# 容器云稳定性，高可用和监控架构

*作者: 光大科技 - 陈盼*

*团队介绍: 我们是光大科技公司云计算团队容器云项目组,致力于容器技术在金融行业的落地与实践,以容器云助力金融行业科技转型。我们的团队拥有经验丰富的容器与云原生领域研发工程师和专家，将不定期分享容器和云原生行业的原创技术文章和实践经验，共同探索与见证云原生时代金融领域的前沿技术和发展趋势。*

## 概要

容器云是云计算领域的大势所趋，光大科技云计算团队持续探索容器云前沿技术，推动容器云平台在集团及其子公司的落地，同时积极参与技术交流与普及相关的社区活动。近日，团队三名工程师作为专家委员会成员为twt社区和红帽联手主办的"2020 trendpro容器云职业技能大赛"编写了容器云相关培训文章。由于集团不少项目正在进行容器化转型或需要与容器云对接，部分同事对容器技术也颇有兴趣，因此我们对原文进行了精简提炼，分享给大家，希望能对大家有所帮助。

文章并未涉及容器技术的基础知识，主要是对基于Kubernetes的容器云的相关介绍，分为容器云稳定性，高可用和监控架构三部分，部分内容可能需要一定的容器技术基础，但不影响读者对容器云建立一个整体的认识; 诸如业务稳定性保障等内容可能为容器化项目的部署与容器云平台的接入提供参考。篇幅所限，高可用和监控架构部分的内容仅提供原文链接，感兴趣可移步阅读。

## 容器云稳定性

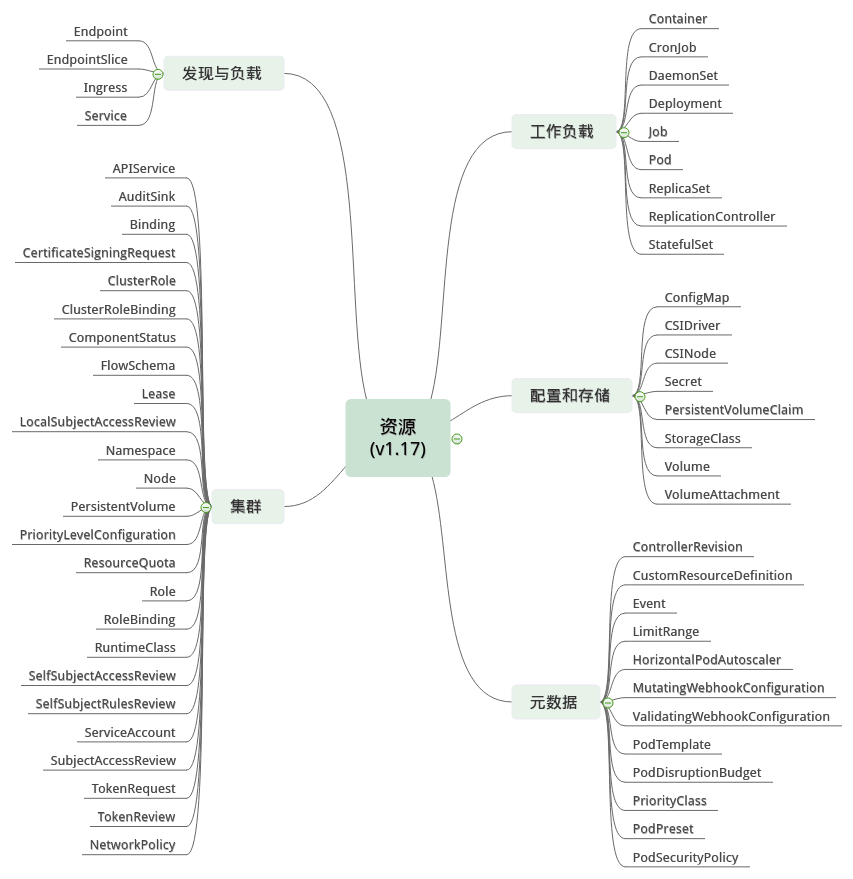
容器云作为承载大规模业务应用的平台级基础设施，其稳定性的重要性不言而喻。稳定性的概念比较泛化，本部分将从API、平台和业务三个维度阐述Kubernetes的稳定性设计。

### API稳定性设计

Kubernetes API是Kubernetes系统的重要组成部分，组件之间的所有操作和通信以及外部对Kubernetes的调用都是由API Server处理的REST API调用。API的设计对于产品内部通信和外部协作的兼容性，扩展性和稳定性有着举足轻重的影响。

#### API资源

Kubernetes中所有内容都被抽象为**资源**。所有资源都可以使用清单文件（manifest file）进行描述，使用Etcd数据库以JSON或Protobuf的形式进行存储并由API Server统一管理。

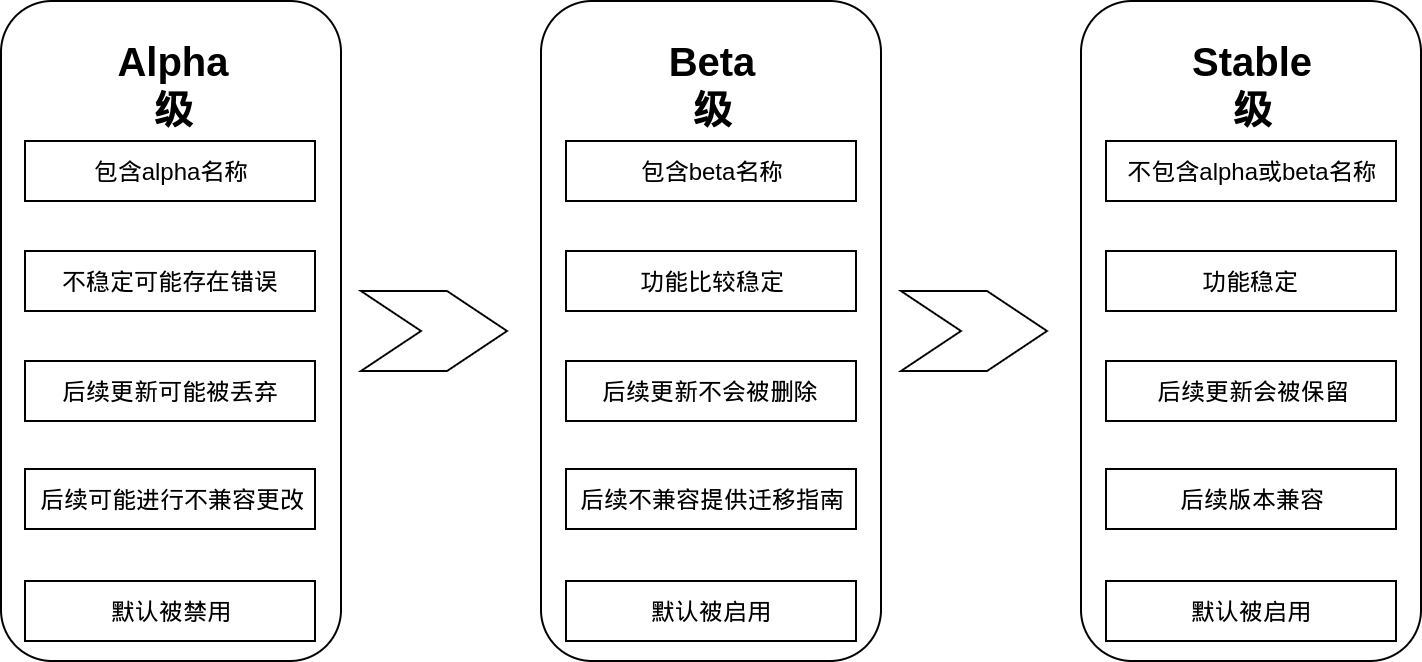


原创图片

Kubernetes基本上是以**API Group（API群组）**的方式组织各种API的。API群组是一组相关的API对象的集合，使用群组概念能够更方便的管理和扩展API。

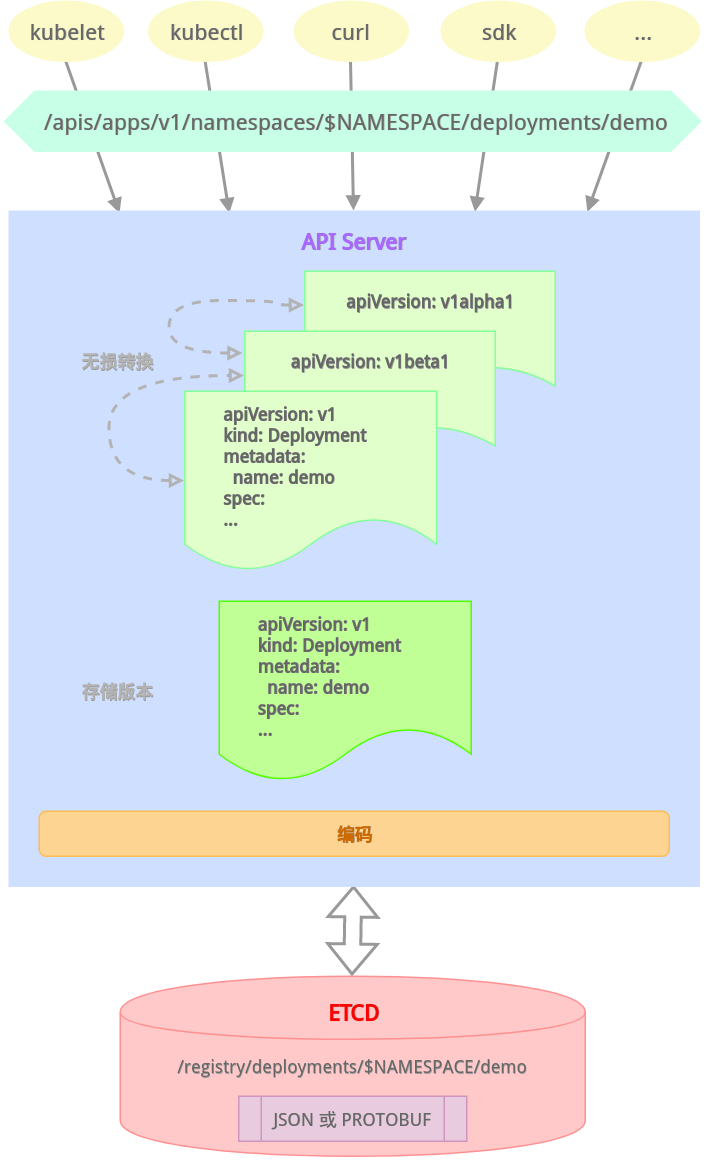
#### API版本

为了在兼容旧版本的同时不断升级新的API，Kubernetes支持多种API版本，不同的API版本代表其处于不同的稳定性阶段，低稳定性的API版本在后续的产品升级中可能成为高稳定性的版本。API通过三级渐进式版本共存与演化策略，在不断吸纳新的功能特性并给予其足够的孵化空间的同时，保证了整体API的可用性和稳定性。



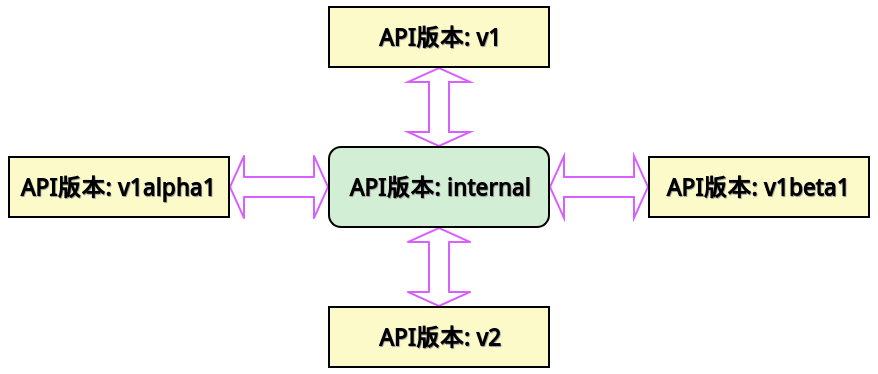
原创图片

用户向Kubernetes发起资源构建请求时只提供了一个资源清单文件（如deployment.yaml），但事实上Kubernetes基于可用性和稳定性的考虑，却能够支持同时使用不同稳定性的API版本访问同一资源，返回不同版本的资源数据。这一灵活的特性有赖于API Server的资源数据**无损耗转换**机制。



原创图片

Etcd数据库中只存储了资源的一个指定版本，同时API Server中维护着一个**internal**版本，需要作版本转换时，任意原版本都先转换为internal版本，再由internal版本转换到指定的目的版本，如此只要每个版本都可转换为internal版本，则可以支持任意版本之间的转换。而保证版本转换过程中不出现数据丢失（即无损转换）则是依靠**annotations（注解）**实现。



原创图片

#### API自定义

Kubernetes因其平台级基础设施的特殊性，与服务器、网络、存储、虚拟化、身份认证等等绝大多数计算机软硬件技术领域存在广泛交集，这需要大量的适配与对接，此外作为底层容器编排引擎，也需要满足高度的可扩展性以面对大量的功能特性扩展需求。

Kubernetes提供了两种API扩展机制保证核心API足够精简的同时满足庞杂的适配对接和特性扩展需求:

1. 自定义资源类型（CRD）：即CustomResourceDefinitions。允许用户通过资源清单的方式定义任意全新的资源对象类型，并由API Server管理自定义资源的整个生命周期，用户还可以通过定义相应的控制器对自定义资源及其他相关资源进行监视，协调和管理。通常将自定义资源和自定义控制器配合工作的方式统称为CRD方式。
2. API Server聚合（AA）：即API Server Aggregattion。其前身是用户API Server（UAS），UAS允许用户设计一套自定义的API Server与Kubernetes主API Server并行生效，可以在不影响原API Server的前提下实现更加复杂和定制化的逻辑和功能，但这种方式对代码开发要求比较高。自定义API Server可以选择与主API Server进行聚合也可以独立存在，但独立存在的方式无法与Kubernetes很好的集成，因此自定义API Server普遍采用API Server聚合的方式。Kubernetes API Server中内置API聚合层用于将自定义API动态无缝注册到主API Server中。

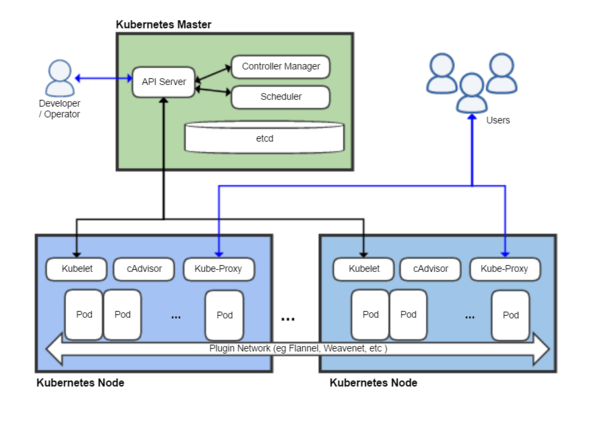
### 平台稳定性优化

不同的容器云平台解决方案在整体架构和实现上是存在差异的，这里仅讨论Kubernetes集群本身的稳定性优化（平台高可用也是保证稳定性的重要手段，将在单独的章节介绍）。

Kubernetes集群作为一种基础设施，其优化涉及到方方面面的内容，如api-server、etcd、kubelet等集群组件的优化，集群节点，网络和存储的优化等等。当集群到达一定规模的时候，一些细微优化往往能够让集群性能得到显著提升，从而降低各方面性能瓶颈影响集群稳定性的可能。

#### 组件优化

Kubernetes集群是通过api-server、etcd、kube-controller-manager、kubelet等一系列系统组件组织起来的，它们是集群的核心。这些组件提供了丰富调优参数和机制以满足不同的集群规模和系统环境。此外还有一些重要的如coredns、autoscaler等附加组件，对这些组件的优化是Kubernetes集群优化的关键。



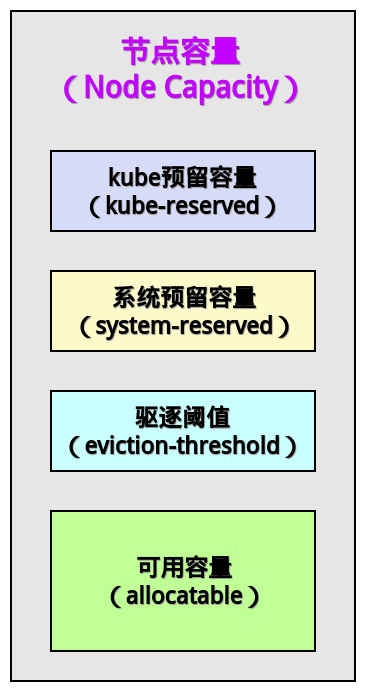
图片源自网络: https://xz.aliyun.com/t/4276

1. 对etcd的优化主要从**网络IO延迟**和**磁盘IO延迟**两方面着手，包括但不限于心跳间隔和选举超时参数自定义、快照阈值调整、进程优先级设置、独立节点与更换SSD磁盘等内容。
2. 对kube-apiserver的优化主要是高可用架构、并发请求控制、数据分库存储、缓存参数自定义等，核心思想是缓解api server压力。
3. 对kube-controller-manager和kube-scheduler的优化主要是高可用架构、自定义到api server的请求间隔、可选的控制器列表和自定义调度器等。
4. 对kubelet的优化可以通过自定义诸如节点最大Pod数量、并行拉取镜像开关、镜像拉取超时时间、与api server通信间隔等参数实现。
5. 对kube-proxy的优化主要是更改运行模式，当默认的iptables模式无法满足需求时，可考虑更换为ipvs模式。
6. 对coredns的优化主要是高可用架构以及根据需要设置DNS策略。

#### 节点优化

节点（Node）是负载工作容器的机器，可以是物理机也可以是虚拟机。针对节点的优化主要分为以下三个方面:

1. 基础环境优化。包括操作系统选择，节点容量规划，内核参数配置等内容。
2. master节点优化。master节点运行着集群的控制平面组件，需要根据集群规模合理规划Master节点的CPU和内存配置。此外应通过污点避免业务容器运行在master节点，如etcd负载较大，可考虑将etcd组件运行在单独的服务器上。
3. worker节点优化。主要包括硬性设定节点负载上限，关注软性指标数据，避免关键组件（如docker，kubelet，kube-proxy）等因内存不足而被OOM杀死，配置节点容量预留参数等。



原创图片

此外，当节点为虚拟机时，由于不同的虚拟机可能运行在同一个物理机上，而Kubernetes的默认调度机制只能保证同一工作负载的副本不被调度到相同节点上，而不能保证不被调度到相同物理机上，因此需要配合节点标签和自定义调度器实现节点亲和穿透。

#### 网络优化

网络是Kubernetes中的核心部件，Kubernetes对集群网络进行了重新抽象，以实现整个集群网络的扁平化。网络规划和性能表现关系到集群内系统组件，应用容器之间能否正常高效地通信，也关系到集群对外服务的可用性和服务质量。从宏观上而言，Kubernetes的网络优化主要包括:

1. 网络插件。网络插件负责管理集群内部的网络，其优化包括网络CNI插件的选择和插件本身的参数调整。各种CNI插件在网络性能（underlay/overlay），网络策略（network policy）支持度，集群规模上都有着不小的差别，各插件参数也不相同，因此需要根据实际需要选型。
2. 入口方案。入口方案关系到外部如何对集群内进行访问，同样存在诸如NodePort、LoadBalancer、Ingress、Service Mesh等众多解决方案，各有利弊，优化参数也各不相同。多种入口方案可以并存。

此外，节点操作系统内核针对网络也有大量的优化参数，必要时也需要进行单独的优化调整。

#### 存储优化

Kubernetes内置多种存储插件（In-tree插件），同时允许对接独立存储插件（Out-of-tree插件），提供了非常丰富的存储支持，并且通过**持久卷（PersistentVolume）**子系统对各种存储资源进行抽象，为集群提供了统一的存储API。

存储主要分为**块存储**，**文件存储**和**对象存储**，各有适用场景，不同的集群组件和业务应用应当选择适当类型的存储。

|  |  |  |
| --- | --- | --- |
| **存储类型** | **描述** | **例子** |
| Block（块存储） | 在操作系统中作为块设备  也称为存储区域网络（SAN）  适用于需要完全控制存储并绕过文件系统的低层直接操作文件的应用程序  不可共享，每次只有一个客户端可以挂载这种类型的端点 | AWS EBS  Ceph RBD |
| File（文件存储） | 在操作系统中作为要挂在的文件系统导出  也称为网络附加存储（NAS）  不同协议实现和厂商并行性，延迟，文件锁定机制等可能差异较大  可以有多个客户端共享访问 | RHEL NFS  NetApp NFS |
| Object（对象存储） | 通过REST API端点访问  应用程序必须在应用或容器中构建其驱动程序  适用于非结构化数据存储需求  可以有多个客户端共享访问 | AWS S3 |

Kubernetes通过In-tree存储插件原生支持主流块存储和文件存储的挂载，对象存储要求容器应用通过API访问或者开发Out-of-tree插件（Flexvolume或CSI方式）进行对接。

主流块存储支持持久卷动态置备，但文件存储需要依赖独立的驱动插件实现，如Redhat的NFS Provisioner和NetApp的Trident。

针对Kubernetes平台，还存在专门的云原生存储编排系统，用于支持多种不同类型存储的自动化管理。能够将分布式存储软件转化为自我管理、自我调节和自我修复的存储服务，可以视情况使用。

### 业务稳定性保障

在Kubernetes中，业务应用是以容器的方式运行，并以Pod（一组密切相关的容器）作为基本调度单位的。为了保证Pod正常的运行和对外服务，Kubernetes集成了**负载均衡**、**健康检查**、**服务质量**、**弹性伸缩**、**变更策略**等平台级服务保障机制，不再需要依赖额外的工具或系统保障业务的稳定性。合理利用Kubernetes的业务稳定性保障机制不仅能够保证服务质量，还能够降低部署和维护复杂度，增加服务发布的灵活性。

#### 负载均衡

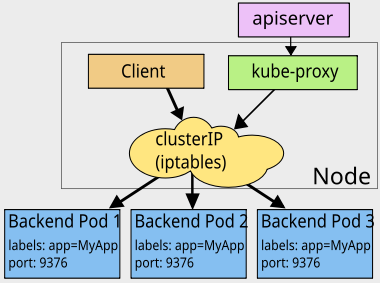
负载均衡既能分摊应用节点的服务压力，又能保障应用的不间断服务，是最典型的应用稳定性保障手段。目前Kubernetes中主要的负载均衡机制及其应用场景如下：

|  |  |
| --- | --- |
| **负载均衡** | **应用场景** |
| Service | 使用Service提供cluster内部的负载均衡，借助cloud provider提供的LB提供外部访问 |
| Ingress Controller | 使用Service提供cluster内部的负载均衡，通过自定义LB提供外部访问 |
| Service Load Balancer | load balancer直接运行在容器中，实现Bare Metal的Service Load Balancer |
| Custom Load Balancer | 自定义负载均衡，并替代kube-proxy，一般在物理部署Kubernetes时使用，方便接入已有的外部服务 |

其中Service和Ingress是两种最常用的负载实现方式:

1. Service

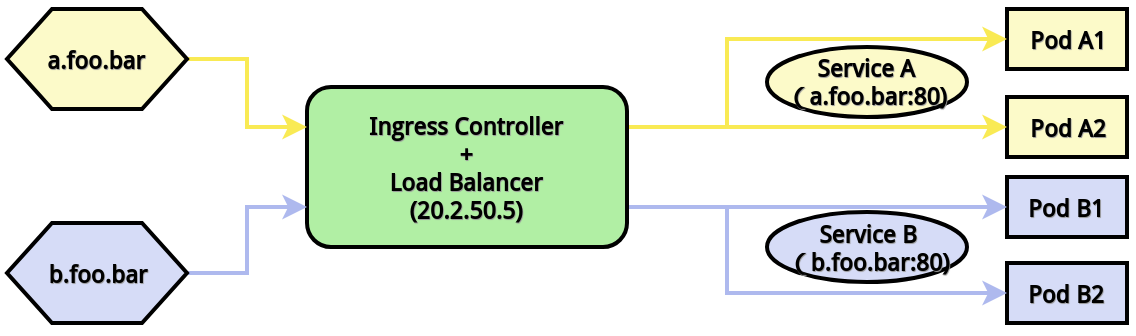
* Service是对一组提供相同功能的Pod的抽象，并为它们提供一个统一的虚IP入口。借助Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service通过标签来选取服务后端，一般配合Replication Controller或者Deployment来保证后端容器的正常运行。



图片源自网络: https://Kubernetes.io/docs/concepts/services-networking/service/

1. Ingress

* Ingress配合相应的Controller和LoadBalancer主要用来将服务暴露到集群之外，并且可以自定义服务的访问策略。Ingress并不会取代Service，而是作用于Service之上，根据Service关联的Endpoint将外部请求代理或转发到对应的Endpoint上。



原创图片

#### 健康检查

当Pod中的容器出现正常或异常退出时，Kubernetes会根据Pod指定的重启策略（restartPolicy）决定处理动作和Pod状态转换，容器的健康状态和容器内应用的健康状态并不总是一致的，容器正常运行时，容器内的应用可能由于阻塞，内部错误等原因无法正常处理请求，或者由于应用尚未初始化完成而无法正常接受请求。为了提供更精确的健康检查，Kubernetes提供了**Probe（探针）**机制。

* Liveness Probe：存活探针，指示容器是否正在运行。如果存活探测失败，则 kubelet 会杀死容器，并且容器将受到其重启策略的影响。如果容器不提供存活探针，则默认状态为 Success。
* ReadinessProbe：就绪探针，指示容器是否准备好服务请求。如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service 的端点中删除该 Pod 的 IP 地址。初始延迟之前的就绪状态默认为 Failure。如果容器不提供就绪探针，则默认状态为 Success。
* StartupProbe：启动探针，指示容器中的应用是否已经启动。如果提供了启动探针，则禁用所有其他探针，直到它成功为止，此后其他探针将接管容器探测。如果启动探针失败，kubelet 将杀死容器，容器服从其重启策略进行重启。

Probe支持以下三种检查方法:

* ExecAction：在容器中执行指定的命令进行检查，当命令执行成功（返回码为0），检查成功。
* TCPSocketAction：对于容器中的指定TCP端口进行检查，当TCP端口被占用，检查成功。
* HTTPGetAction：发生一个HTTP请求，当返回码介于200~400之间时，检查成功。

Probe只是检查容器或应用的健康状态，Pod是否会重启由重启策略根据检查结果进行控制。

#### 服务质量

为了实现资源被有效调度和分配的同时提高资源利用率，Kubernetes针对不同服务质量的预期，通过 QoS（Quality of Service）来对 pod 进行服务质量管理。对于一个pod来说，服务质量体现在两个具体的指标：CPU和内存。当节点上内存资源紧张时，Kubernetes会根据预先设置的不同QoS类别进行相应处理。

QoS主要分为**Guaranteed**、**Burstable**和**Best-Effort**三级，优先级从高到低。可以通过kubectl describe命令或者在Pod的status字段查看其QoS级别。

1. Guaranteed：Pod中的所有容器都且仅设置了 CPU 和内存的 limits或者pod中的所有容器都设置了 CPU 和内存的 requests 和 limits ，且单个容器内的requests==limits（requests不等于0）。
2. Burstable：pod中任意一个容器的requests和limits的设置不相同。
3. Best-Effort：Pod中所有容器的resources均未设置requests与limits。

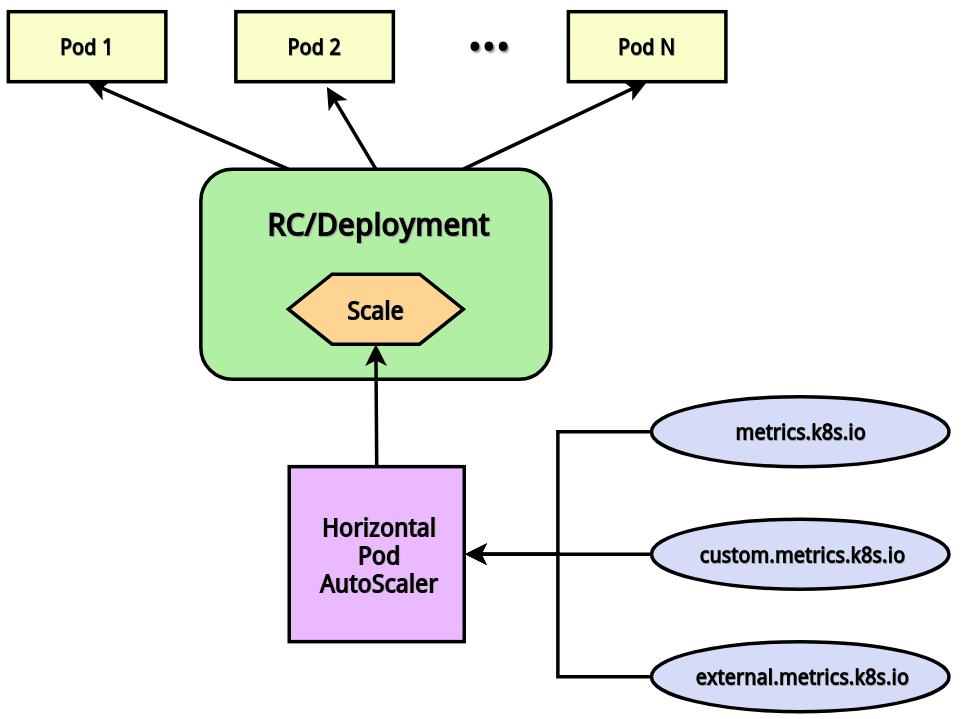
Kubernetes 通过cgroup给Pod设置QoS级别，由于CPU是可抢占式资源，而内存是不可抢占的，也就是说CPU资源不足不会导致Pod被杀死，而内存资源不足会。当内存资源不足时先杀死优先级低的Pod，这是通过系统的OOM评分机制实现的。

#### 弹性伸缩

在实际的应用部署过程中，应用的容量规划和实际负载之间往往是存在落差的，对于业务应用的请求数量在不同时段或不同时期往往存在明显的波峰波谷现象。一方面实际负载难以预料，另一方面如果只按照波峰请求规划容量则难免存在资源浪费，而低于波峰请求的容量规划又可能导致应用节点资源利用率过高，影响应用服务的稳定性。

在Kubernetes平台中为Pod提供了水平自动伸缩（Horizontal Pod Autoscaling，HPA）的特性。HPA可以根据Pod的CPU利用率（或其他应程序提供的度量指标custom metrics）自动伸缩一个Replication Controller、Deployment 或者Replica Set中的Pod数量（或者基于一些应用程序提供的度量指标，目前这一功能处于alpha版本），保证应用的稳定性。

此外Kubernetes还实验性的提供了垂直自动伸缩（Vertical Pod Autoscaler，VPA）特性，它允许根据Pod使用的资源指标自动调整Pod的CPU和内存的requests值。这一特性目前尚不成熟，自动伸缩过程中会导致Pod重启，且不能和HPA一起工作，实际环境中建议使用HPA。



原创图片

Horizontal Pod Autoscaler支持三种度量指标：

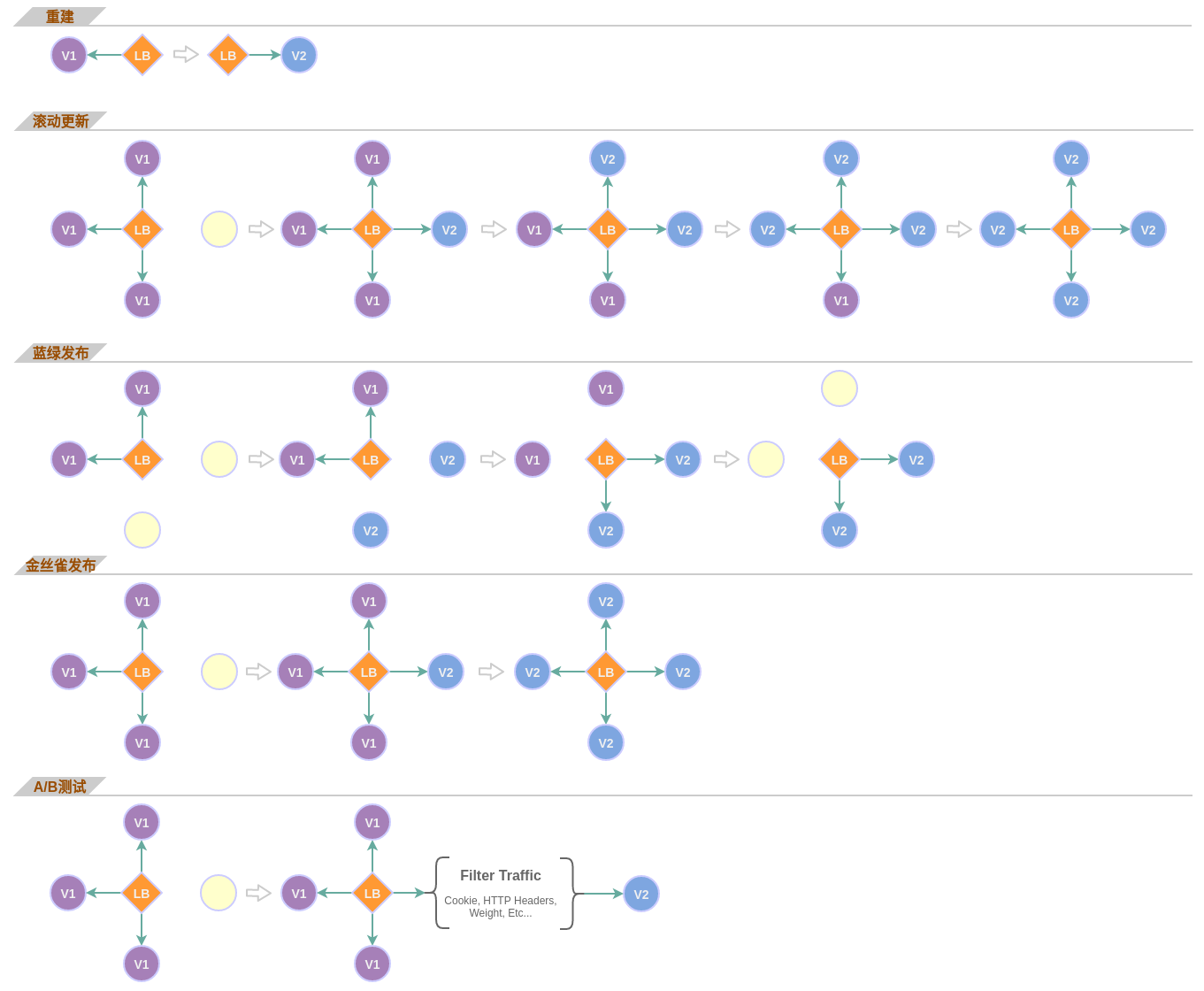
1. 资源度量指标（resource metric）：HPA默认使用的度量指标，这一指标默认度量Pod的CPU利用率，另外也可以选择度量内存占用。
2. Pod度量指标（pod metric）：一种自定义度量指标，这些指标从某一方面描述了Pod，在不同Pod之间进行平均，并通过与一个目标值比对来确定副本的数量。
3. 对象度量指标（object metric）：一种自定义度量指标，这些度量指标用于描述一个在相同名字空间（namespace）中的其他对象。 请注意这些度量指标用于描述这些对象，并非从对象中获取。对象度量指标并不涉及平均计算。

#### 变更策略

应用通常是需要持续维护和更新的，应用在更新维护时根据变更策略的不同会对应用的服务可用性和服务质量产生不同的影响，同时也对应用的运行环境和资源占用提出了不同的要求。

在对应用服务稳定性，资源环境约束和业务场景需求等多方因素进行权衡后，选择合适的的变更策略是非常重要的。Kubernetes中提供了几种不同的应用变更策略以应对不同的需求：

* recreate：重建，停止旧版本后部署新版本。
* rolling-update：滚动更新，以顺序更新的方式发布新版本。
* blue/green：蓝绿发布，新版本与旧版本共存并进行流量切换。
* canary：金丝雀发布，将新版本先面向部分用户发布，然后逐步完成全量发布。
* a/b testing：A/B测试，以精确的方式（HTTP头部、cookie、权重等）向部分用户发布新版本。A/B测是一种基于数据统计做出业务决策的技术，并不是Kubernetes支持的原生策略，需要依赖诸如IstioLinkerd，Traefik等高级组件实现。



原创图片

变更策略各有优劣，Kubernetes实现这些变更策略的方式也有较大差异（有的策略需要手动实现, 具体实现在此略去），此外有一些第三方的Ingress Controller能够对这些变更策略提供更好的支持。用户可以根据实际场景选择合适的变更策略。

## 容器云高可用

**[容器云平台高可用设计-谢楚瑜](https://wiki.ebcpaas.com/" \l "/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9A%84%E9%AB%98%E5%8F%AF%E7%94%A8%E8%AE%BE%E8%AE%A1)(**[https://wiki.ebcpaas.com/#/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9A%84%E9%AB%98%E5%8F%AF%E7%94%A8%E8%AE%BE%E8%AE%A1](https://wiki.ebcpaas.com/" \l "/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9A%84%E9%AB%98%E5%8F%AF%E7%94%A8%E8%AE%BE%E8%AE%A1)**)**

## 容器云监控架构

**[容器云平台监控架构设计-吴涛](https://wiki.ebcpaas.com/" \l "/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9B%91%E6%8E%A7%E6%9E%B6%E6%9E%84%E8%AE%BE%E8%AE%A1%E5%8F%8A%E4%BC%98%E5%8C%96)(**[https://wiki.ebcpaas.com/#/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9B%91%E6%8E%A7%E6%9E%B6%E6%9E%84%E8%AE%BE%E8%AE%A1%E5%8F%8A%E4%BC%98%E5%8C%96](https://wiki.ebcpaas.com/" \l "/posts/%E6%9E%B6%E6%9E%84%E5%88%86%E6%9E%90-%E5%AE%B9%E5%99%A8%E4%BA%91%E5%B9%B3%E5%8F%B0%E7%9B%91%E6%8E%A7%E6%9E%B6%E6%9E%84%E8%AE%BE%E8%AE%A1%E5%8F%8A%E4%BC%98%E5%8C%96)**)**