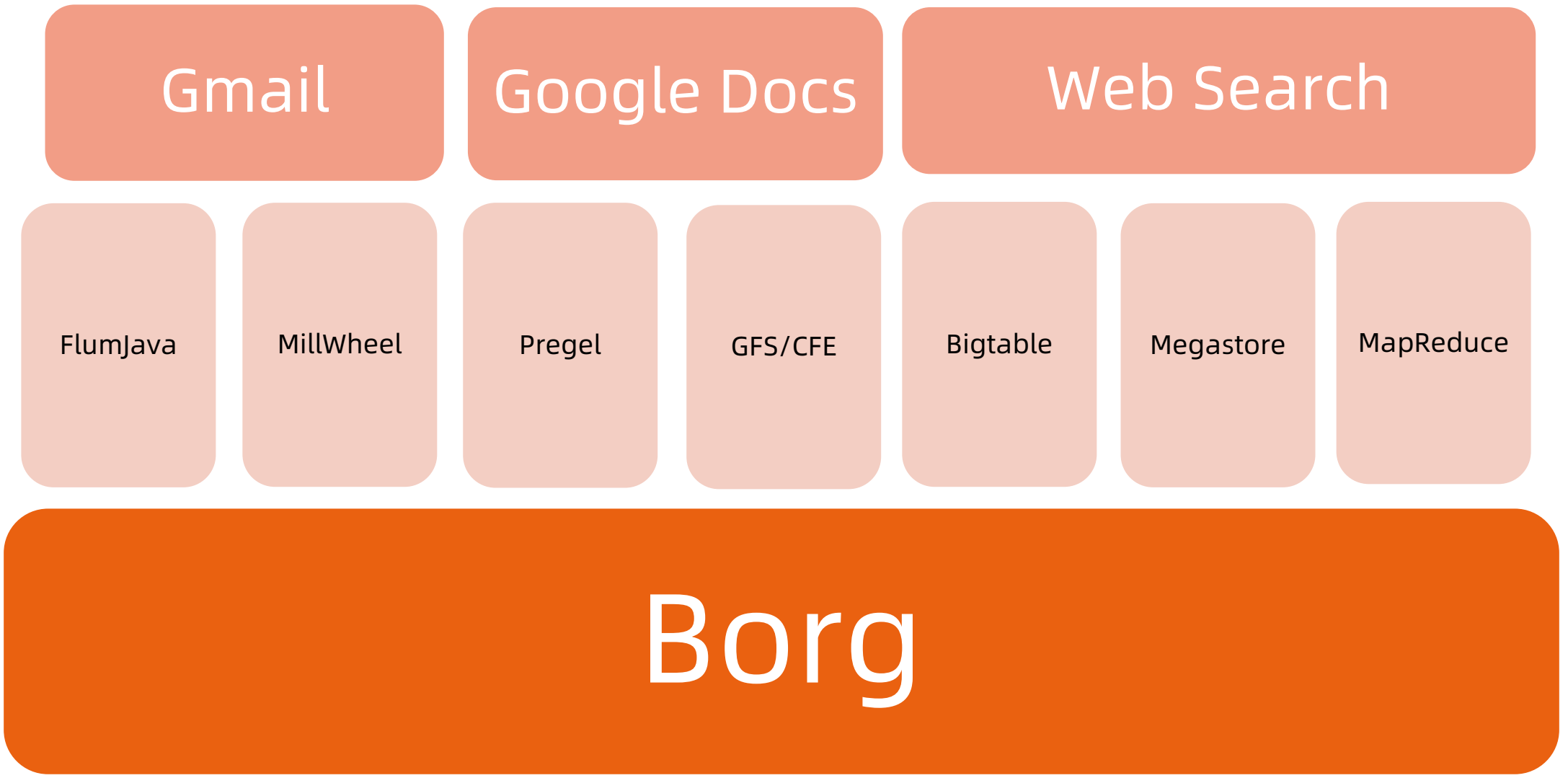
Kubernetes 架构原则和对象设计

孟凡杰

前 eBay 资深架构师

Kubernetes 架构基础

Borg


Gmail
Google Docs
Web Search
FlumJava
MillWheel
Pregel
GFS/CFE
Bigtable
Megastore
MapReduce
Borg

Google Borg 简介

特性

- 物理资源利用率高。
- 服务器共享，在进程级别做隔离。
- 应用高可用，故障恢复时间短。
- 调度策略灵活。
- 应用接入和使用方便，提供了完备的 Job 描述语言，服务发现，实时状态监控和诊断工具。

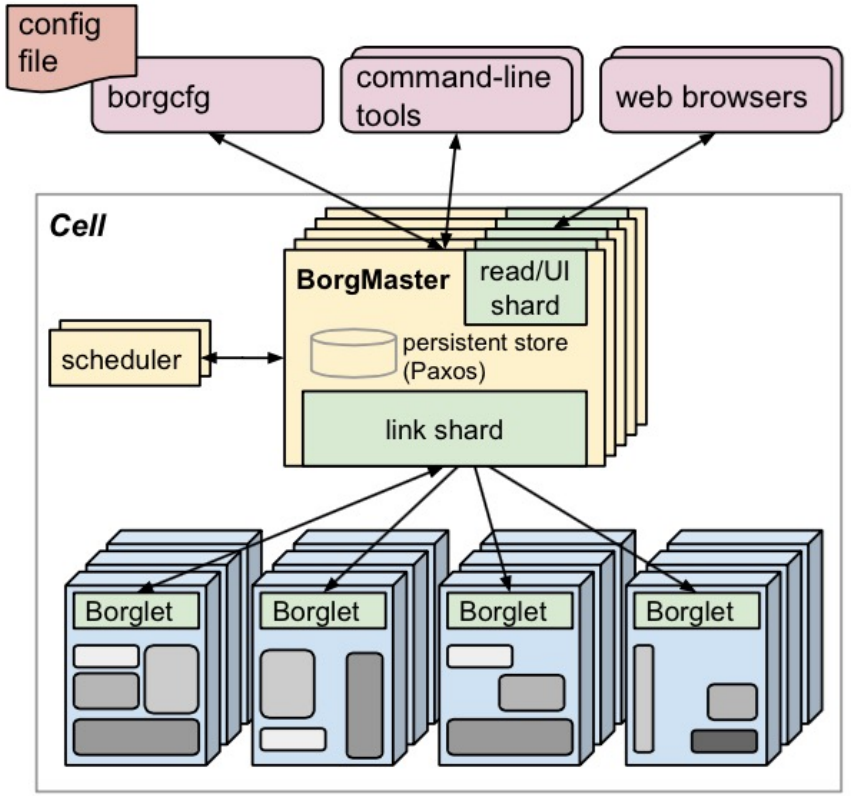
优势

- 对外隐藏底层资源管理和调度、故障处理等。
- 实现应用的高可靠和高可用。
- 足够弹性，支持应用跑在成千上万的机器上。

基本概念

Workload	Cell	Job 和 Task	Naming
• prod：在线任务，长期运行、对延时敏感、面向终端用户等，比如 Gmail, Google Docs, Web Search 服务等。 • non-prod：离线任务，也称为批处理任务（Batch），比如一些分布式计算服务等。	• 一个 Cell 上跑一个集群管理系统 Borg。 • 通过定义 Cell 可以让 Borg 对服务器资源进行统一抽象，作为用户就无需知道自己的应用跑在哪台机器上，也不用关心资源分配、程序安装、依赖管理、健康检查及故障恢复等。	• 用户以 Job 的形式提交应用部署请求。一个 Job 包含一个或多个相同的 Task，每个 Task 运行相同的应用程序，Task 数量就是应用的副本数。 • 每个 Job 可以定义属性、元信息和优先级，优先级涉及到抢占式调度过程。	• Borg 的服务发现通过 BNS（Borg Name Service）来实现。 • 50.jfoo.ubar.cc.borg.google.com 可表示在一个名为 cc 的 Cell 中由用户 uBar 部署的一个名为 jFoo 的 Job 下的第50个 Task。

Borg 架构



Borgmaster 主进程：

- 处理客户端 RPC 请求，比如创建 Job，查询 Job 等。
- 维护系统组件和服务的状态，比如服务器、Task 等。
- 负责与 Borglet 通信。

Scheduler 进程：

- 调度策略
    - Worst Fit
    - Best Fit
    - Hybrid
- 调度优化
    - Score caching：当服务器或者任务的状态未发生变更或者变更很少时，直接采用缓存数据，避免重复计算。
    - Equivalence classes: 调度同一 Job 下多个相同的 Task 只需计算一次。
    - Relaxed randomization: 引入一些随机性，即每次随机选择一些机器，只要符合需求的服务器数量达到一定值时，就可以停止计算，无需每次对 Cell 中所有服务器进行 feasibility checking。

Borglet：

Borglet 是部署在所有服务器上的 Agent，负责接收 Borgmaster 进程的指令。

应用高可用

- 被抢占的 non-prod 任务放回 pending queue，等待重新调度。
- 多副本应用跨故障域部署。所谓故障域有大有小，比如相同机器、相同机架或相同电源插座等，一挂全挂。
- 对于类似服务器或操作系统升级的维护操作，避免大量服务器同时进行。
- 支持幂等性，支持客户端重复操作。
- 当服务器状态变为不可用时，要控制重新调度任务的速率。因为 Borg 无法区分是节点故障还是出现了短暂的网络分区，如果是后者，静静地等待网络恢复更利于保障服务可用性。
- 当某种“任务 @ 服务器”的组合出现故障时，下次重新调度时需避免这种组合再次出现，因为极大可能会再次出现相同故障。
- 记录详细的内部信息，便于故障排查和分析。
- 保障应用高可用的关键性设计原则：无论何种原因，即使 Borgmaster 或者 Borglet 挂掉、失联，都不能杀掉正在运行的服务（Task）。

Borg 系统自身高可用

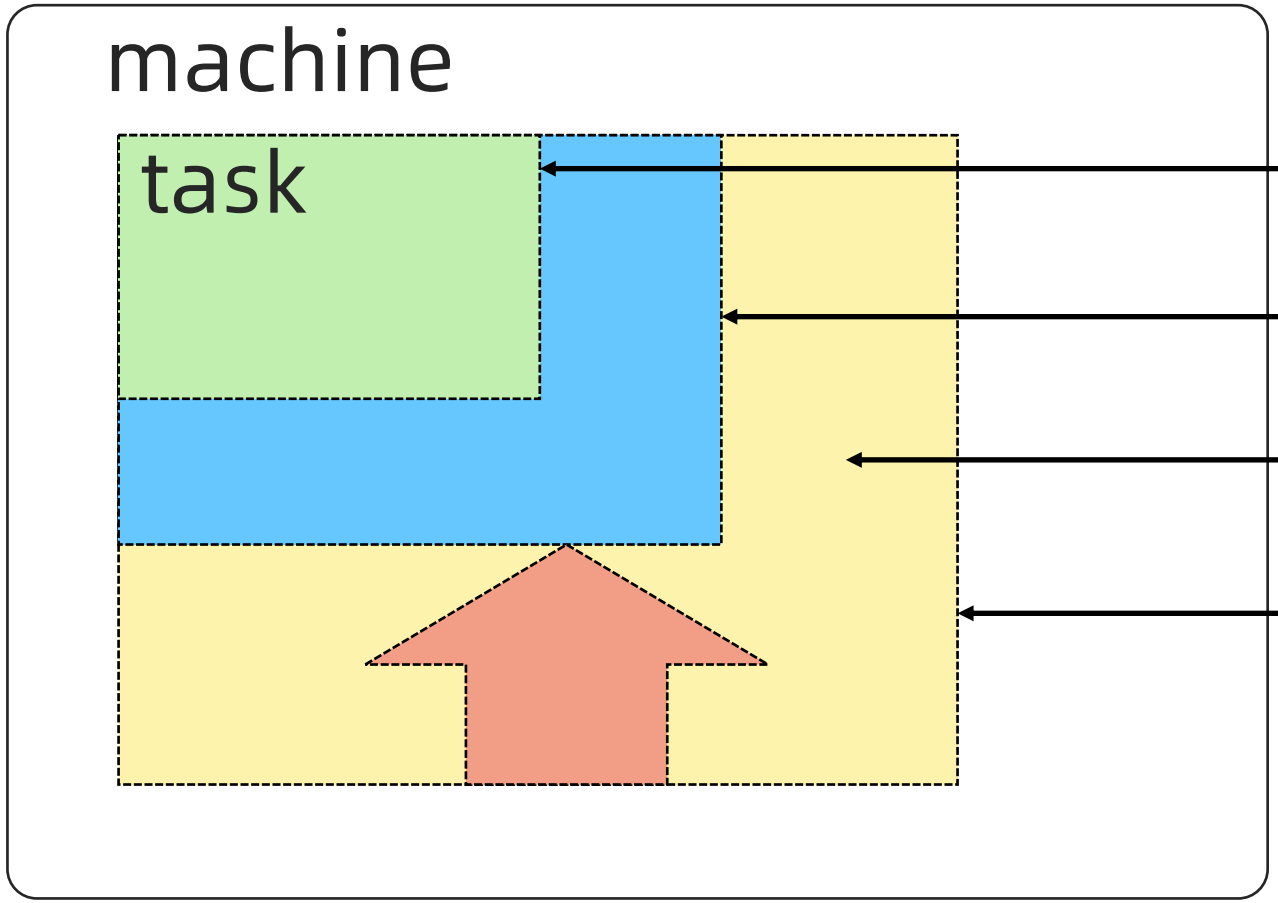
- Borgmaster 组件多副本设计。
- 采用一些简单的和底层（low-level）的工具来部署 Borg 系统实例，避免引入过多的外部依赖。
- 每个 Cell 的 Borg 均独立部署，避免不同 Borg 系统相互影响。

资源利用率

极客时间 训练营

- 通过将在线任务（prod）和离线任务（non-prod，Batch）混合部署，空闲时，离线任务可以充分利用计算资源；繁忙时，在线任务通过抢占的方式保证优先得到执行，合理地利用资源。
- 98% 的服务器实现了混部。
- 90% 的服务器中跑了超过 25 个 Task 和 4500 个线程。
- 在一个中等规模的 Cell 里，在线任务和离线任务独立部署比混合部署所需的服务器数量多出约 20%-30%。可以简单算一笔账，Google 的服务器数量在千万级别，按 20% 算也是百万级别，大概能省下的服务器采购费用就是百亿级别了，这还不包括省下的机房等基础设施和电费等费用。

Brog 调度原理



在 task 启动 300s 后，进行资源回收工作，把保留资源设置为实际使用资源 + 安全资源，并每过几秒再重新计算一次。

隔离性

安全性隔离：

- 早期采用 Chroot jail，后期版本基于 Namespace

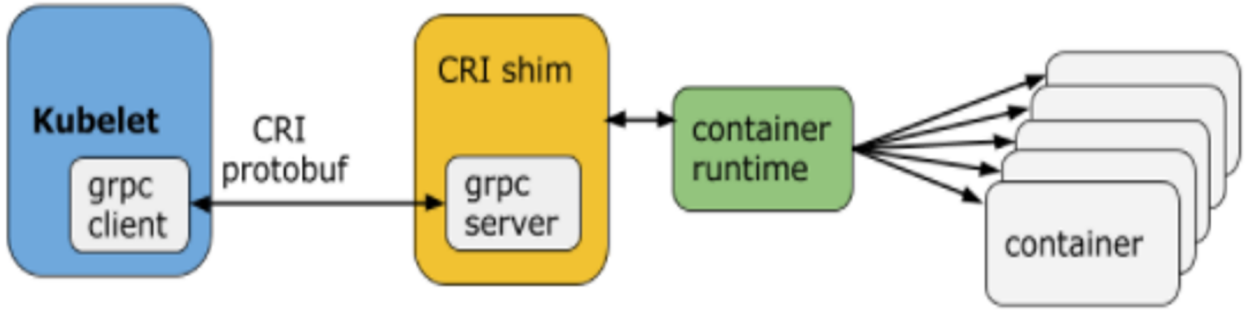
性能隔离：

- 采用基于 Cgroup 的容器技术实现
- 在线任务（prod）是延时敏感（latency-sensitive）型的，优先级高，而离线任务（non-prod，Batch）优先级低
- Borg 通过不同优先级之间的抢占式调度来优先保障在线任务的性能，牺牲离线任务
- Borg 将资源类型分成两类：
  - 可压榨的（compressible），CPU 是可压榨资源，资源耗尽不会终止进程
  - 不可压榨的（non-compressible），内存是不可压榨资源，资源耗尽进程会被终止

什么是 Kubernetes（K8s）？

Kubernetes 是谷歌开源的容器集群管理系统，是 Google 多年大规模容器管理技术 Borg 的开源版本，主要功能包括：

- 基于容器的应用部署、维护和滚动升级；
- 负载均衡和服务发现；
- 跨机器和跨地区的集群调度；
- 自动伸缩；
- 无状态服务和有状态服务；
- 插件机制保证扩展性。



命令式（ Imperative）vs 声明式（ Declarative）

命令式系统关注 “如何做”

在软件工程领域，命令式系统是写出解决某个问题、完成某个任务或者达到某个目标的明确步骤。此方法明确写出系统应该执行某指令，并且期待系统返回期望结果。

声明式系统关注 “做什么”

在软件工程领域，声明式系统指程序代码描述系统应该做什么而不是怎么做。仅限于描述要达到什么目的，如何达到目的交给系统。

声明式（Declaritive）系统规范

极客时间 训练营

命令式：

- 我要你做什么，怎么做，请严格按照我说的做。

声明式：

- 我需要你帮我做点事，但是我只告诉你我需要你做什么，不是你应该怎么做。
- 直接声明：我直接告诉你我需要什么。
- 间接声明：我不直接告诉你我的需求，我会把我的需求放在特定的地方，请在方便的时候拿出来处理。

幂等性：

- 状态固定，每次我要你做事，请给我返回相同结果。

面向对象的：

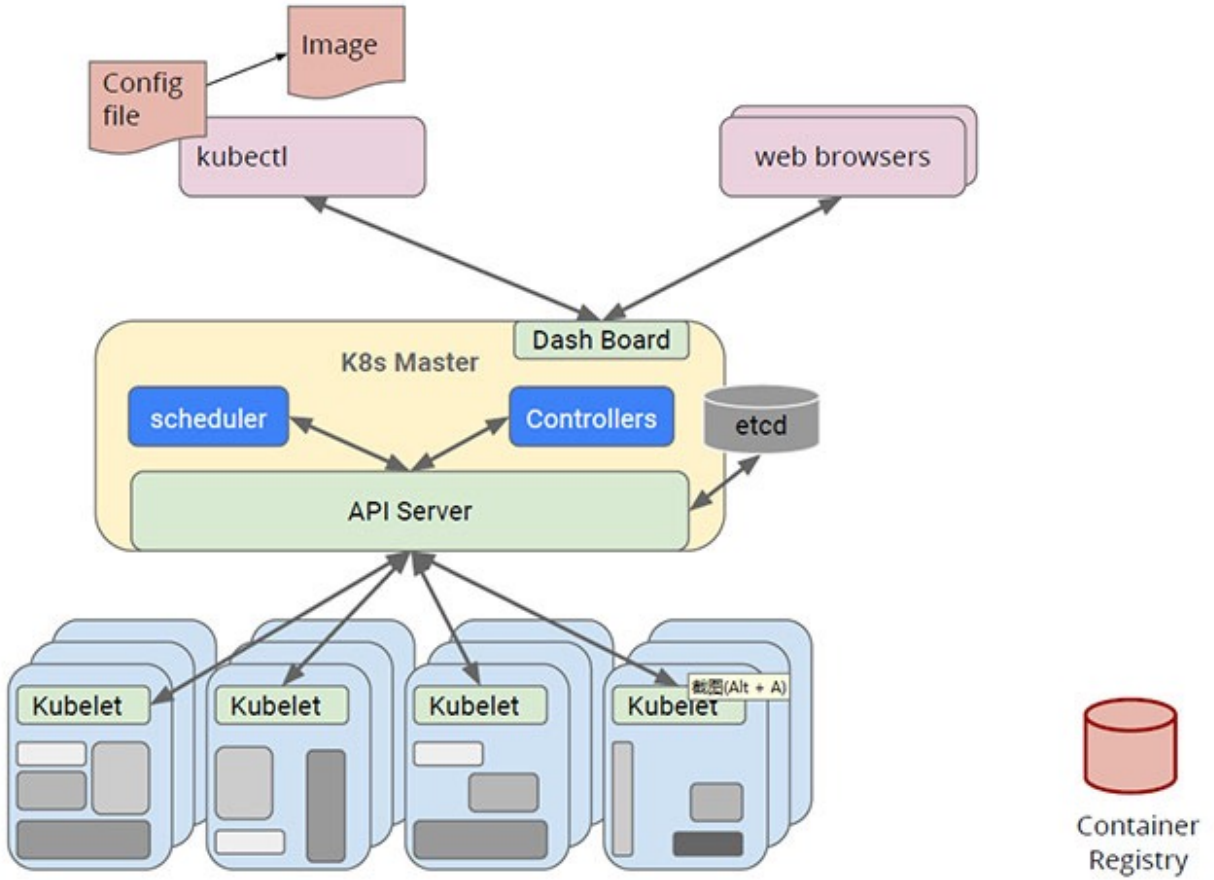
- 把一切抽象成对象。

Kubernetes：声明式系统

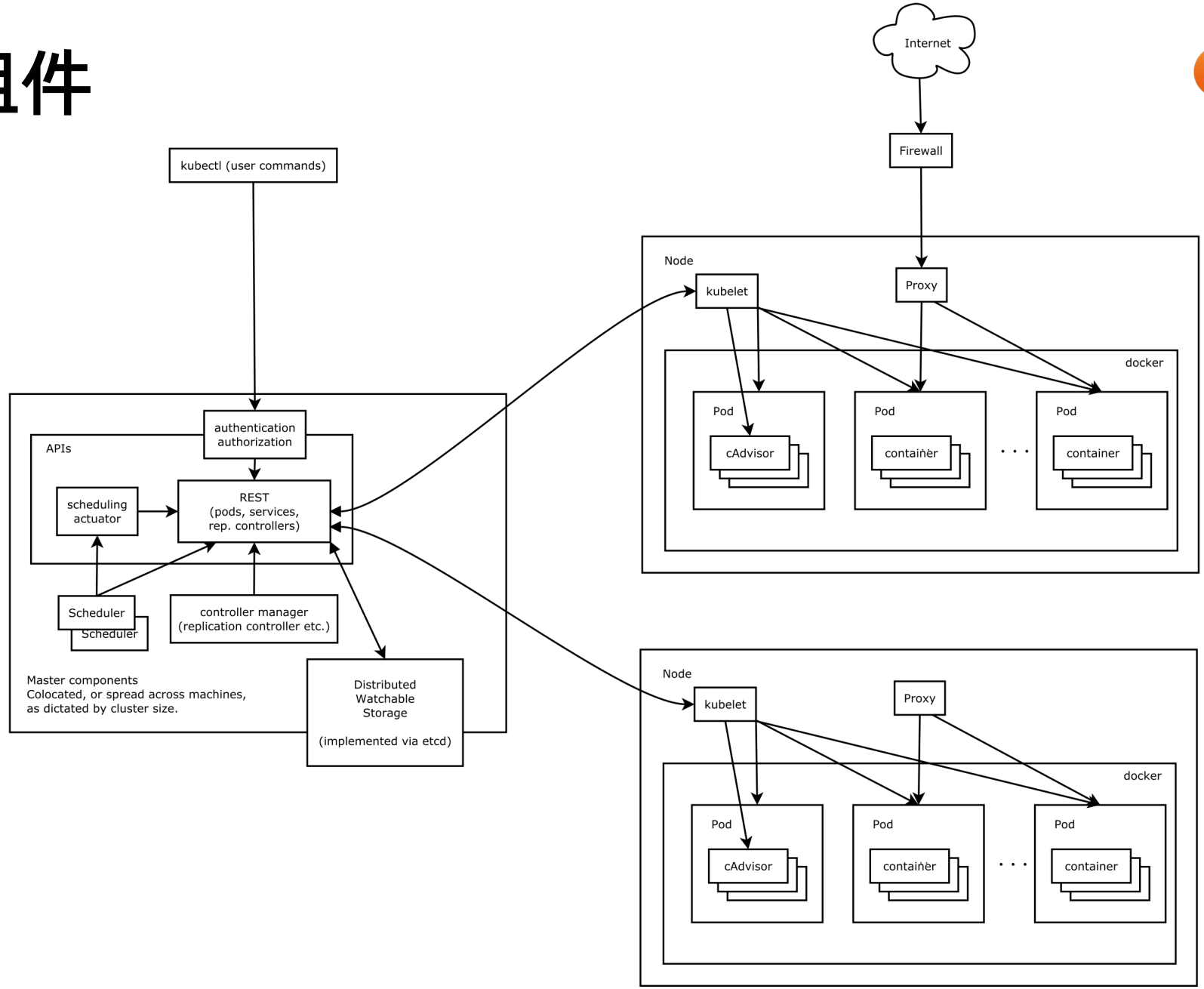
Kubernetes 的所有管理能力构建在对象抽象的基础上，核心对象包括：

- Node：计算节点的抽象，用来描述计算节点的资源抽象、健康状态等
- Namespace：资源隔离的基本单位，可以简单理解为文件系统中的目录结构
- Pod：用来描述应用实例，包括镜像地址、资源需求等。 Kubernetes 中最核心的对象，也是打通应用和基础架构的秘密武器
- Service：服务如何将应用发布成服务，本质上是负载均衡和域名服务的声明

Kubernetes 采用与 Borg 类似的架构



主要组件


Internet
Firewall
kubectl (user commands)
Node
kubelet
Proxy
docker
Pod
cAdvisor
container
Pod
container
Pod
container
authentication
authorization
APIs
scheduling
actuator
REST
(pods, services,
rep. controllers)
Scheduler
Scheduler
controller manager
(replication controller etc.)
Master components
Colocated, or spread across machines,
as dictated by cluster size.
Distributed
Watchable
Storage
(implemented via etcd)
Node
kubelet
Proxy
docker
Pod
cAdvisor
container
Pod
container
Pod
container

Kubernetes 的主节点（Master Node）

极客时间 训练营

API服务器 API Server	这是 Kubernetes 控制面板中唯一带有用户可访问 API 以及用户可交互的组件。API 服务器会暴露一个 RESTful 的 Kubernetes API 并使用 JSON 格式的清单文件（manifest files）。
群的数据存储 Cluster Data Store	Kubernetes 使用“etcd”。这是一个强大的、稳定的、高可用的键值存储，被 Kubernetes 用于长久储存所有的 API 对象。
控制管理器 Controller Manager	被称为“kube-controller manager”，它运行着所有处理集群日常任务的控制器。包括了节点控制器、副本控制器、端点（endpoint）控制器以及服务账户等。
调度器 Scheduler	调度器会监控新建的 pods（一组或一个容器）并将其分配给节点。

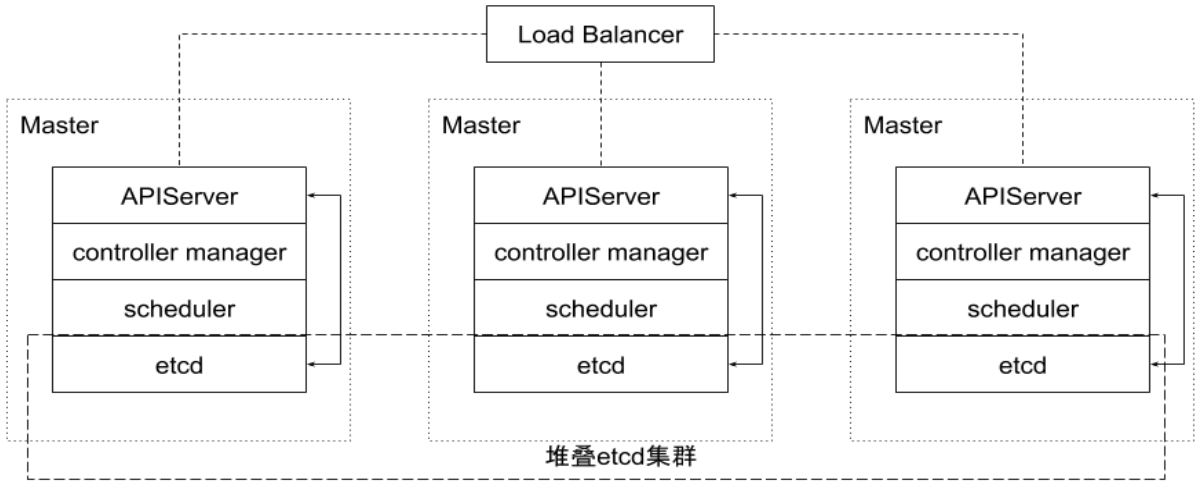
Kubernetes 的工作节点（Worker Node）

Kubelet	负责调度到对应节点的 Pod 的生命周期管理，执行任务并将 Pod 状态报告给主节点的渠道，通过容器运行时（拉取镜像、启动和停止容器等）来运行这些容器。它还会定期执行被请求的容器的健康探测程序。
Kube-proxy	它负责节点的网络，在主机上维护网络规则并执行连接转发。它还负责对正在服务的 pods 进行负载平衡。

etcd

etcd 是 CoreOS 基于 Raft 开发的分布式 key-value 存储，可用于服务发现、共享配置以及一致性保障（如数据库选主、分布式锁等）。

- 基本的 key-value 存储；
- 监听机制；
- key 的过期及续约机制，用于监控和服务发现；
- 原子 CAS 和 CAD，用于分布式锁和 leader 选举。



直接访问 etcd 的数据

极客时间 训练营

- 通过 etcd 进程查看启动参数
- 进入容器
    - ps -ef|grep etcd
    - sh: ps: command not found
- 怎么办？到主机 Namespace 查看 cert 信息
- 进入容器查询数据

export ETCDCTL_API=3

etcdctl --endpoints https://localhost:2379 --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key --cacert /etc/kubernetes/pki/etcd/ca.crt get --keys-only --prefix /

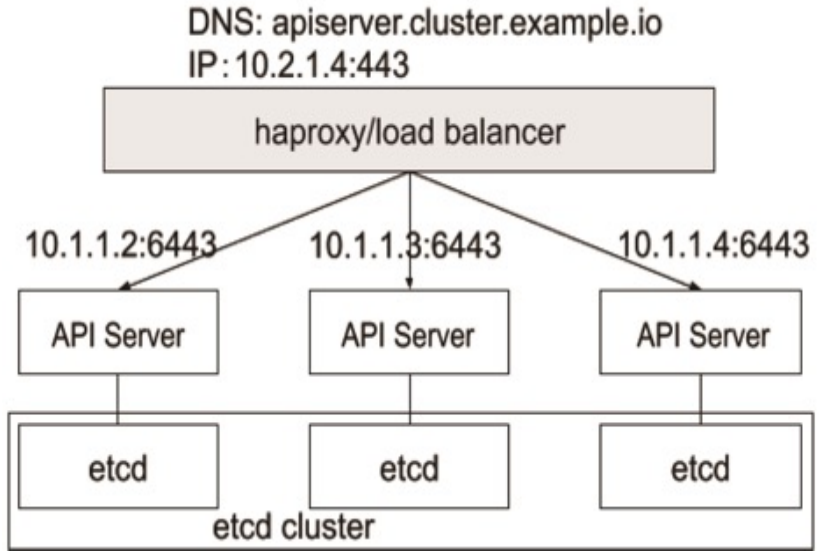
- 监听对象变化

etcdctl --endpoints https://localhost:2379 --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key --cacert /etc/kubernetes/pki/etcd/ca.crt watch --prefix /registry/services/specs/default/mynginx

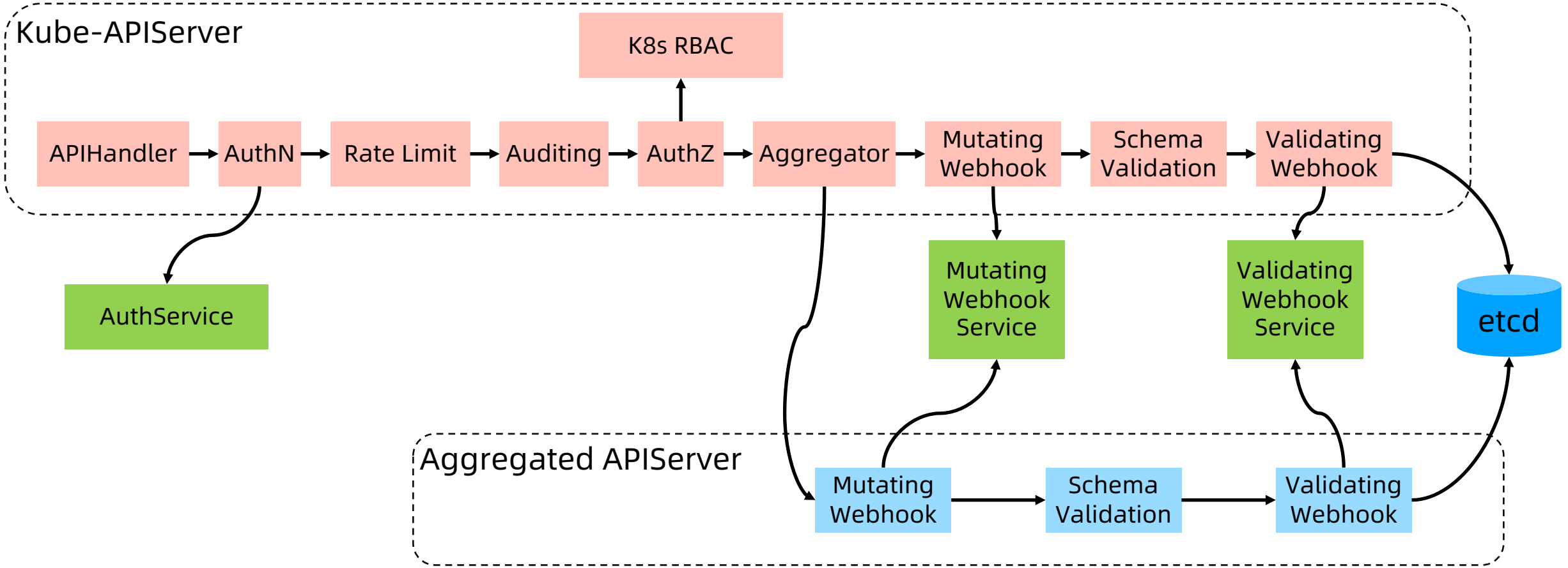
APIServer

Kube-APIServer 是 Kubernetes 最重要的核心组件之一，主要提供以下功能：

- 提供集群管理的 REST API 接口，包括：
  - 认证 Authentication；
  - 授权 Authorization；
  - 准入 Admission（Mutating & Valiating）。
- 提供其他模块之间的数据交互和通信的枢纽（其他模块通过 APIServer 查询或修改数据，只有 APIServer 才直接操作 etcd）。
- APIServer 提供 etcd 数据缓存以减少集群对 etcd 的访问。



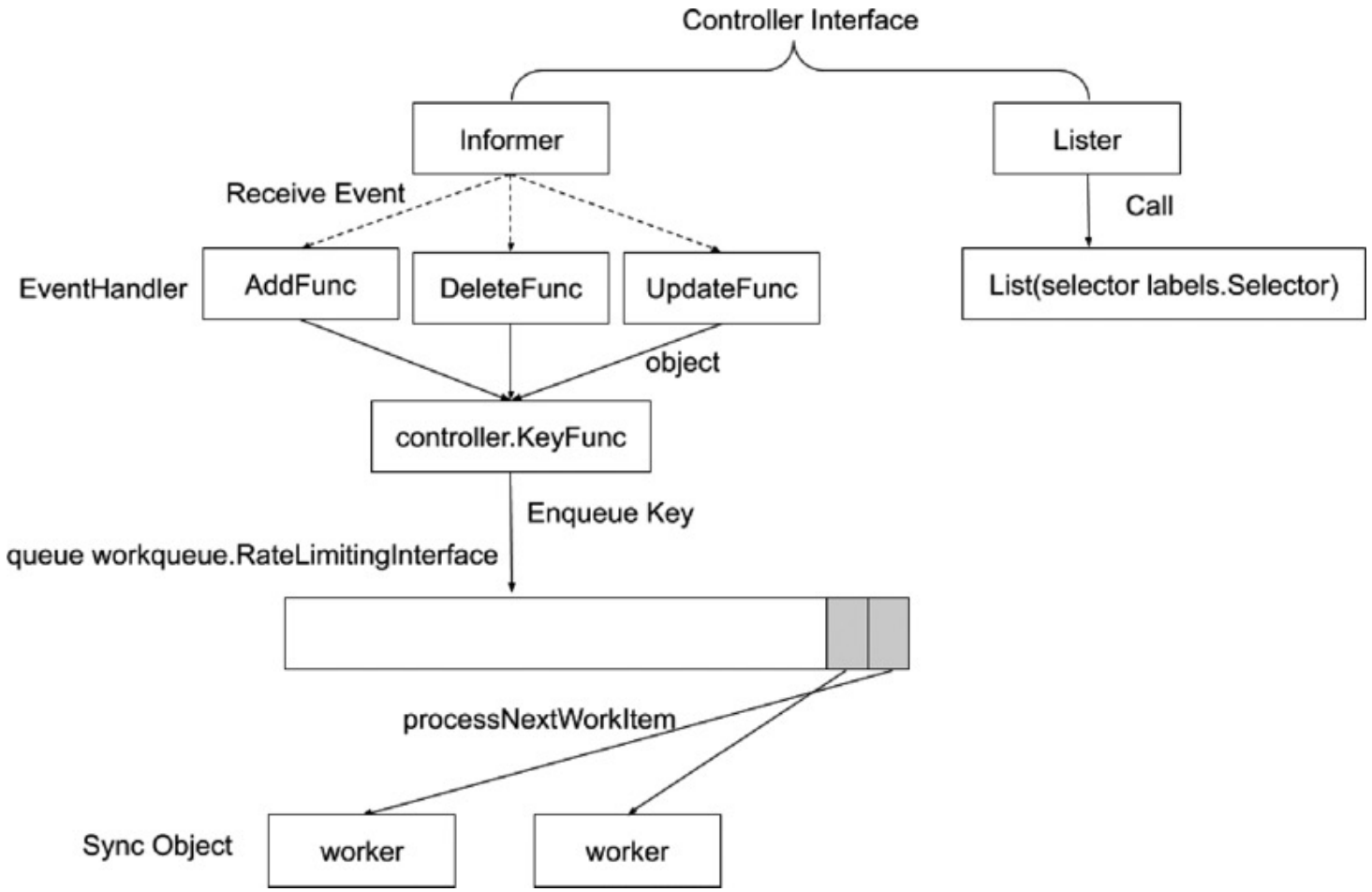
APIServer 展开



Controller Manager

- Controller Manager 是集群的大脑，是确保整个集群动起来的关键；
- 作用是确保 Kubernetes 遵循声明式系统规范，确保系统的真实状态（Actual State）与用户定义的期望状态（Desired State）一致；
- Controller Manager 是多个控制器的组合，每个 Controller 事实上都是一个 control loop，负责侦听其管控的对象，当对象发生变更时完成配置；
- Controller 配置失败通常会触发自动重试，整个集群会在控制器不断重试的机制下确保最终一致性（ Eventual Consistency）。

控制器的工作流程


Controller Interface
Informer
Lister
Receive Event
Call
EventHandler
AddFunc
DeleteFunc
UpdateFunc
List(selector labels.Selector)
object
controller.KeyFunc
Enqueue Key
queue workqueue.RateLimitingInterface
processNextWorkItem
Sync Object
worker
worker

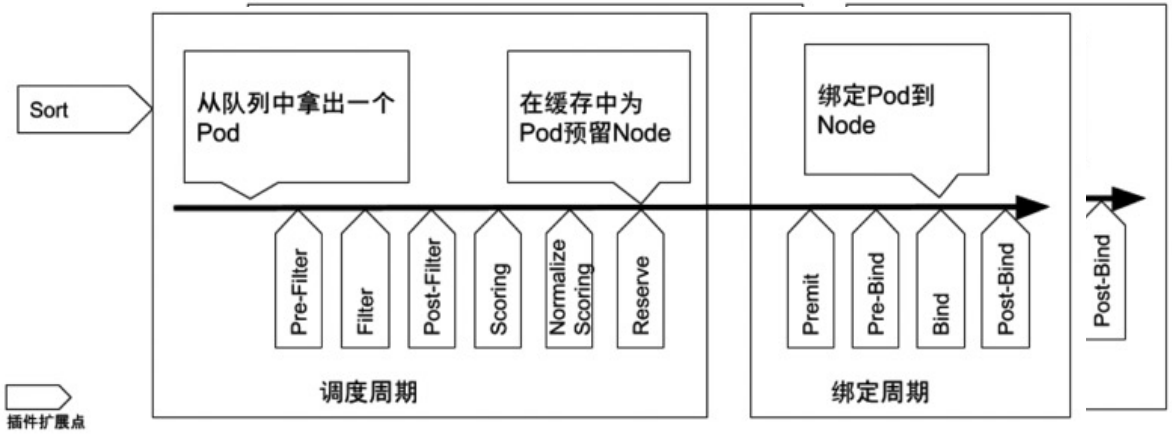
Scheduler

特殊的 Controller，工作原理与其他控制器无差别；

Scheduler 的特殊职责在于监控当前集群所有未调度的 Pod，并且获取当前集群所有节点的健康状况和资源使用情况，为待调度 Pod 选择最佳计算节点，完成调度。

调度阶段分为：

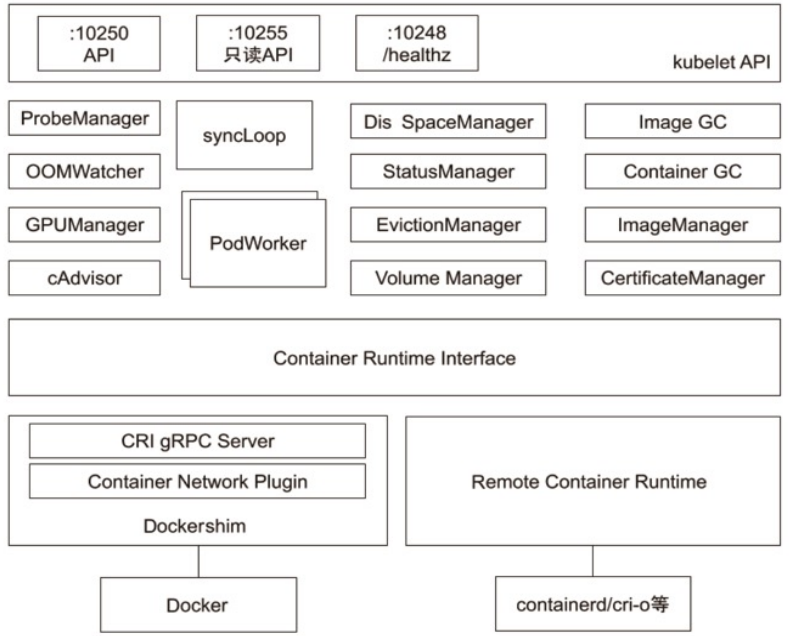
- Predict：过滤不能满足业务需求的节点，如资源不足、端口冲突等。
- Priority：按既定要素将满足调度需求的节点评分，选择最佳节点。
- Bind：将计算节点与 Pod 绑定，完成调度。



Kubelet

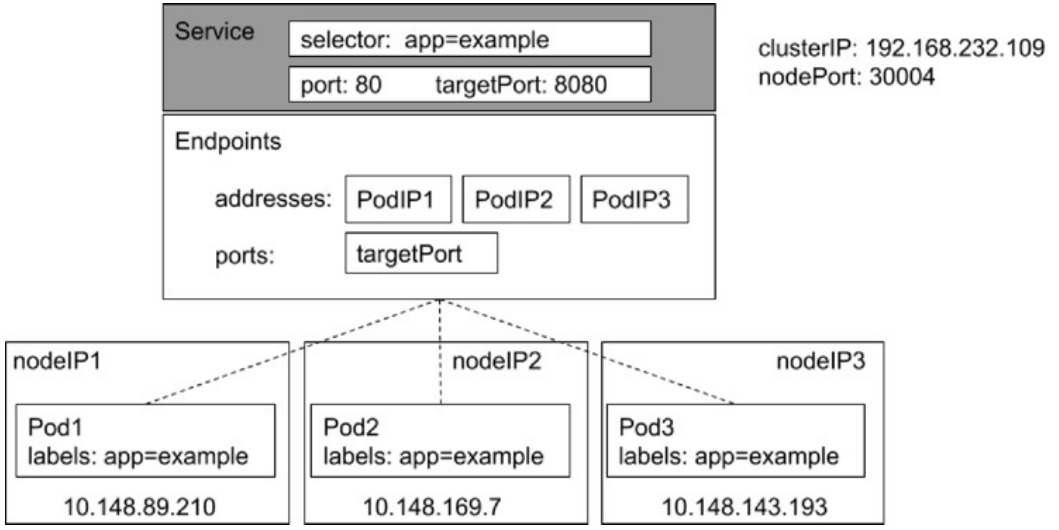
Kubernetes 的初始化系统（init system）

- 从不同源获取 Pod 清单，并按需求启停 Pod 的核心组件：
  - Pod 清单可从本地文件目录，给定的 HTTPServer 或 Kube-APIServer 等源头获取；
  - Kubelet 将运行时，网络和存储抽象成了 CRI，CNI，CSI。
- 负责汇报当前节点的资源信息和健康状态；
- 负责 Pod 的健康检查和状态汇报。



Kube-Proxy

- 监控集群中用户发布的服务，并完成负载均衡配置。
- 每个节点的 Kube-Proxy 都会配置相同的负载均衡策略，使得整个集群的服务发现建立在分布式负载均衡器之上，服务调用无需经过额外的网络跳转（Network Hop）。
- 负载均衡配置基于不同插件实现：
  - userspace。
  - 操作系统网络协议栈不同的 Hooks 点和插件：
    - iptables；
    - ipvs。

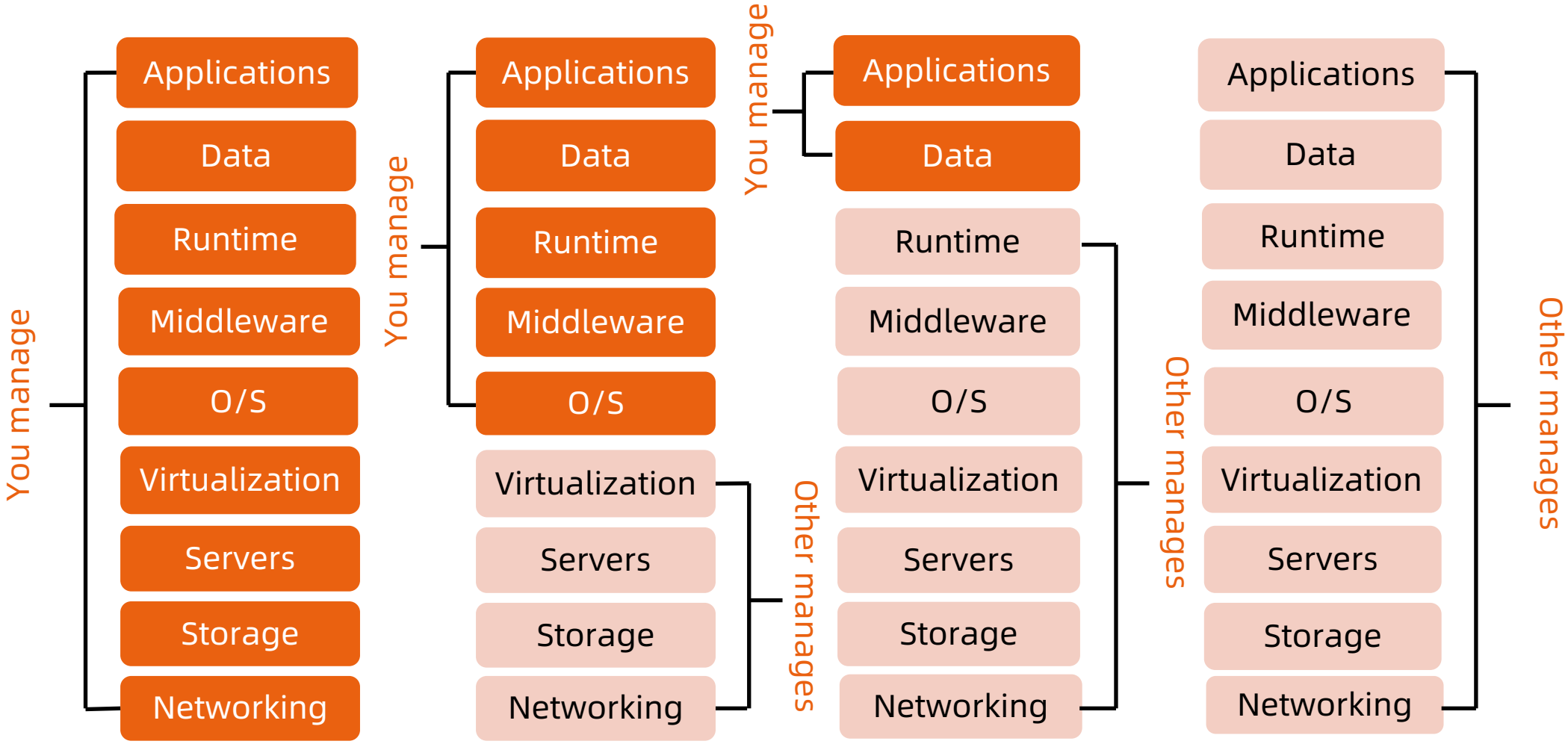


推荐的 Add-ons

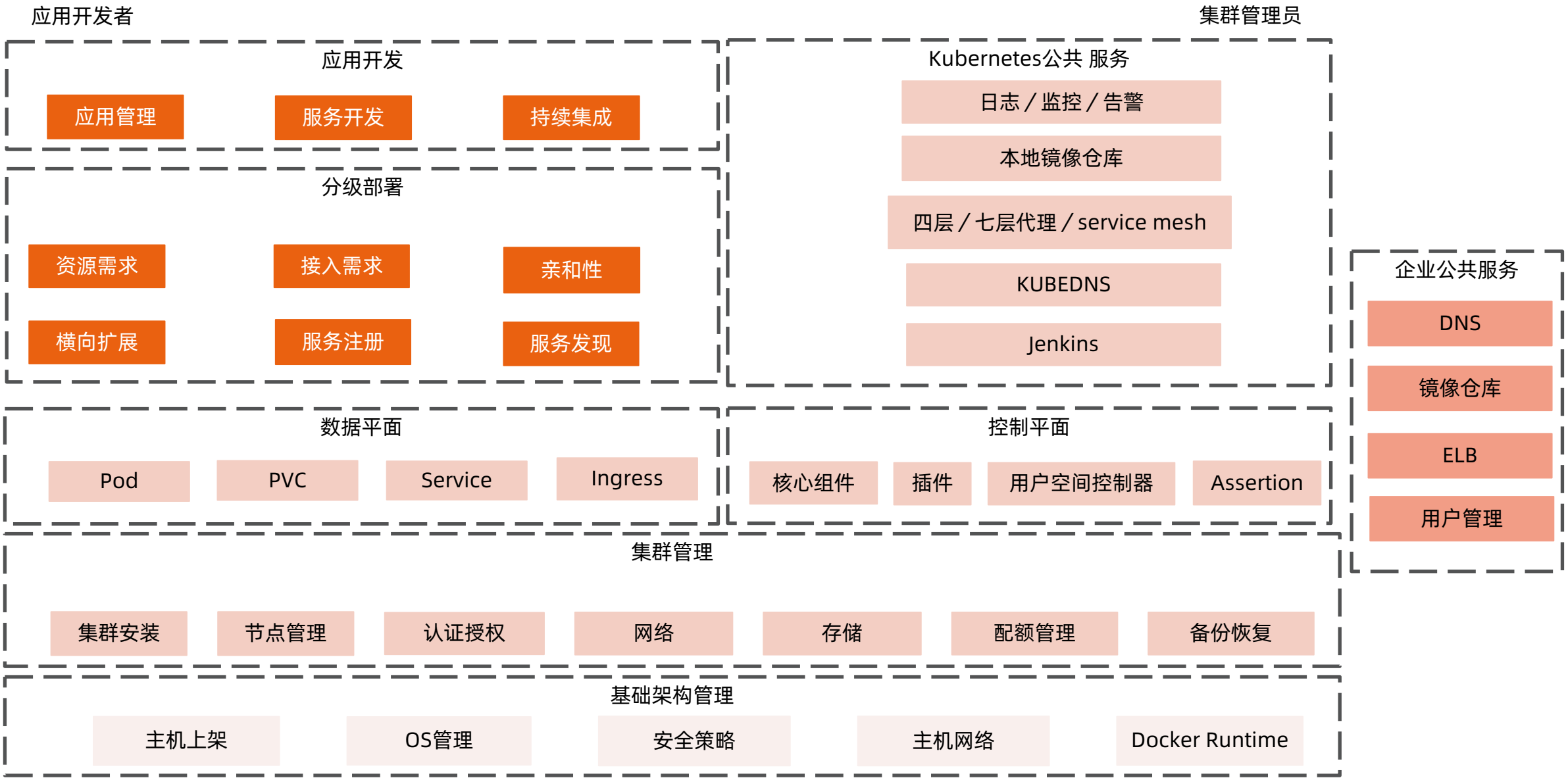
- kube-dns：负责为整个集群提供 DNS 服务；
- Ingress Controller：为服务提供外网入口；
- MetricsServer：提供资源监控；
- Dashboard：提供 GUI；
- Federation：提供跨可用区的集群；
- Fluentd-Elasticsearch：提供集群日志采集、存储与查询。

深入理解 Kubernetes

云计算的传统分类


You manage
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You manage
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Other manages
You manage
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Other manages
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Other manages

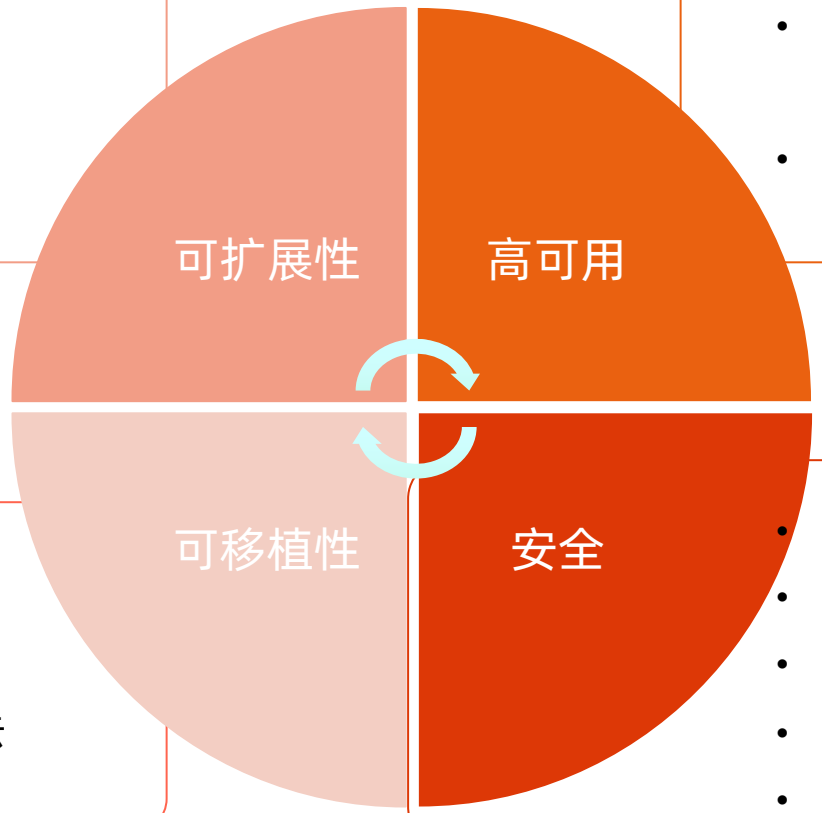
Kubernetes 生态系统



Kubernetes 设计理念

极客时间 训练营

- 基于微服务部署应用
- 横向扩容缩容
- 自动扩容缩容

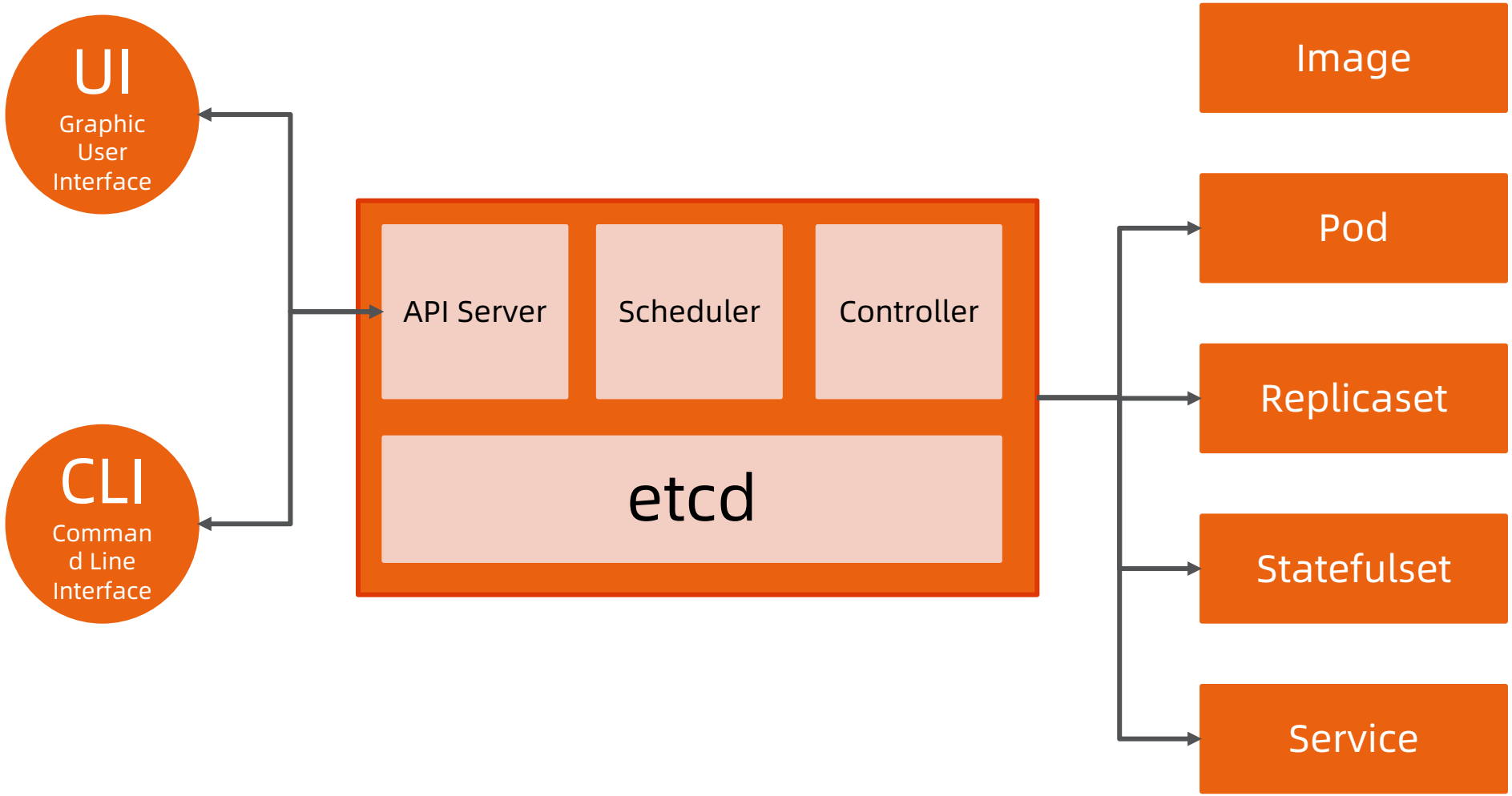


- 基于 replicaset，statefulset 的应用高可用
- Kubernetes 组件本身高可用

- 多种 host Os 选择
- 多种基础架构的选择
- 基于集群联邦建立混合云

- 基于 TLS 提供服务
- Serviceaccount 和 user
- 基于 Namespace 的隔离
- secret
- Taints，psp， networkpolicy

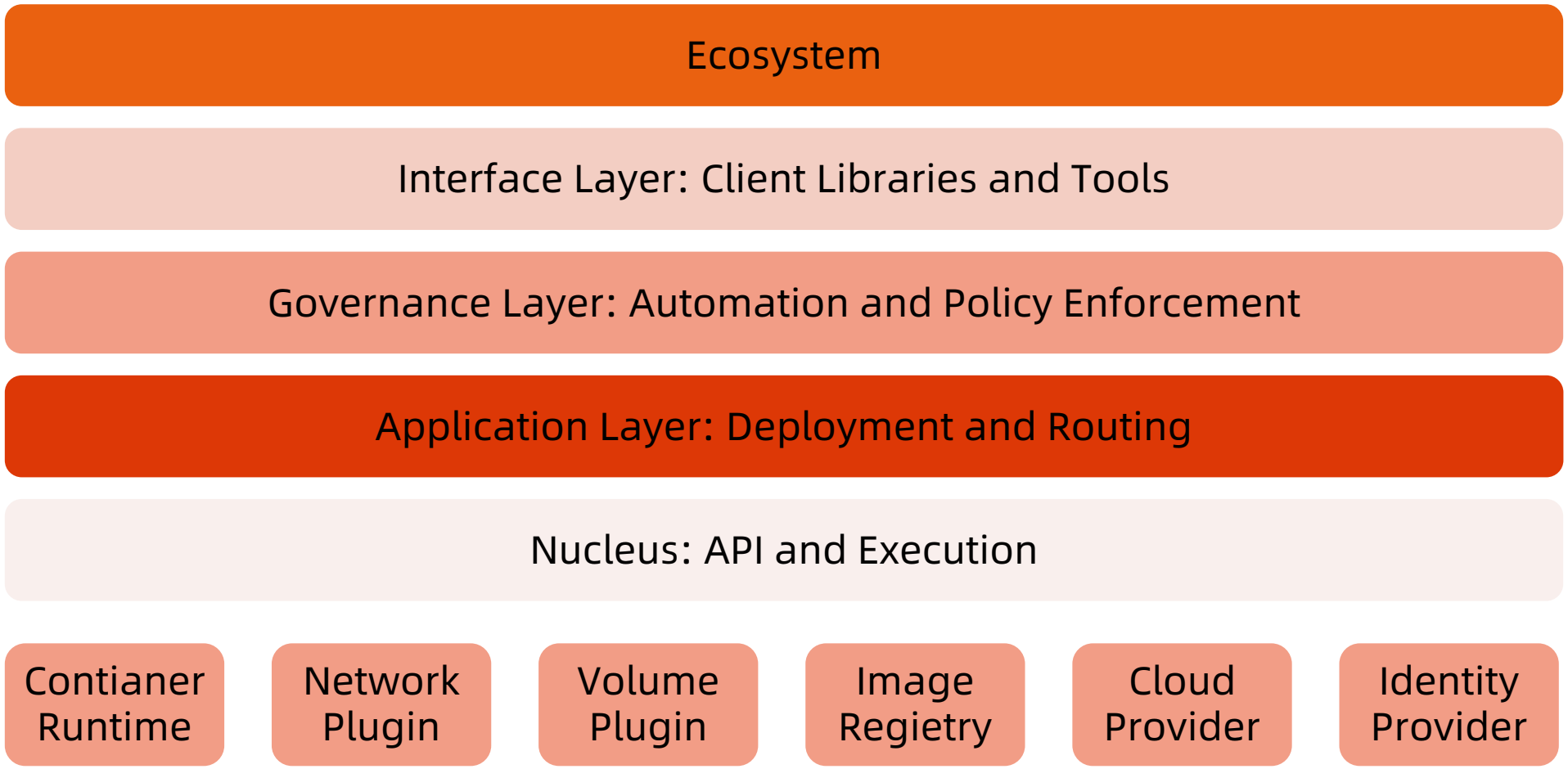
Kubernetes Master



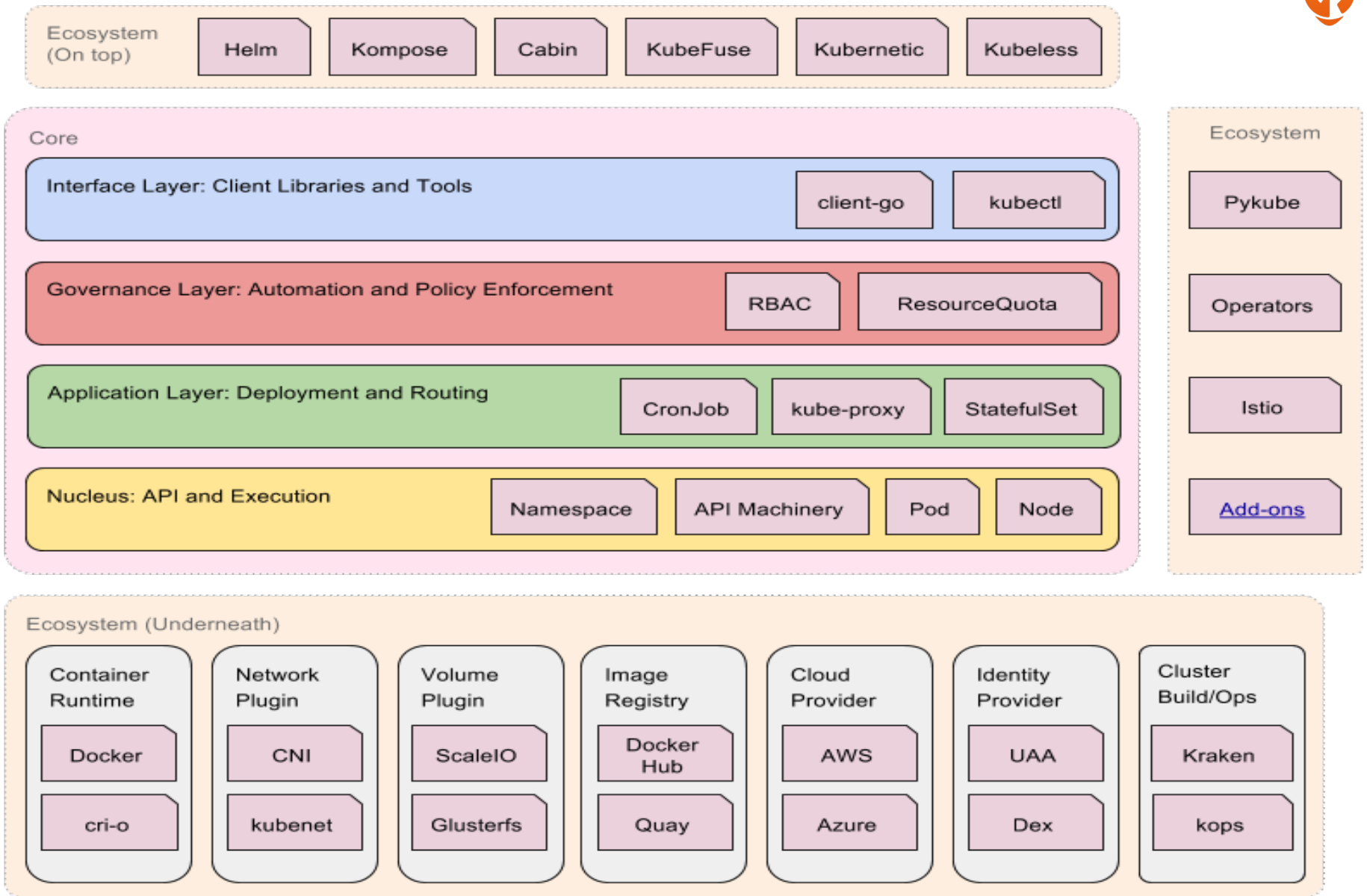
分层架构

- 核心层：Kubernetes 最核心的功能，对外提供 API 构建高层的应用，对内提供插件式应用执行环境。
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS 解析等）。
- 管理层：系统度量（如基础设施、容器和网络的度量）、自动化（如自动扩展、动态 Provision 等）、策略管理（RBAC、Quota、PSP、NetworkPolicy 等）。
- 接口层：Kubectl 命令行工具、客户端 SDK 以及集群联邦。
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴：
  - Kubernetes 外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS 应用、ChatOps 等；
  - Kubernetes 内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等。

分层架构


Ecosystem
Interface Layer: Client Libraries and Tools
Governance Layer: Automation and Policy Enforcement
Application Layer: Deployment and Routing
Nucleus: API and Execution
Contianer Runtime
Network Plugin
Volume Plugin
Image Regietry
Cloud Provider
Identity Provider

Kubernetes Core and Ecosystem By Example



API 设计原则

极客时间 训练营

- 所有 API 都应是声明式的。相对于命令式操作，声明式操作对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更易被用户使用，可以使系统向用户隐藏实现的细节，同时也保留了系统未来持续优化的可能性。此外，声明式的 API 还隐含了所有的 API 对象都是名词性质的，例如 Service、Volume 这些 API 都是名词，这些名词描述了用户所期望得到的一个目标对象。

- API 对象是彼此互补而且可组合的。这实际上鼓励 API 对象尽量实现面向对象设计时的要求，即“高内聚，松耦合”，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。

- 高层 API 以操作意图为基础设计。如何能够设计好 API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对 Kubernetes 的高层 API 设计，一定是以 Kubernetes 的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。

API 设计原则

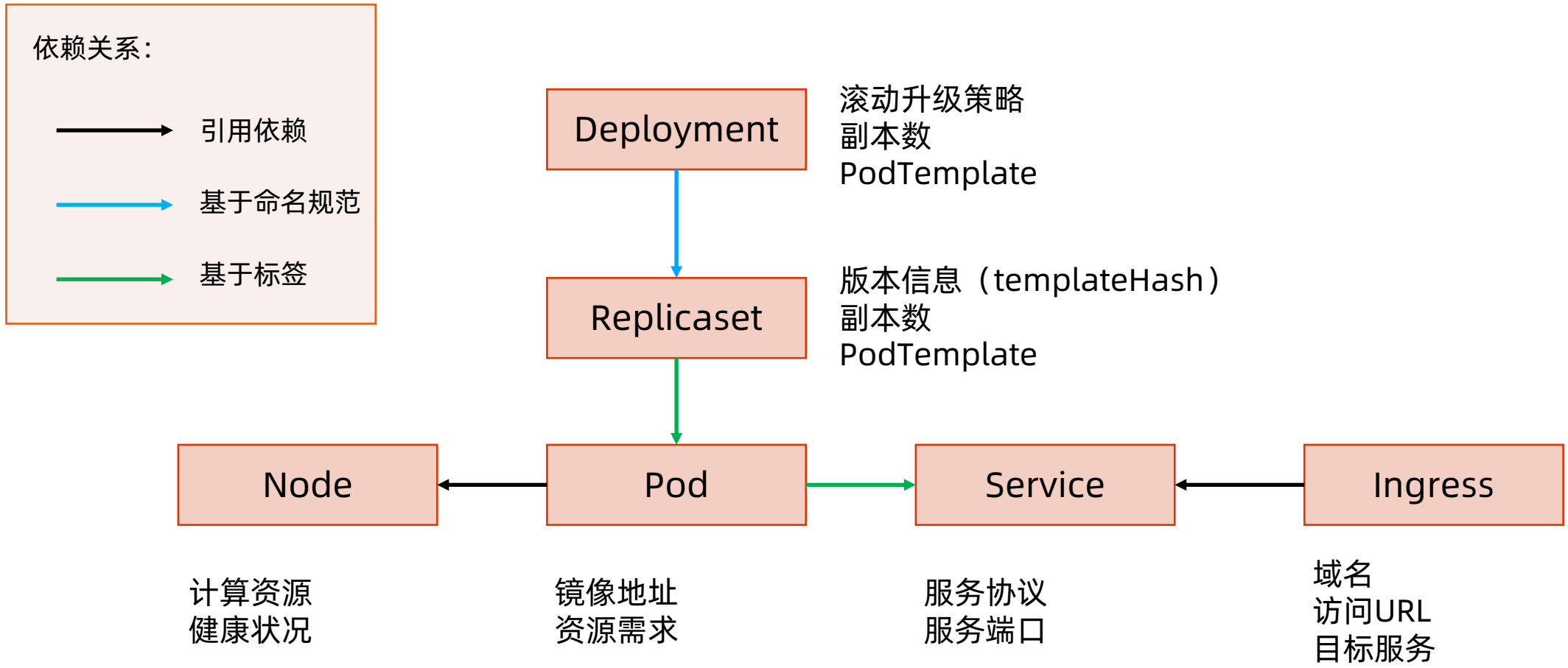
- 低层 API 根据高层 API 的控制需要设计。设计实现低层 API 的目的，是为了被高层 API 使用，考虑减少冗余、提高重用性的目的，低层 API 的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。
- 尽量避免简单封装，不要有在外部 API 无法显式知道的内部隐藏的机制。简单的封装，实际没有提供新的功能，反而增加了对所封装 API 的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如 StatefulSet 和 ReplicaSet，本来就是两种 Pod 集合，那么 Kubernetes 就用不同 API 对象来定义它们，而不会说只用同一个 ReplicaSet，内部通过特殊的算法再来区分这个 ReplicaSet 是有状态的还是无状态。
- API 操作复杂度与对象数量成正比。这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是 API 的操作复杂度不能超过 O(N)，N 是对象的数量，否则系统就不具备水平伸缩性了。

API 设计原则

- API 对象状态不能依赖于网络连接状态。由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证 API 对象状态能应对网络的不稳定，API 对象的状态就不能依赖于网络连接状态。

- 尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。

Kubernetes 如何通过对象的组合完成业务描述



架构设计原则

- 只有 APIServer 可以直接访问 etcd 存储，其他服务必须通过 Kubernetes API 来访问集群状态；
- 单节点故障不应该影响集群的状态；
- 在没有新请求的情况下，所有组件应该在故障恢复后继续执行上次最后收到的请求（比如网络分区或服务重启等）；
- 所有组件都应该在内存中保持所需要的状态，APIServer 将状态写入 etcd 存储，而其他组件则通过 APIServer 更新并监听所有的变化；
- 优先使用事件监听而不是轮询。

引导（Bootstrapping）原则

- Self-hosting 是目标
- 减少依赖，特别是稳态运行的依赖
- 通过分层的原则管理依赖
- 循环依赖问题的原则
    - 同时还接受其他方式的数据输入（比如本地文件等），这样在其他服务不可用时还可以手动配置引导服务
    - 状态应该是可恢复或可重新发现的
    - 支持简单的启动临时实例来创建稳态运行所需要的状态；使用分布式锁或文件锁等来协调不同状态的切换（通常称为 pivoting 技术）
    - 自动重启异常退出的服务，比如副本或者进程管理器等

课后练习 4.1

用 Kubeadm 安装 Kubernetes 集群。

核心技术概念和 API 对象

API 对象是 Kubernetes 集群中的管理操作单元。Kubernetes 集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的 API 对象，支持对该功能的管理操作。例如副本集 Replica Set 对应的 API 对象是 RS。

每个 API 对象都有 3 大类属性：元数据 metadata、规范 spec 和状态 status。元数据是用来标识 API 对象的，每个对象都至少有 3 个元数据：Namespace，name 和 uid；除此以外还有各种各样的标签 labels 用来标识和匹配不同的对象，例如用户可以用标签 env 来标识区分不同的服务部署环境，分别用 env=dev、env=testing、env=production 来标识开发、测试、生产的不同服务。规范描述了用户期望 Kubernetes 集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器 Replication Controller 设置期望的 Pod 副本数为 3；status 描述了系统实际当前达到的状态（Status），例如系统当前实际的 Pod 副本数为 2；那么复制控制器当前的程序逻辑就是自动启动新的 Pod，争取达到副本数为 3。

Kubernetes 中所有的配置都是通过 API 对象的 spec 去设置的，也就是用户通过配置系统的理想状态来改变系统，这是 Kubernetes 重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为 3 的操作运行多次也还是一个结果，而给副本数加 1 的操作就不是声明式的，运行多次结果就错了。

Kubernetes 基础

- Kubernetes 基本对象
- Pod
- Node
- Namespace
- Service
- Label
- Annotations

Pod

- Pod 是一组紧密关联的容器集合，它们共享 PID、IPC、Network 和 UTS Namespace，是 Kubernetes 调度的基本单位。
- 设计理念：支持多个容器在一个 Pod 中共享网络和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Node

- Node 是 Pod 真正运行的主机，可以物理机，也可以是虚拟机。
- 为了管理 Pod，每个 Node 节点上至少要运行 container runtime

（比如 Docker 或者 Rkt）、Kubelet 和 Kube-proxy 服务。

Namespace

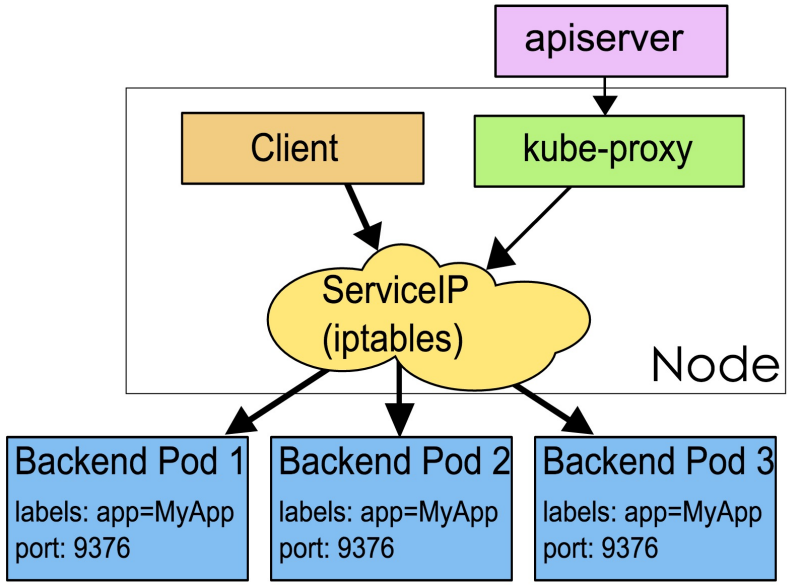
Namespace 是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。

常见的 pods, services, replication controllers 和 deployments 等都是属于某一个 Namespace 的（默认是 default），而 Node, persistentVolumes 等则不属于任何 Namespace。

Service

Service 是应用服务的抽象，通过 labels 为应用提供负载均衡和服务发现。匹配 labels的 Pod IP 和端口列表组成 endpoints，由 Kube-proxy 负责将服务 IP 负载均衡到这些 endpoints 上。

每个 Service 都会自动分配一个 cluster IP（仅在集群内部可访问的虚拟地址）和 DNS 名，其他容器可以通过该地址或 DNS 来访问服务，而不需要了解后端容器的运行。



Service Spec

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
  - port: 8078 # the port that this
service should serve on
    name: http
    # the container on each pod to
connect to, can be a name
    # (e.g. 'www') or a number (e.g.
80)
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```

Label

Label 是识别 Kubernetes 对象的标签，以 key/value 的方式附加到对象上（key 最长不能超过 63 字节，value 可以为空，也可以是不超过 253 字节的字符串）。Label 不提供唯一性，并且实际上经常是很多对象（如 Pods）都使用相同的 label 来标志具体的应用。

Label 定义好后其他对象可以使用 Label Selector 来选择一组相同 label 的对象（比如 ReplicaSet 和 Service 用label 来选择一组 Pod）。Label Selector 支持以下几种方式：

- 等式，如 app=nginx 和 env!=production
- 集合，如 env in (production, qa)
- 多个 label（它们之间是 AND 关系），如 app=nginx,env=test

Annotations

Annotations 是 key/value 形式附加于对象的注解。不同于 Labels 用于标志和选择对象，Annotations 则是用来记录一些附加信息，用来辅助应用部署、安全策略以及调度策略等。比如 deployment 使用 annotations 来记录 rolling update 的状态。

Try it

通过类似 Docker run 的命令在 Kubernetes 运行容器

kubectl run --image=nginx:alpine nginx-app --port=80

kubectl get deployment

kubectl describe deployment/rs/pod

kubectl expose deployment nginx-app --port=80 --target-port=80

kubectl describe svc

kubectl describe ep

Pod

Kubernetes 有很多技术概念，同时对应很多 API 对象，最重要的也是最基础的是微服务 Pod。Pod 是在 Kubernetes 集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod 的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，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod 对多容器的支持是 Kubernetes 最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个 Nginx 容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod 是 Kubernetes 集群中所有业务类型的基础，可以看作运行在 Kubernetes 集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前 Kubernetes 中的业务主要可以分为长期伺服型（long-running）、批处理型（Batch）、节点后台支撑型（Node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和StatefulSet，本文后面会一一介绍。

复制控制器（Replication Controller，RC）

RC 是 Kubernetes 集群中最早的保证 Pod 高可用的 API 对象。通过监控运行中的 Pod 来保证集群中运行指定数目的 Pod 副本。指定的数目可以是多个也可以是 1 个；少于指定数目，RC 就会启动运行新的 Pod 副本；多于指定数目，RC 就会杀死多余的 Pod 副本。即使在指定数目为 1 的情况下，通过 RC 运行 Pod 也比直接运行 Pod 更明智，因为 RC 也可以发挥它高可用的能力，保证永远有 1 个 Pod 在运行。RC 是 Kubernetes 较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的 Web 服务。

副本集（Replica Set，RS）

RS 是新一代 RC，提供同样的高可用能力，区别主要在于 RS 后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为 Deployment 的理想状态参数使用。

部署（Deployment）

部署表示用户对 Kubernetes 集群的一次更新操作。部署是一个比 RS 应用模式更广的 API 对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的 RS，然后逐渐将新 RS 中副本数增加到理想状态，将旧 RS 中的副本数减小到 0 的复合操作；这样一个复合操作用一个 RS 是不太好描述的，所以用一个更通用的 Deployment 来描述。以 Kubernetes 的发展方向，未来对所有长期伺服型的的业务的管理，都会通过 Deployment 来管理。

服务（Service）

RC、RS 和 Deployment 只是保证了支撑服务的微服务 Pod 的数量，但是没有解决如何访问这些服务的问题。一个 Pod 只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的 IP 启动一个新的 Pod，因此不能以确定的 IP 和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在 Kubernetes 集群中，客户端需要访问的服务就是 Service 对象。每个 Service 会对应一个集群内部有效的虚拟 IP，集群内部通过虚拟 IP 访问一个服务。在 Kubernetes 集群中微服务的负载均衡是由 Kube-proxy 实现的。Kube-proxy 是Kubernetes 集群内部的负载均衡器。它是一个分布式代理服务器，在 Kubernetes 的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的 Kube-proxy 就越多，高可用节点也随之增多。与之相比，我们平时在服务器端使用反向代理作负载均衡，还要进一步解决反向代理的高可用问题。

任务（Job）

Job 是 Kubernetes 用来控制批处理型任务的 API 对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job 管理的 Pod 根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的 spec.completions 策略而不同：单 Pod 型任务有一个 Pod 成功就标志完成；定数成功型任务保证有 N 个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

后台支撑服务集（DaemonSet）

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的 Pod，有些节点上又没有这类 Pod 运行；而后台支撑型服务的核心关注点在 Kubernetes 集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类 Pod 运行。节点可能是所有集群节点也可能是通过 nodeSelector 选定的一些特定节点。典型的后台支撑型服务包括存储、日志和监控等在每个节点上支撑 Kubernetes 集群运行的服务。

有状态服务集（StatefulSet）

Kubernetes 在 1.3 版本里发布了 Alpha 版的 PetSet 以支持有状态服务，并从 1.5 版本开始重命名为 StatefulSet。在云原生应用的体系里，有下面两组近义词；第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC 和 RS 主要是控制提供无状态服务的，其所控制的 Pod 的名字是随机设置的，一个 Pod 出故障了就被丢弃掉，在另一个地方重启一个新的 Pod，名字变了、名字和启动在哪儿都不重要，重要的只是 Pod 总数；而 StatefulSet 是用来控制有状态服务，StatefulSet 中的每个 Pod 的名字都是事先确定的，不能更改。StatefulSet 中 Pod 的名字的作用，并不是《千与千寻》的人性原因，而是关联与该 Pod 对应的状态。

对于 RC 和 RS 中的 Pod，一般不挂载存储或者挂载共享存储，保存的是所有 Pod 共享的状态，Pod 像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于 StatefulSet 中的 Pod，每个 Pod 挂载自己独立的存储，如果一个 Pod 出现故障，从其他节点启动一个同样名字的 Pod，要挂载上原来 Pod 的存储继续以它的状态提供服务。

适合于 StatefulSet 的业务包括数据库服务 MySQL 和 PostgreSQL，集群化管理服务 Zookeeper、etcd 等有状态服务。StatefulSet 的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用 StatefulSet，Pod 仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，StatefulSet 做的只是将确定的 Pod 与确定的存储关联起来保证状态的连续性。StatefulSet 还只在 Alpha 阶段，后面的设计如何演变，我们还要继续观察。

集群联邦（Federation）

Kubernetes 在 1.3 版本里发布了 beta 版的 Federation 功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。 Kubernetes 的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足 Kubernetes 的调度和计算存储连接要求。而联合集群服务就是为提供跨 Region 跨服务商 Kubernetes 集群服务而设计的。

每个 Kubernetes Federation 有自己的分布式存储、API Server 和 Controller Manager。用户可以通过 Federation 的 API Server 注册该 Federation 的成员 Kubernetes Cluster。当用户通过 Federation 的 API Server 创建、更改 API 对象时，Federation API Server 会在自己所有注册的子 Kubernetes Cluster 都创建一份对应的 API 对象。在提供业务请求服务时， Kubernetes Federation 会先在自己的各个子 Cluster 之间做负载均衡，而对于发送到某个具体 Kubernetes Cluster 的业务请求，会依照这个 Kubernetes Cluster 独立提供服务时一样的调度模式去做 Kubernetes Cluster 内部的负载均衡。而 Cluster 之间的负载均衡是通过域名服务的负载均衡来实现的。

所有的设计都尽量不影响 Kubernetes Cluster 现有的工作机制，这样对于每个子 Kubernetes 集群来说，并不需要更外层的有一个 Kubernetes Federation，也就是意味着所有现有的 Kubernetes 代码和机制不需要因为 Federation 功能有任何变化。

存储卷（Volume）

Kubernetes 集群中的存储卷跟 Docker 的存储卷有些类似，只不过 Docker 的存储卷作用范围为一个容器，而 Kubernetes 的存储卷的生命周期和作用范围是一个 Pod。每个 Pod 中声明的存储卷由 Pod 中的所有容器共享。 Kubernetes 支持非常多的存储卷类型，支持多种公有云平台的存储，包括 AWS，Google 和 Azure 云；支持多种分布式存储包括 GlusterFS 和 Ceph；也支持较容易使用的主机本地目录 hostPath 和 NFS。

Kubernetes 还支持使用 Persistent Volume Claim 即 PVC 这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如 AWS，Google 或 GlusterFS 和 Ceph），而将有关存储实际技术的配置交给存储管理员通过 Persistent Volume 来配置。

存储 PV 和 PVC

PV 和 PVC 使得 Kubernetes 集群具备了存储的逻辑抽象能力，使得在配置 Pod 的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给 PV 的配置者，即集群的管理者。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 Kubernetes 集群管理员配置；而 PVC 和 Pod 是资源的使用者，根据业务服务的需求变化而变化，由 Kubernetes 集群的使用者即服务的管理员来配置。

节点（Node）

Kubernetes 集群中的计算能力由 Node 提供，最初 Node 称为服务节点 Minion，后来改名为 Node。 Kubernetes 集群中的 Node 也就等同于 Mesos 集群中的 Slave 节点，是所有 Pod 运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行 Kubelet 管理节点上运行的容器。

密钥对象（Secret）

Secret 是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用 Secret 的好处是可以避免把敏感信息明文写在配置文件里。Kubernetes 集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问 AWS 存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个 Secret 对象，而在配置文件中通过 Secret 对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

用户（User Account）& 服务帐户（Service Account）

顾名思义，用户帐户为人提供账户标识，而服务账户为计算机进程和 Kubernetes 集群中运行的 Pod 提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的 Namespace 无关，所以用户账户是跨 Namespace 的；而服务帐户对应的是一个运行中程序的身份，与特定 Namespace 是相关的。

名字空间（Namespace）

名字空间为 Kubernetes 集群提供虚拟的隔离作用， Kubernetes 集群初始有两个名字空间，分别是默认名字空间 default 和系统名字空间 Kube-system，除此以外，管理员可以创建新的名字空间满足需要。

RBAC 访问授权

Kubernetes 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， Kubernetes 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，具体的用户在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。

资源限制

Kubernetes 通过 Cgroups 提供容器资源管理的功能，可以限制每个容器的 CPU 和内存使用，比如对于刚才创建的 deployment，可以通过下面的命令限制 nginx 容器最多只用 50% 的 CPU 和 128MB 的内存：

```
$ kubectl set resources deployment nginx-app -c=nginx --limits=cpu=500m,memory=128Mi

deployment "nginx" resource requirements updated
```

等同于在每个 Pod 中设置 resources limits


```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
```

健康检查

Kubernetes 作为一个面向应用的集群管理工具，需要确保容器在部署后确实处在正常的运行状态。Kubernetes 提供了两种探针（Probe，支持 exec、TCP Socket 和 http 方式）来探测容器的状态：

LivenessProbe：探测应用是否处于健康状态，如果不健康则删除并重新创建容器。

ReadinessProbe：探测应用是否启动完成并且处于正常服务状态，如果不正常则不会接收来自 Kubernetes Service 的流量。

对于已经部署的 deployment，可以通过 kubectl edit deployment/nginx-app 来更新 manifest，增加健康检查部分。

健康检查 spec

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx-default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

```
spec:
    containers:
    - image: nginx
      imagePullPolicy: Always
      name: http
      resources: {}
      terminationMessagePath:
/dev/termination-log
      terminationMessagePolicy: File
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
```

```
livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 15
        timeoutSeconds: 1
      readinessProbe:
        httpGet:
          path: /ping
          port: 80
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

课后练习 4.2

- 启动一个 Envoy Deployment
- 要求 Envoy 的启动配置从外部的配置文件 Mount 进 Pod
- 进入 Pod 查看 Envoy 进程和配置
- 更改配置的监听端口并测试访问入口的变化
- 通过非级联删除的方法逐个删除对象

THANKS