# 1简述

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。

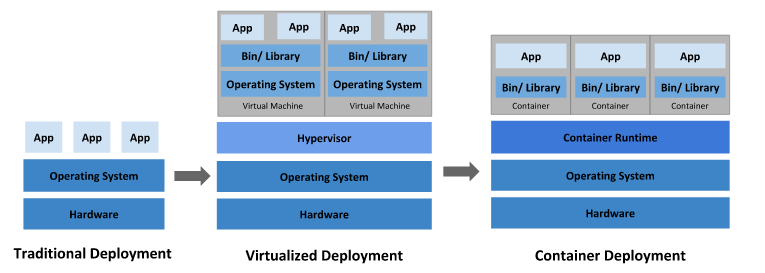
Labels可以让用户自由组织他们的资源。Annotations可以让用户添加额外特性信息来支持他们的工作流，让管理工具更容易知道检查点状态。另外K8S Controller Plane是构造在一套API之上的。用户可以定制API开发自己的控制器，比如调度器，使用命令行工具调用自己的API。

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。

# 2组件

## 2.1 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。

## 2.2 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。

## 2.3 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

# 3 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。

## 3.1 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。

## 3.2 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。

## 3.3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。

## 3.4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

# 4 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。

## 4.1 了解对象

### 4.1.1 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。

### 4.1.2 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"。

### 4.1.3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。

## 4.2 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。

### 4.2.1 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 |

### 4.2.2 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

### 4.2.3 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

### 4.2.3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

## 4.3 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。

## 4.4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。

### 4.4.1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。

### 4.4.2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:

### 4.4.3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）

### 4.4.4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

## 4.5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。

### 4.5.1 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。

### 4.5.2 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

### 4.5.3 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。

piVersion: 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]}

### 4.5.4 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

## 4.6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。

### 4.6.1 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。

### 4.6.2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

## 4.7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 ""

### 4.7.1 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"

### 4.7.2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

### 4.7.3 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

### 4.7.4 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

## 4.8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

### 4.8.1 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

### 4.8.2 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

### 4.8.3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

...

# 5 K8S架构

## 5.1 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。

### 5.1.1 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

### 5.1.2 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。

### 5.1.3 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。

### 5.1.4 API对象

Node在K8S REST API中是一种顶级资源。

## 5.2 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）。

### 5.2.1 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。

### 5.2.2 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。

## 5.3 云控制管理器相关概念

pass

# 6 容器

## 6.1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。

### 6.1.1 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

### 6.1.2 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。

### 6.1.3使用私有仓库

从私有仓库读取镜像可能需要秘钥。可以有多种认证方式：

* 使用Google 镜像仓库

每个集群

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

所有的Pod可以读取工程私有仓库

.....其他云厂商的私有仓库配置见官网。

* ....

### 6.1.4 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容器仓库（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供应商上运行。

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

Docker将私有仓库的密钥存储在$ 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程序凭据。

* {--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。

以下是配置节点以使用私有仓库的建议步骤。 在此示例中，在桌面/笔记本电脑上运行这些操作：

1）在每个想使用证书的节点运行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

如果想获取节点名称：

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

如果想获取节点ip：

nodes=$(kubectl get nodes -o

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

4）将本地.docker / config.json复制到上面的搜索路径列表之一

例如：

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

通过创建使用私有镜像的pod进行验证：

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

pod/private-image-test-1 created

如果一切正常

kubectl logs private-image-test-1

SUCCESS

如果失败

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。

将私有仓库项添加到.docker / config.json后，所有pod将对任何私有仓库中的镜像具有读权限。

6.1.5 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。

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，则使用本地映像（优先或排他或者对应）。

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，或者作为对私有仓库进行身份验证的替代方法。

所有pod都可以读取任何预拉取镜像

### 6.1.5 在pod中指定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。

**使用docker config创建secret**

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，则可以将凭证文件导入作为Kubernetes密钥，而不是使用上述命令。详情见官网链接。 如果您使用多个私有容器仓库，这特别有用，因为kubectl create secret docker-registry创建一个后只能使用于单个私有仓库。

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

**在pod中引用imagePullSecrets**

现在可以通过增加imagePullSecrets来创建引用secret的pod

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

需要对使用私有仓库的每个pod执行此操作。

但是，可以通过在serviceAccount资源中设置imagePullSecrets来自动设置此字段。详细介绍见官网。

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。

### 6.1.6使用案例

一些解决方式来配置私有仓库。这里有一些常见使用案例和建议。

1）集群运行非私有（即开源）镜像，无需隐藏这些镜像。

使用DockerHub上公共映像。

无需配置。

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

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

使用托管的私有Docker仓库。

它可能托管在Docker Hub或其他地方。

如上所述，在每个节点上手动配置.docker / config.json。

2.1）或者，使用开放读取访问权限在防火墙后面运行内部私有仓库。

不需要Kubernetes配置。

2.2）或者，在使用GCE / Google Kubernetes Engine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。

2.3）或者，在更改节点配置不方便的集群上，请使用imagePullSecrets

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）一个多租户集群，每个租户都需要拥有私有仓库。

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仓库。

4.3）为每个租户生成仓库凭据，secret，并为每个租户命名空间填充secret。

4.4）租户将这个秘密添加到每个命名空间的imagePullSecrets。

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

## 6.2 容器环境变量

此页面描述容器环境中容器可用的资源。

### 6.2.1 容器环境

K8S容器环境提供了许多重要资源给容器。

* 一个文件系统，是镜像和一个或多个卷的组合。
* 容器自身信息
* 集群中其他对象的信息

关于容器信息：

容器的主机名是运行容器的Pod的名称。 它可以通过hostname命令或libc中的gethostname函数调用获得。

Pod名称和命名空间可通过downward API作为环境变量使用。

Pod定义中的用户定义环境变量也可用于容器，Docker镜像中静态指定的任何环境变量也是如此。

关于集群信息：

创建Container时运行的所有服务的列表可作为环境变量用于该Container。 这些环境变量与Docker link的语法相匹配。

对于名为foo的映射到一个名为bar的容器的服务，以下变量被定义：

FOO\_SERVICE\_HOST = <运行服务的主机>

FOO\_SERVICE\_PORT = <运行服务的端口>

服务具有专用IP地址，如果启用了DNS插件，则可通过DNS使用容器。

## 6.3 运行时类

该特性在K8S v1.14处于beta阶段。

该页面描述了RuntimeClass资源和运行时选择机制。

警告：RuntimeClass在v1.14 beta升级中有重大更改。 如果您在v1.14之前使用RuntimeClass，请参阅将RuntimeClass从Alpha升级到Beta。

### 6.3.1 运行时类

RuntimeClass是用于选择容器运行时配置的功能。 容器运行时配置用于运行Pod中的容器。

### 6.3.2 动机

您可以在不同的Pod之间设置不同的RuntimeClass，以提供性能与安全性之间的平衡。 例如，如果您的部分工作负载需要高级别的信息安全保障，您可以选择安排这些Pod在使用硬件虚拟化的容器运行环境中运行。您将受益于代替的运行环境的额外隔离，但会牺牲一些额外的开销。

您还可以使用RuntimeClass运行具有相同容器环境但具有不同设置的不同Pod。

1）设置

确保RuntimeClass特性开启，默认是打开的。RuntimeClass特性必须在apiservers和kubelets侧打开。

1.1）在节点上配置CRI实现（取决于运行环境）

RuntimeClass提供的配置依赖于Container Runtime Interface（CRI）实现。 有关如何配置的CRI实现，请参阅相应的文档。

注意：RuntimeClass当前假定整个集群中的同构节点配置（这意味着所有节点的配置方式与容器环境相同）。 任何异构性（变化的配置）必须通过调度功能独立于RuntimeClass进行管理（请参阅将Pod分配给节点）。

配置具有相应的handler名称，由RuntimeClass引用。 handler必须是有效的DNS 1123标签（字母数字+ - 字符）

1.2）创建相应的RuntimeClass资源

步骤1中的配置设置应各自具有关联的handler名称，用于标识配置。 对于每个handler处理程序，创建相应的RuntimeClass对象。

RuntimeClass资源目前只有2个重要字段：RuntimeClass名称（metadata.name）和处理程序（handler）。 对象定义如下所示：

apiVersion: node.k8s.io/v1beta1 # RuntimeClass is defined in the node.k8s.io API group

kind: RuntimeClass

metadata:

name: myclass # The name the RuntimeClass will be referenced by

# RuntimeClass is a non-namespaced resource

handler: myconfiguration # The name of the corresponding CRI configuration

注意：建议将RuntimeClass写操作（create / update / patch / delete）限制为集群管理员。 这通常是默认值

2）使用

在集群中配置Runtimeclass后使用非常简单。在Pod的spec中指定runtimeclassName.例如：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

runtimeClassName: myclass

# ..

这将指示Kubelet使用命名的RuntimeClass来运行此pod。 如果指定的RuntimeClass不存在，或者CRI无法运行相应的handler，则pod将进入Failed终结阶段。

如果未指定runtimeClassName，则将使用默认的RuntimeHandler，这相当于禁用RuntimeClass功能。

3）CRI配置

dockershim

K8S内置dockershim CRI不支持运行时handler。

containerd

运行时处理程序通过/etc/containerd/config.toml上的containerd配置进行配置。

有效的handler配置在runtimes子路径下：

[plugins.cri.containerd.runtimes.${HANDLER\_NAME}]

cri-o

运行时handler通过cri-o的配置文件/etc/crio/crio.conf进行设置。有效的handler在[crio.runtime.runtimes.${HANDLER\_NAME}]

runtime\_path = "${PATH\_TO\_BINARY}"

4）更新RuntimeClass从Alpha到Beta

runtimeclass beta特性包括下面改变：

* node.k8s.io API组和runtimeclasses.node.k8s.io资源已从CustomResourceDefinition迁移到内置API。
* spec已在RuntimeClass定义中内联（即不再有RuntimeClassSpec）
* runtimeHandler字段已重命名为handler
* 现在所有API版本都需要handler字段。这意味着在Alpha API中也需要的runtimeHandler字段。
* handler字段必须是有效的DNS标签（RFC 1123），这意味着（在所有版本中）它不能再包含点(.) 字符。 有效handler匹配以下正则表达式：^ [a-z0-9]（[ - a-z0-9] \* [a-z0-9]）?$。

必要操作：从RuntimeClass功能的alpha版本升级到beta需要执行以下操作：

* 升级到v1.14后必须重新创建RuntimeClass资源，并且应手动删除runtimeclasses.node.k8s.io CRD：

kubectl delete customresourcedefinitions.apiextensions.k8s.io runtimeclasses.node.k8s.io

* Alpha RuntimeClasses具有未指定或空的runtimeHandler或使用点(.)的handler不再有效，必须迁移到有效的handler配置（参见上文）。

## 6.4 容器生命周期钩子

该页面描述了kubelet管理的容器如何使用Container生命周期钩子框架来运行在管理生命周期中由事件触发的代码。

### 6.4.1 概述

类似于许多具有组件生命周期钩子的编程语言框架，例如Angular，Kubernetes为Containers提供了生命周期钩子。 钩子使Container能够了解其管理生命周期中的事件，并在执行相应的生命周期钩子时运行在handler中实现的代码

### 6.4.2 容器钩子

有两种容器钩子：

PostStart

在创建容器后立即执行此挂钩。 但是无法保证挂钩将在容器ENTRYPOINT之前执行。 没有参数传递给handler程序。

prestop

由于API请求或管理事件（例如活动探测失败，抢占，资源争用等），在容器终止之前立即调用此钩子。 如果容器已处于已终止或已完成状态，则对preStop挂接的调用将失败。 它是阻塞的，意味着它是同步的，所以它必须在删除容器的调用之前完成。 没有参数传递给处理程序

1）钩子handker实现

容器可以通过实现和注册该钩子的handler程序来访问钩子。 可以为Container实现两种类型的钩子handler程序：

Exec - 在Container的cgroups和命名空间内执行特定命令，例如pre-stop.sh。 命令消耗的资源将计算在Container上。

HTTP - 对Container上的特定端点执行HTTP请求

2）钩子handler执行

当Container生命周期管理钩子被调用时，Kubernetes管理系统会在注册该钩子的Container中执行handler。

钩子handler回调在包含Container的Pod的上下文中是同步的。 这意味着对于PostStart钩子，Container ENTRYPOINT和钩子是异步激活。 但是，如果挂钩运行或挂起太长时间，则Container无法达到运行状态。

PreStop挂钩的行为类似。 如果挂钩在执行期间挂起，则Pod阶段将保持Terminating状态，并在终止的FinallyGracePeriodSeconds间隔时间达到后被终止。 如果PostStart或PreStop挂钩失败，则会终止Container。

用户应使其钩子handler尽可能轻量级。但是有些情况下，长时间运行的命令是有意义的，例如在停止Container之前保存状态。

3）挂钩传递保障

钩子传递至少是一次，这意味着对于任何给定的事件，例如对于PostStart或PreStop，可以多次调用钩子。 由钩子实现来正确处理。

通常，只进行单次传递。 例如，如果HTTP挂钩接收器关闭且无法获取流量，则不会尝试重新发送。 然而，在一些罕见的情况下，可能会发生双重递送。 例如，如果一个kubelet在发送一个钩子的过程中重新启动，那么在kubelet重新启动后可能会重新发送一个钩子。

4）调试钩子handler

在Pod事件中不公开钩子handler的日志。 如果handler由于某种原因失败，它会广播一个事件。 对于PostStart，这是FailedPostStartHook事件，对于PreStop，这是FailedPreStopHook事件。 您可以通过运行kubectl describe pod <pod\_name>来查看这些事件。 以下是运行此命令的一些事件输出示例：

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {default-scheduler } Normal Scheduled Successfully assigned test-1730497541-cq1d2 to gke-test-cluster-default-pool-a07e5d30-siqd

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Pulling pulling image "test:1.0"

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Created Created container with docker id 5c6a256a2567; Security:[seccomp=unconfined]

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Pulled Successfully pulled image "test:1.0"

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Started Started container with docker id 5c6a256a2567

38s 38s 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Killing Killing container with docker id 5c6a256a2567: PostStart handler: Error executing in Docker Container: 1

37s 37s 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Killing Killing container with docker id 8df9fdfd7054: PostStart handler: Error executing in Docker Container: 1

38s 37s 2 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} Warning FailedSync Error syncing pod, skipping: failed to "StartContainer" for "main" with RunContainerError: "PostStart handler: Error executing in Docker Container: 1"

1m 22s 2 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Warning FailedPostStartHook

# 7 工作负载

## 7.1 Pod

本节对Pod进行简介，pod是k8s对象模型中最小部署对象。

### 7.1.1 Pod概述

**7.1.1.1 理解pod**

Pod是Kubernetes应用程序的基本执行单元 - Kubernetes应用程序中您创建或部署的Kubernetes对象模型中最小和最简单的单元。 Pod表示在集群上运行的进程。

Pod封装了应用程序的容器（或者在某些情况下，多个容器），存储资源，唯一的网络IP以及管理容器应该如何运行的操作。 Pod表示部署单元：Kubernetes中的一个应用程序的单个实例，可能包含单个容器或少量紧密耦合且共享资源的容器。

Docker是Kubernetes Pod中最常用的容器运行环境，但Pods也支持其他容器运行环境。

Pod在K8S集群中主要以两种方式：

* **运行单个容器的Pod**

“one-container-per-Pod”模型是最常见的Kubernetes用例; 在这种情况下，您可以将Pod视为单个容器的包装，而Kubernetes直接管理Pod而不是容器。

* **运行多个需要紧密合作的容器的Pod**

Pod可以封装由多个相关容器组成的应用程序，这些容器紧密耦合并需要共享资源。这些共处一地的容器可能形成一个统一的服务单元 - 一个容器从共享卷向外部提供文件，而一个单独的“sidecar”容器刷新或更新这些文件。 Pod将这些容器和存储资源作为单个可管理实体包装在一起。 Kubernetes博客提供了有关Pod用例的一些其他信息。有关更多信息，请参阅：

分布式系统工具包：复合容器的模式

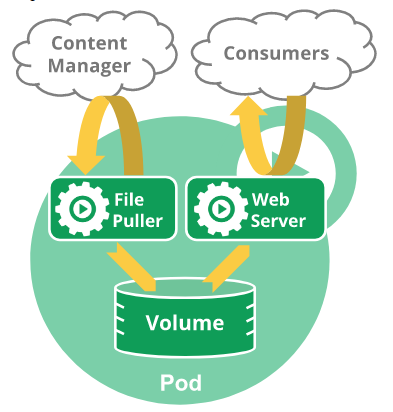
容器设计模式

每个Pod都用于运行给定应用程序的单个实例。 如果要水平扩展应用程序（例如，运行多个实例），则应使用多个Pod，每个实例一个。 在Kubernetes中，这通常被称为副本。副本Pod通常由称为Controller的抽象创建和管理。 有关更多信息，请参阅Pod和控制器。

1.pod如何管理多个容器

Pod旨在支持多个协作流程（例如容器），形成一个有凝聚力的服务单元。 Pod中的容器自动位于集群中的同一物理或虚拟机上，并共同调度。 容器可以共享资源和依赖关系，彼此通信，并协调它们何时以及如何终止。

请注意，在单个Pod中对多个共存和共同管理的容器进行分组是一个相对高级的用例。 您应该仅在容器紧密耦合的特定实例中使用此模式。 例如，您可能有一个容器充当共享卷中文件的Web服务器，以及一个单独的“sidecar”容器，用于从远程源更新这些文件，如下图所示：



一些Pod具有init容器和app容器。 Init容器在app容器启动之前运行并完成。

Pod为其组成容器提供两种共享资源：网络和存储

**网络**

每个Pod都分配有唯一的IP地址。 Pod中的每个容器都共享网络命名空间，包括IP地址和网络端口。Pod中的容器可以使用localhost相互通信。当Pod中的容器与Pod外部的实体通信时，它们必须协调它们如何使用共享网络资源（例如端口）。

**存储**

Pod可以指定一组共享存储卷。Pod中的所有容器都可以访问共享卷，从而允许这些容器共享数据。如果需要重新启动其中一个容器，则卷还允许Pod中的持久数据存活。

**7.1.1.2 运行Pod**

你很少直接在Kubernetes中创建单独的Pod-单例pod。 这是因为Pods被设计为相对短暂的一次性实体。 当创建Pod（由您直接创建或由Controller间接创建）时，它将被安排在集群中的节点上运行。 Pod保留在该节点上，直到进程终止，pod对象被删除，Pod由于资源不足而被驱逐，或者Node失败。

注意：不应将重新启动Pod和重新启动Pod中的容器混淆。Pod本身不运行，但是容器运行的环境会持续存在直到删除为止。

pod本身不能自我修复。 如果将Pod调度到失败的节点，或者调度操作本身失败，则删除Pod; 同样，由于缺乏资源或节点维护，Pod将无法在驱逐中存活。 Kubernetes使用更高级别的抽象，称为Controller，它处理管理相对可处理的Pod实例的工作。 因此，虽然可以直接使用Pod，但在Kubernetes中使用Controller管理pod更为常见。有关Kubernetes如何使用控制器实现Pod缩放和修复的更多信息，请参阅Pod和控制器。

**Pod和控制器**

Controller可以为您创建和管理多个Pod，处理复制和部署，并在集群范围内提供自我修复功能。 例如，如果节点发生故障，Controller可能会通过在不同节点上安排相同的替换来自动替换Pod。

包含一个或多个pod的控制器的一些示例包括：

Deployment

StatefulSet

DaemonSet

通常，控制器使用您提供的Pod模板来创建它负责的Pod

**Pod模板**

Pod模板是pod规范，包含在其他对象中，例如Replication Controllers，Jobs和DaemonSets。 控制器使用Pod模板制作实际的pod。下面的示例是Pod的简单清单，其中包含一个打印消息的容器。

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox

command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']

pod模板不是指定所有副本的当前所需状态，而是像cookie切割器。 切割cookie后，cookie与切割器无关。没有“量子纠缠”。对模板的后续更改甚至切换到新模板对已创建的pod没有直接影响。 类似地，随后可以直接更新由副本控制器创建的pod。某种意义上pod确实指定了属于pod的所有容器的当前所需状态。 这种方法从根本上简化了系统语义并增加了原语的灵活性

### 7.1.2 Pod

Pod是可以在Kubernetes中创建和管理的最小可部署计算单元。

**7.1.2.1 什么是Pod**

一个Pod是一个或多个容器（如Docker容器）的组，具有共享存储/网络，以及如何运行容器的特性。Pod的内容始终是共同定位和共同调度的，并在共享上下文中运行。 Pod模型特定于应用程序的“逻辑主机” - 它包含一个或多个相对紧密耦合的应用程序容器 - 在之前容器世界中，在相同的物理或虚拟机上执行将意味着在同一逻辑主机上执行。

虽然Kubernetes支持的容器运行环境不止于Docker，但Docker是最常见的运行时，它有助于用Docker术语描述Pods。

Pod的共享上下文是一组Linux命名空间，cgroup，以及可能的隔离方面-与Docker容器隔离的东西相同。在Pod的上下文中，各个应用程序可能会进一步的子隔离。

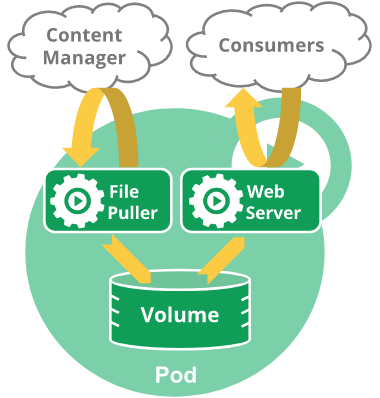
Pod中的容器共享IP地址和端口空间，并可通过localhost找到彼此。它们可以使用标准的进程间通信（如SystemV信号或POSIX共享内存）相互通信。不同Pod中的容器具有不同的IP地址，无需特殊配置不可通过IPC（进程间通信）进行通信。这些容器通常通过Pod IP地址相互通信。

Pod中的应用程序还可以访问共享卷，共享卷被定义为Pod的一部分，可以安装到每个应用程序的文件系统中。

就Docker构造器而言，Pod被建模为一组具有共享命名空间和共享文件系统卷的Docker容器。

与单个应用程序容器一样，Pod被认为是相对短暂的（而不是持久的）实体。正如pod生命周期中所讨论的，Pod被创建，被分配一个唯一的ID（UID），并被安排到它们到被接受的节点，直到（根据重启策略）或删除。如果节点终止，则在超时期限之后，将调度到该节点的Pod进行删除。给定的Pod（由UID定义）不会“重新调度”到新节点;相应地，它可以被一个相同的Pod替换，如果需要，甚至可以使用相同的名称，但是使用新的UID（有关更多详细信息，请参阅副本控制器）。

当某些东西被认为具有与Pod相同的生命周期时，例如卷，这意味着只要该Pod（具有该UID）存在就存在。如果由于任何原因删除了Pod，即使创建了相同的替换，相关的东西（例如卷）也会被销毁并重新创建。



一个多容器Pod，包含文件拉取程序和Web服务器，该服务器使用持久卷在容器之间共享存储。

**7.1.2.2 pod目标**

**管理**

Pod是多个合作进程模式的模型，形成了一个有凝聚力的服务单元。它们通过提供比其组成应用程序集更高级别的抽象来简化应用程序部署和管理。 Pod用作部署，水平扩展和副本的单元。Pod中的容器自动处理共置（共同调度），共享命运（例如终止），协调副本，资源共享和依赖关系管理

**资源共享和通信**

Pod可以实现其成员之间的数据共享和通信。

Pod中的应用程序都使用相同的网络命名空间（相同的IP和端口空间），因此可以相互“查找”并使用localhost进行通信。 因此，Pod中的应用程序必须协调它们对端口的使用。 每个Pod在共享网络空间平面中具有IP地址，该网络空间与网络中的其他物理计算机和Pod完全通信。

Pod中的容器将系统主机名视为与Pod的已配置name相同。 网络部分还有更多相关信息。

除了定义在Pod中运行的应用程序容器之外，Pod还指定了一组共享存储卷。 卷使数据能够在容器重新启动后继续存在，并在Pod中的应用程序之间共享

**7.1.2.3 使用Pod**

Pod可用于托管垂直集成的应用程序堆栈（例如LAMP），但其主要目的是支持共址，共同管理的帮助程序，例如：

内容管理系统，文件和数据加载器，本地缓存管理器等。

日志和检查点备份，压缩，旋转，快照等

数据变更观察者，日志跟踪，日志和监控适配器，活动发布者等。

代理，网桥和适配器

控制器，管理器，配置器和更新器

通常，单个Pod不用于运行同一应用程序的多个实例。

有关更长的说明，请参阅分布式系统工具包：复合容器的模式

**7.1.2.3考虑替代方案**

为什么不在单个容器（Docker）里运行多个程序？

* 透明度。 使Pod内的容器对基础结构可见，使基础结构能够为这些容器提供服务，例如进程管理和资源监视 这为用户提供了许多便利。
* 解耦软件依赖关系。 各个容器可以独立地进行版本化，重建和重新部署。 Kubernetes甚至有一天可能会支持单个容器的实时更新。
* 便于使用。 用户无需运行自己的流程管理器，担心信号和退出代码传播等。
* 效率。 由于基础设施承担更多责任，因此容器的重量可以更轻。

为什么不支持基于亲和力的容器协同调度？

这种方法可以提供协同定位，但不会提供Pod的大部分好处，例如资源共享，IPC，保证命运共享和简化管理。

**7.1.2.4 pod的持久性（或缺乏持久性）**

pod不应被视为持久实例。它们将无法在调度故障，节点故障或其他驱逐（例如由于缺乏资源）或节点维护的情况下存活。

通常，用户不需要直接创建Pod。他们应该几乎总是使用控制器，即使是单个实例，例如，Deployment。控制器提供集群范围的自我修复，以及复制和部署管理。像StatefulSet这样的控制器也可以为有状态Pod提供支持。

使用集合API作为主要的面向用户的原语在集群调度系统中相对常见，包括Borg，Marathon，Aurora和Tupperware。

Pod作为基础单元，以便于：

* 调度程序和控制器可插拔性
* 支持pod级操作，无需通过控制器API“代理”它们
* Pod生命周期与控制器生命周期的解耦，例如自举
* 控制器和服务的分离 - 端点控制器只是监视Pods
* 具有集群级功能和Kubelet级功能的清晰组合 - Kubelet实际上是“pod控制器”
* 高可用性应用程序，期望Pods在终止之前被替换，并且肯定在删除之前被替换，例如在计划驱逐或镜像预拉取的情况下。

**7.1.2.5 Pod终止**

因为Pod表示集群中节点上的正在运行的进程，所以允许这些进程在不再需要时正常终止（与使用KILL信号猛烈杀死并且没有机会清理）非常重要。 用户应该能够请求删除并知道进程何时终止，但也能够确保删除最终完成。 当用户请求删除Pod时，系统会在允许强制终止Pod之前记录预期的宽限期，并将TERM信号发送到每个容器中的主进程。 宽限期到期后，KILL信号将发送到这些进程，然后从API服务器中删除Pod。 如果在等待进程终止时重新启动Kubelet或容器管理器，则将在完整宽限期内重试终止

如下面例子：

1. 用户发送删除Pod的命令，默认宽限期（30秒）

2. api服务器中的pod会随着时间的推移而更新，超过这个时间后，pod会被认为是“死的”，

并且会有宽限期。

3.在客户端命令中列出时，Pod显示为“终止”

4.（与3同时）当Kubelet发现Pod已被标记为终止因为2中的时间已经设置，它开始Pod关闭过程。

4.1 如果其中一个Pod的容器定义了一个preStop挂钩，则会在容器内部调用它。如果在宽限期到期后preStop挂钩仍在运行，则以小（2秒）扩展宽限期调用步骤2。

4.2容器被发送TERM信号。请注意，并非Pod中的所有容器都会同时收到TERM信号，并且如果它们关闭的顺序很重要，则每个容器都需要preStop挂钩。

5.（与3同时）Pod从端点列表中删除以进行维护，并且不再被视为副本控制器的正在运行的Pod集合中的一部分。缓慢关闭的pod无法继续提供流量，因为负载平衡器（如服务代理）会将其从轮询中移除。

6.当宽限期到期时，仍然在Pod中运行的任何进程都将被SIGKILL杀死。

7.Kubelet将通过设置宽限期0（立即删除）完成删除API服务器上的Pod。 Pod从API中消失，不再从客户端可见。

默认情况下，所有删除都在30秒内完成。kubectl delete命令支持--grace period=<seconds>选项，该选项允许用户重写默认值并指定自己的值。值0 强制删除pod。为了执行强制删除，必须指定附加标志--force和--grace period=0

**强制删除Pod**

强制删除POD定义为立即从集群状态和ETCD中删 除POD。执行强制删除时，API服务器不会等待Kubelet确认POD已在其运行的节点上终止。它会立即删除API中的pod，以便可以使用相同的名称创建新的pod。在节点上，设置为立即终止的pod在被强制杀死之前仍将得到一个小的宽限期。

对于某些吊舱，强制删除可能存在潜在危险，应谨慎执行。如果是statefulset pods，请参阅有关从statefulset中删除pods的任务文档。

**7.1.2.6 Pod容器特权模式**

pod中的任何容器都可以使用容器spec的安全上下文中的privileged启用特权模式。这对于希望使用Linux功能（如操作网络堆栈和访问设备）的容器很有用。容器内的进程获得的特权与容器外的进程几乎相同。在特权模式下，将网络和卷插件作为独立的数据包编写起来应该更容易，而不需要编译到Kubelet中。

注意：容器运行环境必须支持特权容器的概念，才能使此设置相关

### 7.1.3 Pod生命周期

Pod的status字段是一个Pod Status对象，它有一个phase字段。

**pod phase**

POD的phase是对POD在其生命周期中所处位置的一个简单、高级的总结。该phase不打算是对容器或pod状态观察的综合汇总，也不打算成为综合状态机。

POD phase值的数量和含义受到严格保护。除了这里记录的，不应该假设pod具有给定phase值。

这里是phase可能的值：

|  |  |
| --- | --- |
| 值 | 描述 |
| Pending | pod已被kubernetes系统接受，但一个或多个容器镜像尚未创建。这包括调度前的时间以及通过网络下载镜像所花费的时间，这可能需要一段时间 |
| Running | POD已绑定到一个节点，并且所有容器都已创建。至少有一个容器仍在运行，或者正在启动或重新启动。 |
| Succeeded | POD中的所有容器都已成功终止，不会重新启动。 |
| Failed | Pod中的所有容器都已终止，并且至少有一个Container已终止失败。 也就是说，Container要么退出非零状态，要么被系统终止 |
| Unknown | 由于某些原因，无法获得POD的状态，通常是由于与POD主机通信时出错。 |
|  |  |

**pod条件**

一个pod有一个pod status，它有一个pod conditions数组，用于检测pod是否满足条件。pod condition数组的每个元素都有六个可能的字段：

* lastProbetime字段提供上次探测POD条件的时间戳。
* lastTransitionTime字段提供POD上次从一种状态转换到另一种状态时的时间戳。
* message字段是一条可读的消息，指示有关转换的详细信息。
* reason字段是唯一的，一个词，最后一次转换的原因。
* status字段的可能值是True，False,Unknown
* type字段是下面可能的值

PodScheduled：pod已经被调度到一个节点

Ready: POD能够服务请求，应该添加到所有匹配到的服务的负载平衡池中。

Initialized：所有初始化容器启动成功

Unschedulable：调度程序现在不能调度pod，例如由于缺乏资源或其他限制；

ContainersReady：pod中所有容器就绪。

**容器探针**

probe是Kubelet定期对容器进行的诊断。为了执行诊断，kubelet调用由容器实现的处理程序。有三种类型的处理程序：

ExecAction：在容器内执行特定的命令。如果命令退出时状态代码为0，则认为诊断成功。

TCPSocketAction：对容器的IP地址指定端口上执行TCP检查。如果端口打开，则认为诊断成功。

HTTPGetAction:在指定的端口和请求路径上对容器的IP地址执行HTTP GET请求。如果响应的状态代码大于或等于200且小于400，则认为诊断成功。

每个探针有三个结果之一：

Success容器通过了诊断。

Failure：容器诊断失败。

Unknown：诊断失败，因此不应采取任何措施

Kubelet可以选择性在运行容器上执行和响应两种探针：

* livenessProbe：指示容器是否正在运行。如果liveness探测失败，kubelet将杀死容器，容器将服从其重新启动策略。如果容器不提供活动性探测，则默认状态为成功。
* readinessProbe：指示容器是否准备好服务请求。如果readiness失败，端点控制器将从与pod匹配的所有服务的端点中删除pod的IP地址。初始延迟前的默认就绪状态是Failure。如果容器不提供就绪探测，则默认状态为成功。

什么时候使用livenessprobe和readinessprobe？

如果容器中的进程在遇到问题或变得不健康时能够自行崩溃，则不一定需要活动探测器；Kubelet将根据POD的restartPolicy自动执行正确的操作。

如果希望在探测失败时终止并重新启动容器，请指定活动探测，并指定restartPolicy为Always或者OnFailure。

如果您只想在探测成功时开始向POD发送流量，请指定readinessprobe。在这种情况下，准备就绪探针可能与活动探针相同，但准备就绪探针存在意味着Pod将在不接收任何流量的情况下启动，并且仅在探针开始成功后才开始接收流量。如果容器需要在启动期间加载大型数据、配置文件或迁移，请指定就绪探测。

如果您希望容器能够自行停机进行维护，那么可以指定一个准备就绪探测器，用于检查特定于准备就绪的端点，该端点与活动性探测器不同。

请注意，如果您只想在POD被删除时释放请求，则不一定需要准备就绪探测器；删除时，无论准备就绪探测器是否存在，POD都会自动将自己置于未读状态。当POD等待POD中的容器停止时，POD仍处于未就绪状态。

有关如何设置活动或准备就绪探测器的详细信息，请参阅配置活动和准备就绪探测器。

**Pod和容器状态**

有关pod容器状态的详细信息，请参阅pod status和containerstatus。请注意，报告Pod状态的信息取决于当前的containers状态

**容器状态**

一旦pod被调度程序分配给一个节点，kubelet就开始使用容器环境创建容器。容器有三种可能的状态：Waiting、Running和Terminated。要检查容器的状态，可以使用kubectl describe pod[pod\_name]。显示该pod中每个容器的状态

Waiting：容器默认状态。如果容器不在第二个Running状态或Terminated状态中，则处于Waiting状态。一个在等候状态的容器仍在运行，如拉取图像、赋予secret等。与此状态一起，显示了一个关于状态的信息和原因，以提供更多的信息。

...

State: Waiting

Reason: ErrImagePull

...

Running：表示容器正在执行，没有问题。一旦容器进入运行状态，就执行poststart hook（如果有）。此状态还显示容器进入运行状态的时间。

...

State: Running

Started: Wed, 30 Jan 2019 16:46:38 +0530

...

Terminated：指示容器已完成其执行并已停止运行。当容器成功完成执行或由于某种原因失败时，它将进入此状态。无论如何，将显示原因和退出代码，以及容器的开始和完成时间。在容器进入终止状态之前，将执行prestop hook（如果有）。

...

State: Terminated

Reason: Completed

Exit Code: 0

Started: Wed, 30 Jan 2019 11:45:26 +0530

Finished: Wed, 30 Jan 2019 11:45:26 +0530

...

**Pod readiness Gate**

k8s v1.14 处于stable版本

为了通过向podstatus中注入额外的反馈或信号来增加pod readiness的可扩展性，kubernetes 1.11引入了一个名为pod ready++特性。您可以使用podspec中的新字段readinessgate来指定要评估pod就绪性的附加条件。如果kubernetes在pod的status.conditions字段中找不到这样的条件，那么该条件的状态将默认为“false”。下面是一个例子：

Kind: Pod

...

spec:

readinessGates:

- conditionType: "www.example.com/feature-1"

status:

conditions:

- type: Ready # this is a builtin PodCondition

status: "False"

lastProbeTime: null

lastTransitionTime: 2018-01-01T00:00:00Z

- type: "www.example.com/feature-1" # an extra PodCondition

status: "False"

lastProbeTime: null

lastTransitionTime: 2018-01-01T00:00:00Z

containerStatuses:

- containerID: docker://abcd...

ready: true

...

新的POD条件必须符合Kubernetes label key格式。由于kubectl patch命令仍然不支持path对象状态，因此必须使用其中一个kubeclient库通过Path操作注入新的pod条件。

随着新Pod条件的引入，只有当以下两个语句都成立时，才会认为Pod就绪：

Pod中的所有容器都已准备就绪。

ReadinessGates中指定的所有条件均为“True”。

为了便于对Pod准备情况评估，引入了新的Pod条件ContainersReady来捕获旧的Pod Ready条件。

在K8S 1.11中，作为alpha特性，“pod ready++”特性必须通过将podreadinessgates特性设置为true来显式启用。

在K8S 1.12中，该功能默认启用

**重启策略**

一般来说，pod不会消失，直到有人摧毁它们。 这可能是人或控制器。 此规则的唯一例外是，具有成功或失败超过一定持续时间（由master中的terminate-pod-gc-threshold确定）的phase的Pod将到期并自动销毁。

有三种控制器可供选择：

* 使用Job 控制 Pods预期终止，例如批量计算。Jobs仅适用于具有OnFailure或Never的restartPolicy的Pod。
* 对不希望终止的Pod（例如，Web服务器）使用ReplicationController，ReplicaSet或Deployment。 ReplicationControllers仅适用于具有restartPolicy等于Always的Pod。
* 需要为每台机器运行一个的Pod使用DaemonSet，因为它们提供特定于机器的系统服务。

所有三种类型的控制器都包含PodTemplate。 建议创建适当的控制器并让它创建Pod，而不是自己直接创建Pod。 这是因为Pods单独对机器故障没有弹性，但是控制器可以。如果节点死亡或与集群的其余部分断开连接，Kubernetes会应用策略将丢失节点上所有Pod的phase设置为Failed

**例子**

高级liveness probe例子

liveness probe由kubelet执行，因此所有请求都在kubelet网络名称空间中进行。

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-http

spec:

containers:

- args:

- /server

image: k8s.gcr.io/liveness

livenessProbe:

httpGet:

# when "host" is not defined, "PodIP" will be used

# host: my-host

# when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and "HTTPS" are allowed

# scheme: HTTPS

path: /healthz

port: 8080

httpHeaders:

- name: X-Custom-Header

value: Awesome

initialDelaySeconds: 15

timeoutSeconds: 1

name: liveness

状态例子

* Pod正在运行并有一个Container。容器出口为成功。

1. 日志完成事件。
2. 如果restartPolicy是：

Always：重启容器; Pod phase为Running。

OnFailure：Pod phase为Successed。

Never：Pod phase为Successed。

* Pod正在运行并有一个Container。容器出口失败。

1. 日志失败事件。
2. 如果restartPolicy是：

Always：重启容器; Pod phase为Running。

OnFailure：重启容器; Pod phase为Running。

Never：Pod阶段变为Failed。

* Pod正在运行并有两个容器。容器1出现故障。

1. 日志失败事件。
2. 如果restartPolicy是：

Always：重启容器; Pod phase为Running。

OnFailure：重启容器; Pod phase为Running。

Never：不要重启容器; Pod phase为Running。

1. 如果Container 1未运行，并且Container 2退出：

日志失败事件。

如果restartPolicy是：

Always：重启容器; Pod phase为Running。

OnFailure：重启容器; Pod phase为Running。

Never：Pod phase为Failed。

* Pod正在运行并有一个Container。容器耗尽内存。

容器终止失败。

记录OOM事件。

如果restartPolicy是：

Always：重启容器; Pod phase为Running。

OnFailure：重启容器; Pod phase为Running。

Never：记录失败事件; Pod phase为Failed。

* Pod正在运行，磁盘已经死亡。

杀死所有容器。

记录适当的事件。

Pod phase为Failed。

如果在控制器下运行，Pod将在其他位置重新创建。

* Pod正在运行，其节点已分离。

节点控制器等待超时。

节点控制器将Pod phase设置为Failed。

如果在控制器下运行，Pod将在其他位置重新创建。

### 7.1.4 初始化容器

初始化容器：在pod中的app容器之前运行的专用容器。init容器可以包含应用程序镜像中不存在的实用程序或安装脚本。

您可以在pod spec中与containers数组（它描述应用程序容器）一起指定init容器。

**理解初始化容器**

一个pod可以有多个容器在其中运行应用程序，但它也可以有一个或多个init容器，这些容器在应用程序容器启动之前运行。init容器与普通容器完全相同，除了：

init容器总是运行到完成。

每个init容器必须在下一个容器启动之前成功完成。

如果一个pod的init容器失败，kubernetes会反复重新启动pod，直到init容器成功。但是，如果pod的重启策略为Never，那么kubernetes不会重启pod。

要为pod指定init容器，请将init containers字段添加到pod spec中，该字段作为类型container的对象数组，与app containers数组一起。init容器的状态作为容器状态的数组（类似于.status.containerStatus字段）返回到.status.initcontainerStatus字段中。

**使用初始化容器**

由于init容器将镜像与应用程序容器分开，因此对于与启动相关的代码，它们具有一些优势：

* init容器可以包含应用程序镜像中不存在的安装程序的实用程序或自定义代码。例如，不需要从另一个镜像生成镜像，只需要在设置期间使用类似于sed、awk、python或dig的工具。
* init容器可以安全地运行实用程序，从而降低应用程序容器镜像的安全性。
* 应用程序镜像构建和部署角色可以独立工作，无需共同构建单个应用程序镜像。
* 与同一个pod中的应用程序容器相比，init容器可以以不同的文件系统视图运行。因此，他们可以访问应用程序容器无法访问的secret。
* 因为init容器在任何应用容器启动之前运行到完成，所以init容器提供一种机制来阻止或延迟应用容器启动，直到满足一组前提条件。一旦满足前提条件，pod中的所有应用程序容器都可以并行启动。

示例

1. 以下是有关如何使用init容器的一些想法：

2. 使用shell单行命令等待创建服务，如：

for i in 1..100 do sleep 1；if dig myservice；then exit 0；fi；done；exit 1

3. 使用如下命令从向下API向远程服务器注册此pod：

curl-x post http://$management\_service\_host:$management\_service\_port/register -d 'instance=$（<pod\_name>）&ip=$（<pod\_ip>）'

4.在用如下命令启动应用程序容器之前，请等待一段时间

sleep 60

5.将Git存储库克隆到卷中

6.将值放入配置文件并运行模板工具以动态生成主应用程序容器的配置文件。例如，将pod\_ip值放入配置中，并使用jinja生成主应用程序配置文件。

**在使用中初始化容器**

这个例子定义了一个简单的pod，它有两个init容器。第一个等待myservice，第二个等待mydb。一旦两个init容器都完成，pod就从其spec部分运行app容器。

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox:1.28

command: ['sh', '-c', 'echo The app is running! && sleep 3600']

initContainers:

- name: init-myservice

image: busybox:1.28

command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']

- name: init-mydb

image: busybox:1.28

command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']

下面的yaml定义myservice和mydb服务

apiVersion: v1

kind: Service

metadata:

name: myservice

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

---

apiVersion: v1

kind: Service

metadata:

name: mydb

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9377

你可以通过下面方式启动pod：

kubectl apply -f myapp.yaml

pod/myapp-pod created

并检查状态：

kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

myapp-pod 0/1 Init:0/2 0 6m

或者更多信息：

kubectl describe -f myapp.yaml

Name: myapp-pod

Namespace: default

[...]

Labels: app=myapp

Status: Pending

[...]

Init Containers:

init-myservice:

[...]

State: Running

[...]

init-mydb:

[...]

State: Waiting

Reason: PodInitializing

Ready: False

[...]

Containers:

myapp-container:

[...]

State: Waiting

Reason: PodInitializing

Ready: False

[...]

Events:

FirstSeen LastSeen Count From SubObjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

16s 16s 1 {default-scheduler } Normal Scheduled Successfully assigned myapp-pod to 172.17.4.201

16s 16s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Pulling pulling image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Pulled Successfully pulled image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Created Created container with docker id 5ced34a04634; Security:[seccomp=unconfined]

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Started Started container with docker id 5ced34a04634

查看初始化容器日志

kubectl logs myapp-pod -c init-myservice # Inspect the first init container

kubectl logs myapp-pod -c init-mydb

此时，这些init容器将等待发现名为mydb和myservices的服务。

您可以使用以下配置来显示这些服务

---

apiVersion: v1

kind: Service

metadata:

name: myservice

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

---

apiVersion: v1

kind: Service

metadata:

name: mydb

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9377

创建mydb和myservice

kubectl apply -f services.yaml

service/myservice created

service/mydb created

你将看到那些初始化容器完成，并且myapp-pod pod进入Running状态。

kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

myapp-pod 1/1 Running 0 9m

这个简单的示例应该为您创建自己的init容器提供一些灵感。 下一步包含指向更详细示例的链接

**详细行为**

在POD启动过程中，在网络和卷初始化之后，每个init容器按顺序启动。每个容器必须在下一个容器启动之前成功退出。如果容器由于运行时而无法启动或因失败而退出，则根据pod restartpolicy重试。但是，如果pod restartpolicy设置为always，则init容器在失败时使用restartpolicy。

在所有初始化容器成功之前，无法准备好POD。init容器上的端口未在服务下聚合。正在初始化的POD处于挂起状态，但应将Initializing设置为“True”。

如果pod重新启动或重新启动，则所有init容器必须再次执行。

对init container spec的更改仅限于container image字段。更改init container image字段等同于重新启动pod。

因为init容器可以重新启动、重试或重新执行，所以init容器代码应该是等幂的。尤其是，写入EmptyDirs上文件的代码应该为输出文件已经存在的可能性做好准备。

init容器具有应用程序容器的所有字段。但是，kubernetes禁止使用readinessprobe，因为init容器不能定义与完成不同的readiness。这在验证期间强制执行。

在pod上使用activedeadlineseconds，在容器上使用livenessProbe，以防止init容器永远失败。活动截止时间包括初始容器。

pod中每个app和init容器的名称必须是唯一的；任何与另一个容器共享名称的容器都会引发验证错误。

**资源**

给定init容器的顺序并按序执行，适用于以下资源使用规则：

* 在所有init容器上定义的任何特定请求资源或限制的最高值是有效的init资源请求/限制
* Pod对资源的有效请求/限制是以下值中的较高者：

所有应用程序容器的请求/限制资源的最高值

资源的有效初始化请求/限制

* 调度是基于有效的资源请求/限制来完成，这意味着init容器可以预留用于在Pod的生命周期间未使用的初始化的资源。
* Pod有效QoS层的QoS（服务质量）和init容器的QoS以及app容器的QoS相似的。

配额和限制是基于Pod请求和上限的有效值。Pod级别控制组（cgroup）基于有效的Pod请求和限制，与调度程序相同。

**Pod重启原因**

一个Pod重启，引起初始化容器的重新执行，有以下原因：

* 一个用户更新pod的特征，引起初始化容器镜像改变。任何对初始化容器镜像的改变重启pod。应用容器镜像改变只会重启应用容器。
* pod基础容器重启。这种情况并不常见，必须由对节点具有root访问权限的人员进行的操作。
* pod中所有容器中断同时restartPolicy被设置为Always，强制重启并且由于垃圾回收，init容器完成记录已丢失

### 7.1.5 pod预设

本节介绍PodPreset概要。PodPreset是一个对象用来在pod创建时注入某些信息。这些信息可以报考secret、卷、卷挂载和环境变量。

**理解PodPreset**

PodPreset是一种API资源，用于在创建时将其他运行时条件注入Pod。 您可以使用标签选择器指定使用PodPreset的Pod。

使用Pod预设允许pod模板作者不必显式提供每个pod的所有信息。 这样，使用特定服务的pod模板的作者不需要知道有关该服务的所有详细信息。

**怎样工作**

K8S提供了一个访问控制器（PodPreset），当启用时，应用PodPreset注入给pod某种请求资源。当一个pod创建请求资源发生时，系统做了下列事项：

1）获得所有可用的PodPreset

2）检查任何一个PodPreset的标签选择器匹配被创建的Pod上的标签

3）尝试合并各种PodPreset中定义的资源到被创建的pod中。

4）如果出错，提示一个事件说明pod上的合并错误，并创建不注入podpreset中定义的资源的Pod。

5）注解生成的修改后的Pod特性，以指示它已被PodPreset修改。注解形式：

podpreset.admission.kubernetes.io/podpreset-<pod-preset name>: "<resource version>"

每个pod可以匹配零或者多个PodPreset，每个podpreset可以应用与零个或者多个pod。当一个PodPreset应用于一个或者多个pod，k8s修改pod的spec属性。对Env、EnvFrom、VolumeMounts的改变，K8S修改Pod中所有容器的spec。对于Volume的改变，k8s修改pod的spec。

注意：Pod Preset能够在适当的时候修改Pod spec的以下字段： - .spec.containers字段。 - initContainers字段（需要Kubernetes版本1.14.0或更高版本）

禁用特定pod的podpreset：

在某些情况下，您希望Pod不会被任何Pod Preset突变所改变。 在这些情况下，您可以在Pod spec中添加注释：podpreset.admission.kubernetes.io/exclude：“true”

**启用PodPreset**

为了在集群中使用podpreset，你必须确保下面设置：

1.已经启用API类型settings.k8s.io/v1alpha1/podpreset。例如在API Server的--runtime-config参数选项中添加settings.k8s.io/v1alpha1=true。在minikube的集群启动时添加标识

--extra-config=apiserver.runtime-config=settings.k8s.io/v1alpha1=true。

2.已经启动访问控制器PodPreset。一种方式可以在APIServer的--enable-admission-plugins可选参数中指定PodPreset。在minikube，增加标识

--extra-config=apiserver.enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota,PodPreset

3. 您已通过在将使用的命名空间中创建PodPreset对象来定义Pod预设

### 7.1.6 中断

本页指南适用于想要构建高可用性应用程序的应用程序所有者，因此需要了解Pods可能发生的中断类型。

它也适用于希望执行自动群集操作的集群管理员，例如升级和自动缩放群集。

**自行和非自行中断**

pod不会消失直到某人（人或者控制器）销毁它们，或者有一个不可避免的硬件或者系统软件错误。

我们把这些不可避免的因素叫做应用的非自行中断。例如：

* 节点的物理机器上的硬件错误
* 集群管理员错误删除VM。
* 云供应商或者管理程序错误导致VM消失
* 内核资源紧张
* 由于集群网络隔离造成节点丢失
* 节点资源不足造成pod驱逐

除资源不足外，大多数用户都应熟悉所有这些条件; 它们不是Kubernetes特有的。

我们将其他条件称为自行中断。 其中包括应用程序所有者启动的操作和集群管理员启动的操作。

典型应用程序所有者操作包括：

* 删除管理pod的deployment或者其他控制器
* 更新部署的pod模板引起重启
* 直接删除pod

集群管理员操作包括：

* 排除一个节点进行维修或者更新
* 从集群中排除一个节点缩小集群
* 从节点中删除pod以允许其他对象适合该节点。

这些操作可以由集群管理员直接执行，也可以由集群管理员或集群主机提供商自动运行。请咨询您的集群管理员或咨询您的云提供商或分发文档，以确定是否为您的集群启用了任何自行中断源。如果未启用，则可以跳过创建Pod中断预算。

注意：并非所有自行中断都受到Pod Disruption Budgets的限制。 例如删除部署或Pod会绕过Pod Disruption Budgets。

**处理中断**

这里有一些方法来减轻非自愿中断：

* 确保你的pod请求它需要的资源
* 如果需要高可用增加应用副本
* 为了高可用运行应用副本，跨机架（使用反亲和）或跨区域（如果使用多区域集群）传播应用程序。

自行中断的频率各不相同。在基本的Kubernetes集群上，根本没有自行中断。 但是，您的集群管理员或托管服务提供商可能会运行一些导致自行中断的其他服务。例如，推出节点软件更新可能会导致自行中断。 此外，集群（节点）自动缩放的某些实现可能会导致自动中断，如磁盘碎片整理和压缩节点。 您的集群管理员或托管服务提供商应记录预期的自行中断级别（如果有的话）。

Kubernetes提供的功能可以帮助您在频繁的自行中断的同时运行高可用性应用程序。 我们将这组功能称为中断预算

**PDB怎么样工作**

应用程序所有者可以为每个应用程序创建PodDisruptionBudget对象（PDB）。PDB限制副本应用程序的pod由于自行中断同时发生故障的数量。例如，基于预案的应用程序希望确保运行的副本数量永远不会低于预案所需的数量。 Web前端可能希望确保服务负载的副本数量永远不会低于总数的某个百分比。

集群管理者和托管提供商应使用通过调用Eviction API而不是直接删除pod或部署来遵守Pod Disruption Budgets。 例如kubectl drain命令和Kubernetes-on-GCE集群升级脚本（cluster / gce / upgrade.sh）。

当集群管理员想要排除节点时，他们使用kubectl drain命令。 该工具试图驱逐机器上的所有pod。 可以暂时拒绝逐出请求并且该工具定期重试所有失败的请求，直到所有pod终止，或者直到达到超时时间。

PDB指定应用程序可以容忍的副本数量，相对于预期的副本数量。例如具有.spec.replicas：5的部署应该在任何给定时间具有5个pod。如果其PDB允许某个时间有4个，则Eviction API将允许一次自动中断一个而不是两个pod。

组成应用程序的pod组使用标签选择器指定，与应用程序控制器使用的标签选择器相同（deploymnet，statefulset等）。“预期”数量的pod是根据pods控制器的.spec.replicas计算的。使用对象的.metadata.ownerReferences从pod中发现控制器。

PDB不能防止非自行中断发生，但它们确实违背了预算。

由于应用滚动升级而被删除或不可用的Pod确实会计入中断预算，但是在执行滚动升级时，控制器（如deployment或者statefulset）不受PDB限制 - 在应用程序更新期间处理故障的方式在控制器的spec中设置。

当使用eviction API驱逐pod时，它会被正常终止（请参阅PodSpec中的terminationGracePeriodSeconds。）

**PDB例子**

考虑一个集群有3个node，node-1到node-3。集群运行多个应用。他们中一个有3个副本初始化为pod-a，pod-b，pod-c。另一个不相关的pod没有PDB，叫做pod-x。初始化的时候，pod的情况如下：

|  |  |  |
| --- | --- | --- |
| Node-1 | Node-2 | Node-3 |
| pod-a available | pod-b available | pod-c available |
| pod-x available |  |  |

所有3个pod是一个deployment的部分，并且他们作为集合，有一个PDB要求在所有时刻至少存活2个Pod。

例如假设集群管理员想要重新启动机器到新的内核版来本，修复内核中的错误。集群管理员首先尝试使用kubectl drain命令排空node-1。该工具试图驱逐pod-a和pod-x。 这立即成功。 两个pod同时进入终止状态。 这使集群处于以下状态：

|  |  |  |
| --- | --- | --- |
| Node-1 | Node-2 | Node-3 |
| pod-a terminating | pod-b available | pod-c available |
| pod-x terminating |  |  |

deployment注意到其中一个pod正在终止，因此它创建了一个名为pod-d的替换。 由于node-1被封锁，它将落在另一个节点上。还创建了pod-y作为pod-x的替代品。

（注意：对于StatefulSet，pod-a，它将被称为pod-1，需要在替换之前完全终止，也称为pod-1，但可以创建不同的UID。例也适用于StatefulSet。）

现在集群处于以下状态：

|  |  |  |
| --- | --- | --- |
| Node-1 draining | Node-2 | Node-3 |
| pod-a terminating | pod-b available | pod-c available |
| pod-x terminating | pod-d starting | pod-y |

在某时刻pod中断，集群状态如下：

|  |  |  |
| --- | --- | --- |
| Node-1 drained | Node-2 | Node-3 |
|  | pod-b available | pod-c available |
|  | pod-d starting | pod-y |

此时，如果不耐烦的集群管理员试图排除node-2或node-3，则drain命令将被阻止，因为部署只有2个可用的pod，并且其PDB至少需要2个pod。.经过一段时间后， pod-d变得可用。群集状态现在看起来像这样：

|  |  |  |
| --- | --- | --- |
| Node-1 drained | Node-2 | Node-3 |
|  | pod-b available | pod-c available |
|  | pod-d available | pod-y |

现在，集群管理员尝试排除node-2。drain命令将尝试以某种顺序驱逐两个pod，首先pod-b然后pod-d。 它将成功驱逐pod-b。 但是，当它试图驱逐pod-d时，它将被拒绝，因为这将只留下一个可用于部署的pod。

部署为pod-b创建了一个名为pod-e的替代品。 由于群集中没有足够的资源来安排pod-e，因此排除将再次阻塞。 群集可能最终处于此状态：

|  |  |  |  |
| --- | --- | --- | --- |
| Node-1 drained | Node-2 | Node-3 | No node |
|  | pod-b available | pod-c available | pod-e pending |
|  | pod-d available | pod-y |  |

此时，集群管理员需要将节点添加回集群以继续升级。你可以看到Kubernetes如何改变发生中断的速度，根据：

* 应用程序需要多少个副本
* 优雅地关闭实例需要多长时间
* 启动新实例需要多长时间
* 控制器的类型
* 集群的资源容量

**分开集群拥有者和应用拥有者角色**

通常，将群集管理器和应用程序所有者视为彼此知之甚少的单独角色很有用。在这些情况下，这种职责分离可能有意义：

* 当有许多应用程序团队共享Kubernetes集群时，角色自然有隔离
* 当第三方工具或服务用于自动化群集管理时

Pod Disruption Budgets通过在角色之间提供接口来支持这种角色分离。如果您的组织中没有这样的责任分离，则可能不需要使用Pod Disruption Budgets

**如何在群集上执行中断操作**

如果您是集群管理员，并且需要对集群中的所有节点执行中断操作，例如节点或系统软件升级，则可以使用以下选项：

* 在升级期间接受停机时间。
* 故障转移到另一个完整的副本集群。

没有停机时间，但对于重复节点和人力协调切换可能都是昂贵的。

* 编写容错中断应用程序并使用PDB。

没有停机时间。

最小的资源重复。

允许更多集群管理自动化。

编写容忍中断的应用程序很棘手，但容忍自行中断的工作在很大程度上与支持自动缩放和容忍非自行中断的工作重叠

## 7.2 控制器

### 7.2.1 ReplicaSet

ReplicaSet的目的是维持一个在任何时间稳定的副本Pod集合。正因为如此，常用来保证指定数量的相同Pod的可用性。

**ReplicaSet怎样工作**

ReplicaSet使用字段定义，包括指定如何识别它可以获取的Pod的选择器，指定它应该维护多少Pod的副本数，以及一个pod模板，指定它应创建的新Pod的数据，以满足副本条件的数量。然后，ReplicaSet通过根据需要创建和删除Pod来达到其目的，以达到所需的数量。当ReplicaSet需要创建新Pod时，它使用其Pod模板。

ReplicaSet与其Pods的链接是通过Pods的metadata.ownerReferences字段，该字段指定当前对象所拥有的资源。ReplicaSet获取的所有Pod在其ownerReferences字段中拥有其拥有的ReplicaSet标识信息。通过此链接，ReplicaSet知道它正在维护的Pod的状态并相应地进行计划。

ReplicaSet使用其选择器标识要获取的Pod。如果Pod没有OwnerReference或者OwnerReference不是控制器并且它与ReplicaSet的选择器匹配，则它将立即被所述ReplicaSet获取。

**什么时候使用ReplicaSet**

ReplicaSet确保在任何给定时间运行指定数量的pod副本。但是Deployment是一个更高级别的概念，它管理ReplicaSet并为Pod提供声明性更新以及许多其他有用的功能。因此，除非您需要自定义更新编排或根本不需要更新，否则我们建议使用Deployment而不是直接使用ReplicaSet。

这实际上意味着您可能永远不需要操作ReplicaSet对象：改为使用Deployment，并在spec部分中定义您的应用程序。

例子

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: frontend

labels:

app: guestbook

tier: frontend

spec:

# modify replicas according to your case

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: php-redis

image: gcr.io/google\_samples/gb-frontend:v3

将此清单保存到frontend.yaml并将其提交到Kubernetes集群将创建已定义的ReplicaSet及其管理的Pod。

kubectl apply -f frontend.yaml

获取当前部署的ReplicaSet

kubectl get rc

可以看见创建的frontend

NAME DESIRED CURRENT READY AGE

frontend 3 3 3 6s

也可以查看ReplicaSet的状态

kubectl describe rc/frontend

将看见类似输出结果：

ame: frontend

Namespace: default

Selector: tier=frontend,tier in (frontend)

Labels: app=guestbook

tier=frontend

Annotations: <none>

Replicas: 3 current / 3 desired

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

Pod Template:

Labels: app=guestbook

tier=frontend

Containers:

php-redis:

Image: gcr.io/google\_samples/gb-frontend:v3

Port: 80/TCP

Requests:

cpu: 100m

memory: 100Mi

Environment:

GET\_HOSTS\_FROM: dns

Mounts: <none>

Volumes: <none>

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-qhloh

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-dnjpy

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-9si5l

最后可以检查拉起的pod

kubectl get pods

看到以下Pod信息的输出结果

NAME READY STATUS RESTARTS AGE

frontend-9si5l 1/1 Running 0 1m

frontend-dnjpy 1/1 Running 0 1m

frontend-qhloh 1/1 Running 0 1m

您还可以验证这些pod的所有者引用是否设置为frontend ReplicaSet。 为此，获取其中一个Pod的yaml：

kubectl get pods frontend-9si5l -o yaml

输出看起来与此类似，frontend ReplicaSet的信息在元数据的ownerReferences字段中设置：

apiVersion: v1

kind: Pod

metadata:

creationTimestamp: 2019-01-31T17:20:41Z

generateName: frontend-

labels:

tier: frontend

name: frontend-9si5l

namespace: default

ownerReferences:

- apiVersion: extensions/v1beta1

blockOwnerDeletion: true

controller: true

kind: ReplicaSet

name: frontend

uid: 892a2330-257c-11e9-aecd-025000000001

...

**非模板pod获取**

虽然您可以毫无问题地创建裸Pod，但强烈建议您确保裸Pod没有与其中一个ReplicaSet的选择器匹配的标签。 这是因为ReplicaSet不限于拥有其模板指定的Pod，它可以按照前面章节部分中提到的指定的方式获取其他Pod。

以前面frontend的例子按照如下资源清单指定Pod：

apiVersion: v1

kind: Pod

metadata:

name: pod1

labels:

tier: frontend

spec:

containers:

- name: hello1

image: gcr.io/google-samples/hello-app:2.0

---

apiVersion: v1

kind: Pod

metadata:

name: pod2

labels:

tier: frontend

spec:

containers:

- name: hello2

image: gcr.io/google-samples/hello-app:1.0

由于这些Pod没有Controller（或任何对象）作为其所有者引用并与frontend ReplicaSet的选择器匹配，因此它们将立即被它们获取。

假设您在部署了frontend ReplicaSet之后创建了Pod，并设置了其初始Pod副本以满足其副本计数要求。

kubectl apply -f pod-rs.yaml

新的Pod将由ReplicaSet获取，然后立即终止，因为ReplicaSet将超过其所需的副本数。

提取pod

kubectl get pods

输出显示新pod或者已经terminated或者在terminated的处理过程。

NAME READY STATUS RESTARTS AGE

frontend-9si5l 1/1 Running 0 1m

frontend-dnjpy 1/1 Running 0 1m

frontend-qhloh 1/1 Running 0 1m

pod2 0/1 Terminating 0 4s

如果你首先创建pod

kubectl apply -f pod-rs.yaml

然后创建replicaset：

kubectl apply -f frontend.yaml

您将看到ReplicaSet已获取Pod并且仅根据其sepc建新的Pod，直到其新Pod的数量与原始匹配其所需的副本数。取得Pods：

kubectl get pods

将在其输出中显示：

NAME READY STATUS RESTARTS AGE

frontend-pxj4r 1/1 Running 0 5s

pod1 1/1 Running 0 13s

pod2 1/1 Running 0 13s

以这种方式，副本集可以拥有一组非同类的Pod。

**写一个replicaset清单**

和其他K8S API对象，replicaset需要apiVersion，kind，metadata字段。对于replicaset，kind字段总是ReplicaSet。在Kubernetes 1.9中，ReplicaSet类型的API版本apps / v1是当前版本，默认情况下已启用。API版本apps / v1beta2已弃用。请参阅frontend.yaml示例的第一行。

ReplicaSet也需要spec。

1.Pod模板

.spec.template是pod模板，内部也需要标签。在我们的frontend.yaml示例中，我们有一个标签：tier：frontend。 注意不要与其他控制器的选择器重叠，以免他们尝试获取这个Pod。

对于模板的重启测试字段，.spec.template.spec.restartPolicy，仅能允许值为Always，也是默认值。

2.Pod选择器

.spec.selector字段是标签选择器。正如早期讨论的，这些标签用来识别潜在的Pod来获取。在我们的frontend.yaml示例中，选择器是：

matchLabels：

tier：frontend

在ReplicaSet中，.spec.template.metadata.labels必须匹配spec.selector，否则将被API拒绝。

注意：对于2个ReplicaSet指定相同.spec.selector但不同的.spec.template.metadata.labels和.spec.template.spec字段的，每个ReplicaSet将忽略由另一个ReplicaSet创建的Pod。

3.Replicas

您可以通过设置.spec.replicas来指定应同时运行的Pod数量。 ReplicaSet将创建/删除其Pod以匹配此数字。如果未指定.spec.replicas，则默认为1

**使用ReplicaSet**

1.删除ReplicaSet和pod

删除一个ReplicaSet和所有的pod，使用kubectl delete。垃圾回收器默认自动删除所有的依赖pod。当使用REST API或者client-go库，你必须使用-d参数对Background或者Foreground设置propagationPolicy。例如：

kubectl proxy --port=8080

curl -X DELETE

'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/frontend' \

> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' \

> -H "Content-Type: application/json"

2.仅仅删除一个ReplicaSet

您可以使用带有--cascade = false选项的kubectl delete删除Replicaset，而不会影响其任何Pod。 使用REST API或client-go库时，必须将propagationPolicy设置为Orphan。 例如：

kubectl proxy --port=8080

curl -X DELETE

'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/frontend' \

> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' \

> -H "Content-Type: application/json"

删除原始ReplicaSet后，您可以创建一个新的ReplicaSet来替换它。只要旧的和新的.spec.selector相同，那么新的将抓取旧的Pod。但是它不会做任何努力使现有的Pod匹配一个新的不同的pod模板。要以受控方式将Pod更新为新spec，请使用Deployment，因为ReplicaSet不直接支持滚动更新。

3.从ReplicaSet隔离Pods

您可以通过更改标签来从副本集中删除Pod。 此技术可用于从服务中删除Pod以进行调试，数据恢复等。以这种方式删除的Pod将自动替换（假设副本的数量也未更改）。

4.缩放一个副本

只需更新.spec.replicas字段即可轻松扩展或缩小ReplicaSet。 ReplicaSet控制器确保具有匹配标签选择器的所需数量的Pod可用且可操作。

5.ReplicaSet作为水平Pod自动缩放

ReplicaSet也可以用来作为Pod水平自动缩放（HPA）。也就是说，HPA可以自动缩放ReplicaSet。 以下是针对我们在上一个示例中创建的ReplicaSet的示例HPA。

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: frontend-scaler

spec:

scaleTargetRef:

kind: ReplicaSet

name: frontend

minReplicas: 3

maxReplicas: 10

targetCPUUtilizationPercentage: 50

将此清单保存到hpa-rs.yaml并将其提交到Kubernetes集群创建定义的HPA，该HPA根据副本的Pod的CPU使用情况自动调整目标ReplicaSet。

kubectl apply -f hpa-rs.yaml

或者，您可以使用kubectl autoscale命令来完成相同的操作（并且更容易！）

kubectl autoscale rs frontend --max=10

**替代replicaset**

Deployment(推荐)

Deployment是一个对象，可以拥有ReplicaSet并通过声明性的服务器端滚动更新来更新它们及其Pod。虽然ReplicaSet可以独立使用，但今天它们主要被Deployments用作协调Pod创建，删除和更新的机制。使用Deployment时，您不必担心管理它们创建的副本集。 Deployment拥有并管理其ReplicaSet。因此，建议您在需要ReplicaSet时使用Deployment。

裸Pod

与用户直接创建Pod的情况不同，ReplicaSet会替换因任何原因而被删除或终止的Pod，例如在节点故障或破坏性节点维护（例如内核升级）的情况下。因此，即使您的应用程序只需要一个Pod，我们也建议您使用ReplicaSet。 可以想象它与流程主管类似，只是它监控多个节点上的多个Pod，而不是单个节点上的单个进程。 ReplicaSet将本地容器重新启动委派给节点上的某个代理程序（例如，Kubelet或Docker）。

Job

对于预期会自行终止的Pod（即批处理作业），使用Job而不是ReplicaSet

DaemonSet

使用DaemonSet代替提供机器级功能的Pods的ReplicaSet，例如机器监视或机器日志记录。 这些Pod具有与机器生命周期相关的生命周期：Pod需要在其他Pod启动之前在机器上运行，并且当机器准备好重新启动/关闭时可以安全终止。

ReplicationController

ReplicaSet是ReplicationControllers的后续版本。 这两个用途相同，行为相似，只是ReplicationController不支持基于集合的选择器。因此，ReplicaSet优先于ReplicationControllers

### 7.2.2 ReplicationController

注意：现在建议使用配置ReplicaSet的Deployment来设置复本。

ReplicationController确保一次运行指定数量的pod副本。 换句话说，ReplicationController确保一个pod或一组同类pod总是可用。

**ReplicationController怎样工作**

如果存在太多pod，则ReplicationController将终止额外的pod。如果太少，ReplicationController将启动更多pod。与手动创建的pod不同，ReplicationController维护的pod在失败，删除或终止时会自动替换。例如，在内核升级等破坏性维护之后，会在节点上重新创建pod。因此，即使应用程序只需要一个pod，也应该使用ReplicationController。 ReplicationController类似于进程管理程序，但是ReplicationController不是监视单个节点上的各个进程，而是监视多个节点上的多个pod。

在讨论中，ReplicationController通常缩写为“rc”或“rcs”，并且作为kubectl命令中的快捷方式。

一个简单的例子是创建一个ReplicationController对象，以无限期地可靠地运行Pod的一个实例。更复杂的用例是运行副本服务的几个相同副本，例如Web服务器

**ReplicationController运行例子**

示例ReplicationController配置运行nginx Web服务器的三个副本

apiVersion: v1

kind: ReplicationController

metadata:

name: nginx

spec:

replicas: 3

selector:

app: nginx

template:

metadata:

name: nginx

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

创建命令kubectl apply -f replication.yaml

replicationcontroller/nginx created

检查ReplicationController状态

kubectl describe replicationcontrollers/nginx

Name: nginx

Namespace: default

Selector: app=nginx

Labels: app=nginx

Annotations: <none>

Replicas: 3 current / 3 desired

Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx

Port: 80/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- ---- ------ -------

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-qrm3m

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-3ntk0

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-4ok8v

创建了三个pod，但没有一个正在运行，可能是因为正在拉取镜像。稍后，相同的命令可能会显示：

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

要以机器可读的形式列出属于ReplicationController的所有pod，可以使用如下命令：

pods=$(kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name})

echo $pods

nginx-3ntk0 nginx-4ok8v nginx-qrm3m

这里选择器与ReplicationController的选择器（在kubectl describe输出中看到）相同，并且在replication.yaml中以不同的形式。 --output = jsonpath选项指定一个表达式，该表达式只从返回列表中的每个pod获取名称。

**写一个ReplicationController spec**

与所有其他Kubernetes配置一样，ReplicationController需要apiVersion，kind和metadata字段。 有关使用配置文件的一般信息，请参阅对象管理。ReplicationController还需要一个.spec部分。

Pod模板

.spec.template是.spec中唯一必需的字段。

.spec.template是一个pod模板。 它与pod具有完全相同的架构，除了它是嵌套的并且没有apiVersion或kind。

除了Pod的必需字段之外，ReplicationController中的pod模板还必须指定适当的标签和适当的重新启动策略。 对于标签，请确保不要与其他控制器重叠。 请参阅pod选择器。

只允许等于Always的.spec.template.spec.restartPolicy，如果未指定，则为默认值。

对于本地容器重新启动，ReplicationControllers委托给节点上的代理程序，例如Kubelet或Docker。

ReplicationController上的标签

ReplicationController本身可以有标签（.metadata.labels）。 通常，您可以将它们设置为与.spec.template.metadata.labels相同; 如果未指定.metadata.labels，则默认为.spec.template.metadata.labels。 但是，允许它们不同，并且.metadata.labels不会影响ReplicationController的行为。

Pod选择器

.spec.selector字段是标签选择器。 ReplicationController管理具有与选择器匹配的标签的所有pod。 它不区分它创建或删除的pod以及另一个人或进程创建或删除的pod。 这允许在不影响正在运行的pod的情况下替换ReplicationController。

如果指定，则.spec.template.metadata.labels必须等于.spec.selector，否则它将被API拒绝。 如果未指定.spec.selector，则默认为.spec.template.metadata.labels。

此外，您通常不应创建任何标签与此选择器匹配的pod，可以直接创建，与另一个ReplicationController或其他控制器（如Job）匹配。 如果这样做，ReplicationController会认为它创建了其他pod。 Kubernetes并没有阻止你这样做。

如果你最终得到了多个具有重叠选择器的控制器，你必须自己管理删除（见下文）

多副本

您可以通过将.spec.replicas设置为要同时运行的pod数来指定应同时运行的pod数。 在任何时间运行的数字可能更高或更低，例如，如果副本只是增加或减少，或者如果正常关闭pod，并且提前开始更换。

如果未指定.spec.replicas，则默认为1

**使用ReplicationController**

* 删除replicationcontroller和pod

要删除ReplicationController及其所有pod，请使用kubectl delete。 在删除ReplicationController本身之前，Kubectl会将ReplicationController缩放为零并等待它删除每个pod。 如果此kubectl命令被中断，则可以重新启动它。

使用REST API或转到客户端库时，需要显式执行这些步骤（将副本扩展为0，等待pod删除，然后删除ReplicationController）。

* 仅删除replicationcontroller

您可以删除ReplicationController而不影响其任何pod。

使用kubectl，为kubectl delete指定--cascade = false选项。

使用REST API或转到客户端库时，只需删除ReplicationController对象即可。

删除原始文件后，您可以创建一个新的ReplicationController来替换它。 只要旧的和新的.spec.selector相同，那么新的将抓取旧的pod。然后它不会做任何努力使现有的pod匹配一个新的不同的pod模板。 要以受控方式将pod更新为新spec，请使用滚动更新

* 从replicationcontroller中隔离pod

可以通过更改标签来从ReplicationController的目标集中删除Pod。 此技术可用于从服务中删除pod以进行调试，数据恢复等。以这种方式删除的Pod将自动替换（假设副本的数量也未更改）。

**常用模式**

* 重新调度

如上所述，无论您是要保持运行1个pod还是1000个，ReplicationController都将确保存在指定数量的pod，即使在节点发生故障或pod终止时（例如，由于另一个控制代理的操作）。

* 缩放

通过简单地更新replicas字段，ReplicationController可以轻松地手动或通过自动缩放控制代理来向上或向下扩展副本数量

* 滚动升级

如＃1353中所述，建议的方法是创建一个具有1个副本的新ReplicationController，逐个扩展新的（+1）和旧（-1）控制器，然后在它达到0个副本后删除旧控制器。 无论意外故障如何，这都可以预测更新pod的集合。

理想情况下，滚动更新控制器会考虑应用程序准备情况，并确保在任何给定时间内有足够数量的pod可以高效地运行。

两个ReplicationControllers需要创建具有至少一个区分标签的pod，例如pod的主容器的image标签，因为通常是镜像更新，可以激发滚动更新。

滚动更新在客户端工具kubectl rolling-update中实现。 访问kubectl滚动更新任务以获取更具体的示例

* 多种发行通道

除了在滚动更新正在进行时，运行多个版本的应用程序之外，通常使用多个版本跟踪长时间运行多个版本，甚至连续运行多个版本。 发布通道将按标签区分。

例如，服务可能会定位所有具有tier（in frontend），environment in(prod)的pod。 现在，你有10个复制的pod组成这个服务层。 但是你希望能够'canary'这个组件的新版本。 您可以为一个ReplicationController设置副本数为9，标签为tier = frontend，environment = prod，track = stable，另一个ReplicationController对象副本数设置为1，标签为tier = frontend， environment = prod，track = canary。 现在该服务涵盖了金丝雀和非金丝雀pod。 但是你可以分别搞乱ReplicationControllers来测试，监视结果等。

* 和service一起使用ReplicationControllers

多个ReplicationControllers可以位于单个服务之后，例如，某些流量转到旧版本，有些流量转到新版本。

ReplicationController永远不会自行终止，但预计它不会像服务一样长寿。服务可以由多个ReplicationControllers控制的pod组成，并且预计可以在服务的生命周期内创建和销毁许多ReplicationController（例如，执行运行服务的pod的更新）。服务本身及其客户端都应该忽略维护服务pod的ReplicationControllers。

**对ReplicationController进行编程**

由ReplicationController创建的Pod旨在是可互换的和语义相同的，尽管它们的配置可能随着时间的推移变得异构。 这显然适用于复本的无状态服务器，但ReplicationControllers也可用于维护主选，分片和工作池应用程序的可用性。 此类应用程序应使用动态工作分配机制，例如RabbitMQ工作队列，而不是静态/一次性定制每个pod的配置，这被视为反模式。 执行的任何pod自定义，例如资源的垂直自动调整（例如，cpu或内存），应由另一个在线控制器进程执行，与ReplicationController本身不同

**ReplicationController的作用**

ReplicationController只是确保所需数量的pod与其标签选择器匹配并且可以运行。目前，只有已终止的pod从其计数中排除。将来，可以考虑系统提供的readiness和其他信息，我们可以对替换策略添加更多控制，并且我们计划发出可以由外部客户使用的事件，以实现任意复杂的替换和/或向下缩放的政策。

ReplicationController永远受限于这种狭隘的责任。它本身不会执行readiness或liveness探测。它不是执行自动缩放，而是由外部自动缩放器控制（如＃492中所述），这将改变其replicas字段。我们不会将调度策略（例如，传播）添加到ReplicationController。我们也不应该验证被控制的pod与当前指定的模板匹配，因为这会妨碍自动调整大小和其他自动化过程。同样，完成期限，排序依赖性，配置扩展和其他功能属于其他地方需要做的。我们甚至计划分析批量pod创建的机制（＃170）。

ReplicationController旨在成为可组合的构建块原语。我们希望在它和其他补充原语之上构建更高级别的API和/或工具，以便将来用户使用。 kubectl目前支持的“宏”操作（运行，缩放，滚动更新）是概念验证的例子。例如，我们可以想象像Asgard管理ReplicationControllers，自动缩放器，服务，调度策略，金丝雀发布等等。

**API对象**

ReplicationController是Kubernetes REST API中的顶级资源。

**ReplicationController的替代**

* ReplicaSet

ReplicaSet是下一代ReplicationController，它支持新的基于集合的标签选择器。 它主要被Deployment用作协调pod创建，删除和更新的机制。 请注意，我们建议使用Deployment而不是直接使用ReplicaSet，除非您需要自定义更新编排或根本不需要更新。

* Deployment(推荐)

Deployment是是一个更高级别的API对象，它以与kubectl rolling-update类似的方式更新其底层ReplicaSet及其Pod。 如果您需要此滚动更新功能，建议使用Deployment，因为与kubectl滚动更新不同，它们是声明性的，服务器端的，并具有其他功能

* 裸Pod

与用户直接创建Pod的情况不同，ReplicationController会替换因任何原因而被删除或终止的Pod，例如在节点故障或破坏性节点维护（例如内核升级）的情况下。因此，即使您的应用程序只需要一个Pod，我们也建议您使用ReplicaSet。 可以想象它与流程主管类似，只是它监控多个节点上的多个Pod，而不是单个节点上的单个进程。 ReplicationController将本地容器重新启动委派给节点上的某个代理程序（例如，Kubelet或Docker）。

* Job

对于预期会自行终止的Pod（即批处理作业），使用Job而不是ReplicationController

DaemonSet

使用DaemonSet代替提供机器级功能的Pods的ReplicationController，例如机器监视或机器日志记录。 这些Pod具有与机器生命周期相关的生命周期：Pod需要在其他Pod启动之前在机器上运行，并且当机器准备好重新启动/关闭时可以安全终止。

### 7.2.3 Deployments

Deployment为Pod和ReplicaSet提供声明性更新。

您在Deployment中描述了所需的状态，并且Deployment控制器以受控速率将实际状态更改为所需状态。 您可以定义Deployment以创建新的ReplicaSet，或者删除现有的部署并使用新的部署获取所有资源。

**使用案例**

下面是Deployment的典型使用案例:

* 创建Deployment，发布ReplicaSet。ReplicaSet在后台创建Pods并检查发布状态是否成功或失败。
* 更新Deployment的podtemplatespec来声明pod的新状态。创建一个新的ReplicaSet，Deployment以受控速率将Pod从旧ReplicaSet移动到新ReplicaSet。 每个新的ReplicaSet都会更新Deployment的修订版。
* 如果当前Deployment的状态不稳定，回滚到Deployment的更早修订版。每个回滚更新Deployment的修订版。
* 扩容Deployment来促进更多负载
* 暂停Deployment以将多个修复程序应用于其PodTemplateSpec，然后恢复它以启动新的部署。
* 使用部署的状态作为发布阻塞的指示符
* 如果不需要时清除旧的ReplicaSet

**创建Deployment**

下面是Deployment的例子。它创建一个ReplicaSet来运行3个nginx的Pod：

nginx-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

在这个例子中

* 一个名为nginx-deployment的Deployment被创建，名字由.metadata.name字段设置。
* Deployment创建3个副本的pod，由replicas字段设置。
* selector字段定义Deployment怎样发现被管理的pod。在这种情况下，您只需选择Pod模板（app：nginx）中定义的标签。 但是，只要Pod模板本身满足规则，就可以使用更复杂的选择规则。

注意：matchLabels字段是{key，value}对的映射。 matchLabels映射中的单个{key，value}等同于matchExpressions的元素，其键字段为“key”，运算符为“In”，而values数组仅包含“value”。 必须满足matchLabels和matchExpressions的所有要求才能匹配。

* template字段包含下面的子字段：
  + pod上标签app:nginx使用labels字段
  + pod模板特征或者.template.spec字段，设置pod运行一个容器nginx，该容器ngxin在Dockerhub上版本为1.7.9
  + 创建容器的名字用name字段设置为nginx

按照下面的步骤创建上面的Deployment：

在开始之前，确保你的k8S集群启动并运行：

1.创建Deployment并运行下面命令：

注意：您可以指定-record标志来编写在资源注释kubernetes.io/change-cause中执行的命令。 它对将来的检查非常有用。例如，要查看每个Deployment修订版中执行的命令。

kubectl apply -f nginx-deployment.yaml

2.运行kubectl get deployments检查是否Deployment被创建。如果Deployment处于正在被创建，打印结果类似如下：

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 0 0 0 1s

当你检查Deployment在集群中状态，下面字段显示：

* NAME：列举Deployment在集群中的名称
* DESIRED：显示应用程序的所需副本数，您在创建部署时定义这些副本。 这是理想的状态。
* CURRENT：显示当前正在运行的副本数量。
* UP-TO-DATE：显示已更新以实现所需状态的副本
* AVAILABLE：显示用户可以使用的应用程序副本数。
* AGE：显示应用程序运行的时间。

3.查看Deployment的发布状态：

kubectl rollout status deployment.v1.apps/nginx-deployment

输出结果类似如下：

waiting for rollout to finish: 2 out of 3 new replicas have been updated...

deployment.apps/nginx-deployment successfully rolled out

4.运行kubectl get deployments，几秒后，打印结果类似如下：

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 18s

请注意，Deployment已创建所有三个副本，并且所有副本都是最新的（它们包含最新的Pod模板）并且可用。

5. 要查看Deployment创建的ReplicaSet（rs），请运行kubectl get rs。 输出类似于：

NAME DESIRED CURRENT READY AGE

nginx-deployment-75675f5897 3 3 3 18s

请注意，ReplicaSet的名称始终格式为[DEPLOYMENT-NAME]-[RANDOM-STRING]。 随机字符串是随机生成的，并使用pod-template-hash作为种子。

6. 要查看为每个Pod自动生成的标签，请运行kubectl get pods --show-labels。 返回以下输出：

NAME READY STATUS RESTARTS AGE LABELS

nginx-deployment-75675f5897-7ci7o 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

nginx-deployment-75675f5897-kzszj 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

nginx-deployment-75675f5897-qqcnn 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

创建的ReplicaSet确保存在3个nginx的pod。

注意：您必须在Deployment中指定适当的选择器和Pod模板标签（在本例中为app：nginx）。 不要将标签或选择器与其他控制器（包括其他Deployment和StatefulSet）重叠。 Kubernetes不会阻止您重叠，如果多个控制器具有重叠的选择器，那么这些控制器可能会发生冲突并出现意外行为。

Pod模板的hash标签

注意：不要改变这个标签

Deployment控制器将pod-template-hash标签添加到Deployment创建或采用的每个ReplicaSet。

此标签可确保部署的子ReplicaSet不重叠。 它是通过散列ReplicaSet的PodTemplate，并使用结果哈希值作为标签值添加到ReplicaSet选择器、Pod模板标签以及ReplicaSet可能具有的任何现有Pod。

**更新Deployment**

注意：当且仅当部署的Pod模板（即.spec.template）发生更改时才会触发Deployment的部署，例如，如果更新模板的标签或容器镜像。 其他更新（例如扩展部署）不会触发部署。

按照下面的步骤更新Deployment：

1. 让我们更新nginx Pod以使用nginx：1.9.1镜像而不是nginx：1.7.9镜像

kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1

输出结果为

deployment.apps/nginx-deployment image updated

相应地，可以编辑Deployment并改变.spec.template.spec.containers[0].image，从nginx;1.7.9到nginx:1.7.9

kubectl edit deployment.v1.apps/nginx-deployment

输出结果类似：

kubectl edit deployment.v1.apps/nginx-deployment

2.查看发布状态

kubectl rollout status deployment.v1.apps/nginx-deployment

输出结果类似：

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

或者deployment.apps/nginx-deployment successfully rolled out。

获取关于Deployment的更多细节：

* 发布成功后，您可以通过运行kubectl get deployments来查看Deployment。 输出类似于：

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 36s

* 运行kubectl get rs以查看Deployment通过创建新的ReplicaSet并将其扩展到3个副本来更新Pod，以及将旧的ReplicaSet缩减为0个副本。

kubectl get rs

输出类似结果：

NAME DESIRED CURRENT READY AGE

nginx-deployment-1564180365 3 3 3 6s

nginx-deployment-2035384211 0 0 0 36s

* 运行get pods应该现在只显示新建的pod

kubectl get pods

输出结果类似：

AME READY STATUS RESTARTS AGE

nginx-deployment-1564180365-khku8 1/1 Running 0 14s

nginx-deployment-1564180365-nacti 1/1 Running 0 14s

nginx-deployment-1564180365-z9gth 1/1 Running 0 14s

下次要更新这些Pod时，只需再次更新Deployment的Pod模板。

Deployment确保在更新时只有一定数量的Pod已关闭。默认情况下，它确保至少所需Pod数量的25％已启动（最大25％不可用）。

Deployment还确保在超过所需数量的Pod之上创建一定数量的Pod。 默认情况下，它确保最多25％的所需Pod启动（最大波动25％）。

例如，如果仔细查看上面的Deployment，您将看到它首先创建了一个新的Pod，然后删除了一些旧的Pod，并创建了新的Pod。 在有足够数量的新Pod出现之前，它不会杀死旧的Pod，并且在足够数量的旧Pod被杀之前不会创建新的Pod。 它确保至少有2个Pod可用，并且总共有最多4个Pod。

* 获取Deployment详情

kubectl describe deployments

类似输出结果如下：

Name: nginx-deployment

Namespace: default

CreationTimestamp: Thu, 30 Nov 2017 10:56:25 +0000

Labels: app=nginx

Annotations: deployment.kubernetes.io/revision=2

Selector: app=nginx

Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx:1.9.1

Port: 80/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True NewReplicaSetAvailable

OldReplicaSets: <none>

NewReplicaSet: nginx-deployment-1564180365 (3/3 replicas created)

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ScalingReplicaSet 2m deployment-controller Scaled up replica set nginx-deployment-2035384211 to 3

Normal ScalingReplicaSet 24s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 1

Normal ScalingReplicaSet 22s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 2

Normal ScalingReplicaSet 22s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 2

Normal ScalingReplicaSet 19s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 1

Normal ScalingReplicaSet 19s deployment-controller Scaled up replica set nginx-deployment-1564180365 to 3

Normal ScalingReplicaSet 14s deployment-controller Scaled down replica set nginx-deployment-2035384211 to 0

在这里，您可以看到，当您第一次创建Deployment时，它创建了一个ReplicaSet（nginx-deployment-2035384211）并直接将其扩展到3个副本。 更新Deployment时，它创建了一个新的ReplicaSet（nginx-deployment-1564180365）并将其扩展为1，然后将旧的ReplicaSet缩小为2，这样至少有2个Pod可用，最多创建了4个Pod 。 然后，它继续使用相同的滚动更新策略向上和向下扩展新旧ReplicaSet。 最后，您将在新的ReplicaSet中拥有3个可用副本，并将旧的ReplicaSet缩减为0

3.滚动

每次Deployment控制器观察到新的Deployment时，都会创建一个ReplicaSet以启动所需的Pod。如果Deployment已更新，则控制其标签与.spec.selector匹配但其模板与.spec.template不匹配的Pod的现有ReplicaSet将按比例缩小。最终，新的ReplicaSet将缩放为.spec.replicas，并且所有旧的ReplicaSet都将缩放为0。

如果在更新Deployment同时现有的发布正在进行中，则Deployment会创建新的ReplicaSet并开始扩展，然后回滚之前正在扩展的ReplicaSet - 它会将其添加到旧列表中ReplicaSet并开始缩小它。

例如，假设您创建一个Deployment来创建nginx：1.7.9的5个副本，但是当创建nginx：1.7.9的仅3个副本时，更新Deployment以创建nginx：1.9.1的5个副本。在这种情况下，Deployment立即开始杀死它创建的3个nginx：1.7.9 Pod，并开始创建nginx：1.9.1 Pods。它不会等待nginx：1.7.9的5个副本被创建。

4. 标签选择器更新

通常不鼓励进行标签选择器更新，建议您事先规划选择器。 在任何情况下，如果您需要执行标签选择器更新，请务必小心谨慎，并确保您已掌握所有含义。

注意：在API版本的apps / v1中，Deployment的标签选择器在创建后是不可变的。

* 进行选择器添加操作时，要求使用新标签更新Deployment规范中的Pod模板标签，否则将返回验证错误。 此更改是非重叠的，这意味着新选择器不会选择使用旧选择器创建的ReplicaSet和Pod，从而导致孤立所有旧ReplicaSet并创建新的ReplicaSet。
* 进行选择器更新操作时，更改选择器键中的现有值 - 会导致与选择器添加时相同的行为
* 进行选择器删除操作时，从Deployment选择器中删除现有key- 不需要对Pod模板标签进行任何更改。 现有ReplicaSet不是孤立的，并且未创建新的ReplicaSet，但请注意，已删除的标签仍存在于任何现有的Pod和ReplicaSet中。

**回滚Deployment**

有时您可能想要回滚Deployment; 例如Deployment不稳定时，例如循环崩溃。默认情况下，所有Deployment发布的历史记录都保留在系统中，以便您可以随时回滚（可以通过修改版本历史记录限制来更改它的设置，比如保存多少历史版本）。

注意：在触发Deployment发布时创建Deployment的修订版。 这意味着当且仅当Deployment的Pod模板（.spec.template）发生更改时才会创建新修订版本，例如，如果更新模板的标签或容器镜像。 其他更新（例如扩展部署）不会创建部署版本，因此您可以方便地同时进行手动或自动扩展。 这意味着当您回滚到早期版本时，仅回滚Deployment的Pod模板部分。

* 假设您在更新Deployment时输入了拼写错误，方法是将镜像名称设置为nginx：1.91而不是nginx：1.9.1：

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=true

输出结果类似：

deployment.apps/nginx-deployment image updated

* 发布被卡住了。 您可以通过检查发布状态来验证它：

kubectl rollout status deployment.v1.apps/nginx-deployment

输出结果类似：

Waiting for rollout to finish: 1 out of 3 new replicas have been updated...

* ctrl+C结束查看发布状态
* 可以看到旧的两个副本nginx-deployment-1564180365 和nginx-deployment-2035384211和新的一个副本nginx-deployment-3066724191

kubectl get rs

输出结果类似：

NAME DESIRED CURRENT READY AGE

nginx-deployment-1564180365 3 3 3 25s

nginx-deployment-2035384211 0 0 0 36s

nginx-deployment-3066724191 1 1 0 6s

* 查看创建的Pod，您会看到新ReplicaSet创建的1 Pod停留在镜像循环下载中。

kubectl get pods

输出结果类似：

NAME READY STATUS RESTARTS AGE

nginx-deployment-1564180365-70iae 1/1 Running 0 25s

nginx-deployment-1564180365-jbqqo 1/1 Running 0 25s

nginx-deployment-1564180365-hysrc 1/1 Running 0 25s

nginx-deployment-3066724191-08mng 0/1 ImagePullBackOff 0 6s

Deployment控制器自动停止错误的发布，并停止扩展新的ReplicaSet。 这取决于您指定的rollingUpdate参数（特别是maxUnavailable）。 默认情况下，Kubernetes将值设置为25％。

* 获取Deployment的描述

kubectl describe deployment

输出结果类似

Name: nginx-deployment

Namespace: default

CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700

Labels: app=nginx

Selector: app=nginx

Replicas: 3 desired | 1 updated | 4 total | 3 available | 1 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx:1.91

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True ReplicaSetUpdated

OldReplicaSets: nginx-deployment-1564180365 (3/3 replicas created)

NewReplicaSet: nginx-deployment-3066724191 (1/1 replicas created)

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-2035384211 to 3

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 1

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 2

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 2

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 1

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 0

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 1

要解决此问题，您需要回滚到稳定的以前版本的Deployment

4.检查Deployment的发布历史

按照下面的步骤检查发布历史

1）首先检查Deployment的修订版

kubectl rollout history deployment.v1.apps/nginx-deployment

输出结果类似：

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 kubectl apply --filename=https://k8s.io/examples/controllers/nginx-deployment.yaml --record=true

2 kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

3 kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.91 --record=true

CHANGE-CAUSE从Deployment注释kubernetes.io/change-cause复制到创建时的修订版本。 您可以通过以下方式指定CHANGE-CAUSE消息：

* 注解Deployment

kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.9.1"

* 附加--record标志以保存正在更改资源的kubectl命令。
* 手动编辑资源的清单。

2）查看每个版本的详情

kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2

输出结果类似：

deployments "nginx-deployment" revision 2

Labels: app=nginx

pod-template-hash=1159050644

Annotations: kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

Containers:

nginx:

Image: nginx:1.9.1

Port: 80/TCP

QoS Tier:

cpu: BestEffort

memory: BestEffort

Environment Variables: <none>

No volumes.

5.回滚到之前版本

按照下面给出的步骤将部署从当前版本回滚到先前版本（版本2）

1）现在，您已决定撤消当前的发布并回滚到上一版本：

kubectl rollout undo deployment.v1.apps/nginx-deployment

类似结果

deployment.apps/nginx-deployment

或者，您可以通过使用--to-revision指定回滚到特定修订版本：

kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2

输出结果类似：

deployment.apps/nginx-deployment

Deployment现在回滚到以前的稳定版本。 如您所见，生成从Deployment控制器生成用于回滚到版本2的DeploymentRollback事件。

2）检查是否回滚成功，Deployment如期运行：

kubectl get deployment nginx-deployment

输出结果类似：

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 30m

3）获取Deployment的详情

kubectl describe deployment nginx-deployment

输出结果如下：

Name: nginx-deployment

Namespace: default

CreationTimestamp: Sun, 02 Sep 2018 18:17:55 -0500

Labels: app=nginx

Annotations: deployment.kubernetes.io/revision=4

kubernetes.io/change-cause=kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true

Selector: app=nginx

Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 25% max unavailable, 25% max surge

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx:1.9.1

Port: 80/TCP

Host Port: 0/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True NewReplicaSetAvailable

OldReplicaSets: <none>

NewReplicaSet: nginx-deployment-c4747d96c (3/3 replicas created)

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ScalingReplicaSet 12m deployment-controller Scaled up replica set nginx-deployment-75675f5897 to 3

Normal ScalingReplicaSet 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 1

Normal ScalingReplicaSet 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 2

Normal ScalingReplicaSet 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 2

Normal ScalingReplicaSet 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 1

Normal ScalingReplicaSet 11m deployment-controller Scaled up replica set nginx-deployment-c4747d96c to 3

Normal ScalingReplicaSet 11m deployment-controller Scaled down replica set nginx-deployment-75675f5897 to 0

Normal ScalingReplicaSet 11m deployment-controller Scaled up replica set nginx-deployment-595696685f to 1

Normal DeploymentRollback 15s deployment-controller Rolled back deployment "nginx-deployment" to revision 2

Normal ScalingReplicaSet 15s deployment-controller Scaled down replica set nginx-deployment-595696685f to 0

**缩放Deployment**

可以通过下面命令缩放一个Deployment

kubectl scale deployment.v1.apps/nginx-deployment --replicas=10

输出结果类似：

deployment.apps/nginx-deployment scaled

假设在集群中启用了HPA自动缩放，您可以为Deployment设置自动缩放器，并根据现有Pod的CPU利用率选择要运行的最小和最大Pod数。

kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --cpu-percent=80

输出结果类似：

deployment.apps/nginx-deployment scaled

1）比例缩放

RollingUpdate Deployments支持同时运行多个版本的应用程序。 当您或自动扩展器扩展一个（正在进行或暂停）RollingUpdate Deployment时，Deployment控制器会在现有活动ReplicaSet（带有pod的ReplicaSet）中平衡其他副本，以降低风险。 这称为比例缩放。

例如运行一个Deployment拥有10个副本，maxSurge=3，maxUnavailable=2

* 确保10个副本在Deployment中正在运行，kubectl get deployment,输出结果为

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 10 10 10 10 50s

* 您更新到一个新的镜像，该镜像恰好在集群内部无法解析。

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:sometag

输出结果类似：

deployment.apps/nginx-deployment image updated

* 镜像更新后使用新的发布ReplicaSet nginx-deployment-1989198191但由于您在上面提到的maxUnavailable要求而被阻止。 查看发布状态：

kubectl get rs

输出结果类似

NAME DESIRED CURRENT READY AGE

nginx-deployment-1989198191 5 5 0 9s

nginx-deployment-618515232 8 8 8 1m

* 然后出现一个新的部署扩展请求。 自动缩放器将Deployment副本增加到15.部署控制器需要决定在哪里添加这些新的5个副本。 如果您没有使用比例缩放，则所有5个都将添加到新的ReplicaSet中。 通过比例缩放，您可以在所有ReplicaSet上传播其他副本。 较大比例的转移到具有最多副本的ReplicaSets，较低比例的将转到具有较少副本的ReplicaSet。 任何剩余的副本都会添加到具有最多副本的ReplicaSet中。 零副本的ReplicaSet不会按比例放大。

在上面的示例中，3个副本添加到旧的ReplicaSet中，2个副本添加到新的ReplicaSet中。 假设新副本变得健康，发布过程最终应将所有副本移动到新的ReplicaSet。 要确认这一点，请运行：

```shell

kubectl get deploy

```

The output is similar to this:

```

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 15 18 7 8 7m

```

发布状态确认副本如何添加到每个副本集。

kubectl get rs

数据结果类似

```

NAME DESIRED CURRENT READY AGE

nginx-deployment-1989198191 7 7 0 7m

nginx-deployment-618515232 11 11 11 7m

```

**暂停和恢复Deployment**

您可以在触发一个或多个更新之前暂停部署，然后恢复它。 这允许您在暂停和恢复之间应用多个修复，而不会触发不必要的部署。

* 例如当一个部署刚被创建，获取Deployment详情

kubectl get deploy

输出结果类

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx 3 3 3 3 1m

获取发布状态

kubectl get rs

输出结果类似：

NAME DESIRED CURRENT READY AGE

nginx-2142116321 3 3 3 1m

* 运行下面命令暂停

kubectl rollout pause deployment.v1.apps/nginx-deployment

输出结果类似：

deployment.apps/nginx-deployment paused

* 更新Deployment的镜像

kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1

输出结果类似：

deployment.apps/nginx-deployment image updated

* 注意没有新发布开始

kubectl rollout history deployment.v1.apps/nginx-deployment

输出结果类似

deployments "nginx"

REVISION CHANGE-CAUSE

1 <none>

* 获取发布状态确保Deployment更新成功

kubectl get rs

输出结果类似：

NAME DESIRED CURRENT READY AGE

nginx-2142116321 3 3 3 2m

* 您可以根据需要进行更多更新，例如，更新将使用的资源：

kubectl set resources deployment.v1.apps/nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi

输出结果类似

deployment.apps/nginx-deployment resource requirements updated

暂停之前部署的初始状态将继续其功能，但只要部署暂停，部署的新更新将不会产生任何影响。

* 最后，恢复部署并观察一个新的ReplicaSet，提供所有新的更新：

kubectl rollout resume deployment.v1.apps/nginx-deployment

输出结果类似：

deployment.apps/nginx-deployment resumed

* 查看滚动状态直到完成为止。

kubectl get rs -w

输出结果类似：

NAME DESIRED CURRENT READY AGE

nginx-2142116321 2 2 2 2m

nginx-3926361531 2 2 0 6s

nginx-3926361531 2 2 1 18s

nginx-2142116321 1 2 2 2m

nginx-2142116321 1 2 2 2m

nginx-3926361531 3 2 1 18s

nginx-3926361531 3 2 1 18s

nginx-2142116321 1 1 1 2m

nginx-3926361531 3 3 1 18s

nginx-3926361531 3 3 2 19s

nginx-2142116321 0 1 1 2m

nginx-2142116321 0 1 1 2m

nginx-2142116321 0 0 0 2m

nginx-3926361531 3 3 3 20s

* 查看发布最新状态

kubectl get rs

输出结果类似：

NAME DESIRED CURRENT READY AGE

nginx-2142116321 0 0 0 2m

nginx-3926361531 3 3 3 28s

注意：您无法回滚暂停的部署直到恢复暂停后。

**Deployment状态**

部署在其生命周期中进入各种状态。 它在发布新的ReplicaSet时处于正在进行中，可以处于完成时，也可以无法进行。

**处理中的Deployment**

* 部署创建一个新的ReplicaSet。
* 部署正在扩展其最新的ReplicaSet。
* 部署正在缩减其旧的ReplicaSet。
* 新Pod已准备就绪或可用（至少准备MinReadySeconds）。

您可以使用kubectl rollout status监视Deployment的进度。

**完成后的Deployment**

Kubernetes在具有以下特征时将部署标记为完成：

* 与部署关联的所有副本都已更新为您指定的最新版本，这意味着您已请求的任何更新已完成。
* 可以使用与部署关联的所有副本。
* 没有旧的部署副本正在运行。

您可以使用kubectl rollout status检查部署是否已完成。 如果发布成功完成，则kubectlrollout status将返回零退出代码。

kubectl rollout status deployment.v1.apps/nginx-deployment

输出结果类似：

Waiting for rollout to finish: 2 of 3 updated replicas are available...

deployment.apps/nginx-deployment successfully rolled out

$ echo $?

0

**失败的Deployment**

您的部署可能会在尝试部署其最新的ReplicaSet时遇到困难，而无法完成。 这可能是由于以下一些因素造成的：

* 配额不足
* readiness探针失败
* 镜像拉取错误
* 权限不足
* 限制范围
* 应用程序运行时配置错误

检测此情况的一种方法是在部署规范中指定截止时间参数：

（.specpec.progressDeadlineSeconds）。 .spec.progressDeadlineSeconds表示部署控制器在指示（在Deployment状态中）部署进度已停止之前等待的秒数。

以下kubectl命令设置progressDeadlineSeconds，以使控制器10分钟后报告缺少部署进度：

kubectl patch deployment.v1.apps/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'

输出类似结果：

deployment.apps/nginx-deployment patched

超过截止日间后，Deployment控制器会将DeploymentCondition与以下属性一起添加到Deployment的.status.conditions：

* Type=Progressing
* Status=False
* Reason=ProgressDeadlineExceeded

注意：除了使用Reason = ProgressDeadlineExceeded报告状态条件之外，Kubernetes不对停顿的部署采取任何操作。更高级别的协调器可以利用它并相应地采取相应措施，例如，将部署回滚到其先前版本。

注意：如果您暂停部署，Kubernetes不会根据您指定的截止日期检查进度。 您可以安全地在部署和暂停期间暂停部署，而不会触发超出截止日间的条件。

由于您设置的超时时间较短或由于任何其他可被视为瞬态的错误，您可能会遇到部署的暂时性错误。 例如，假设您的配额不足。 如果您描述部署，您将注意到以下部分：

kubectl describe deployment nginx-deployment

输出结果类似：

<...>

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True ReplicaSetUpdated

ReplicaFailure True FailedCreate

<...>

如果运行kubectl get deployment nginx-deployment -o yaml，输出结果类似：

status:

availableReplicas: 2

conditions:

- lastTransitionTime: 2016-10-04T12:25:39Z

lastUpdateTime: 2016-10-04T12:25:39Z

message: Replica set "nginx-deployment-4262182780" is progressing.

reason: ReplicaSetUpdated

status: "True"

type: Progressing

- lastTransitionTime: 2016-10-04T12:25:42Z

lastUpdateTime: 2016-10-04T12:25:42Z

message: Deployment has minimum availability.

reason: MinimumReplicasAvailable

status: "True"

type: Available

- lastTransitionTime: 2016-10-04T12:25:39Z

lastUpdateTime: 2016-10-04T12:25:39Z

message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded quota:

object-counts, requested: pods=1, used: pods=3, limited: pods=2'

reason: FailedCreate

status: "True"

type: ReplicaFailure

observedGeneration: 3

replicas: 2

unavailableReplicas: 2

最终，一旦超出部署进度截止日间，Kubernetes将更新状态和进度条件的原因：

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing False ProgressDeadlineExceeded

ReplicaFailure True FailedCreate

您可以通过缩小部署，缩小可能正在运行的其他控制器或增加命名空间中的配额来解决配额不足的问题。 如果您满足配额条件，然后部署控制器完成“部署”发布，您将看到部署状态更新成功条件（Status = True和Reason = NewReplicaSetAvailable）。

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True NewReplicaSetAvailable

Type = Available ，Status = True表示您的部署具有最低可用性。 最低可用性由部署策略中指定的参数决定。 Type = Progress ，Status = True表示您的部署正处于发布阶段且正在进行中或已成功完成其进度并且所需的最小新副本可用（在我们的例子中，Reason = NewReplicaSetAvailable意味着部署已完成）。

您可以使用kubectl rollout status检查部署是否无法进展。 如果部署已超过进度截止期限，kubectl rollout status将返回非零退出代码。

kubectl rollout status deployment.v1.apps/nginx-deployment

输出结果类似：

Waiting for rollout to finish: 2 out of 3 new replicas have been updated...

error: deployment "nginx" exceeded its progress deadline

$ echo $?

1

操作失败的Deployment

适用于完成部署的所有操作也适用于失败的部署。 如果需要在Deployment Pod模板中应用多个调整，可以向上/向下扩展，回滚到以前的版本，甚至可以暂停它

**清除策略**

您可以在部署中设置.spec.revisionHistoryLimit字段，以指定要保留此部署的旧ReplicaSet数。 其余的将在后台进行垃圾收集。 默认情况下，它是10。

注意：将此字段显式设置为0将导致清理部署的所有历史记录，从而部署将无法回滚

**Canary Deployment**

如果要使用部署将多版本部署到用户或服务器的子集，则可以按照管理资源中描述的canary模式创建多个部署，每个版本一个

**编写Deployment特征**

与所有其他Kubernetes配置一样，部署需要apiVersion，kind和metadata字段。 有关使用配置文件的一般信息，请参阅部署应用程序，配置容器以及使用kubectl管理资源文档。

部署还需要.spec部分

Pod模板

.spec.template和.spec.selector是.spec唯一必需的字段。

.spec.template是Pod模板。 它与Pod具有完全相同的架构，除了它是嵌套的并且没有apiVersion或kind。

除了Pod的必填字段外，部署中的Pod模板还必须指定适当的标签和适当的重新启动策略。 对于标签，请确保不要与其他控制器重叠。 见选择器）。

只允许等于Always的.spec.template.spec.restartPolicy，如果未指定，则为默认值

Replicas

.spec.replicas是一个可选字段，用于指定所需Pod的数量。 默认为1

Selector

.spec.selector是一个必填字段，它指定此部署所针对的Pod的标签选择器。

.spec.selector必须与.spec.template.metadata.labels匹配，否则它将被API拒绝。

在API版本apps / v1中，.spec.selector和.metadata.labels如果未设置，则不会默认等同于.spec.template.metadata.labels。 所以必须明确设置它们。 另请注意，在apps / v1中创建部署后，.spec.selector是不可变的。

如果其模板与.spec.template不同，或者此类Pod的总数超过.spec.replicas，则部署可以终止其标签与选择器匹配的Pod。 如果Pod的数量小于所需的数量，它会启动有.spec.template的新Pod。

注意：您不应创建其标签与此选择器匹配的其他Pod，比如可以直接创建另一个Deployment，也可以创建另一个控制器，例如ReplicaSet或ReplicationController。 如果这样做，第一个部署认为它创建了这些其他Pod。 Kubernetes并没有阻止你这样做。

Strategy

spec.strategy指定用于替换旧Pod的策略。 .spec.strategy.type可以是“Recreate”或“RollingUpdate”。 “RollingUpdate”是默认值。

* Recreate

在.spec.strategy.type == Recreate时创建新的Pod之前，所有现有Pod都将被终止。

* Rolling Update

当.spec.strategy.type == RollingUpdate时，部署以滚动更新方式更新Pod。 您可以指定maxUnavailable和maxSurge来控制滚动更新过程

Max Unavailable

.spec.strategy.rollingUpdate.maxUnavailable是一个可选字段，指定更新过程中可用的最大Pod数。 该值可以是绝对数（例如，5）或所需Pod的百分比（例如，10％）。 通过四舍五入计算绝对数字的百分比。 如果.spec.strategy.rollingUpdate.maxSurge为0，则该值不能为0.默认值为25％。

例如，当此值设置为30％时，旧的ReplicaSet可以在滚动更新开始时立即按比例缩小到所需Pod的70％。 准备好新的Pod后，可以进一步缩小旧的ReplicaSet，然后扩展新的ReplicaSet，确保在更新期间始终可用的Pod总数至少为所需Pod的70％

Max Surge

.spec.strategy.rollingUpdate.maxSurge是一个可选字段，指定可以在所需数量的Pod上创建的最大Pod数。 该值可以是绝对数（例如，5）或所需Pod的百分比（例如，10％）。 如果MaxUnavailable为0，则该值不能为0.绝对数量是通过向上舍入的百分比计算的。 默认值为25％。

例如，当此值设置为30％时，可以在滚动更新开始时立即按比例放大新的ReplicaSet，这样旧的和新的Pod的总数不会超过所需Pod的130％。 一旦旧的Pod被杀死，新的ReplicaSet可以进一步扩展，确保在更新期间任何时间运行的Pod总数最多为所需Pod的130％

Progress Deadline Seconds

.spec.progressDeadlineSeconds是一个可选字段，它指定在系统报告部署失败进度之前要等待部署进度的秒数 - 表现为Type = Progressing，Status = False,Reason = ProgressDeadlineExceeded资源的状态。 部署控制器将继续重试部署。 将来，一旦实现自动回滚，部署控制器将在观察到这种情况后立即回滚部署。

如果指定，则此字段必须大于.spec.minReadySeconds

Min Ready Seconds

.spec.minReadySeconds是一个可选字段，它指定新创建的Pod应该准备好而不会使其任何容器崩溃的最小秒数，认为是可用的。 默认为0（Pod一旦准备好就会被视为可用）。 要了解有关何时认为Pod已准备就绪的详细信息，请参阅容器探测器。

Rollback To

字段.spec.rollbackTo已在API版本extensions / v1beta1和apps / v1beta1中弃用，并且在从apps / v1beta2的API版本开始不再受支持。 相反，应该使用kubectl rollout undo

Revision History Limit

部署的修订历史记录存储在它控制的ReplicaSet中。

.spec.revisionHistoryLimit是一个可选字段，指定要保留以允许回滚的旧ReplicaSet的数量。 这些旧的ReplicaSets消耗了etcd中的资源并且拥挤了kubectl get rs的输出。 每个Deployment修订版的配置都存储在其ReplicaSet中; 因此，一旦删除旧的ReplicaSet，您将无法回滚到该部署版本。 默认情况下，将保留10个旧的ReplicaSet，但其理想值取决于新部署的频率和稳定性。

更具体地说，将此字段设置为零意味着将清除所有具有0个副本的旧ReplicaSet。 在这种情况下，无法撤消新的“部署”发布，因为它的修订历史记录已清除

Paused

.spec.paused是一个可选的布尔字段，用于暂停和恢复部署。 暂停部署与未暂停部署之间的唯一区别是，暂停部署的PodTemplateSpec的任何更改都不会触发新的部署，只要它暂停即可。默认情况下，部署在创建时不会暂停

**Deployment的替代**

kubectl rolling update

kubectl滚动更新以类似的方式更新Pod和ReplicationControllers。 但建议使用部署，因为它们是声明性的，服务器端，并且具有其他功能，例如即使在滚动更新完成后回滚到任何先前的修订版本

### 7.2.4 StatefulSets

StatefulSet是用于管理有状态应用程序的工作负载API对象。

注意：Statefulset在1.9版本中GA，处于稳定版本。

管理一组Pod的部署和扩展，并保证这些Pod的排序和唯一性。

与Deployment类似，StatefulSet管理基于相同容器规范的Pod。 与部署不同，StatefulSet为其每个Pod维护一个严格标识。 这些pod是根据相同的规范创建的，但不可互换：每个pod都有一个持久的标识符，它在任何重新调度时都会保留。

StatefulSet以与任何其他Controller相同的模式运行。 您在StatefulSet对象中定义所需的状态，StatefulSet控制器进行必要的更新实现从当前状态到达那里。

**使用Statefulset**

Statefulset对于需要以下一个或多个条件的应用程序非常有用：

* 稳定，独特的网络标识符。
* 稳定，持久的存储。
* 有序，优雅的部署和扩展。
* 有序的自动滚动更新。

在上文中，对于Pod的（重新）调度，稳定与持久性同义。如果应用程序不需要任何稳定标识符或有序部署，删除或扩展，则应使用提供一组无状态副本的控制器部署应用程序。Deployment或ReplicaSet等控制器可能更适合您的无状态需求

**限制**

* StatefulSet在1.9版本之前是beta资源，在1.5之前的任何Kubernetes版本中都没有。
* 给定Pod的存储必须由PersistentVolume Provisioner根据请求的storage class进行配置，或者由管理员预先配置。
* 删除和/或缩放StatefulSet将不会删除与StatefulSet关联的卷。这样做是为了确保数据安全，这通常比自动清除所有相关的StatefulSet资源更有价值。
* StatefulSets目前要求Headless Service负责Pod的网络身份。您负责创建此服务。
* 删除StatefulSet时，StatefulSets不提供对pod终止的任何保证。要在StatefulSet中实现pod的有序和正常终止，可以在删除之前将StatefulSet缩小到0。
* 使用具有默认Pod Management Policy（OrderedReady）的Rolling Updates时，可能需要手动干预以修复损坏的状态。

**组件**

下面的示例演示了StatefulSet的组件

* 名为nginx的Headless Service用于控制网络域。
* 名为web StatefulSet，有一个Spec，表明有3个nginx容器副本的将在唯一的Pod中启动。
* volumeClaimTemplates将使用PersistentVolume Provisioner配置的PersistentVolumes提供稳定的存储。

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx # has to match .spec.template.metadata.labels

serviceName: "nginx"

replicas: 3 # by default is 1

template:

metadata:

labels:

app: nginx # has to match .spec.selector.matchLabels

spec:

terminationGracePeriodSeconds: 10

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "my-storage-class"

resources:

requests:

storage: 1Gi

**Pod选择器**

您必须设置StatefulSet的.spec.selector字段以匹配其.spec.template.metadata.labels的标签。 在Kubernetes 1.8之前，.spec.selector字段在省略时为默认值。 在1.8及更高版本中，如果未指定匹配的Pod Selector，则会在StatefulSet创建期间导致验证错误

**Pod标识符**

StatefulSet Pod具有唯一的标识，该标识由序数，稳定的网络标识和稳定的存储组成。标识符对Pod作严格限制，无论它（重新）安排在哪个节点上。

有序索引

对于具有N个副本的StatefulSet，StatefulSet中的每个Pod将被分配一个整数序数，从0到N-1，在Set上是唯一的。

稳定网络ID

StatefulSet中的每个Pod都从StatefulSet的名称和Pod的序号中获取其主机名。 构造的主机名的模式是$（statefulset name） - $（ordinal）。 上面的示例将创建三个名为web-0，web-1，web-2的Pod。 StatefulSet可以使用Headless Service来控制其Pod的域名。 此服务管理的域采用以下形式：$（service name）.$（namespace）.svc.cluster.local，其中“cluster.local”是集群域。 在创建每个Pod时，它将获得匹配的DNS子域，采用以下形式：$（podname）。$（governing service domain），其中governing service由StatefulSet上的serviceName字段定义。

如限制部分所述，您负责创建负责Pod的网络身份的Headless Service。

以下是Cluster Domain，Service name，StatefulSet名称以及它们如何影响StatefulSet的Pod的DNS名称的一些示例。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cluster Domain | Service (ns/name) | StatefulSet (ns/name) | StatefulSet Domain | Pod DNS | Pod Hostname |
| cluster.local | default/nginx | default/web | nginx.default.svc.cluster.local | web-{0..N-1}.nginx.default.svc.cluster.local | web-{0..N-1} |
| cluster.local | foo/nginx | foo/web | nginx.foo.svc.cluster.local | web-{0..N-1}.nginx.foo.svc.cluster.local | web-{0..N-1} |
| kube.local | foo/nginx | foo/web | nginx.foo.svc.kube.local | web-{0..N-1}.nginx.foo.svc.kube.local | web-{0..N-1} |

注意：集群域设置为cluster.local，除非进行了其他配置。

**稳定存储**

Kubernetes为每个VolumeClaimTemplate创建一个PersistentVolume。 在上面的nginx示例中，每个Pod将接收一个PersistentVolume，其StorageClass为my-storage-class，1 Gib为预配置存储。 如果未指定StorageClass，则将使用默认的StorageClass。 当Pod（重新）调度到节点上时，其volumeMounts将挂载与其PersistentVolume声明关联的PersistentVolumes。 请注意，删除Pods或StatefulSet时，不会删除与Pods的PersistentVolume声明关联的PersistentVolumes。 这必须手动完成。

**Pod名称标签**

当StatefulSet控制器创建Pod时，它会添加一个标签statefulset.kubernetes.io/pod-name，该标签值为Pod的名称。 此标签允许您将服务附加到StatefulSet中的特定Pod

**部署和扩展保证**

* 对于具有N个副本的StatefulSet，当部署Pod时，将按顺序从{0..N-1}开始创建它们。
* 当删除Pod时，它们将以{N-1..0}的相反顺序终止。
* 在将缩放操作应用于Pod之前，其所有前任必须是Running and Ready。
* 在Pod终止之前，其所有后继者必须完全关闭

StatefulSet不应指定pod.Spec.TerminationGracePeriodSeconds为0.这种做法是不安全的，强烈反对。有关进一步说明，请参阅强制删除StatefulSet Pods。

当创建上面的nginx示例时，将按web-0，web-1，web-2的顺序部署三个Pod。在web-0运行和就绪之前不会部署web-1，并且在web-1运行并准备好之前不会部署web-2。如果web-0失败，在web-1运行并准备就绪后，但在启动web-2之前，web-2将不会启动，直到web-0成功重新启动并变为Running and Ready。

如果用户通过修补StatefulSet以使replicas = 1来扩展已部署的示例，则Web-2将首先终止。在完全关闭并删除web-2之前，不会终止web-1。如果web-0在web-2终止并且完全关闭之后失败，但在web-1终止之前，web-1将不会终止，直到web-0运行并准备就绪。

Pod管理策略

在Kubernetes 1.7及更高版本中，StatefulSet允许您放宽其排序保证，同时通过其.spec.podManagementPolicy字段保留其唯一性和身份保证

OrderedReady Pod管理

OrderedReady 是statefulset的默认pod管理方式，实现的行为见上面描述。

Parallel Pod管理

Parallel Pod管理告诉Statefulset控制器并行启动或终止所有Pod，并且在启动或终止另一个Pod之前不等待Pod变为Running and Ready或完全终止。 此选项仅影响缩放操作的行为。 更新不受影响。

**更新策略**

在Kubernetes 1.7及更高版本中，StatefulSet的.spec.updateStrategy字段允许您为StatefulSet中的Pod配置和禁用容器，标签，资源请求/限制和注释的自动滚动更新。

OnDelete

OnDelete更新策略实现了遗留（1.6和先前）行为。 当StatefulSet的.spec.updateStrategy.type设置为OnDelete时，StatefulSet控制器不会自动更新StatefulSet中的Pod。 用户必须手动删除Pod才能使控制器创建新的Pod，以反映对StatefulSet的.spec.template所做的修改

RollingUpdate

RollingUpdate更新策略为StatefulSet中的Pod实现自动滚动更新。 当.spec.updateStrategy未指定时，这是默认策略。 当StatefulSet的.spec.updateStrategy.type设置为RollingUpdate时，StatefulSet控制器将删除并重新创建StatefulSet中的每个Pod。 它将以与Pod终止相同的顺序（从最大序数到最小序列）继续进行，一次更新一个Pod。 在更新其前任之前，它将等待更新的Pod正在运行并准备就绪。

* Partitions

通过指定.spec.updateStrategy.rollingUpdate.partition，RollingUpdate更新策略可以进行分区。 如果指定了分区，则更新StatefulSet的.spec.template时，将更新序列大于或等于该分区的所有Pod。 序号小于分区的所有Pod都不会更新，即使它们被删除，也会用之前的版本重新创建。 如果StatefulSet的.spec.updateStrategy.rollingUpdate.partition大于其.spec.replicas，则对其.spec.template的更新将不会传播到其Pod。在大多数情况下，您不需要使用分区，但如果您要进行更新，推发布金丝雀或执行分阶段发布，它们非常有用。

* Forced Rollback

使用具有默认Pod管理策略（OrderedReady）的Rolling Updates时，可能会进入需要手动干预进行修复的损坏状态。

如果将Pod模板更新为永远不会运行和就绪的配置（例如，由于错误的二进制或应用程序级配置错误），StatefulSet将停止发布并等待。

在这种状态下，仅将Pod模板恢复为良好配置是不够的。 由于已知问题，StatefulSet将继续等待损坏的Pod变为就绪（从未发生），然后它将尝试将其恢复为可工作的配置。

还原模板后，还必须删除StatefulSet已尝试使用错误配置运行的任何Pod。 然后，StatefulSet将开始使用恢复的模板重新创建Pod

### 7.2.5 DaemonSet

DaemonSet确保所有（或某些）节点运行Pod的副本。 随着节点添加到集群中，将添加Pod。 随着节点从节点中删除，这些Pod将被垃圾收集。 删除DaemonSet将清除它创建的Pod。

一些典型应用案例如下：

* 在每个节点上运行集群存储daemon程序，例如glusterd，ceph。
* 在每个节点上运行日志收集守护程序，例如fluentd或logstash。
* 在每个节点上运行节点监视守护程序，例如Prometheus Node Exporter，Sysdig Agent，collectd，Dynatrace OneAgent，AppDynamics Agent，Datadog代理，New Relic代理，Ganglia gmond或Instana Agent。

在一个简单的例子中，覆盖所有节点的一个DaemonSet将用于每种类型的守护进程。 更复杂的设置可能会为单一类型的守护程序使用多个DaemonSet，但具有不同的标志和/或不同的内存以及针对不同硬件类型的cpu请求。

**编写DaemonSet的规范**

创建一个DaemonSet

您可以在YAML文件中描述DaemonSet。例如下面的daemonset.yaml文件描述了一个运行fluentd-elasticsearch Docker镜像的DaemonSet：

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

namespace: kube-system

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

tolerations:

- key: node-role.kubernetes.io/master

effect: NoSchedule

containers:

- name: fluentd-elasticsearch

image: gcr.io/fluentd-elasticsearch/fluentd:v2.5.1

resources:

limits:

memory: 200Mi

requests:

cpu: 100m

memory: 200Mi

volumeMounts:

- name: varlog

mountPath: /var/log

- name: varlibdockercontainers

mountPath: /var/lib/docker/containers

readOnly: true

terminationGracePeriodSeconds: 30

volumes:

- name: varlog

hostPath:

path: /var/log

- name: varlibdockercontainers

hostPath:

path: /var/lib/docker/containers

创建基于yaml 的DaemonSet：

kubectl apply -f daemonset.yaml

必要的字段

与所有其他Kubernetes配置一样，DaemonSet需要apiVersion，kind和metadata字段。

DaemonSet还需要一个.spec部分。

Pod模板

.spec.template是.spec中必需的字段之一。

.spec.template是一个pod模板。 它与Pod具有完全相同的架构，除了它是嵌套的并且没有apiVersion或kind。

除了Pod的必填字段外，DaemonSet中的Pod模板还必须指定适当的标签（请参阅pod选择器）。

DaemonSet中的Pod模板必须具有等于Always的RestartPolicy，或者未指定，默认为Always

Pod选择器

.spec.selector字段是一个pod选择器。它的工作方式与Job的.spec.selector相同。

从Kubernetes 1.8开始，您必须指定与.spec.template标签匹配的pod选择器。当为空时，pod选择器将不再设置默认值。选择器默认与kubectl apply不兼容。此外，一旦创建了DaemonSet，其.spec.selector就无法变异。改变pod选择器可能导致Pod的无意孤立，并且发现它对用户来说是混乱的。

.spec.selector是一个由两个字段组成的对象：

matchLabels - 与ReplicationController的.spec.selector相同。

matchExpressions - 允许通过指定键，值列表以及与键和值相关的运算符来构建更复杂的选择器。

指定两者时，结果为AND。

如果指定了.spec.selector，则它必须与.spec.template.metadata.labels匹配。具有这些不匹配的配置将被API拒绝。

此外，您通常不应直接创建任何标签与此选择器匹配的Pod，可以通过另一个DaemonSet或通过其他控制器（如ReplicaSet）创建。否则，DaemonSet控制器会认为这些Pod是由它创建的。 Kubernetes不会阻止你这样做。您可能希望这样做的一种情况是在节点上手动创建具有不同值的Pod以进行测试。

在某些节点上运行Pod

如果指定.spec.template.spec.nodeSelector，则DaemonSet控制器将在与该节点选择器匹配的节点上创建Pod。 同样，如果指定.spec.template.spec.affinity，则DaemonSet控制器将在与该节点关联相匹配的节点上创建Pod。 如果您未指定任何一个，则DaemonSet控制器将在所有节点上创建Pod

**DaemonPod怎么被调度的**

由DaemonSet控制器进行调度（从1.12起默认情况下禁用）

通常，运行Pod的机器由Kubernetes调度程序选择。 但是，由DaemonSet控制器创建的Pod已经选择了机器（在创建Pod时指定了.spec.nodeName，因此调度程序会忽略它）。 因此：

DaemonSet控制器不检查节点的不可调度字段。

即使尚未启动调度程序，DaemonSet控制器也可以生成Pod，这可以帮助集群引导程序

默认由调度器调度（从1.12默认启用）

K8S v1.15 beta版

DaemonSet确保所有符合条件的节点都运行Pod的副本。 通常，Kubernetes调度程序选择Pod运行的节点。 但是DaemonSetPod是由DaemonSet控制器创建和调度的。 这引入了以下问题：

不一致的Pod行为：等待安排的正常Pod已创建并处于Pending状态，但DaemonSet pod未在Pending状态下创建。 这使用户感到困惑。

Pod抢占由默认调度程序处理。启用抢占后，DaemonSet控制器将在不考虑pod优先级和抢占的情况下制定调度决策。

ScheduleDaemonSetPods允许您使用默认调度程序而不是DaemonSet控制器来调度DaemonSet，方法是将NodeAffinity项添加到DaemonSetpod，而不是.spec.nodeName项。 然后使用默认调度程序将pod绑定到目标主机。 如果DaemonSet pod的节点亲和性已存在，则替换它。 DaemonSet控制器仅在创建或修改DaemonSetPod时执行这些操作，并且不对DaemonSet的spec.template进行任何更改。

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchFields:

- key: metadata.name

operator: In

values:

- target-host-name

此外，node.kubernetes.io/unschedulable:NoSchedule 容忍会自动添加到DaemonSet Pods。 在调度DaemonSet Pod时，默认调度程序会忽略unschedulable的节点。

污点和容忍

尽管Daemon Pods识别污点和容忍度，但根据相关特性，DaemonSet Pods会自动添加以下容忍。

|  |  |  |  |
| --- | --- | --- | --- |
| 容忍key | 影响 | 版本 | 描述 |
| node.kubernetes.io/not-ready | NoExecute | 1.13+ | 当存在诸如网络分区之类的节点问题时，不会驱逐DaemonSet pod |
| node.kubernetes.io/unreachable | NoExecute | 1.13+ | 当存在诸如网络分区之类的节点问题时，不会驱逐DaemonSet pod |
| node.kubernetes.io/disk-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/memory-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/unschedulable | NoSchedule | 1.12+ | DaemonSet pods可以通过默认调度程序容忍unschedulable属性 |
| node.kubernetes.io/network-unavailable | NoSchedule | 1.12+ | 使用主机网络的DaemonSet pod可以通过默认调度程序容忍网络不可用属性 |

**与Daemon Pod交互**

在DaemonSet中与Pod通信的一些可能模式是：

推送：DaemonSet中的Pod配置为将更新发送到另一个服务，例如统计数据库。 他们没有客户。

NodeIP和已知端口：DaemonSet中的Pod可以使用hostPort，以便可以通过节点IP访问pod。 客户端以某种方式知道节点IP列表，并按惯例了解端口。

DNS：使用相同的pod选择器创建无头服务，然后使用端点资源发现DaemonSets或从DNS检索多个A记录。

服务：使用相同的Pod选择器创建服务，并使用该服务在随机节点上访问守护程序。 （无法到达特定节点。）

更新DaemonSet

**更新Daemonset**

**DaemonSet的替代**