# 1概述

K8S是一个管理容器化的工作负载和服务的开源平台，具有可移植、可扩展等特点，当前生态系统正在迅速扩大。谷歌于2014年开源Kubernetes工程。

特点：

* 容器平台
* 微服务平台
* 可移植的云平台

K8S提供了一个以容器为中心的管理环境。它对用户的工作负载涉及的计算、网络和存储的基础设施进行编排，通过IaaS的灵活性提供了PaaS层的易用性。

特定应用的工作流可以被流化从而加载开发效率。最初可接受的临时编排常常需要大量的自动化设置。因此K8S也被设计成一个平台用于构建组件和工具的生态系统，使部署、缩放和应用管理变得更简单。

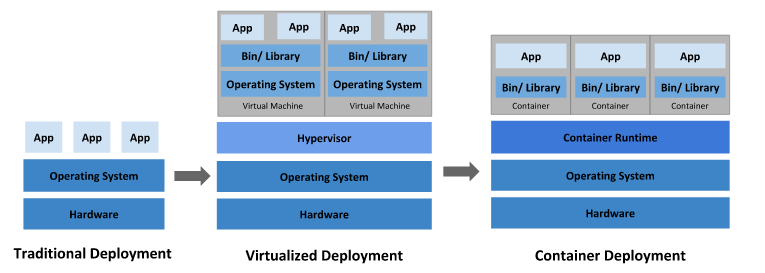
Labels可以让用户自由组织他们的资源。Annotations可以让用户添加额外特性信息来支持他们的工作流，让管理工具更容易知道检查点状态。另外K8S Controller Plane是构造在一套API之上的。用户可以定制API开发自己的控制器，比如调度器，使用命令行工具调用自己的API。

K8S不是传统意义上所有特性都包括的PaaS。由于K8S在容器级别操作，而不是硬件级别操作，它提供了一般可用的类似PaaS的特性，比如部署、缩放、负载均衡、日志、监控。但是K8S不是独立的，这些默认的解决方法是可选的、可插拔的。K8S提供了构建块来构建开发平台，但是支持用户自主选择。

* 不限制支持的应用类型。K8S目标是支持大量各种类型的工作负载，包括有状态的、无状态的和数据处理的工作负载。如果应用能在容器中运行，就能很好的在K8S中运行。
* 不部署源码，不构建应用。CI&CD工作流由组织文化和技术要求进行决定。
* 不提供应用级服务。比如中间件、数据处理框架、数据库、集群存储系统（Ceph）可以在K8S中运行。
* 不主宰日志、监控或者告警。它提供一些机制来进行收集和导出度量。
* 不提供也不要求一种配置语言。它提供一种声明式API。
* 不提供也不采用任何机器配置、维护、管理和自我修复系统。

另外K8S不是一个纯粹的编排系统。事实上，它消除了编排的需要。编排技术执行一个已定义的工作流：首先做A，然后做B，然后做C。相反，K8S是由一些独立的、可组合的控制进程的集合，持续地运行当前状态直到期望的状态。它不关心怎样从状态A到状态C。中心化的控制也不需要。这使得系统更易于使用，功能更强大，更具弹性，并且可扩展。

从应用部署时代的演变历史就可以明白K8S流行的原因。传统到虚拟，再到容器化。



容器时尤其是容器的进程生命周期由基础架构管理而不是被容器内部的管理进程隐藏起来。

使用容器的好处：

* 敏捷应用创建和部署。相比较使用虚拟机镜像，容器镜像的创建更容器和更高效。
* 持续开发，集成和部署。提供可靠和频繁容器镜像快速构建和部署以及回退（由于镜像不可变）
* 开发和部署分离。在构建/发布时创建应用容器，而不是在发布时，因此从基础架构中解耦应用。
* 透明性。不仅仅提供系统级信息和机制，也提供应用健康检查和其他信号。
* 开发、测试、生产环境一致性：在便携机上和在云环境上运行是一样的。
* 云上和操作系统可移植：支持Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine,以及其他环境。
* 应用中心化管理：提升了抽象级别，虚拟硬件上运行操作系统到使用逻辑资源在操作系统上运行应用。
* 解耦、分布、弹性和拆分微服务：应用被拆分更小、独立的片，能够被动态部署和管理，而不是在单节点机器上运行巨大的栈应用。
* 资源隔离：可预期的应用性能。
* 资源利用率：高效和细粒度。

容器是一种好方式来绑定和运行应用。在生产环境下，需要管理应用容器和保证应用不会宕机。例如如果一个容器挂掉，另一个容器需要重启。过去由系统处理这些不是很简单。

K8S就是拯救者。K8S提供框架运行分布式弹性系统，它关注扩展需求、挂机、部署模式和其他方面，比如容易管理大量应用部署。

K8S支持以下特性：

* 服务发现和负载均衡

K8S能够通过DNS或者使用他们自己IP暴露服务。如果进入某个容器的流量变高，K8S能够负载均衡和分发网络流量来保证部署稳定。

* 存储编排

K8S允许依据你的选择来挂载系统，比如本地存储，公有云等。

* 自动升级和回滚

可以描述部署的容器的期望状态，K8S能够按控制速率改变当前状态到期望状态。例如可以按部署自动创建新容器，移除已经存在的容器并释放占用的资源给新容器。

* 自动资源绑定

允许为每个容器设置CPU 、内存（RAM）。当容器设置请求资源，K8S能更好做决定来管理容器资源。

* 自身健康检查

K8S重启失败的容器，替换容器，杀死对自身健康检查失败的容器，直到容器准备就绪再通知客户端。

* 秘钥和配置管理

K8S可以存储和管理敏感信息，比如密码、OAuth tokens、SSH秘钥。可以部署和更新秘钥和应用配置不需要重启容器镜像，并且不会再栈信息中暴露秘钥。

## 组件

### Master组件

Master组件提供集群控制面板。Master组件决定集群的全局策略，比如调度和识别并响应集群事件（比如根据副本控制器中指定的副本数启动新的Pod）。

Master组件可以在集群中任何机器上运行。然而为了简化，典型应用是安装脚本在同一个机器上运行所有的Master组件而不是在这些机器上面运行用户容器。

* kube-apiserver

kube-apiserver组件在master节点上运行，暴露了K8S API。它是K8S控制面板的前端，被设计成通过部署多实例实现水平缩放。

* etcd

etcd作为K8S所有集群数据的后台存储，是一个持久化、高可用的键值对存储。如果K8S集群使用etcd作为后端存储，确保对集群数据有一个备份方案。

* kube-scheduler

kube-scheduler组件监控那个新创建的还没有被分配node的pod，然后选择一个node作为pod的宿主。决定调度的因素需要考虑独立和集中资源要求、硬件、软件和策略约束、亲和性、反亲和性、本地数据、工作负载间干扰和界限。

* kube-controller-manager

kube-controller-manager组件运行逻辑控制器，每个控制器是独立的进程，为了减少复杂性，他们都被编译成一个独立的二进制，运行独立进程。

包括如下控制器：

1）节点控制器：当节点宕机时负责通知和作出响应。

2）副本控制器：负责控制系统中每个副本控制器中设置的Pod副本数。

3）端点控制器：添加Endpoints对象数据，也就是用于关联Service和Pod。

4）Service Account和Token控制器：创建默认的账户和API访问token来访问新命名空间。

* cloud-controller-manager

cloud-controller-manager运行与底层云厂商相交互的控制器。在K8S 1.6版本中属于alpha特性。cloud-controller-manager只运行云产商的特定控制器循环。你必须在kube-controller-manager中禁用这些控制器循环。可以在启动kube-controller-manager时通过加参数--cloud-provider=external来禁用控制器环。

cloud-controller-manager允许云供应商的代码和Kubernetes代码彼此独立地发展。在之前发布版本中，在功能方面K8S核心代码是独立于云供应商代码。在未来版本中，关于云供应商的特定代码由云供应商维护并在K8S中与cloud-controller-manager相关联。以下控制器有云供应商依赖：

* Node Controller：检查云供应商，来识别节点是否已经在停止响应后被删除。
* Route Controller：设置底层云架构的路由
* Service Controller：创建、更新、删除云供应商的负载均衡
* Volume Controller：创建、关联、挂载卷，与云供应商交互来编排卷存储。

### Node组件

Node组件在每个node节点上运行，维护运行的pod和提供K8S运行环境。

* kubelet

kubelet在集群每个node节点上作为代理。它确保容器在pod中运行。kubelet掌控一些pod的特征，这些特征提供各种机制并确保容器按照描述的特征运行和健康检查。kubelet不会管理非K8S创建的容器。

* kube-proxy

kube-proxy是一个网络代码，在集群的每个node上运行。通过维护主机上的网络规则并执行连接转发来实现Kubernetes服务抽象。kube-proxy负责请求转发，通过后端函数实现TCP和UDP流量转发、循环TCP和UDP转发。

* 容器运行环境

容器运行环境是运行容器的软件。K8S支持多种容器运行环境：Docker\containerd\cri-o\rklet以及任何K8S CRI接口的实现软件。

### Addons

Addons是实现集群特性的pod和Services。这些pod被Deployment、ReplicationController等控制。addon对象在kube-system命名空间中创建。以下是可选的Addon,还有更多可选插件。

* DNS

当其他插件不是严格必须的，但是所有K8S集群应该有集群DNS，因为许多例子依赖于它。集群DNS是一个DNS服务，就像环境中其他DNS服务一样，它是为K8S服务提供DNS记录。K8S启动的容器会在DNS搜索中自动包含此DNS服务器。

* Web UI(Dashboard)

Dashboard是基于web的用户界面来查看K8S集群。允许用户管理和查看错误的运行在集群上的应用，同时也查看集群自身状况。

* Container Resource Monitoring

Container Resource Monitoring在中心数据库中记录关于容器的时间序列度量值，并提供UI来浏览数据。

* 集群级别的日志

## K8S API

API 详情见https://kubernetes.io/docs/concepts/overview/kubernetes-api/

K8S API是系统声明式配置架构的基础。kubectl命令行常常用来创建、更新、删除和查询。K8S使用一些列API资源来存储序列化状态。K8S自身也被分解成多个与API交互的组件。

以经验而言，任何成功的系统需要随使用场景或者存在的某种变化来成长和改变。因此我们期望K8S API可以持续改变和成长。但是在一段较长的时间内我们不打算打破与现有客户端的兼容性。一般来说，新API资源和新的字段可以期望被频繁添加。资源或字段的清除需要遵循API弃用策略。

### API版本

为了更容易的消除字段、资源结构的表示，K8S支持多版本API，分别在不同的API路径，例如/api/v1或者/apis/extensions/v1beat1。我们选择在API级别而不是在资源或字段级别进行版本控制，以确保API提供清晰，一致的系统资源和行为视图，并支持控制对生命周期或实验API的访问。一个JSON或者ProtoBuf序列化模型支持同样的模型改变指南，两种格式在下面都有描述。注意API版本和软件版本不直接相关。API和发布版本控制提议描述了API版本控制和软件版本控制之间的关系。不同API版本意味着不同级别的稳定和支持。

1）Alpha级别

>版本名称包含alpha，例如v1alpha1。

>可以有bug。允许特性存在bug，默认禁止。

>支持特性在任何时间没有通知的情况可以被删除。

> API可能会在以后的软件版本中以不兼容的方式更改，恕不另行通知。

>由于错误风险增加和缺乏长期支持，建议仅在短期测试集群中使用

2）Beta级别

>版本名称包含beta，例如v2beta3

>代码被较好的测试，特性是安全的，默认是允许的。

>整个特性不会删除，但是详情可能改变。

>对象模型或者语义在后续的beta或者稳定发布可能以不兼容的方式改变。发生这种情况时，我们将提供迁移到下一版本的说明。 这可能需要删除，编辑和重新创建API对象。 编辑过程可能需要一些思考。 对于依赖该功能的应用程序，这可能需要应用停机。

>建议仅用于非关键业务用途，因为后续版本中可能存在不兼容的更改。如果您有多个可以独立升级的群集，您可以放宽此限制。

3）stable级别

>版本名称包含vx,其中x是一个整数。

>许多后续版本的已发布软件中将出现稳定版本的功能。

### API 组

为了更容易的扩展K8S API，我们实现API组。API组在REST路径中指定和在序列化对象域apiVersion中指定。

当前有一些API组在使用中：

1）核心组通常在REST路径中/api/v1，域用apiVersion: v1。

2）命名组的使用方式为/apis/$GROUP\_NAME/$VERSION和apiVersion: $GROUP\_NAME/$VERSION，例如apiVersion: batch/v1

有两种方式使用定制资源来扩展API：

1）CustomResourceDefinition来让用户使用基本的CRUD操作。

2）需要全套Kubernetes API语义的用户可以实现自己的apiserver并使用聚合器使其无缝对接客户端。

### 启动API组

某个资源和API组默认是开启的。它们可以通过设置apiserver参数--runtime-config来开始或者关闭。--runtime-config接受逗号分隔的值。禁用batch/v1，可以设置--runtime-config=batch/v1=false，启用batch/v2alpha1，可以设置--runtime-config=batch/v2alpha1。这个参数可以接受逗号分隔的多个键值对。特别注意的是，修改--runtime-config来启用或者禁止组或资源需要重启apiserver和controller-manager才可以生效。

### 启动组资源

DaemonSets，Deployment,HorizontalPodAutoscalers,Ingress,Jobs,ReplicaSets默认启用。其他扩展资源可以通过设置--runtime-config参数。比如禁用deployment和ingress：

--runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/ingresses=false

## K8S对象

本章节解释K8S对象用K8S API怎样表示和怎样在yaml文件中定义K8S对象。

K8S对象是K8S系统中持久化实例，K8S用这些实例来标示当前集群状态。特别地，它们可以这样描述：

1）什么容器化的应用正在运行以及在哪个node上

2）那些应用的可用资源有哪些

3）围绕这些应用的策略，比如重启、更新、容错

一个K8S对象是一种意图记录，一旦你创建对象，K8S系统将会持续工作确保对象存在。通过创建一个对象，有效告诉K8S系统你想要集群工作负载达到什么状态，也是集群期望的状态。

关于K8S对象的操作涉及是否创建、修改和删除，需要使用K8S API。当使用kubectl命令行接口，CLI会调用K8S API。可以在自己的程序中使用二进制客户端直接调用K8S API。

### 了解对象

**对象特性和状态**

每种K8S对象包含两个嵌套的控制对象配置的域：对象特性和对象status。特性是必须设置的，它描述对象期望的状态-对象拥护的特性。status描述对象的确切状态，K8S系统会使用和更新状态。在任意时刻，K8S控制面板积极管理对象的状态来匹配设置的期望状态。

例如K8S Deployment可以代表运行集群中的应用。当创建Deployment时设置应用副本为3。K8S系统读取Deployment的特征并启动3个实例-更新状态来匹配设置的特征。如果实例中任何一个失败，K8S系统通过进行修正来响应特征和状态之间的差异-在这种情况下，启动一个副本实例。

**描述对象**

当你在K8S中创建对象，必须提供对象的spec来描述期望的状态，也包括对象的一些基本信息。当你使用K8S API创建对象（或者kubectl），API请求参数必须在请求体中以json格式包含基本信息。大部分情况下使用以yaml文件作为参数传给kubectl 。kubectl转换json发送API请求。

**nginx-deployment.yaml**

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 2 # tells deployment to run 2 pods matching the template

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

kubectl创建Deployment的方式kubectl apply -f nginx-deployment.yaml --record

一般输出结果为"deployment.apps/nginx-deployment created"。

**必要的域**

在yaml文件中定义对象需要的域：

apiVersion: 使用什么API版本来创建K8S对象

kind: 准备创建什么类型的对象

metadata: 元数据帮助唯一区分对象，比如名称，UID和可选的命名空间。

另外需要提供对象的特性域，每个对象的特性域不同，包含的域嵌套也不同。

### 对象管理

kubectl CLI支持多种不同方式创建和管理K8S对象。本章节做简单概述。

**管理技术**

提示：K8S对象应该仅仅被一种技术管理。混合和匹配多种技术在同样的对象上面导致无法识别的行为。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **管理技术** | **操作对象** | **推荐环境** | **支持写的数量** | **学习曲线** |
| 命令行 | 运行对象 | 开发 | 1+ | 低 |
| 命令式对象配置 | 单个文件 | 生产 | 1 | 中 |
| 声明式对象配置 | 文件目录 | 生产 | 1+ | 高 |

**命令行**

当使用命令行时，用户直接操作集群中的运行对象。用户提供操作数给kubectl命令作为参数或者标识。这是最简单的方式在集群中来启动或者运行一次性任务。因为这种技术直接操作运行对象，它不提供之前配置的历史记录。

创建一个Deployment对象来运行一个nginx容器实例。

**kubectl run nginx --image nginx**

另一种方式

**kubectl create deployment nginx --image nginx**

权衡

对比对象配置的优势：

>命令行简单易学，容易记忆

>命令只需要一个步骤即可对集群进行更改。

对比对象配置的劣势：

>命令行不可以和之前处理过程进行交互

>命令不提供与更改关联的审计跟踪。

>除了什么是运行的以外，命令行不提供记录源。

>命令行不提供用于创建新对象的模板

**命令式对象配置**

在命令式对象配置中，kubectl指定操作数（创建、替换等），可选的参数和至少一个文件。指定的文件必须以yaml或者json格式包含完整的对象定义。

注意：命令replace会用新的spec替换已经存在的spec，删除配置文件中丢失对象的所有更改。如果资源类型的spec是独立于配置文件进行更新的，那么replace命令不应该在这种资源类型上使用。例如LoadBalancer的externalIPs域独立于群集的配置进行更新。

例如

kubectl create -f nginx.yaml

删除两个配置文件中定义的对象

kubectl delete -f nginx.yaml -f redis.yaml

kubectl replace -f nginx.yaml

权衡

**对比命令行的优势：**

>对象配置可以存储在诸如Git的源控制系统中

>对象配置可以与进程集成，例如在推送和审计跟踪之前查看更改

>对象配置提供了用于创建新对象的模板。

**对比命令行的劣势：**

>对象配置需要对对象模式有基本的了解。

>对象配置需要编写YAML文件的附加步骤。

**对比声明式对象配置的优势：**

>命令式对象配置行为更简单，更易于理解。

>从Kubernetes 1.5版开始，命令式对象配置更加成熟。

**对比声明式对象配置的劣势：**

>命令对象配置最适合文件，而不是目录。

>活动对象的更新必须反映在配置文件中，否则在下次更换时会丢失

**声明式对象配置**

使用声明式对象配置时，用户对本地目录中的对象配置文件进行操作，但是用户不定义要对文件执行的操作。 kubectl会自动检测每个对象的创建，更新和删除操作。 这使得能够处理目录，其中可能需要不同对象的不同操作。

注意：声明式对象配置保留其他编写者所做的更改，即使更改未合并到对象配置文件。 这可以通过使用patch API操作来仅写入识别到的差异，而不是使用replace API操作来替换整个对象配置。

处理在configs目录下的所有对象配置文件，创建或者补丁运行的对象。可以先diff查看将要生成的改变，然后在apply。

**kubectl diff -f configs/**

**kubectl apply -f configs/**

递归处理目录：

**kubectl diff -R -f configs/**

**kubectl apply -R -f configs/**

权衡

**对比命令式对象配置的优势：**

>即使它们未合并回配置文件，也会保留直接对运行的对象所做的更改。

>声明式对象配置更好地支持对目录进行操作并自动检测每个对象的操作类型（创建，修补，删除）

**对比命令式对象配置的劣势：**

>声明式对象配置更难以调试，并在意外时难以理解结果

>使用diffs进行部分更新可创建复杂的合并和打补丁操作

### 命名

所有在K8S API中的对象可以被名称和UID唯一识别。对用户指定的非唯一属性，K8S提供标签和注解。

1）名称

客户端设置用于在资源URL中指定对象的字符串，例如/ api/v1/pods/some-name，给定类型的一个对象一次只可以有一个给定的名称。但是，如果删除该对象，则可以创建具有相同名称的新对象。

为了方面，K8S对象的名称最长253个字符，可包括小写数字字母，连接符合点，但是某些资源有严格的限制。

apiVersion: v1

kind: Pod

metadata:

name: nginx-demo

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

2）UID

K8S系统在对象的生命周期内生成的唯一Id来区别对象。

### 命名空间

Kubernetes支持由同一物理集群构建多个虚拟集群， 这些虚拟集群称为名称空间。

**何时使用命名空间**

命名空间旨在用于多个用户分布在多个团队或项目中的场景。对于具有几个到几十个用户的集群，您根本不需要创建或考虑名称空间。

命名空间的特性：

>命名空间提供名称范围。资源名称在同一名称空间中必须是唯一的，而跨名称空间中可以有同名。命名空间不能彼此嵌套，并且每个Kubernetes资源只能位于一个命名空间中

>命名空间是一种在多个用户之间划分群集资源的方法（通过资源配额）。在Kubernetes的未来版本中，默认情况下，同一名称空间中的对象将具有相同的访问控制策略。

>没有必要使用多个名称空间来分隔略有不同的资源，例如同一软件的不同版本：使用标签来区分同一名称空间中的资源。

**使用命名空间**

apiVersion: v1

kind: Namespace

metadata:

name: <insert-namespace-name-here>

创建命名空间kubectl create -f ./my-namespace.yaml

查看命名空间 kubectl get namespace

NAME STATUS AGE

default Active 1d

kube-system Active 1d

kube-public Active 1d

K8S创建3个默认的命名空间：

default：为没有指定命名空间的对象设此命名空间。

kube-system: 该命名空间分配给K8S系统创建的对象。

kube-public: 此命名空间是自动创建的，并且所有用户（包括未经过身份验证的用户）都可以读取。 此命名空间主要用于群集使用，为了某些资源在整个群集中可见且可公开读取。 此命名空间的公共方面只是一个约定，而不是一个要求。

当前请求设置命名空间：

kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>

kubectl get pods --namespace=<insert-namespace-name-here>

当前上下文设置命名空间：

您可以在该上下文中为所有后续kubectl命令永久保存命名空间

kubectl config set-context --current --namespace=<insert-namespace-name-here>

kubectl config view | grep namespace:

**命名空间和DNS**

当你创建Service时，同时创建一个对应的DNS实例。实例的结构是

<service-name>.<namespace-name>.svc.cluster.local,

这意味着如果一个容器访问服务<service-name>，将解析出命名空间中的服务。这对于在多个名称空间（如开发，分段和生产）中使用相同的配置非常有用。 如果要跨命名空间访问，则需要使用完全限定的域名（FQDN）

**不是所有对象都在命名空间中**

大多数Kubernetes资源（例如，pod，服务，副本控制器等）都在某些名称空间中。 但是，命名空间资源本身并不在命名空间中。 并且低级资源（例如节点和persistentVolumes）不在任何名称空间中。可通过命令查看：

kubectl api-resources --namespaced=true

kubectl api-resources --namespaced=false

### 标签和选择器

标签是附加到对象（例如Pod）上的键/值对。标签旨在用于指定对用户有意义且相关的对象的标识属性，但不直接暗示核心系统的语义。 标签可用于组织和选择对象的子集。标签可以在创建时附加到对象，后续也可以随时添加和修改。 每个对象都可以定义一组键/值标签。每个Key对于给定对象必须是唯一的。

"metadata": {

"labels": {

"key1" : "value1",

"key2" : "value2"

}

}

标签支持高效的查询和监视，非常适合在UI和CLI中使用。非识别信息应使用注释记录。

**用途**

标签使用户能够以松散耦合的方式将他们自己的组织结构映射到系统对象，而无需客户端存储这些映射。服务部署和批处理流水线通常是多维实体（例如，多个分区或多部署，多个发布通道，多层级，每层多个微服务）。 管理通常需要交叉操作，这打破了严格的多层封装，特别是由基础设施而不是用户确定的严格的层次结构。一些标签例子：

* "release" : "stable", "release" : "canary"
* "environment" : "dev", "environment" : "qa", "environment" : "production"
* "tier" : "frontend", "tier" : "backend", "tier" : "cache"
* "partition" : "customerA", "partition" : "customerB"
* "track" : "daily", "track" : "weekly"

记住对象的标签key必须唯一。

**语法和字符集**

标签是键值对，有效的标签键有两部分语义：一个可选的前缀和名称，中间用/分隔。名称是必要的，最大63位，以字母数字字符（[a-z0-9A-Z]）开头和结尾，直接带有破折号（ - ），下划线（\_），点（.）和字母数字。前缀是可选的，如果设置了必须是DNS子域格式：一系列由点（.）分隔的DNS标签，总共不超过253个字符，后跟斜杠（/）。

如果省略前缀，则假定标签Key对用户是私有的。向最终用户对象添加标签的自动系统组件（例如kube-scheduler，kube-controller-manager，kube-apiserver，kubectl或其他第三方自动化组件）必须指定前缀。K8S核心组件预留标签前缀包括kubernetes.io/和k8s.io/。

有效标签值必须为不超过63个字符，可以不设标签。格式为开头和结尾为字母数字，中间可以破折号（ - ），下划线（\_），点（.）和字母数。

apiVersion: v1

kind: Pod

metadata:

name: label-demo

labels:

environment: production

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

**标签选择器**

区别于名称和UID，标签不唯一。一般来说，期望更多的对象携带同样的标签。

通过标签选择器，客户端或者用户能识别对象集合。标签选择器是Kubernetes中的核心分组语法。API目前支持两种类型的选择器：基于等式和基于集合。 标签选择器可以由逗号分隔的多个条件组成。在多个条件的情况下，必须满足所有要求，因此逗号分隔符充当逻辑AND（&&）运算符。

空或非指定选择器的语义取决于上下文，使用选择器的API资源类型应说明它们的有效性和含义。

注意：对于某些API资源类型（例如ReplicaSet），两个实例的标签选择器不能在命名空间内重叠，或者控制器可以将其视为冲突的指令，并且无法确定应该存在多少副本。

1）基于等式条件

基于等式或不等式的标签选择器允许按标签键值对进行过滤。匹配对象的必须满足所有指定的标签约束，它们也可能有其他标签。 允许三种运算符=，==，!=。 前两个都代码等式，而后者代表不等式。

environment = production

tier != frontend

前者选择标签中存在key=environment，value=production的所有资源对象。后者选择标签中存在key=tier，value!= fronten的所有资源对象。可以选择标签中带有production而不带frontend的资源，使用方式是逗号分隔：environment=production,tier!=frontend

基于等式条件的标签选择另一种场景是POD指定节点。

apiVersion: v1

kind: Pod

metadata:

name: cuda-test

spec:

containers:

- name: cuda-test

image: "k8s.gcr.io/cuda-vector-add:v0.1"

resources:

limits:

nvidia.com/gpu: 1

nodeSelector:

accelerator: nvidia-tesla-p100

2）基于集合条件

基于集合的标签选择器允许根据一组值过滤键。支持三种运算符：in，notin和exists（仅支持key）。例如

environment in (production, qa)

tier notin (frontend, backend)

partition

!partition

第一个示例选择key= environment和value=production||qa的所有资源。

第二个示例选择所有key=tier但value!= frontend, backend的资源，以及所有没有带key=tier的标签的资源。

第三个例子选择所有资源，包括带有key= partition的标签，不考虑value。

第四个示例选择所有资源，包括不带有key= partition的标签，不考虑value。 类似地，逗号分隔符充当AND运算符。 因此partition，environment notin（qa）可以过滤标签带有key=partition（无论值为什么）且key= environment但value!=qa的资源对象。 基于集合的标签选择器可以进行等价转换。

比如environment=production与 environment in （production）。

还可以与等式选择器混合使用partition in (customerA, customerB),environment!=qa

仅供参考：

selector:

matchLabels:

app: myweb

matchExpressions:

- {key: tier, operator: In, values: [frontend]}

- {key: environment, operator: NotIn, values: [dev]}

**API**

1）列举和监视

LIST和WATCH操作可以指定标签选择器，使用查询参数来过滤对象集。 如果允许这两个要求（在此在URL查询字符串中显示指定）。

基于等式条件:

?labelSelector= URLEncoder.encode("environment=production,tier=frontend","UTF-8")

基于集合条件：

?labelSelector=

URLEncoder.encode ("environment in (production,qa),tier in (frontend)","UTF\_8")

标签选择器格式也可以通过REST客户端来列举和监视资源对象。例如使用kubectl调用apiserver

基于等式

kubectl get pods -l environment=production,tier=frontend

基于集合

kubectl get pods -l 'environment in (production),tier in (frontend)'

基于OR或操作

kubectl get pods -l 'environment in (production,qa)'

通过存在运算符限制否匹配

kubectl get pods -l 'environment,environment notin (frontend)'

2）在API对象中设置引用

一些K8S对象比如services和replicationcontrollers使用标签选择器来设置其他资源，比如pods。

Service和ReplicationController

与Service相关联的一组pod使用标签选择器进行定义。类似地，ReplicationController对管理的pod的数量也使用标签选择器来定义。

这两个对象的标签选择器都是在json或yaml文件中使用映射定义的，并且只支持基于等式的选择器：

"selector": {

"component" : "redis",

}

或者

selector:

component: redis

3）基于集合的资源对象

较新的资源对象比如Job、Deployment、ReplicaSet、DaemonSet,也支持集合选择器。

selector:

matchLabels:

component: redis

matchExpressions:

- {key: tier, operator: In, values: [cache]}

- {key: environment, operator: NotIn, values: [dev]}

其中上式中matchLabels和matchExpression是And操作，必须全部满条件。

matchExpressions支持的操作包括In, NotIn, Exists, 和DoesNotExist。

4）选择node集合

选择标签的一个应用场景是约束pod可以调度的节点集

### 注解

可以使用Kubernetes注解annotation将任意非标识元数据附加到对象。工具和库等客户端可以检索此元数据。

**附加元数据**

您可以使用标签或注释将元数据附加到K8S对象。标签可用于选择对象和查找满足特定条件的对象集合。相反，注释不用于识别和选择对象。注释中的元数据可以是小的或大的，结构化的或非结构化的，并且可以包括标签不允许的字符。

注解类似标签，也是键值对形式：

"metadata": {

"annotations": {

"key1" : "value1",

"key2" : "value2"

}

}

以下是可以在注释中记录的一些信息示例：

* 声明式配置管理的字段。将这些字段作为注释附加，可以将它们与客户端或服务器设置的默认值以及自动生成的字段和自动调整大小或自动调整系统设置的字段区分开来。
* 构建，发布或镜像信息，如时间戳，版本ID，git分支，PR编号，镜像哈希值和注册表地址。
* 标识日志记录，监视，分析或审核存储库
* 可用于调试目的的客户端库或工具信息：例如名称，版本和构建信息。
* 用户或工具/系统出处信息，例如来自其他生态系统组件的相关对象的URL。
* 轻量发布工具元数据：例如，配置或检查点。
* 负责人的电话或寻呼机号码，或指定可在何处找到该信息的目录条目，例如团队网站
* 实现来自终端用户的指令，用于修改行为或使用非标准功能。

您可以将此类信息存储在外部数据库或目录中，而不是使用注释，但这会使生成用于部署，管理，自检等共享客户端库和工具变得更加困难。

**语法和字符集**

注解是键值对，有效的标签键有两部分语义：一个可选的前缀和名称，中间用/分隔。名称是必要的，最大63位，以字母数字字符（[a-z0-9A-Z]）开头和结尾，直接带有破折号（ - ），下划线（\_），点（.）和字母数字。前缀是可选的，如果设置了必须是DNS子域格式：一系列由点（.）分隔的DNS标签，总共不超过253个字符，后跟斜杠（/）。

如果省略前缀，则假定标签Key对用户是私有的。向最终用户对象添加标签的自动系统组件（例如kube-scheduler，kube-controller-manager，kube-apiserver，kubectl或其他第三方自动化组件）必须指定前缀。K8S核心组件预留标签前缀包括kubernetes.io/和k8s.io/。

pod添加注解，指明镜像仓库地址imageregistry。

apiVersion: v1

kind: Pod

metadata:

name: annotations-demo

annotations:

imageregistry: "https://hub.docker.com/"

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

### 字段选择器

字段选择器可以根据一个域或者多个域的值来选择K8S资源。例如

metadata.name=my-service

metadata.namespace!=default

status.phase=Pending

kubectl get pods --field-selector status.phase=Running

注意: 字段选择器本质上是资源过滤器。 默认情况下不用选择器/过滤器，这意味着将选择指定类型的所有资源。以下两种方式是等价的。

kubectl get pods

kubectl get pods --field-selector ""

**支持的字段**

支持的字段选择器因Kubernetes资源类型而异。所有资源类型都支持metadata.name和metadata.namespace字段。使用不受支持的字段选择器会产生错误。

kubectl get ingress --field-selector foo.bar=baz

Error from server (BadRequest): Unable to find {"extensions" "v1beta1" "ingresses"} that match label selector "", field selector "foo.bar=baz": "foo.bar" is not a known field selector: only "metadata.name", "metadata.namespace"

**支持的操作**

操作运算符支持=或==，!=

例如查找所有命名空间中services除了default命名空间

kubectl get services --all-namespaces --field-selector metadata.namespace!=default

**链式选择器**

与label以及其他选择器一样，字段选择器可以用逗号连接起来。

kubectl get pods --field-selector=status.phase!=Running,spec.restartPolicy=Always

**多资源类型**

您可以跨多种资源类型使用字段选择器。

kubectl get statefulsets,services --all-namespaces --field-selector metadata.namespace!=default

### 推荐标签

您可以使用比kubectl和dashboard更多的工具来可视化和管理Kubernetes对象。 一组通用的标签允许工具以互操作的方式工作，以所有工具都能理解的通用方式描述对象。

除支持工具外，一些推荐的标签还以可查询的方式描述应用程序。

元数据描述应用程序的概念。Kubernetes不是一个服务平台（PaaS），也没有或强制应用的形式。应用程序是非正式的并使用元数据进描述， 应用程序包含的内容的定义是松散的。

注意：推荐的标签使得管理应用更容易但对一些核心工具不是必须的。

共享标签和注释共享一个共同的前缀：app.kubernetes.io。 没有前缀的标签对用户是私有的。 共享前缀可确保共享标签不会干扰自定义用户标签

**标签**

为了充分利用这些标签，应将它们应用于每个资源对象。

|  |  |  |  |
| --- | --- | --- | --- |
| Key | 描述 | 例子 | 类型 |
| app.kubernetes.io/name | 应用名称 | Mysql | 字符串 |
| app.kubernetes.io/instance | 唯一应用实例名称 | Wordpress-abcxyz | 字符串 |
| app.kubernetes.io/version | 当前应用版本，可以版本号，哈希值等 | 5.7.21 | 字符串 |
| app.kubernetes.io/component | 在架构中组件角色 | Database | 字符串 |
| app.kubernetes.io/part-of | 更高级别应用名称 | Wordpress | 字符串 |
| app.kubernetes.io/managed-by | 应用管理工具 | Helm | 字符串 |

apiVersion: apps/v1

kind: StatefulSet

metadata:

labels:

app.kubernetes.io/name: mysql

app.kubernetes.io/instance: wordpress-abcxzy

app.kubernetes.io/version: "5.7.21"

app.kubernetes.io/component: database

app.kubernetes.io/part-of: wordpress

app.kubernetes.io/managed-by: helm

**应用和应用实例**

应用程序可以一次或多次安装到Kubernetes集群中，在某些情况下，可以安装在同一名称空间中。 例如，wordpress可以不止一次安装，其中不同的网站是wordpress的不同安装

应用程序的名称和实例名称分别记录。这使得应用程序的应用程序和实例可以识别。 应用程序的每个实例都必须具有唯一的名称

**例子**

1）一个简单的stateless service

Deployment常用来监视应用自身。

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app.kubernetes.io/name: myservice

app.kubernetes.io/instance: myservice-abcxzy

...

service常用来暴露服务

apiVersion: v1

kind: Service

metadata:

labels:

app.kubernetes.io/name: myservice

app.kubernetes.io/instance: myservice-abcxzy

...

2）带数据库的web应用

wordpress 应用的 Deployment

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app.kubernetes.io/name: wordpress

app.kubernetes.io/instance: wordpress-abcxzy

app.kubernetes.io/version: "4.9.4"

app.kubernetes.io/managed-by: helm

app.kubernetes.io/component: server

app.kubernetes.io/part-of: wordpress

wordpress 应用的 Service

apiVersion: v1

kind: Service

metadata:

labels:

app.kubernetes.io/name: wordpress

app.kubernetes.io/instance: wordpress-abcxzy

app.kubernetes.io/version: "4.9.4"

app.kubernetes.io/managed-by: helm

app.kubernetes.io/component: server

app.kubernetes.io/part-of: wordpress

...

wordpress的数据库

apiVersion: apps/v1

kind: StatefulSet

metadata:

labels:

app.kubernetes.io/name: mysql

app.kubernetes.io/instance: mysql-abcxzy

app.kubernetes.io/version: "5.7.21"

app.kubernetes.io/managed-by: helm

app.kubernetes.io/component: database

app.kubernetes.io/part-of: wordpress

...

暴露mysql服务作为wordpress的部分：

apiVersion: v1

kind: Service

metadata:

labels:

app.kubernetes.io/name: mysql

app.kubernetes.io/instance: mysql-abcxzy

app.kubernetes.io/version: "5.7.21"

app.kubernetes.io/managed-by: helm

app.kubernetes.io/component: database

app.kubernetes.io/part-of: wordpress

...

# 2 K8S架构

## 2.1 Nodes

Node是Kubernetes中的工作机器，以前称minion小兵。节点可以是VM或物理机，具体取决于集群。每个节点都包含运行pod所需的服务，并由Master组件管理。Node上的服务包括容器运行时，kubelet和kube-proxy。

**Node状态**

节点的状态包括如下信息：

* 地址
* 状态
* 容量和可分配
* 信息

kubectl describe node <node name>

1. 地址

如下这些字段的使用取决于您的云提供商或裸机配置。

HostName: 节点内核报告的主机名 可以通过kubelet的参数--hostname-override参数覆盖。

ExternalIP：通常是可外部路由的节点的IP地址（可从群集外部获得）

InternalIP：通常是仅在群集内可路由的节点的IP地址

1. 状态

字段conditions描述所有Running节点的状态。例如包括：

|  |  |
| --- | --- |
| Node状态 | 描述 |
| OutOfDisk | True如果节点上的可用空间不足以添加新的pod时为False |
| Ready | 如果节点正常且准备接受pod，则为True;如果节点不健康且不接受pod，则为False;如果节点控制器在最后一个时间段内node-monitor-grace-period中没有从节点收到消息，则为Unknown（默认为 40秒） |
| MemoryPressure | 如果节点内存存在压力，则为True；如果节点内存压力为低则为False |
| PIDPressure | 如果进程运行存在压力，也就是说节点上有太多进程则为True ; 否则为False |
| DiskPressure | 如果磁盘大小存在压力即磁盘容量低则为True;否则为False |
| NetworkUnavailable | 如果未正确配置节点的网络则为True，否则为False |

节点的状态以JSON格式表示，如下：

"conditions": [

{

"lastHeartbeatTime": "2019-07-09T06:54:22Z",

"lastTransitionTime": "2019-07-08T11:28:29Z",

"message": "kubelet has sufficient disk space available",

"reason": "KubeletHasSufficientDisk",

"status": "False",

"type": "OutOfDisk"

},

{

"lastHeartbeatTime": "2019-07-09T06:54:22Z",

"lastTransitionTime": "2019-07-08T11:28:29Z",

"message": "kubelet has sufficient memory available",

"reason": "KubeletHasSufficientMemory",

"status": "False",

"type": "MemoryPressure"

},

{

"lastHeartbeatTime": "2019-07-09T06:54:22Z",

"lastTransitionTime": "2019-07-08T11:28:29Z",

"message": "kubelet has no disk pressure",

"reason": "KubeletHasNoDiskPressure",

"status": "False",

"type": "DiskPressure"

},

{

"lastHeartbeatTime": "2019-07-09T06:54:22Z",

"lastTransitionTime": "2019-07-08T11:28:29Z",

"message": "kubelet has sufficient PID available",

"reason": "KubeletHasSufficientPID",

"status": "False",

"type": "PIDPressure"

},

{

"lastHeartbeatTime": "2019-07-09T06:54:22Z",

"lastTransitionTime": "2019-07-08T12:05:02Z",

"message": "kubelet is posting ready status",

"reason": "KubeletReady",

"status": "True",

"type": "Ready"

}

]

如果Ready的状态保持Unknown或者False的时长超过了参数pod-eviction-timeout（kube-controller-manager启动参数）的值，所有节点上的pod被节点控制器进行调度删除。默认的驱逐时间段是5分钟。在某些情况下，当节点无法访问时，apiserver无法与节点上的kubelet通信。在重新建立与apiserver的通信之前，不能将删除pod的决定传送到kubelet。同时，计划删除的pod可以继续在分区节点上运行。

在1.5之前的Kubernetes版本中，节点控制器会强制从apiserver中删除这些无法访问的pod。 但是，在1.5及更高版本中，节点控制器不会强制删除pod，直到确认它们已停止在群集中运行。 可以看到pod可能在无法访问的节点上运行，处于“Terminating”或“Unknown”状态。如果Kubernetes无法从底层基础架构推断出节点已经永久离开群集，则群集管理员可能需要手动删除节点对象。 从Kubernetes中删除节点对象会导致节点上运行的所有Pod对象从apiserver中删除，并释放它们的名称。

在版本1.12中，TaintNodesByCondition功能被提升为beta版，因此节点生命周期控制器会自动创建表示状态的污点。 类似地，调度程序在考虑节点时忽略状态; 对应地它会查看Node的污点和Pod的容忍度。

现在用户可以在旧的调度模型和更灵活的新调度模型之间进行选择。在旧模式下被调度的pod没有任何容忍度。但是新模式下可以在该节点上调度容忍特定节点污点的Pod。

小心：启用此功能会在状态查看和污点创建之间产生一个小延迟。 此延迟通常小于一秒，但它可以增加成功调度但被kubelet拒绝的Pod的数量。

1. 容量和可分配

描述节点上可用的资源：CPU，内存以及可以在节点上调度的最大pod数。

容量块中的字段表示节点具有的资源总量。可分配块指示节点上可被Pod消耗的资源量。

1. 信息

有关节点的一般信息，例如内核版本，Kubernetes版本（kubelet和kube-proxy版本），Docker版本（如果使用的话），操作系统名称。信息由Kubelet从节点收集

**管理**

与pod和service不同，Kubernetes本身并不创建节点：节点由Google Compute Engine等云供应商在外部创建，或者节点存在于物理或虚拟机池中。 因此，当Kubernetes创建节点时，它会创建一个表示节点的对象。 创建后，Kubernetes会检查节点是否有效。

如果按下面内容创建Node

{

"kind": "Node",

"apiVersion": "v1",

"metadata": {

"name": "10.240.79.157",

"labels": {

"name": "my-first-k8s-node"

}

}

}

Kubernetes在内部创建节点对象（表示节点），并通过基于metadata.name字段的检查节点运行状况。如果节点有效 - 即如果所有必需的服务都在运行 - 它有资格运行pod。 否则，对于任何群集活动，它将被忽略，直到它变为有效的节点。

注意：Kubernetes保留无效节点的对象，并不断检查它是否有效。您必须显式删除Node对象才能停止此过程。

目前，有三个组件与Kubernetes节点接口交互：node-controller，kubelet和kubectl。

**1） Node Controller**

节点控制器是Kubernetes主组件，它管理节点的各个方面。节点控制器在节点的生命周期中具有多个角色。

第一种是在注册时为节点分配CIDR块（如果打开了CIDR分配）。

第二种方法是使节点控制器的内部节点列表与云供供商的可用机器列表保持同步。在云环境中运行时，只要节点不健康，节点控制器就会询问云提供商该节点的VM是否仍然可用。 如果不是，则节点控制器从其节点列表中删除该节点。

第三是监测节点的健康状况。节点控制器负责在节点变得无法访问时将NodeStatus的NodeReady条件更新为ConditionUnknown（即节点控制器因某种原因停止接收心跳，例如由于节点关闭），然后逐出驱逐节点中的所有pod （如果节点仍然无法访问，则使用正常终止）。 （默认超时为40秒，开始报告ConditionUnknown，之后5分支开始驱逐pod。）节点控制器每隔--node-monitor-period秒检查每个节点的状态。

在1.13之前的Kubernetes版本中，NodeStatus是节点的心跳。 从Kubernetes 1.13开始，节点租用功能作为alpha功能引入。启用节点租用功能后，每个节点在kube-node-lease命名空间中都有一个关联的Lease对象，该对象由节点定期更新，NodeStatus和节点租用都被视为来自节点的心跳。 Node Lease经常更新，而NodeStatus仅在发生一些改变或者过了超时时间（默认值为1分钟）从节点报告给主节点。 由于节点租约比NodeStatus轻量得多，因此从可伸缩性和性能角度来看，此功能使节点心跳检查显着更容易。

在Kubernetes 1.4中，我们更新了节点控制器的逻辑，以便在大量节点到达主站时遇到问题时更好地处理（例如因为主站有网络问题）。从1.4开始，节点控制器在决定pod驱逐时查看集群中所有节点的状态。

在大多数情况下，节点控制器将驱逐率限制为每秒--node-eviction-rate（默认值0.1），这意味着它不会每10秒从多个节点驱逐pod。

当给定可用zone中的节点变得不健康时，节点逐出行为会发生变化。节点控制器同时检查区域中节点的不健康百分比（NodeReady是ConditionUnknown或ConditionFalse时表示不健康）。如果不健康节点的比例小于--unhealthy-zone-threshold（默认值为0.55），那么驱逐率会降低：如果群集很小（即小于或等于--large-cluster-size-threshold， 默认为50）然后停止驱逐，否则驱逐率降低到--secondary-node-eviction-rate（默认值0.01）。 每个可用区域实施这些策略的原因是一个可用区域可能是主服务器分区的部分，同时其他可用区域也与主服务器保持连接。如果您的群集未跨越多个云提供商可用区域，则只有一个可用区域（整个群集）。

在可用区域之间传播节点的一个关键原因是当整个区域出现故障时，工作负载可以转移到健康区域。因此如果区域中的所有节点都不健康，则节点控制器以正常速率--node-eviction-rate驱逐。 特殊情况是所有区域完全不健康（即群集中没有健康的节点）。 在这种情况下，节点控制器假定主连接存在一些问题并在某些连接恢复之前停止所有驱逐。

从Kubernetes 1.6开始，当pod不能容忍taints时NodeController还负责驱逐在具有NoExecute 污点的节点上运行的pod。NodeController负责添加以及节点无法访问或未就绪等问题节点相对应的污点，注意这是一种默认禁用的alpha功能。

从版本1.8开始，节点控制器可以负责创建表示节点条件的污点，这是1.8版的alpha功能

**2）节点的自我注册**

当kubelet的参数--register-node为true（默认值）时，kubelet将尝试向API服务器注册自己。 这是大多数发行版使用的首选模式。

自我注册时，kubelet进行下面的操作

* --kubeconfig - 用于向apiserver验证自身的凭据的路径
* -cloud-provider - 如何告诉云供供商读取有关自身的元数据。
* --register-node - 是否自动注册API服务器。
* --register-with-taints - 使用注册的节点指定污点列表（逗号分隔<key> = <value>：<effect>）。 如果register-node为false，则忽略。
* --node-ip - 节点的IP地址。
* --node-labels - 在集群中注册节点时添加的标签。
* --node-status-update-frequency - 指定kubelet将节点状态发布到master的频率。

启用节点授权模式和NodeRestriction admission插件后，仅授权kubelet创建/修改自己的节点资源。

**手动节点管理**

集群管理员可以创建和修改节点对象。

如果管理员希望手动创建节点对象，请设置kubelet标志--register-node = false。

管理员可以修改节点资源（无论--register-node的设置如何）。 修改包括在节点上设置标签并将其标记为不可调度。节点上的标签可以与pod上的节点选择器结合使用以控制调度，例如， 将pod限制为仅有资格在节点的子集上运行。将节点标记为不可调度可防止将新pod调度到该节点，但不会影响节点上的任何现有pod。 这在节点重启等之前作为准备步骤很有用。例如要将节点标记为不可调度，请运行以下命令：

kubectl cordn $NODENAME

注意：由DaemonSet控制器创建的Pod绕过Kubernetes调度程序，不遵守节点上的不可调度属性。 这里假设守护进程属于机器，即使它在准备重新启动时正在耗尽应用程序。

1）节点容量

节点的容量（cpus的数量和内存）是节点对象的一部分。通常节点在创建节点对象时注册自己并报告其容量。 如果您正在进行手动节点管理，则需要在添加节点时设置节点容量。

Kubernetes调度程序确保节点上的所有pod都有足够的资源。 它检查节点上容器请求的总和不大于节点容量。 它包括由kubelet启动的所有容器，但不包括由容器运行时直接启动的容器，也不包括在容器外部运行的任何进程启动的容器。可以设置非pod进程的保留资源。

**API对象**

Node在K8S REST API中是一种顶级资源。

## 2.2 Master -Node通信

本文档对master（实际上是apiserver）和Kubernetes集群之间的通信路径进行了编目。 目的是允许用户自定义其安装以强化网络配置，以便群集可以在不受信任的网络上运行（或在云提供商上的完全公共IP上运行）。

### 2.2.1 集群到Master

从集群到master的所有通信路径都在apiserver处终止（其他主服务器组件均未设计为公开远程服务）。在典型部署中，apiserver被配置为在安全HTTPS端口（443）上侦听远程连接，其中启用了一种或多种形式的客户端认证。应启用一种或多种授权形式，尤其是在允许匿名请求或service account token。

应为节点配置群集的公共根证书，以便它们可以携带有效的客户端凭据安全地连接到apiserver。例如在默认GKE部署中，采用客户端证书的形式提供给kubelet的客户端凭证，比如kubelet TLS。

希望连接到apiserver的Pod可以通过利用service account安全地执行此操作，因为Kubernetes在实例化Pod时自动将公共根证书和有效的bearer token注入到pod中。 kubernetes服务（在所有名称空间中）配置了一个虚拟IP地址，该地址被重定向（通过kube-proxy）到apiserver上的HTTPS端点。

主组件也是通过安全端口与群集服务器通信。

因此，默认情况下，从群集（节点和节点上运行的pod）到apiserver的连接的默认操作模式是安全的，可以在不受信任和/或公共网络上运行。

### 2.2.2 Master到集群

从主服务器（apiserver）到集群有两条主要通信路径。 第一个是从apiserver到在集群中的每个节点上运行的kubelet进程。 第二种是通过apiserver的代理功能从apiserver到任何节点，pod或服务。

1）apiserver到kubelet

从apiserver到kubelet连接用于：

* 获取pod的日志。
* 附加（通过kubectl）到运行的pod。
* 提供kubelet的端口转发功能。

这些连接终止于kubelet的HTTPS端点。 默认情况下，apiserver不会验证kubelet的服务证书，这会使连接受到中间人攻击，并且不安全地运行在不受信任的和/或公共网络上。

要验证此连接，请使用--kubelet-certificate-authority参数为apiserver提供根证书包，以用于验证kubelet的服务证书。如果无法做到这一点，请在apiserver和kubelet之间使用SSH隧道，以避免连接不受信任或公共网络。

最后，应启用Kubelet身份验证和/或授权以保护kubelet API

2）apiserver到node，pod，service

从apiserver到节点，pod或service的连接默认为普通的HTTP连接，因此既未经过身份验证也未加密。它们可以通过在API URL加上前缀https进行安全连接，但它们不会验证HTTPS端点提供的证书，也不会提供客户端凭据，因此连接将被加密 ，它不会提供任何完整性保证。 这些连接目前在不受信任和/或公共网络上运行是不安全的。

3）SSH通道

Kubernetes支持SSH隧道来保护Master - > Cluster通信路径。 在此配置中，apiserver启动到集群中每个节点的SSH隧道（连接到侦听端口22的ssh服务器），并通过隧道传递发往kubelet，节点，pod或服务的所有流量。 此隧道可确保流量不会在运行节点的网络外部暴露。SSH隧道目前已被弃用，因此除非您知道自己在做什么，否则不应选择使用它们。 正在设计替代此通信渠道。

## 2.3 云控制管理器相关概念

pass

# 3 容器

## 3.1镜像

您创建Docker镜像并将其推送到仓库中，然后在Kubernetes pod中引用它。容器的image属性支持与docker命令相同的语法，包括私有仓库和tag。

### 3.1.1 更新镜像

默认的镜像更新策略是IfNotPresent，意味着如果镜像已经存在kubelet跳过拉取镜像。如果希望一直强制拉取镜像，可以进行如下操作：

* 设容器imagePullPolicy为Always
* 忽略imagePullPolicy并设置镜像tag为:latest（注意应该避免使用）。
* 忽略imagePullPolicy和镜像tag
* 开启AlwaysPullImages授权控制

### 3.1.2 用manifest构建多架构镜像

Docker CLI现在支持下面的命令行docker manifest和子命令例如create、annotate和push。这些命令可以用来构建和推送manifest。可以使用docker manifest inspect来查看manifest。

相关docker文档：

https://docs.docker.com/edge/engine/reference/commandline/manifest/

相关例子链接：

https://cs.k8s.io/?q=docker%20manifest%20(create%7Cpush%7Cannotate)&i=nope&files=&repos=

这些命令完全依赖于Docker CLI并且完全在Docker CLI上实现。 您需要编辑$ HOME / .docker /config.json并将experimental=enabled为启用或者您可以在调用CLI命令时将DOCKER\_CLI\_EXPERIMENTAL环境变量设置为enabled启用。

注意：请使用Docker 18.06或更高版本，低版本有错误或不支持实验命令行选项。示例https://github.com/docker/cli/issues/1135会在containerd下导致问题

如果您在上传旧的manifest时遇到问题，只需清理$ HOME/.docker / manifests中的旧清单即可重新开始。

对于Kubernetes，我们通常使用带后缀的镜像-$(ARCH)。为了向后兼容，请生成带有后缀的旧镜像。比如pause和pause-amd64

### 3.1.3使用私有仓库

私人仓库管理机构可能需要密钥才能从中读取图像。凭证可以通过多种方式提供：

* 使用Google Container Registry

>每个集群

>在Google Compute Engine或Google Kubernetes Engine上自动配置

>所有pod都可以读取项目的私有仓库

* 使用AWS EC2容器注册表（ECR）

>使用IAM角色和策略来控制对ECR仓库的访问

>自动刷新ECR登录凭据

* 使用Oracle云基础架构注册表（OCIR）

>使用IAM角色和策略来控制对OCIR仓库的访问

* 使用Azure容器注册表（ACR）
* 使用IBM Cloud Container Registry
* 配置节点来验证私有注册表

>所有pod都可以读取任何已配置的私有注册表

>需要集群管理员进行节点配置

* 预拉取镜像

>所有pod都可以使用节点上缓存的任何镜像

>需要root权限才能设置所有节点

* 在Pod上指定ImagePullSecrets

>只有提供自己密钥的pod才能访问私有注册表

如果在云供应商虚机上拉取镜像可以看详细配置，见官网。

https://kubernetes.io/docs/concepts/containers/images/#building-multi-architecture-images-with-manifests

1）配置节点以验证私有仓库

注意：如果您在Google Kubernetes Engine上运行，则每个节点上都会有一个.dockercfg，其中包含Google Container Registry的凭据。 你不能使用这种方法。

注意：如果您在AWS EC2上运行并且正在使用EC2容器注册表（ECR），则每个节点上的kubelet将管理和更新ECR登录凭据。 你不能使用这种方法。

注意：如果您可以控制节点配置，则此方法适用。 它不能可靠地在GCE和任何其他进行自动节点替换的云提供商上运行。

注意：Kubernetes截至目前仅支持docker配置的auths和HttpHeaders部分。 这意味着不支持凭证助手（credHelpers或credsStore）。

Docker将私有注册表的密钥存储在$ HOME / .dockercfg或$ HOME / .docker / config.json文件中。 如果您将相同的文件放在下面的搜索路径列表中，则kubelet会在拉取镜像时将其用作凭据。

{--root-dir:-/var/lib/kubelet}/config.json

{cwd of kubelet}/config.json

${HOME}/.docker/config.json

/.docker/config.json

{--root-dir:-/var/lib/kubelet}/.dockercfg

{cwd of kubelet}/.dockercfg

${HOME}/.dockercfg

/.dockercfg

注意：您可能必须在环境文件中为kubelet明确设置HOME = / root。

下面是推荐步骤来配置你的节点使用私有仓库。

1.在每个希望使用凭证的节点上运行docker login [server]，该操作会更新$HOME/.docker/config.json。

2. 在编辑器中查看$ HOME / .docker / config.json以确保它只包含您要使用的凭据。

3.列举所有节点，例如

3.1）如果想要节点名称

nodes= $(kubectl get nodes -o jsonpath='{range.items[\*].metadata}{.name} {end}')

3.2）如果想获取IPs

nodes=$(kubectl get nodes -o

jsonpath='{range .items[\*].status.addresses[?(@.type=="ExternalIP")]}{.address} {end}')

4. 复制本地.docker/config.json到上面列举的某个搜索路径下，例如

for n in $nodes; do scp ~/.docker/config.json root@$n:/var/lib/kubelet/config.json; done

验证创建使用私有镜像的pod

kubectl apply -f - <<EOF

apiVersion: v1

kind: Pod

metadata:

name: private-image-test-1

spec:

containers:

- name: uses-private-image

image: $PRIVATE\_IMAGE\_NAME

imagePullPolicy: Always

command: [ "echo", "SUCCESS" ]

EOF

如果一切顺利的话，可以查看pod

kubectl logs private-image-test-1

SUCCESS

如果失败可以看到：

kubectl describe pods/private-image-test-1 | grep "Failed"

Fri, 26 Jun 2015 15:36:13 -0700 Fri, 26 Jun 2015 15:39:13 -0700 19 {kubelet node-i2hq} spec.containers{uses-private-image} failed Failed to pull image "user/privaterepo:v1": Error: image user/privaterepo:v1 not found

您必须确保群集中的所有节点都具有相同的.docker / config.json。 否则，pod将在某些节点上运行，而无法在其他节点上运行。 例如，如果使用节点自动缩放，则每个实例模板都需要包含.docker / config.json或挂载包含它的驱动器。

将私有注册表项添加到.docker / config.json后，所有pod将对任何私有注册表中的映像具有读访问权限

2）预拉取镜像

注意：如果您在Google Kubernetes Engine上运行，则每个节点上都会有一个.dockercfg，其中包含Google Container Registry的凭据。 你不能使用这种方法。

注意：如果您可以控制节点配置，则此方法是合适的。 它不能可靠地在GCE和任何其他进行自动节点替换的云提供商上运行

默认情况下，kubelet将尝试从指定的注册表中提取每个镜像。但是如果容器的imagePullPolicy属性设置为IfNotPresent或Never，则使用本地镜像（相应地、优先、排他）。

如果您希望依赖预先提取的镜像作为仓库身份验证的替代，则必须确保群集中的所有节点都具有相同的预拉取镜像。这可以用于预加载某些镜像以提高速度，或者作为对私有注册表进行身份验证的替代方法。所有pod都可以读取任何预拉镜像

3）在Pod上指定镜像ImagePullSecrets

注意：目前这种方法是Google Kubernetes Engine，GCE以及自动创建节点的任何云提供商的推荐方法。

K8S支持在pod上私有仓库秘钥。

### 3.1.4 使用Docker配置创建Secret

运行下面的命令，替换大写的值

kubectl create secret docker-registry <name>

--docker-server=DOCKER\_REGISTRY\_SERVER

--docker-username=DOCKER\_USER

--docker-password=DOCKER\_PASSWORD

--docker-email=DOCKER\_EMAIL

如果您已经拥有Docker凭证文件，则可以将凭证文件作为Kubernetes密钥导入，而不是使用上述命令。如果您使用多个私有容器注册表，参照

https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/#registry-secret-existing-credentials这将特别有用，因为kubectl创建秘密docker-registry创建一个仅适用于单个私有注册表的Secret。

注意：Pod只能在自己的命名空间中引用镜像拉取secrets，因此每个命名空间需要执行一次此过程

### 3.1.5 在pod中引用一个imagePullSecrets

cat <<EOF > pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: foo

namespace: awesomeapps

spec:

containers:

- name: foo

image: janedoe/awesomeapp:v1

imagePullSecrets:

- name: myregistrykey

EOF

cat <<EOF >> ./kustomization.yaml

resources:

- pod.yaml

EOF

这需要在每个pod进行操作来使用私有仓库。但是，可以通过在serviceAccount资源中设置imagePullSecrets来自动设置此字段。 检查将ImagePullSecrets添加到serviceAccount以获取详情。您可以将其与每个节点的.docker / config.json结合使用。 凭证将被合并。 这种方法适用于Google Kubernetes Engine。

**使用案例**

有许多配置私有注册表的解决方案。 以下是一些常见用例和建议的解决方案。

1. 集群仅运行非专有（例如开源）镜像。无需隐藏镜像。

1.1-运行DockerHub上的公共镜像

无需配置；

在GCE / Google Kubernetes Engine上，自动使用本地镜像来提高速度和可用性。

2. 集群运行一些专有镜像，应该隐藏给公司外部的人，但对所有集群用户都可见。

2.1-使用托管的私有docker仓库

可以是一个Dockerhub上托管的仓库或者其他地方；

如上所述，在每个节点上手动配置.docker / config.json。

2.2-或者使用开放读取访问权限在防火墙后面运行的内部私有注册表。

无需K8S的配置

2.3-或者在GCE / Google Kubernetes Engine上时，请使用该项目的Google Container Registry。

与集群自动调节相比，它可以比手动节点配置更好地工作。

3. 具有专有镜像的集群，其中一些需要更严格的访问控制。

确保AlwaysPullImages准入控制器处于活动状态。 否则，所有Pod都可能访问所有镜像。

将敏感数据移动到Secret资源中，而不是将其打包在镜像中

4. 一个多租户群集，每个租户都需要拥有私有注册表

4.1-确保AlwaysPullImages准入控制器处于活动状态。 否则，所有租户的所有Pod都可能 访问所有镜像。

4.2运行需要授权的私有注册表。

4.3为每个租户生成注册表凭据secret，并为每个租户命名空间添加secret

4.4 租户将这个secret添加到每个命名空间的imagePullSecrets。

如果需要访问多个注册表，则可以为每个注册表创建一个secret。Kubelet会将任何imagePullSecrets合并到一个虚拟的.docker / config.json中

## 3.2 容器环境变量

https://kubernetes.io/docs/concepts/containers/container-environment-variables/

# 4 工作负载

# 5 服务、负载均衡、网络

# 6 存储

# 7 配置

# 8 安全

# 9 调度

# 10 策略

# 11 集群管理

# 12 拓展K8S