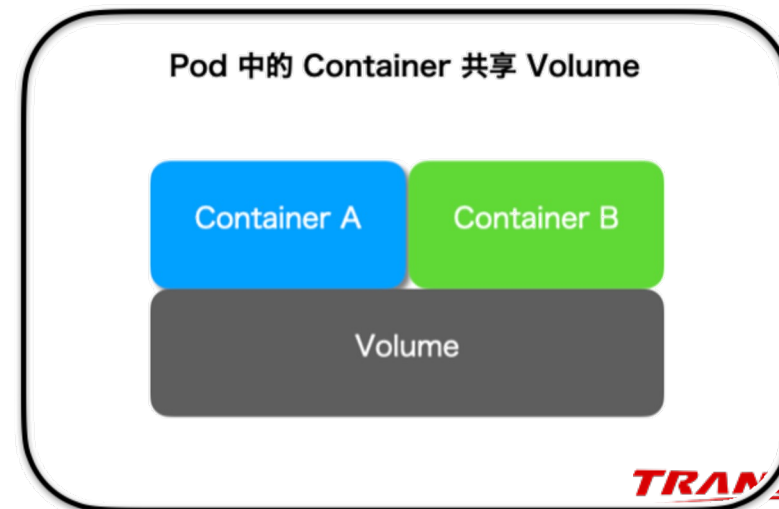
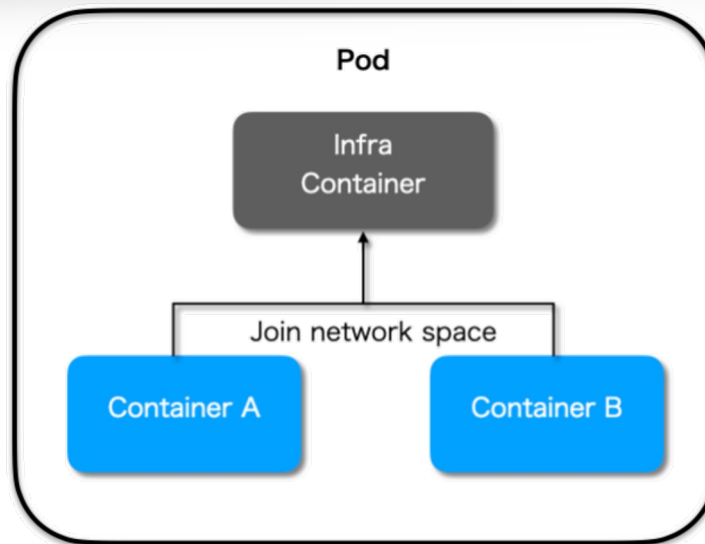


- Pod 总是运行在 Node 上
- 每个 Kubernetes 节点至少运行以下组件：
 1. kubelet 是负责 Kubernetes Master 和 所有节点之间通信的进程，它管理机器上运行的 Pod 和容器。
 2. 容器运行时(例如 Docker、rkt) 负责从镜像仓库中拉取容器镜像，解包容器并运行应用程序。

➤ Pause 容器 (Infra 容器)

Infra 容器使用一个特殊的镜像，叫做：k8s.gcr.io/pause，它占用极少的资源。Infra 容器被创建后会初始化 Network Namespace，之后用户容器就可以加入到 Infra 容器中了。所以对于 Pod 中的容器 A 和 B 来说，它们：

- 能够直接使用 localhost 通信；
- 看到与 Infra 容器相同的网络设备
- Pod 只有一个 IP 地址，也就是该 Pod 的 Network Namespace 对应的 IP 地址；
- 所有网络资源均一个 Pod 一份，被 Pod 中所有容器共享；
- Pod 的生命周期仅与 Infra 容器一致，与用户容器无关。



➤ Pod 生命周期的 5 种状态值

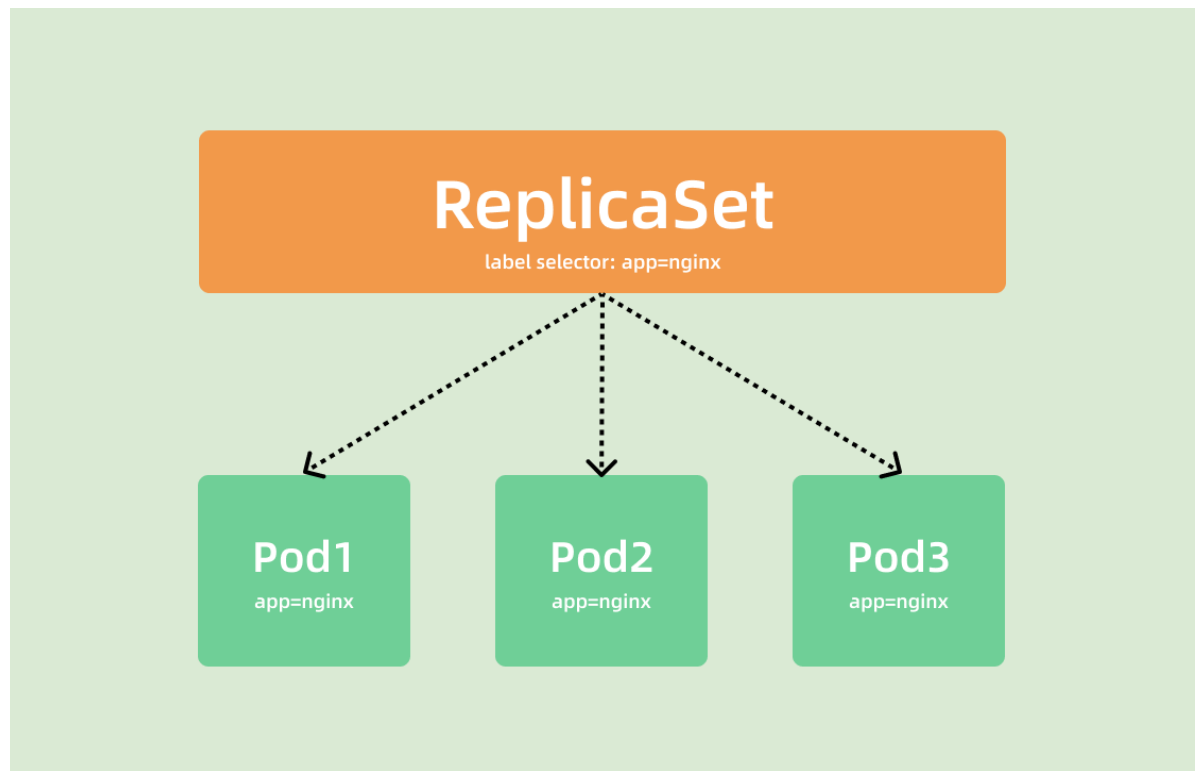
状态	解释
Pending	已经被系统接受，但Pod内有一个或多个容器的镜像还未就绪
Running	Pod内所有的容器都已被创建，且至少有一个容器处于运行状态
Succeed	Pod内所有的容器都正常退出，不需要重启，任务完成
Failed	Pod内所有的容器都退出，其中至少一个容器是非正常退出
Unknown	由于某种原因无法获取Pod的状态，如所在节点无法汇报状态、网络不通

- **ReplicaSet**

Pod作为Kubernetes中调度运行的最小单元，在实际使用时通常并不直接创建Pod使用，因为Pod本身并不具备故障自愈的能力（即在出现故障或者异常挂掉后不能自动恢复），而作为一个成熟的编排系统来说，故障自愈能力尤为关键。

因此，Kubernetes设计了多种工作负载用来满足各种实际需求场景，通过控制器模式对Pod进行故障自愈管理。

而ReplicaSet就是其中一种最为简单的用来管理一组Pod的工作负载类型（**主要用于无状态负载**），其通过标签选择器关联出一组Pod，并对这些Pod 的健康状态进行监控和管理，在Pod的Running数目不满足预期时，尽可能的拉起相应的Pod使得满足期望的运行数量。同时也可以通过对replicas的修改来实现Pod扩缩容的能力。

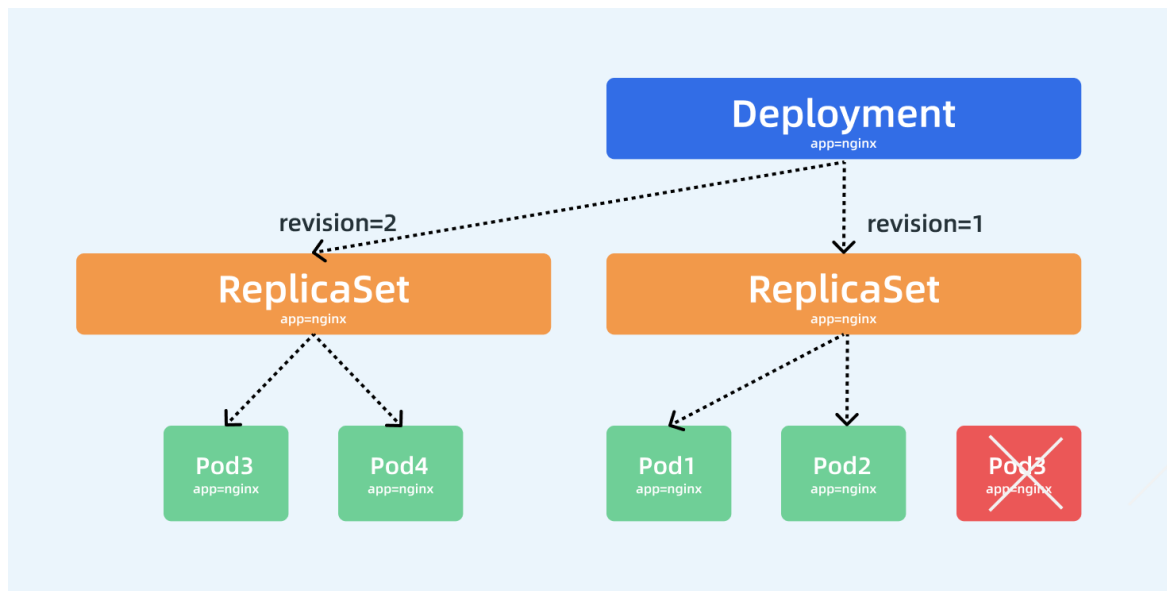


• Deployment

在实际的生产环境中ReplicaSet也很少直接使用，而被用于运行无状态工作负载使用最多的是Deployment，比ReplicaSet更加高级的工作负载类型。

因为Deployment实际是通过管理ReplicaSet来提供和ReplicaSet完全一样的功能，并且在此基础上通过对于多ReplicaSet的管理实现了滚动更新能力。从和ReplicaSet资源定义上来看，其主要多出了如下关键字段：

- **Strategy:** 更新策略，支持Recreate和RollingUpdate两种策略。并且通过定义MaxSurge和MaxUnavailable来设置在更新过程中最多可多出来的Pod数目以及最大不可用的Pod数目，以此来保证在更新过程中Pod对外服务是尽可能可用的；
- **RevisionHistoryLimit:** 保留更新历史数目限制，由于Deployment管理多ReplicaSet，因此每次更新实际都是新建了一个ReplicaSet，并将老ReplicaSet缩容，新ReplicaSet扩容的一个过程。这样便可以随时进行回滚到历史版本，此处的历史限制决定了可以回滚到哪些版本。



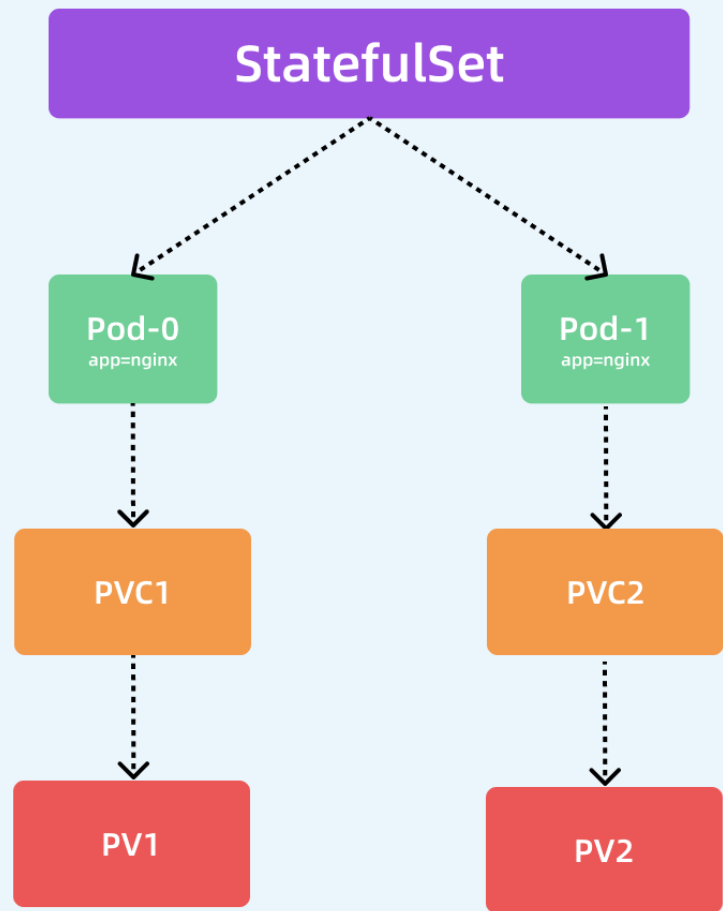
- **StatefulSet**

我们在前文中描述的ReplicaSet和Deployment主要用于无状态工作负载的场景（即不需要将数据持久化到本地的应用，比如web应用），而在面对有状态工作负载的场景（即需要将数据持久化到本地，比如MySQL数据库），Kubernetes设计实现了StatefulSet用于该需求。StatefulSet具有以下特性：

- 稳定的、唯一的网络标识符（按照负载名称加序号的方式命名，比如sts-0）
- 稳定的、持久化的存储（volumes处定义PVC并进行动态创建）
- 有序的、优雅的部署和缩放（正序扩容Pod，倒序缩容Pod）
- 有序的、优雅的删除和终止（正序创建Pod，倒序删除Pod）
- 有序的、自动滚动更新（OnDelete、RollingUpdate）

除了有序的管理Pod创建、删除、扩缩外，还可以以并行的方式进行操作，可以根据不同的应用需求进行设置。

除此之外，StatefulSet控制器还会为每个工作负载都创建一个Headless Service，用于方便通过Service Name直接解析出Pod的IP列表，而不是解析出一个虚拟IP在由iptables或者ipvs规则转发到Pod中。

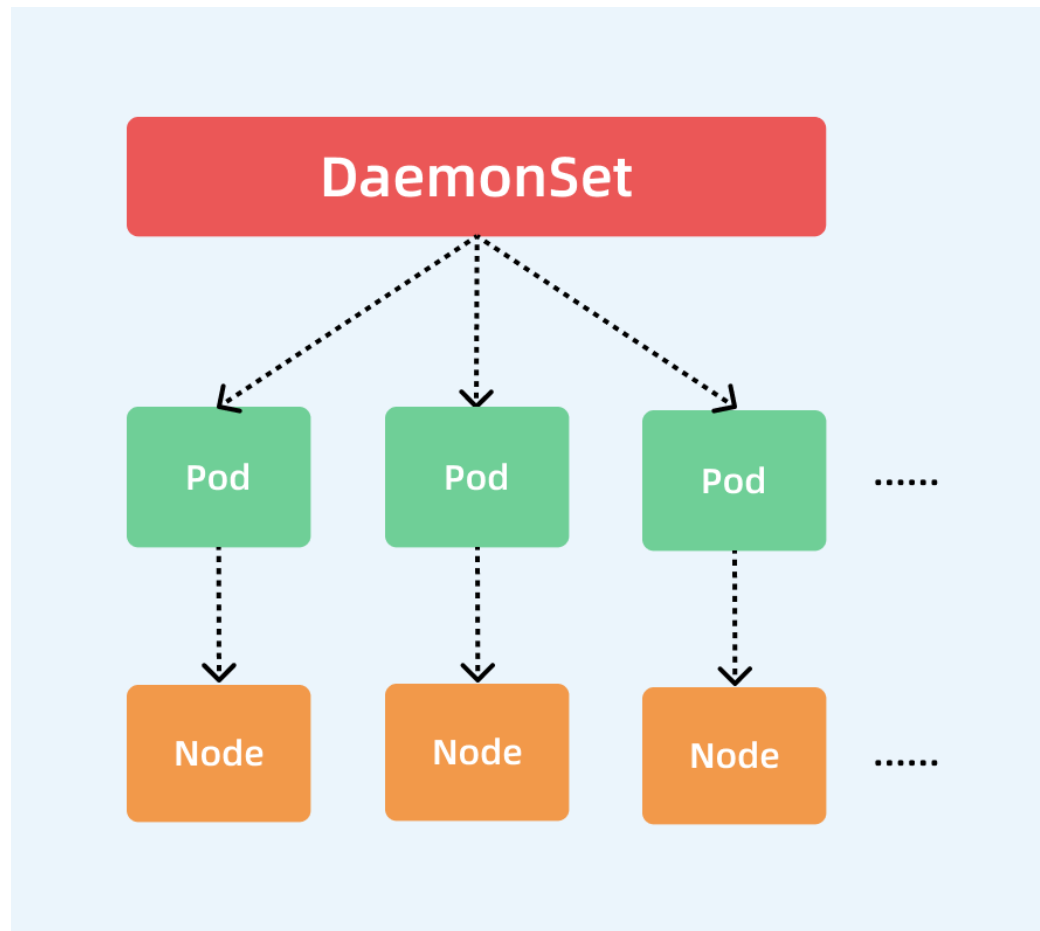


- **DaemonSet**

在某些需求场景中希望每个节点都部署一个守护进程来完成节点管理和监控采集等功能，比如我们熟知的Flannel、Prometheus的Node Exporter以及日志采集的Fluentd等。Kubernetes为此设计实现了DaemonSet工作负载类型来满足这类需求。

DaemonSet类型工作负载并不需要经过调度器调度，因为其在创建Pod时直接设置了nodeName字段，使得调度器不会经过任何调度策略处理；另外针对有污点的Node，如果Pod没有设置任何容忍则默认不会进行部署，而不是向其他类型的工作负载会创建Pod，但是因为调度条件不满足处于Pending状态。

- **UpdateStrategy**: 更新策略，支持OnDelete和RollingUpdate两种策略。
由于DaemonSet的Pod不可能超出Node数目，所以其未定义MaxSurge字段，而是定义了MaxUnavailable来设置在更新过程中最最大不可用的Pod数目；
- **RevisionHistoryLimit**: 保留的更新历史数目限制，默认值为10。



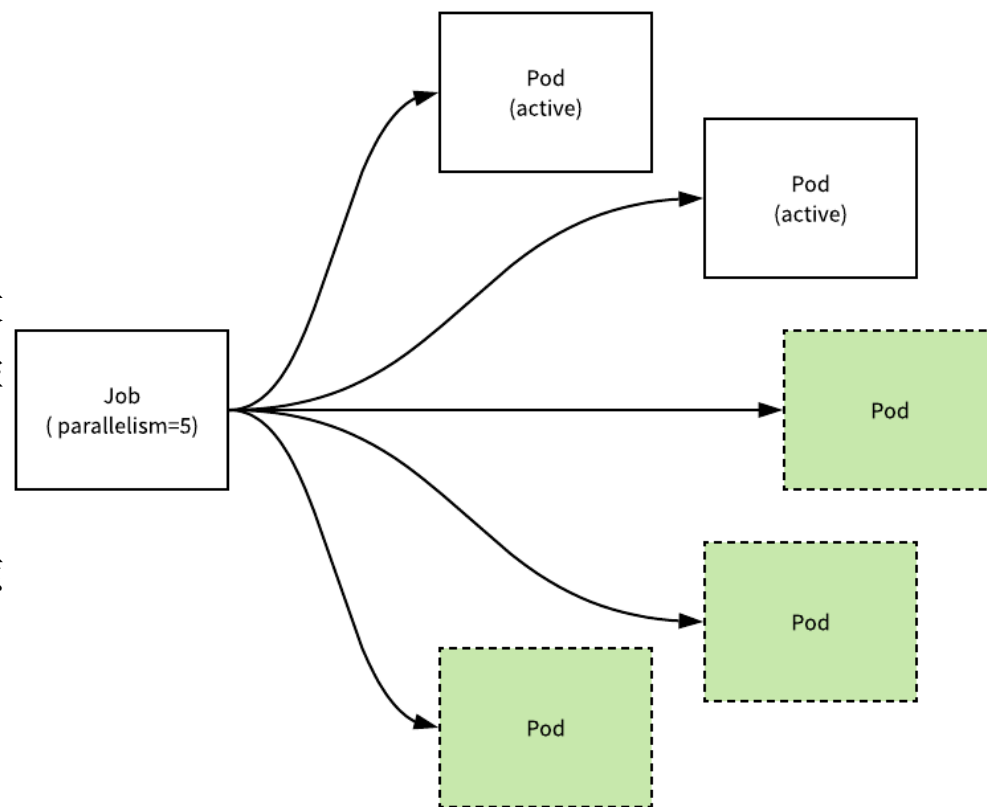
• Job

在一些批处理和分析场景中，希望作业执行完毕即退出，即一次性的任务。

Kubernetes设计实现了Job类型工作负载用来支持。同前文所述的Deployment和Statefulset等有比较本质的区别是，Job所管理的Pod不是常驻型的服务（最终都会达到Completed或者Failed状态），因此在执行失败时不会无条件的尝试恢复，而是根据具体的重启策略进行重试。主要包括OnFailure和Never两种，分别在失败情况下重启以及永不重启。Always是不支持的，因为这和Job的定位不匹配。Job控制器在尝试重启时会遵循一定的退避策略，退避上限通过backoffLimit进行设置。

除此之外Job中也没有replicas的定义，而是变为parallelism和completions的设置。其中parallelism表示同时并行执行的Pod数目，这会涉及到具体可用资源是否能够同时满足这么多Pod运行的问题。completions则表示有多少个Pod达到了Completed状态，一旦达到Job则认为完成不在启动新的Pod。

当前我们一提到批处理都会考虑大数据批处理引擎MapReduce、Spark等，而这些批处理框架通常无法完全用Job进行表示，因此Job实际只能处理一些较为简单的作业。

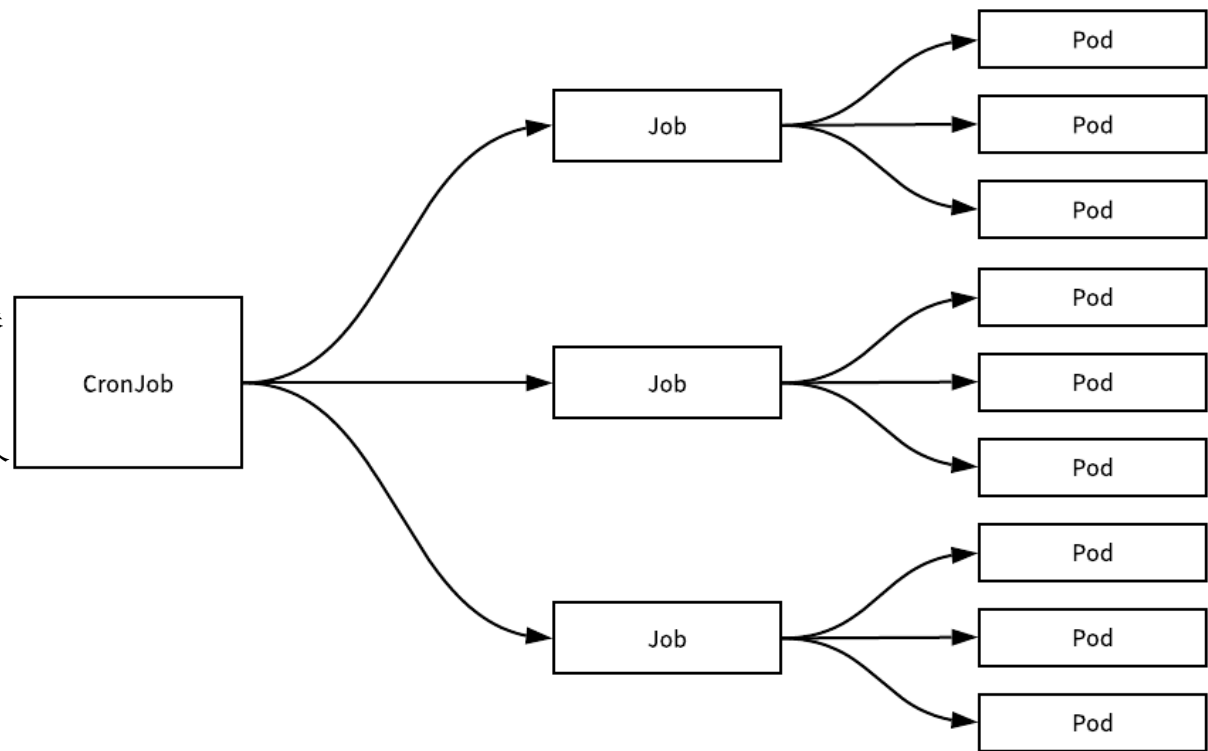


• CronJob

既然有跑批的需求，自然就有定时调度处理的需求。Kubernetes中也默认实现了CronJob来满足此类需求。

CronJob复用了Job模板的定义，并增加了如下关键字段：

- **Schedule:** cron表达式，声明按照什么样的时间规则调度；
- **ConcurrencyPolicy:** 并发策略，支持设置Allow（允许并发执行，启动新Job）、Forbid（禁止并发执行，跳过本次）、Replace（替换正在执行中的Job）三种策略，默认为Allow；
- **Suspend:** 挂起后续的执行计划，默认为false。比如需要跑完10个Pod，而当前仅跑完了5个，那么剩余5个将不会再跑；

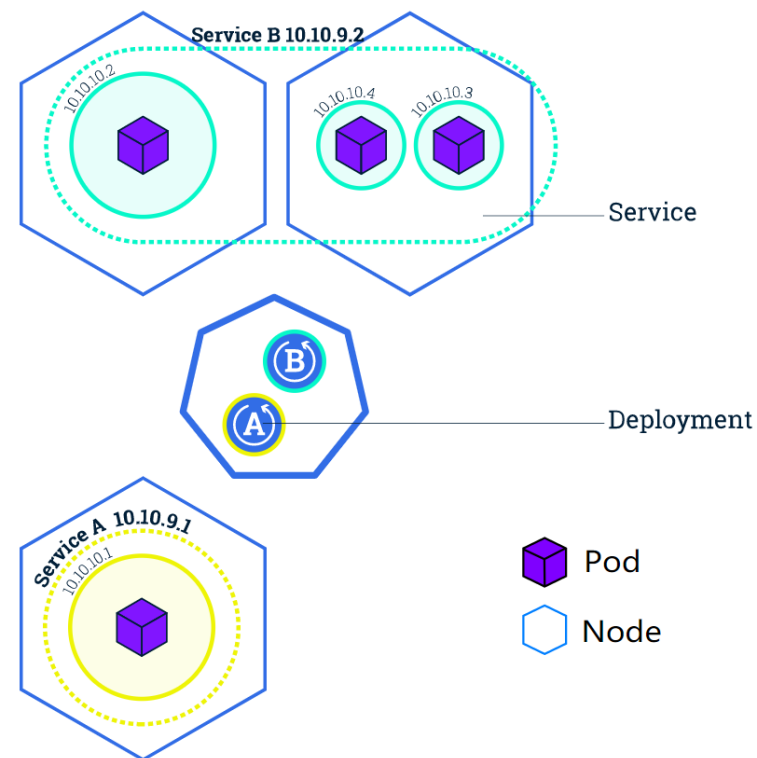


➤ Service

- 是一个面向微服务架构的设计，解决容器集群的负载均衡问题。
- 解决Pod地址可变的问题。Pod可能故障或重启，所以地址是可变的。
- 一个Service可以看作是一组提供某一类功能的Pod的对外访问接口。
- 分配不随Pod位置变化而变化的虚拟访问地址（Cluster IP）。
- Service作用于哪些Pod是通过Label Selector定义的。

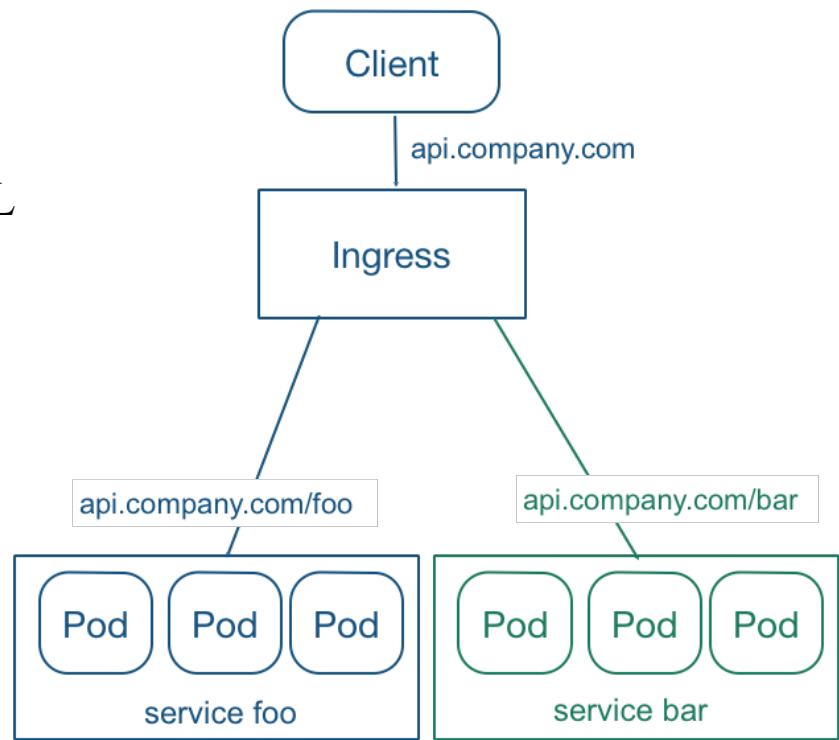
➤ 种类

- Cluster IP：最常见的类型，也是默认类型，只能集群内部相互访问。
- Node Port：可以在集群外部通过访问Node上的IP+Port进行访问。



➤ Ingress

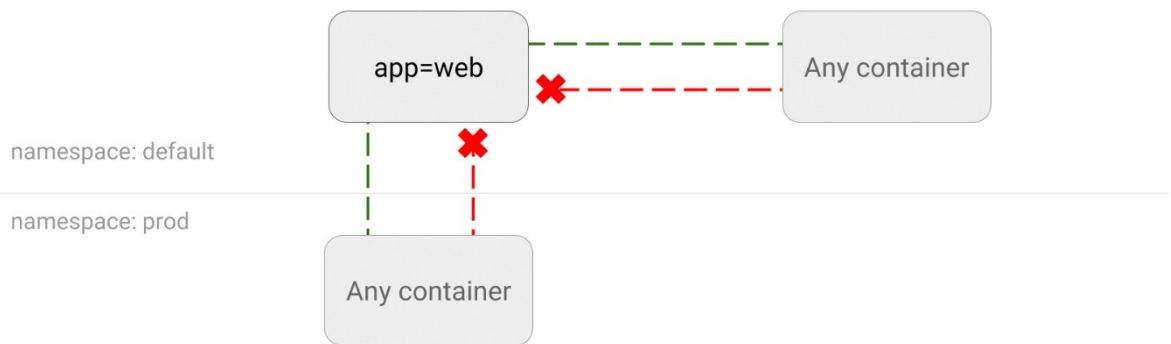
- service 和 pod 的 IP 仅可在集群内部访问。
- Ingress 是为进入集群的请求提供路由规则的集合。
- Ingress 可以给 service 提供集群外部访问的 URL、负载均衡、SSL 终止、HTTP 路由等。
- 通常需要部署一个 Ingress Controller，监听 Ingress 和 service 的变化，并根据规则配置负载均衡并提供访问入口。



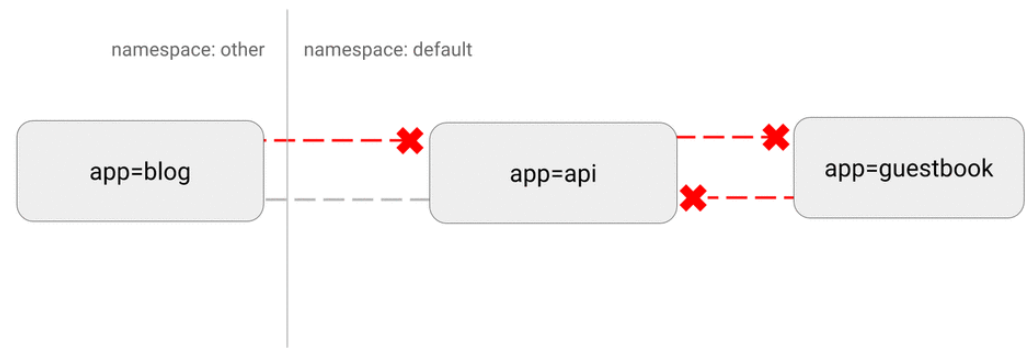
➤ NetworkPolicy

- NetworkPolicy 提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。
- 默认情况下，所有 Pod 之间是全通的。每个 Namespace 可以配置独立的网络策略，来隔离 Pod 之间的流量。
- 通过使用标签选择器（包括 namespaceSelector 和 podSelector）来控制 Pod 之间的流量

禁止访问指定服务



禁止namespace中所有Pod之间的互相访问



➤ PersistentVolume

- PersistentVolume即PV，是持久化存储资源，可以是本地存储，也可以是网络存储。PV 跟 Volume (卷) 类似，不过会有独立于 Pod 的生命周期。
- PV 的访问模式（accessModes）有三种：
 - ✓ ReadWriteOnce (RWO)：是最基本的方式，可读可写，但只支持被单个节点挂载。
 - ✓ ReadOnlyMany (ROX)：可以以只读的方式被多个节点挂载。
 - ✓ ReadWriteMany (RWX)：这种存储可以以读写的方式被多个节点共享。不是每一种存储都支持这三种方式，像共享方式，目前支持的还比较少，比较常用的是 NFS。在 PVC 绑定 PV 时通常根据两个条件来绑定，一个是存储的大小，另一个就是访问模式。
- PV 的回收策略（persistentVolumeReclaimPolicy，即 PVC 释放卷的时候 PV 该如何操作）也有三种：
 - ✓ Retain，不清理，保留 Volume（需要手动清理）
 - ✓ Recycle，删除数据，即 `rm -rf /thevolume/*`（只有 NFS 和 HostPath 支持）
 - ✓ Delete，删除存储资源，比如删除 AWS EBS 卷（只有 AWS EBS, GCE PD, Azure Disk 和 Cinder 支持）

➤ PersistentVolumeClaim

- PersistentVolumeClaim即PVC，是对PV的请求声明。可以声明申请存储空间大小和访问模式。
- 存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的。
 - ✓ PV和Node是资源的提供者，根据集群的基础设施变化而变化，由K8s集群管理员配置。
 - ✓ PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，有K8s集群的使用者即服务的管理员来配置。

➤ StorageClass

- StorageClass提供动态创建PV的能力，不仅节省了管理成本，还可以封装不同类型的存储供 PVC 选用。
- StorageClass 包括四个部分：
 - ✓ **provisioner**: 指定 Volume 插件的类型，包括内置插件（如 `kubernetes.io/glusterfs`）和外部插件（如 `external-storage` 提供的 `ceph.com/cephfs`）。
 - ✓ **mountOptions**: 指定挂载选项，当 PV 不支持指定的选项时会直接失败。比如 NFS 支持 `hard` 和 `nfsvers=4.1` 等选项。
 - ✓ **parameters**: 指定 provisioner 的选项，比如 `kubernetes.io/aws-ebs` 支持 `type`、`zone`、`iopsPerGB` 等参数。
 - ✓ **reclaimPolicy**: 指定回收策略，同 PV 的回收策略。

➤ AdmissionController

- 即准入控制器，位于 API Server 中，在对象被持久化之前，准入控制器拦截对 API Server 的请求，一般用来做身份验证和授权。其中包含两个特殊的控制器：

MutatingAdmissionWebhook 和 ValidatingAdmissionWebhook。

➤ ValidatingWebhookConfiguration

- 对应ValidatingAdmissionWebhook，用来做验证准入控制的相关配置。
- 在实现CRD字段值合法校验等场景中使用。

➤ MutatingWebhookConfiguration

- 对应MutatingAdmissionWebhook，用来做修改准入控制的相关配置。
- 在实现CRD字段默认值等场景中使用。

➤ ConfigMap

- ConfigMap 用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。
- ConfigMap 可以通过两种方式在 Pod 中使用，分别为：设置环境变量和在 Volume 中直接挂载文件或目录。

➤ Secret

- Secret 解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者 Pod Spec 中。Secret 可以以 Volume 或者环境变量的方式使用。
- Secret 有三种类型：
 - ✓ **Opaque**: base64 编码格式的 Secret，用来存储密码、密钥等；但数据也通过 `base64 --decode` 解码得到原始数据，所有加密性很弱。
 - ✓ **kubernetes.io/dockerconfigjson**: 用来存储私有 docker registry 的认证信息。
 - ✓ **kubernetes.io/service-account-token**: 用于被 serviceaccount 引用。serviceaccount 创建时 Kubernetes 会默认创建对应的 secret。Pod 如果使用了 serviceaccount，对应的 secret 会自动挂载到 Pod 的 `/run/secrets/kubernetes.io/serviceaccount` 目录中。

➤ RBAC

- K8s在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control, RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control, ABAC），RBAC主要是引入了角色（Role）和角色绑定（RoleBinding）的抽象概念。
- 在ABAC中，K8s集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户再跟一个或多个角色相关联。
- 显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和使用更容易扩展和重用。

➤ ServiceAccount


- Service Account 是为了方便 Pod 里面的进程调用 Kubernetes API 或其他外部服务而设计的。它与 User Account 不同：
 - ✓ User Account 是为人设计的，而 Service Account 则是为 Pod 中的进程调用 Kubernetes API 而设计；
 - ✓ User Account 是跨 namespace 的，而 Service Account 则是仅局限它所在的 namespace；
 - ✓ 每个 namespace 都会自动创建一个 default Service Account
 - ✓ Token controller 检测 Service Account 的创建，并为它们创建 [secret](#)
 - ✓ 开启 ServiceAccount Admission Controller 后
 - 每个 Pod 在创建后都会自动设置 spec.serviceAccountName 为 default（除非指定了其他 ServiceAccount）
 - 验证 Pod 引用的 Service Account 已经存在，否则拒绝创建
 - 如果 Pod 没有指定 ImagePullSecrets，则把 Service Account 的 ImagePullSecrets 加到 Pod 中
 - 每个 container 启动后都会挂载该 Service Account 的 token 和 ca.crt 到 /var/run/secrets/kubernetes.io/serviceaccount

➤ Role/ClusterRole

- Kubernetes的核心组件通常会依赖kubeconfig文件获得最大的操作资源权限，但是对于普通组件来说权限过大，因此需要依赖细粒度的权限规则定义进行限制。
- Role代表一组操作Kubernetes资源的权限规则集合，作用域为某个Namespace。
- ClusterRole同样也是代表一组操作K8s资源的权限规则集合，但是作用域为整个集群。

• RoleBinding/ClusterRoleBinding

- Role和ClusterRole仅仅是声明了可访问Kubernetes资源的权限规则集合，需要依赖角色绑定来将权限实际作用在具体的容器中。
- RoleBinding和ClusterRoleBinding即是用来关联Role和ClusterRole与ServiceAccount的，使容器内进程在访问API Server时受到细粒度的访问控制。

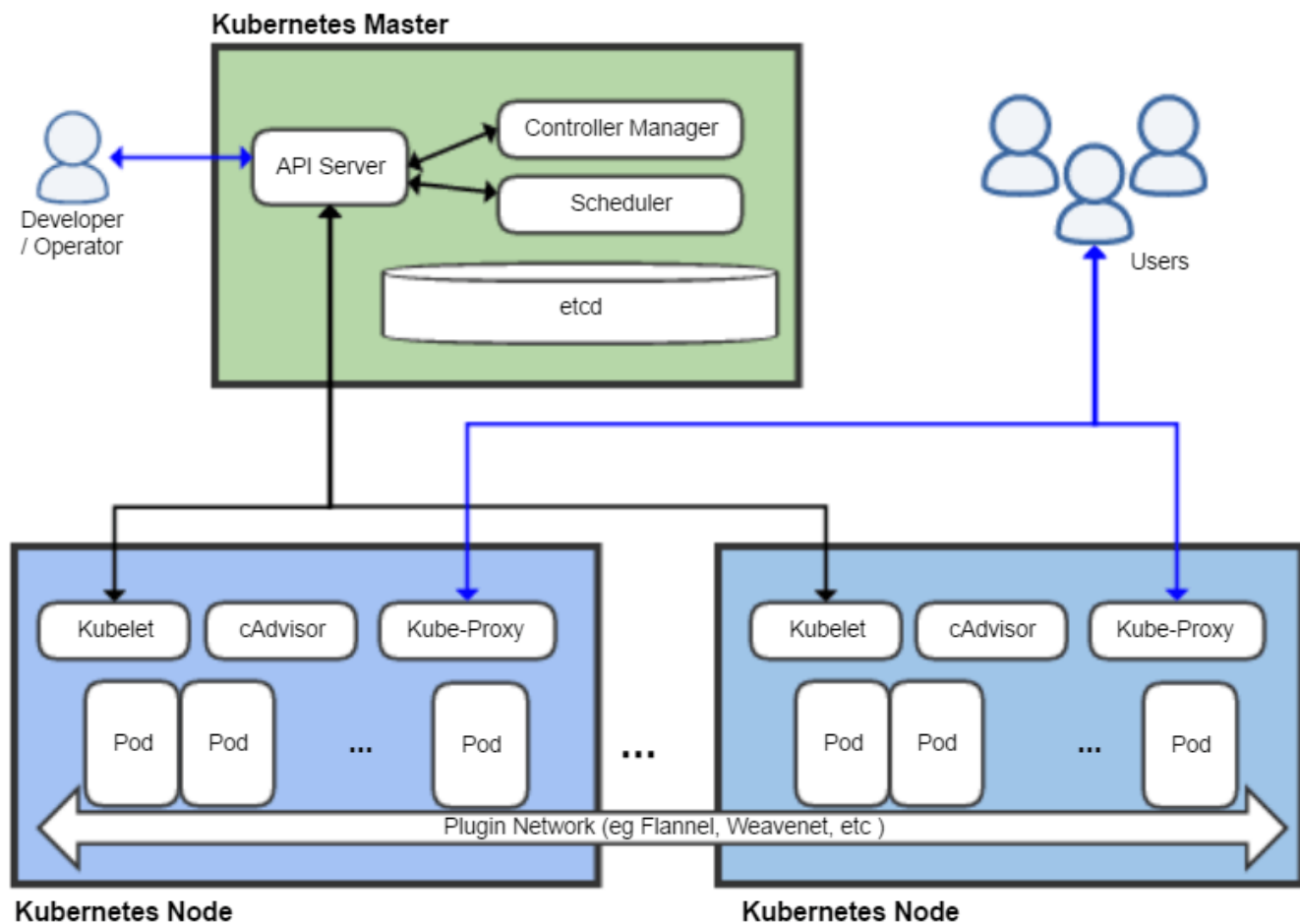


3 chapter

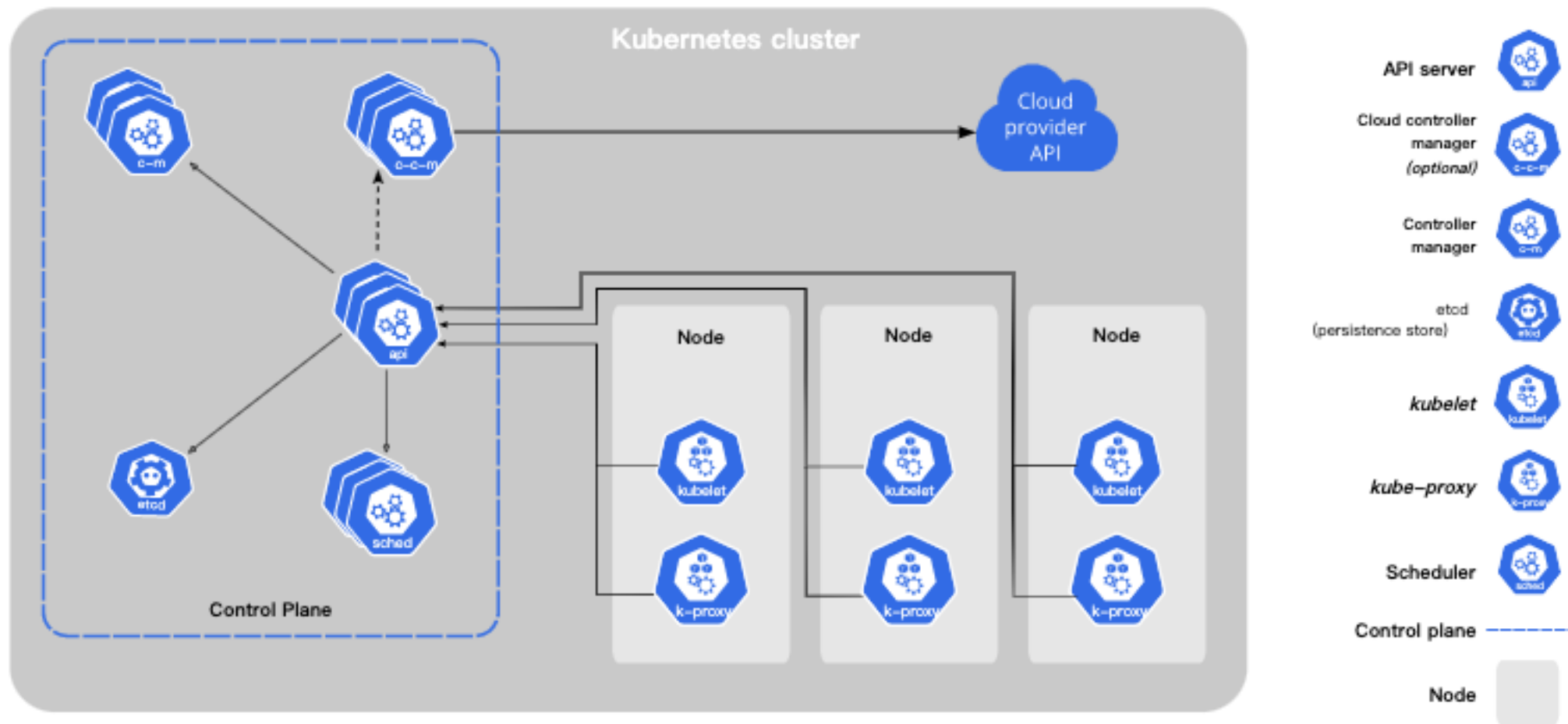
Kubernetes 架构

✓ Kubernetes 架构

➤ 系统架构



➤ 系统架构

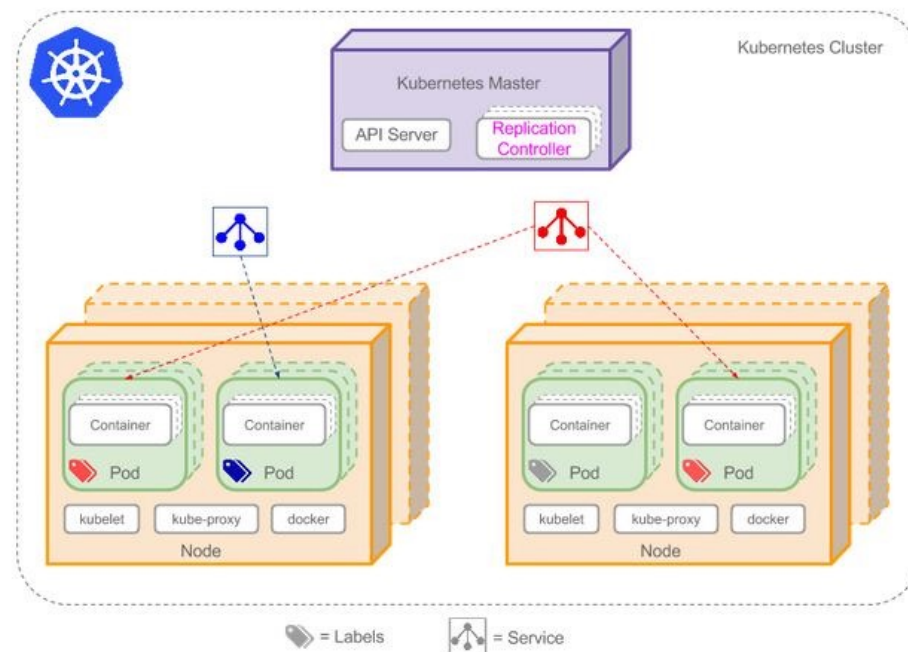


➤ Master

- Kubernetes集群的大脑，提供所有与管理相关的操作，如调度、监控、对资源的操作等
- 为了实现高可用，可以运行多个Master

➤ Node

- 承担工作负载的节点，如运行Pod
- Node负责监控并汇报容器的状态，同时根据Master的要求管理容器生命周期



➤ Etcd

- Etcd 是CoreOS 开源的一个高可用强一致性的分布式存储服务
- Kubernetes 使用Etcd 作为数据存储后端，把需要记录的pod、rc、service 等资源信息存储在Etcd 中。
- Etcd 使用raft 算法将一组主机组成集群，raft 集群中的每个节点都可以根据集群运行的情况在三种状态间切换：follower, candidate 与 leader。leader 和 follower 之间保持心跳。
- 如果follower 在一段时间内没有收到来自leader 的心跳，就会转为candidate，发出新的选主请求。
- 当一个节点获得了大于一半节点的投票后会转为leader

➤ kube-apiserver

- REST API服务端，接收来自客户端和其他组件的请求，更新Etcd中的数据，响应对API资源操作的请求。
- 作为Kubernetes系统的入口，封装了核心对象的增删改查操作，以REST API方式提供给外部客户端和内部组件调用。
- 提供资源对象的唯一操作入口，其他组件必须通过它提供的API来操作资源，只有它与Etcd通信，其他组件通过它访问集群状态。

➤ 功能

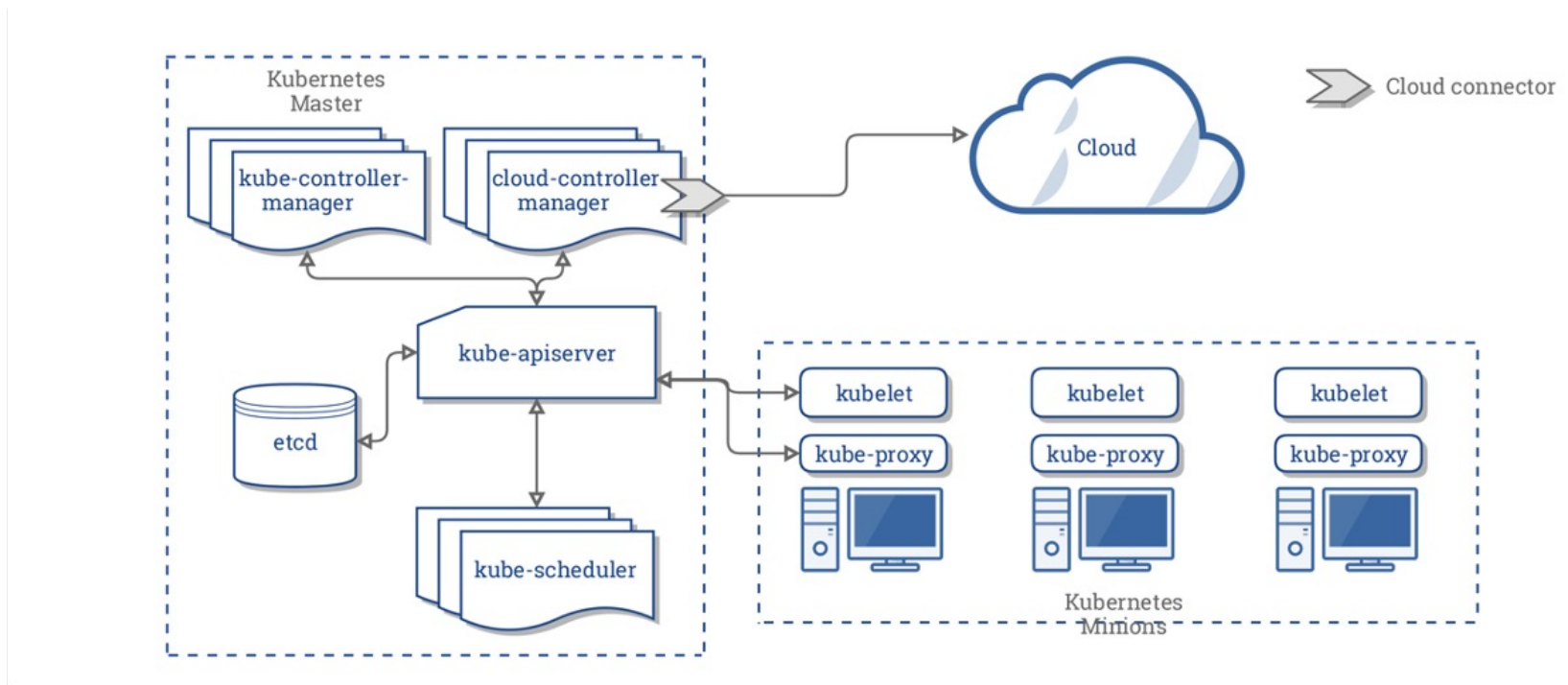
- 提供集群管理的REST API接口，如认证、授权、数据校验、集群状态变更等。
- 作为与其他模块进行数据交互和通信的枢纽。
- 是资源配额控制的入口。
- 保证集群状态访问的安全。

➤ kube-scheduler

- 负责集群的资源调度，为新建的Pod分配机器
- 资源调度工作抽象成一个组件，可以很方便的替换成其他的调度器

➤ kube-controller-manager

- 负责执行各种控制器，保证Kubernetes的正常运行。包括故障检测、自动扩展、滚动更新等



➤ kubelet

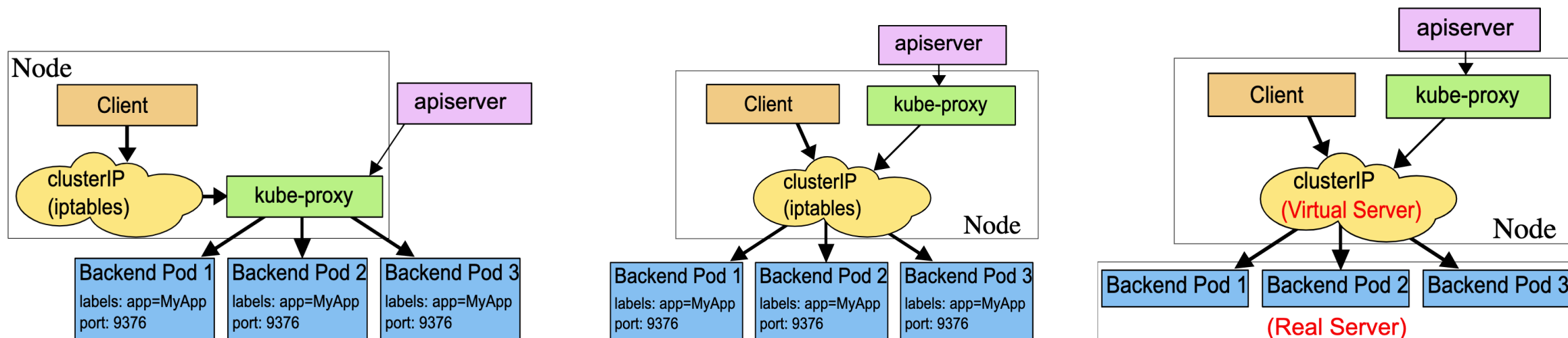
- 是Node上的pod管家，可以理解成【Server-Agent】架构中的agent。
- 可以与Yarn中Node Manager进行类比。

➤ 作用

- 负责Node节点上pod的创建、修改、监控、删除等全生命周期的管理。
- 在API Server上注册节点信息，定期向Master汇报节点资源使用情况，并通过cAdvisor监控容器和节点资源。

➤ kube-proxy

- 为了解决外部网络能够访问跨机器集群中容器提供的应用服务而设计的。
- 当service创建时，与Service同名的endpoint（服务端口）列表会被创建，服务端口列表的形式如: pod_ip1:port,pod_ip2:port。
- kube-proxy会将对svc的请求，转发到对应的后端pod。
- 三种代理模式: userspace & iptables & ipvs。TCOS默认使用ipvs模式。

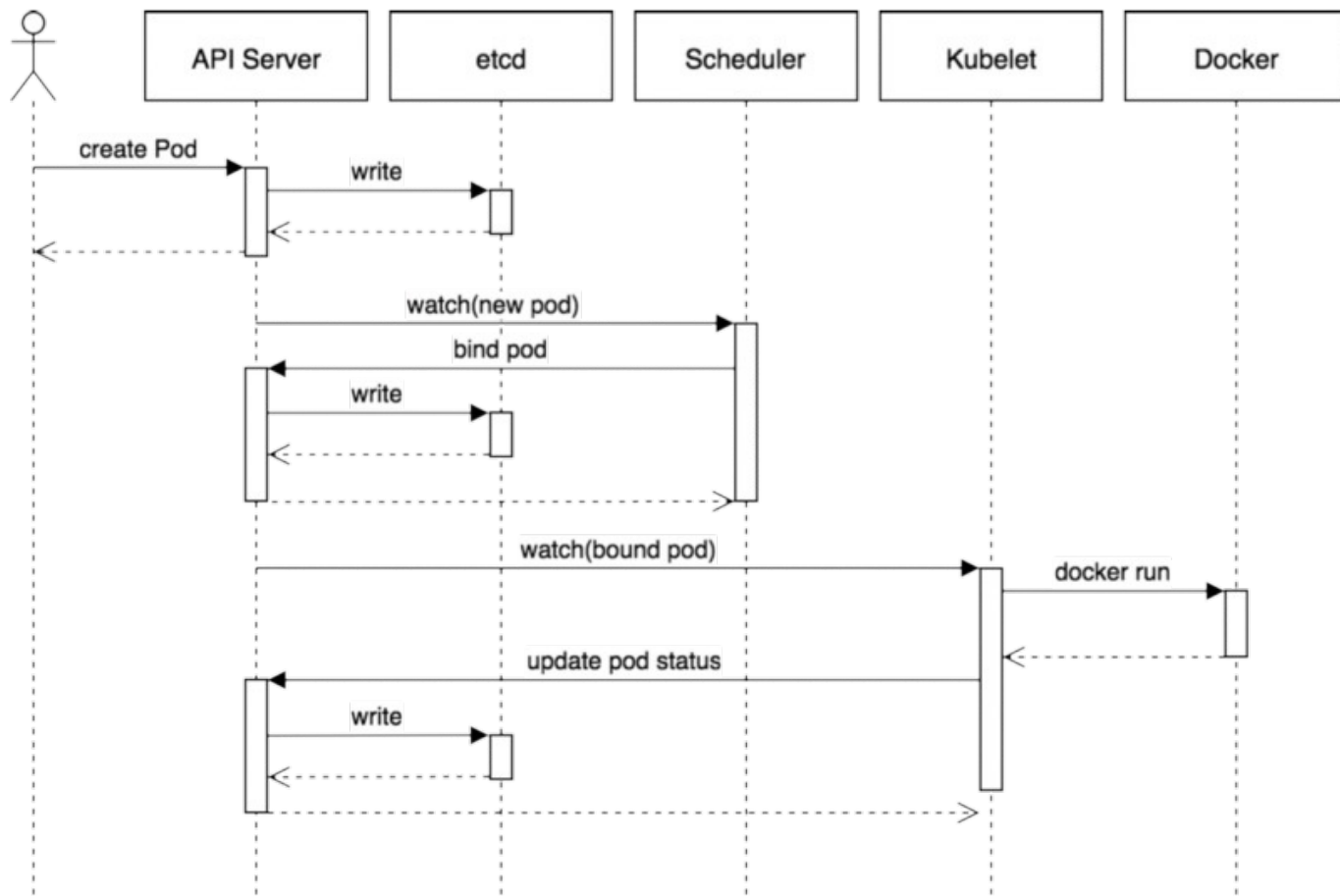


➤ 扩展组件

- CNI 插件：如Flannel、Calico、Antrea，为Kubernetes集群打通容器网络
- CoreDNS：负责为整个集群提供 DNS 服务
- Ingress Controller：为服务提供外网七层协议的访问入口
- Metrics Server：提供资源监控指标采集接口
- Prometheus + Grafana：提供监控告警和可视化Dashboard
- EFK（Elastic Search + Fluentd + Kibana）：提供集群日志采集、存储与查询

➤ 工作原理

- 典型的创建Pod的流程
 - 1. 用户通过 REST API 创建一个 Pod
 - 2. API Server 将其写入 etcd
 - 3. Scheduler 检测到未绑定 Node 的 Pod, 开始调度并更新 Pod 的 Node 绑定
 - 4. Kubelet 检测到有新的Pod 调度过来, 通过 Container Runtime 运行该 Pod
 - 5. Kubelet 通过 Container Runtime 取到 Pod 状态, 并更新到 API Server 中





Q&A

TRANSWARP
星环科技

温故知新

- 思考题1： 1) 部署 nginx deployment 和 nginx svc； 2) 通过 svc 的 nodeport 访问； 3) 对 nginx deployment 扩容

