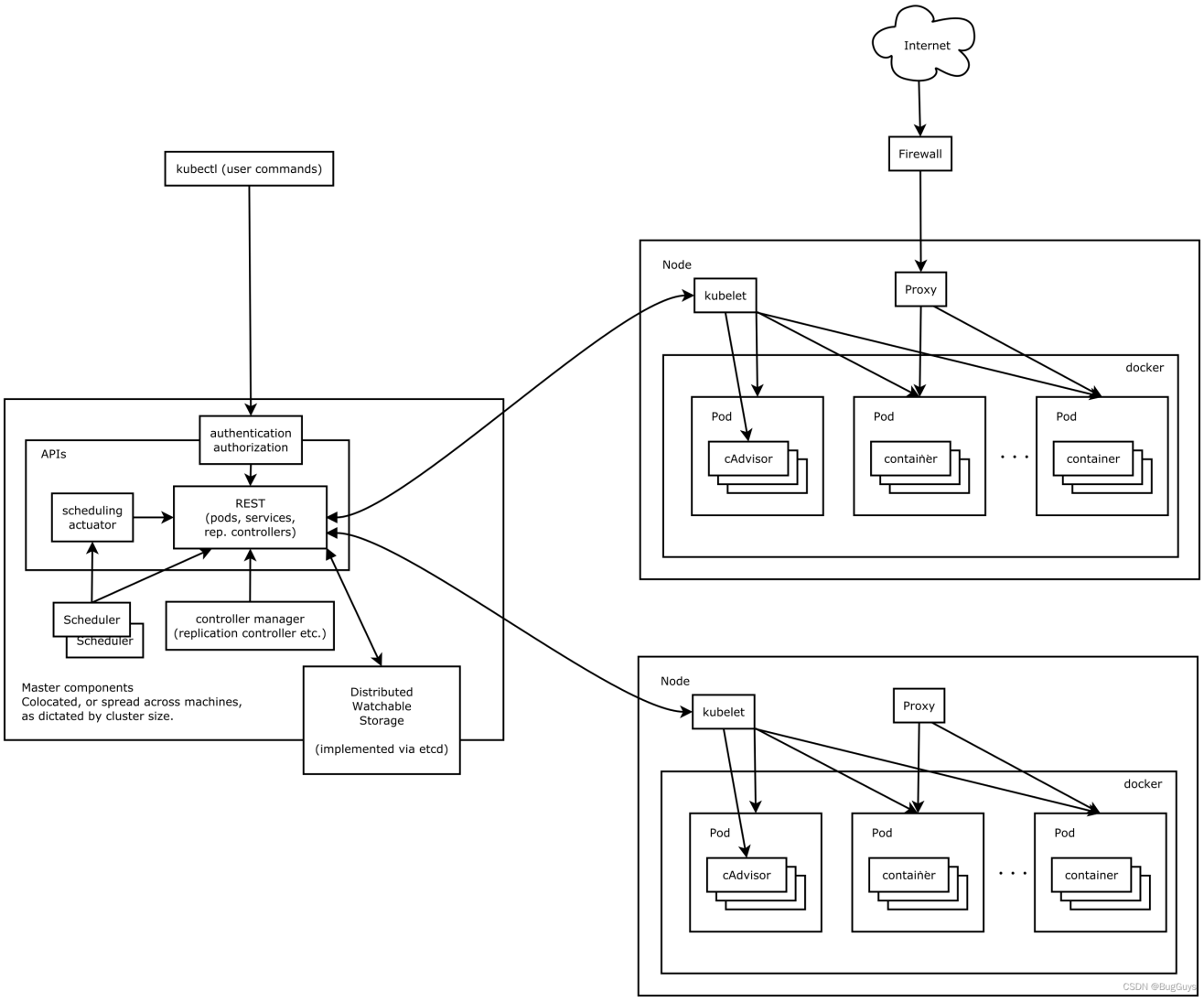
K8s架构|全面整理K8s的架构介绍_BugGuys的博客-CSDN博客

K8S架构与核心技术介绍

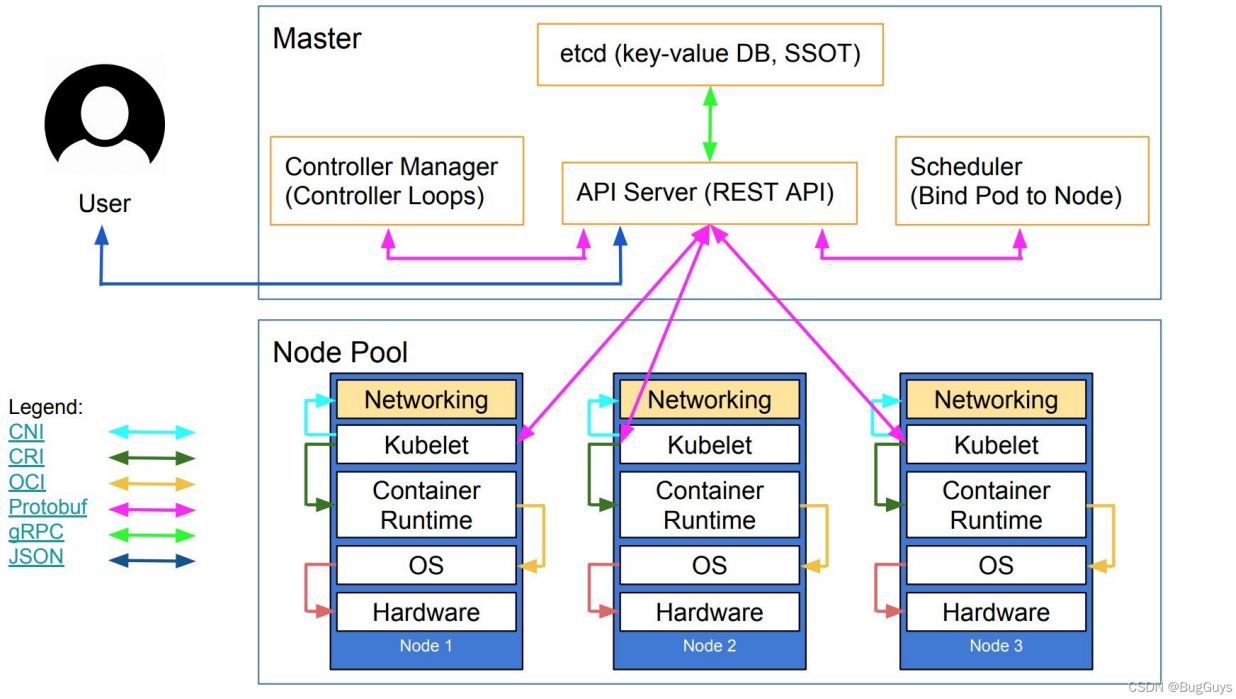
> 参考文献：
>
> https://jimmysong.io/kubernetes-handbook/concepts/concepts.html
>
> https://www.infoq.cn/article/kubernetes-and-cloud-native-applications-part01/

1. 架构图

1.1 整体结构图



1.2 组件间的协议



- CNI: CNI是Container Network Interface的是一个标准的，通用的接口 ;用于连接容器管理系统和网络插件。提供一个容器所在的network namespace，将network interface插入该network namespace中（比如veth的一端），并且在宿主机做一些必要的配置（例如将veth的另一端加入bridge中），最后对namespace中的interface进行IP和路由的配置。现有解决方案：flannel，calico，weave。
  参考链接：https://blog.csdn.net/zhonglinzhang/article/details/82697524

- CRI： 容器运行时接口(Container Runtime Interface)；CRI包含了一组protocol buffers，gRPC API，相关的库; 提供可插拔的容器运行时 ;k8s节点的底层由一个叫做“容器运行时”的软件进行支撑，它负责比如启停容器这样的事情；Docker是K8s中最常用的容器运行时；

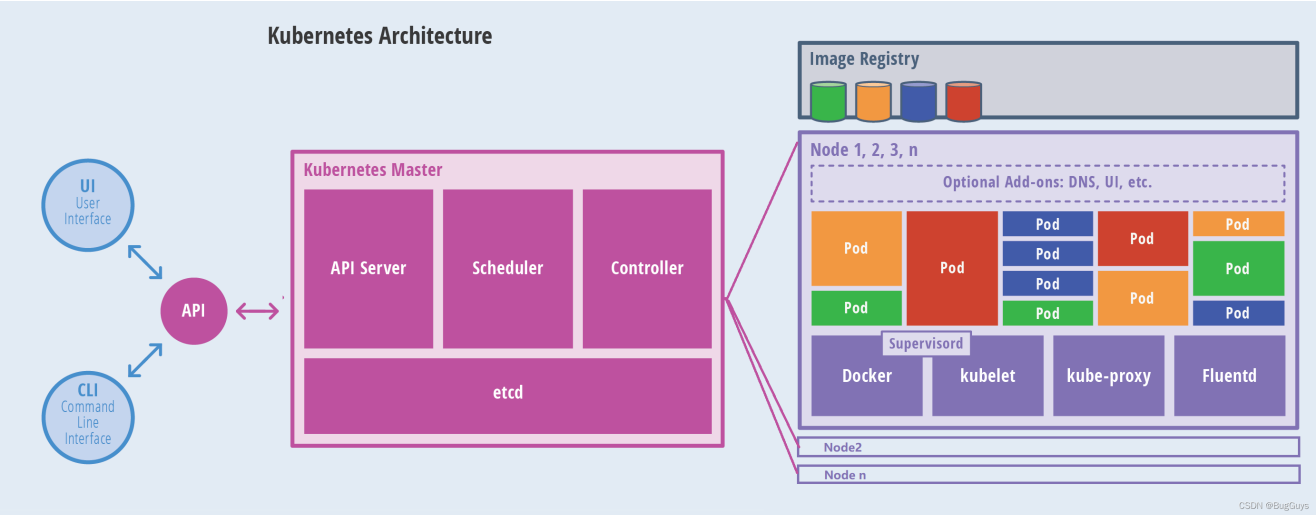
  参考链接：https://www.kubernetes.org.cn/1079.html

- OCI： 围绕容器的格式和运行时制定一个开放的工业化标准,并推动这个标准,保持容器的灵活性和开放性,容器能运行在任何的硬件和系统上，容器不应该绑定到特定的客户机或编排堆栈,不应该与任何特定的供应商紧密关联,并且可以跨多种操作系统；

  参考链接：https://www.jianshu.com/p/c7748893ab00

  官网：https://opencontainers.org/

1.3 master与node架构图



1.4 分层架构图

核心组件：

- etcd 保存了整个集群的状态；

  > etcd是构建一个高可用的分布式键值(key-value)数据库。etcd内部采用`raft`协议作为一致性算法，etcd基于Go语言实现 ;主要用于共享配置和服务发现；
  >
  > 原理动画演示：http://thesecretlivesofdata.com/raft/

- 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；

- controller manager 负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；

- scheduler 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上；
- kubelet 负责维护容器的生命周期，同时也负责 Volume（CSI）和网络（CNI）的管理；
- Container runtime 负责镜像管理以及 Pod 和容器的真正运行（CRI）；
- kube-proxy 负责为 Service 提供 cluster 内部的服务发现和负载均衡；

分层介绍：

- 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）、Service Mesh（部分位于应用层）；
- 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）、Service Mesh（部分位于管理层）；
- 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、GitOps、SecOps 等
  - Kubernetes 内部：CRI、CNI、CSI、镜像仓库、Cloud Provider、集群自身的配置和管理等

---

2. K8s核心技术概念

2.1 API对象

API对象是K8S的集群中管理操作单元；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：标识API对象；

API对象至少含有3个元数据：namespace,name,uid;

API对象通过spec去设置配置； 用户通过配置系统的理想状态来改变系统 ， 所有的操作都是声明（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 ；

2.2 Pod

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 K8 最基础的设计理念 ；

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

业务	API对象
长期伺服型（long-running）	Deployment
批处理型（batch）	Job
节点后台支撑型（node-daemon）	DaemonSet
有状态应用型（stateful application）	StatefulSet

2.3 RC：副本控制器 Replication Controller

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。

2.4 RS：副本集 Replica Set

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。

> 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；
>
> 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

2.5 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。

2.6 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 K8 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。

2.7 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。

2.8 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持 Kubernetes 集群运行的服务。

2.9 有状态服务集：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；

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。

2.10 Volume：存储卷

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；

2.11 Node：节点

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。

2.12 Secret：密钥对象

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。在 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。

2.13 namespace：命名空间

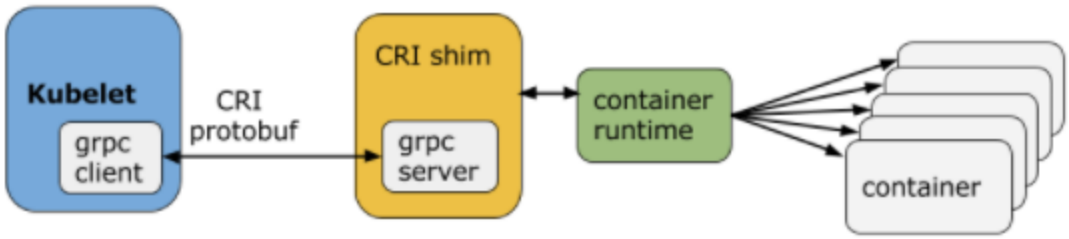
命名空间为 Kubernetes 集群提供虚拟的隔离作用，Kubernetes 集群初始有两个命名空间，分别是默认命名空间 default 和系统命名空间 kube-system，除此以外，管理员可以可以创建新的命名空间满足需要。

2.14 开放接口

CRI

容器运行时接口(Container Runtime Interface)； CRI中定义了**容器**和**镜像**的服务的接口，因为容器运行时与镜像的生命周期是彼此隔离的，因此需要定义两个服务。该接口使用Protocol Buffer，基于gRPC ；

Container Runtime实现了CRI gRPC Server，包括`RuntimeService`和`ImageService`。该gRPC Server需要监听本地的Unix socket，而kubelet则作为gRPC Client运行。



- **RuntimeService**：容器和Sandbox运行时管理。
- **ImageService**：提供了从镜像仓库拉取、查看、和移除镜像的RPC。

CNI

Container Network Interface的是一个标准的通用的接口 ； 由一组用于配置 Linux 容器的网络接口的规范和库组成，同时还包含了一些插件。CNI 仅关心容器创建时的网络分配，和当容器被删除时释放网络资源。

CSI

Container Storage Interface代表容器存储接口； 借助 CSI 容器编排系统（CO）可以将任意存储系统暴露给自己的容器工作负载 ； 部署 CSI 兼容卷驱动后，用户可以使用 `csi` 作为卷类型来挂载驱动提供的存储。

2.15 资源对象

类别	名称
资源对象	Pod、ReplicaSet、ReplicationController、Deployment、StatefulSet、DaemonSet、Job、CronJob、HorizontalPodAutoscaling、Node、Namespace、Service、Ingress、Label、CustomResourceDefinition
存储对象	Volume、PersistentVolume、Secret、ConfigMap
策略对象	SecurityContext、ResourceQuota、LimitRange
身份对象	ServiceAccount、Role、ClusterRole

3. 集群资源管理

3.1 Node

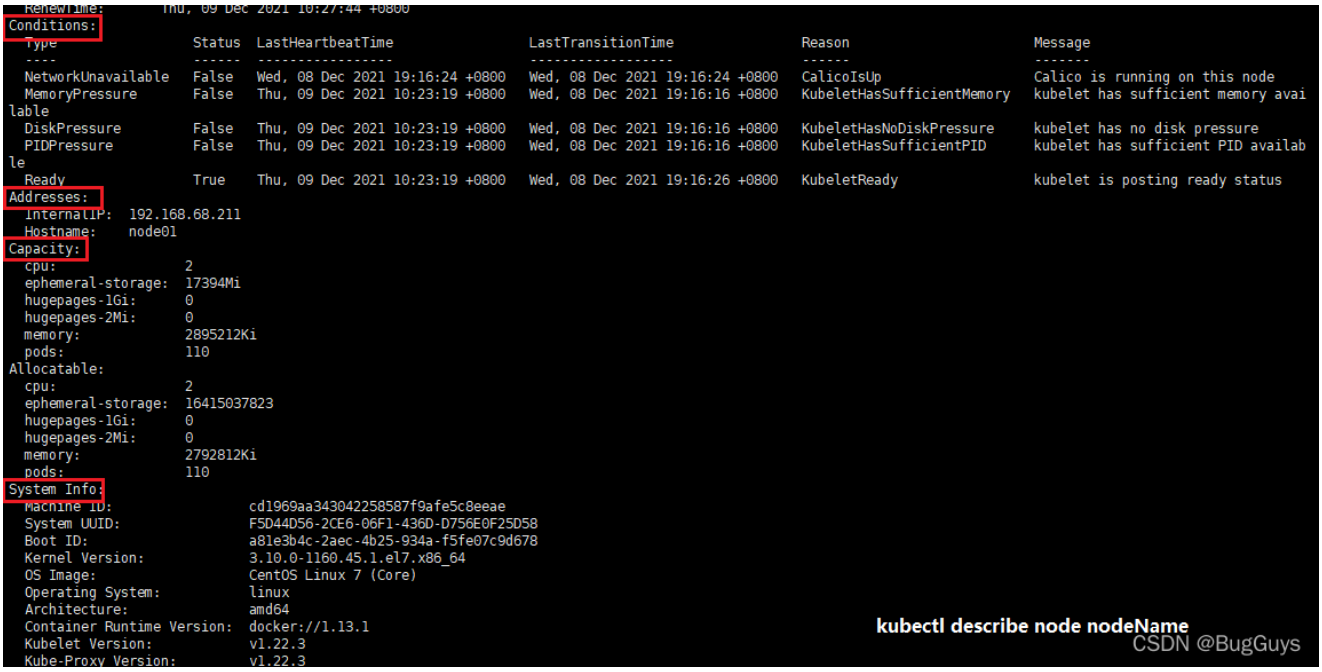
描述：

是K8S的集群工作节点，可以是物理机也可以是虚拟机；

Node状态：

- Address
  - HostName：可以被 kubelet 中的 `--hostname-override` 参数替代。
  - ExternalIP：可以被集群外部路由到的 IP 地址。
  - InternalIP：集群内部使用的 IP，集群外部无法访问。
- Condition
  - OutOfDisk：磁盘空间不足时为 `True`
  - Ready：Node controller 40 秒内没有收到 node 的状态报告为 `Unknown`，健康为 `True`，否则为 `False`。
  - MemoryPressure：当 node 有内存压力时为 `True`，否则为 `False`。
  - DiskPressure：当 node 有磁盘压力时为 `True`，否则为 `False`。

- Capacity
  - CPU
  - 内存
  - 可运行的最大 Pod 个数
- SystemInfo：节点的一些版本信息，如 OS、kubernetes、docker 等




```
# 查看节点信息
kubectl describe node NODENAME
# 禁止Pod调度到该节点上
kubectl corndon NODENAME
# 驱逐该节点上的所有 Pod
kubectl drain NODENAME
```

- 1
- 2
- 3
- 4
- 5
- 6

3.2 Namespace

描述：

简写ns。在集群中，我们可以使用namespace划分出多个“虚拟集群”，这些ns之间可以完全隔离，也可以跨ns，让一个ns中的service访问到其他ns的服务； 比如 Traefik ingress 和 kube-systemnamespace 下的 service 就可以为整个集群提供服务，这些都需要通过 RBAC 定义集群级别的角色来实现；

注意：

不是所有的资源对象都对应ns，其中`node`和`persistentVolume`就不属于任何ns；

3.3 Label

描述：

label是资源上的标识，用来对它们进行区分和选择；

特点：

1. 一个label会以kv形式附加到各种对象上；Node，Pod，Service
2. 一个资源对象可以定义人一多数量的Label，同一个Label也可以被添加到任意数量的资源对象上去；
3. label通常再资源对象确定时定义，也可以在资源创建后动态添加或删除；

可以通过label实现资源的多维度分组，以便灵活，方便的进行资源分配，调度，配置，部署等管理工作；

常用的标签如下：

> 版本标签：“version”:“release”,
>
> 环境标签：`"environment": "dev"`,`"environment": "qa"`,`"environment": "production"`
>
> 架构标签：`"tier": "frontend"` ,`"tier": "backend"`,`"tier": "cache"`

标签选择器：用于查询和筛选拥有某些标签的资源对象；

- 等式选择器 *equality-based*

  > 可以使用`=`、`==`、`!=`操作符，可以使用逗号分隔多个表达式

  name=slave:选择所有包含k=name，v=slave的对象

  env!=prod:选择所有包括k=env且v!=prod的对象

- 集合选择器 *set-based*

  > 可以使用`in`、`notin`、`!`操作符，另外还可以没有操作符，直接写出某个label的key，表示过滤有某个key的object而不管该key的value是何值，`!`表示没有该label的object

  name in (master, slave):选择所有包含k=name且v=master或者v=slave对象

  name not in (frontend):选择所有包含k=name且v不等于frontend的对象

标签的选择条件可以使用多个，此时将多个标签选择器进行组合，使用，进行分隔即可；

```
name=slave,env!=prod
name not in(frontend),env!=prod
```

- 1
- 2

Label key的组成：

- 不得超过63个字符
- 可以使用前缀，使用/分隔，前缀必须是DNS子域，不得超过253个字符，系统中的自动化组件创建的label必须指定前缀，`kubernetes.io/`由kubernetes保留
- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

Label value的组成：

- 不得超过63个字符
- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

3.4 污点和容忍

描述：

Taint（污点）和 Toleration（容忍）可以作用于 node 和 pod 上，其目的是优化 pod 在集群间的调度，这跟节点亲和性类似，只不过它们作用的方式相反，具有 taint 的 node 和 pod 是互斥关系，而具有节点亲和性关系的 node 和 pod 是相吸的。另外还有可以给 node 节点设置 label，通过给 pod 设置 nodeSelector 将 pod 调度到具有匹配标签的节点上。

Taint 和 toleration 相互配合，可以用来避免 pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个 taint ，这表示对于那些不能容忍这些 taint 的 pod，是不会被该节点接受的。如果将 toleration 应用于 pod 上，则表示这些 pod 可以（但不要求）被调度到具有相应 taint 的节点上。

污点

通过node上添加污点属性，来决定是否允许Pod调度过来；

Node被设置上污点之后，就和Pod之间存在了一种相斥的关系，进而拒绝Pod调度过来，甚至可以将已存在的Pod驱逐出去；

污点格式：`key=value:effect`，key和value是污点的标签，effect描述污点的作用，支持如下三个选项：

- PreferNoSchedule: k8s尽量不把pod调度到该污点Node上，除非没有其他节点可调度；
- NoSchedule: k8s将不会把Pod调度到该污点Node上，已存在的Pod将继续运行；
- NoExecute：k8s将不会把Pod调度到具有该污点的Node上，同时也会将Node上已存在的Pod驱逐；

命令：

```
# 设置污点
kubectl taint nodes node1 key=value:effect
# 去除污点
kubectl taint nodes node1 key:effect-
# 去除所有污点
kubectl taint nodes node1 key-

# 例如：
# 为node1设置污点
kubectl taint nodes node1 tag=test:PreferNoSchedule
# 为node1取消污点
kubectl taint nodes node1 tag:PreferNoSchedule-
# 为node1删除去除污点
kubectl taint nodes node1 tag-
# 查看node1 污点
kubectl describe node node1
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

容忍

若将pod调度到一个有污点的node上去，需要用到容忍；

> 污点就是拒绝，容忍就是忽略，Node通过污点拒绝pod调度上去，pod通过容忍忽略拒绝；

pod的配置

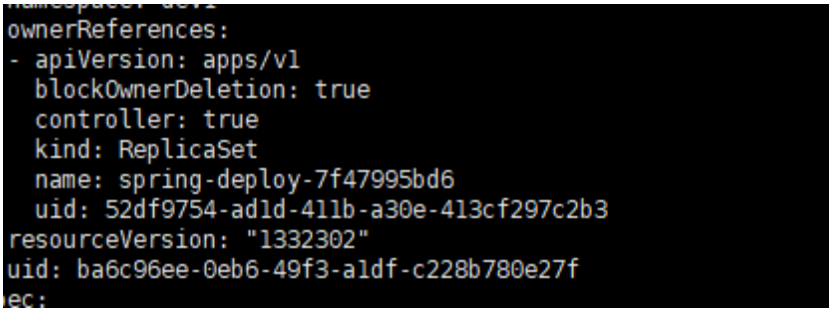
```
spec:
        containers:
        - name: nginx

        tolerations: # 添加容忍
        - key: "tag" # 对应要容忍的污点的键，空着意味匹配所有的键
          operator: "Equal" # 操作符，支持Equal和Exists(默认)
          value: "tests" # 容忍的污点的值
          effect: "NoExecute" # 添加容忍规则，要个污点规则一致
          tolerationSeconds: # 容忍时间, 当effect为NoExecute时生效，表示pod在Node上的停留时
间
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

3.5 垃圾回收

描述：

对于 ReplicaSet、StatefulSet、DaemonSet、Deployment、Job、 CronJob 等创建或管理的对象，会自动为其设置 ownerReferences字段的值；K8s的垃圾回收机制需要依赖于ownerReferences；他标记了从属对象的所有者；



目前有两大类GC：

- 级联删除：所有者删除，从属对象也被删除；

  分为前台级联删除Foreground和后台级联删除Background模式；

  - Foreground前台级联删除：所有从属对象删除完毕后删除所有者对象；当所有者删除时会进入“正在删除”状态，仍可以通过RestApi查询到当前对象；
  - Background后台级联删除： 立即删除所有者的对象，并由垃圾回收器在后台删除其从属对象 ； 用等待删除从属对象的时间；

- 孤儿删除orphan：所有者删除，从属对象变成孤儿；

  直接删除所有者对象，并将从属对象中的 ownerReference 元数据设置为默认值。之后垃圾回收器会确定孤儿对象并将其删除；

垃圾回收器的工作流程：

由Scanner、Garbage Processor 和 Propagator 组成；

- Scanner：它会检测 K8s 集群中支持的所有资源，并通过控制循环周期性地检测。它会扫描系统中的所有资源，并将每个对象添加到"脏队列"（dirty queue）中。
- Garbage Processor：它由在"脏队列"上工作的 worker 组成。每个 worker 都会从"脏队列"中取出对象，并检查该对象里的 OwnerReference 字段是否为空。如果为空，那就从“脏队列”中取出下一个对象进行处理；如果不为空，它会检测 OwnerReference 字段内的 owner resoure object 是否存在，如果不存在，会请求 API 服务器删除该对象。
- Propagator ：用于优化垃圾回收器，它包含以下三个组件：
  - EventQueue：负责存储 k8s 中资源对象的事件；
  - DAG（有向无环图）：负责存储 k8s 中所有资源对象的 owner-dependent 关系；
  - Worker：从 EventQueue 中取出资源对象的事件，并根据事件的类型会采取操作;

  在有了 Propagator 的加入之后，我们完全可以仅在 GC 开始运行的时候，让 Scanner 扫描系统中所有的对象，然后将这些信息传递给 Propagator 和“脏队列”。只要 DAG 一建立起来之后，那么 Scanner 其实就没有再工作的必要了。

总结

从 Kubernetes 的系统架构、技术概念和设计理念，我们可以看到 Kubernetes 系统最核心的两个设计理念：一个是 **容错性**，一个是 **易扩展性**。容错性实际是保证 Kubernetes 系统稳定性和安全性的基础，易扩展性是保证 Kubernetes 对变更友好，可以快速迭代增加新功能的基础。

功能

- 具备完善的集群管理能力；
- 透明的服务注册和服务发现机制；
- 内建负载均衡，故障发现和自我修复；
- 服务滚动升级和在线扩容；
- 可扩展的自动调度机制；
- 资源配额管理；