1. 简介

kubernetes，简称K8s，是用8代替8个字符“ubernete”而成的缩写。是一个开源的，用于管理云平台中多个主机上的容器化的应用，Kubernetes的目标是让部署容器化的应用简单并且高效（powerful）,Kubernetes提供了应用部署，规划，更新，维护的一种机制。 [1]

传统的应用部署方式是通过插件或脚本来安装应用。这样做的缺点是应用的运行、配置、管理、所有生存周期将与当前操作系统绑定，这样做并不利于应用的升级更新/回滚等操作，当然也可以通过创建虚拟机的方式来实现某些功能，但是虚拟机非常重，并不利于可移植性。

新的方式是通过部署容器方式实现，每个容器之间互相隔离，每个容器有自己的文件系统 ，容器之间进程不会相互影响，能区分计算资源。相对于虚拟机，容器能快速部署，由于容器与底层设施、机器文件系统解耦的，所以它能在不同云、不同版本操作系统间进行迁移。

容器占用资源少、部署快，每个应用可以被打包成一个容器镜像，每个应用与容器间成一对一关系也使容器有更大优势，使用容器可以在build或release 的阶段，为应用创建容器镜像，因为每个应用不需要与其余的应用堆栈组合，也不依赖于生产环境基础结构，这使得从研发到测试、生产能提供一致环境。类似地，容器比虚拟机轻量、更“透明”，这更便于监控和管理。

2.Kubernetes概述

Kubernetes是Google开源的一个容器编排引擎，它支持自动化部署、大规模可伸缩、应用容器化管理。在生产环境中部署一个应用程序时，通常要部署该应用的多个实例以便对应用请求进行负载均衡。

在Kubernetes中，我们可以创建多个容器，每个容器里面运行一个应用实例，然后通过内置的负载均衡策略，实现对这一组应用实例的管理、发现、访问，而这些细节都不需要运维人员去进行复杂的手工配置和处理。

通过Kubernetes你可以：

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用功能
- 节省资源，优化硬件资源的使用

我们的目标是促进完善组件和工具的生态系统，以减轻应用程序在公有云或私有云中运行的负担。

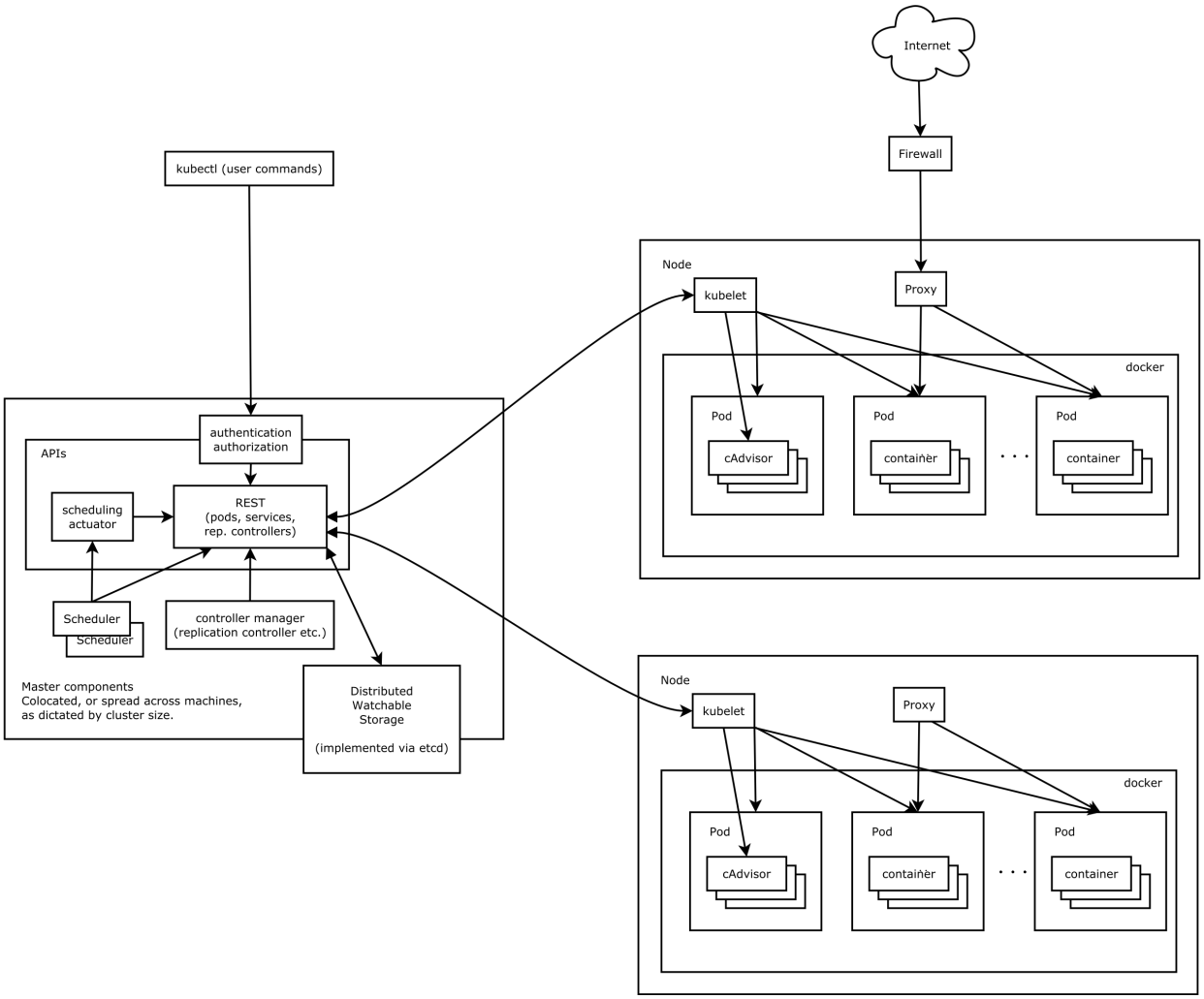
3.Kubernetes特点

可以在物理或虚拟机的Kubernetes集群上运行容器化应用，Kubernetes能提供一个以“**容器为中心的基础架构**”，满足在生产环境中运行应用的一些常见需求，如：

- 多个进程（作为容器运行）协同工作。（Pod）
- 存储系统挂载
- Distributing secrets
- 应用健康检测
- 应用实例的复制
- Pod自动伸缩/扩展
- Naming and discovering
- 负载均衡
- 滚动更新
- 资源监控
- 日志访问
- 调试应用程序
- 提供认证和授权

4.Kubernetes架构

Kubernetes借鉴了Borg的设计理念，比如Pod、Service、Labels和单Pod单IP等。
Kubernetes的整体架构跟Borg非常像，如下图所示:



Kubernetes主要由以下核心组件组成：

1.API Server

作为Kubernetes系统的入口，其封装了核心对象的增删改查操作，以RESTful API接口方式提供给外部客户和内部组件调用。维护的REST对象持久化到Etcd中存储。apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；

2.Scheduler

为新建立的Pod进行节点(node)选择(即分配机器)，按照预定的调度策略将Pod调度到相应的node上；负责集群的资源调度。组件抽离，可以方便替换成其他调度器。

3.Controller Manager

负责执行各种控制器，目前已经提供了很多控制器来保证Kubernetes的正常运行。

- Replication Controller

管理维护Replication Controller，关联Replication Controller和Pod，保证Replication Controller定义的副本数量与实际运行Pod数量一致。

- Node Controller

管理维护Node，定期检查Node的健康状态，标识出(失效|未失效)的Node节点。

- Namespace Controller

管理维护Namespace，定期清理无效的Namespace，包括Namesapce下的API对象，比如Pod、Service等。

- Service Controller

管理维护Service，提供负载以及服务代理。

- EndPoints Controller

管理维护Endpoints，关联Service和Pod，创建Endpoints为Service的后端，当Pod发生变化时，实时更新Endpoints。

- Service Account Controller

管理维护Service Account，为每个Namespace创建默认的Service Account，同时为Service Account创建Service Account Secret。

- Persistent Volume Controller

管理维护Persistent Volume和Persistent Volume Claim，为新的Persistent Volume Claim分配Persistent Volume进行绑定，为释放的Persistent Volume执行清理回收。

- Daemon Set Controller

管理维护Daemon Set，负责创建Daemon Pod，保证指定的Node上正常的运行Daemon Pod。

- Deployment Controller

管理维护Deployment，关联Deployment和Replication Controller，保证运行指定数量的Pod。当Deployment更新时，控制实现Replication Controller和 Pod的更新。

- Job Controller

管理维护Job，为Jod创建一次性任务Pod，保证完成Job指定完成的任务数目

- Pod Autoscaler Controller

实现Pod的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执行Pod的伸缩动作。

4.scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的node上；

5.kubelet负责维护容器的生命周期，同时也负责Volume（CVI）和网络（CNI）的管理；

6.Container runtime负责镜像管理以及Pod和容器的真正运行（CRI）；

7.kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

除了核心组件，还有一些推荐的Add-ons：

8.kube-dns负责为整个集群提供DNS服务

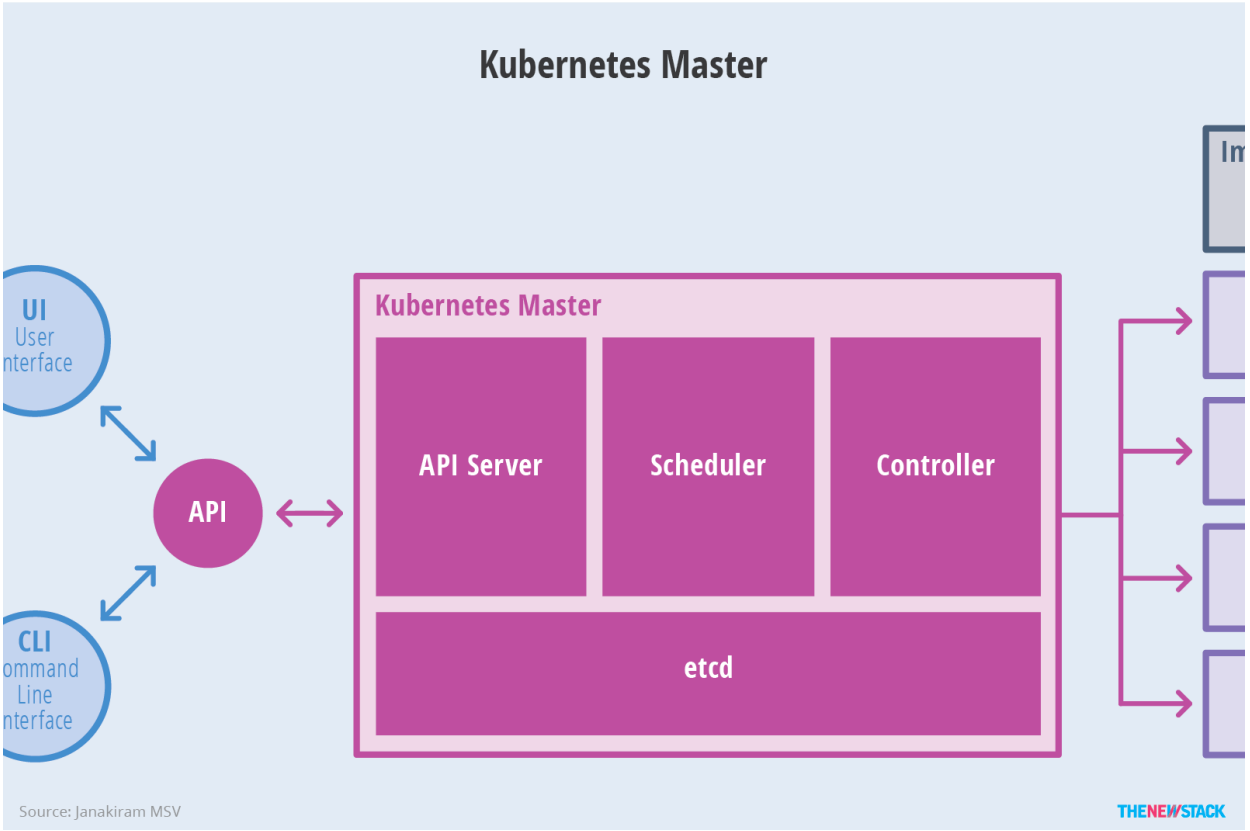
9.Ingress Controller为服务提供外网入口

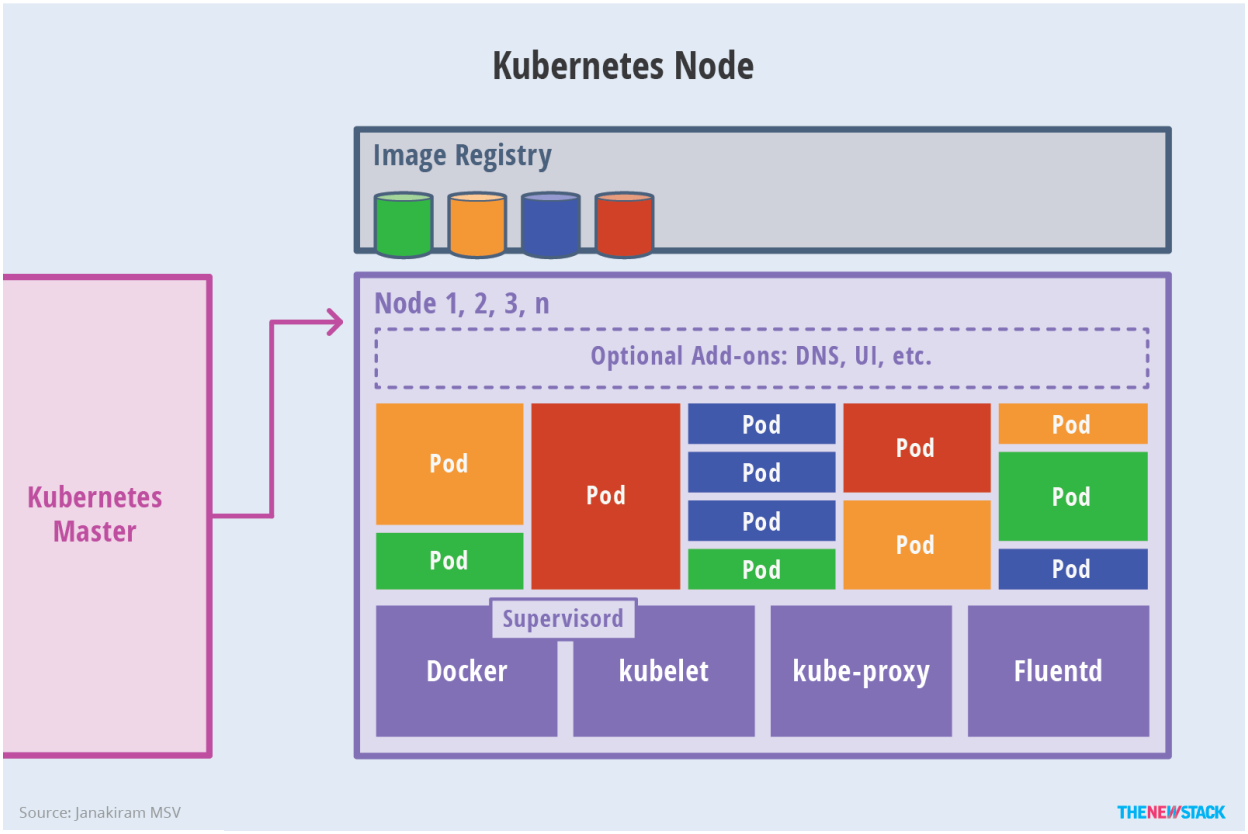
10.Heapster提供资源监控

11.Dashboard提供GUI

12.Federation提供跨可用区的集群

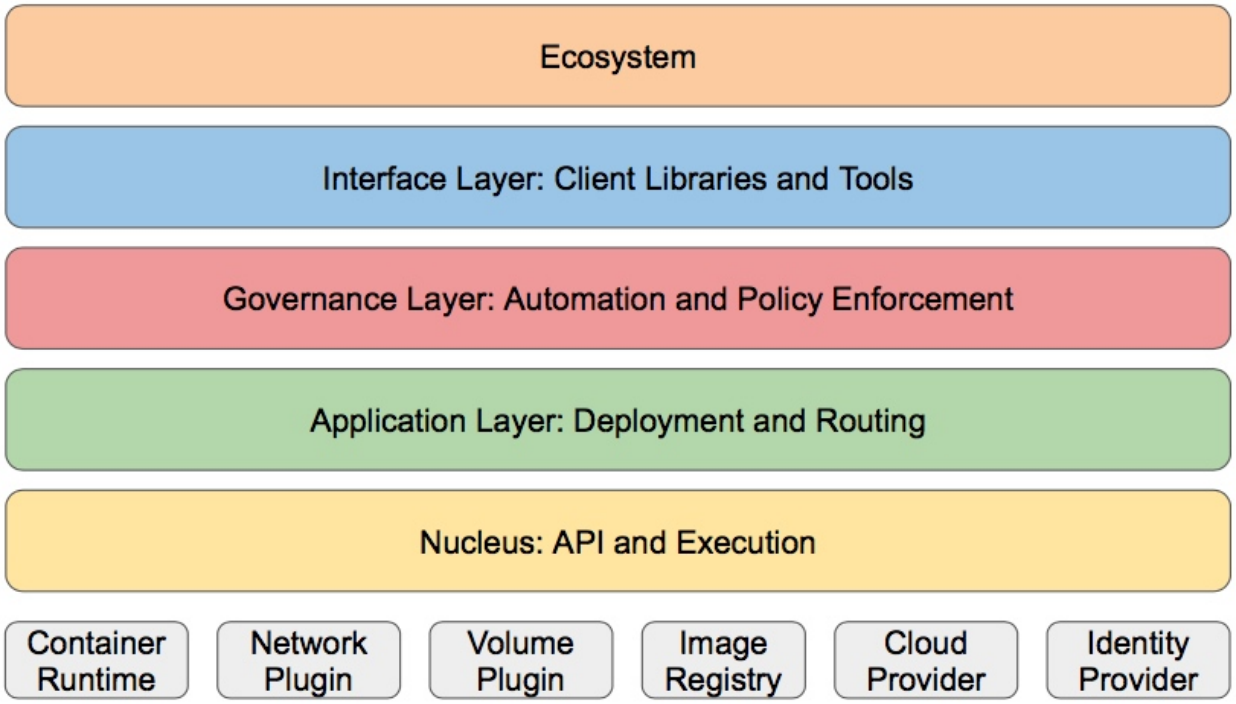
13.Fluentd-elasticsearch提供集群日志采集、存储与查询





分层架构

Kubernetes设计理念和功能其实就是一个类似Linux的分层架构，如下图所示



- 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）以及策略管理（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

5.Kubernetes的核心技术概念和API对象

在Kubernetes中，几乎一切都是对象,Kubernetes对象本质上一种用于持久化的实体。常见的对象包括：Node，Pod，Deployment，ReplicationController， ReplicaSet等等。我们通常通过在描述文件中指定kind来创建不同种类的对象。Kubernetes通过etcd存储我们创建的对象，从而使应用按照你期望的方式稳定运行在容器中。
通常一个Kubernetes对象可以包含以下信息：

- 需要运行的应用以及运行在哪些node上
- 应用可以使用哪些资源
- 应用运行时的一些配置，例如重启策略，升级以及容错性

因此，一个Kubernetes对象，其实就是你意图的体现（通过.yaml文件来描述）。一旦你创建了一个对象后，Kubernetes会确保这个对象一直处于你所期望的状态。

API对象是K8s集群中的管理操作单元。K8s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。

metadata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。

spec规范描述了用户期望K8s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。
K8s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k8s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。

类别	名称
工作负载型资源对象	Pod Replicaset ReplicationController Deployments StatefulSets Daemonset Job CronJob
服务发现及负载均衡	Service Ingress
配置与存储	Volume、Persistent Volume、CSl 、 configmap、 secret
集群资源	Namespace Node Role ClusterRole RoleBinding ClusterRoleBinding
元数据资源	HPA PodTemplate LimitRang

- **Pod**

Pod是Kubernetes创建或部署的最小/最简单的基本单位，一个Pod代表集群上正在运行的一个进程。

一个Pod封装一个应用容器（也可以有多个容器），存储资源、一个独立的网络IP以及管理控制容器运行方式的策略选项。Pod代表部署的一个单位：Kubernetes中单个应用的实例，它可能由单个容器或多个容器共享组成的资源。

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。

- **Replication Controller，RC**

RC是K8s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K8s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。

- **Replica Set，RS**

大多数kubectl 支持Replication Controller 命令的也支持ReplicaSets。rolling-update命令除外，如果要使用rolling-update，请使用Deployments来实现。
虽然ReplicaSets可以独立使用，但它主要被 Deployments用作pod 机制的创建、删除和更新。当使用Deployment时，你不必担心创建pod的ReplicaSets，因为可以通过Deployment实现管理ReplicaSets。

- **Deployment**

Deployment为Pod和Replica Set（升级版的 Replication Controller）提供声明式更新。
你只需要在 Deployment 中描述您想要的目标状态是什么，Deployment controller 就会帮您将 Pod 和ReplicaSet 的实际状态改变到您的目标状态。您可以定义一个全新的Deployment 来创建 ReplicaSet 或者删除已有的 Deployment 并创建一个新的来替换。

- **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K8s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K8s集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是K8s集群内部的负载均衡器。它是一个分布式代理服务器，在K8s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做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

- **Job**

对于ReplicaSet、ReplicationController等类型的控制器而言，它希望pod保持预期数目、持久运行下去，除非用户明确删除，否则这些对象一直存在，它们针对的是耐久性任务，如web服务等。对于非耐久性任务，比如压缩文件，任务完成后，pod需要结束运行，不需要pod继续保持在系统中，这个时候就要用到Job。因此说Job是对ReplicaSet、ReplicationController等持久性控制器的补充。

- **DaemonSet**

DaemonSet确保集群中每个（部分）node运行一份pod副本，当node加入集群时创建pod，当node离开集群时回收pod。如果删除DaemonSet，其创建的所有pod也被删除，DaemonSet中的pod覆盖整个集群。

当需要在集群内每个node运行同一个pod，使用DaemonSet是有价值的，以下是典型使用场景：

运行集群存储守护进程，如glusterd、ceph。

运行集群日志收集守护进程，如fluentd、logstash。

运行节点监控守护进程，如Prometheus Node Exporter, collectd, Datadog agent, New Relic agent, or Ganglia gmond。

- StatefulSet

RC、Deployment、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，有状态的集合，管理所有有状态的服务

StatefulSet本质上是Deployment的一种变体，在v1.9版本中已成为GA版本，它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)，还必须要用到共享存储。

在Deployment中，与之对应的服务是service，而在StatefulSet中与之对应的headless service，headless service，即无头服务，与service的区别就是它没有Cluster IP，解析它的名称时将返回该Headless Service对应的全部Pod的Endpoint列表。

除此之外，StatefulSet在Headless Service的基础上又为StatefulSet控制的每个Pod副本创建了一个DNS域名，这个域名的格式为：

- **Volume**

K8s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K8s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。K8s支持非常多的存储卷类型，特别的，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。K8s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。

- **Persistent Volume，PV**

PersistentVolume（一些简称PV）：由管理员添加的的一个存储的描述，是一个全局资源，包含存储的类型，存储的大小和访问模式等。它的生命周期独立于Pod，例如当使用

它的Pod销毁时对PV没有影响。

- **Persistent Volume Claim，PVC**

PersistentVolumeClaim（一些简称PVC）：是Namespace里的资源，描述对PV的一个请求。请求信息包含存储大小，访问模式等。

- **Node**

K8s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。K8s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。

- **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。

6.kubernetes存储

由于容器本身是非持久化的，因此需要解决在容器中运行应用程序遇到的一些问题。首先，当容器崩溃时，kubelet将重新启动容器，但是写入容器的文件将会丢失，容器将会以镜像的初始状态重新开始；第二，在通过一个Pod中一起运行的容器，通常需要共享容器之间一些文件。Kubernetes通过存储卷解决上述的两个问题。

Volume 类型

Kubernetes支持Volume类型有：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- fc (fibre channel)
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo

- secret
- persistentVolumeClaim
- downwardAPI
- projected
- azureFileVolume
- azureDisk
- vsphereVolume
- Quobyte
- PortworxVolume
- ScaleIO
- StorageOS
- local

emptyDir

使用emptyDir，当Pod分配到Node上时，将会创建emptyDir，并且只要Node上的Pod一直运行，Volume就会一直存。当Pod（不管任何原因）从Node上被删除时，emptyDir也同时会删除，存储的数据也将永久删除。注：删除容器不影响emptyDir。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath

hostPath允许挂载Node上的文件系统到Pod里面去。如果Pod需要使用Node上的文件，可以使用hostPath。

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
```

```
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
```

NFS

NFS 是Network File System的缩写，即网络文件系统。Kubernetes中通过简单地配置就可以挂载NFS到Pod中，而NFS中的数据是可以永久保存的，同时NFS支持同时写操作。Pod被删除时，Volume被卸载，内容被保留。这就意味着NFS能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间相互传递。

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  containers:
    - name: web
      image: nginx
      imagePullPolicy: Never        #如果已经有镜像，就不需要再拉取镜像
      ports:
        - name: web
          containerPort: 80
          hostPort: 80        #将容器的80端口映射到宿主机的80端口
      volumeMounts:
        - name : nfs        #指定名称必须与下面一致
          mountPath: "/usr/share/nginx/html"        #容器内的挂载点
  volumes:
    - name: nfs            #指定名称必须与上面一致
      nfs:            #nfs存储
        server: 192.168.66.50        #nfs服务器ip或是域名
        path: "/test"                #nfs服务器共享的目录
```

RBD

RBD允许Rados Block Device格式的磁盘挂载到Pod中，同样的，当pod被删除的时候，rbd也仅仅是被卸载，内容保留，rbd能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

```
apiVersion: v1
kind: Pod
metadata:
  name: rbd
spec:
  containers:
    - image: kubernetes/pause
      name: rbd-rw
      volumeMounts:
      - name: rbdpd
        mountPath: /mnt/rbd
  volumes:
    - name: rbdpd
      rbd:
        monitors:
        - '10.16.154.78:6789'
        - '10.16.154.82:6789'
        - '10.16.154.83:6789'
        pool: kube
        image: foo
        fsType: ext4
        readOnly: true
        user: admin
        keyring: /etc/ceph/keyring
        imageformat: "2"
        imagefeatures: "layering"
```

monitors：这是 Ceph 集群的 monitor 监视器，Ceph 集群可以配置多个 monitor，如有多多个配置所有，本次我们搭建的 Ceph 集群只有一个 monitor，所以这里需要修改为 10.222.76.119:6789
pool：这是 Ceph 集群中存储数据进行归类区分使用，可使用 ceph osd pool ls 命令列出所有，默认创建的 pool 为 rbd，所以这里可以修改为 rbd，也可以创建一个新的名称为 kube 的 pool。
image：这是 Ceph 块设备中的磁盘映像文件，可使用 rbd create ... 命令创建指定大小的映像，这里我们就创建 foo
fsType：文件系统类型，默认使用 ext4 即可。
readOnly：是否为只读，这里测试使用只读即可。

user：这是 Ceph Client 访问 Ceph 存储集群所使用的用户名，这里我们使用 admin 即可。
keyring：这是 Ceph 集群认证需要的密钥环，记得搭建 Ceph 存储集群时生成的 ceph.client.admin.keyring 么，就是这个文件。
imageformat：这是磁盘映像文件格式，可以使用 2，或者老一些的 1
imagefeatures： 这是磁盘映像文件的特征，需要 uname -r 查看集群系统内核所支持的特性，这里我们安装的 Ceontos7 内核版本为 3.10.0-693.5.2.el7.x86_64 只支持 layering。

PV和PVC

一旦 Pod 删除掉，那么挂载 volume 中的数据就不会存在了，那是因为 volume 跟 pod 的生命周期是一样的。那么有什么办法可以解决呢？ 也就是即使 pod 被删除，volume 中的数据依旧存在。为此，k8s 提供了两种 API 资源方式：PersistentVolume 和 PersistentVolumeClaim 来解决这个问题。

PersistentVolume (PV) 可以理解为在集群中已经由管理员配置的一块存储，作为集群的资源，而且拥有独立与 Pod 的生命周期，意思就是 Pod 删除了，但 PV 还在，PV 上的数据依旧存在。
PersistentVolumeClaim (PVC) 可以理解为用户对存储资源的请求，跟 Pod 类似，Pod 消耗节点资源，PVC 消耗 PV 资源，Pod 可以配置请求分配特定大小的资源，比如 CPU、内存，PVC 可以配置请求特定大小资源和访问方式，比如 RWO（ReadWriteOnce） ROX（ReadOnlyMany） RWX（ReadWriteMany）。
PV 支持 Static 静态请求，即提前准备好固定大小的资源。同时支持 Dynamic 动态请求，当静态 PV 不能满足需求时，k8s 集群可以提供动态分配的方式，但是需要配置 StorageClasses。

Provisioning

PV可以通过两种方式提供：
Static：管理员在集群里创建PV资源，每个PV包含详细的真实存储的信息供PVC使用。
Dynamic：当集群里没有PV符合PVC请求时，集群会尝试动态生成PV。前提是管理员提供过StorageClass资源并且PVC里有StorageClass的描述。

Binding

当集群中新添加一个PVC时，k8s里的PVController（下一篇文章介绍）会试图查找最合适（存储大小和访问模式）的PV并建立绑定关系。 最合适的意思是PVC一定满足PV的要求，单也可能比PVC要求的要多，例如PVC请求5G存储，但当前最小的PV是10G，那么这个PV也会被分配给PVC。 注意一个PV只能绑定给一个PVC。

Reclaiming

有三种回收策略：
* Retained：PV会保持原有数据并允许用户手动回收数据。

* Recycled：删除数据（rm -rf /thevolume/*）并允许PV被绑定到其它PVC。
* Deleted： 删除数据并删除PV。

创建PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: ceph-rbd-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  rbd:
    monitors:
      - 10.222.76.119:6789
    pool: rbd
    image: ceph-rbd-pv-test
    user: admin
    secretRef:
      name: ceph-secret
    fsType: ext4
    readOnly: false
  persistentVolumeReclaimPolicy: Recycle
```

Capacity

通过capacity给PV设置特定的大小。

Access Modes

k8s不会真正检查存储的访问模式或根据访问模式做访问限制，只是对真实存储的描述，最终的控制权在真实的存储端。目前支持三种访问模式：

* ReadWriteOnce – PV以 read-write 挂载到一个节点
* ReadOnlyMany – PV以read-only方式挂载到多个节点
* ReadWriteMany – PV以read-write方式挂载到多个节点

Reclaim

当前支持的回收策略:

* Retain – 允许用户手动回收
* Recycle – 删除PV上的数据（“rm -rf /thevolume/*”）
* Delete – 删除PV

Phase

Available – PV可以被使用

Bound – PV被绑定到PVC
Released – 被绑定的PVC被删除，可以被Reclaim
Failed – 自动回收失败

创建PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ceph-rbd-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Phase

- Pending – 等待可用的PV
- Bound – PV被绑定到PVC
- Lost – 找不到绑定的PV

创建挂载PVC的POD

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: rbd-pvc-pod
  name: ceph-rbd-pv-pod1
spec:
  containers:
  - name: ceph-rbd-pv-busybox
    image: busybox
    command: ["sleep", "60000"]
    volumeMounts:
    - name: ceph-rbd-vol1
      mountPath: /mnt/ceph-rbd-pvc/busybox
      readOnly: false
  volumes:
  - name: ceph-rbd-vol1
```

```
  persistentVolumeClaim:
    claimName: ceph-rbd-pv-claim
```

cephfs

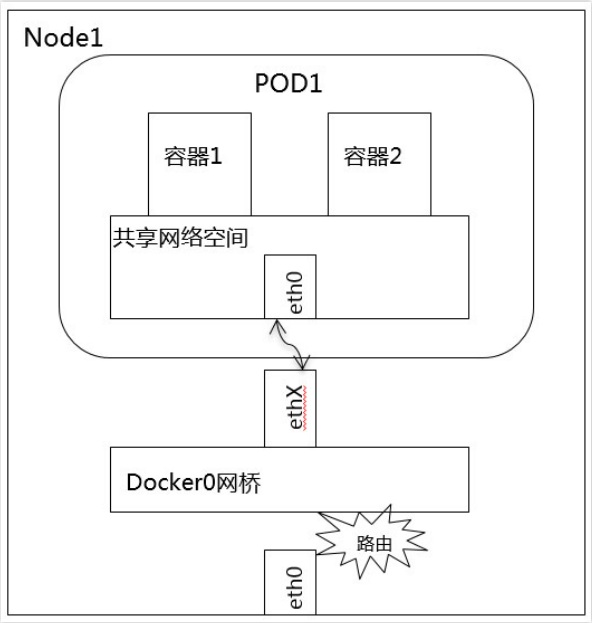
cephfs Volume可以将已经存在的CephFS Volume挂载到pod中，与emptyDir特点不同，pod被删除的时，cephfs仅被被卸载，内容保留。cephfs能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

Kubernetes网络模型

在Kubernetes网络中存在两种IP（Pod IP和Service Cluster IP），Pod IP 地址是实际存在于某个网卡(可以是虚拟设备)上的，Service Cluster IP它是一个虚拟IP，是由kube-proxy使用Iptables规则重新定向到其本地端口，再均衡到后端Pod的。

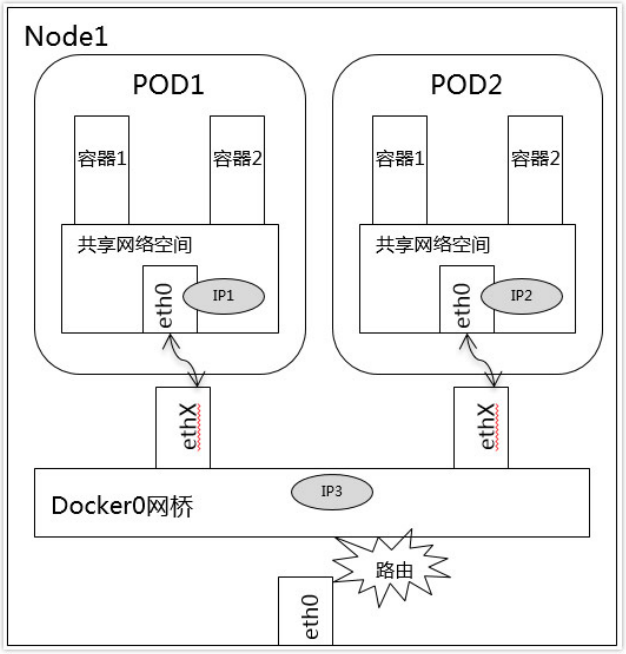
1、容器间通信：

同一个Pod的容器共享同一个网络命名空间，它们之间的访问可以用localhost地址 + 容器端口就可以访问。



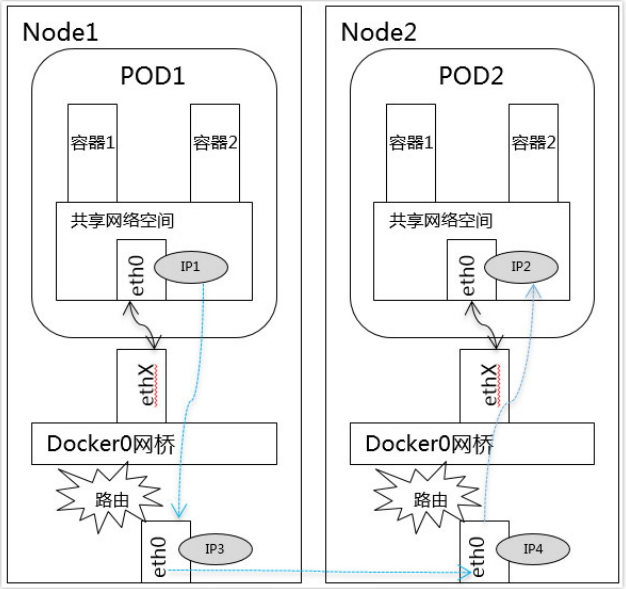
2、同一Node中Pod间通信：

同一Node中Pod的默认路由都是docker0的地址，由于它们关联在同一个docker0网桥上，地址网段相同，所有它们之间应当是能直接通信的。



3、不同Node中Pod间通信：

不同Node中Pod间通信要满足2个条件： Pod的IP不能冲突； 将Pod的IP和所在的Node的IP关联起来，通过这个关联让Pod可以互相访问。



容器网络方案：

隧道方案（Overlay Networking）

隧道方案在IaaS层的网络中应用也比较多，大家共识是随着节点规模的增长复杂度会提升，而且出了网络问题跟踪起来比较麻烦，大规模集群情况下这是需要考虑的一个点。

- **Weave**：UDP广播，本机建立新的BR，通过PCAP互通
- **Open vSwitch（OVS）**：基于VxLan和GRE协议，但是性能方面损失比较严重
- **Flannel**：UDP广播，VxLan
- **Racher**：IPsec

路由方案

路由方案一般是从3层或者2层实现隔离和跨主机容器互通的，出了问题也很容易排查。

- **Calico：**基于BGP协议的路由方案，支持很细致的ACL控制，对混合云亲和度比较高。
- **Macvlan：**从逻辑和Kernel层来看隔离性和性能最优的方案，基于二层隔离，所以需要二层路由器支持，大多数云服务商不支持，所以混合云上比较难以实现。

Flannel容器网络：

Flannel之所以可以搭建kubernets依赖的底层网络，是因为它可以实现以下两点：

- 它给每个node上的docker容器分配相互不想冲突的IP地址；
- 它能给这些IP地址之间建立一个覆盖网络，同过覆盖网络，将数据包原封不动的传递到目标容器内。

Calico介绍

- Calico是一个纯3层的数据中心网络方案，而且无缝集成像OpenStack这种IaaS云架构，能够提供可控的VM、容器、裸机之间的IP通信。Calico不使用重叠网络比如flannel和libnetwork重叠网络驱动，它是一个纯三层的方法，使用虚拟路由代替虚拟交换，每一台虚拟路由通过BGP协议传播可达信息（路由）到剩余数据中心。