

Kubernetes中文文档

linfan

Published
with GitBook



目錄

Introduction	0
User's Guide	1
099-Cloud Native Deployments of Hazelcast using Kubernetes	2
第一章	3
第1.1节 什么是Kubernetes？	3.1
第二章	4
第2.1节 Kubernetes概览	4.1
第2.2节 创建Kubernetes集群	4.2
第2.3节 从本地环境起步	4.3
第2.4节 基于Docker本地运行Kubernetes	4.4
第2.6节 从本地运行k8s开始 v1.0	4.5
第2.7节 容器引擎	4.6
第2.8节 入门指南：完整的解决方案	4.7
第2.9节 Google Computer Engine入门	4.8
第2.10节 AWS EC2快速入门	4.9
第2.11节 在Azure上使用CoreOS和Weave的Kubernetes	4.10
第2.12节	4.11
第2.13节 从零开始	4.12
第2.14节	4.13
第2.15节 Juju上部署入门	4.14
第2.16节 Racksapec上部署入门	4.15
第2.17节 CloudStack入门指南	4.16
第2.18节 vSphere的入门指南	4.17
第2.19节 libvirt CoreOS的入门指南	4.18
第2.20节	4.19
第2.21节	4.20
第2.22节 离线安装（使用裸机和CoreOS系统）	4.21
第2.23节	4.22
第2.24节 从Ferdora入门Kubernetes	4.23
第2.25节 从CentOS入门Kubernetes	4.24

第2.26节 在Ubuntu物理节点上部署Kubernets	4.25
第2.27节 利用Docker安装多节点Kubernetes	4.26
第2.28节 如何在Mesos上运行Kubernetes	4.27
第三章	5
第3.1章	6
第3.1.1节 Kubernetes用户指南：应用程序管理	6.1
第3.1.2节	6.2
第3.1.3节	6.3
第3.1.4节	6.4
第3.1.5节 Pods	6.5
第3.1.6节 Labels	6.6
第3.1.7节 Replication Controller	6.7
第3.1.8节 Services in Kubernetes	6.8
第3.1.9节 Volumes	6.9
第3.1.10节	6.10
第3.1.11节 Secrets	6.11
第3.1.12节 Identifiers	6.12
第3.1.13节 Namespaces	6.13
第3.1.14节	6.14
第3.1.15节 Annotations	6.15
第3.1.16节	6.16
第3.1.17节	6.17
第3.1.18节	6.18
第3.1.19节	6.19
第3.1.20节	6.20
第3.1.21节	6.21
第3.1.22节	6.22
第3.1.23节 基础教程	6.23
第3.1.24节 Kubernetes 101 - Kubectl CLI和Pods	6.24
第3.1.25节 Kubernetes 201 - 标签,副本控制,服务和健康检查	6.25
第3.1.26节	6.26
第3.1.27节 配置kubernetes	6.27
第3.1.28节	6.28
第3.1.29节 管理应用：部署持续运行的应用	6.29

第3.1.30节 管理应用：连接应用	6.30
第3.1.31节 管理应用：在生产环境中使用Pods和容器	6.31
第3.1.32节 Kubernetes用户指南：管理应用：管理部署	6.32
第3.1.33节	6.33
第3.1.34节 Kubernetes UI	6.34
第3.1.35节	6.35
第3.1.36节	6.36
第3.1.37节 使用kubectl exec检查容器中的环境变量	6.37
第3.1.38节 应用：kubectl proxy和apiserver proxy	6.38
第3.1.39节 应用：kubectl port-forward	6.39
第3.2章	7
第3.2.1节 Kubernetes集群管理员手册	7.1
第3.2.2节 Kubernetes架构	7.2
第3.2.3节	7.3
第3.2.4节	7.4
第3.2.5节	7.5
第3.2.6节	7.6
第3.2.7节	7.7
第3.2.8节 Kubernetes集群管理指南：集群组件	7.8
第3.2.9节	7.9
第3.2.10节 Kube-API Server	7.10
第3.2.11节 授权插件	7.11
第3.2.12节 认证插件	7.12
第3.2.13节 API Server端口配置	7.13
第3.2.14节 Admission Controller	7.14
第3.2.15节 Service Accounts集群管理指南	7.15
第3.2.16节 Kube调度器	7.16
第3.2.17节	7.17
第3.2.18节	7.18
第3.2.19节	7.19
第3.2.20节	7.20
第3.2.21节 网络	7.21
第3.2.22节	7.22

第3.2.23节	7.23
第3.2.24节	7.24
第3.2.25节	7.25
第3.2.26节	7.26
第3.2.27节	7.27
第3.2.28节	7.28
第3.2.29节	7.29
第3.2.30节	7.30
第3.2.31节	7.31
第3.2.32节	7.32
第3.2.33节	7.33
第3.2.34节 从集群级别日志到Google云日志平台	7.34
第3.2.35节	7.35
第四章	8
第4.1章	9
第4.1.1节	9.1
第4.1.2节 使用Kubernetes在云上原生部署cassandra	9.2
第4.1.3节 Spark例子	9.3
第4.1.4节 Storm示例	9.4
第4.1.5节 示例: 分布式任务队列 Celery, RabbitMQ和Flower	9.5
第4.1.6节 Kubernetes在Hazelcast平台上部署原生云应用	9.6
第4.1.7节 Meteor on Kuberentes	9.7
第4.1.8节	9.8
第4.1.9节	9.9
第4.1.10节	9.10
第4.1.11节	9.11
第4.1.12节	9.12
第4.1.13节	9.13
第4.1.14节	9.14
第4.2章	10
第4.2.1节	10.1
第4.2.2节 配置文件使用入门	10.2
第4.2.3节 环境向导示例	10.3
第4.2.4节 Downward API范例	10.4

第4.2.5节	10.5
第4.2.6节	10.6
第4.2.7节	10.7
第4.2.8节	10.8
第4.2.9节 在Kubernetes上运行你的第一个容器	10.9
第4.2.10节 Kubernetes DNS 实例实战	10.10
第4.2.11节	10.11
第4.2.12节	10.12
第4.2.13节	10.13
第4.2.14节	10.14
第4.2.15节	10.15
第4.2.16节	10.16
第4.2.17节	10.17
第4.2.18节	10.18
第4.2.19节	10.19
第4.2.20节	10.20
第4.2.21节	10.21
第4.2.22节 滚动升级示例	10.22
第4.2.23节 explorer	10.23
第五章	11
第5.1章	12
第5.1.1节	12.1
第5.1.2节	12.2
第5.1.3节	12.3
第5.1.4节 运维API	12.4
第5.1.5节 参数说明	12.5
第5.1.6节	12.6
第5.1.7节	12.7
第5.2章	13
第5.2.1节	13.1
第5.2.2节 kubectl	13.2
第5.2.3节 kubectl annotate	13.3
第5.2.4节 kubectl api-versions	13.4

第5.2.5节 kubectl apply	13.5
第5.2.6节 kubectl attach	13.6
第5.2.7节 kubectl cluster-info	13.7
第5.2.8节 kubectl config	13.8
第5.2.9节 kubectl config set-cluster	13.9
第5.2.10节 kubectl config set-context	13.10
第5.2.11节 kubectl config set-credentials	13.11
第5.2.12节 kubectl config set	13.12
第5.2.13节 kubectl config unset	13.13
第5.2.14节 kubectl config use-context	13.14
第5.2.15节 kubectl config view	13.15
第5.2.16节 kubectl create	13.16
第5.2.17节 kubectl delete	13.17
第5.2.18节 kubectl describe	13.18
第5.2.19节 kubectl edit	13.19
第5.2.20节 kubectl exec	13.20
第5.2.21节	13.21
第5.2.22节	13.22
第5.2.23节	13.23
第5.2.24节 kubectl logs	13.24
第5.2.25节	13.25
第5.2.26节	13.26
第5.2.27节	13.27
第5.2.28节	13.28
第5.2.29节	13.29
第5.2.30节	13.30
第5.2.31节	13.31
第5.2.32节 kubectl version	13.32
第5.2.33节	13.33
第5.2.34节	13.34
第六章	14
第6.1节 故障排查	14.1
第6.2节	14.2
第6.3节 应用程序相关的故障排查	14.3

第3.2.34节 集群相关的故障排查	14.4
第6.4节 日常使用的常见问题	14.5
第6.5节 开发和调试的常见问题	14.6
第6.6节 服务相关的常见问题	14.7
第6.7节	14.8
第6.8节	14.9
第七章	15
第7.1节	15.1
第7.2节	15.2

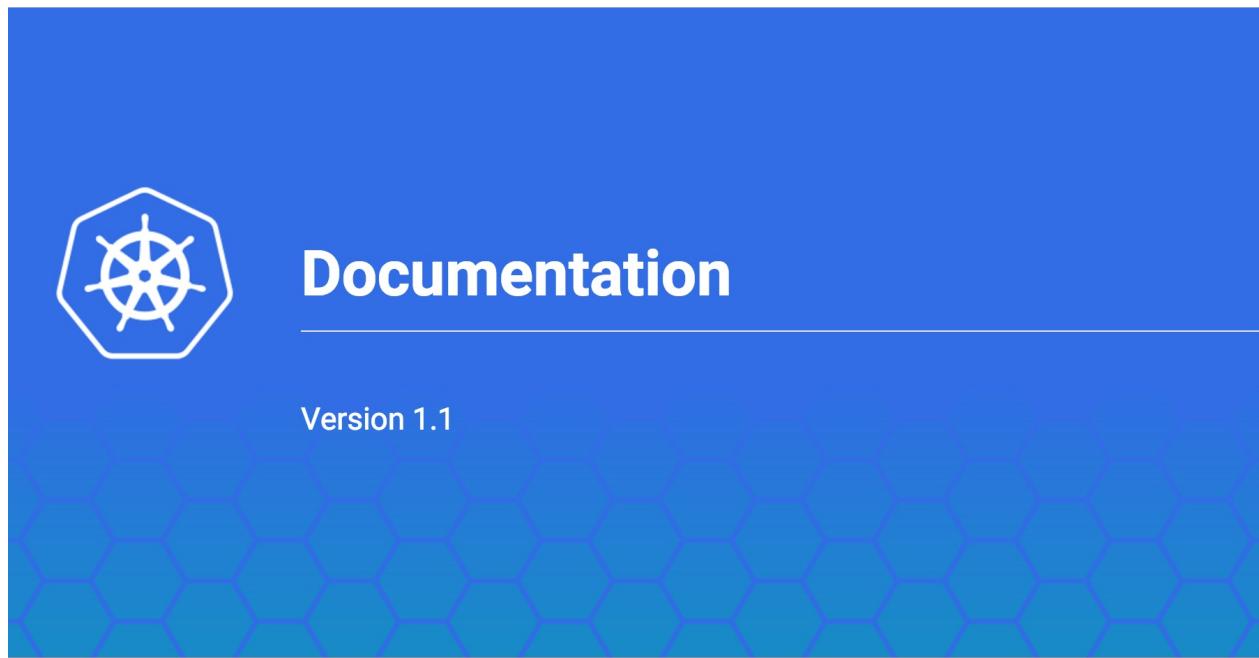
Kubernetes中文文档

Kubernetes文档的中文译本。

由DockOne社区组织翻译。

版本 1.1

- 原始文档链接：<http://kubernetes.io/v1.1/index.html>
- 最新文档GitHub目录：<https://github.com/kubernetes/kubernetes/tree/master/docs>



使用Vagrant

译者：卢文泉 校对：无

使用Vagrant（和VirtualBox）运行Kubernetes是在本地机器（Linux, Mac OS X）进行运行/测试/开发的简单方法。

预备知识

1. 从<http://www.vagrantup.com/downloads.html>下载最新版 vagrant >=1.6.2
2. 下载以下中的一项：
 - i. 从<http://www.vagrantup.com/downloads.html>下载4.3.28版本的VirtualBox
 - ii. 版本号>=5的VMWare Fusion和合适的Vagrant VMWare Fusion provider
 - iii. 版本号>=9的VMWare Workstation以及Vagrant VMWare Fusion provider
 - iv. 版本号>=9的Parallels Desktop以及Vagrant Parallels provider
 - v. 支持硬件虚拟化并且带有 libvirt 的KVM。Vagrant-libvirt，为fedora提供官方的rpm，可以使用 `yum install vagrant-libvirt` 命令安装

设置

简单运行下列代码就能够设置一个集群

```
export KUBERNETES_PROVIDER=vagrant
curl -sS https://get.k8s.io | bash
```

或者，你也可以下载[Kubernetes发行版](#)、解压归档文件。打开终端运行命令启动本地集群：

```
cd kubernetes

export KUBERNETES_PROVIDER=vagrant
./cluster/kube-up.sh
```

环境变量**KUBERNETES_PROVIDER**用来告诉所有不同的集群管理脚本该使用哪一个脚本管理器（比如Vagrant）。如果你忘记设置这个变量，默认假设你运行Google Compute Engine上运行k8s。

默认情况下，Vagrant会创建一个单独的master VM（被称为kubernetes-master），以及一个节点VM（被称为kubernetes-minion-1）。每个VM会占用1G内存，所以确保你有至少2G-4G的空余内存（以及合适的空闲硬盘空间）。

Vagrant会提供集群中每台机器运行Kubernetes所有必须的组件。每台机器会花费几分钟完成初始化设置。

如果你下载了多个Vagrant provider，Kubernetes通常会选择最恰当的那个。但是，你可以通过设置环境变量**VAGRANT_DEFAULT_PROVIDER**的值来让Kubernetes使用哪个Vagrant provider:

```
export VAGRANT_DEFAULT_PROVIDER=parallels
export KUBERNETES_PROVIDER=vagrant
./cluster/kube-up.sh
```

默认情况下每个集群中的VM运行Fedora系统。

通过下面的命令连接master或任意节点：

```
vagrant ssh master
vagrant ssh minion-1
```

如果你运行超过一个节点，可以通过下面命令连接其它节点：

```
vagrant ssh minion-2
vagrant ssh minion-3
```

集群中的每个节点会下载**docker daemon**和**kubelet**。

The master node instantiates the Kubernetes master components as pods on the machine. (这句翻译待讨论)

master节点会实例化Kubernetes master 组件作为 pods 运行在机器上。

查看kubernetes-master上的服务状态或者日志：

```
[vagrant@kubernetes-master ~] $ vagrant ssh master
[vagrant@kubernetes-master ~] $ sudo su

[root@kubernetes-master ~] $ systemctl status kubelet
[root@kubernetes-master ~] $ journalctl -ru kubelet

[root@kubernetes-master ~] $ systemctl status docker
[root@kubernetes-master ~] $ journalctl -ru docker

[root@kubernetes-master ~] $ tail -f /var/log/kube-apiserver.log
[root@kubernetes-master ~] $ tail -f /var/log/kube-controller-manager.log
[root@kubernetes-master ~] $ tail -f /var/log/kube-scheduler.log
```

查看任意节点的服务状态：

```
[vagrant@kubernetes-master ~] $ vagrant ssh minion-1
[vagrant@kubernetes-master ~] $ sudo su

[root@kubernetes-master ~] $ systemctl status kubelet
[root@kubernetes-master ~] $ journalctl -ru kubelet

[root@kubernetes-master ~] $ systemctl status docker
[root@kubernetes-master ~] $ journalctl -ru docker
```

使用Vagrant与kubernetes集群交互

启动好你的kubernetes集群后，你可以使用常用的Vagrant命令管理集群中的节点。

源代码改变后推送更新到Kubernetes代码：

```
./cluster/kube-push.sh
```

暂停与重启集群：

```
vagrant halt
./cluster/kube-up.sh
```

删除集群：

```
vagrant destroy
```

一旦你的Vagrant机器运行并且提供相应资源，第一件要做的事情就是检查能否使用**kubectl.sh**脚本。

你可能需要先编译二进制文件，可以使用make命令：

```
$ ./cluster/kubectl.sh get nodes

NAME          LABELS
10.245.1.4   <none>
10.245.1.5   <none>
10.245.1.3   <none>
```

验证你的master

当我们使用vagrant provider管理Kubernetes时，**cluster/kubectl.sh**脚本会创建证书保存在 `home` 或 `~` (ps : Linux中使用 `~` 表示当前用户的家目录) 目录下的**.kubernetes_vagrant_auth**文件中，这样以后就不会出现验证的提示了。

```
cat ~/.kubernetes_vagrant_auth
{
  "User": "vagrant",
  "Password": "vagrant",
  "CAFile": "/home/k8s_user/.kubernetes.vagrant.ca.crt",
  "CertFile": "/home/k8s_user/.kubecfg.vagrant.crt",
  "KeyFile": "/home/k8s_user/.kubecfg.vagrant.key"
}
```

现在你可以使用**cluster/kubectl.sh**脚本了。例如使用下面的命令列出已经启动的节点：

```
./cluster/kubectl.sh get nodes
```

运行容器

你的集群已经运行起来了，你可以列出集群中的节点：

```
$ ./cluster/kubectl.sh get nodes

NAME          LABELS
10.245.2.4   <none>
10.245.2.3   <none>
10.245.2.2   <none>
```

现在开始运行一些容器吧！

现在你可以用**cluster/kube-* .sh**的任何命令来与你的VMs交互。启动容器之前并没有 `pods`、`服务` 和 `复制控制器`：

```
$ ./cluster/kubectl.sh get pods
NAME      READY     STATUS    RESTARTS   AGE
$ ./cluster/kubectl.sh get services
NAME      LABELS      SELECTOR      IP(S)      PORT(S)
$ ./cluster/kubectl.sh get replicationcontrollers
CONTROLLER      CONTAINER(S)      IMAGE(S)      SELECTOR      REPLICAS
```

启动一个运行nginx、使用复制控制器并且副本数为3的容器：

```
$ ./cluster/kubectl.sh run my-nginx --image=nginx --replicas=3 --port=80
```

(此时) 列出pods, 会看到3个容器已经被启动了，并且处于等待状态：

```
$ ./cluster/kubectl.sh get pods
NAME      READY     STATUS    RESTARTS   AGE
my-nginx-5kq0g  0/1      Pending   0          10s
my-nginx-gr3hh  0/1      Pending   0          10s
my-nginx-xql4j  0/1      Pending   0          10s
```

你需要等待 (Vagrant) 分配好资源，这时可以通过命令来监测节点：

```
$ vagrant ssh minion-1 -c 'sudo docker images'
kubernetes-minion-1:
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTU
<none>          <none>  96864a7d2df3  26 hours ago  204.4
google/cadvisor  latest   e0575e677c50  13 days ago  12.64
kubernetes/pause  latest   6c4579af347b  8 weeks ago  239.8
```

下载好docker nginx镜像后容器就会启动，通过下面的命令查看：

```
$ vagrant ssh minion-1 -c 'sudo docker ps'
kubernetes-minion-1:
CONTAINER ID      IMAGE      COMMAND      CREATED      VIRTU
dbe79bf6e25b      nginx:latest  "nginx"      21 seconds ago  204.4
fa0e29c94501      kubernetes/pause:latest  "/pause"      8 minutes ago  12.64
aa2ee3ed844a      google/cadvisor:latest  "/usr/bin/cadvisor"  38 minutes ago  239.8
65a3a926f357      kubernetes/pause:latest  "/pause"      39 minutes ago  239.8
```

这时再次列举pods、服务和复制控制器，会看到：

```
$ ./cluster/kubectl.sh get pods
NAME          READY   STATUS    RESTARTS   AGE
my-nginx-5kq0g  1/1     Running   0          1m
my-nginx-gr3hh  1/1     Running   0          1m
my-nginx-xql4j  1/1     Running   0          1m

$ ./cluster/kubectl.sh get services
NAME      LABELS      SELECTOR      IP(S)      PORT(S)

$ ./cluster/kubectl.sh get replicationcontrollers
CONTROLLER      CONTAINER(S)      IMAGE(S)      SELECTOR      REPLICAS
my-nginx        my-nginx        nginx        run=my-nginx      3
```

我们没有启动任何服务，因此列举服务是什么也没有。但是我们看到3个副本正常显示。查看[guestbook](#)了解如何创建服务。现在你已经可以尝试增加或减少副本了：

```
$ ./cluster/kubectl.sh scale rc my-nginx --replicas=2
$ ./cluster/kubectl.sh get pods
NAME          READY   STATUS    RESTARTS   AGE
my-nginx-5kq0g  1/1     Running   0          2m
my-nginx-gr3hh  1/1     Running   0          2m
```

祝贺你！

解决常见问题

我一直在下载（很大的）**box** (**ps : box**指**vagrant**封装好的**vm**，可以下载直接使用，不用自己再从**iso**安装)

`Vagrantfile`默认从S3下载box。当调用**kube-up.sh**脚本时你可以提供name和可选的连接作为参数来修改下载站点：

```
export KUBERNETES_BOX_NAME=choose_your_own_name_for_your_kuber_box
export KUBERNETES_BOX_URL=path_of_your_kuber_box
export KUBERNETES_PROVIDER=vagrant
./cluster/kube-up.sh
```

我只是创建了集群，但是遇到授权错误

很可能你正试图连接的集群中的`~/.kubernetes_vagrant_auth`文件不正确：

```
rm ~/.kubernetes_vagrant_auth
```

使用**kubectl.sh**创建正确的证书：

```
cat ~/.kubernetes_vagrant_auth
{
  "User": "vagrant",
  "Password": "vagrant"
}
```

我已经创建了集群，但是看不到运行的容器！

如果是你第一次创建集群，每个节点上的**kubelet**会制定一个docker下载镜像和依赖镜像的时间表。这可能会占用很多时间并且推迟初始化pod、分配资源。

我想修改Kubernetes代码

你可以设置集群来 `hacking`，参考[vagrant developer guide](#)

已经运行**Vagrant up**命令，但是节点没有生效

```
( vagrant ssh minion-1 ) 连接到一个节点查看日志和salt minion日志 ( sudo cat
/var/log/salt/minion )
```

我想修改节点数

我们可以控制通过主机上环境变量**NUM_MINIONS**实例化的节点的数量。如果你计划使用副本，我们强烈支持你启动足够的节点来满足最大可能的副本大小。如果你不打算使用副本，你可以运行单个节点来节省一些系统资源。只需要设置**NUM_MINIONS**为1：

```
export NUM_MINIONS=1
```

我想给我的**VMs**分配更多内存

使用环境变量**KUBERNETES_MEMORY**控制分配给虚拟机的内存大小。只需要把它设置为你想要的大小，比如：

```
export KUBERNETES_MEMORY=2048
```

如果你想要更细粒度的控制内存，你可以单独设置master和节点的内存大小，比如：

```
export KUBERNETES_MASTER_MEMORY=1536
export KUBERNETES_MINION_MEMORY=2048
```

运行vagrant使VM待机，但是失效

vagrant suspend命令很可能弄混了网络。这种情况下不支持这条命令。

我想通过vagrant通过NFS同步文件

可以设置环境变量**KUBERNETES_VAGRANT_USE_NFS**的值为true来保证vagrant使用nfs同步虚拟机的文件夹。nfs比virtualbox和vmware的 共享文件夹 都要快并且不需要额外的设置。可以阅读[vagrant docs](#)获取在主机上配置nfs的细节。这项设置不会影响libvirt provider，它默认使用nfs：

```
export KUBERNETES_VAGRANT_USE_NFS=true
```

099-Cloud Native Deployments of Hazelcast using Kubernetes

Kubernetes在Hazelcast平台上的原生云部署

这篇文档主要是描述Kubernetes在Hazelcast平台上的原生云部署。当我们提到原生云的时，意味着我们当前的应用程序是运行在一个集群之上，同时使用这个集群的基础设施实现这个应用程序。值得注意的是，在此情况下，一个定制化的Hazelcast引导程序被用来使Hazelcast可以动态的发现已经加入集群的Hazelcast节点。当拓扑结构发生变化时，需要Hazelcast节点自身进行交流和处理。本文档同样也尝试去解释Kubernetes的核心组件：Pods、Service和ReplicationController。

前提

下面的例子假定你已经安装了Kubernetes集群并且可以运行。同时你也已经在你的路径下安装了kubectl命令行工具。可以从“[开始](#)”这个小节中得到相应平台的安装指令。

给急性子的备注

下面的介绍会有点长，如果你想跳过直接开始的话，请看结尾处的“[拓展部分](#)”。资源

资源

免费的资源如下：

Hazelcast Discovery - <https://github.com/pires/hazelcast-kubernetes-bootstrapper>

Dockerfile -<https://github.com/pires/hazelcast-kubernetes>

Docker Trusted Build - <https://quay.io/repository/pires/hazelcast-kubernetes>

简单的单调度单元的Hazelcast节点

在Kubernetes中，最小的应用单元就是[Pod](#)。一个Pod就是同一个主机调度下的一个或者多个容器。在一个Pod中的所有容器共享同一个网络命名空间，同时可以有选择性的共享同一个数据卷。在这种情况下，我们不能单独运行一个Hazelcast pod，因为它的发现机制依赖于Service的定义。

添加一个Hazelcast 服务

在Hazelcast 中，一个[Service](#)被描述为执行同一任务的Pods集合。比如，一个Hazelcast集群中的节点集合。Service 的一个重要用途就是通过建立一个均衡负载器将流量均匀的分到集合中的每一个成员。此外，Service还可以作为一个标准的查询器，使动态变化的POD集合提供有效通过Kubernetes 的API。实际上，这个就是探索机制的工作原理，就是在service的基础上去发现Hazelcast pods。下面是对Service的描述：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: hazelcast
  name: hazelcast
spec:
  ports:
    - port: 5701
  selector:
    name: hazelcast
```

这里值得注意的是[selector](#)（选择器）。在标签的上层有一个查询器，它标识了被Service所覆盖的Pods集合。在这种情况下，[selector](#)就是代码中 name: hazelcast 。在接下来的 Replication Controller说明书中，你会看到Pods中有对应的标签，那么它就会被这个Service 中对应的成员变量所选中。创建该Service的命令如下：`$ kubectl create -f examples/hazelcast/hazelcast-service.yaml`

添加一个拷贝节点

Kubernetes 和 Hazelcast 真正强大的地方在于它们可以轻松的建立一个可拷贝的、大小可调的Hazelcast集群。在Kubernetes 中，存在一个叫做[Replication Controller](#)的管理器，专门用来管理相同Pods的拷贝集合。和Service一样，它也存在一个在集合成员变量中定义的选择查询器。和Service不同的是，它对拷贝的个数有要求，会通过创建或者删除Pods来确保当前Pods的数量符合要求。Replication Controllers会通过匹配响应的选择查询器来确认要接收的Pods，下面我们将创建一个单拷贝的Replication Controller去接收已经存在的Hazelcast Pod。

```

apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: hazelcast
  name: hazelcast
spec:
  replicas: 1
  selector:
    name: hazelcast
  template:
    metadata:
      labels:
        name: hazelcast
    spec:
      containers:
        - resources:
            limits:
              cpu: 0.1
          image: quay.io/pires/hazelcast-kubernetes:0.5
          name: hazelcast
        env:
          - name: "DNS_DOMAIN"
            value: "cluster.local"
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        ports:
          - containerPort: 5701
            name: hazelcast

```

在这段代码中，有一些需要注意的东西。首先要注意的是，我们运行的是`quay.io/pires/hazelcast-kubernetes` image, tag 0.5。这个busybox安装在JRE8上。尽管如此，它还是添加了一个用户端的应用程序，从而可以发现集群中的Hazelcast节点并且引导一个Hazelcast实例。`HazelcastDiscoveryController`通过内置的搜索服务器来探索Kubernetes API Server，之后用Kubernetes API来发现新的节点。你可能已经注意到了，我们会告知Kubernetes，容器会暴露端口。最终，我们需要告诉集群的管理器，我们需要一个CPU核。对于Hazelcast pod而言，replication controller块的配置基本相同，以上就是声明，它只是给管理器提供一种简单的建立新节点的方法。其它的部分就是查询选择器和条件1中，配置下来的要求的POD数量。最后，我们需要根据你的Kubernetes集群的DNS配置来设置`DNS_DOMAIN`的环境变量。创建该控制器的命令：`$ kubectl create -f examples/hazelcast/hazelcast-controller.yaml` 当控制器成功的准备好后，你就可以查询服务端点：`$ kubectl get endpoints hazelcast -o json` { "kind": "Endpoints", "apiVersion": "v1", "metadata": { "name": "hazelcast", "namespace": "default", "selfLink": "/api/v1/namespaces/default/endpoints/hazelcast", "uid": "094e507a-2700-11e5-abbc-080027eae546", "resourceVersion": "4094", "creationTimestamp": "2015-07-10T12:34:41Z",

"labels": { "name": "hazelcast" }, "subsets": [{ "addresses": [{ "ip": "10.244.37.3", "port": 5701, "protocol": "TCP" }] }] } } 你可以看到Service 发现那些被replication controller建立的Pods。这下变的更加有趣了。让我们把集群提高到2个Pod。 \$ kubectl scale rc hazelcast --replicas=2 现在，如果你去列出集群中的Pods,你应该会看到2个 hazelcast pods: '\$ kubectl get pods' NAME READY STATUS RESTARTS AGE hazelcast-nanfb 1/1 Running 0 40s hazelcast-nsyzn 1/1 Running 0 2m kube-dns-xudrp 3/3 Running 0 1h 如果想确保每一个 Pods都在工作，你可以通过`log`命令来进行日志检查，如下：

```
$ kubectl log hazelcast-nanfb hazelcast
2015-07-10 13:26:34.443 INFO 5 --- [main] com.github.pires.hazelcast.ApplicationConfigApplicationConfigApplicationCo
2015-07-10 13:26:34.535 INFO 5 --- [main] s.c.a.AnnotationConfigApplicationConfigAnnotationConfigApplicationCo
2015-07-10 13:26:35.888 INFO 5 --- [main] o.s.j.e.a.AnnotationMBeanExporter
2015-07-10 13:26:35.924 INFO 5 --- [main] c.g.p.h.HazelcastDiscoveryController
2015-07-10 13:26:37.259 INFO 5 --- [main] c.g.p.h.HazelcastDiscoveryController
2015-07-10 13:26:37.404 INFO 5 --- [main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.405 INFO 5 --- [main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.415 INFO 5 --- [main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.852 INFO 5 --- [main] com.hazelcast.spi.OperationService
2015-07-10 13:26:37.879 INFO 5 --- [main] c.h.s.i.o.c.ClassicOperationExecutor
2015-07-10 13:26:38.531 INFO 5 --- [main] com.hazelcast.system
2015-07-10 13:26:38.532 INFO 5 --- [main] com.hazelcast.system
2015-07-10 13:26:38.533 INFO 5 --- [main] com.hazelcast.instance.Node
2015-07-10 13:26:38.534 INFO 5 --- [main] com.hazelcast.core.LifecycleService
2015-07-10 13:26:38.672 INFO 5 --- [cached1] com.hazelcast.nio.tcp.SocketConnect
2015-07-10 13:26:38.683 INFO 5 --- [cached1] c.h.nio.tcp.TcpIpConnectionManager
2015-07-10 13:26:45.699 INFO 5 --- [ration.thread-1] com.hazelcast.cluster.ClusterService

Members [2] {
    Member [10.244.37.3]:5701
    Member [10.244.77.3]:5701 this
}

2015-07-10 13:26:47.722 INFO 5 --- [main] com.hazelcast.core.LifecycleService
2015-07-10 13:26:47.723 INFO 5 --- [main] com.github.pires.hazelcast.ApplicationConfigApplicationConfigApplicationCo
```

接着是4个Pods: \$ kubectl scale rc hazelcast --replicas=4 然后通过刚才的操作去检查这4个成员是否连接。

拓展部分

对于那些急性子，下面是这一章所用到的所有命令：

```
# 建立一个service去跟踪所有的hazelcast nodes
kubectl create -f examples/hazelcast/hazelcast-service.yaml

#建立一个 replication controller去拷贝hazelcast nodes
kubectl create -f examples/hazelcast/hazelcast-controller.yaml

#升级成2个节点
kubectl scale rc hazelcast --replicas=2

#升级成4个节点
kubectl scale rc hazelcast --replicas=4
```

Hazelcast 的搜索源代码

[点击这里](#)

什么是Kubernetes？

译者：razr 校对：iT2afL0rd

Kubernetes一个用于容器集群的自动化部署、扩容以及运维的开源平台。

使用Kubernetes，你可以快速高效地响应客户需求：

- 动态地对应用进行扩容。
- 无缝地发布新特性。
- 反使用需要的资源以优化硬件使用。

我们希望培育出一个组件及工具的生态，帮助大家减轻在公有云及私有云上运行应用的负担。

Kubernetes是：

- 简洁的：轻量级，简单，易上手
- 可移植的：公有，私有，混合，多重云（multi-cloud）
- 可扩展的：模块化，插件化，可挂载，可组合
- 可自愈的：自动布置，自动重启，自动复制

Kubernetes项目是Google在2014年启动的。Kubernetes构建在[Google公司十几年的大规模高负载生产系统运维经验](#)之上，同时结合了社区中各项最佳设计和实践。

准备好开始了吗？

想了解为何要使用[容器技术](#)？

下面是一些关键点：

- 以应用程序为中心的管理：将抽象级别从在虚拟硬件上运行操作系统上升到了在使用特定逻辑资源的操作系统上运行应用程序。这在提供了PaaS的简洁性的同时拥有IaaS的灵活性，并且相对于运行[12-factor应用程序](#)有过之而无不及。
- 开发和运维的关注点分离：提供构建和部署的分离；这样也就将应用从基础设施中解耦。
- 敏捷的应用创建和部署：相对使用虚拟机镜像，容器镜像的创建更加轻巧高效。
- 持续开发，持续集成以及持续部署：提供频繁可靠地构建和部署容器镜像的能力，同时可以快速简单地回滚(因为镜像是固化的)。
- 松耦合，分布式，弹性，自由的[微服务](#)：应用被分割为若干独立的小型程序，可以被动态地部署和管理 -- 而不是一个运行在单机上的超级臃肿的大程序。
- 开发，测试，生产环境保持高度一致：无论是再笔记本电脑还是服务器上，都采用相同方

式运行。

- 兼容不同的云平台或操作系统上: 可运行与Ubuntu, RHEL, on-prem或者Google Container Engine, 覆盖了开发, 测试和生产的各种不同环境。
- 资源分离: 带来可预测的程序性能。
- 资源利用: 高性能, 大容量。

Kubernetes不是：

Kubernetes不是PaaS（平台即服务）。

- Kubernetes并不对支持的应用程序类型有任何限制。它并不指定应用框架, 限制语言类型, 也不仅仅迎合 [12-factor应用程序](#)模式. Kubernetes旨在支持各种多种多样的负载类型: 只要一个程序能够在容器中运行, 它就可以在Kubernetes中运行。
- Kubernetes并不关注代码到镜像领域。它并不负责应用程序的构建。不同的用户和项目对持续集成流程都有不同的需求和偏好, 所以我们分层支持持续集成但并不规定和限制它的工作方式。
- 另一方面, 确实有不少PaaS系统运行在Kubernetes之上, 比如[OpenShift](#)和[Deis](#)。同样你也可以将定制的PaaS系统, 结合一个持续集成系统再Kubernetes上进行实施: 只需生成容器镜像并通过Kubernetes部署。
- 由于Kubernetes运行在应用层而不是硬件层, 所以它提供了一些一般PaaS提供的功能, 比如部署, 扩容, 负载均衡, 日志, 监控, 等等。无论如何, Kubernetes不是一个单一应用, 所以这些解决方案都是可选可插拔的。

Kubernetes并不是单单的"编排系统"; 它排除了对编排的需要:

- "编排"的技术定义为按照指定流程执行一系列动作: 执行A, 然后B, 然后C。相反, Kubernetes有一系列控制进程组成, 持续地控制从当前状态到指定状态的流转。无需关注你是如何从A到C: 只需结果如此。这样将使得系统更加易用, 强大, 健壮和弹性。

Kubernetes这个名字是什么意思? k8s又是什么?

Kubernetes这个名字源自希腊语, 意思是“舵手”, 也是“管理者”, “治理者”等词的源头。k8s是Kubernetes的简称(用数字『8』替代中间的8个字母『ubernete』)。

Kubernetes概览

译者 : kz 校对 : 无

Kubernetes是一个能在集群中跨多主机管理容器化应用的的开源系统。Kubernetes的意在让部属容器化和基于微服务的应用变得简单但是强大。

Kubernetes提供了诸多机制来进行应用部署，调度，更新，维护和伸缩。一个Kubernetes的关键特性是能它能主动的管理容器来保证集群的状态不断地符合用户的期望状态。一个运维人员能够启动微服务，然后让调度器来找到合适的安置点。我们也想不断的改善工具和用户体验，让他们能通过如金丝雀部署等模式来放出应用。

Kubernetes支持Docker和Rocket容器，对其他的容器镜像格式和容器runtime会在未来加入。

尽管Kubernetes目前集中关注持续运行的无状态（如web服务器和内存中的对象缓存）和云原生状态的的应用（如NoSQL数据库），在很快的将来，也会支持所有的其他的在生产集群环境能常见的workload类型，例如批处理，流式处理，和传统的数据库。

Kubernetes中，所有的容器都运行在pod中，一个pod来容纳一个单独的容器，或者多个合作的容器。在后一种情况，pod中的容器被保证放置在同一个机器上，可以共享资源。一个pod也能包含零个或者更多的volume，volume是对一个容器私有的目录或者可以在pod中的容器间共享。对于用户每个创建的pod，系统会找一个健康运转并且有足够的容量的机器，然后开始将相应的容器在那里启动。如果一个容器失败，它会被Kubernetes的node agent自动重启，这个node agent被称作Kubelet。但是如果pod或者他的机器出故障，它不会被自动转移或者重启，除非用户也定义了一个replication controller，我们马上就会讲到它。

用户可以自己创建并管理pod，但是Kubernetes极大的简化了系统管理，它能让用户指派两个常见的跟pod相关的活动：基于相同的pod配置，部署多个pod副本，和创建替换的pod当一个pod或者它所在的机器发生故障的时候。Kubernetes的API对象用来管理这些行为的被称作replication controller，它用模板的形式定义了pod，然后系统根据模板实例化出一些pod（特别是由用户）。pod的副本集合可以共同组成一整个应用，一个微服务，或者在一个多层应用的一层。一旦pod创建好，系统会持续的监控他们的健康状态，和它们运行时所在的机器的健康状况。如果一个pod因为软件问题或者所在机器故障出现问题，replication控制器会自动在健康的机器上创建一个新的pod，来保证pod的集合处于一个期望的冗余水平。一个或者多个应用的多个pod能共享一个机器。注意一个replication控制器在点那个非副本pod的情况下也会被需要，如果用户想在pod或者其运行所在的机器出现故障的时候能重新创建。

能够方便的指代一个pod集合是一个常见需要用到的功能。例如，需要限制一个修改型操作到一个有限制的集合里，或者限制需要查询状态的pod集合范围。作为一个常见的机制，Kubernetes绝大多数的API对象都允许用户添加任意的key-value键值对，被称作label，并且

可以让用户用一系列的label选择器（对label的键值对查询）来限制API操作的目标。每一个资源也有一个字符串类型的键值可以被外部的工具来存储或者获取任意的元数据，被称作 annotations（备注）。

Kubernetes支持一种独特的网络模型。Kubernetes鼓励用扁平的地址空间，并且不会动态的分配端口，而是采用让用户可以选择任意合适自己的端口。为了实现这点，它给每一个pod分配了一个ip地址。

现代的Internal应用通常由层级的微服务构建而来，例如一组前端和分布式的内存内键值存储交互，和副本复制的存储服务交互。为了构建这种架构，Kubernetes提供了service的抽象，其提供了一稳定的IP地址和DNS名字，来对应一组动态的pod，例如一组构成一个微服务的pod。这个pod组是通过label选择器来定义的，因为可以指定任何的pod组。当一个运行在Kubernetes pod里的容器连接到这个地址时，这个连接会被本地的代理转发（称作kube proxy）。该代理运行在来源机器上。转发的目的地是一个相应的后端容器，确切的后端是通过round-robin的策略进行选择，以均衡负载。kube proxy也会追踪后端的pod组的动态变化，如当pod被位于新机器上的新的pod取代的时候，因而服务的IP和DNS名字不用改变。

每一个Kubernetes中的资源，如pod，都通过一个URI来被识别，并且有一个UID。URI中一个总要的组件是，对象的类型（如：pod），对象的名字，和对象的namespace（命名空间）。对于一个特定的对象类型，每一个名字在其命名空间都是独一无二的。在一个对象的名字没有带着命名空间的形式给出，那就是默认的命名空间。UID在时间和空间的范围都是唯一的。

创建Kubernetes集群

译者：razr 校对：钟健鑫

Kubernetes可以在多种平台运行，从笔记本电脑，到云服务商的虚拟机，再到机架上的裸机服务器。要创建一个Kubernetes集群，根据不同场景需要做的也不尽相同，可能是运行一条命令，也可能是配置自己的定制集群。这里我们将引导你根据自己的需要选择合适的解决方案。

选择正确的解决方案

如果你只是想试一试Kubernetes，我们推荐[基于Docker的本地方案](#)。

[基于Docker的本地方案](#)是众多能够完成快速搭建的[本地集群](#)方案中的一种，但是局限于单台机器。

当你准备好扩展到多台机器和更高可用性时，[托管](#)解决方案是最容易搭建和维护的。

[全套云端方案](#)只需要少数几个命令就可以在更多的云服务提供商搭建Kubernetes。

[定制方案](#)需要花费更多的精力，但是覆盖了从零开始搭建Kubernetes集群的通用建议到分步骤的细节指引。

本地服务器方案

本地服务器方案再一台物理机上创建拥有一个或者多个Kubernetes节点的单机集群。创建过程是全自动的，且不需要任何云服务商的账户。但是这种单机集群的规模和可用性都受限于单台机器。

本地服务器方案有：

- [本地Docker](#) (上手建议)
- [Vagrant](#) (任何支持Vagrant的平台：Linux, MacOS, 或者Windows。)
- [无虚拟机本地集群](#) (Linux)

托管方案

[Google Container Engine](#) 提供创建好的Kubernetes集群。

全套云端方案

以下方案让你可以通过几个命令就在很多IaaS云服务中创建Kubernetes集群，并且有很活跃的社区支持。

- [GCE](#)
- [AWS](#)
- [Azure](#)

定制方案

Kubernetes可以在云服务提供商和裸机环境运行，并支持很多基本操作系统。

如果你再如下的指南中找到了符合你需要的，可直接使用。某些指南可能有些过时，但是比起从零开始还是有不少参考价值。如果你确实因为特殊原因或因为想了解底层原理，想要从零开始搭建，可以试试参考[从零开始](#)指南。

如果你对在新的平台支持Kubernetes感兴趣，可以看看我们的[写新方案的建议](#)。

云

以下是上文没有列出的云服务商或云操作系统支持的方案。

- [AWS + coreos](#)
- [GCE + CoreOS](#)
- [AWS + Ubuntu](#)
- [Joyent + Ubuntu](#)
- [Rackspace + CoreOS](#)

私有虚拟机

- [Vagrant](#)（采用CoreOS和flannel）
- [CloudStack](#)（采用Ansible, CoreOS和flannel）
- [Vmware](#)（采用Debian）
- [juju.md](#)（采用Juju, Ubuntu和flannel）
- [Vmware](#)（采用CoreOS和flannel）
- [libvirt-coreos.md](#)（采用CoreO）
- [oVirt](#)
- [libvirt](#)（采用Fedora和flannel）
- [KVM](#)（采用Fedora和flannel）

裸机

- [Offline](#)（无需互联网，采用CoreOS和flannel）

- [fedora/fedora_ansible_config.md](#)
- [Fedora单节点](#)
- [Fedora多节点](#)
- [Centos](#)
- [Ubuntu](#)
- [Docker多节点](#)

集成

- [Kubernetes on Mesos \(采用GCE\)](#)

Table of Solutions

以下用表格形式列出上面的所有方案。

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms
GKE			GCE	docs	[✓][3] (F)
Vagrant	Saltstack	Fedora	flannel	docs	[✓][2] (F)
GCE	Saltstack	Debian	GCE	docs	[✓][1] (F)
Azure	CoreOS	CoreOS	Weave	docs	(C) (F) (C) (C) (C)
Docker Single Node	custom	N/A	local	docs	(F)
Docker Multi Node	Flannel	N/A	local	docs	(F)
Bare-metal	Ansible	Fedora	flannel	docs	(F)
Digital Ocean	custom	Fedora	Calico	docs	(C)
Bare-metal	custom	Fedora	<i>none</i>	docs	(F)
Bare-metal	custom	Fedora	flannel	docs	(C)
libvirt	custom	Fedora	flannel	docs	(C)
KVM	custom	Fedora	flannel	docs	(C)

Mesos/Docker	custom	Ubuntu	Docker	docs	(()
Mesos/GCE				docs	(()
AWS	CoreOS	CoreOS	flannel	docs	(()
GCE	CoreOS	CoreOS	flannel	docs	(()
Vagrant	CoreOS	CoreOS	flannel	docs	(()
Bare-metal (Offline)	CoreOS	CoreOS	flannel	docs	(()
Bare-metal	CoreOS	CoreOS	Calico	docs	(()
CloudStack	Ansible	CoreOS	flannel	docs	(()
Vmware		Debian	OVS	docs	(()
Bare-metal	custom	CentOS	<i>none</i>	docs	(()
AWS	Juju	Ubuntu	flannel	docs	(()
OpenStack/HPCloud	Juju	Ubuntu	flannel	docs	(()
Joyent	Juju	Ubuntu	flannel	docs	(()
AWS	Saltstack	Ubuntu	OVS	docs	(()
Azure	Saltstack	Ubuntu	OpenVPN	docs	(()
Bare-metal	custom	Ubuntu	Calico	docs	(()
Bare-metal	custom	Ubuntu	flannel	docs	(()
Local			<i>none</i>	docs	(()

libvirt/KVM	CoreOS	CoreOS	libvirt/KVM	docs	((
oVirt				docs	((
Rackspace	CoreOS	CoreOS	flannel	docs	((
any	any	any	any	docs	((

注意：以上表格按照支持级别和测试及使用的版本进行排序。

表格中列说明：

- **IaaS Provider** 是指提供Kubernetes运行环境的虚拟机或物理机（节点）资源的提供商。
- **OS** 是指节点上运行的基础操作系统。
- **Config. Mgmt** 是指节点上安装和管理Kubernetes软件的配置管理系统。
- **Networking** 是指实现[网络模型](#)的软件。*none* 表示只支持一个节点，或支持单物理节点上的虚拟机节点。
- **Conformance** 表示使用该种配置创建的集群是否通过了项目一致性测试，支持 Kubernetes v1.0.0的API和基本特性。
- **Support Levels (支持级别)**
 - **Project** : Kubernetes贡献者们经常使用该配置，所以通常最新的版本可使用。
 - **Commercial** : 某些厂商负责在自己的平台支持。
 - **Community** : 在社区中有活跃支持，但可能最新版本不适用。
 - **Inactive**: 对于初次使用Kubernetes的用户不推荐，并且有可能在将来被移除。
- **Notes** 说明，比如适用的Kubernetes版本。

从本地环境起步

译者：razr 校对：无

使用以下方案创建基于单台物理机且运行一到多个Kubernetes节点的单个集群。

参考[选择合适的方案获取更多信息。](#)

目录：

- [本地（基于Docker）](#)
- [Vagrant](#)
- [本地（无虚拟机）](#)

基于Docker本地运行Kubernetes

译者：razr 校对：无

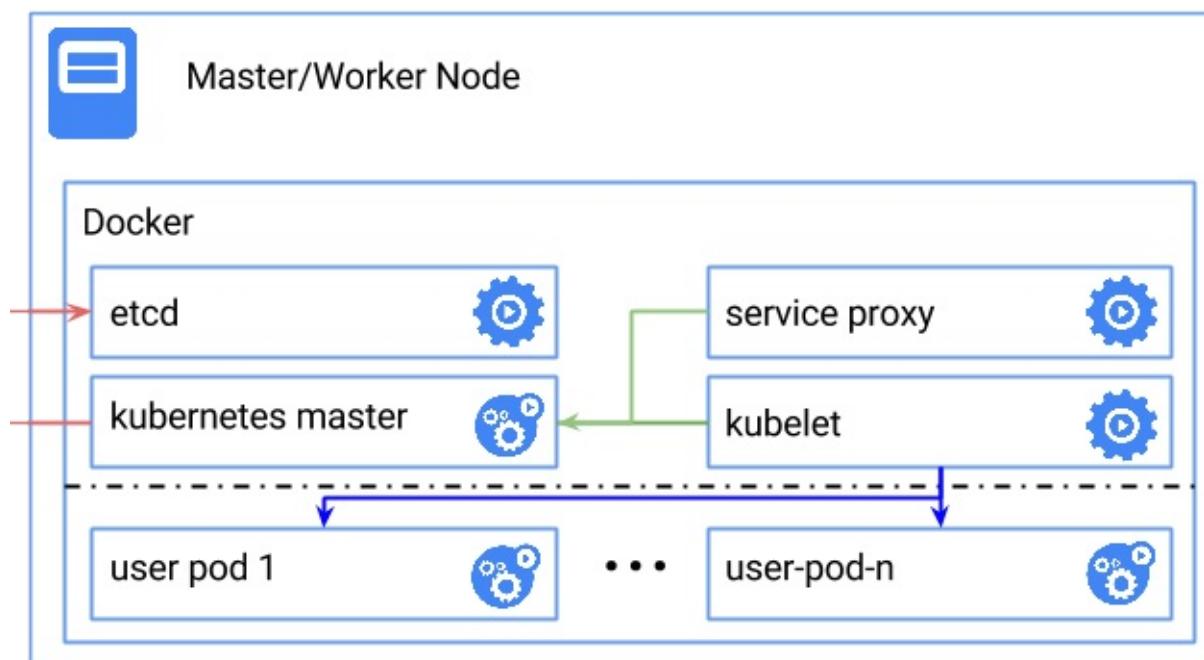
目录

- 概览
- 先决条件
- 第一步：运行etcd
- 第二步：启动master
- 第三步：启动service proxy
- 测试
- 运行一个应用
- 暴露为service
- 关于关闭集群的说明

概览

下面的指引将高速你如何通过Docker创建一个单机、单节点的Kubernetes集群。

下图是最终的结果：



先决条件

1. 你必须拥有一台安装有Docker的机器。

2. 你的内核必须支持 memory and swap accounting。确认你的linux内核开启了如下配置：

```
CONFIG_RESOURCE_COUNTERS=y
CONFIG_MEMCG=y
CONFIG_MEMCG_SWAP=y
CONFIG_MEMCG_SWAP_ENABLED=y
CONFIG_MEMCG_KMEM=y
```

3. 以命令行参数方式，在内核启动时开启 memory and swap accounting 选项：

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

注意：以上只适用于GRUB2。通过查看/proc/cmdline可以确认命令行参数是否已经成功传给内核：

```
$cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.18.4-aufs root=/dev/sda5 ro cgroup_enable=memory
swapaccount=1
```

第一步：运行Etcd

```
docker run --net=host -d gcr.io/google_containers/etcd:2.0.12 /usr/local/bin/etcd --addr=
```

第二步：启动master

```
docker run \
--volume=/:/rootfs:ro \
--volume=/sys:/sys:ro \
--volume=/dev:/dev \
--volume=/var/lib/docker/:/var/lib/docker:ro \
--volume=/var/lib/kubelet/:/var/lib/kubelet:rw \
--volume=/var/run:/var/run:rw \
--net=host \
--pid=host \
--privileged=true \
-d \
gcr.io/google_containers/hyperkube:v1.0.1 \
/hyperkube kubelet --containerized --hostname-override="127.0.0.1" --address="0.0.0.0
```

这一步实际上运行的是 kubelet，并启动了一个包含其他master组件的[pod](./user-guide/pods.md)。

第三步：运行service proxy

```
docker run -d --net=host --privileged gcr.io/google_containers/hyperkube:v1.0.1 /hyperkub
```



测试

此时你应该已经运行起了一个Kubernetes集群。你可以下载kubectl二进制程序进行测试：
[\(OS X\)](#) ([linux](#))

注意：再OS/X上你需要通过ssh设置端口转发：

```
boot2docker ssh -L8080:localhost:8080
```

列出集群中的节点：

```
kubectl get nodes
```

应该输出以下内容：

NAME	LABELS	STATUS
127.0.0.1		Ready

如果你运行了不同的Kubernetes集群，你可能需要指定 `-s http://localhost:8080` 选项来访问本地集群。

运行一个应用

```
kubectl -s http://localhost:8080 run nginx --image=nginx --port=80
```

运行 `docker ps` 你应该就能看到nginx在运行。下载镜像可能需要等待几分钟。

暴露为service

```
kubectl expose rc nginx --port=80
```

运行以下命令来获取刚才创建的service的IP地址。有两个IP，第一个是内部的(`CLUSTER_IP`)，第二个是外部的负载均衡IP。

```
kubectl get svc nginx
```

同样你也可以通过运行以下命令只获取第一个IP (CLUSTER_IP) :

```
kubectl get svc nginx --template={{.spec.clusterIP}}
```

通过第一个IP (CLUSTER_IP) 访问服务：

```
curl <insert-cluster-ip-here>
```

注意如果再OSX上需要再boot2docker虚拟机上运行curl。

关于关闭集群的说明

上面的各种容器都是运行在 `kubelet` 程序的管理下，它会保证容器一直运行，甚至容器意外退出时也不例外。所以，如果想关闭集群，你需要首先关闭 `kubelet` 容器，再关闭其他。

可以使用 `docker kill $(docker ps -aq)`。注意这样会关闭Docker下运行的所有容器，请谨慎使用。

从本地运行k8s开始 v1.0

译者：卢文泉 校对：无

[原文地址](#)

环境需求

Linux

没有运行Linux？考虑下使用[Vagrant](#)在虚拟机中运行Linux，或者像[Google Compute Engine](#)这样的云提供商上运行。

Docker

至少Docker1.3+。确保Docker守护进程一直运行，并确保能交互（比如 `docker ps`）。一些Kubernetes组件需要root权限运行，这样这些组件才能和Docker正常、良好地工作。

etcd

你需要配置[etcd](#)到环境变量，请确保安装etcd并且正确配置到\$PATH中。

go

go版本至少1.3+，请确保安装好go并配置好\$PATH。

启动集群

新打开一个单独的终端，运行下面的指令（由于启动/停止kubernetes守护进程需要root权限，所以使用root权限运行整个脚本会使操作更加容易）：

```
cd kubernetes  
hack/local-up-cluster.sh
```

这个操作将会构建和启动一个轻量的本地集群，包含一个master和一个节点。输入ctrl+C关闭集群。

运行容器

成功运行集群后，我想你很定迫不及待地想启动你的容器了。

现在你可以使用 `cluster/kubectl.sh` 脚本中的命令来和本地集群交互了：

```
cluster/kubectl.sh get pods
cluster/kubectl.sh get services
cluster/kubectl.sh get replicationcontrollers
cluster/kubectl.sh run my-nginx --image=nginx --replicas=2 --port=80

##在等待命令完成前，你可以打开一个新终端查看docker拉取镜像
sudo docker images
##你会看到docker正在拉去nginx镜像
sudo docker ps
## 你会看到你的容器正在运行
exit
## end wait

## 查看kubernetes相关信息
cluster/kubectl.sh get pods
cluster/kubectl.sh get services
cluster/kubectl.sh get replicationcontrollers
```

运行用户定义的pod

要注意[容器](#)和[pod](#)之前的不同。如果你只向kubernetes请求前者（容器），kubernetes会创建一个新的封装好的pod给你。但是（通过这个pod）你不能在本地主机查看到nginx的开始页面。为了验证nginx正确在容器中运行，你需要在容器中运行 `curl`（通过[docker exec](#)执行）。

你可以通过[用户定义的 manifest](#) 来控制pod的信息。指定端口就可以在浏览器中访问nginx了：

```
cluster/kubectl.sh create -f docs/user-guide/pod.yaml
```

祝你好运

解决问题

我不能通过IP访问服务

一些使用 `iptables` 工具的防火墙软件不能很好地与kubernetes配合。如果你在网络上遇到麻烦，首先尝试关闭系统的防火墙或者其它使用 `iptables` 的系统。此外，通过 `journalctl --since yesterday | grep avc` 指令检查SELinux（【译者注】指安全增强型Linux系统）是否屏

蔽了什么。

集群IPs默认范围：`10.0...`，这有可能引起docker 容器IP和集群IP冲突。如果你发现容器IP也在集群的IP范围内，编辑 `hack/local-cluster-up.sh` 脚本修改集群的IP范围。

当副本控制器的副本数大于1时我无法创建副本！哪里出了问题？

(也许是) 你只运行一个node节点。指超过了一个给定pod支持的最大副本数。如果你对运行更大副本感兴趣，我们鼓励你使用vagrant或者在云上操作。

我修改Kubernetes代码后该如何运行它？

```
cd kubernetes  
hack/build-go.sh  
hack/local-up-cluster.sh
```

kubectl表明启动容器但是 get pod 和 docker ps 命令没有显示

本地 `local-up-cluster.sh` 脚本不会启动DNS服务。类似的解决方案见[这里](#)。或者你可以手动启动。相关文档见[这里](#)

容器引擎

译者：李加庆 校对：无

自动化的容器管理

对于运行Docker容器来说，Googler容器引擎是一款强大的集群管理器和编排系统。容器引擎将你的容器牵引到集群中并会基于你定义的一些需求(比如CPU和内存)自动管理它们。容器引擎是基于开源的Kubernetes系统构建出来的，可以使你灵活地利用私有云、混合云或者公有云的基础设施所带来的好处。

分分钟建立集群

几分钟就可以建立一个可管理的虚拟机容器集群，并且随时可以部署。你的集群配备了很多功能，比如日志记录和容器健康检查，使得应用管理更加容易。

声明管理

在一个简单的JSON配置文件中声明你的容器需求，比如CPU/内存的预留量，副本数量，保活政策。容器引擎将设置你的容器为声明式的，并积极地管理你的应用程序，以确保满足要求。

灵活&开源

随着红帽，微软，IBM，Mirantis OpenStack以及VMware（名单数量不断增加中）将Kubernetes集成到它们的平台上，你就可以更加轻易地移动工作负载，或是充分利用多个云提供商带来的优势。

容器引擎的特点

全面管理

容器引擎由值得信赖的Google工程师全面管理，来确保您的集群是可用和最新的。

私有的容器Registry

Google容器Registry使得存储和访问您的私有Docker镜像更加容易。

可伸缩

随着您的应用需求的改变，轻松调整分配给容器的集群资源或者容器集群的大小。

Docker支持

容器引擎支持常见的Docker容器格式

日志功能

使用复选框来启用Google云日志，更加轻松地洞察应用程序的运行情况。

混合组网

为您的容器集群保留一组IP地址，通过Google云端VPN使得您的集群IP与私有网络IP共存。

对于我们来说，容器引擎充分发挥了**Google**基础设施的力量，而没有禁锢我们，它让我们平和地对待基础设施，并且让我们专注于开发伟大的软件。 - **Brian Fitzpatrick, Founder & CTO, Tock**

入门指南：完整的解决方案

译者：李加庆 校对：无

使用这些解决方案在您的云端构建Kubernetes集群。

第一次来？更多详情参见[选择合适的解决方案](#)。

目录：

- GCE
- AWS
- Azure

Google Computer Engine入门

译者：李加庆 校对：无

下面的例子用4个节点虚拟机和1个主虚拟机（也就是说集群中使用了5个虚拟机）创建了一个Kubernetes集群。您可以在您的工作站（或是任何您觉得合适的地方）安装和控制这个集群。

开始之前

如果您想要一个简化的入门体验和图形用户界面来管理集群，请考虑尝试使用谷歌容器引擎（GKE）安装和管理托管集群。如果您想使用自定义的二进制文件或者原生的开源Kubernetes，请看下面的说明。

前提条件

1. 您需要一个可以结算费用的Google云平台账户，更多细节详见[Google开发者控制台](#)。
2. 安装gcloud是必要的。**gcloud**可以作为[谷歌云SDK](#)的一部分来安装。
3. 然后，确保您已经安装了**gcloud preview**命令行组件。在命令行中运行**gcloud preview**。如果它要求安装任何组件，就按照提示安装。如果它只是显示帮助文本，那就大功告成了。这是必需的因为集群安装脚本使用**gcloud preview**命名空间中的GCE实例组，所以这一步是必要的。您还需要在开发控制台中启用[计算引擎实例组管理器API](#)。
4. 确保gcloud被设定使用您想使用的谷歌云平台项目。您可以使用**gcloud**配置清单项目检查当前的项目，并通过`gcloud config set project <project-id>`来更改它。
5. 确保您拥有基于gcloud身份验证登录的GCloud凭据。
6. 确保您可以通过命令行启动一个GCE虚拟机。至少确保您可以实践GCE快速入门的[创建实例](#)部分。
7. 确保您在没有交互式提示符的情况下可以ssh连接到VM。参阅GCE快速入门[登录实例](#)部分。

启动集群

您可以使用下面任意一条命令（以防万一。我们列出了两个，以防您的机器上只安装了某一个）来安装客户端并启动集群：

```
curl -sS https://get.k8s.io | bash
```

或者

```
wget -q -O - https://get.k8s.io | bash
```

此命令完成后，会有一个master虚拟机和四个worker虚拟机作为Kubernetes集群来运行。默认情况下，一些容器已经运行在集群上。一些容器如kibana和elasticsearch提供[日志记录](#)，而heapster提供[监控服务](#)。

上面提到的命令运行脚本创建一个名为或者前缀为“kubernetes”的集群。它定义了一个特定的集群配置，所以只能运行一次。

或者，您可以从[这个页面](#)下载并安装最新的Kubernetes发行版，然后运行/cluster/kube-up.sh脚本来启动集群：

```
cd kubernetescluster/kube-up.sh
```

如果您想在项目中运行不止一个集群，想使用不同的名字或者不同数量的工作节点，在启动集群之前参见[/cluster/gce/config-default.sh](#)文件来进行更细粒度的配置。

如果您遇到问题，请参见[故障排除](#)章节，给[Google容器组](#)发邮件或者在IRC freenode的[#google-containers](#)频道提问。

接下来的几个步骤将向您展示：

- 1.如何在您的工作站上安装命令行客户端来管理集群
- 2.如何使用集群的一些示例
- 3.如何删除群集
- 4.如何启动使用非默认选项的集群（如更大的集群）

在您的工作站中安装Kubernetes命令行工具

集群启动脚本会在工作站中留下一个正在运行的集群和一个**kubernetes**目录。下一步是要确保**kubectl**工具是在您的path中。

该**kubectl**工具控制Kubernetes集群管理器。它可以让您检查集群资源，创建、删除和更新组件以及更多功能。您会用它来查看新集群并生成示例应用程序。添加适当的二进制文件夹到您的path中以便可以访问kubectl：

```
# OS X  
export PATH=<path/to/kubernetes-directory>/platforms/darwin/amd64:$PATH#  
  
Linux  
export PATH=<path/to/kubernetes-directory>/platforms/linux/amd64:$PATH
```

注：gcloud还附带kubectl， 默认情况下被添加到您的path中。然而， gcloud所捆绑的kubectl版本可能会比通过get.k8s.io安装脚本下载的版本旧。我们建议您使用下载的二进制文件，以避免与客户机/服务器版本偏差所带来的潜在问题。

为bash配备Kubernetes命令行工具自动补全

您会发现让**kubectl**自动补全非常有用：

```
$ source ./contrib/completions/bash/kubectl
```

注：这种补全会在您的bash会话一直有效，如果您想要它永久有效，您需要在bash profile中添加这一行。另外的一个选择，在大多数linux发行版中，您也可以像下面这样复制完整的文件到您的 `bash_completions.d`：

```
$ cp ./contrib/completions/bash/kubectl /etc/bash_completion.d/
```

但是，你在更新kubectl的时候也要更新它。

启动您的集群

监控您的集群

一旦kubectl在您的path中，您就可以用它来查看您的集群。也就是运行：

```
$ kubectl get --all-namespaces services
```

应该会显示一组服务,看起来像这样：

```

NAMESPACE NAME LABELS SELECTOR IP(S) PORT(S)
default kubernetes component=apiserver,provider=kubernetes <none> 10.0.0.1 443/TCP
kube-system kube-dns k8s-app=kube-dns,kubernetes.io/cluster-service=true,kubernetes.io/name=kube-dns
kube-system kube-ui k8s-app=kube-ui,kubernetes.io/cluster-service=true,kubernetes.io/name=kube-ui
kube-system monitoring-grafana kubernetes.io/cluster-service=true,kubernetes.io/name=Grafana
kube-system monitoring-heapster kubernetes.io/cluster-service=true,kubernetes.io/name=Heapster
kube-system monitoring-influxdb kubernetes.io/cluster-service=true,kubernetes.io/name=InfluxDB

```

同样，您可以查看在集群启动时创建的pod。您可以这样做：

```
$ kubectl get --all-namespaces pods
```

您会看到如下所示的pod列表（名字细节会有所不同）：

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	fluentd-cloud-logging-kubernetes-minion-63uo	1/1	Running	0	14m
kube-system	fluentd-cloud-logging-kubernetes-minion-c1n9	1/1	Running	0	14m
kube-system	fluentd-cloud-logging-kubernetes-minion-c4og	1/1	Running	0	14m
kube-system	fluentd-cloud-logging-kubernetes-minion-ngua	1/1	Running	0	14m
kube-system	kube-dns-v5-7ztia	3/3	Running	0	15m
kube-system	kube-ui-v1-curt1	1/1	Running	0	15m
kube-system	monitoring-heapster-v5-ex4u3	1/1	Running	1	15m
kube-system	monitoring-influx-grafana-v1-piled	2/2	Running	0	15m

一些pod启动时可能会花费数秒（在此期间，它们会显示为挂起状态），但是在一小段时间后再次检查，它们就都是运行状态了。

运行一些例子

然后，通过一个简单的[nginx示例](#)来拿新的集群练练手。想要了解更多完整的应用程序，请查看[示例目录](#)。该[Guestbook示例](#)就是一个很好的“入门”演练。

拆除集群

使用**kube-down.sh**脚本来移除/删除/拆除集群：

```

cd kubernetes
cluster/kube-down.sh

```

同样地，同一目录下的**kube-up.sh**会做好备份。您无需重新运行curl或wget命令：安装Kubernetes集群所需要的一切都已在您的工作站中就绪。

定制集群

上面的脚本依赖于Google存储来暂存Kubernetes发行版。然后，它会启动（默认）单个master虚拟机以及4个worker虚拟机。您可以通过编辑 **kubernetes/cluster/gce/config-default.sh** 来调整一些参数，您可以在[这里](#)查看集群创建成功的记录。

故障排除

项目设置

您需要启用谷歌云存储API和谷歌云存储JSON API，新项目是默认启用的。如果没有启用，可以在谷歌云控制台完成。更多详情请参阅[谷歌云存储JSON API](#)

正如[前提条件](#)部分所列那样，还要确保地是，您已经启用计算引擎实例组管理器**API**，并且能够根据[GCE快速入门](#)中的指导那样，从命令行中启动一个GCE虚拟机。

集群初始化挂起

如果Kubernetes启动脚本挂起并等待API可用，您可以通过ssh方式连接到主虚拟机和节点虚拟机来查看日志如：

```
/var/log/startupscript.log
```

一旦您解决了这个问题，在再次尝试运行kube-up.sh之前，您应该在部分集群创建后运行kube-down.sh来做一下清理。

SSH

如果您无法通过SSH连接到您的实例，请确保GCE防火墙没有屏蔽虚拟机的22端口。默认情况下，是可以正常连接到实例的，但是，如果您编辑了防火墙规则或者创建了一个新的非默认的网络，您需要将其暴露：

```
gcloud compute firewall-rules create default-ssh --network=<network-name> --description "
```

此外，您的GCE SSH密钥必须要么没有密码，要么需要使用ssh-agent。

网络通信

该实例必须能够使用自己的私有IP连接到对方。该脚本使用“默认”网络，该网络有一个被称为“default-allow-internal”的防火墙规则，它允许私有IP地址上的任何端口跑流量。如果这个规则在默认网络中缺失或者您修改了在**cluster/config-default.sh**中正在使用的网络，那么使用下面的字段值创建一个新的规则：

- Source Ranges:10.0.0.0/8
- Allowed Protocols and Port:tcp:1-65535;udp:1-65535;icmp

AWS EC2快速入门

译者：李加庆 校对：无

前提条件

1. 您需要一个AWS账户，访问<http://aws.amazon.com>获得。
2. 安装并配置AWS命令行界面。
3. 你需要一个拥有EC2全部权限的AWS实例配置文件和角色。

注：这个脚本默认使用“默认”的AWS实例配置文件，您可以使用 `AWS_DEFAULT_PROFILE` 环境变量来明确地配置AWS实例配置文件：

```
export AWS_DEFAULT_PROFILE=myawsprofile
```

启动集群

支持程序：**get-kube**

```
#使用 wget
export KUBERNETES_PROVIDER=aws; wget -q -O - https://get.k8s.io | bash

#使用 curl
export KUBERNETES_PROVIDER=aws; curl -sS https://get.k8s.io | bash
```

注：这个脚本调用[cluster/kube-up.sh](#)，而[cluster/kube-up.sh](#)反过来使用[cluster/aws/config-default.sh](#)调用[cluster/aws/util.sh](#)。

这个过程需要约5至10分钟。一旦集群启动，你的主虚拟机和节点虚拟机的IP地址将被打印，同样地，有关运行在集群中的默认服务（监控，日志，DNS）的信息也会被打印。用户凭据和安全令牌都写在 `~/.kube/config` 中，它们对使用CLI或HTTP基本认证是必要的。

默认情况下，该脚本将会使用在美国西部-2A(俄勒冈州)运行了两个t2.micro实例的ubuntu，提供一个新的VPC和一个四节点的k8s集群。您可以根据下面的文本，重写定义在[config-default.sh](#)中的变量来改变这种默认的行为：

```

export KUBE_AWS_ZONE=eu-west-1c
export NUM_MINIONS=2
export MINION_SIZE=m3.medium
export AWS_S3_REGION=eu-west-1
export AWS_S3_BUCKET=mycompany-kubernetes-artifacts
export INSTANCE_PREFIX=k8s
...

```

该脚本也会尝试创建或者复用名为“kubernetes”的密钥对和名为“kubernetes-master”及“kubernetes-minion”的IAM文件。如果这些文件已经存在，请确保您想要在这里使用它们。

注：如果使用已存在的“kubernetes”密钥对，那么您必须设置AWS_SSH_KEY密钥指向您的私有密钥。

替代方案

这里提供一个[例子](#),可以让你使用EC2用户数据,建立一个基于CoreOS的Kubernetes集群。

开始使用您的集群

命令行管理工具：**kubectl**

集群启动脚本将会在您的工作站留下一个kubernetes目录。可以与之替代的是，您还可以从[这个页面](#)下载最新的Kubernetes发行版。接下来，在**PATH**中添加适当的二进制文件夹，以便可以访问kubectl：

```

# OS X export PATH=<path/to/kubernetes-directory>/platforms/darwin/amd64:$PATH
# Linux export PATH=<path/to/kubernetes-directory>/platforms/linux/amd64:$PATH

```

此工具的最新文档页面可以在这里找到：[kubectl manual](#)。默认情况下，**kubectl**将使用集群启动时生成的**kubeconfig**文件对API进行身份验证。更多相关信息，请阅读[kubeconfig文件](#)。

示例

看一个简单的[nginx示例](#)，尝试使用一下您的新集群。“Guestbook”应用程序是另外一个流行的Kubernetes入门示例：[guestbook 例子](#)。更多完整的应用程序,请查看[示例目录](#)。

拆除集群

确保您用来提供给集群的环境变量仍在输出，然后调用下面kubernetes目录中的脚本：

```
cluster/kube-down.sh
```

补充阅读

更多关于管理和使用Kubernetes集群的细节请参见[Kubernetes文档](#)。

在Azure上使用CoreOS和Weave的Kubernetes

译者：李加庆 校对：无

介绍

在本指南中我将演示如何在Azure云端部署Kubernetes集群。您将使用CoreOS与[Weave](#), Weave以透明而可靠的方式实现了简单、安全的网络。本指南的目的是提供一个即开即装即用的实现方法，以便最终可以稍加改变就可以投入到生产环境中。本文将演示如何提供一个专门的Kubernetes主节点和ETCD节点，并展示如何轻松地扩展集群。

前提条件

1. 您需要一个Azure账号。

开始吧！

开始之前，您需要checkout下代码：

```
git clone https://github.com/kubernetes/kubernetes  
cd kubernetes/docs/getting-started-guides/coreos/azure/
```

您需要在您的机器上安装Node.js，如果您之前使用过Azure CIL，那么您应该已经安装了。首先，您需要安装一些依赖：

```
npm install
```

现在，您需要做的是：

```
./azure-login.js -u <your_username>  
./create-kubernetes-cluster.js
```

这个脚本会提供适用于生产环境的集群，集群中有一个3个专用的ETCD节点形成环形：1个kubernetes主节点和2个kubernetes节点。**KUBE-00**虚拟机将是主节点，您的工作负荷只会部署在**KUBE-01**节点和**KUBE-02**节点上。最初，所有的虚拟机都是单核的，以确保自由层的用

户无需额外的代价就可以复制它。稍后我将展示如何添加更多更大的虚拟机。

图1 图2

一旦Azure虚拟机创建完成，你应该可以看到下面这样的信息：

```
...
azure_wrapper/info: Saved SSH config, you can use it like so: `ssh -F ./output/kube_1c14
azure_wrapper/info: The hosts in this deployment are:
[ 'etcd-00', 'etcd-01', 'etcd-02', 'kube-00', 'kube-01', 'kube-02' ]
azure_wrapper/info: Saved state into `./output/kube_1c1496016083b4_deployment.yml`
```

像下面这样登陆进主节点：

```
ssh -F ./output/kube_1c1496016083b4_ssh_conf kube-00
```

注：配置文件名字可能有所不同，确保使用你所看到的那个。

检查一下集群中的两个节点：

```
core@kube-00 ~ $ kubectl get nodes
NAME      LABELS
kube-01   kubernetes.io/hostname=kube-01
kube-02   kubernetes.io/hostname=kube-02
          STATUS
          Ready
          Ready
```

部署工作负载

现在让我们按照Guestbook的实例来部署：

```
kubectl create -f ~/guestbook-example
```

您需要等待pod部署完成，然后执行下面的命令，等待**STATUS**从**Pending**变为**Running**：

```
kubectl get pods --watch
```

注：大部分的时间将会花在下载每个节点的Docker容器镜像上。最后您将会看到：

NAME	READY	STATUS	RESTARTS	AGE
frontend-0a9xi	1/1	Running	0	4m
frontend-4wahe	1/1	Running	0	4m
frontend-6136j	1/1	Running	0	4m
redis-master-talmr	1/1	Running	0	4m
redis-slave-12zfd	1/1	Running	0	4m
redis-slave-3nbce	1/1	Running	0	4m

扩展

两个单核的节点肯定是无法满足现如今的生产系统，让我们通过添加几个更大的节点来扩展集群。您需要再打开一个您机器上的终端窗口，进入相同的工作目录（也就是说这个目录：`~/Workspace/kubernetes/docs/getting-started-guides/coreos/azure/`）首先，让我们设置一下新虚拟机的大小：

```
export AZ_VM_SIZE=Large
```

现在，我们使用先前部署的状态文件和添加的一系列节点来运行扩展脚本：

```
core@kube-00 ~ $ ./scale-kubernetes-cluster.js ./output/kube_1c1496016083b4_deployment.yml
...
azure_wrapper/info: Saved SSH config, you can use it like so: `ssh -F ./output/kube_8f98
azure_wrapper/info: The hosts in this deployment are:
[ 'etcd-00',
  'etcd-01',
  'etcd-02',
  'kube-00',
  'kube-01',
  'kube-02',
  'kube-03',
  'kube-04' ]
azure_wrapper/info: Saved state into `./output/kube_8f984af944f572_deployment.yml`
```

注：这一步在 `./output` 下产生了一些新文件。

回到**kube-00**：

```
core@kube-00 ~ $ kubectl get nodes
NAME      LABELS                                     STATUS
kube-01   kubernetes.io/hostname=kube-01           Ready
kube-02   kubernetes.io/hostname=kube-02           Ready
kube-03   kubernetes.io/hostname=kube-03           Ready
kube-04   kubernetes.io/hostname=kube-04           Ready
```

您可以看到又有两个节点顺利地加入进来，现在，让我们来扩展Guestbook实例的数量。

首先，再次检查一下有多少replication controller：

```
core@kube-00 ~ $ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)           SELECTOR
frontend     php-redis     kubernetes/example-guestbook-php-redis:v2  name=frontend
redis-master master       redis               name=redis-mast
redis-slave  worker       kubernetes/redis-slave:v2    name=redis-slav
```

基于现在有四个节点，让我们进行相应地扩展：

```
core@kube-00 ~ $ kubectl scale --replicas=4 rc redis-slave
>>>>> coreos/azure: Updates for 1.0
scaled
core@kube-00 ~ $ kubectl scale --replicas=4 rc frontend
scaled
```

现在，再来检查下：

```
core@kube-00 ~ $ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)           SELECTOR
frontend     php-redis     kubernetes/example-guestbook-php-redis:v2  name=frontend
redis-master master       redis               name=redis-mast
redis-slave  worker       kubernetes/redis-slave:v2    name=redis-slav
```

现在，您已经拥有了更多的前端Guestbook和redis slave实例。如果您查看一下所有 `name=frontend` 的节点，您会看到每个节点上都运行着一个实例。

```
core@kube-00 ~/guestbook-example $ kubectl get pods -l name=frontend
NAME        READY   STATUS    RESTARTS   AGE
frontend-0a9xi  1/1    Running   0          22m
frontend-4wahe  1/1    Running   0          22m
frontend-6l36j  1/1    Running   0          22m
frontend-z9oxo  1/1    Running   0          41s
```

将应用暴露给外部

Kubernetes 1.0 中没有原生的Azure负载均衡器，不过，下面演示了如何将Guestbook应用暴露给Internet。

```
./expose_guestbook_app_port.sh ./output/kube_1c1496016083b4_ssh_conf
Guestbook app is on port 31605, will map it to port 80 on kube-00
info: Executing command vm endpoint create
+ Getting virtual machines
+ Reading network configuration
+ Updating network configuration
info: vm endpoint create command OK
info: Executing command vm endpoint show
+ Getting virtual machines
data: Name : tcp-80-31605
data: Local port : 31605
data: Protcol : tcp
data: Virtual IP Address : 137.117.156.164
data: Direct server return : Disabled
info: vm endpoint show command OK
```

然后，您就可以通过上面所展示的**kube-00**的Azure虚拟ip（在我的例子中，也就是指<http://137.117.156.164/>），在任何地方连接它。

接下来

现在您已经拥有一个运行在Azure上规模性的集群，祝贺！

或许，您应该尝试部署其他的[应用示例](#)，或者动手写一个自己的。

移除...

如果您不希望顾虑Azure的费用问题，您可以移除集群。正如您看到的，移除它非常简单：

```
./destroy-cluster.js ./output/kube_8f984af944f572_deployment.yaml
```

注：确保使用最新的状态文件，因为扩展之后生成了新的文件。顺便说一下，如果您喜欢，您可以使用文中所示脚本部署多个集群。

从零开始

译者：王乐 校对：无

这部文档是面对想要订制Kubernetes集群的读者。如果你发现现有的入门指南已经可以满足你对[这个列表](#)上所列的需求，我们建议你继续阅读这个根据前人积累经验所写的新手指南。但如果你有入门指南所不能满足的对IaaS，网络，配置管理或对操作系统有特殊要求，这个指南将会提供给你一个指导性的概述。这个指南也会对希望对现有集群配置有进一步理解的读者提供帮助。

设计和准备

学习

1. 你应该已经熟悉Kubernetes了。我们建议根据其他入门指南架设一个临时的集群。这样可以帮助你先熟悉Kubernetes命令行(`kubectl`)和一些概念(`pods`, `services`, 等等)。
2. 当你浏览完其他入门指南的时候，你应该已经安装好了`kubectl`。如果没有，你可以根据[这个](#)说明安装。

Cloud Provider

Kubernetes有一个概念叫Cloud Provider，是指一个提供管理TCP负载均衡，节点（实例）和路由器接口的模块。`pkg/cloudprovider/cloud.go`里具体定义了这个接口。当然，你也可以不用实现这个Cloud Provider去新建一个自定义的聚群（例如，使用裸机集群）。取决于不同部件是如何设置的，并不是所有接口都需要实现的。

节点

- 你可以使用虚拟机或物理机。
- 为了运行本指南给出的例子和测试用例，你最好有4个节点以上。
- 虽说很多入门指南讲主节点和普通节点做了区分，但严格意义上讲，这不是必要的。
- 节点需要运行在x86_64的Linux系统上。当然也可能运行在其他的系统和CPU架构上，但这个指南不会提供相关的帮助。
- 对于一个拥有10个以上节点的集群来说，Apiserver和etcd可以一起安装在一台1核和1GB内存的机子上。
- 你可以给其他节点分配合理的内存和CPU内核。不是所有节点需要同样的配置。

网络

Kubernetes有一个独特的[网络模型](#)。

Kubernetes给每一个pod分配IP地址。当你新建一个集群，为了保证Pod获得IP地址，你需要给Kubernetes分配一个IP地址池。最简单的做法是每当

节点与节点之间的通信可以以一下两种方式实现：

- 配置网络完成Pod的IP地址路由
 - 因为一切从头开始，所以难度会大一些。
 - Google Compute Engine ([GCE](#)) 和 [AWS](#)会指导如何使用这种方式
 - 需要编程配置路由器和交换机去实现Pod的IP地址路由。
 - 可在Kubernetes环境外配置或者通过在Cloud Provider模块的“路由”接口里实现。
 - 通常情况下，提供最优网络性能。
- 建立一个拓扑网络
 - 较容易建立
 - 因为数据流量被封装，所以每个Pod的IP地址是可以路由的。
 - 例如：
 - [Flannel](#)
 - [Weave](#)
 - [Open vSwitch \(OVS\)](#)
 - 不需要CLoud Provider模块里的“路由”部分
 - 较为不太理想的性能（具体的性能弱化取决于你的实际情况）

你需要为Pod所需要的IP地址选一个IP地址范围。

- 一些可选择配置方式：
 - GCE：每一个项目有一个自己的IP子网“10.0.0.0/8”。项目中的每一个Kubernetes集群从中获得一个“/16”的子网。每一个节点从'16'的子网里获IP地址。
 - AWS：在一个组织内使用一个VPC。从中分配地址给每一个聚群或者使用给不同的集群分配不同的VPC。
 - 暂不支持支持IPv6
- 给每一的node的Pod地址分配一个CIDR子网或者一个
 - 你总共需要`max-pods-per-node * max-number-of-nodes`个IP地址。一个“/24”如果缺少IP地址，一个“/26”（62个节点）或者“/27”（30个节点）也能满足。
 - 例如，使用“10.10.0.0/16” “10.10.0.0/24” “10.10.255.0/24”
 - 需要路由设置或连接到拓扑网络

Kubernetes 会给每个[service](#)分配一个IP地址。但是service的地址并不一定是可路由的。当数据离开节点时，`kube-proxy`需要将Service IP地址翻译成Pod IP地址。因此，你需要给Service也分配一个地址段。这个网段叫做“`SERVICE_CLUSTER_IP_RANGE`”。例如，你可

以这样设置“SERVICE_CLUSTER_IP_RANGE=“10.0.0.0/16”，这样的话就会允许65534个不同的Service同时运行。请注意，你可以调整这个IP地址段。但不允许在Service和Pod运行的时候移除这个网段。

同样，你需要为主节点选一个静态IP地址。 — 命名这个IP地址为“MASTER_IP”。 — 配置防火墙，从而允许访问apiserver端口80和443。 — 使用sysctl设置“net.ipv4.ip_forward = 1”从而开启IPv4 forwarding。

集群命名

为你的集群选个名字。要选一个简短不会和其他服务重复的名字。

- 用kubectl来访问不同的集群。比如当你想在其他的区域测试新的Kubernetes版本。
- Kubernetes集群可以建立一个Cloud Provider资源（例如，AWS ELB）。所以不同的集群要能区分他们之间的相关资源，这个名字叫做“CLUSTERNAME”。

软件安装包

你需要以下安装包

- etcd
- 以下容器二选一：
 - docker
 - rkt
- Kubernetes
 - kubelet
 - kube-proxy
 - kube-apiserver
 - kube-controller-manager
 - kube-scheduler

下载和解压缩Kubernetes安装

Kubernets安装版本包包含所有Kuberentes的二进制发行版本和所对应的etcd。你可使直接使用这个二进制发行版本（推荐）或者按照[开发者文档](#)说明编译这些Kubernetes的二进制文件。本指南只讲述了如何直接使用二进制发行版本。下载[最新安装版本](#)并解压缩。之后找到你下载“./kubernetes/server/kubernetes-server-linux-amd64.tar.gz”的路径，并解压缩这个压缩包。然后在其中找到“./kubernetes/server/bin”文件夹。里面有有所需的可运行的二进制文件。

选择安装镜像

在容器外运行docker, kuberlet和kube-proxy, 就像你运行任何后台系统程序。所以你只需要这些基本的二进制运行文件。etcd, kube-apiserver, kube-controller-manager和kube-scheduler, 我们建议你在容器里运行etcd, kube-apiserver, kube-controller-manager和kube-scheduler。所以你需要他们的容器镜像。你可以选择不同的Kubernetes镜像：

- 你可以使用Google Container Registry (GCR)上的镜像文件：
 - 例如“gcr.io/google_containers/hyperkube:\$TAG”。这里的“TAG”是指最新的发行版本的标示。这个表示可以从[最新发行说明](#)找到。
 - 确定\$TAG和你使用的kubelet, kube-proxy的标签是一致的。
 - [hyperkube](#)是一个集成二进制运行文件
 - 你可以使用“hyperkube kubelet ...”来启动kubelet，用“hyperkube apiserver ...”运行apiserver, 等等。
- 生成你自己的镜像：
 - 使用私有镜像服务器。
 - 例如，可以使用“docker load -i kube-apiserver.tar”将“./kubernetes/server/bin/kube-apiserver.tar”文件转化成dokcer镜像。
 - 之后可使用“docker images”来验证镜像是否从制定的镜像服务器加载成功。

对于etcd, 你可以：

- 使用上Google Container Registry (GCR)的镜像，例如“gcr.io/google_containers/etcd:2.0.12”
- 使用[Docker Hub](#)或者[Quay.io](#)上的镜像，例如“quay.io/coreos/etcd:v2.2.0”
- 使用你操作系统自带的etcd
- 自己编译
 - 你可以使用这个命令行“cd kubernetes/cluster/images/etcd; make”

我们建议你使用Kubernetes发行版本里提供的etcd版本。Kubernetes发行版本里的二进制运行文件只是和这个版本的etcd测试过。你可以
在“kubernetes/cluster/images/etcd/Makefile”里“ETCD_VERSION”所对应的值找到所推荐的版本号。接下来本文档会假设你已经选好镜像标示并设置了对应的环境变量。设置好了最新的标示和正确的镜像服务器：

- "HYPERKUBE_IMAGE==gcr.io/google_containers/hyperkube:\$TAG"
- "ETCD_IMAGE=gcr.io/google_containers/etcd:\$ETCD_VERSION"

安全模式

这里有两种主要的安全选项：

- 使用HTTP访问apiserver
 - 配合使用防火墙。
 - 这种方法比较易用。

- 使用HTTPS访问apiserver
 - 配合电子证书和用户登录信息使用。
 - 推荐使用
 - 设置电子证书较为复杂。

如果要用HTTPS这个方式，你需要准备电子证书和用户登录信息。

准备安全证书

你需要准备多个证书：

- 主节点会是一个HTTPS服务器，所以需要一个证书。
- 如果Kuberlets需要通过HTTPS提供API服务时，这些kuberlets需要出示证书向主节点证明主从关系。

除非你决定要一个真正CA来生成这些证书的话，你需要一个根证书，并用这个证书来给主节点，kuberlet和kubectl的证书签名。

- 参见“cluster/gce/util.sh”脚本里的“create-certs”
- 并参见“cluster/saltbase/salt/generate-cert/make-ca-cert.sh”和“cluster/saltbase/salt/generate-cert/make-cert.sh”

你需要修改以下部分(我们之后也会用到这些参数的)

- “CA_CERT”
 - 放置在apiserver所运行的节点上，比如“/srv/kubernetes/ca.crt”。
- “MASTER_CERT”
 - 用CA_CERT来签名
 - 放置在apiserver所运行的节点上，比如“/srv/kubernetes/server.crt”。
- “MASTER_KEY”
 - 放置在apiserver所运行的节点上，比如“/srv/kubernetes/server.key”。
- “KUBELET_CERT”
 - 可选配置
- “KUBELET_KEY”
 - 可选配置

准备登录信息

管理员（及任何用户）需要：

- 对应验证身份的令牌或密码。
- 令牌可以是长字符串，比如32个字符
 - 可参见“TOKEN=\$(dd if=/dev/urandom bs=128 count=1 2>/dev/null | base64 | tr -d '=+/' | dd bs=32 count=1 2>/dev/null)“”

你的令牌和密码需要保存在apiserver上的一个文件里。本指南使用这个文件”/var/lib/kube-apiserver/known_tokens.csv“。文件的具体格式在[认证文档](#)里描述了。至于如何把登录信息分发给用户，Kubernetes是将登录信息放入[kubeconfig文件](#)里。

管理员可以按如下步骤创建kubeconfig文件：

- 如果你已经在非定制化的集群上运行过Kubernetes(例如，按照入门指南架设过Kubernetes)，那么你已经有”\$HOME/.kube/config“文件了。
- 你需要在kuberconfig文件里添加证书，密钥和主节点IP：
 - 如果你选择了“firewall-only”的安全设置，你需要按如下设置apiserver：
 - “kubectl config set-cluster \$CLUSTER_NAME --server=[http://\\$MASTER_IP](http://$MASTER_IP) --insecure-skip-tls-verify=true”
 - 否则，按如下设置你的apiserver的IP，证书，用户登录信息：
 - “kubectl config set-cluster \$CLUSTER_NAME --certificate-authority=\$CA_CERT --embed-certs=true --server=[https://\\$MASTER_IP](https://$MASTER_IP)”
 - “kubectl config set-credentials \$USER --client-certificate=\$CLI_CERT --client-key=\$CLI_KEY --embed-certs=true --token=\$TOKEN”
 - 设置你的集群为缺省集群：
 - “kubectl config set-context \$CONTEXT_NAME --cluster=\$CLUSTER_NAME --user=\$USER”
 - “kubectl config use-context \$CONTEXT_NAME”

接下来，为kubelets和kube-proxy准备kubeconfig文件。至于要准备多少不同的文件，这里有几个选项：

1. 使用和管理员同样的登陆账号
 - 这是最简单的建设方法。
2. 所有的kubelet使用同一个令牌和kubeconfig文件，另外一套给所有的kube-proxy使用，在一套给管理员使用。
 - 这个设置和GCE的配置类似。
3. 为每一个kubelet，kube-proxy和管理员准备不同的登陆账号。
 - 这个配置在实现中，目前还不支持。

为了生成这个文件，你可以参照“cluster/gce/configure-vm.sh”中的代码直接从”\$HOME/.kube/config“拷贝过去或者参考以下模版：

```

apiVersion: v1
kind: Config
users:
- name: kubelet
  user:
    token: ${KUBELET_TOKEN}
clusters:
- name: local
  cluster:
    certificate-authority-data: ${CA_CERT_BASE64_ENCODED}
contexts:
- context:
  cluster: local
  user: kubelet
  name: service-account-context
current-context: service-account-context

```

把kubeconfig文件放置到每一个节点上。本章节之后的事例会假设kubeconfig文件已经放置在“/var/lib/kube-proxy/kubeconfig”和“/var/lib/kubelet/kubeconfig”里。

在节点上配置和安装基础软件

这个章节讨论的是如何配置Kubernetes节点。你应该在每个节点运行三个后台进程：

- docker or rkt
- kubelet *kube-proxy

Docker容器

对最低Docker版本的要求是随着kubelet的版本变化的。最新的稳定版本通常是个好选择。如果版本太低，Kubelet记录下警报并拒绝运行pod，所以你可以选择个版本试一下。

如果你之前安装Docker的节点没有Kubernetes相关的配置，你可能已经有Docker新建的网桥和iptables的规则。所以你或许希望在为Kubernetes配置Docker前根据以下命令移除之前的配置。

```

iptables -t nat -F
ifconfig docker0 down
brctl delbr docker0

```

如何配置Docker取决于你网络是基于routable-vip还是overlay-network。这里有一些建议的Docker选项：

- 为每一个节点的CIDR网段建立你自己的网桥，命名为cbr0并为docker设置 --

```
bridge=cbr0。
```

- 配置 `--iptables=false`，所以docker不会为host-port设置iptables(这个控制在docker旧版本不够细致，以后会在新版本里修复)

所以kube-proxy可以代替docker来设置iptables。

- `--ip-masq=false`
 - 如果你将PodIP设置为可路由寻址，你会希望将这个选项设置为false。否则，docker会将NodeIP重写为PodIP的源地址。
 - 一些环境(例如,GCE)下需要伪装(masquerade)离开这个云环境的流量。这个配置是取决于具体的云环境的。
 - 如果你在使用overlay网络，请参考其他资料。
- `--mtu=`
 - 但使用Flannel的时候，需要这个选项。因为UDP包封装造成过大的数据包。
- `--insecure-registry $CLUSTER_SUBNET`
 - 为链接没有SSL安全链接的私有registry。

你或许希望为Docker提高可以打开文件的数目：

- `DOCKER_NFILE=1000000` 这里的设置取决于你的节点的操作系统。比如，GCE上基于Debian的发行版本使用 `/etc/default/docker` 这个配置文件。

在进行下一步安装前，可以参考Docker文档里的实例来确保docker在你的系统上正常工作。

rkt

rkt是类似Docker的技术。你只需要二选一安装Docker或者rkt。最低的版本是[v0.5.6](#)。

[systemd](#)是在节点上运行rkt必须的。与rkt v0.5.6所对应的最低版本是[systemd 215](#)。

[rkt metadata service](#)也是必须安装的，来支持rkt的网络部分。你可以用以下命令来运行rkt的metadata服务 `sudo systemctl run rkt metadata-service`

接下来你需要来设置kubelet的标记：

- `--container-runtime=rkt`

kubelet

所有的节点都要运行kubelet。参考[选择安装镜像](#)

可参考的参数：

- 如果选择HTTPS的安全配置：
 - `--api-servers=https://$MASTER_IP`
 - `--kubeconfig=/var/lib/kubelet/kubeconfig`

- 否则，使用防火墙的安全配置：
 - `--api-servers=http://$MASTER_IP`
- `--config=/etc/kubernetes/manifests`
- `--cluster-dns=` 是用来配置DNS服务器的地址(参考[Starting Addons.](#))
- `--cluster-domain=` 是为DNS集群地址使用的DNS域名前缀。
- `--docker-root=`
- `--root-dir=`
- `--configure-cbr0=` (参考之前的介绍)
- `--register-node` (参考章节[节点](#))

kube-proxy

所有的节点都要运行kube-proxy。(并不一定要在主节点上运行kube-proxy，但最好还是与其它节点保持一致) 可参考如何获得kubelet二进制运行包来获得kube-proxy二进制运行包。

可参考的参数

- 如果选择HTTPS的安全配置
 - `--api-servers=https://$MASTER_IP`
 - `--kubeconfig=/var/lib/kube-proxy/kubeconfig`
 - 否则，使用防火墙的安全配置：
- `--api-servers=http://$MASTER_IP`

网络

为了pod的网络通信，需要给每一个节点分配一个自己的CIDR网段。这个叫做 `NODE_X_POD_CIDR`。

需要给每一个节点新建一个叫 `cbr0` 网桥。网桥会在[networking documentation](#)里做详细介绍。约定俗成，`$NODE_X_POD_CIDR`里的第一个IP地址作为这个网桥的IP地址。这个地址叫做 `NODE_X_BRIDGE_ADDR`。比如，`NODE_X_POD_CIDR`是 `10.0.0.0/16`，那么 `NODE_X_BRIDGE_ADDR`是 `10.0.0.1/16`。注意：这里用 `/16` 这个后缀是因为之后也会这么使用。

- 推荐的自动化步骤：
 1. 在初始化的脚本里，设置kubelet的选项为 `--configure-cbr0=true`，并重启kubelet服务。Kubelet会自动设置cbr0。它会一直等待，直到节点controller正确设置Node.Spec.PodCIDR。因为你目前还没有设置好apiserver和节点controller，所以网桥不会马上完成设置。
- 人工步骤：
 1. 设置kubelet的选项 `--configure-cbr0=false`，并重启kubelet。
 2. 新建网桥

- 比如 `brctl addbr cbr0` .
3. 设定合适的MTU
 - 比如 `ip link set dev cbr0 mtu 1460` (注意: 真实的MTU值是由你的网络环境所决定的)
 4. 把集群网络加入网桥(docker会连接在这个网桥的另一端)。
 - 比如 `ip addr add $NODE_X_BRIDGE_ADDR dev cbr0`
 5. 开启网桥
 - 比如 `ip link set dev cbr0 up`

在你关闭了Docker的IP伪装的情况下，为了让pod之间相互通信，你可能需要为去往集群网络外的流量做目的IP地址伪装，例如：

```
iptables -t nat -A POSTROUTING ! -d ${CLUSTER_SUBNET} -m addrtype ! --dst-type LOCAL -j M
```

这样会重写从PodIP到节点IP的数据流量的原地址。内核[connection tracking](#)会确保发向节点的回复能够到达pod。

注意: 需不需要IP地址伪装是视环境而定的。在一些环境下是不需要IP伪装的。例如GCE这样的环境从pod发出的数据是不允许发向Internet的，但如果在同一个GCE项目里是不会有问题的。

其他

- 如果需要，为你的系统安装包管理器开启自动升级。
- 为所有的节点设置日志轮询(比如，使用[logrotate](#))。
- 建立liveness-monitoring (比如，使用[monit](#))。
- 建立存储插件的支持(可选)
 - 为可选的存储类型安装所需的客户端程序，比如为GlusterFS安装 `glusterfs-client` 。

使用配置管理工具

之前架设服务器的步骤都是使用“传统”的系统管理方式。你可以尝试使用系统配置工具来自动化架设流程。你可以参考其他入门指南，比如使用[Saltstack](#)，[Ansible](#)，[Juju](#)和[CoreOS Cloud Config](#)。

引导安装集群

通常情况下，基本的节点服务(kubelet, kube-proxy和docker)都是由传统的系统配置方式完成建立和管理的。其他的Kubernetes的相关部分都是由Kubernetes本身来完成配置和管理的：

- 配置和管理的选项在Pod spec(yaml or json)而不是/etc/init.d文件或systemd unit里定义的。
- 他们都是由Kubernetes而不是init来负责运行的。

etcd

你需要运行一个或多个etcd实例。

- 推荐方式: 运行一个etcd实例，将日志保存在类似RAID, GCE PD的永久存储空间上。
- 或者: 运行3个或者5个etcd实例。
 - 日志可以保存在 Log can be written to non-durable storage because storage is replicated.
 - 运行一个apiserver，这个apiserver连接到其中一个etc实例上。参见[cluster-troubleshooting](#)获取更多的有关集群可用性的信息。

启动一个etcd实例：

1. 复制 `cluster/saltbase/salt/etcd/etcd.manifest` 2. 做有必要的设置修改 3. 将这个文件放到 `kubelet manifest` 的文件夹中

Apiserver, Controller Manager和Scheduler

在主节点上，apiserver, controller manager, scheduler会运行在各自的pod里。

启动以上三个服务的步骤大同小异：

1. 从为pod所提供的template开始。
2. 设置 `HYPERNODE_IMAGE` 的值为[选择安装镜像](#)中所设置的值。
3. 可参考以下的template来决定你集群所需的选项。
4. 在 `commands` 列表里设置所需的运行选项（例如，`$ARGN`）。
5. 将完成的template放在kubelet manifest的文件夹内。
6. 验证pod是否运行。

Apiserver pod模版

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-apiserver"
  }
}
```

```
},
"spec": {
  "hostNetwork": true,
  "containers": [
    {
      "name": "kube-apiserver",
      "image": "${HYPERKUBE_IMAGE}",
      "command": [
        "/hyperkube",
        "apiserver",
        "$ARG1",
        "$ARG2",
        ...
        "$ARGN"
      ],
      "ports": [
        {
          "name": "https",
          "hostPort": 443,
          "containerPort": 443
        },
        {
          "name": "local",
          "hostPort": 8080,
          "containerPort": 8080
        }
      ],
      "volumeMounts": [
        {
          "name": "srvkube",
          "mountPath": "/srv/kubernetes",
          "readOnly": true
        },
        {
          "name": "etcssl",
          "mountPath": "/etc/ssl",
          "readOnly": true
        }
      ],
      "livenessProbe": {
        "httpGet": {
          "path": "/healthz",
          "port": 8080
        },
        "initialDelaySeconds": 15,
        "timeoutSeconds": 15
      }
    }
  ],
  "volumes": [
    {
      "name": "srvkube",
      "hostPath": {
```

```

        "path": "/srv/kubernetes"
    }
},
{
    "name": "etcssl",
    "hostPath": {
        "path": "/etc/ssl"
    }
}
]
}
}

```

可选设置的apiserver的选项：

- `--cloud-provider=` 参见 [cloud providers](#)
- `--cloud-config=` 参见 [cloud providers](#)
- 如果你想在主节点运行proxy，你需要设置 `--address=${MASTER_IP}` 或者 `--bind-address=127.0.0.1` 和 `--address=127.0.0.1`。
- `--cluster-name=$CLUSTER_NAME`
- `--service-cluster-ip-range=$SERVICE_CLUSTER_IP_RANGE`
- `--etcd-servers=http://127.0.0.1:4001`
- `--tls-cert-file=/srv/kubernetes/server.cert`
- `--tls-private-key-file=/srv/kubernetes/server.key`
- `--admission-control=$RECOMMENDED_LIST`
 - 参考 [admission controllers](#).
- 只有当你相信你的集群用户可以使用root权限来运行pod时，开启这个选项 `--allow-privileged=true`，

如果你是按照firewall-only的安全方式来配置的，你需要以下设置：

- `--token-auth-file=/dev/null`
- `--insecure-bind-address=${MASTER_IP}`
- `--advertise-address=${MASTER_IP}`

如果你是按照HTTPS的安全方式来配置的，你需要以下设置：

- `--client-ca-file=/srv/kubernetes/ca.crt`
- `--token-auth-file=/srv/kubernetes/known_tokens.csv`
- `--basic-auth-file=/srv/kubernetes/basic_auth.csv`

这个pod使用 `hostPath` 加载多个节点文件系统目录。这些加载的目录的用途是：

- 加载 `/etc/ssl` 目录可以允许apiserver找到SSL根证书，从而验证例如云服务提供商所提供的外部服务。
 - 如果你不使用任何云服务提供商，你就不需要配置这里（比如，只使用物理裸

机)。

- 加载 `/srv/kubernetes` 目录可以允许读取存储在节点磁盘上的证书和认证信息。
- 可选, 你也可以加在 `/var/log` 目录从而将日志记录在这个目录里(没有在template里举例标明)。
 - 如果你想用类似journalctl的工具从根文件系统来访问日志的话, 可以加载这个目录。 *TODO* 描述如何架设proxy-ssh。

Cloud Providers

Apiserver支持多个cloud providers。

- `--cloud-provider` 选项的值可以是 `aws` , `gce` , `mesos` , `openshift` , `ovirt` , `rackspace` , `vagrant` 或者不'未设置。
- 未设置选项可以用来设置物理裸机。
- 在[这里](#)添加新的IaaS。

一些cloud providers需要配置文件。这种情况下, 你需要将配置文件放置在apiserver的镜像中或者通过 `hostPath` 来加载。

- 如果cloud providers需要配置文件, 设置 `-cloud-config=` 这个选项。
- `aws` , `gce` , `mesos` , `openshift` , `ovirt` 和 `rackspace` 会使用到这个选项。
- 你必须需要将配置文件放置在apiserver的镜像中或者通过 `hostPath` 来加载。
- 云配置文件的语法[Gcfg](#)。
- AWS格式是用类型来定义的[AWSCloudConfig](#)。
- 其他的云服务提供商也有类似的对应文件。
- 比如在GCE里: 在[这个文件](#)找 `gce.conf` 字节。

Scheduler pod template

完成scheduler pod的template :

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-scheduler"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-scheduler",
        "image": "$HYPERKUBE_IMAGE",
        "command": [
          "/hyperkube",
          "scheduler",
          "--master=127.0.0.1:8080",
          "$SCHEDULER_FLAG1",
          ...
          "$SCHEDULER_FLAGN"
        ],
        "livenessProbe": {
          "httpGet": {
            "host": "127.0.0.1",
            "path": "/healthz",
            "port": 10251
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ]
  }
}
```

通常，不需要额外设置scheduler。

你或许想加载 `/var/log` 并将输出记录在这个日志目录里。

Controller Manager Template

完成controller manager pod的template：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-controller-manager"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-controller-manager",
        "image": "$HYPERKUBE_IMAGE",
        "command": [
          "/hyperkube",
          "controller-manager",
          "--master=127.0.0.1:8080",
          ...
          "$CONTROLLER_MANAGER_FLAGS"
        ],
        "livenessProbe": {
          "httpGet": {
            "host": "127.0.0.1",
            "path": "/healthz",
            "port": 10251
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ]
  }
}
```

```
"spec": {
    "hostNetwork": true,
    "containers": [
        {
            "name": "kube-controller-manager",
            "image": "$HYPERKUBE_IMAGE",
            "command": [
                "/hyperkube",
                "controller-manager",
                "$CNTRLMNGR_FLAG1",
                ...
                "$CNTRLMNGR_FLAGN"
            ],
            "volumeMounts": [
                {
                    "name": "srvkube",
                    "mountPath": "/srv/kubernetes",
                    "readOnly": true
                },
                {
                    "name": "etcssl",
                    "mountPath": "/etc/ssl",
                    "readOnly": true
                }
            ],
            "livenessProbe": {
                "httpGet": {
                    "host": "127.0.0.1",
                    "path": "/healthz",
                    "port": 10252
                },
                "initialDelaySeconds": 15,
                "timeoutSeconds": 15
            }
        }
    ],
    "volumes": [
        {
            "name": "srvkube",
            "hostPath": {
                "path": "/srv/kubernetes"
            }
        },
        {
            "name": "etcssl",
            "hostPath": {
                "path": "/etc/ssl"
            }
        }
    ]
}
```

配合controller manager所使用的选项：

- `--cluster-name=$CLUSTER_NAME`
- `-cluster-cidr=`
 - TODO: 解释这个选项
- `--allocate-node-cidrs=`
 - TODO: 解释这个选项
- `--cloud-provider=` 和 `--cloud-config` 在apiserver章节里解释过。
- `--service-account-private-key-file=/srv/kubernetes/server.key`，这个值是[service account](#)功能所使用的。
- `--master=127.0.0.1:8080`

运行和验证Apiserver, Scheduler和Controller Manager

将每个完成的pod template放置在kubelet的配置文件夹中（文件夹地址是在kubelet的`--config=`选项所指向的地址，通常是`/etc/kubernetes/manifests`）。没有放置顺序关系：scheduler和controller manager会一直尝试连接到apiserver，直到连接成功。

用`ps`或者`docker ps`来检测每一个进程是否正常运行。比如，你可以这样看apiserver的容器是否被kubelet启动了：

```
$ sudo docker ps | grep apiserver:  
5783290746d5      gcr.io/google_containers/kube-apiserver:e36bf367342b5a80d7467fd7611ad
```

之后尝试连接apiserver：

```
$ echo $(curl -s http://localhost:8080/healthz)  
ok  
$ curl -s http://localhost:8080/api  
{  
  "versions": [  
    "v1"  
  ]  
}
```

如果kubelets使用`--register-node=true`这个选项，他们会开始自动注册到apiserver上。很快，你就可以使用`kubectl get nodes`命令看到所有的节点。否则，你需要手动注册这些节点。

日志

TODO 如何开启日志。

监控

TODO 如何开启监控。

DNS

TODO 如何运行DNS。

故障排除

运行validate-cluster

TODO 解释如何运行“cluster/validate-cluster.sh”。

检查pods和services

你可以尝试阅读“[检查的集群](#)”这一节，例如[GCE](#)。你应该检查Service。通过“mirro pods”去检查apiserver, scheduler和controller-manager以及运行的插件。

例子

到这里你应该能够运行一些基本的实例了，例如[nginx example](#)。

运行测试

你可以试着运行[一致性测试](#)。测试失败的结果可能会给你些排除故障的线索。

网络

节点之间必须用私有IP链接。可以通过ping或者SSH来确定节点之间的联通。

获得帮助

CoreOS上部署入门指南

在Coreos运行Kubernetes有多个指南：

CoreOS官方指南

这些指南由CoreOs维护，“CoreOS Way”提供了TLS协议，DNS add-on等部署Kubernetes的指南。指南都通过了Kubernetes的一致性测试，当然，也鼓励自己[测试自己的部署](#)。

Vagrant Multi-Node

这是在Vagrant上搭建多节点集群的指南。部署者可以单独配置etcd nodes, master nodes和worker nodes节点数，来调出一个完整的HA的控制板。

Vagrant Single-Node

这是一个在本地快速搭建Kubernetes开发环境的方式。简单的 `git clone`, `vagrant up` 和配置 `kubectl` 三步就可以了。

Full Step by Step Guide

这是在任何云或者机器上搭建一个TLS协议的HA集群的一般的指南。根据角色重复master或者worker的部署去配置更多的机器。

社区指南

这些指南由社区人员维护，包括一些特殊平台，使用案例和在CoreOS使用不同的方式来配置Kubernetes的经验。

Multi-node Cluster

在选择的平台中：AWS, GCE, 或者VMware Fusion搭建一个单个master, multi-worker集群的指南。

Easy Multi-node Cluster on Google Compute Engine

在GCE上通过脚本安装一个单个master, multi-worker的集群的指南。使用[fleet](#)来管理Kubernetes的部件。

Multi-node cluster using cloud-config and Weave on Vagrant

配置一个Vagrant-based集群，包括3台带有Weave网络的机器的指南。

Multi-node cluster using cloud-config and Vagrant

通过选择的虚拟管理程序：VirtualBox, Parallels或者Parallels配置一个单个master, multi-worker本地集群的指南。

Multi-node cluster with Vagrant and fleet units using a small OS X App

这是通过OS X应用程序控制运行一个单个master, multi-worker集群指南。Under the hood使用Vagrant。

Resizable multi-node cluster on Azure with Weave

在Azure上运行一个HA etcd集群，包括一个单个master的指南。使用Azure node.js CLI去扩展集群。

Multi-node cluster using cloud-config, CoreOS and VMware ESXi

在VMware ESXi配置一个单个master, 单个worker集群的指南。

Juju上部署入门

译者：王乐 校对：无

Juju使通过配置部署Kubernetes，在集群中安装和配置所有系统更加简单。可通过一个命令增加集群尺寸来简单的扩展集群

内容列表

- 部署需求
 - Ubuntu上部署
 - Docker相关部署
- 运行Kubernetes集群
- 检测集群
- 运行多个容器！
- 扩展集群
- 运行“k8petsore”示例应用
- 删除集群
- 更多信息
 - 云兼容

部署需求

注意：如果你运行kube-up，在Ubuntu上——所有的依赖会为您处理。你可以跳转到这个部分：运行Kubernetes集群

Ubuntu上部署

在您的本地Ubuntu系统安装Juju客户端。

```
sudo add-apt-repository ppa:juju/stable
sudo apt-get update
sudo apt-get install juju-core juju-quickstart
```

Docker相关部署

如果您不使用Ubuntu，而是使用Docker，您可以运行以下命令：

```
mkdir ~/.juju
sudo docker run -v ~/.juju:/home/ubuntu/.juju -ti jujusolutions/jujubox:latest
```

此时你可以在当前路径上获取 `juju quickstart` 命令。

为您所选择允许的云设置证书：

```
juju quickstart --constraints="mem=3.75G" -i
```

这里 `constraints` flag 是可选的，当请求一个新的虚拟机时，他改变 Juju 生成的虚拟机尺寸。大的虚拟机相比小虚拟机运行的更快，但是花费更多。

根据对话选择 `save` 和 `use`。快速入门将启动 Juju 跟节点，根据用户接口设置 Juju 页面。

运行 Kubernetes 集群

启动集群之前需要导出环境变量 `KUBERNETES_PROVIDER`。

```
export KUBERNETES_PROVIDER=juju
cluster/kube-up.sh
```

如果这是第一次运行 `kube-up.sh` 脚本，它会安装 Juju 部署入门的所有依赖关系，另外它会根据通用配置运行一个窗口，允许你选择云服务商，输入适当的访问凭证。

下一步它会部署 Kubernetes master, etcd, 2个带有 flannel 的 nodes，flannel 是基础软件定义网络 (SDN)，它可以使在不同的主机上的容器互相通信。

检测集群

`juju status` 命令提供了集群中每个单元的信息：

```
$ juju status --format=oneline
- docker/0: 52.4.92.78 (started)
  - flannel-docker/0: 52.4.92.78 (started)
  - kubernetes/0: 52.4.92.78 (started)
- docker/1: 52.6.104.142 (started)
  - flannel-docker/1: 52.6.104.142 (started)
  - kubernetes/1: 52.6.104.142 (started)
- etcd/0: 52.5.216.210 (started) 4001/tcp
- juju-gui/0: 52.5.205.174 (started) 80/tcp, 443/tcp
- kubernetes-master/0: 52.6.19.238 (started) 8080/tcp
```

你可以使用 `juju ssh` 去访问任何一个单元：

```
juju ssh kubernetes-master/0
```

运行多个容器！

在Kubernetes主节点 `kubectl` 是可用的。我们ssh登录去运行一些容器，但也可以通过设置 `KUBERNETES_MASTER` 为“`kubernetes-master/0`”的ip地址来使用本地 `kubectl`。

在启动一个容器前无pods可获取

```
kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubectl get replicationcontrollers
CONTROLLER   CONTAINER(S)  IMAGE(S)  SELECTOR  REPLICAS
```

我们将跟随aws-coreos实例。创建一个pod清单：`pod.json`

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "hello",
    "labels": {
      "name": "hello",
      "environment": "testing"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "hello",
        "image": "quay.io/kelseyhightower/hello",
        "ports": [
          {
            "containerPort": 80,
            "hostPort": 80
          }
        ]
      }
    ]
  }
}
```

使用`kubectl`创建pod：

```
kubectl create -f pod.json
```

获取pod信息：

```
kubectl get pods
```

测试hello应用，我们需要定位容器所在的节点。使用Juju去作用于容器的更好的工具是在工作节点上，但是我们可以使用 `juju run` 和 `juju status` 去找到我们的hello应用。

```
juju run --unit kubernetes/0 "docker ps -n=1"
...
juju run --unit kubernetes/1 "docker ps -n=1"
CONTAINER ID        IMAGE                               COMMAND      CREATED
02beb61339d8        quay.io/kelseyhightower/hello:latest   /hello      About an h
```

我们看到“kubernetes/1”有我们的容器，我们可以打开80端口：

```
juju run --unit kubernetes/1 "open-port 80"
juju expose kubernetes
sudo apt-get install curl
curl $(juju status --format=oneline kubernetes/1 | cut -d' ' -f3)
```

最后删除pod：

```
juju ssh kubernetes-master/0
kubectl delete pods hello
```

扩展集群

我们可以想这样增加节点单元：

```
juju add-unit docker # creates unit docker/2, kubernetes/2, docker-flannel/2
```

运行“**k8petstore**”示例应用

[k8petstore示例](#)可以像一个[juju action](#)获取到。

```
juju action do kubernetes-master/0
```

注意：这个示例既包含curl状态来练习这个应用。这个应用自动生成“prestore”日志写到redis上，并且允许你在浏览器上可视化吞吐量。

删除集群

```
./kube-down.sh
```

或者破坏你当前的Juju环境（使用 `juju env` 命令）

```
juju destroy-environment --force `juju env`
```

更多信息

Kubernetes的分支和包可以在github.com的 `kubernetes` 项目中找到：

- 镜像包
 - [Kubernetes主节点分支](#)
 - [Kubernetes节点分支](#)
- [关于Juju的更多信息](#)

云兼容

Juju运行在本地和各种公共云提供商。Juju当前和[Amazon Web Service](#), [Windows Azure](#), [DigitalOcean](#), [Google Compute Engine](#), [HP Public Cloud](#), [Joyent](#), [LXC](#), 任何[OpenStack](#)部署, [Vagrant](#), 和 [Vmware vSphere](#)。

如果你没有在多个云列表看到你比较喜欢的云供应商, 可以配置为[手动配置](#)。

Kubernetes包已经在GCE和AWS上测试, 使用1.0.0版本验证通过。

Racksapce上部署入门

译者：安雪艳 校对：无

内容列表

- 入门介绍
- 部署需求
- 供应商：Racksapce
- 编译
- 集群
- 注意点：
- 网络设计

入门介绍

- 支持版本: v0.18.1

一般来说， Racksapce的dev-build-and-up.sh工作流程类似于Google Compute Engine。具体实现是不同的，因为Rackspace使用CoreOS， Rackspace云文件和整体网络设计。

这些脚本应该用于部署Kubernetes的开发环境。如果你的账号利用RackConnect或者非标准的网络，若不修改这些脚本，这些脚本将无法使用。

注意: rackspace脚本不依赖 saltstack 而依赖于配置文件cloud-init。

当前集群设计受如下启发：

- corekube
- Angus Lees

部署需求

1. Python2.7
2. 需要安装 nova 和 swiftly 。推荐使用python virtualenv来安装这些包。
3. 确认已经有与OpenStack APIs交互的合适的环境变量。获取更详细信息请查看 Rackspace文档。

供应商 : Rackspace

- 使用 `KUBERNETES_PROVIDER=rackspace` 从源码编译自己的发布版本并且运行 `bash hack/dev-build-and-up.sh`
- 注意: 我们脚本中还没有提供`get.k8s.io`安装方法。
 - 请使用 `export KUBERNETES_PROVIDER=rackspace; wget -q -O - https://get.k8s.io | bash` 来安装Kubernetes最近发布版本

编译

1. Kubernetes二进制文件将通过 `build/` 下的通用编译脚本编译。
2. 如果已经设置了环境变量 `ENV KUBERNETES_PROVIDER=rackspace`， 脚本将上传 `kubernetes-server-linux-amd64.tar.gz` 到云文件。
3. 通过`swiftly`创建一个云文件容器，且在这个对象内启用一个临时URL。
4. 编译过的 `kubernetes-server-linux-amd64.tar.gz` 将上传到这个容器内，当`master/nodes` 启动时这个URL将传到`master/nodes`。

集群

有一个特殊的 `cluster/rackspace` 脚本目录有如下步骤：

1. 创建一个云网络并且所有的实例附属于这个网络。
 - `flanneld`使用这个网络作为下一跳路由。这些路由使每个节点上的容器和这个私有网络内其他的容器之间通信。
2. 如果需要，将创建且上传一个SSH key。这个key必须用于ssh登录机器（我们不捕获密码）
3. 通过 `nova` CLI创建主节点server和额外节点。生成一个 `cloud-config.yaml` 作为户数据和整个系统的配置
4. 然后，我们通过 `$NUM_MINIONS` 定义的数量启动节点。

注意点

- 脚本设置 `eth2` 成为云网络，容器可通过它通信。
- `config-default.sh` 中条目的数量可通过环境变量覆盖。
- 旧版本请选择：
 - 使用 `git checkout v0.9` 同步到 `v0.9`
 - 下载 `snapshot of v0.9`

- 使用 `git checkout v0.3` 同步到 `v0.3`
- 下载 `snapshot of v0.3`

网络设计

- `eth0` - servers/containers访问网络的公有接口
- `eth1` - ServiceNet - 集群内部通信 (`k8s`, `etcd`, `etc`) 通过这个接口。`cloud-config` 文件使用特殊CoreOS标识符 `$ private_ipv4` 配置服务
- `eth2` - Cloud Network - `k8s` pods使用这个和其他pods通信。服务代理通过这个接口传输流量。

CloudStack入门指南

译者:tiger 校对:无

内容列表

- [介绍](#)
- [前提条件](#)
- [克隆脚本](#)
- [创建一个 Kubernetes 集群](#)

简介

[CloudStack](#)是一个用于构建基于硬件虚拟化的公有云和私有云（传统IaaS）的软件。在 CloudStack 上部署 Kubernetes 有好几种方法，需要根据 CloudStack 所使用的哪种云和有哪些可用镜像来决定。例如 [Exoscale](#) 就提供了一个 [coreOS](#) 的可用模版，因此也可以使用在 coreOS 部署 Kubernetes 的指令来部署。CloudStack 同样也提供了一个 Vagrant 插件，因此也可以用 Vagrant 来部署 Kubernetes，既可以选择原有的基于 shell 脚本的部署方式，也可以选择新的基于 Salt 的部署方式。

CloudStack的 [CoreOS](#) 模版会[每日](#)构建。在执行安装 Kubernetes 部署指令之前需要先将模版[注册](#)到云上。

本指引使用了[Ansible playbook](#)。完全自动化构建，单个 Kubernetes 部署脚本基于 coreOS 指令构建。

[Ansible](#) 脚本基于 coreOS 镜像将 Kubernetes 部署到 CloudStack 基础云上。该脚本创建一个SSH密钥对、一个安全组和相关规则并最终通过云初始化配置来启动coreOS实例。

前提条件

```
$ sudo apt-get install -y python-pip  
$ sudo pip install ansible  
$ sudo pip install cs
```

[cs](#) 是一个 CloudStack API 的 python 模块。

可以通过使用API密钥和HTTP的方式来配置CloudStack终端。

你可以将它们定义成环境变量: `CLOUDSTACK_ENDPOINT` , `CLOUDSTACK_KEY` , `CLOUDSTACK_SECRET` 和 `CLOUDSTACK_METHOD` .

或者通过创建 `~/.cloudstack.ini` 文件的方式：

```
[cloudstack]
endpoint = <your cloudstack api endpoint>
key = <your api access key>
secret = <your api secret key>
method = post
```

我们需要使用 http POST 请求来将 *large* 用户的数据上传到各个coreOS实例。

克隆脚本

```
$ git clone --recursive https://github.com/runseb/ansible-kubernetes.git
$ cd ansible-kubernetes
```

`ansible-cloudstack` 模块被设置成了该仓库的一个子模块，因此需要使用 `--recursive`。

创建一个 **Kubernetes** 集群

你只需要简单运行如下脚本。

```
$ ansible-playbook k8s.yml
```

编辑 `k8s.yml` 文件中的一些变量。

```
vars:
  ssh_key: k8s
  k8s_num_nodes: 2
  k8s_security_group_name: k8s
  k8s_node_prefix: k8s2
  k8s_template: Linux CoreOS alpha 435 64-bit 10GB Disk
  k8s_instance_type: Tiny
```

它将启动一个 Kubernetes 主机和一些计算节点（默认为2个）。`instance_type` 和 `template` 默认指定的是 `exoscale`，编辑这两个参数来指定你 CloudStack 云的模板和实例类型（即服务提供商）。

如果你想修改一些其他参数的话，请参照 `roles/k8s` 里面的任务和模版。

一旦脚本执行完成，命令行会打印出Kubernetes主机的IP地址：

```
TASK: [k8s | debug msg='k8s master IP is {{ k8s_master.default_ip }}'] *****
```

使用 `core` 用户和刚创建的密钥通过 ssh 登录到主机，你可以列出集群中的所有机器：

```
$ ssh -i ~/.ssh/id_rsa_k8s core@<master IP>
$ fleetctl list-machines
MACHINE          IP           METADATA
a017c422...     <node #1 IP>   role=node
ad13bf84...     <master IP>    role=master
e9af8293...     <node #2 IP>   role=node
```

vSphere的入门指南

译者：tiger 校对：无

下面的示例使用了4个虚拟工作节点和1个虚拟主机（即集群中一共有5台虚拟机）来创建一个Kubernetes集群。集群是通过你的工作站（或任何你觉得方便的地方）来安装和控制的。

内容列表

- [前提条件](#)
- [安装](#)
- [启动集群](#)
- [其他：部署失败调试](#)

前提条件

1. 需要有一台ESXi机器或vCenter实例的管理员权限。
2. 需要先安装Go（1.2或以上版本）。下载地址：www.golang.org。
3. 需要在环境变量中添加 `GOPATH` 并将 `$GOPATH/bin` 添加到 `PATH` 中。

```
export GOPATH=$HOME/src/go
mkdir -p $GOPATH
export PATH=$PATH:$GOPATH/bin
```

4. 安装govc工具来和ESXi/vCenter进行交互：

```
go get github.com/vmware/govmomi/govc
```

5. 需要预先下载或编译[二进制版本](#)

Setup

下载一个预置了Debian 7.7 的VMDK，把它作为基础镜像来使用：

```
curl --remote-name-all https://storage.googleapis.com/govmomi/vmdk/2014-11-11/kube.vmdk.gz
md5sum -c kube.vmdk.gz.md5
gzip -d kube.vmdk.gz
```

将VMDK导入vSphere中：

```
export GOVC_URL='user:pass@hostname'
export GOVC_INSECURE=1 # If the host above uses a self-signed cert
export GOVC_DATASTORE='target datastore'
export GOVC_RESOURCE_POOL='resource pool or cluster with access to datastore'

govc import.vmdk kube.vmdk ./kube/
```

验证VMDK是否已经正确上传并扩展到~3GiB：

```
govc datastore.ls ./kube/
```

检查文件 `cluster/vsphere/config-common.sh` 是否已经配置了必填参数。该导入镜像的游客登录帐号为 `kube:kube`。

启动集群

现在继续部署Kubernetes。整个过程需要大约10分钟。

```
cd kubernetes # Extracted binary release OR repository root
export KUBERNETES_PROVIDER=vsphere
cluster/kube-up.sh
```

参见根目录下的README和《谷歌计算引擎入门指南》。一旦你成功到达了这一步，你的vSphere Kubernetes就可以像其他Kubernetes集群一样正常工作了。

开始享受**Kubernetes**之旅吧！

其他：部署失败调试

`kube-up.sh` 输出可以查看部署集群中各个虚拟机的ip地址，你可以用 `kube` 账户登录到任何虚拟机上查看并找出到底发生了什么状况。（通过你的SSH密钥或密码'kube'来登录）

libvirt CoreOS的入门指南

译者：mr.art 校对：无

亮点

1. 超高速启动集群（几秒钟，而不是几分钟的游离状态）
2. 使用COW来节省磁盘使用空间
3. 使用KSM来节省内存占用空间

环境准备

1. 安装dnsmasq
2. 安装ebtables
3. 安装qemu
4. 安装libvirt
5. 启用和启动libvirt守护进程，例如：
 - systemctl enable libvirtd
 - systemctl start libvirtd
6. [开启libvirt接入的用户权限](#)
7. 确保qemu用户可以使用\$HOME

在分布式环境中，libvirt的接入默认被拒绝或者每次接入需要密码。

可以使用如下命令进行测试：

```
virsh -c qemu:///system pool-list
```

如果接入报错，请阅读<https://libvirt.org/acl.html> 和<https://libvirt.org/aclpolkit.html>。

简而言之，如果libvirt是在包含Polkit的环境下编译（例如：Arch, Fedora 21），可以创建内容如下/etc/polkit-1/rules.d/50-org.libvirt.unix.manage.rules文件来给libvirt接入所有用户的权限

```
sudo /bin/sh -c "cat - > /etc/polkit-1/rules.d/50-org.libvirt.unix.manage.rules" << EOF
polkit.addRule(function(action, subject) {
```

```
    if (action.id == "org.libvirt.unix.manage" &&
        subject.user == "$USER") {
        return polkit.Result.YES;
        polkit.log("action=" + action);
        polkit.log("subject=" + subject);
    }
}
```

}); EOF

(\$USER为你的登陆用户名)

如果libvirt在不包含Polkit的环境下编译（例如：14.04.1 LTS），检查libvirt unix socket的使用权限：
`$ ls -l /var/run/libvirt/libvirt-sock`
`srwxrwx--- 1 root libvirtd 0 févr.`
`12 16:03 /var/run/libvirt/libvirt-sock $ usermod -a -G libvirtd $USER $USER needs to`
`logout/login to have the new group be taken into account`

Qemu是以一个具体的用户运行，他必须能接入到VMs驱动中。

所有的虚拟机都需要磁盘驱动器（CoreOS disk image, Kubernetes binaries, cloud-init files, 等等.），这些驱动都放入到./cluster/libvirt-coreos/libvirt_storage_pool中。如果你的\$HOME是可以读的，那么一起都好。如果你的\$HOME是私有的，那么脚本

cluster/kube-up.sh将会报如下错误：
`error: Cannot access storage file`
`'$HOME/.../kubernetes/cluster/libvirt-`
`coreos/libvirt_storage_pool/kubernetes_master.img'` (as uid:99, gid:78): Permission
`denied`

可以通过如下方法修复这个问题：

- 在cluster/libvirt-coreos/config-default.sh中设置POOL_PATH到一个目录：
 - 在文件系统找一个有大量空闲的磁盘空间
 - 这个目录你的用户具有可写权限
 - qemu用户可以接入
- 允许你的qemu用户可以接入到所有的存储池 设置访问权限：

`setfacl -m g:kvm:--x ~`

安装

在默认的情况下，libvirt-coreos将创建一个Kubernetes master和3个Kubernetes nodes。因为VM的驱动用了COW和由于内存释放和KSM，有许多的资源被过度的分配。开始运行你的本地cluster，打开一个shell并且运行：

`cd kubernetes export KUBERNETES_PROVIDER=libvirt-coreos cluster/kube-up.sh`

离线安装（使用裸机和CoreOS系统）

译者：林帆 校对：无

这个部分将介绍在CoreOS上运行Kubernetes的方法。比较特别的是，我们将关注于如何在离线的情况下完成部署操作，这对于进行概念验证和在访问受限的网络环境时，都将派上用场。

内容目录：

- 前提条件
- 总体设计思路
- 示例环境
- 构建CentOS的PXELINUX服务端
- 添加CoreOS系统到PXE
- 配置DHCP服务
- 部署Kubernetes
- CoreOS的Cloudinit启动配置
 - master.yml
 - node.yml
- 新建pxelinux.cfg文件
- 指定pxelinux目标
- 创建测试的Pod
- 用于调试的指令

前提条件

1. 已经有一个主机安装有运行着PXE服务端的CentOS6系统
2. 至少有两个裸机主机节点

总体设计思路

- 通过TFTP服务管理以下目录
 - /tftpboot/(coreos)(centos)(RHEL)
 - /tftpboot/pxelinux.0/(MAC) -> 链接到Linux镜像中的config文件
- 将PXELinux的链接更新到正确状态
- 将DHCP服务配置根据实际主机需求更新到正确状态
- 构建部署CoreOS的主机节点，并运行Etcd的集群

- 在不使用公有网络的情况下完成后续配置
- 部署更多的CoreOS节点作为Kubernetes的Node节点

示例环境

节点	MAC	IP
CoreOS/etc/Kubernetes Master	d0:00:67:13:0d:00	10.20.30.40
CoreOS Slave 1	d0:00:67:13:0d:01	10.20.30.41
CoreOS Slave 2	d0:00:67:13:0d:02	10.20.30.42

构建CentOS的PXELINUX服务端

完整的构建CentOS PXELinux环境方法参见[这篇文章档](#)。下面简单叙述这个流程：

1. 安装CentOS上所需的包：

```
sudo yum install tftp-server dhcp syslinux
```

2. 编辑TFTP服务配置，将 disable 一项修改为 no：

```
vi /etc/xinetd.d/tftp
```

3. 拷贝所需的syslinux镜像文件

```

su -
mkdir -p /tftpboot
cd /tftpboot
cp /usr/share/syslinux/pxelinux.0 /tftpboot
cp /usr/share/syslinux/menu.c32 /tftpboot
cp /usr/share/syslinux/memdisk /tftpboot
cp /usr/share/syslinux/mboot.c32 /tftpboot
cp /usr/share/syslinux/chain.c32 /tftpboot
/sbin/service dhcpcd start
/sbin/service xinetd start
/sbin chkconfig tftp on

```

4. 生成默认的启动菜单

```

mkdir /tftpboot/pxelinux.cfg
touch /tftpboot/pxelinux.cfg/default

```

5. 编辑启动菜单 vi /tftpboot/pxelinux.cfg/default，内容如下：

```
default menu.c32 prompt 0 timeout 15 ONTIMEOUT local display boot.msg
MENU TITLE Main Menu
LABEL local MENU LABEL Boot local hard drive LOCALBOOT 0
```

现在你就已经有了一个可以用于构建CoreOS节点的PXELinux服务了，可以通过本地的VirtualBox虚拟机或裸机上验证这个服务。

添加CoreOS系统到PXE

配置DHCP服务

部署Kubernetes

CoreOS的Cloudinit启动配置

master.yml

node.yml

新建pxelinux.cfg文件

指定pxelinux目标

创建测试的Pod

用于调试的指令

从Ferdora入门Kubernetes

译者：张以法 校对：无

本节内容

- [前提条件](#)
- [说明](#)

前提条件

1. 你需要2台或以上安装有Fedora的机器

说明

本文是针对[Fedora](#)系统的Kubernetes入门教程。通过手动配置，你将会理解所有底层的包、服务、端口等。

本文只会让一个节点（之前称从节点）工作。多节点需要在Kubernetes之外配置一个可用的[网络环境](#)，尽管这个额外的配置条件是显而易见的，（本节也不会去配置）。

Kubernetes包提供了一些服务：kube-apiserver,kube-scheduler,kube-controller-manager,kubelet,kube-proxy。这些服务通过systemd进行管理，配置信息都集中存放在一个地方：`/etc/kubernetes`。我们将会把这些服务运行到不同的主机上。第一台主机，`fed-master`，将是Kubernetes的master主机。这台机器上将运行kube-apiserver,kube-controller-manager和kube-scheduler这几个服务，此外，master主机上还将运行etcd（如果etcd运行在另一台主机上，那就不需要了，本文假设etcd和Kubernetesmaster运行在同一台主机上）。其余的主机，`fed-node`，将是从节点，上面运行kubelet, proxy和docker。

系统信息：

主机：

```
fed-master = 192.168.121.9
fed-node = 192.168.121.65
```

准备主机：

- 在所有主机上（fed-master和fed-node）都安装Kubernetes，。这对docker也适用。在fed-master上安装etcd。本文在kubernetes-0.18及之后版本上通过测试。
- yum命令之后的 `--enablerepo=update-testing` 参数可以保证安装最新的Kubernetes预览版。如果不加这个参数，你安装的版本将是Fedora提供的较旧的稳定版。
- 如果你想获取最新的版本，你可以[从Fedora Koji上下载最新的RPM包](#)，然后yum install 安装。而不是用下面的命令。

```
yum -y install --enablerepo=updates-testing kubernetes
```

- 安装etcd和iptables

```
yum -y install etcd iptables
```

- 在所有主机的/etc/hosts文件中加入master和node节点，如果DNS中已经有了主机名，就不需要加了。需要保证fed-master和fed-node之间可以正常通信，可使用ping测试网络是否连通。

```
echo "192.168.121.9 fed-master
192.168.121.65 fed-node" >> /etc/hosts
```

- 编辑/etc/kubernetes/config文件，加入以下内容。所有主机都是（包括master和node）。

```
# Comma separated list of nodes in the etcd cluster
KUBE_MASTER="--master=http://fed-master:8080"

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

- 禁用master和node上的防火墙，因为如果有其他防火墙规则管理工具的话，docker会无法正常运行。请注意以默认方式安装的fedora服务器上，iptables服务并不存在。

```
systemctl disable iptables-services firewalld
systemctl stop iptables-services firewalld
```

配置master主机上Kubernetes服务

- 按照下面的示例编辑/etc/kubernetes/apiserver文件。注意--service-cluster-ip-range参数后跟的IP地址必须是在任何地方都未使用的地址段，这些地址无需路由也无需分配到任何东西上。

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://127.0.0.1:4001"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

# Add your own!
KUBE_API_ARGS=""
```

- 编辑/etc/etcd/etcd.conf文件，使得etcd监听除了127.0.0.1之外的所有IP，如果没有做这步，将会产生“connection refused”这样的错误。

```
ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:4001
```

- 启动master上恰当的服务

```
for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

- 添加node结点：
- 在Kubernetes的master上创建下面node.json文件：

```
{
  "apiVersion": "v1",
  "kind": "Node",
  "metadata": {
    "name": "fed-node",
    "labels":{ "name": "fed-node-label"}
  },
  "spec": {
    "externalID": "fed-node"
  }
}
```

现在在你的Kubernetes集群内部创建一个node对象，执行命令

```
$ kubectl create -f ./node.json

$ kubectl get nodes
NAME           LABELS             STATUS
fed-node       name=fed-node-label  Unknown
```

请注意上面的命令，它只在内部创建了一个fed-node的引用（原文representation），并不会真的提供fed-node节点。同时，假定fed-node节点（在name中指定的）能够被解析并可从Kubernetes的master节点访问。下面本文将论述如何提供Kubernetes node节点(fed-node)。

配置node节点上的Kubernetes服务

我们需要在节点上配置kubelet

- 按照下面的示例编辑/etc/kubernetes/kubelet文件：

```
###  
# Kubernetes kubelet (node) config  
  
# The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces)  
KUBELET_ADDRESS="--address=0.0.0.0"  
  
# You may leave this blank to use the actual hostname  
KUBELET_HOSTNAME="--hostname_override=fed-node"  
  
# location of the api-server  
KUBELET_API_SERVER="--api_servers=http://fed-master:8080"  
  
# Add your own!  
#KUBELET_ARGS=""
```

- 启动节点上 (fed-node) 上恰当的服务

```
for SERVICES in kube-proxy kubelet docker; do  
    systemctl restart $SERVICES  
    systemctl enable $SERVICES  
    systemctl status $SERVICES  
done
```

- 检测以确认现在集群中fed-master能够看到fed-node，而且状态变为Ready了

```
kubectl get nodes
NAME           LABELS             STATUS
fed-node       name=fed-node-label  Ready
```

- 节点的删除 如果要想从Kubernetes集群中删除fed-node节点，需要在fed-master上执行下面命令（请别这样做，只是提示删除的方法）

```
kubectl delete -f ./node.json
```

你应该完成了吧！

集群现在应该在运行了，启动一个用于测试的Pod吧。

你应该有一个可用的集群，查看[101](#)节！

从CentOS入门Kubernetes

译者：张以法 校对：无

本节内容

- [前提条件](#)
- [启动一个集群](#)

前提条件

你需要2台或以上安装有[CentOS](#)的机器

启动一个集群

本文是针对CentOS系统的Kubernetes入门教程。通过手动配置，你将会理解所有底层的包、服务、端口等。

本文只会让一个节点工作。多节点需要在Kubernetes之外配置一个可用的的[网络环境](#)，尽管这个额外的配置条件是显而易见的，（本节也不会去配置）。

Kubernetes包提供了一些服务：kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, kube-proxy。这些服务通过[systemd](#)进行管理，配置信息都集中存放在一个地方：`/etc/kubernetes`。我们将会把这些服务运行到不同的主机上。第一台主机，`centos-master`，将是Kubernetes 集群的master主机。这台机器上将运行kube-apiserver, kube-controller-manager和kube-scheduler这几个服务，此外，master主机上还将运行etcd。其余的主机，`fed-minion`，将是从节点，将会运行kubelet, proxy和docker。

系统信息：

主机：

```
centos-master = 192.168.121.9  
centos-minion = 192.168.121.65
```

准备主机：

- 添加virt7-testing源，在所有主机上（centos-master和centos-minion），使用下面信息添

加源：

```
[virt7-testing]
name=virt7-testing
baseurl=http://cbs.centos.org/repos/virt7-testing/x86_64/os/
gpgcheck=0
```

- 在所有主机上（centos-master和centos-minion）都安装Kubernetes。这对etcd，docker和cadvisor也适用。

```
yum -y install --enablerepo=virt7-testing kubernetes
```

*注意使用etcd-0.4.6-7(这是该文档的临时版本)

如果你没有配套virt7-testing源安装etcd 0.4.6-7版，请用下面命令卸载它：

```
yum erase etcd
```

原因是在当前的的 virt7-testing源中，etcd包被更新了，会引起服务错误。执行下面两行命令安装etcd-0.4.6-7

```
yum install http://cbs.centos.org/kojifiles/packages/etcd/0.4.6/7.el7.centos/x86_64/etcd-
yum -y install --enablerepo=virt7-testing kubernetes
```

- 在所有主机的/etc/hosts文件中加入master和node节点，如果DNS中已经有了主机名，就不需要加了。

```
echo "192.168.121.9 centos-master
192.168.121.65 centos-minion" >> /etc/hosts
```

- 编辑/etc/kubernetes/config文件，加入以下内容：

```
# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://centos-master:4001"

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

- 禁用master和node上的防火墙，因为如果有其他防火墙规则管理工具的话，docker会无法正常运行。

```
systemctl disable iptables-services firewalld  
systemctl stop iptables-services firewalld
```

配置master主机上Kubernetes服务

- 按照下面的示例编辑/etc/kubernetes/apiserver文件：

```
# The address on the local server to listen to.  
KUBE_API_ADDRESS="--address=0.0.0.0"  
  
# The port on the local server to listen on.  
KUBE_API_PORT="--port=8080"  
  
# How the replication controller and scheduler find the kube-apiserver  
KUBE_MASTER="--master=http://centos-master:8080"  
  
# Port kubelets listen on  
KUBELET_PORT="--kubelet_port=10250"  
  
# Address range to use for services  
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"  
  
# Add your own!  
KUBE_API_ARGS=""
```

- 启动master上恰当的服务

```
for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do  
    systemctl restart $SERVICES  
    systemctl enable $SERVICES  
    systemctl status $SERVICES  
done
```

配置node节点上的Kubernetes服务 我们需要在节点上配置kubelet并启动kubelet和proxy

- 按照下面的示例编辑/etc/kubernetes/kubelet文件：

```
# The address for the info server to serve on
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname_override=centos-minion"

# Add your own!
KUBELET_ARGS=""
```

- 启动节点上 (fed-node) 上恰当的服务

```
for SERVICES in kube-proxy kubelet docker; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

你应该完成了！

- 检查以确认现在集群中fed-master能够看到fed-node

```
$ kubectl get nodes
NAME           LABELS      STATUS
centos-minion <none>     Ready
```

集群现在应该在运行了，启动一个用于测试的pod吧。

你应该有一个功能正常的集群，[查看101节](#)！

在Ubuntu物理节点上部署Kubernetes

译者：王乐 校对：无

介绍

这片文档介绍了如何在Ubuntu节点上部署Kubernetes，这里我们用1个主节点和3个普通节点的安装来作为范例。你可以轻松变动设置扩展到任意数量的节点。最初的想法是受到[@jainvipin](#)的Ubuntu单节点部署工作的启发。单节点的部署介绍也涵盖在本章节中。

[浙江大学云团队](#)会维护这项工作。

前提条件

- 所有节点上已经安装docker版本1.2+和用来控制Linux网桥的bridge-utils。
- 所有的机器可相互通信。主节点需要连接到Internet去下载必须的文件。
- 本指南介绍的步骤已经在Ubuntu 14.04 LTS 64bit server上测试过了。但在Ubuntu 15不能正常工作，这是因为Ubuntu 15使用systemd代替了upstart。
- 本指南所依赖于etcd-2.0.12, flannel-0.5.3和k8s-1.0.6。或许能兼容这些软件的较高版本。
- 所有的服务器能够使用ssh远程密钥认证登入，而不是用密码登入。

开始建立一个集群

建立正确的目录

复制Github上kubernetes的文件库到本地。

```
$ git clone https://github.com/kubernetes/kubernetes.git
```

配置和运行Kubernetes集群

启动过程将会首先自动下载所需的二进制安装文件。默认会下载etc版本2.0.12, Flannel版本0.5.3和k8s版本1.0.6。你可以按一下方法来设置参数`ETCD_VERSION`, `FLANNEL_VERSION` 和 `KUBE_VERSION`，从而来选定你想要的etc, Flannel和k8s版本。

```
$ export KUBE_VERSION=1.0.5
$ export FLANNEL_VERSION=0.5.0
$ export ETCD_VERSION=2.2.0
```

请注意我们在这里使用Flannel是用来建立overlay网络，但这并不是必须的。事实上你可以不借助其他工具直接建立k8s集群，或者使用Flannel，Open vSwitch或SDN来建立网络。

这是一个集群IP地址配置的例子：

IP Address	Role
10.10.103.223	node
10.10.103.162	node
10.10.103.250	both master and node

首先cluster/ubuntu/config-default.sh文件内配置集群信息。以下是个简单的参考。

```
export nodes="vcap@10.10.103.250 vcap@10.10.103.162 vcap@10.10.103.223"

export role="ai i i"

export NUM_MINIONS=${NUM_MINIONS:-3}

export SERVICE_CLUSTER_IP_RANGE=192.168.3.0/24

export FLANNEL_NET=172.16.0.0/16
```

第一个参数 `nodes` 定义了你所有的节点。主节点列在第一位并使用空格来做分割，比如 `<user_1@ip_1> <user_2@ip_2> <user_3@ip_3>`

接下来参数 `role` 按顺序定义了主机的角色，"ai"代表主机可以是主节点或普通节点，"a"代表主节点，"i"代表普通节点。

参数 `NUM_MINIONS` 定义了所有节点的数量。

参数 `SERVICE_CLUSTER_IP_RANGE` 定义了Kubernetes服务的IP地址范围。

请确保你所定义的私有IP地址范围是合理的，因为一些IaaS提供商会保留一些私有IP地址。根据RFC1918，一共有三个私有IP地址网段，如下所列。你最好不要选址一个和你自己私有地址网段冲突的网段。

10.0.0.0	-	10.255.255.255 (10/8 prefix)
172.16.0.0	-	172.31.255.255 (172.16/12 prefix)
192.168.0.0	-	192.168.255.255 (192.168/16 prefix)

参数 `FLANNEL_NET` 定义了Flannel网络所用的IP地址范围。这个地址不能和 `SERVICE_CLUSTER_IP_RANGE` 的地址冲突。

注意: 在部署时, 主节点需要连接到Internet去下载必须的文件。如果你的主机在私有网络里, 你可以在`cluster/ubuntu/config-default.sh`中设置参数 `PROXY_SETTING` 来配置Internet代理。`PROXY_SETTING="http_proxy=http://server:port https_proxy=https://server:port"`

当以上所有参数正确设置后, 你就可以用 `cluster/` 中的命令来启动整个集群了。

```
$ KUBERNETES_PROVIDER=ubuntu ./kube-up.sh
```

这个脚本自动 `scp` 传输二进制安装和配置文件到所有的主机上。之后启动这个主机上的kubernetes的服务。你唯一需要做的是当有提示的时候输入所需密码。

```
Deploying node on machine 10.10.103.223  
...  
[sudo] password to start node:
```

如果一切运行正常, k8s集群启动后你会看见以下提示信息。

```
Cluster validation succeeded
```

测试

你可以运行 `kubectl` 命令来检测刚升级的Kubernetes集群是否工作正常。`kubectl` 运行文件放置在 `cluster/ubuntu/binaries` 文件夹中。你也可以设置环境变量PATH来调用 `kubectl`。

比如, 使用 `$ kubectl get nodes` 来检测你的节点是否正常运行。

```
$ kubectl get nodes  
NAME           LABELS          STATUS  
10.10.103.162  kubernetes.io/hostname=10.10.103.162  Ready  
10.10.103.223  kubernetes.io/hostname=10.10.103.223  Ready  
10.10.103.250  kubernetes.io/hostname=10.10.103.250  Ready
```

你也可以使用Kubernetes[guest-example](#)来建立Redis后台集群。

部署插件

假设你开始运行k8s集群, 这一节将会介绍如何在现有集群上部署类似DNS和UI等插件。

可以在`cluster/ubuntu/config-default.sh` 中配置在DNS。

```
ENABLE_CLUSTER_DNS="${KUBE_ENABLE_CLUSTER_DNS:-true}"  
  
DNS_SERVER_IP="192.168.3.10"  
  
DNS_DOMAIN="cluster.local"  
  
DNS_REPLICAS=1
```

参数 `DNS_SERVER_IP` 定义了DNS的IP，这个IP必须在 `SERVICE_CLUSTER_IP_RANGE` 的范围内。参数 `DNS_REPLICAS` 描述了集群中运行了多少个DNS pod。

默认情况下，我们也可以配置kube-ui插件。

```
ENABLE_CLUSTER_UI="${KUBE_ENABLE_CLUSTER_UI:-true}"
```

当以上参数设置完毕之后，运行以下命令。

```
$ cd cluster/ubuntu  
$ KUBERNETES_PROVIDER=ubuntu ./deployAddons.sh
```

稍等之后，你可以用命令 `$ kubectl get pods --namespace=kube-system` 来检测这个集群中的DNS和pod的UI是否运行。

下一步

我们目前的工作集中在以下的功能：1.使用[kube-in-docker](#)在Docker中运行kubernetes，从而消除不同操作系统的区别。2.拆除部署脚本：一键式清除和重建整个部署。

故障排除

通常，这一步非常简单：1.下载和复制安装和配置文件到在每个节点上。2.根据用户的输入的IP来配置主节点上的 `etcd`。3.在每个普通节点上新建和启动Flannel网络。

如果你遇到什么问题，首先检查主节点上 `etcd` 的配置。1.查看 `/var/log/upstart/etcd.log` 中的日志。2.以下命令或许有帮助，第一个是停止这个集群，第二个是启动这个集群。

```
$ KUBERNETES_PROVIDER=ubuntu ./kube-down.sh  
$ KUBERNETES_PROVIDER=ubuntu ./kube-up.sh
```

3.你也可以在 `/etc/default/{component_name}` 里做客制化设置，之后用以下命令重启服务 `$ sudo service {component_name} restart`。

升级集群

如果你已经有了Kubernetes集群并希望升级到新的版本，你需要根据 `cluster/` 文件夹中的命令来升级整个或部分集群。

```
$ KUBERNETES_PROVIDER=ubuntu ./kube-push.sh [-m|-n]
```

默认情况下会升级全部的节点，你也可以用 `-m` 来升级主节点或 `-n` 来升级某个普通节点。如果没有给出所要升级的版本，这个脚本会尝试使用本地的二进制安装文件。你应该确认所有的文件都在 `cluster/ubuntu/binaries` 文件夹中准备好了。

```
$ tree cluster/ubuntu/binaries
binaries/
├── kubectl
├── master
│   ├── etcd
│   ├── etcdctl
│   ├── flanneld
│   ├── kube-apiserver
│   ├── kube-controller-manager
│   └── kube-scheduler
└── minion
    ├── flanneld
    ├── kubelet
    └── kube-proxy
```

用以下命令来获得帮助。

```
$ KUBERNETES_PROVIDER=ubuntu ./kube-push.sh -h
```

这里有几个实例：

- 把主节点升级到版本1.0.5: `$ KUBERNETES_PROVIDER=ubuntu ./kube-push.sh -m 1.0.5`
- 把节点10.10.103.223升级到版本1.0.5：`$ KUBERNETES_PROVIDER=ubuntu ./kube-push.sh -n 10.10.103.223 1.0.5`
- 把主节点和其他节点升级到版本1.0.5: `$ KUBERNETES_PROVIDER=ubuntu ./kube-push.sh 1.0.5`

这个脚本不会删除你集群上的资源，只是替换了二进制安装包。

测试

你可以运行 `kubectl` 命令来检测刚升级的Kubernetes集群是否工作正常，参考[测试](#)。为了确保升级后的集群版本和你期望的一致，以下命令会有所帮助。

- 升级主节点或所有节点: `$ kubectl version`。 检查服务器版本。
- 升级节点10.10.102.223: `$ ssh -t vcap@10.10.102.223 'cd /opt/bin && sudo ./kubelet --version'`

利用Docker安装多节点Kubernetes

译者：王乐 校对：无

注意：这个介绍一定程度上会比[single node](#)里的介绍要更进一步。如果你有兴趣探索Kubernetes，我们建议你从这里开始。注意：Docker 1.7.0里的一个[bug](#)影响Docker上多节点的正常安装。请安装Docker版本1.6.2或1.7.1。

前提条件

- 你需要一台安装有正确版本的Docker的主机。

概括介绍

本指南会指导架设有2个节点的Kubernetes集群。其中包括一个支持API服务器和编排工作的主节点，和一个从主节点接受任务的从节点。你可以按照同样的步骤添加任意数量的从节点，从而假设一个庞大的集群。

这里的图标展示了最终的结果：



引导启动Docker

本指南同样运行两个Docker后台程序实例的模式 1) 一个引导启动的Docker容器实例用来运行例如 `flanneld` 和 `etcd` 系统后台程序。2) 一个`_主_Docker`容器实例来服务Kubernetes和用户容器。

这个模式是有必要的，因为 `flannel` 的后台程序是负责建立和管理Kubernetes新建的Docker容器间的相互通信。所以 `flannel` 必须要运行在`主_Docker`后台程序之外。为了利用容器来方便部署和管理，我们使用了这个较简单的引导启动`_`的Docker后台程序来实现这一点。

在安装前你可以在每个节点上选择k8s的版本：

```
export K8S_VERSION=<your_k8s_version (e.g. 1.0.3)>
```

否则，我们会使用最新的 `hyperkube` 镜像当作默认k8s版本。

主节点

第一步是初始化主节点。 复制Kubernetes在Github上的repo，并在主节点的主机上以root身份运行脚本[master.sh](#):

```
cd kubernetes/docs/getting-started-guides/docker-multinode/  
./master.sh
```

Master done!

参考[这里](#) for detailed instructions explanation.

添加从节点

当主节点正常运行后，你可以在不同的主机上添加更多的从节点。

复制Kubernetes在Github上的repo，并在从节点的主机上以root身份运行脚本[worker.sh](#):

```
export MASTER_IP=<your_master_ip (e.g. 1.2.3.4)>  
cd kubernetes/docs/getting-started-guides/docker-multinode/  
./worker.sh
```

Worker done!

参考[这里](#) for detailed instructions explanation.

部署DNS

参考[这里](#) for instructions.

测试你的集群

一旦你的集群创建完毕，你可以进行[测试](#)

请参见[examples directory](#)里更多的完整应用。

如何在Mesos上运行Kubernetes

译者：王乐 校对：无

关于在Mesos上运行Kubernetes

Mesos允许Kubernetes和其他Mesos构架，例如：[Hadoop](#),[Spark](#)和[Chronos](#)来动态分享集群资源。

Mesos确保同一集群上的运行的不同架构上的应用之间相互隔离，和较公平资源的分配。

Mesos集群几乎可以部署在任何IaaS云服务上或是你自己的数据中心。当Kubernetes运行在Mesos上，你可以轻松将Kubernetes上运行的任务在任意云平台之间转移。

本指南将具体介绍如何架设在Mesos集群上运行Kubernetes。我们一步一步指导你怎么在Mesos集群上添加Kubernetes和如何启动你的第一个pod来运行nginx网站服务器。

注意: [现有实现上的已知问题](#)，集中式的监控和日志功能尚不支持。如果你发现以下步骤有什么问题，请[在kubernetes-mesos项目报告这个问题](#)。

你可以在[contrib directory](#)找到进一步的有关信息。

前提条件

- 理解[Apache Mesos](#)
- 一个运行在[Google Compute Engine](#)的Mesos集群
- 一个连接到这个集群[VPN连接](#)
- 一台在集群里并可以作为Kubernetes主节点的主机，这个主机需安装一下软件：
 - GoLang > 1.2
 - make (i.e. build-essential)
 - Docker

注意: 你可以，但是你不要在运行Mesos master的主机上部署Kubernetes-Mesos。

部署Kubernetes-Mesos

用SSH登入Kubernetes主节点。参考以下命令行时需要替换其中的地址为正确的IP地址。

```
ssh jclouds@${ip_address_of_master_node}
```

编译Kubernetes-Mesos。

```
git clone https://github.com/kubernetes/kubernetes
cd kubernetes
export KUBERNETES_CONTRIB=mesos
make
```

设置环境变量。可以用 `hostname -i` 来获得主节点的内部IP地址。

```
export KUBERNETES_MASTER_IP=$(hostname -i)
export KUBERNETES_MASTER=http://${KUBERNETES_MASTER_IP}:8888
```

请注意`KUBERNETES_MASTER`用来定义API的endpoint的。如果 `~/.kube/config` 已存在并指向其他endpoint, 你需要在最后一步使用`kubectl`时加上 `--server=${KUBERNETES_MASTER}` 选项。

部署etcd

启动etcd并监测它是否正常运行：

```
sudo docker run -d --hostname $(uname -n) --name etcd \
-p 4001:4001 -p 7001:7001 quay.io/coreos/etcd:v2.0.12 \
--listen-client-urls http://0.0.0.0:4001 \
--advertise-client-urls http://${KUBERNETES_MASTER_IP}:4001
```

\$ sudo docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
fd7bac9e2301 quay.io/coreos/etcd:v2.0.12 "/etcd" 5s ago Up 3s 2379/tcp, 2380/.

最好检测一下你可以连接到你的etcd服务

```
curl -L http://${KUBERNETES_MASTER_IP}:4001/v2/keys/
```

如果连接没有问题，你可以看见所显示的etcd秘钥（如果说有的话）。

启动Kubernetes-Mesos服务

设置正确的环境变量 `PATH` 以便访问Kubernetes-Mesos中的二进制运行文件：

```
export PATH="$(pwd)/_output/local/go/bin:$PATH"
```

确认你的Mesos master：根据具体Mesos的安装，Mesos master的地址可能会是 host:port 的样子，比如 mesos-master:5050 的 host:port，也可能是ZooKeeper的URL，比如 zk://zookeeper:2181/mesos。在生产环境中，推荐使用ZooKeeper的URL，这样可避免Kubernetes在Mesos master的地址有变化的时候出现问题。

```
export MESOS_MASTER=<host:port or zk:// url>
```

在当前目录下新建一个云配置文件 mesos-cloud.conf，文件内写入一下内容：

```
$ cat <mesos-cloud.conf
[mesos-cloud]
mesos-master = ${MESOS_MASTER}
EOF
```

现在启动kubernetes-mesos的API服务和主节点上的scheduler：

```
$ km apiserver \
--address=${KUBERNETES_MASTER_IP} \
--etcd-servers=http://${KUBERNETES_MASTER_IP}:4001 \
--service-cluster-ip-range=10.10.10.0/24 \
--port=8888 \
--cloud-provider=mesos \
--cloud-config=mesos-cloud.conf \
--secure-port=0 \
--v=1 >apiserver.log 2>&1 &

$ km controller-manager \
--master=${KUBERNETES_MASTER_IP}:8888 \
--cloud-provider=mesos \
--cloud-config=./mesos-cloud.conf \
--v=1 >controller.log 2>&1 &

$ km scheduler \
--address=${KUBERNETES_MASTER_IP} \
--mesos-master=${MESOS_MASTER} \
--etcd-servers=http://${KUBERNETES_MASTER_IP}:4001 \
--mesos-user=root \
--api-servers=${KUBERNETES_MASTER_IP}:8888 \
--cluster-dns=10.10.10.10 \
--cluster-domain=cluster.local \
--v=2 >scheduler.log 2>&1 &
```

使用Disown显示后台运行的任务。

```
disown -a
```

验证KM服务

设置正确的环境变量 PATH 以便访问kubectl：

```
export PATH=<path/to/kubernetes-directory>/platforms/linux/amd64:$PATH
```

使用 kubectl 和kubernetes-mesos交互：

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

```
# 注意：显示的service IP有可能不同
$ kubectl get services
NAME           LABELS
k8sm-scheduler component=scheduler,provider=k8sm
kubernetes     component=apiserver,provider=kubernetes
                SELECTOR   IP(S)        PORT
                    10.10.10.113  10251/TCP
                    10.10.10.1    443/TCP
```

最后，登陆地址 `http://<mesos-master-ip:port>` 访问Kubernetes服务网页。确定你有有效的VPN连接。点击 `Frameworks` 标签并寻找一个名叫"Kubernetes"的激活framework。

启动一个pod

在本地文件里写入描述pod的JSON：

```
$ cat <<EOPOD >nginx.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
EOPOD
```

用 kubectl CLI把这个pod的描述发送到Kubernetes：

```
$ kubectl create -f ./nginx.yaml
pods/nginx
```

`dockerd` 会从Internet下载相对应的镜像，这个过程大概持续一到两分钟的时间。我们可以用 `kubectl` CLI去观察pod的状态：

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          14s
```

在Mesos的网页上确认每一个pod都正常运行。点击Kubernetes的framework选项后，会显示在Kubernetes启动pod的Mesos任务。

启动kube-dns

Kube-dns是为Kubernetes集群提供基于DNS的发现服务的插件。参见[Kubernetes里的DNS](#)以获得更多信息。

`kube-dns` 插件运行在集群内的pod上。这个pod包涵三个运行在一起的容器：

- 一个本地etc实例
- DNS服务器`skydns`
- 完成Kubernetes集群状态到skydns交互的`kube2sky`进程

`skydns`容器通过53端口向集群提供DNS服务。etcd communication works via local 127.0.0.1 communication

我们假设 `kube-dns`会使用

- service IP `10.10.10.10`
- 和 `cluster.local` 域名。

注意这两个值已经在设置apiserver时使用了。

在[cluster/addons/dns/skydns-rc.yaml.in](#)目录下可以找到用一个replication controller建立一个有3个容器的pod的模版。为了获得一个正确的replication controller的yaml文件，以下这几步是必须的：

- 将`替换为`1`
- 将`替换为`cluster.local.`
- 将参数`--kube_master_url=${KUBERNETES_MASTER}` 添加到`kube2sky`容器的命令行中

除此之外service模版[cluster/addons/dns/skydns-svc.yaml.in](#)需要一下变动：

- 将 ` 替换为 10.10.10.10`.

自动化这一步：

```
sed -e "s/{{ pillar\['dns_replicas'\] }}/1/g;" \
"s,\(command = \"/kube2sky\"\),\\1\\\"$'\n'"      --kube_master_url=${KUBERNETES_MASTE
"s/{{ pillar\['dns_domain'\] }}/cluster.local/g" \
cluster/addons/dns/skydns-rc.yaml.in > skydns-rc.yaml
sed -e "s/{{ pillar\['dns_server'\] }}/10.10.10.10/g" \
cluster/addons/dns/skydns-svc.yaml.in > skydns-svc.yaml
```

现在 kube-dns pod 和 service 已经启动了：

```
kubectl create -f ./skydns-rc.yaml
kubectl create -f ./skydns-svc.yaml
```

使用 `kubectl get pods --namespace=kube-system` 来检查 pod 上 3 个容器正常运行。请注意运行 kube-dns 的 pod 是在 `kube-system` 的命名空间下，不是在 `default` 下面。

为了监测新的 DNS 服务正常，我们启动一个 pod 运行 busybox 来查找 DNS。第一步新建一个 `busybox.yaml` 文件：

```
cat <<EOF >busybox.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: busybox
    restartPolicy: Always
EOF
```

之后启动 pod：

```
kubectl create -f ./busybox.yaml
```

当pod启动和正常运行后，运行以下命令找到Kubernetes主节点的service的地址。通常情况下地址为10.10.10.1。

```
kubectl exec busybox -- nslookup kubernetes
```

如果一切运行正常，你会看到一下显示：

```
Server: 10.10.10.10
Address 1: 10.10.10.10

Name: kubernetes
Address 1: 10.10.10.1
```

下一步？

试着运行一下[Kubernetes examples](#).

阅读在[Mesos](#)上运行Kubernetes的架构[contrib directory](#).

注意：一些示例需要在集群上安装Kubernetes DNS。我们未来会在本指南里加入如何支持Kubernetes DNS的介绍的。

注意：请注意这里有一些[known issues with the current Kubernetes-Mesos implementation](#).

Kubernetes用户指南：应用程序管理

译者：钟健鑫 校对：无

目录

- [Kubernetes用户指南：应用程序管理](#)
 - [实战入门](#)
 - [进阶路线](#)
 - [概念指南](#)
 - [阅读延伸](#)

用户指南旨在为了每一个想运行程序或者服务在已有Kubernetes集群中的人。

Kubernetes集群的安装和管理相关内容在[集群管理指南](#)中可以找到。开发者指南是为了那些想写代码直接访问Kubernetes的API,或者为Kubernetes项目共享代码的每一个人。

请确认你已经完成这个事列：[运行用户指南中实例的先决条件](#).

实战入门

- | [Kubernetes 101](#)
- | [Kubernetes 201](#)

进阶路线图

如果你对Kubernetes布什很熟悉的话, 我们建议你按顺序阅读下面的部分:

- 1.快速入门: 运行并展示一个应用程序
- 2.容器的配置与启动: 配置容器的公共参数
- 3.部署可持续运行的容器
- 4.链接应用程序: 将应用程序展示给客户和用户
- 5.在生产环境中使用容器
- 6.部署管理
- 7.应用程序的自我检查和调试
 - 1.使用Kubernetes的Web用户界面
 - 2.日志
 - 3.监控
 - 4.通过exec命令进入容器
 - 5.通过代理连接容器

- 6.通过接口转发连接容器

概念指南

概论：Kubernetes中概念的简要概述

Cluster：集群是指由Kubernetes使用一系列的物理机、虚拟机和其他基础资源来运行你的应用程序。

Node：一个node就是一个运行着Kubernetes的物理机或虚拟机，并且pod可以在其上面被调度。.

Pod：一个pod对应一个由相关容器和卷组成的容器组

Label：一个label是一个被附加到资源上的键/值对，譬如附加到一个Pod上，为它传递一个用户自定的并且可识别的属性.Label还可以被应用来组织和选择子网中的资源

selector是一个通过匹配labels来定义资源之间关系得表达式，例如为一个负载均衡的service指定所目标Pod.

Replication Controller：replication controller 是为了保证一定数量被指定的Pod的复制品在任何时间都能正常工作.它不仅允许复制的系统易于扩展，还会处理当pod在机器在重启或发生故障的时候再次创建一个

Service：一个service定义了访问pod的方式，就像单个固定的IP地址和与其相对应的DNS名之间的关系。

Volume：一个volume是一个目录，可能会被容器作为未见系统的一部分来访问。

Kubernetes volume 构建在Docker Volumes之上,并且支持添加和配置volume目录或者其他存储设备。

Secret：Secret 存储了敏感数据，例如能允许容器接收请求的权限令牌。

Name：用户为Kubernetes中资源定义的名字

Namespace：Namespace 好比一个资源名字的前缀。它帮助不同的项目、团队或是客户可以共享cluster,例如防止相互独立的团队间出现命名冲突

Annotation：相对于label来说可以容纳更大的键值对，它对我们来说可能是不可读的数据，只是为了存储不可识别的辅助数据，尤其是一些被工具或系统扩展用来操作的数据

阅读延伸

API 相关资源

更多可使用的API资源

Pods and containers

- Pod 的生命周期和重启策略
- 生命周期事件钩子
- 计算资源，例如cpu和内存
- 指定命令和请求的能力
- Downward API（用于向下支持容器访问Pod的API,如Docker）：从pod中访问系统配置
- 镜像和仓库
- 从使用docker-cli转向kubectl
- 配置集群时的提示和技巧
- 把pod分配到指定node
- 展示为一组正在运行的pod左滚动更新

Pods

译者 : kz 校对 : 无

在Kubernetes中，与用采用单独的应用容器方式不同，pod是最小的部署单元，可以对其进行创建，调度和管理操作。

pod是什么？

A *pod* (as in a pod of whales or pea pod) corresponds to a colocated group of applications running with a shared context. Within that context, the applications may also have individual cgroup isolations applied. A pod models an application-specific "logical host" in a containerized environment. It may contain one or more applications which are relatively tightly coupled — in a pre-container world, they would have executed on the same physical or virtual host.

一个pod（正如鲸群中的群，或者豌豆荚中的豆）对应着处在一个共享情景的一组共存一组应用。在该上下文中，应用可能使用了单独的cgroup隔离。一个pod将处在一个容器化环境的以应用为中心的“逻辑主机”。它可能包含一个或者多个应用，它们之间很可能紧密耦合 - 在一容器之前的世界里，他们本应该在同样的物理或者虚拟主机上。

The context of the pod can be defined as the conjunction of several Linux namespaces:

- PID namespace (applications within the pod can see each other's processes)
- network namespace (applications within the pod have access to the same IP and port space)
- IPC namespace (applications within the pod can use SystemV IPC or POSIX message queues to communicate)
- UTS namespace (applications within the pod share a hostname)

pod的上下文可以被定义成Linux几种命名空间的结合：

- PID命名空间（处在同一个pod中的应用可以看到彼此的进程）
- 网络命名空间（处在同一个pod中的应用能访问一样的IP和端口空间）
- IPC命名空间（处在同一个pod中的应用可以用SystemV IPC或者POSIX消息队列来进行通讯）
- UTS命名空间（处在同一个pod中的应用公用一个主机名）

Applications within a pod also have access to shared volumes, which are defined at the pod level and made available in each application's filesystem. Additionally, a pod may define top-level cgroup isolations which form an outer bound to any individual isolation applied to

constituent applications.

In terms of Docker constructs, a pod consists of a colocated group of Docker containers with shared volumes. PID namespace sharing is not yet implemented with Docker.

Like individual application containers, pods are considered to be relatively ephemeral rather than durable entities. As discussed in [life of a pod](#), pods are scheduled to nodes and remain there until termination (according to restart policy) or deletion. When a node dies, the pods scheduled to that node are deleted. Specific pods are never rescheduled to new nodes; instead, they must be replaced (see [replication controller](#) for more details). (In the future, a higher-level API may support pod migration.)

处在同一个pod中的应用也能访问一样的的共享volume，卷是在pod的层级上定义的，并且在在每个应用的文件系统中被设为可用。另外，一个pod也可以定义顶级的cgroup隔离，这给任何一个应用到应用到单独的隔离构成了一个外围的约束。

至于Docker的构成，一个pod包含一组共存一组Docker容器，他们有共享的卷。PID的命名空间对于Docker来说还没有实现。

就如单独的应用容器一样，pod被认为是比较临时性的而不是持久性的实体。就如在一个pod的生命周期中讨论到的，pod被调度到node然后直到被结束或者删除前一直被保留在那里。当一个node挂掉时，被调度到该node的pod会被删除。特定的pod从来不会被调度到新的节点；他们必须被替换。（在以后，一个更高层次的API可能会支持pod的迁移。）

Motivation for pods

Resource sharing and communication

Pods facilitate data sharing and communication among their constituents.

The applications in pod all use the same network namespace/IP and port space, and can find and communicate with each other using localhost. Each pod has an IP address in a flat shared networking namespace that has full communication with other physical computers and containers across the network. The hostname is set to the pod's Name for the application containers within the pod. [More details on networking](#).

In addition to defining the application containers that run in the pod, the pod specifies a set of shared storage volumes. Volumes enable data to survive container restarts and to be shared among the applications within the pod.

Management

Pods also simplify application deployment and management by providing a higher-level abstraction than the raw, low-level container interface. Pods serve as units of deployment and horizontal scaling/replication. Co-location (co-scheduling), fate sharing, coordinated replication, resource sharing, and dependency management are handled automatically.

Pod的设计动机

资源共享与通讯

Pod有助于数据的在组成成员之间的共享和通讯。

一个pod中应用使用相同的网络命名空间/IP和端口空间，可以使用localhost找到彼此并进行通信。每一个pod在扁平的共享网络命名空间中有一个IP地址，能与网络上其他的物理计算机和容器进行完整的通讯。pod中的应用容器的主机名设置成pod的名称。

除了定义运行在pod中应用容器之外，pod指定了一组共享的存储卷。卷能让数据不受容器重启的影响，并且可以在pod的容器之间共享。

管理

通过提供更高层次的抽象，而不是提供一个原始，低层次的容器接口，pod也简化了应用的部署与管理。pod作为部署的单元，也是水平伸缩复制的单元。同地点安放，命运共享，协作的副本复制，资源贡献，和依赖管理都能自动的处理。

Uses of pods

Pods can be used to host vertically integrated application stacks, but their primary motivation is to support co-located, co-managed helper programs, such as:

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

Individual pods are not intended to run multiple instances of the same application, in general.

For a longer explanation, see [The Distributed System ToolKit: Patterns for Composite Containers](#).

Pod的使用

Pod可以用来host垂直集成的应用栈，但是它们主要的动机还是在于支持同地安放，共同管理的助手程序，比如：

- 内容管理系统，文件和数据加载器，本地缓存管理器等等
- 日志和检查点备份，压缩，滚动和快照等等
- 数据更改监视者，日志尾部查看器，日志和监控适配器，事件发布器等等
- 代理，桥，和适配器
- 控制器，管理器，配置器，和更新器等等

单独的pod总体来说不适合拿来运行同一应用的多个实例。

要更详细的解释，请参看，分布式系统的工具包：组合容器的模式。

Alternatives considered

Why not just run multiple programs in a single (Docker) container?

1. Transparency. Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
2. Decoupling software dependencies. The individual containers may be rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday.
3. Ease of use. Users don't need to run their own process managers, worry about signal and exit-code propagation, etc.
4. Efficiency. Because the infrastructure takes on more responsibility, containers can be lighter weight.

Why not support affinity-based co-scheduling of containers?

That approach would provide co-location, but would not provide most of the benefits of pods, such as resource sharing, IPC, guaranteed fate sharing, and simplified management.

考虑过的其他方案

为什么不在一个单独的容器里面运行多个程序？

1. 透明度。让处在pod中的容器对于基础设施可见，使得基础设施能向这些容器提供服务，比如进程管理，和资源监控。这有助于为用户提供很多的便利。

2. 解耦软件依赖。独立的容器可以单独的重建并重新部署。Kubernetes可能甚至支持在线更新单独的容器。
3. 使用的容易度。用户不用运行他们自己的进程管理器，担心信号和退出代码的串升，等等。
4. 效率。因为基础设施承担了更多的责任，容器可以做的更加轻量。

为什么不支持容器基于亲和力的共同调度？

这种方式可以提供共同安放的特性，但是不会提供pod的其他绝大部分的优点，例如资源共享，IPC，有保证的命运共享，和简化的管理。

Durability of pods (or lack thereof)

Pods aren't intended to be treated as durable [pets](#). They won't survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

In general, users shouldn't need to create pods directly. They should almost always use controllers (e.g., [replication controller](#)), even for singletons. Controllers provide self-healing with a cluster scope, as well as replication and rollout management.

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including [Borg](#), [Marathon](#), [Aurora](#), and [Tupperware](#).

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to "proxy" them via controller APIs
- decoupling of pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services — the endpoint controller just watches pods
- clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the "pod controller"
- high-availability applications, which will expect pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions, image prefetching, or live pod migration [#3949](#)

The current best practice for pets is to create a replication controller with `replicas` equal to `1` and a corresponding service. If you find this cumbersome, please comment on [issue #260](#).

pod的持久性

Pod不想被当做持久化的宠物。他们无法幸免于调度错误，节点失效，或者其他故障，比如资源的短缺，或者在节点处于维护状态时。

总体来说，用户不应该直接的创建pod。他们绝大多数情况下只应该使用控制器，即使对于单例也如此。控制器在一个集群的范围累提供了自我修复的功能，和复制以及扩展开的管理。

在集群管理系统中，使用一系列API作为主要的面对用户的原语是相对常见的做法，包含Borg, Marathon, Aurora和Tupperware。

Pod作为原语暴露出来是为了有助于：

- 调度器和控制器的插件性
- 支持pod级别的操作，不用通过控制器API来对他们进行代理
- 将pod的生命周期和控制器的生命周期解耦，比如为了启动
- 将controller和service解耦 - 端点控制器只监视pod
- 清理Kubelet级别的功能，取而代之集群级别的功能 - Kubelet是实际上的pod控制器
- 高可用的应用，pod在被结束前会被取代，自然在删除之前也会被取代，比如在有计划的踢出，镜像预取，和在线的pod迁移的情况下。

目前对于宠物的最佳实践是，创建一个副本等于1和有对应service的一个replication控制器。如果你觉得这太麻烦，请在这里留下你的意见。

Termination of Pods

容器的终止

Because pods represent running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (vs being violently killed with a KILL signal and having no chance to clean up). Users should be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When a user requests deletion of a pod the system records the intended grace period before the pod is allowed to be forcefully killed, and a TERM signal is sent to the main process in each container. Once the grace period has expired the KILL signal is sent to those processes and the pod is then deleted from the API server. If the Kubelet or the container manager is restarted while waiting for processes to terminate, the termination will be retried with the full grace period.

因为pod代表着一个集群中节点上运行的进程，让这些进程不再被需要，优雅的退出是很重要的（与粗暴的用一个KILL信号去结束，让应用没有机会进行清理操作）。用户应该能请求删除，并且在室进程终止的情况下能知道，而且也能保证删除最终完成。当一个用户请求删除pod，系统记录想要的优雅退出时间段，在这之前Pod不允许被强制的杀死，TERM信号会发

送给容器主要的进程。一旦优雅退出的期限过了，KILL信号会送到这些进程，pod会从API服务器其中被删除。如果在等待进程结束的时候，Kubelet或者容器管理器重启了，结束的过程会带着完整的优雅退出时间段进行重试。

An example flow:

1. User sends command to delete Pod, with default grace period (30s)
2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period.
3. Pod shows up as "Terminating" when listed in client commands
4. (simultaneous with 3) When the Kubelet sees that a Pod has been marked as terminating because the time in 2 has been set, it begins the pod shutdown process.
 - i. If the pod has defined a [preStop hook](#), it is invoked inside of the pod. If the `prestop` hook is still running after the grace period expires, step 2 is then invoked with a small (2 second) extended grace period.
 - ii. The processes in the Pod are sent the TERM signal.
5. (simultaneous with 3), Pod is removed from endpoints list for service, and are no longer considered part of the set of running pods for replication controllers. Pods that shutdown slowly can continue to serve traffic as load balancers (like the service proxy) remove them from their rotations.
6. When the grace period expires, any processes still running in the Pod are killed with SIGKILL.
7. The Kubelet will finish deleting the Pod on the API server by setting grace period 0 (immediate deletion). The Pod disappears from the API and is no longer visible from the client.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows a user to override the default and specify their own value. The value `0` indicates that delete should be immediate, and removes the pod in the API immediately so a new pod can be created with the same name. On the node pods that are set to terminate immediately will still be given a small grace period before being force killed.

一个示例流程：

1. 用户发送一个命令来删除Pod，默认的优雅退出时间是30秒
2. API服务器中的Pod更新时间，超过该时间Pod被认为死亡
3. 在客户端命令的里面，Pod显示为"Terminating (退出中)"的状态
4. （与第3同时）当Kubelet看到Pod标记为退出中的时候，因为第2步中时间已经设置了，它开始pod关闭的流程
 - i. 如果该Pod定义了一个停止前的钩子，其会在pod内部被调用。如果钩子在优雅退出时间段超时仍然在运行，第二步会意一个很小的优雅时间断被调用

- ii. 进程被发送TERM的信号
- 5. (与第三步同时进行) Pod从service的列表中被删除, 不在被认为是运行着的pod的一部分。缓慢关闭的pod可以继续对外服务, 当负载均衡器将他们轮流移除。
- 6. 当优雅退出时间超时了, 任何pod中正在运行的进程会被发送SIGKILL信号被杀死。
- 7. Kubelet会完成pod的删除, 将优雅退出的时间设置为0(表示立即删除)。pod从API中删除, 不在对客户端可见。

默认情况下, 所有的删除操作的优雅退出时间都在30秒以内。kubectl delete命令支持--grace-period=的选项, 以运行用户来修改默认值。0表示删除立即执行, 并且立即从API中删除pod这样一个新的pod会在同时被创建。在节点上, 被设置了立即结束的的pod, 仍然会给一个很短的优雅退出时间段, 才会开始被强制杀死。

Privileged mode for pod containers

pod容器的特权模式

From kubernetes v1.1, any container in a pod can enable privileged mode, using the `privileged` flag on the `SecurityContext` of the container spec. This is useful for containers that want to use linux capabilities like manipulating the network stack and accessing devices. Processes within the container get almost the same privileges that are available to processes outside a container. With privileged mode, it should be easier to write network and volume plugins as separate pods that don't need to be compiled into the kubelet.

If the master is running kubernetes v1.1 or higher, and the nodes are running a version lower than v1.1, then new privileged pods will be accepted by api-server, but will not be launched. They will be pending state. If user calls `kubectl describe pod FooPodName`, user can see the reason why the pod is in pending state. The events table in the describe command output will say: `Error validating pod "FooPodName"."FooPodNamespace" from api, ignoring: spec.containers[0].securityContext.privileged: forbidden '<*>(0xc2089d3248)true'`

If the master is running a version lower than v1.1, then privileged pods cannot be created. If user attempts to create a pod, that has a privileged container, the user will get the following error: `The Pod "FooPodName" is invalid. spec.containers[0].securityContext.privileged: forbidden '<*>(0xc20b222db0)true'`

对于kubernetes v1.1来说, 任何pod中的容器都可以启用特权模式, 在容器定义的SecurityContext里设置privileged标记。这对于想使用linux能力的容器来说是有用, 比如想利用网络栈并访问设备。容器之外的进程能获得的特权, 容器中的进程会获得同样的特权。在特权模式下, 编写网络和卷插件会更容易, 因为单独的pod不需要被编译进kubelet。

API Object

Pod is a top-level resource in the kubernetes REST API. More details about the API object can be found at: [Pod API object](#).

API对象

在kubernetes的REST API中， pod是一个顶级的资源。更多的详情可以在这里找到。

Labels

译者 : kz 校对 : 无

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but which do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

```
"labels": {  
    "key1" : "value1",  
    "key2" : "value2"  
}
```

We'll eventually index and reverse-index labels for efficient queries and watches, use them to sort and group in UIs and CLIs, etc. We don't want to pollute labels with non-identifying, especially large and/or structured, data. Non-identifying information should be recorded using [annotations](#).

Labels是附加到如pod等对象的键值对。Label旨在用来指定对象那些用于辨识性目的的属性，这些属性与用户相关性大，他们来说具有意义，但是这些属性却对于不直接表达核心系统的语义（semantics）。Label可以用来组织并且选择对象的子集。Label可以在对象创建的时候依附到对象上，随后可以在任意时间被添加和修改。每一个对象可以定义一组键值对label。每一个键对于一个给定的对象必须是唯一不可重复。

```
"labels": {  
    "key1" : "value1",  
    "key2" : "value2"  
}
```

我们最终会索引并且反向索引（reverse-index）labels，以获得更高效的查询和监视，把他们用到UI或者CLI中用来排序或者分组等等。我们不想用那些不具有指认效果的label来污染label，特别是那些体积较大和结构型的数据。不具有指认效果的信息应该使用annotation来记录。

Motivation

目标

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings.

Service deployments and batch processing pipelines are often multi-dimensional entities (e.g., multiple partitions or deployments, multiple release tracks, multiple tiers, multiple micro-services per tier). Management often requires cross-cutting operations, which breaks encapsulation of strictly hierarchical representations, especially rigid hierarchies determined by the infrastructure rather than by users.

Example labels:

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

These are just examples; you are free to develop your own conventions.

Label可以让用户将他们自己的有组织目的的结构以一种松耦合的方式应用到系统的对象上，且不需要客户端存放这些对应关系 (mappings)。

服务部署和批处理管道通常是多维的实体（例如多个分区或者部署，多个发布轨道，多层，每层多微服务）。管理通常需要跨域式的切割操作，这会打破有严格层级展示关系的封装，特别对那些是由基础设施而非用户决定的很死板的层级关系。

一些示例label:

- "发行版本 (release) " : " (稳定版) stable" , " (发行版) release" : "canary"
- "环境 (environment) " : " (开发) dev" , "environment" : "qa" , "environment" : " (生产) production"
- "tier" : "frontend" , "tier" : "backend" , "tier" : "cache"
- "分区 (partition) " : " (客户A) customerA" , "partition" : "customerB"
- "track" : "daily" , "track" : "weekly"

Syntax and character set

语法和字符集

Labels are key value pairs. Valid label keys have two segments: an optional prefix and name, separated by a slash (/). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (_), dots (.), and alphanumerics between. The prefix is optional. If specified, the prefix must be a DNS subdomain: a series of DNS labels separated by dots (.), not longer than 253 characters in total, followed by a slash (/). If the prefix is omitted, the label key is presumed to be private to the user. Automated system components (e.g. `kube-scheduler`, `kube-controller-manager`, `kube-apiserver`, `kubectl`, or other third-party automation) which add labels to end-user objects must specify a prefix. The `kubernetes.io/` prefix is reserved for Kubernetes core components.

Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character ([a-z0-9A-Z]) with dashes (-), underscores (_), dots (.), and alphanumerics between.

Labels是键值对。合法的键值对有两个部分：一个可选的前缀和名字，由“/”分隔。名字部分是必须，长度一定是64个字符或者更短，首位字符必须是字母数字型（[a-z0-9A-Z]），其他地方的字符必须是横线，下划线，点，或者其他字母数字字符。前缀是可选的。如果没有指定，前缀必须是一个dns的子域：一系列有点分隔的DNS label，总长度不能超过253个字符，以下划线“/”结尾。如果前缀被省略掉了，label的键会被认为是对于用户私有的。系统自动的组件（如kube-scheduler, kube-controller-manager, kube-apiserver, kubectl或者其他第三方自动工具）如要向用户最终的对象添加label，必须指定前缀。前缀 `kubernetes.io/` 是保留给Kubernetes核心组件用的。

合法的label值必须是63个或者更短的字符。要么是空，要么首位字符必须为字母数字字符，中间必须是横线，下划线，点或者数字字母。

Label选择器

Unlike [names and UIDs](#), labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: *equality-based* and *set-based*. A label selector can be made of multiple *requirements* which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as an *AND* logical operator.

An empty label selector (that is, one with zero requirements) selects every object in the collection.

A null label selector (which is only possible for optional selector fields) selects no objects.

与name和UID不同，label不提供唯一性。通常，我们会看到很多对象有着一样的label。

通过label选择器，客户端/用户能方便辨识出一组对象。label选择器是kubernetes中核心的组织原语。

API目前支持两种选择器：基于相等的和基于集合的。一个label选择器一可以由多个必须条件组成，由逗号分隔。在多个必须条件指定的情况下，所有的条件都必须满足，因而逗号起着AND逻辑运算符的作用。

一个空的label选择器（即有0个必须条件的选择器）会选择集合中的每一个对象。

一个null型label选择器（仅对于可选的选择器字段才可能）不会返回任何对象。

Equality-based requirement

基于相等性的条件

Equality- or inequality-based requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well. Three kinds of operators are admitted `=`, `==`, `!=`. The first two represent *equality* (and are simply synonyms), while the latter represents *inequality*. For example:

```
environment = production
tier != frontend
```

The former selects all resources with key equal to `environment` and value equal to `production`. The latter selects all resources with key equal to `tier` and value distinct from `frontend` , and all resources with no labels with the `tier` key. One could filter for resources in `production` excluding `frontend` using the comma operator:

```
environment=production,tier!=frontend
```

基于相等性或者不相等性的条件允许用label的键或者值进行过滤。匹配的对象必须满足所有指定的label约束，尽管他们可能也有额外的label。有三种运算符是允许的，“`=`”，“`==`”和“`!=`”。前两种代表相等性（他们是同义运算符），后一种代表非相等性。例如：

```
environment = production
tier != frontend
```

第一个选择所有键等于 `environment` 值为 `production` 的资源。后一种选择所有键为 `tier` 值不等于 `frontend` 的资源，和那些没有键为 `tier` 的label的资源。

要过滤所有处于 `production` 但不是 `frontend` 的资源，可以使用逗号操作符，`environment=production,tier!=frontend`。

基于set的条件

Set-based label requirements allow filtering keys according to a set of values. Three kinds of operators are supported: `in`, `notin` and `exists` (only the key identifier). For example:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

The first example selects all resources with key equal to `environment` and value equal to `production` or `qa`. The second example selects all resources with key equal to `tier` and values other than `frontend` and `backend`, and all resources with no labels with the `tier` key. The third example selects all resources including a label with key `partition`; no values are checked. The fourth example selects all resources without a label with key `partition`; no values are checked. Similarly the comma separator acts as an *AND* operator. So filtering resources with a `partition` key (no matter the value) and with `environment` different than `qa` can be achieved using `partition,environment notin (qa)`. The *set-based* label selector is a general form of equality since `environment=production` is equivalent to `environment in (production)`; similarly for `!=` and `notin`.

Set-based requirements can be mixed with *equality-based* requirements. For example:

```
partition in (customerA, customerB),environment!=qa .
```

基于集合的label条件允许用一组值来过滤键。支持三种操作符：`in`，`notin`，和 `exists`(仅针对key符号)。例如：

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

第一个例子，选择所有键等于 `environment`，且value等于 `production` 或者 `qa` 的资源。第二个例子，选择所有键等于 `tier` 且值是除了 `frontend` 和 `backend` 之外的资源，和那些没有 label的键是 `tier` 的资源。第三个例子，选择所有所有有一个label的键为 `partition` 的资源；值是什么不会被检查。第四个例子，选择所有的没有label的键名为 `partition` 的资源；值是什么不会被检查。

类似的，逗号操作符相当于一个**AND**操作符。因而要使用一个 `partition` 键（不管值是什么），并且 `environment` 不是 `qa` 过滤资源可以用 `partition,environment notin (qa)`。

基于集合的选择器是一个相等性的宽泛的形式，因为 `environment=production` 相当于 `environment in (production)`，与 `!= and notin` 类似。

基于集合的条件可以与基于相等性的条件混合。例如，`partition in (customerA, customerB), environment!=qa`。

API

LIST and WATCH filtering

LIST 和WATCH过滤

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter. Both requirements are permitted:

```
* _equality-based_ requirements: `?labelSelector=environment%3Dproduction,tier%3Dfrontend
* _set-based_ requirements: `?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in%
```

LIST和WATCH操作，可以使用query参数来指定label选择器来过滤返回对象的集合。两种条件都可以使用：基于相等性条件：`?labelSelector=environment%3Dproduction,tier%3Dfrontend`

基于集合条件的：`?`

```
labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29
```

Both label selector styles can be used to list or watch resources via a REST client. For example targetting `apiserver` with `kubectl` and using *equality-based* one may write:

两种label选择器风格都可以用来通过REST客户端来列表或者监视资源。比如使用 `kubectl` 来针对 `apiserver`，并且使用基于相等性的条件，可以用：

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

or using *set-based* requirements: 或者使用基于集合的条件：

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

As already mentioned *set-based* requirements are more expressive. For instance, they can implement the *OR* operator on values:

如以上已经提到的，基于集合的条件表达性更强。例如，他们可以实现值上的OR操作：

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

or restricting negative matching via `exists` operator:

或者通过`exists`操作符进行否定限制匹配：

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

Set references in API objects

Some Kubernetes objects, such as `service` and `replicationcontroller`, also use label selectors to specify sets of other resources, such as `pods`.

一些Kubernetes对象，比如`service`和`replication controller`的，也使用label选择器来指定其他资源的集合，比如`pods`。

Service and ReplicationController

The set of pods that a `service` targets is defined with a label selector. Similarly, the population of pods that a `replicationcontroller` should manage is also defined with a label selector.

Labels selectors for both objects are defined in `json` or `yaml` files using maps, and only *equality-based* requirement selectors are supported:

```
"selector": {
    "component" : "redis",
}
```

or

```
selector:
  component: redis
```

this selector (respectively in `json` or `yaml` format) is equivalent to `component=redis` or `component in (redis)`.

一个`service`针对的`pods`的集合是用label选择器来定义的。类似的，一个`replicationcontroller`管理的`pods`的群体也是用label选择器来定义的。

对于这两种对象的Label选择器是用map定义在json或者yaml文件中的，并且只支持基于相等性的条件：

```
"selector": {
    "component" : "redis",
}
```

或者

```
selector:
  component: redis
```

这个选择器（分别是位于json或者yaml格式的）相等于 `component=redis` 或者 `component in (redis)`。

Job and other new resources

Job和其他新的资源

Newer resources, such as [job](#), support *set-based* requirements as well.

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` is a map of `{key,value}` pairs. A single `{key,value}` in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose `key` field is "key", the `operator` is "In", and the `values` array contains only "value". `matchExpressions` is a list of pod selector requirements. Valid operators include In, NotIn, Exists, and DoesNotExist. The values set must be non-empty in the case of In and NotIn. All of the requirements, from both `matchLabels` and `matchExpressions` are ANDed together -- they must all be satisfied in order to match.

较新的资源，如job，也支持基于集合的条件。

```
selector:
  matchLabels:相当于一个
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` 是一个键值对的映射。一个单独的 `{key,value}` 相当于 `matchExpressions` 的一个元素，它的键字段是"key",操作符是 `In`，并且值数组值包含"value"。 `matchExpressions` 是一个pod的选择器条件的列表。合法的操作符包含`In`, `NotIn`, `Exists`, and `DoesNotExist`。在`In`和`NotIn`的情况下，值的组必须不能为空。所有的条件，包含 `matchLabels` and `matchExpressions` 中的，会用AND符号连接，他们必须都被满足以完成匹配。

Replication Controller

译者 : kz 校对 : 无

What is a *replication controller*?

A *replication controller* ensures that a specified number of pod "replicas" are running at any one time. In other words, a replication controller makes sure that a pod or homogenous set of pods are always up and available. If there are too many pods, it will kill some. If there are too few, the replication controller will start more. Unlike manually created pods, the pods maintained by a replication controller are automatically replaced if they fail, get deleted, or are terminated. For example, your pods get re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, we recommend that you use a replication controller even if your application requires only a single pod. You can think of a replication controller as something similar to a process supervisor, but rather than individual processes on a single node, the replication controller supervises multiple pods across multiple nodes.

As discussed in [life of a pod](#), `ReplicationController` is *only* appropriate for pods with `RestartPolicy = Always`. (Note: If `RestartPolicy` is not set, the default value is `Always`.) `ReplicationController` should refuse to instantiate any pod that has a different restart policy. As discussed in [issue #503](#), we expect other types of controllers to be added to Kubernetes to handle other types of workloads, such as build/test and batch workloads, in the future.

A replication controller will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be composed of pods controlled by multiple replication controllers, and it is expected that many replication controllers may be created and destroyed over the lifetime of a service (for instance, to perform an update of pods that run the service). Both services themselves and their clients should remain oblivious to the replication controllers that maintain the pods of the services.

For local container restarts, replication controllers delegate to an agent on the node, for example the [Kubelet](#) or Docker.

How does a replication controller work?

Pod template

A replication controller creates new pods from a template, which is currently inline in the `ReplicationController` object, but which we plan to extract into its own resource #170.

Rather than specifying the current desired state of all replicas, pod templates are like cookie cutters. Once a cookie has been cut, the cookie has no relationship to the cutter. There is no quantum entanglement. Subsequent changes to the template or even switching to a new template has no direct effect on the pods already created. Similarly, pods created by a replication controller may subsequently be updated directly. This is in deliberate contrast to pods, which do specify the current desired state of all containers belonging to the pod. This approach radically simplifies system semantics and increases the flexibility of the primitive, as demonstrated by the use cases explained below.

Pods created by a replication controller are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but replication controllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the [etcd lock module](#) or [RabbitMQ work queues](#), as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (e.g., cpu or memory), should be performed by another online controller process, not unlike the replication controller itself.

Labels

The population of pods that a replication controller is monitoring is defined with a [label selector](#), which creates a loosely coupled relationship between the controller and the pods controlled, in contrast to pods, which are more tightly coupled to their definition. We deliberately chose not to represent the set of pods controlled using a fixed-length array of pod specifications, because our experience is that approach increases complexity of management operations, for both clients and the system.

The replication controller should verify that the pods created from the specified template have labels that match its label selector. Though it isn't verified yet, you should also ensure that only one replication controller controls any given pod, by ensuring that the label selectors of replication controllers do not target overlapping sets. If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself with `--cascade=false` until there are no controllers with an overlapping superset of selectors.

Note that replication controllers may themselves have labels and would generally carry the labels their corresponding pods have in common, but these labels do not affect the behavior of the replication controllers.

Pods may be removed from a replication controller's target set by changing their labels. This technique may be used to remove pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Similarly, deleting a replication controller using the API does not affect the pods it created. Its `replicas` field must first be set to `0` in order to delete the pods controlled. (Note that the client tool, `kubectl`, provides a single operation, `delete` to delete both the replication controller and the pods it controls. If you want to leave the pods running when deleting a replication controller, specify `--cascade=false`. However, there is no such operation in the API at the moment)

Responsibilities of the replication controller

The replication controller simply ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, `readiness` and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The replication controller is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in [#492](#)), which would change its `replicas` field. We will not add scheduling policies (e.g., `spreading`) to the replication controller. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsewhere. We even plan to factor out the mechanism for bulk pod creation ([#170](#)).

The replication controller is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The "macro" operations currently supported by kubectl (run, stop, scale, rolling-update) are proof-of-concept examples of this. For instance, we could imagine something like [Asgard](#) managing replication controllers, auto-scalers, services, scheduling policies, canaries, etc.

Common usage patterns

Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, a replication controller will ensure that the specified number of pods exists, even in the event of node failure or pod termination (e.g., due to an action by another control agent).

Scaling

The replication controller makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the `replicas` field.

Rolling updates

The replication controller is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in [#1353](#), the recommended approach is to create a new replication controller with 1 replica, scale the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two replication controllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

Rolling update is implemented in the client tool [kubectl](#)

Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with `tier` in `(frontend)`, `environment` in `(prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a replication controller with `replicas` set to 9 for the bulk of the replicas, with labels `tier=frontend, environment=prod, track=stable`, and another replication controller with `replicas` set to 1 for the canary, with labels `tier=frontend, environment=prod, track=canary`. Now the service is covering both the canary and non-canary pods. But you can mess with the replication controllers separately to test things out, monitor the results, etc.

API Object

Replication controller is a top-level resource in the kubernetes REST API. More details about the API object can be found at: [ReplicationController API object](#).

Services in Kubernetes

译者 : kz 校对 : 无

Overview

Kubernetes `Pods` are mortal. They are born and they die, and they are not resurrected.

`ReplicationControllers` in particular create and destroy `Pods` dynamically (e.g. when scaling up or down or when doing `rolling updates`). While each `Pod` gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of `Pods` (let's call them backends) provides functionality to other `Pods` (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set?

Enter `Services`.

A Kubernetes `Service` is an abstraction which defines a logical set of `Pods` and a policy by which to access them - sometimes called a micro-service. The set of `Pods` targeted by a `Service` is (usually) determined by a `Label Selector` (see below for why you might want a `Service` without a selector).

As an example, consider an image-processing backend which is running with 3 replicas. Those replicas are fungible - frontends do not care which backend they use. While the actual `Pods` that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The `Service` abstraction enables this decoupling.

For Kubernetes-native applications, Kubernetes offers a simple `Endpoints` API that is updated whenever the set of `Pods` in a `Service` changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend `Pods`.

Defining a service

A `Service` in Kubernetes is a REST object, similar to a `Pod`. Like all of the REST objects, a `Service` definition can be POSTed to the apiserver to create a new instance. For example, suppose you have a set of `Pods` that each expose port 9376 and carry a label `"app=MyApp"`.

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
    "selector": {
      "app": "MyApp"
    },
    "ports": [
      {
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376
      }
    ]
  }
}
```

This specification will create a new `Service` object named "my-service" which targets TCP port 9376 on any `Pod` with the `"app=MyApp"` label. This `Service` will also be assigned an IP address (sometimes called the "cluster IP"), which is used by the service proxies (see below). The `Service`'s selector will be evaluated continuously and the results will be POSTed to an `Endpoints` object also named "my-service".

Note that a `Service` can map an incoming port to any `targetPort`. By default the `targetPort` will be set to the same value as the `port` field. Perhaps more interesting is that `targetPort` can be a string, referring to the name of a port in the backend `Pods`. The actual port number assigned to that name can be different in each backend `Pod`. This offers a lot of flexibility for deploying and evolving your `Services`. For example, you can change the port number that pods expose in the next version of your backend software, without breaking clients.

Kubernetes `Services` support `TCP` and `UDP` for protocols. The default is `TCP`.

Services without selectors

Services generally abstract access to Kubernetes `Pods`, but they can also abstract other kinds of backends. For example:

- You want to have an external database cluster in production, but in test you use your own databases.
- You want to point your service to a service in another `Namespace` or on another cluster.
- You are migrating your workload to Kubernetes and some of your backends run outside

of Kubernetes.

In any of these scenarios you can define a service without a selector:

```
{  
    "kind": "Service",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "my-service"  
    },  
    "spec": {  
        "ports": [  
            {  
                "protocol": "TCP",  
                "port": 80,  
                "targetPort": 9376  
            }  
        ]  
    }  
}
```

Because this service has no selector, the corresponding `Endpoints` object will not be created. You can manually map the service to your own specific endpoints:

```
{  
    "kind": "Endpoints",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "my-service"  
    },  
    "subsets": [  
        {  
            "addresses": [  
                { "IP": "1.2.3.4" }  
            ],  
            "ports": [  
                { "port": 9376 }  
            ]  
        }  
    ]  
}
```

NOTE: Endpoint IPs may not be loopback (127.0.0.0/8), link-local (169.254.0.0/16), or link-local multicast ((224.0.0.0/24)).

Accessing a `Service` without a selector works the same as if it had selector. The traffic will be routed to endpoints defined by the user (`1.2.3.4:9376` in this example).

Virtual IPs and service proxies

Every node in a Kubernetes cluster runs a `kube-proxy`. This application watches the Kubernetes master for the addition and removal of `Service` and `Endpoints` objects. For each `Service` it opens a port (randomly chosen) on the local node. Any connections to `Service` port will be proxied to one of the corresponding backend `Pods`. Which backend `Pod` to use is decided based on the `SessionAffinity` of the `Service`. Lastly, it installs iptables rules which capture traffic to the `Service`'s cluster IP (which is virtual) and `Port` then redirects that traffic to the backend `Pod` (`Endpoints`).

The net result is that any traffic bound for the `Service` is proxied to an appropriate backend without the clients knowing anything about Kubernetes or `Services` or `Pods`.

By default, the choice of backend is round robin. Client-IP based session affinity can be selected by setting `service.spec.sessionAffinity` to `"ClientIP"` (the default is `"None"`).

As of Kubernetes 1.0, `Services` are a "layer 3" (TCP/UDP over IP) construct. We do not yet have a concept of "layer 7" (HTTP) services.

Multi-Port Services

Many `Services` need to expose more than one port. For this case, Kubernetes supports multiple port definitions on a `Service` object. When using multiple ports you must give all of your ports names, so that endpoints can be disambiguated. For example:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
    "selector": {
      "app": "MyApp"
    },
    "ports": [
      {
        "name": "http",
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376
      },
      {
        "name": "https",
        "protocol": "TCP",
        "port": 443,
        "targetPort": 9377
      }
    ]
  }
}
```

Choosing your own IP address

You can specify your own cluster IP address as part of a `Service` creation request. To do this, set the `spec.clusterIP` field. For example, if you already have an existing DNS entry that you wish to replace, or legacy systems that are configured for a specific IP address and difficult to re-configure. The IP address that a user chooses must be a valid IP address and within the `service-cluster-ip-range` CIDR range that is specified by flag to the API server. If the IP address value is invalid, the apiserver returns a 422 HTTP status code to indicate that the value is invalid.

Why not use round-robin DNS?

A question that pops up every now and then is why we do all this stuff with virtual IPs rather than just use standard round-robin DNS. There are a few reasons:

- There is a long history of DNS libraries not respecting DNS TTLs and caching the results of name lookups.
- Many apps do DNS lookups once and cache the results.

- Even if apps and libraries did proper re-resolution, the load of every client re-resolving DNS over and over would be difficult to manage.

We try to discourage users from doing things that hurt themselves. That said, if enough people ask for this, we may implement it as an alternative.

Discovering services

Kubernetes supports 2 primary modes of finding a `Service` - environment variables and DNS.

Environment variables

When a `Pod` is run on a `Node`, the kubelet adds a set of environment variables for each active `Service`. It supports both [Docker links compatible](#) variables (see [makeLinkVariables](#)) and simpler `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

For example, the Service `"redis-master"` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11 produces the following environment variables:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

This does imply an ordering requirement - any `Service` that a `Pod` wants to access must be created before the `Pod` itself, or else the environment variables will not be populated. DNS does not have this restriction.

DNS

An optional (though strongly recommended) [cluster add-on](#) is a DNS server. The DNS server watches the Kubernetes API for new `Services` and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all `Pods` should be able to do name resolution of `Services` automatically.

For example, if you have a `Service` called `"my-service"` in Kubernetes Namespace `"my-ns"` a DNS record for `"my-service.my-ns"` is created. `Pods` which exist in the `"my-ns"` Namespace should be able to find it by simply doing a name lookup for `"my-service"`. `Pods` which exist in other Namespaces must qualify the name as `"my-service.my-ns"`. The result of these name lookups is the cluster IP.

Kubernetes also supports DNS SRV (service) records for named ports. If the `"my-service.my-ns"` Service has a port named `"http"` with protocol `TCP`, you can do a DNS SRV query for `"_http._tcp.my-service.my-ns"` to discover the port number for `"http"`.

Headless services

Sometimes you don't need or want load-balancing and a single service IP. In this case, you can create "headless" services by specifying `"None"` for the cluster IP (`spec.clusterIP`).

For such `Services`, a cluster IP is not allocated. DNS is configured to return multiple A records (addresses) for the `Service` name, which point directly to the `Pods` backing the `Service`. Additionally, the kube proxy does not handle these services and there is no load balancing or proxying done by the platform for them. The endpoints controller will still create `Endpoints` records in the API.

This option allows developers to reduce coupling to the Kubernetes system, if they desire, but leaves them freedom to do discovery in their own way. Applications can still use a self-registration pattern and adapters for other discovery systems could easily be built upon this API.

Publishing services - service types

For some parts of your application (e.g. frontends) you may want to expose a Service onto an external (outside of your cluster, maybe public internet) IP address, other services should be visible only from inside of the cluster.

Kubernetes `ServiceTypes` allow you to specify what kind of service you want. The default and base type is `ClusterIP`, which exposes a service to connection from inside the cluster. `NodePort` and `LoadBalancer` are two types that expose services to external traffic.

Valid values for the `ServiceType` field are:

- `ClusterIP` : use a cluster-internal IP only - this is the default and is discussed above. Choosing this value means that you want this service to be reachable only from inside of the cluster.
- `NodePort` : on top of having a cluster-internal IP, expose the service on a port on each

node of the cluster (the same port on each node). You'll be able to contact the service on any `<NodeIP>:NodePort` address.

- `LoadBalancer` : on top of having a cluster-internal IP and exposing service on a `NodePort` also, ask the cloud provider for a load balancer which forwards to the `Service` exposed as a `<NodeIP>:NodePort` for each Node.

Note that while `NodePort`'s can be TCP or UDP, `LoadBalancer`'s only support TCP as of Kubernetes 1.0.

Type NodePort

If you set the `type` field to `"NodePort"`, the Kubernetes master will allocate a port from a flag-configured range (default: 30000-32767), and each Node will proxy that port (the same port number on every Node) into your `Service`. That port will be reported in your `Service`'s `spec.ports[*].nodePort` field.

If you want a specific port number, you can specify a value in the `nodePort` field, and the system will allocate you that port or else the API transaction will fail (i.e. you need to take care about possible port collisions yourself). The value you specify must be in the configured range for node ports.

This gives developers the freedom to set up their own load balancers, to configure cloud environments that are not fully supported by Kubernetes, or even to just expose one or more nodes' IPs directly.

Note that this Service will be visible as both `<NodeIP>:spec.ports[*].nodePort` and `spec.clusterIp:spec.ports[*].port`.

Type LoadBalancer

On cloud providers which support external load balancers, setting the `type` field to `"LoadBalancer"` will provision a load balancer for your `Service`. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer will be published in the `Service`'s `status.loadBalancer` field. For example:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
    "selector": {
      "app": "MyApp"
    },
    "ports": [
      {
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376,
        "nodePort": 30061
      }
    ],
    "clusterIP": "10.0.171.239",
    "loadBalancerIP": "78.11.24.19",
    "type": "LoadBalancer"
  },
  "status": {
    "loadBalancer": {
      "ingress": [
        {
          "ip": "146.148.47.155"
        }
      ]
    }
  }
}
```

Traffic from the external load balancer will be directed at the backend `Pods`, though exactly how that works depends on the cloud provider. Some cloud providers allow the `loadBalancerIP` to be specified. In those cases, the load-balancer will be created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, an ephemeral IP will be assigned to the `loadBalancer`. If the `loadBalancerIP` is specified, but the cloud provider does not support the feature, the field will be ignored.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes services can be exposed on those `externalIPs`. Traffic that ingresses into the cluster with the external IP (as destination IP), on the service port, will be routed to one of the service endpoints. `externalIPs` are not managed by Kubernetes and are the responsibility of the cluster administrator.

In the ServiceSpec, `externalIPs` can be specified along with any of the `ServiceTypes`. In the example below, my-service can be accessed by clients on 80.11.12.10:80 (externalIP:port)

```
{  
    "kind": "Service",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "my-service"  
    },  
    "spec": {  
        "selector": {  
            "app": "MyApp"  
        },  
        "ports": [  
            {  
                "name": "http",  
                "protocol": "TCP",  
                "port": 80,  
                "targetPort": 9376  
            }  
        ],  
        "externalIPs" : [  
            "80.11.12.10"  
        ]  
    }  
}
```

Shortcomings

We expect that using iptables and userspace proxies for VIPs will work at small to medium scale, but may not scale to very large clusters with thousands of Services. See [the original design proposal for portals](#) for more details.

Using the kube-proxy obscures the source-IP of a packet accessing a `Service`. This makes some kinds of firewalling impossible.

LoadBalancers only support TCP, not UDP.

The `Type` field is designed as nested functionality - each level adds to the previous. This is not strictly required on all cloud providers (e.g. Google Compute Engine does not need to allocate a `NodePort` to make `LoadBalancer` work, but AWS does) but the current API requires it.

Future work

In the future we envision that the proxy policy can become more nuanced than simple round robin balancing, for example master-elected or sharded. We also envision that some services will have "real" load balancers, in which case the VIP will simply transport the packets there.

There's a [proposal](#) to eliminate userspace proxying in favor of doing it all in iptables. This should perform better and fix the source-IP obfuscation, though is less flexible than arbitrary userspace code.

We intend to have first-class support for L7 (HTTP) services .

We intend to have more flexible ingress modes for services which encompass the current ClusterIP , NodePort , and LoadBalancer modes and more.

The gory details of virtual IPs

The previous information should be sufficient for many people who just want to use services . However, there is a lot going on behind the scenes that may be worth understanding.

Avoiding collisions

One of the primary philosophies of Kubernetes is that users should not be exposed to situations that could cause their actions to fail through no fault of their own. In this situation, we are looking at network ports - users should not have to choose a port number if that choice might collide with another user. That is an isolation failure.

In order to allow users to choose a port number for their services , we must ensure that no two services can collide. We do that by allocating each service its own IP address.

To ensure each service receives a unique IP, an internal allocator atomically updates a global allocation map in etcd prior to each service. The map object must exist in the registry for services to get IPs, otherwise creations will fail with a message indicating an IP could not be allocated. A background controller is responsible for creating that map (to migrate from older versions of Kubernetes that used in memory locking) as well as checking for invalid assignments due to administrator intervention and cleaning up any IPs that were allocated but which no service currently uses.

IPs and VIPs

Unlike `Pod` IP addresses, which actually route to a fixed destination, `Service` IPs are not actually answered by a single host. Instead, we use `iptables` (packet processing logic in Linux) to define virtual IP addresses which are transparently redirected as needed. When clients connect to the VIP, their traffic is automatically transported to an appropriate endpoint. The environment variables and DNS for `Services` are actually populated in terms of the `Service`'s VIP and port.

As an example, consider the image processing application described above. When the backend `Service` is created, the Kubernetes master assigns a virtual IP address, for example 10.0.0.1. Assuming the `Service` port is 1234, the `Service` is observed by all of the `kube-proxy` instances in the cluster. When a proxy sees a new `Service`, it opens a new random port, establishes an `iptables` redirect from the VIP to this new port, and starts accepting connections on it.

When a client connects to the VIP the `iptables` rule kicks in, and redirects the packets to the `Service proxy`'s own port. The `Service proxy` chooses a backend, and starts proxying traffic from the client to the backend.

This means that `Service` owners can choose any port they want without risk of collision. Clients can simply connect to an IP and port, without being aware of which `Pods` they are actually accessing.

API Object

Service is a top-level resource in the kubernetes REST API. More details about the API object can be found at: [Service API object](#).

Volumes

译者 : kz 校对 : 无

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. First, when a container crashes kubelet will restart it, but the files will be lost - the container starts with a clean slate. Second, when running containers together in a `Pod` it is often necessary to share files between those containers. The Kubernetes `Volume` abstraction solves both of these problems.

Familiarity with `pods` is suggested.

Background

Docker also has a concept of `volumes`, though it is somewhat looser and less managed. In Docker, a volume is simply a directory on disk or in another container. Lifetimes are not managed and until very recently there were only local-disk-backed volumes. Docker now provides volume drivers, but the functionality is very limited for now (e.g. as of Docker 1.7 only one volume driver is allowed per container and there is no way to pass parameters to volumes).

A Kubernetes volume, on the other hand, has an explicit lifetime - the same as the pod that encloses it. Consequently, a volume outlives any containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many type of volumes, and a Pod can use any number of them simultaneously.

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, a pod specifies what volumes to provide for the pod (the `spec.volumes` field) and where to mount those into containers(the `spec.containers.volumeMounts` field).

A process in a container sees a filesystem view composed from their Docker image and volumes. The `Docker image` is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image. Volumes can not mount onto other volumes or have hard links to other volumes. Each container in the Pod must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

- `emptyDir`
- `hostPath`
- `gcePersistentDisk`
- `awsElasticBlockStore`
- `nfs`
- `iscsi`
- `flocker`
- `glusterfs`
- `rbd`
- `gitRepo`
- `secret`
- `persistentVolumeClaim`

We welcome additional contributions.

emptyDir

An `emptyDir` volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node. As the name says, it is initially empty. Containers in the pod can all read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted forever. NOTE: a container crashing does *NOT* remove a pod from a node, so the data in an `emptyDir` volume is safe across container crashes.

Some uses for an `emptyDir` are:

- scratch space, such as for a disk-based mergesortcw
- checkpointing a long computation for recovery from crashes
- holding files that a content-manager container fetches while a webserver container serves the data

By default, `emptyDir` volumes are stored on whatever medium is backing the machine - that might be disk or SSD or network storage, depending on your environment. However, you can set the `emptyDir.medium` field to `"Memory"` to tell Kubernetes to mount a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that unlike disks, tmpfs is cleared on machine reboot and any files you write will count against your container's memory limit.

hostPath

A `hostPath` volume mounts a file or directory from the host node's filesystem into your pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

For example, some uses for a `hostPath` are:

- running a container that needs access to Docker internals; use a `hostPath` of `/var/lib/docker`
- running cAdvisor in a container; use a `hostPath` of `/dev/cgroups`

Watch out when using this type of volume, because:

- pods with identical configuration (such as created from a `podTemplate`) may behave differently on different nodes due to different files on the nodes
- when Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a `hostPath`

gcePersistentDisk

A `gcePersistentDisk` volume mounts a Google Compute Engine (GCE) [Persistent Disk](#) into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a PD are preserved and the volume is merely unmounted. This means that a PD can be pre-populated with data, and that data can be "handed off" between pods.

Important: You must create a PD using `gcloud` or the GCE API or UI before you can use it

There are some restrictions when using a `gcePersistentDisk`:

- the nodes on which pods are running must be GCE VMs
- those VMs need to be in the same GCE project and zone as the PD

A feature of PD is that they can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a PD with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, PDs can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

Using a PD on a pod controlled by a ReplicationController will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a PD

Before you can use a GCE PD with a pod, you need to create it.

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

Example pod

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: gcr.io/google_containers/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  volumes:
    - name: test-volume
      # This GCE PD must already exist.
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

awsElasticBlockStore

An `awsElasticBlockStore` volume mounts an Amazon Web Services (AWS) [EBS Volume](#) into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between pods.

Important: You must create an EBS volume using `aws ec2 create-volume` or the AWS API before you can use it

There are some restrictions when using an `awsElasticBlockStore` volume:

- the nodes on which pods are running must be AWS EC2 instances
- those instances need to be in the same region and availability-zone as the EBS volume
- EBS only supports a single EC2 instance mounting a volume

Creating an EBS volume

Before you can use a EBS volume with a pod, you need to create it.

```
aws ec2 create-volume --availability-zone eu-west-1a --size 10 --volume-type gp2
```

Make sure the zone matches the zone you brought up your cluster in. (And also check that the size and EBS volume type are suitable for your use!)

AWS EBS Example configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
    - image: gcr.io/google_containers/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-ebs
          name: test-volume
  volumes:
    - name: test-volume
      # This AWS EBS volume must already exist.
      awsElasticBlockStore:
        volumeID: aws:///
        fsType: ext4
```

(Note: the syntax of volumeID is currently awkward; #10181 fixes it)

nfs

An `nfs` volume allows an existing NFS (Network File System) share to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `nfs` volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data, and that data can be "handed off" between pods. NFS can be mounted by multiple writers simultaneously.

Important: You must have your own NFS server running with the share exported before you can use it

See the [NFS example](#) for more details.

For example, [this file](#) demonstrates how to specify the usage of an NFS volume within a pod.

In this example one can see that a `volumeMount` called `nfs` is being mounted onto `/usr/share/nginx/html` in the container `web`. The volume "nfs" is defined as type `nfs`, with the NFS server serving from `nfs-server.default.kube.local` and exporting directory `/` as the share. The mount being created in this example is writeable.

iscsi

An `iscsi` volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `iscsi` volume are preserved and the volume is merely unmounted. This means that an `iscsi` volume can be pre-populated with data, and that data can be "handed off" between pods.

Important: You must have your own iSCSI server running with the volume created before you can use it

A feature of iSCSI is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

See the [iSCSI example](#) for more details.

flocker

[Flocker](#) is an open-source clustered container data volume manager. It provides management and orchestration of data volumes backed by a variety of storage backends.

A `flocker` volume allows a Flocker dataset to be mounted into a pod. If the dataset does not already exist in Flocker, it needs to be created with Flocker CLI or the using the Flocker API. If the dataset already exists it will reattached by Flocker to the node that the pod is scheduled. This means data can be "handed off" between pods as required.

Important: You must have your own Flocker installation running before you can use it

See the [Flocker example](#) for more details.

glusterfs

A `glusterfs` volume allows a [Glusterfs](#) (an open source networked filesystem) volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `glusterfs` volume are preserved and the volume is merely unmounted. This means that a `glusterfs` volume can be pre-populated with data, and that data can be "handed off" between pods. GlusterFS can be mounted by multiple writers simultaneously.

Important: You must have your own GlusterFS installation running before you can use it

See the [GlusterFS example](#) for more details.

rbd

An `rbd` volume allows a [Rados Block Device](#) volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `rbd` volume are preserved and the volume is merely unmounted. This means that a RBD volume can be pre-populated with data, and that data can be "handed off" between pods.

Important: You must have your own Ceph installation running before you can use RBD

A feature of RBD is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, RBD volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

See the [RBD example](#) for more details.

gitRepo

A `gitRepo` volume is an example of what can be done as a volume plugin. It mounts an empty directory and clones a git repository into it for your pod to use. In the future, such volumes may be moved to an even more decoupled model, rather than extending the Kubernetes API for every such use case.

Here is a example for gitRepo volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - mountPath: /mypath
          name: git-volume
  volumes:
    - name: git-volume
      gitRepo:
        repository: "git@somewhere:me/my-git-repository.git"
        revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

secret

A `secret` volume is used to pass sensitive information, such as passwords, to pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly. `secret` volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

Important: You must create a secret in the Kubernetes API before you can use it

Secrets are described in more detail [here](#).

persistentVolumeClaim

A `persistentVolumeClaim` volume is used to mount a [PersistentVolume](#) into a pod. PersistentVolumes are a way for users to "claim" durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

See the [PersistentVolumes example](#) for more details.

downwardAPI

A `downwardAPI` volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

See the [downwardAPI volume example](#) for more details.

Resources

The storage media (Disk, SSD, etc) of an `emptyDir` volume is determined by the medium of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an `emptyDir` or `hostPath` volume can consume, and no isolation between containers or between pods.

In the future, we expect that `emptyDir` and `hostPath` volumes will be able to request a certain amount of space using a [resource](#) specification, and to select the type of media to use, for clusters that have several media types.

Secrets

译者 : kz 校对 : 无

Objects of type `secret` are intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a `secret` is safer and more flexible than putting it verbatim in a `pod` definition or in a docker image. See [Secrets design document](#) for more information.

Overview of Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: either as files in a `volume` mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

Service Accounts Automatically Create and Attach Secrets with API Credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and it automatically modifies your pods to use this type of secret.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the apiserver, this is the recommended workflow.

See the [Service Account](#) documentation for more information on how Service Accounts work.

Creating a Secret Manually

This is an example of a simple secret, in yaml format:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: dmFsdWUtMg0K
  username: dmFsdWUtMQ0K
```

The data field is a map. Its keys must match `DNS_SUBDOMAIN`, except that leading dots are also allowed. The values are arbitrary data, encoded using base64. The values of username and password in the example above, before base64 encoding, are `value-1` and `value-2`, respectively, with carriage return and newline characters at the end.

Create the secret using `kubectl create`.

Once the secret is created, you can need to modify your pod to specify that it should use the secret.

Manually specifying a Secret to be Mounted on a Pod

This is an example of a pod that mounts a secret in a volume:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "mypod",
    "namespace": "myns"
  },
  "spec": {
    "containers": [
      {
        "name": "mypod",
        "image": "redis",
        "volumeMounts": [
          {
            "name": "foo",
            "mountPath": "/etc/foo",
            "readOnly": true
          }
        ]
      }
    ],
    "volumes": [
      {
        "name": "foo",
        "secret": {
          "secretName": "mysecret"
        }
      }
    ]
  }
}
```

Each secret you want to use needs its own `spec.volumes`.

If there are multiple containers in the pod, then each container needs its own `volumeMounts` block, but only one `spec.volumes` is needed per secret.

You can package many files into one secret, or use many secrets, whichever is convenient.

See another example of creating a secret and a pod that consumes that secret in a volume [here](#).

Manually specifying an `imagePullSecret`

Use of `imagePullSecrets` is described in the [images documentation](#)

Arranging for `imagePullSecrets` to be Automatically Attached

You can manually create an `imagePullSecret`, and reference it from a `serviceAccount`. Any pods created with that `serviceAccount` or that default to use that `serviceAccount`, will get their `imagePullSecret` field set to that of the service account. See [here](#) for a detailed explanation of that process.

Automatic Mounting of Manually Created Secrets

We plan to extend the service account behavior so that manually created secrets (e.g. one containing a token for accessing a github account) can be automatically attached to pods based on their service account. *This is not implemented yet.* See [issue 9902](#).

Details

Restrictions

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type `secret`. Therefore, a secret needs to be created before any pods that depend on it.

Secret API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage creation of very large secrets which would exhaust apiserver and kubelet memory. However, creation of many smaller secrets could also exhaust memory. More comprehensive limits on memory usage

due to secrets is a planned feature.

Kubelet only supports use of secrets for Pods it gets from the API server. This includes any pods created using kubectl, or indirectly via a replication controller. It does not include pods created via the kubelets `--manifest-url` flag, its `--config` flag, or its REST API (these are not common ways to create pods.)

Consuming Secret Values

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
$ ls /etc/foo/  
username  
password  
$ cat /etc/foo/username  
value-1  
$ cat /etc/foo/password  
value-2
```

The program in a container is responsible for reading the secret(s) from the files. Currently, if a program expects a secret to be stored in an environment variable, then the user needs to modify the image to populate the environment variable from the file as an step before running the main program. Future versions of Kubernetes are expected to provide more automation for populating environment variables from files.

Secret and Pod Lifetime interaction

When a pod is created via the API, there is no check whether a referenced secret exists. Once a pod is scheduled, the kubelet will try to fetch the secret value. If the secret cannot be fetched because it does not exist or because of a temporary lack of connection to the API server, kubelet will periodically retry. It will report an event about the pod explaining the reason it is not started yet. Once the a secret is fetched, the kubelet will create and mount a volume containing it. None of the pod's containers will start until all the pod's volumes are mounted.

Once the kubelet has started a pod's containers, its secret volumes will not change, even if the secret resource is modified. To change the secret used, the original pod must be deleted, and a new pod (perhaps with an identical `PodSpec`) must be created. Therefore, updating a secret follows the same workflow as deploying a new container image. The `kubectl rolling-update` command can be used ([man page](#)).

The `resourceVersion` of the secret is not specified when it is referenced. Therefore, if a secret is updated at about the same time as pods are starting, then it is not defined which version of the secret will be used for the pod. It is not possible currently to check what resource version of a secret object was used when a pod was created. It is planned that pods will report this information, so that a replication controller restarts ones using an old `resourceVersion`. In the interim, if this is a concern, it is recommended to not update the data of existing secrets, but to create new ones with distinct names.

Use cases

Use-Case: Pod with ssh keys

To create a pod that uses an ssh key stored as a secret, we first need to create a secret:

```
{  
  "kind": "Secret",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "ssh-key-secret"  
  },  
  "data": {  
    "id-rsa": "dmFsdWUtMg0KDQo=",  
    "id-rsa.pub": "dmFsdWUtMQ0K"  
  }  
}
```

Note: The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted.

Now we can create a pod which references the secret with the ssh key and consumes it in a volume:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "secret-test-pod",
    "labels": {
      "name": "secret-test"
    }
  },
  "spec": {
    "volumes": [
      {
        "name": "secret-volume",
        "secret": {
          "secretName": "ssh-key-secret"
        }
      }
    ],
    "containers": [
      {
        "name": "ssh-test-container",
        "image": "mySshImage",
        "volumeMounts": [
          {
            "name": "secret-volume",
            "readOnly": true,
            "mountPath": "/etc/secret-volume"
          }
        ]
      }
    ]
  }
}
```

When the container's command runs, the pieces of the key will be available in:

```
/etc/secret-volume/id-rsa.pub
/etc/secret-volume/id-rsa
```

The container is then free to use the secret data to establish an ssh connection.

Use-Case: Pods with prod / test credentials

This example illustrates a pod which consumes a secret containing prod credentials and another pod which consumes a secret with test environment credentials.

The secrets:

```
{  
    "apiVersion": "v1",  
    "kind": "List",  
    "items":  
    [ {  
        "kind": "Secret",  
        "apiVersion": "v1",  
        "metadata": {  
            "name": "prod-db-secret"  
        },  
        "data": {  
            "password": "dmFsdWUtMg0KDQo=",  
            "username": "dmFsdWUtMQ0K"  
        }  
    },  
    {  
        "kind": "Secret",  
        "apiVersion": "v1",  
        "metadata": {  
            "name": "test-db-secret"  
        },  
        "data": {  
            "password": "dmFsdWUtMg0KDQo=",  
            "username": "dmFsdWUtMQ0K"  
        }  
    }]  
}
```

The pods:

```
{  
    "apiVersion": "v1",  
    "kind": "List",  
    "items":  
    [ {  
        "kind": "Pod",  
        "apiVersion": "v1",  
        "metadata": {  
            "name": "prod-db-client-pod",  
            "labels": {  
                "name": "prod-db-client"  
            }  
        },  
        "spec": {  
            "volumes": [  
                {  
                    "name": "secret-volume",  
                    "secret": {  
                        "secretName": "prod-db-secret"  
                    }  
                }  
            ]  
        }  
    }]
```

```
],
  "containers": [
    {
      "name": "db-client-container",
      "image": "myClientImage",
      "volumeMounts": [
        {
          "name": "secret-volume",
          "readOnly": true,
          "mountPath": "/etc/secret-volume"
        }
      ]
    }
  ],
  {
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata": {
      "name": "test-db-client-pod",
      "labels": {
        "name": "test-db-client"
      }
    },
    "spec": {
      "volumes": [
        {
          "name": "secret-volume",
          "secret": {
            "secretName": "test-db-secret"
          }
        }
      ],
      "containers": [
        {
          "name": "db-client-container",
          "image": "myClientImage",
          "volumeMounts": [
            {
              "name": "secret-volume",
              "readOnly": true,
              "mountPath": "/etc/secret-volume"
            }
          ]
        }
      ]
    }
  }
]
```

Both containers will have the following files present on their filesystems:

```
/etc/secret-volume/username
/etc/secret-volume/password
```

Note how the specs for the two pods differ only in one field; this facilitates creating pods with different capabilities from a common pod config template.

You could further simplify the base pod specification by using two Service Accounts: one called, say, `prod-user` with the `prod-db-secret`, and one called, say, `test-user` with the `test-db-secret`. Then, the pod spec can be shortened to, for example:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "prod-db-client-pod",
    "labels": {
      "name": "prod-db-client"
    }
  },
  "spec": {
    "serviceAccount": "prod-db-client",
    "containers": [
      {
        "name": "db-client-container",
        "image": "myClientImage",
      }
    ]
  }
}
```

Use-case: Secret visible to one container in a pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (e.g. over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Security Properties

Protections

Because `secret` objects can be created independently of the `pods` that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing pods. The system can also take additional precautions with `secret` objects, such as avoiding writing them to disk where possible.

A secret is only sent to a node if a pod on that node requires it. It is not written to disk. It is stored in a tmpfs. It is deleted once the pod that depends on it is deleted.

On most Kubernetes-project-maintained distributions, communication between user to the apiserver, and from apiserver to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.

Secret data on nodes is stored in tmpfs volumes and thus does not come to rest on the node.

There may be secrets for several pods on the same node. However, only the secrets that a pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the secrets of another pod.

There may be several containers in a pod. However, each container in a pod has to request the secret volume in its `volumeMounts` for it to be visible within the container. This can be used to construct useful [security partitions at the Pod level](#).

Risks

- In the API server secret data is stored as plaintext in etcd; therefore:
 - Administrators should limit access to etcd to admin users
 - Secret data in the API server is at rest on the disk that etcd uses; admins may want to wipe/shred disks used by etcd when no longer in use
- Applications still need to protect the value of secret after reading it from the volume, such as not accidentally logging it or transmitting it to an untrusted party.
- A user who can create a pod that uses a secret can also see the value of that secret. Even if apiserver policy does not allow that user to read the secret object, the user could run a pod which exposes the secret.
- If multiple replicas of etcd are run, then the secrets will be shared between them. By default, etcd does not secure peer-to-peer communication with SSL/TLS, though this can be configured.
- It is not possible currently to control which users of a Kubernetes cluster can access a

secret. Support for this is planned.

- Currently, anyone with root on any node can read any secret from the apiserver, by impersonating the kubelet. It is a planned feature to only send secrets to nodes that actually require them, to restrict the impact of a root exploit on a single node.

Identifiers

译者 : kz 校对 : 无

All objects in the Kubernetes REST API are unambiguously identified by a Name and a UID.

For non-unique user-provided attributes, Kubernetes provides [labels](#) and [annotations](#).

Names

Names are generally client-provided. Only one object of a given kind can have a given name at a time (i.e., they are spatially unique). But if you delete an object, you can make a new object with the same name. Names are used to refer to an object in a resource URL, such as `/api/v1/pods/some-name`. By convention, the names of Kubernetes resources should be up to maximum length of 253 characters and consist of lower case alphanumeric characters, `-`, and `.`, but certain resources have more specific restrictions. See the [identifiers design doc](#) for the precise syntax rules for names.

UIDs

UIDs are generated by Kubernetes. Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID (i.e., they are spatially and temporally unique).

Namespaces

译者 : kz 校对 : 无

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.

Namespaces are a way to divide cluster resources between multiple uses (via [resource quota](#)).

In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use [labels](#) to distinguish resources within the same namespace.

Working with Namespaces

Creation and deletion of namespaces is described in the [Admin Guide documentation for namespaces](#)

Viewing namespaces

You can list the current namespaces in a cluster using:

```
$ kubectl get namespaces
NAME      LABELS      STATUS
default          Active
kube-system      Active
```

Kubernetes starts with two initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system

Setting the namespace for a request

To temporarily set the namespace for a request, use the `--namespace` flag.

For example:

```
$ kubectl --namespace= run nginx --image=nginx  
$ kubectl --namespace= get pods
```

Setting the namespace preference

You can permanently save the namespace for all subsequent kubectl commands in that context.

First get your current context:

```
$ export CONTEXT=$(kubectl config view | grep current-context | awk '{print $2}')
```

Then update the default namespace:

```
$ kubectl config set-context $(CONTEXT) --namespace=
```

Namespaces and DNS

When you create a [Service](#), it creates a corresponding [DNS entry](#). This entry is of the form `<service-name>. <namespace-name>. svc.cluster.local`, which means that if a container just uses `<service-name>` it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

Not All Objects are in a Namespace

Most kubernetes resources (e.g. pods, services, replication controllers, and others) are in a some namespace. However namespace resources are not themselves in a namespace. And, low-level resources, such as [nodes](#) and [persistentVolumes](#), are not in any namespace. Events are an exception: they may or may not have a namespace, depending on the object the event is about.

Annotations

译者 : kz 校对 : 无

We have [labels](#) for identifying metadata.

It is also useful to be able to attach arbitrary non-identifying metadata, for retrieval by API clients such as tools, libraries, etc. This information may be large, may be structured or unstructured, may include characters not permitted by labels, etc. Such information would not be used for object selection and therefore doesn't belong in labels.

Like labels, annotations are key-value maps.

```
"annotations": {  
    "key1" : "value1",  
    "key2" : "value2"  
}
```

Possible information that could be recorded in annotations:

- fields managed by a declarative configuration layer, to distinguish them from client-and/or server-set default values and other auto-generated fields, fields set by auto-sizing/auto-scaling systems, etc., in order to facilitate merging
- build/release/image information (timestamps, release ids, git branch, PR numbers, image hashes, registry address, etc.)
- pointers to logging/monitoring/analytics/audit repos
- client library/tool information (e.g. for debugging purposes -- name, version, build info)
- other user and/or tool/system provenance info, such as URLs of related objects from other ecosystem components
- lightweight rollout tool metadata (config and/or checkpoints)
- phone/pager number(s) of person(s) responsible, or directory entry where that info could be found, such as a team website

Yes, this information could be stored in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, etc.

基础教程

译者：White 校对：无

请参看后面的『Kubernetes 101』和『Kubernetes 102』部分。

- [Kubernetes 101](#)
- [Kubernetes 201](#)

Kubernetes 101 - Kubectl CLI和Pods

译者：张迪 校对：无

Kubernetes 101 - Kubectl CLI and Pods

对于Kubernetes 101，我们将覆盖的内容包括kubectl, pods, volumes, 以及多容器。

为了kubectl使用率示例可以正常运行，请确保你已经在本地有示例目录，可以从<https://github.com/kubernetes/kubernetes/releases>中或者从<https://github.com/kubernetes/kubernetes>中得到的。

内容列表

- Kubernetes 101 - Kubectl CLI 和 Pods
 - Kubectl CLI
 - Pods
 - Pod 定义
 - Pod 管理
 - Volumes
 - Volume 类型
 - 多容器
 - 下面是什么？

Kubectl CLI

同Kubernetes交互的最简单方法是通过kubectl命令行界面

对于kubectl的更多信息，包括它的使用率，命令，参数，请阅读kubectl CLI参考手册<http://kubernetes.io/v1.1/docs/user-guide/kubectl/kubectl.html>。

如果你没有安装好或者配置好kubectl，在继续阅读之前按照<http://kubernetes.io/v1.1/docs/user-guide/prereqs.html>把预置条件完成。

Pods

在Kubernetes，一组单个或者多个容器叫做pod。Pod中的容器是部署在一起的，并且作为一组来启动，停止和复制。

点击pods链接<http://kubernetes.io/v1.1/docs/user-guide/pods.html>得到更多细节信息。

Pod定义

最简单的pod定义描述了单个容器的部署。比如，一个nginx web服务器的pod可以定义如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
```

```
- name: nginx
  image: nginx
  ports:
    - containerPort: 80
```

Pod的定义声明了某个预期状态。预期状态是Kubernetes模型中非常重要的概念。很多事情在系统中体现为预期状态，而Kubernetes有责任确保当前状态匹配预期状态。举个例子，当你创建一个Pod，你指明其中的容器进入运行。如果容器没有运行（例如程序错误，...），为了驱使容器进入预期状态，Kubernetes将持续为你（再）创建这些容器。这个过程将一直持续到该Pod被删除。

获取更多细节信息，请查阅设计文档<http://kubernetes.io/v1.1/docs/design/README.html>。

Pod管理

创建一个包含nginx server的pod (`pod-nginx.yaml` <http://kubernetes.io/v1.1/docs/user-guide/walkthrough/pod-nginx.yaml>) :

```
$ kubectl create -f docs/user-guide/walkthrough/pod-nginx.yaml
```

列出所有pods:

```
$ kubectl get pods
```

在Pod部署的大部分环境里，Pod的IP都是外部不可见的。最便捷的测试Pod是否工作的方法是创建一个BusyBoxPod并且在上面远程运行命令。请查看可执行命令文档http://kubernetes.io/v1.1/docs/user-guide/kubectl/kubectl_exec.html以找到更多细节。

如果Pod IP可以访问，你应该可以利用curl访问在80端口访问http端点。

```
$ curl http://$(kubectl get pod nginx -o=template -t={{.status.podIP}})
```

通过名字删除pod：

```
$ kubectl delete pod nginx
```

Volumes

这对于一个简单的静态网站服务器很不错，那么对于需要持续性存储情况如何？

容器文件系统仅仅存在于容器的生存周期。所以，如果你的应用状态需要忍受迁移、重启和崩溃，你需要配置一些持续性存储。

在下面的例子中，我们将创建一个Redis Pod，这个Pod包括一个named Volume和包含Volume安装路径的Volume安装点。

1、定义一个Volume：

```
volumes:
  - name: redis-persistent-storage
    emptyDir: {}
```

1. 在容器定义内定义一个Volume安装点

```
volumeMounts:
  # 名字必须与下面的Volume名字一致
  - name: redis-persistent-storage
    # 容器中的安装点
    mountPath: /data/redis
```

带有持续存储Volume的Redis Pod定义举例（pod-

`redis.yaml`<http://kubernetes.io/v1.1/docs/user-guide/walkthrough/pod-redis.yaml>）：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-persistent-storage
          mountPath: /data/redis
  volumes:
    - name: redis-persistent-storage
      emptyDir: {}
```

案例下载<http://kubernetes.io/v1.1/docs/user-guide/walkthrough/pod-redis.yaml>

注：•volume安装点名字是一个指向一个特定空目录volume的指针。•volume安装目录是容器内安装特定空目录volume的路径。

Volume 类型 •EmptyDir：创建一个在容器失效和重启情况下可以持续的新目录 •HostPath：将已经存在的目录安装在节点文件系统上（e.g. /var/logs）。

查阅 volumes <http://kubernetes.io/v1.1/docs/user-guide/volumes.html>可以得到更多细节。

多容器

注：下面的例子在语义上正确，但是某些镜像（比如 *kubernetes/git-monitor*）目前还不存在。我们正在努力将这些内容加入到可以工作的例子中。

However, often you want to have two different containers that work together. An example of this would be a web server, and a helper job that polls a git repository for new updates: 然而，通常你会希望存在两种容器在一起工作。一个例子是网站服务器，和一个可以从git仓库中拉出更新的帮助任务。

```
apiVersion: v1
kind: Pod
metadata:
  name: www
spec:
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - mountPath: /srv/www
          name: www-data
          readOnly: true
    - name: git-monitor
      image: kubernetes/git-monitor
      env:
        - name: GIT_REPO
          value: http://github.com/some/repo.git
      volumeMounts:
        - mountPath: /data
          name: www-data
  volumes:
    - name: www-data
      emptyDir: {}
```

注意我们也在这里增加了一个volume。在这个例子里，这个volume同时被安装在两个容器中。在网页服务器容器中，由于并不需要写这个目录，因此被标注为只读。

最后，我们也引入了一个给**git-monitor**容器使用的环境变量，这个变量允许我们将我们希望跟踪的特定git仓库作为参数使用

后续是什么？

继续学习Kubernetes 201<http://kubernetes.io/v1.1/docs/user-guide/walkthrough/k8s201.html> 或者阅读[guestbook example](http://kubernetes.io/v1.1/examples/guestbook/README.html)<http://kubernetes.io/v1.1/examples/guestbook/README.html>，这可以得到一个完整的用例。

Kubernetes 201 - 标签,副本控制,服务和健康检查

译者 :White 校对 :无

如果你浏览过[Kubernetes 101](#), 你能够学习到kubectl, pods, 卷, 多容器的概念。在Kubernetes 201中, 我们将学习201剩下的部分, 包含一些Kubernetes稍微高级的主题, 讨论关于应用的生产环境化, 部署以及扩展。

为了让kubectl操作的示例可以正常工作, 确保本地存在示例的目录, 从[发布](#)或者是[源](#)获取。

内容列表

- [Kubernetes 201 - 标签, 副本控制, 服务和健康检查](#)
 - [标签](#)
 - [副本控制器](#)
 - [副本控制器管理](#)
 - [服务](#)
 - [服务管理](#)
 - [健康检查](#)
 - [进程健康检查](#)
 - [应用程序健康检查](#)
 - [下面是什么?](#)

标签

已经学习了Pods以及如何创建他们, 你可能会在紧急的情况下创建许多, 许多pods。请做!但是最终你需要一个系统来通过组管理这些pods。为了实现这个功能, 在Kubernetes系统中使用标签。标签是一个键值对, 在Kubernetes中会标记到每一个对象上。标签选择器将RESTful `list` 请求传递给apiserver来获取和标签选择器匹配的对象列表。

增加一个标签, 在pod定义文件中的元数据下增加一个标签部分:

```
labels:  
  app: nginx
```

例如, 下面是一个带标签的nginx pod定义[pod-nginx-with-label.yaml](#):

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80

```

创建一个标签过的pod [pod-nginx-with-label.yaml](#):

```
$ kubectl create -f docs/user-guide/walkthrough/pod-nginx-with-label.yaml
```

列出所有标签 `app=nginx` 为nginx的pod :

```
$ kubectl get pods -l app=nginx
```

更多信息，参考[标签](#)。其他两个Kubernetes构建部分：副本控制和服务，使用标签作为其核心概念。

副本控制器

好的，现在你已经知道如何创建许多，多容器，标签化的pods, 使用他们来构建应用程序，你也许会很想开始构建一堆完整的独立pods,但如果你这样做，整个主机的运营问题摆在了眼前。例如：如何做到让pods的数量扩展，增加或者减少， 如何保证所有的pods是同质化的？

副本控制器对象可以给出这些问题的答案。Replication controllers are the objects to answer these questions.一个副本控制器可以将创建一个pod模板（一个饼干切割机如果你会的话）以及需要的一定数量的副本到单个Kubernetes对象。副本控制器也包含一个标签选择器来识别由副本控制器管理的对象集合。副本控制器通过不断的测量这个集合对象申请的容量大小来采取创建或者删除pods的动作。

例如，下面是用副本控制器来初始化两个nginx pods[replication-controller.yaml](#)):

副本控制器管理

创建nginx副本控制器[replication-controller.yaml](#):

```
$ kubectl create -f docs/user-guide/walkthrough/replication-controller.yaml
```

列出所有副本控制器：

```
$ kubectl get rc
```

通过名称删除副本控制器：

```
$ kubectl delete rc nginx-controller
```

更多信息，请参考[副本控制器](#)。

服务

一旦你拥有一个pods的副本集合，你需要在能够在应用程序层之间提供连接的抽象。例如，如果你已经有个一个副本控制器来管理后端任务，当你需要重新扩展你的后端应用的时候，你不需要重新配置你的前端应用。同样，如果后端的pods被调度（或者重新调度）到不同的机器上，你也不需要重新配置前端应用。在Kubernetes中，对服务的抽象能够达到这个目标。一个服务会提供一个指向pods集合（通过标签选择）的IP地址。如果支持的话，它同时能够提供负载均衡功能。

例如，这里有一个在之前例子中通过nginx副本控制器创建的pods之间提供负载均衡功能的服务[service.yaml](#)：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
    - port: 8000 # the port that this service should serve on
      # the container on each pod to connect to, can be a name
      # (e.g. 'www') or a number (e.g. 80)
      targetPort: 80
      protocol: TCP
      # just like the selector in the replication controller,
      # but this time it identifies the set of pods to load balance
      # traffic to.
    selector:
      app: nginx
```

服务管理

创建一个nginx服务[service.yaml](#)：

```
$ kubectl create -f docs/user-guide/walkthrough/service.yaml
```

列出所有的服务：

```
$ kubectl get services
```

对于大多数供应商，服务的IP地址外部无法访问。测试服务可以访问的最简单的方式为创建一个busybox pod在上面远程执行命令。详细内容查看[命令执行文档](#)。

一旦提供的服务IP地址可以访问，你便可以通过80端口使用curl命令来访问http终端节点：

```
$ export SERVICE_IP=$(kubectl get service nginx-service -o=template -t={{.spec.clusterIP}}
$ export SERVICE_PORT=$(kubectl get service nginx-service -o=template '-t={{(index .spec.
$ curl http://${SERVICE_IP}:${SERVICE_PORT}
```

通过名称删除服务：

```
$ kubectl delete service nginx-controller
```

一旦服务被创建，就会分配一个唯一的IP地址。这个地址同服务的生命周期绑定，服务存活期间不会发生变化。通过配置Pods来和服务通信，并且能够同一些自动被负载均衡的pods进行通信，这些服务中的pod是由标签选择器识别出来的集合中的一员。

更多信息，查看[服务](#)。

健康检查

我写的代码，永远不会崩溃，对不对？不幸的是，Kubernetes问题列表中另有说明...

一个更好的方法是使用管理系统来进行定期的应用程序健康检查和修复工作，而不是尝试编写无bug的代码。你的应用程序之外本身有一套监控系统负责监控和修复工作。很重要的一点是这个系统必须放置在应用程序外部，应用程序一旦失败以及健康检查代理是应用程序的一部分，这个代理也会失败并且你永远不会知道。在Kubernetes中，这个健康检查监控代理是Kubelet。

进程健康检查

最简单形式的健康检查是进程级别的健康检查。Kubelet不停的问Docker进程容器进程是不是还在运行，如果不在运行，容器进程就会被重启。至今为止，在你运行的全部Kubernetes示例中，这种健康检查已经开启。所有在Kubernetes中运行的单个容器都存在这种机制。

应用程序健康检查

然而，多数情况下底层健康检查是不够的，例如，考虑下面的代码：

```
lockOne := sync.Mutex{}
lockTwo := sync.Mutex{}

go func() {
    lockOne.Lock();
    lockTwo.Lock();
    ...
}()

lockTwo.Lock();
lockOne.Lock();
```

这是计算机科学中典型的“死锁”问题。从Docker的角度看，你的应用仍然在进行操作并且进程仍然在运行，但是从应用程序角度来看，你的代码锁死了，再也不会正确响应了。

为了解决这个问题，Kubernetes支持用户自己实现应用程序健康检查。这些检查通过Kubelet来确保应用程序按照你定义的“正确方式”来操作。

目前，有三种应用程序健康检查机制你可以选择：

- **HTTP 检查检查** - Kubelet会调用web钩子。如果返回200到399之间的返回码，代表成功，反之代表失败，在[这里](#)查看健康检查示例。
- **容器执行** - Kubelet会在容器里执行一条命令，如果返回值为0认为是成功。[在这里](#)查看健康检查示例。
- **TCP套接字** - Kubelet将会尝试打开一个套接字连接到容器。如果可以建立连接，认为容器是健康的，否则认为是失败的。

所有情况下，如果Kubelet一旦发现失败，容器会被重启。

可以在容器配置文件的 `livenessProbe` 部分配置你的容器的健康检查功能。你也可以指定 `initialDelaySeconds` 参数，这个参数指的是从容器启动到进行健康检查的宽松期。让你的容器有足够的时间进行任何初始化工作。

这里有一个HTTP健康检查的pod配置示例[pod-with-http-healthcheck.yaml](#)：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
    - name: nginx
      image: nginx
      # defines the health checking
      livenessProbe:
        # an http probe
        httpGet:
          path: /_status/healthz
          port: 80
        # length of time to wait for a pod to initialize
        # after pod startup, before applying health checking
        initialDelaySeconds: 30
        timeoutSeconds: 1
      ports:
        - containerPort: 80
```

更多关于健康检查信息，参考[容器探针](#)。

下一步是什么？

一个完整的应用程序，查看[留言板的例子](#)。

配置kubernetes

译者：李昂 校对：无

除了像 `kubectl run` 和 `kubectl expose` 这些必要的命令，它们在[各种地方](#)都有介绍，`kubernetes`也支持可声明式的配置。配置文件也需要必要的命令，这样就可以在代码审查中检查版本控制和文件改动，而代码审查对复杂的具有鲁棒性的可靠生产系统是非常重要的。

在声明式风格中，所有的配置都保存在YAML或者JSON配置文件中，使用Kubernetes的API资源模式（schema）作为配置的模式（schema）。`kubectl` 命令可以创建、更新、删除以及获取API资源。`kubectl` 命令用 `ApiVersion`（目前是“v1”），`kind` 资源，`name` 资源去创建合适的API路径来执行特殊的操作。

使用配置文件创建一个容器

Kubernetes是在[Pod](#)中来运行容器的。一个包含了一个简单的HelloWorld容器的Pod可以被如下的YAML文件指定：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec: # specification of the pod's contents
  restartPolicy: Never
  containers:
    - name: hello
      image: "ubuntu:14.04"
      command: ["/bin/echo", "hello", "world"]
```

`metadata.name` 的值 `hello-world`，将会成为创建成功后Pod的名称，这个名称必须在集群中唯一，而 `container[0].name` 只是容器在Pod中的昵称。`image` 就是Docker image的名称且Kubernetes默认会从[Docker Hub](#)中拉取镜像。

`restartPolicy : Never` 指明了我们只是想运行容器一次然后就终止Pod。

`Command` 覆盖了docker容器的 `Entrypoint`。命令的参数（相当于Docker的 `Cmd`）可以指定 `args` 参数，如下所示：

```
command: ["/bin/echo"]
args: ["hello", "world"]
```

创建这个pod就可以使用 `create` 命令了

```
$ kubectl create -f ./hello-world.yaml
pods/hello-world
```

当成功创建时，`kubectl` 打印出资源类型和资源名称。

配置验证

如果你不确定指定的资源是否正确，你可以 `kubectl` 帮你验证。

```
$ kubectl create -f ./hello-world.yaml --validate
```

假设我们指定的是 `entrypoint` 而不是 `command`，你会看到如下输出：

```
I0709 06:33:05.600829    14160 schema.go:126] unknown field: entrypoint
I0709 06:33:05.600988    14160 schema.go:129] this may be a false alarm,
see https://github.com/GoogleCloudPlatform/kubernetes/issues/6842
pods/hello-world
```

`kubectl create --validate` 会警告已经检测出问题，除非缺少必须的字段或者字段值不合法，最后`kubectl`还是会创建出资源。一定要小心，未知的API字段会被忽略。这个pod没有 `command` 字段而被创建，`command`字段是一个可选字段，因为image镜像可以指定 `Entrypoint`。访问[Pod API object](#)来查看合法字段列表。

环境变量和增加变量

Kubernetes[没有自动的在shell中的运行命令](#)（不是所有镜像都有shell）。如果你想在shell中运行你的命令，例如增加环境便令（使用 `env` 字段），你可以按如下做法：

```

apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec: # specification of the pod's contents
  restartPolicy: Never
  containers:
  - name: hello
    image: "ubuntu:14.04"
    env:
    - name: MESSAGE
      value: "hello world"
    command: ["/bin/sh", "-c"]
    args: ["/bin/echo \"${MESSAGE}\""]

```

然而，一个shell需要的不仅是增加环境变量。如果你使用 `$(ENVVAR)` 语法 Kubernetes 将会为你做这些事。

查看pod状态

使用get命令，你可以看到你已经创建的pod（事实上是集群中你的所有pod）。如果你在创建后用足够快的速度输入 get 命令，你会看到下面的内容：

```

$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
hello-world  0/1      Pending   0          0s

```

初始化的时候，新创建的pod还未被调度，也就是还没有节点被选中去运行它。pod创建后会进行调度但是它很快，所以你正常是不会看到pods处于未被调度的状态，除非有问题产生。

在pod被调度之后，如果节点中还没有镜像那么就会先通过docker拉取响应的镜像。一切就绪后，你会看到容器在运行：

```

$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
hello-world  1/1      Running   0          5s

```

`Ready` 列指示正在运行在pod中的容器数量。

而开始运行后很快这个容器将会被终止。`kubectl` 会显示容器已经不再运行以及推出状态：

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hello-world   0/1     ExitCode:0   0          15s
```

查看pod输出

你会想要查看你运行命令的输出。像 `docker logs` 一样，`kubectl logs` 也会显示输出：

```
$ kubectl logs hello-world
hello world
```

删除pods

当你看完的输出，你应该删除pod：

```
$ kubectl delete pod hello-world
pods/hello-world
```

就像 `create` 一样，删除成功时 `kubectl` 也会打印出资源类型和资源名称。

你也可以使用 资源/名称格式指定一个pod：

```
$ kubectl delete pods/hello-world
pods/hello-world
```

终止pods不会自动的删除，你可以观察他们的最终状态，所以请确定清理了你的已经结束的pods。

在另一方面，为了在节点中释放磁盘空间容器和他们的日志也会被自动删除。

下面的内容

[学习部署持久运行的应用](#)

管理应用：部署持续运行的应用

译者：it2af10rd 校对：无

在前面的章节里，我们了解了如何用 `kubectl run` 快速部署一个简单的复制的应用以及如何用 `pods (configuring-containers.md)` 配置并生成单次运行的容器。本文，我们将使用基于配置的方法来部署一个持续运行的复制的应用。

用配置文件生成复制品集合

Kubernetes用 `Replication Controllers` 创建并管理复制的容器集合（实际上是复制的 `Pods`）。 `Replication Controller` 简单地确保在任一时间里都有特定数量的pod副本在运行。如果运行的太多，它会杀掉一些；如果运行的太少，它会启动一些。这和谷歌计算引擎的 `Instance Group Manager` 以及AWS的Auto-scaling Group（不带扩展策略）类似。在[快速开始](#)章节里用 `kubectl run` 创建的用来跑Nginx的 `Replication Controller` 可以用下面的YAML描述：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
        ports:
          - containerPort: 80
```

和指定一个单独的Pod相比，不同的是设置了这里的`kind`域为`ReplicationController`，设定了需要的副本（`replicas`）数量以及把Pod的定义放到了`template`域下面。`pods`的名字不需要显示指定，因为它们是由 `replication controller` 的名字生成的。要查看支持的域列表，可以看[replication controller API object](#)。和创建pods一样，也可以用 `create` 命令来创建这个 `replication controller`：

```
$ kubectl create -f ./nginx-rc.yaml
replicationcontrollers/my-nginx
```

replication controller 会替换删除的或者因不明原因终止的（比如节点失败）pods，这和直接创建的pods的情况是不一样。基于这样的考量，对于一个需要持续运行的应用，即便你的应用只需要一个单独的pod，我们也推荐使用 replication controller。对于单独的pod，在配置文件里可以省略 replicas 这个域，因为不设置的时候默认就只有一个副本。

查看replication controller的状态

可以用 get 命令查看你创建的replication controller：

```
$ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
my-nginx     nginx         nginx      app=nginx  2
```

这说明你的controller会确保有两个nginx的副本。和直接创建的pod一样，也可以用 get 命令查看这些副本：

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
my-nginx-065jq   1/1    Running   0          51s
my-nginx-buaiq   1/1    Running   0          51s
```

删除replication controllers

如果想要结束你的应用并且删除replication controller。和在[快速开始](#)里一样，用下面的命令：

```
$ kubectl delete rc my-nginx
replicationcontrollers/my-nginx
```

这个操作默认会把由replication controller管理的pods一起删除。如果pods的数量比较大，这个操作要花一些时间才能完成。如果想要pods继续运行，不被删掉，可以在delete的时候指定参数 --cascade=false 。如果在删除replication controller之前想要删除pods，pods只是被替换了，因为replication controller会再起新的pods，确保pods的数量。

Labels

Kubernetes使用自定义的键值对（称为[Labels](#)）分类资源集合，例如pods和replication controller。在前面的例子里，pod的模板里只设定了一个单独的label，键是 app，值为 nginx。所有被创建的pod都带有这个label，可以用带-L参数的命令查看：

```
$ kubectl get pods -L app
NAME      READY   STATUS    RESTARTS   AGE   APP
my-nginx-afv12  0/1     Running   0          3s    nginx
my-nginx-lg99z  0/1     Running   0          3s    nginx
```

pod模板带的label默认会被复制为replication controller的label。Kubernetes中所有的资源都支持labels：

```
$ kubectl get rc my-nginx -L app
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS   APP
my-nginx     nginx         nginx       app=nginx   2           nginx
```

更重要的是，pod模板的label会被用来创建 selector，这个 selector 会匹配所有带这些 labels的pods。用 kubectl get 的[go语言模板输出格式](#)就可以看到这个域：

```
$ kubectl get rc my-nginx -o template --template="{{.spec.selector}}"
map[app:nginx]
```

如果你想要在pod模板里指定labels，但是又不想要被选中，可以显示指定 selector 来解决，不过需要确保 selector 能够匹配由pod模板创建出来的pod的label，并且不能匹配由其他 replication controller创建的pods。对于后者，最直接最保险的方法是给replication controller 分配一个独特的label，并且在pod模板和selector里都进行指定。

后续

[学习展示应用给用户和客户，以及把应用的各层拼接起来。](#)

管理应用：连接应用

译者：it2af10rd 校对：无

Kubernetes容器连接模型

既然已经有了一个可持续运行的、可复制的应用，现在就可以在网络中将它暴露出来了。在讨论Kubernetes的网络连接方式之前，很值得和Docker的常规网络连接方式做个对比。

Docker默认使用私有网络连接方式，所以只有在同一台物理机器上的容器之前才可以通信。为了能让Docker容器可以跨节点通信，必须要给机器的IP地址分配端口号，这个端口之后会被用来转发或者路由给容器。很明显，这意味着容器要么很小心地协调使用端口，要么有动态分配地端口。

在一定的规模下，为多个开发者协调端口号非常困难。这也会把集群级别的问题暴露给用户，这是在用户的控制之外的。Kubernetes假定pods之间是可以通信的，不管它们落到哪个主机上。我们给每个pod指定集群私有的IP地址（cluster-private-IP address），所以不需要显示地创建pod之间的链接，也不需要映射容器的端口到主机的端口。这意味着pod里的容器可以在本机（localhost）上访问各自的端口，而且在没有NAT的情况下，集群中所有的pod也可以互相可见的。本文剩下的内容将会详细阐述如何在这样的网络模型中运行可靠的服务。

这个指南中用了一个简单的nginx服务来演示验证这个概念（proof of concept）。同样的原理也在一个更完整的[Jenkins CI 应用](#)中体现了。

在集群中暴露Pod

在前面的例子中已经演示过，让我们把注意力集中在网络的视角再来一次。创建一个nginx的 Pod，请注意它定义了容器的端口：

```
$ cat nginxrc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

这使得它从集群中的任一节点都可以被访问到。检查一下供pod运行的节点：

```
$ kubectl create -f ./nginxrc.yaml
$ kubectl get pods -l app=nginx -o wide
my-nginx-6isf4   1/1       Running   0           2h       e2e-test-beeps-minion-93ly
my-nginx-t26zt   1/1       Running   0           2h       e2e-test-beeps-minion-93ly
```

检查pod的IP地址：

```
$ kubectl get pods -l app=nginx -o json | grep podIP
      "podIP": "10.245.0.15",
      "podIP": "10.245.0.14",
```

你应该可以ssh到集群里的任何一个节点，而且用curl也能够访问这两个IP。要注意的是容器并没有用节点的80端口，也没用任何特殊的会把流量路由到pod的NAT规则。这意味着你可以在同一个节点上用同样的 `containerPort` 配置运行多个nginx pod，而且通过IP就可以在其他pod或者集群里的其他节点访问它们。和Docker类似，端口也可以在节点的网络接口中发布出来，但是在Kubernetes的这种网络模型下，这样的需求从根本上减少了。

如果你很好奇，可以在[how we achieve this](#)读到更多细节。

创建Service

现在我们有了运行态的nginx，它们运行在一个水平的，集群范围的地址空间内。理论上，我们已经可以和这些pod直接交互了，但是如果一个节点死掉了会发生什么？它里面的Pod也会死掉，然后Replication Controller会创建一个新的Pod，但是IP是不一样的。这就是Service可

以解决的问题。

Kubernetes Service是对在集群中某处运行的一系列Pod的逻辑集合的抽象定义，这些Pod提供的功能是一样的。每个Service被创建的时候会被分配一个唯一的IP地址（也叫 `clusterIP`）。这个地址和Service绑定，只要Service活着就不会改变。Pod可以配置成和Service交互，并且知道和Service的通信会被自动地负载均衡到Service成员中的某个Pod。

可以用下面的yaml为两个nginx副本创建一个Service：

```
$ cat nginxsvc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    app: nginx
```

这个定义会创建一个Service，这个Service会把带有`app=nginx` Label的Pod的TCP 80端口暴露到Service的抽象端口（`targetPort`：是容器可以接收流量的端口，`port`：是Service的抽象端口，可以是除用来访问Service的端口之外的任何端口）。在Service定义中支持的所有域可以在[Service API 对象](#)中查看。

查看Service：

```
$ kubectl get svc
NAME      LABELS      SELECTOR      IP(S)      PORT(S)
nginxsvc  app=nginx  app=nginx  10.0.116.146  80/TCP
```

在前面提到过，Service是由一组Pod支撑的。这些Pod通过`endpoints`暴露出来。Service会持续评估Selector，并把结果发送给Endpoint对象（也叫做`nginxsvc`）。当一个Pod死了之后，它就会被自动地从Endpoint里面删掉，能够匹配Service的Selector的新Pod会被自动加到Endpoint里。检查Endpoint的时候也会看到IP和前一步里创建的Pod是一样的：As mentioned previously, a Service is backed by a group of pods. These pods are exposed through `endpoints`. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named `nginxsvc`. When a pod dies, it is automatically removed from the endpoints, and new pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the ips are the same as the pods created in the first step:

```
$ kubectl describe svc nginxsvc
Name:           nginxsvc
Namespace:      default
Labels:          app=nginx
Selector:        app=nginx
Type:           ClusterIP
IP:             10.0.116.146
Port:           <unnamed> 80/TCP
Endpoints:      10.245.0.14:80,10.245.0.15:80
Session Affinity: None
No events.

$ kubectl get ep
NAME      ENDPOINTS
nginxsvc  10.245.0.14:80,10.245.0.15:80
```

现在你应该可以从集群里的任一节点上用curl命令访问**10.0.116.146:80**上的nginx Service了。要注意的是Service的IP完全是虚拟的，跟物理网络没有关系，如果你对它的工作原理有兴趣，可以去看看[service proxy](#)。 You should now be able to curl the nginx Service on **10.0.116.146:80** from any node in your cluster. Note that the Service ip is completely virtual, it never hits the wire, if you're curious about how this works you can read more about the [service proxy](#).

访问Service

Kubernetes支持两种主要的模式来发现Service：环境变量和DNS。环境变量在安装之后就可以直接使用，DNS模式需要[kube-dns 集群插件](#)。 Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the [kube-dns cluster addon](#).

环境变量

当一个Pod在某个节点上运行的时候，`kubelet` 为每个活跃的Service添加一系列的环境变量。这会引入环境变量排序的问题。想知道为什么，检查一下运行中的nginx Pod的环境： When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx pods:

```
$ kubectl exec my-nginx-6isf4 -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
```

注意这里并没有提到Service，这是因为这些副本是在Service之前创建的。这样做的另一个缺点是，调度器也许会把两个Pod放到相同的机器上，如果机器出问题，整个Service就不工作了。正确的方式是把这两个Pod杀掉，然后等Replication Controller重新创建它们。现在Service是在Pod副本之前存在了，因此Service获得了调度器级别的Pod扩散能力（只要所有的节点的容量是一样的），而且环境变量也是正确的： Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 pods and waiting for the replication controller to recreate them. This time around the Service exists before the replicas. This will give you scheduler level Service spreading of your pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
$ kubectl scale rc my-nginx --replicas=0; kubectl scale rc my-nginx --replicas=2;
$ kubectl get pods -l app=nginx -o wide
NAME        READY   STATUS    RESTARTS   AGE     NODE
my-nginx-5j8ok   1/1    Running      0          2m     node1
my-nginx-90vaf   1/1    Running      0          2m     node2

$ kubectl exec my-nginx-5j8ok -- printenv | grep SERVICE
KUBERNETES_SERVICE_PORT=443
NGINXSERVICE_HOST=10.0.116.146
KUBERNETES_SERVICE_HOST=10.0.0.1
NGINXSVC_SERVICE_PORT=80
```

DNS

Kubernetes提供了一个DNS集群插件Service，这个Service使用 skydns 自动给其他Service分配DNS。可以用下面的命令检查它是否在集群中运行： Kubernetes offers a DNS cluster addon Service that uses skydns to automatically assign dns names to other Services. You can check if it's running on your cluster:

```
$ kubectl get services kube-dns --namespace=kube-system
NAME      LABELS      SELECTOR      IP(S)      PORT(S)
kube-dns  <none>     k8s-app=kube-dns  10.0.0.10  53/UDP
                                         53/TCP
```

如果它没有运行，你可以[启用它](#)。本篇的剩余部分假设你已经有一个长期IP（nginxsvc）的Service，以及一个DNS服务器 If it isn't running, you can [enable it](#). The rest of this section will assume you have a Service with a long lived ip (nginxsvc), and a dns server that has assigned a name to that ip (the kube-dns cluster addon), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname). Let's create another pod to test this:

```
$ cat curlpod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: curlpod
spec:
  containers:
    - image: radial/busyboxplus:curl
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
      name: curlcontainer
  restartPolicy: Always
```

And perform a lookup of the nginx Service

```
$ kubectl create -f ./curlpod.yaml
default/curlpod
$ kubectl get pods curlpod
NAME      READY     STATUS    RESTARTS   AGE
curlpod   1/1      Running   0          18s

$ kubectl exec curlpod -- nslookup nginxsvc
Server:  10.0.0.10
Address 1: 10.0.0.10
Name:    nginxsvc
Address 1: 10.0.116.146
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A [secret](#) that makes the certificates accessible to pods

You can acquire all these from the [nginx https example](#), in short:

```
$ make keys secret KEY=/tmp/nginx.key CERT=/tmp/nginx.crt SECRET=/tmp/secret.json
$ kubectl create -f /tmp/secret.json
secrets/nginxsecret
$ kubectl get secrets
NAME          TYPE      DATA
default-token-il9rc  kubernetes.io/service-account-token  1
nginxsecret    Opaque
```

Now modify your nginx replicas to start a https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

```
$ cat nginx-app.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
  labels:
    app: nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    app: nginx
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

Noteworthy points about the nginx-app manifest:

- It contains both rc and service specification in the same file
- The [nginx server](#) serves http traffic on port 80 and https traffic on 443, and nginx Service exposes both ports.

- Each container has access to the keys through a volume mounted at /etc/nginx/ssl. This is setup before the nginx server is started.

```
$ kubectl delete rc,svc -l app=nginx; kubectl create -f ./nginx-app.yaml  
replicationcontrollers/my-nginx  
services/nginxsvc  
services/nginxsvc  
replicationcontrollers/my-nginx
```

At this point you can reach the nginx server from any node.

```
$ kubectl get pods -o json | grep -i podip  
    "podIP": "10.1.0.80",  
node $ curl -k https://10.1.0.80  
...  
<h1>Welcome to nginx!</h1>
```

Note how we supplied the -k parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Lets test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

```
$ cat curlpod.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: curlrc
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
        - name: secret-volume
          secret:
            secretName: nginxsecret
      containers:
        - name: curlpod
          command:
            - sh
            - -c
            - while true; do sleep 1; done
          image: radial/busyboxplus:curl
          volumeMounts:
            - mountPath: /etc/nginx/ssl
              name: secret-volume

$ kubectl create -f ./curlpod.yaml
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
curlpod   1/1      Running   0          2m
my-nginx-7006w 1/1      Running   0          24m

$ kubectl exec curlpod -- curl https://nginxsvc --cacert /etc/nginx/ssl/nginx.crt
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used **NodePort**, so your nginx https replica is ready to serve traffic on the internet if your node has a public ip.

```
$ kubectl get svc nginxsvc -o json | grep -i nodeport -C 5
{
    "name": "http",
    "protocol": "TCP",
    "port": 80,
    "targetPort": 80,
    "nodePort": 32188
},
{
    "name": "https",
    "protocol": "TCP",
    "port": 443,
    "targetPort": 443,
    "nodePort": 30645
}

$ kubectl get nodes -o json | grep ExternalIP
{
    "type": "ExternalIP",
    "address": "104.197.63.17"
}
--
},
{
    "type": "ExternalIP",
    "address": "104.154.89.170"
}
$ curl https://104.197.63.17:30645 -k
...
<h1>Welcome to nginx!</h1>
```

Lets now recreate the Service to use a cloud load balancer, just change the **Type** of Service in the `nginx-app.yaml` from **NodePort** to **LoadBalancer**:

```
$ kubectl delete rc, svc -l app=nginx
$ kubectl create -f ./nginx-app.yaml
$ kubectl get svc -o json | grep -i ingress -A 5
    "ingress": [
        {
            "ip": "104.197.68.43"
        }
    ]
}
$ curl https://104.197.68.43 -k
...
<title>Welcome to nginx!</title>
```

What's next?

Learn about more Kubernetes features that will help you run containers reliably in production.

管理应用：在生产环境中使用Pods和容器

译者：张鑫 校对：无

永久性存储

容器中的文件系统会随着容器的停止而消失；如果容器重启或出错停止，则会使得容器文件中的数据丢失，而且容器即使重新运行也会忘记之前的数据和状态。如果需要比容器的生命更为长久的存储方案，我们需要使用数据卷。这个需求对于有状态服务更为重要，例如数据库，键-值存储等。

举例来说，Redis是一个键-值存储缓存（我们在[Guestbook](#)和一些其他的例子中使用过）。我们可以用如下方法为它加一个数据卷：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis
spec:
  template:
    metadata:
      labels:
        app: redis
        tier: backend
    spec:
      # 为Pod提供一个数据卷
      volumes:
        - name: data
          emptyDir: {}
      containers:
        - name: redis
          image: kubernetes/redis:v1
          ports:
            - containerPort: 6379
          # 将数据卷加载到Pod中
          volumeMounts:
            - mountPath: /redis-master-data
              name: data    # 必须和上面定义的数据卷名字匹配
```

`emptyDir` 数据卷的生命周期与Pod的生命周期一致，因此会比任何一个容器更长久；如果某个容器失效或重启了，我们的数据还会继续存在。

除了本地磁盘所支持的 `emptyDir`，Kubernetes还支持多种网络存储方案，包括GCE的PD，EC2的EBS。这些存储方案更适合用来存储关键数据，同时还能处理在节点上动态加载，卸载数据卷。详情请查看[数据卷文档](#)。

分发认证信息

很多应用需要认证信息，例如密码，OAuth令牌，和TLS的秘钥等，用来和其他的应用、数据库、服务等进行认证。在容器镜像中或是通过环境变量来存储这些私密信息并不理想，因为只要能够访问这些镜像，或是Pod/容器配置，或是宿主机的文件系统，或是Docker Daemon，一个人就可以获取这些私密信息。

Kubernetes提供一个叫做秘密的机制用来帮助分发敏感的认证信息。一个秘密是一种含有map数据的资源。举例来说，一个含有用户名和密码的秘密可以定义如下：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: dmFsdWUtMg0K
  username: dmFsdWUtMQ0K
```

如同其它资源一样，这个秘密可以通过 `create` 命令来生成，同时可以通过 `get` 命令来查看：

```
$ kubectl create -f ./secret.yaml
secrets/mysecret
$ kubectl get secrets
NAME          TYPE           DATA
default-token-v9pyz  kubernetes.io/service-account-token  2
mysecret       Opaque         2
```

要使用秘密，我们需要在一个pod的模版中引用它。`secret` 数据卷可以让我们像内存目录一样来把秘密加载到容器里。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis
spec:
  template:
    metadata:
      labels:
        app: redis
        tier: backend
    spec:
      volumes:
        - name: data
          emptyDir: {}
        - name: supersecret
          secret:
            secretName: mysecret
      containers:
        - name: redis
          image: kubernetes/redis:v1
          ports:
            - containerPort: 6379
          # Mount the volume into the pod
          volumeMounts:
            - mountPath: /redis-master-data
              name: data    # must match the name of the volume, above
            - mountPath: /var/run/secrets/super
              name: supersecret
```

如果要查看更多详情，可以参考关于秘密的详细[文档](#)，[例子](#)和[设计](#)。

与私有镜像仓库进行认证

我们还可以利用秘密来传递私有镜像仓库的认证信息。

第一步，我们生成一个 `.dockercfg` 文件（例如可以通过运行 `docker login <registry.domain>` 命令）。然后，我们将生成的 `.dockercfg` 文件放到一个秘密资源里，例如：

```
$ docker login
Username: janedoe
Password: *****
Email: jdoe@example.com
WARNING: login credentials saved in /Users/jdoe/.dockercfg.
Login Succeeded

$ echo $(cat ~/.dockercfg)
{ "https://index.docker.io/v1/": { "auth": "ZmFrZXBhc3N3b3JkMTIK", "email": "jdoe@example.com" }

$ cat ~/.dockercfg | base64
eyAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogeyAiYXV0aCI6ICJabUZyWlhCaGMzTjNiM0prTVRJSyIsI

$ cat > /tmp/image-pull-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockercfg: eyAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogeyAiYXV0aCI6ICJabUZyWlhCaGMzTjNiM0prTVRJSyIsI
type: kubernetes.io/dockercfg
EOF

$ kubectl create -f ./image-pull-secret.yaml
secrets/myregistrykey
```



现在你就可以生成一个pod，并通过添加 `imagePullSecrets` 部分来引用上述生成的秘密：

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

协同容器

Pods支持将多个容器一起调度，保证它们运行在同一个宿主机上。他们可以被用来支持一整个应用栈，但主要应用场景是用来运行辅助程序。经典的例子包括数据发布、获取程序，代理等等。

这些容器往往需要通过文件系统相互通信，而这个通信过程可以通过在两个容器中都加入同样的数据卷来实现。这个模式的一个例子就是网络服务器通过git存储库来等待程序最近更新：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: www-data
          emptyDir: {}
      containers:
        - name: nginx
          image: nginx
          # This container reads from the www-data volume
          volumeMounts:
            - mountPath: /srv/www
              name: www-data
              readOnly: true
        - name: git-monitor
          image: myrepo/git-monitor
          env:
            - name: GIT_REPO
              value: http://github.com/some/repo.git
          # This container writes to the www-data volume
          volumeMounts:
            - mountPath: /data
              name: www-data
```

更多例子请查看我们的[博客文章](#)以及[演示文稿幻灯片](#)。

资源管理

只有在有足够的CPU和存储的地方Kubernetes的调度器才会放置应用程序，但在知道应用程序需要多少资源的时候也只能这么做。指定的CPU太小的后果就是，如果太多其他容器被安排到同一个节点的话，CPU资源会被耗尽。同样地，容器也可能会由于在耗尽存储要不到其他内存的情况下而不可预见的终止运行，对于大内存的应用程序尤其可能出现这种情况。

如果没有指定资源需求，那么资源数量在名义上是被假定的（这个默认值是通过Limitrange给默认值的命名空间设定的）。它也可以通过`kubectl describe limitrange limits`被查看。）我们可以明确地把要求的资源数量列举如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          resources:
            limits:
              # cpu units are cores
              cpu: 500m
              # memory units are bytes
              memory: 64Mi
```

容器的运行超过指定限度的时候会因为内存耗尽而终止程序，所以制定一个略高于预期的值提高了总体的可信度。

如果不确定该请求多少资源，那么我们可以先不指定启动应用程序资源，然后根据资源使用情况量监控来确定适当的值。

健康检查

许多应用程序经过长时间的运行，最终过渡到了无法运行的状态，除了重启，无法恢复。Kubernetes提供活性探针来检测并且对相应状况进行相应的补救措施。

通常来检测一个应用程序的方法是用HTTP，详细说明如下：

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          livenessProbe:
            httpGet:
              # Path to probe; should be cheap, but representative of typical behavior
              path: /index.html
              port: 80
            initialDelaySeconds: 30
            timeoutSeconds: 1

```

很多时候，应用程序只是暂时无法服务，之后会自己恢复。通常在这样的情况下，我们不愿意关闭应用程序，但是也不想发送请求，因为应用层程序不正确回应或者根本不回应。常见的这样的场景是加载大数据或者在应用程序启动期间配置文件。Kubernetes提供探针来检测和缓解这种情况。探针的配置只是用到 `readinessProbe` 字段。由容器组成的pod报告说他们还没准备好通过Kubernetes服务接收通信量。

要知道更多细节（比如：如何指定命令基础上的调查），请查看[walkthrough](#)里的例子，[standalone](#)的例子，还有[文档](#)。

生命周期插件接口和终止通知

当然，节点和应用程序在任何时候都可能会运行失败，但是当应用程序的终止被人为下达的时候，很多应用程序受益于正常关机来完成正在处理的请求。为了支持这样的情况，Kubernetes支持两种信息通知：Kubernetes将发送终止信号给应用程序，这种信号可以被接收到来平滑的终止程序。如果应用程序没有尽早终止，无条件终止命令将会在10秒后发送。Kubernetes支持（可选择的）预停止生命周期插件接口，这个动作执行在发送终止信号之前。

预停止插件接口的细节跟调查细节相似，但是没有时限相关参数。例如：

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          lifecycle:
            preStop:
              exec:
                # SIGTERM triggers a quick exit; gracefully terminate instead
                command: ["/usr/sbin/nginx", "-s", "quit"]

```

终止信息

为了达到一个相当高水平的实用性，特别是为了积极开发应用，快速调试失败是很重要的。除了一般的日志采集，Kubernetes还能通过查出重大错误原因来加速调试，并在某种程度上通过kubectl或者UI陈列出来。可以指定一个'terminationMessagePath'来让容器写下它的“death rattle”，比如声明失败消息，堆栈跟踪，免责条款等等。默认途径是’/dev/termination-log’。

这是一个小例子：

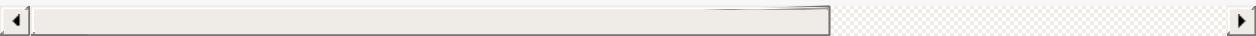
```

apiVersion: v1
kind: Pod
metadata:
  name: pod-w-message
spec:
  containers:
    - name: messenger
      image: "ubuntu:14.04"
      command: ["/bin/sh", "-c"]
      args: ["sleep 60 && /bin/echo Sleep expired > /dev/termination-log"]

```

消息和上一个（也就是：最近的）终止的其他状态是一起被记录的。

```
$ kubectl create -f ./pod.yaml
pods/pod-w-message
$ sleep 70
$ kubectl get pods/pod-w-message -o template -t "{{range .status.containerStatuses}}{{.la
Sleep expired
$ kubectl get pods/pod-w-message -o template -t "{{range .status.containerStatuses}}{{.la
0
```



Kubernetes用户指南：管理应用：管理部署

译者：it2af10rd 校对：无

组织资源配置

很多应用程序需求创建多重资源，比如Replication Controller和Service。对于多重资源的管理，可以简单地把它们聚在一起，放在同一个文件里（在YAML里面用`---`分隔）。例如：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

多重资源可以和单一资源以同样的方式创建：

```
$ kubectl create -f ./nginx-app.yaml
services/my-nginx-svc
replicationcontrollers/my-nginx
```

资源会按照在文件中出现的顺序被创建。因此，最好先定义Service，这会保证当Replication Controller创建Pod的时候，调度器可以铺开和这个Service相关的Pod。

`kubectl create` 也接受多重 `-f` 参数：

```
$ kubectl create -f ./nginx-svc.yaml -f ./nginx-rc.yaml
```

而且除单个文件之外还可以传一个目录：

```
$ kubectl create -f ./nginx/
```

`kubectl` 会读取任何后缀名为 `.yaml`，`.yml` 或者 `.json` 的文件。

推荐的做法是把同样的微服务或者应用层相关的资源放在同一个文件里，然后把整个应用相关的文件都放在同一个目录里。如果应用中的各个层通过DNS互相绑定了，那么你就可以简单地一起部署它们。

URL也可以成为配置文件的源，这样对于直接部署GitHub上的配置文件非常好用：

```
$ kubectl create -f https://raw.githubusercontent.com/GoogleCloudPlatform/kubernetes/master/replicationcontrollers/nginx
```

kubectl批量操作

`kubectl` 可以批量做的不仅仅只有创建资源。它也可以从配置文件里抽取资源名字来执行其他操作，特别是用来删除创建的重复资源：

```
$ kubectl delete -f ./nginx/replicationcontrollers/my-nginx/services/my-nginx-svc
```

在只有两个资源的情况下，可以简单地在命令行上指定资源或者名字：

```
$ kubectl delete replicationcontrollers/my-nginx services/my-nginx-svc
```

为了处理更大量的资源，可以用Label来过滤资源。Selector用 `-l` 参数来指定：

```
$ kubectl delete all -lapp=nginx replicationcontrollers/my-nginx services/my-nginx-svc
```

因为 `kubectl` 用它可以接受的语法输出资源名字，用 `$()` 或者 `xargs` 就可以简单地把这些才做串起来：

```
$ kubectl get $(kubectl create -f ./nginx/ | grep my-nginx)
CONTROLLER CONTAINER(S) IMAGE(S) SELECTOR      REPLICAS
my-nginx    nginx        nginx       app=nginx   2
NAME         LABELS      SELECTOR      IP(S)        PORT(S)
my-nginx-svc app=nginx  app=nginx   10.0.152.174 80/TCP
```

有效地使用Label

目前为止我们使用的例子，对任何资源来说，最多只有一个Label。有许多必须要使用多重Label的场景，来把不同的集合区分开。

比如，不同的应用 The examples we've used so far apply at most a single label to any resource. There are many scenarios where multiple labels should be used to distinguish sets from one another.

For instance, different applications would use different values for the `app` label, but a multi-tier application, such as the [guestbook example](#), would additionally need to distinguish each tier. The frontend could carry the following labels:

```
labels:
  app: guestbook
  tier: frontend
```

while the Redis master and slave would have different tier labels, and perhaps even an additional role label:

```
labels:
  app: guestbook
  tier: backend
  role: master
```

and

```
labels:
  app: guestbook
  tier: backend
  role: slave
```

The labels allow us to slice and dice our resources along any dimension specified by a label:

```
$ kubectl create -f ./guestbook-fe.yaml -f ./redis-master.yaml -f ./redis-slave.yaml
replicationcontrollers/guestbook-fe
replicationcontrollers/guestbook-redis-master
replicationcontrollers/guestbook-redis-slave
$ kubectl get pods -Lapp -ltier -lrole
NAME READY STATUS RESTARTS AGE APP TIER
guestbook-fe-4nlpb 1/1 Running 0 1m guestbook front
guestbook-fe-ght6d 1/1 Running 0 1m guestbook front
guestbook-fe-jpy62 1/1 Running 0 1m guestbook front
guestbook-redis-master-5pg3b 1/1 Running 0 1m guestbook backe
guestbook-redis-slave-2q2yf 1/1 Running 0 1m guestbook backe
guestbook-redis-slave-qgazl 1/1 Running 0 1m guestbook backe
my-nginx-divi2 1/1 Running 0 29m nginx <n/a>
my-nginx-o0ef1 1/1 Running 0 29m nginx <n/a>
$ kubectl get pods -lapp=guestbook,role=slave
NAME READY STATUS RESTARTS AGE
guestbook-redis-slave-2q2yf 1/1 Running 0 3m
guestbook-redis-slave-qgazl 1/1 Running 0 3m
```

Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. For example, it is common practice to deploy a canary of a new application release (specified via image tag) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out. For instance, a new release of the guestbook frontend might carry the following labels:

```
labels:
  app: guestbook
  tier: frontend
  track: canary
```

and the primary, stable release would have a different value of the track label, so that the sets of pods controlled by the two replication controllers would not overlap:

```
labels:
  app: guestbook
  tier: frontend
  track: stable
```

The frontend service would span both sets of replicas by selecting the common subset of their labels, omitting the track label:

```
selector:
  app: guestbook
  tier: frontend
```

Updating labels

Sometimes existing pods and other resources need to be relabeled before creating new resources. This can be done with kubectl label. For example:

```
$ kubectl label pods -lapp=nginx tier=fe
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-v4-9gw19  1/1     Running   0          14m
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-v4-hayza  1/1     Running   0          13m
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-v4-mde6m   1/1     Running   0          17m
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-v4-sh6m8   1/1     Running   0          18m
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-v4-wf0f4   1/1     Running   0          16m
$ kubectl get pods -lapp=nginx -ltier
NAME        READY   STATUS    RESTARTS   AGE   TIER
my-nginx-v4-9gw19  1/1     Running   0          15m   fe
my-nginx-v4-hayza  1/1     Running   0          14m   fe
my-nginx-v4-mde6m   1/1     Running   0          18m   fe
my-nginx-v4-sh6m8   1/1     Running   0          19m   fe
my-nginx-v4-wf0f4   1/1     Running   0          16m   fe
```

Scaling your application

When load on your application grows or shrinks, it's easy to scale with kubectl. For instance, to increase the number of nginx replicas from 2 to 3, do:

```
$ kubectl scale rc my-nginx --replicas=3
scaled
$ kubectl get pods -lapp=nginx
NAME        READY   STATUS    RESTARTS   AGE
my-nginx-1jgkf  1/1     Running   0          3m
my-nginx-divi2  1/1     Running   0          1h
my-nginx-o0ef1  1/1     Running   0          1h
```

Updating your application without a service outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag, as in the canary deployment scenario above. kubectl supports several update operations, each of which is applicable to different scenarios.

To update a service without an outage, kubectl supports what is called “[rolling update](#)”, which updates one pod at a time, rather than taking down the entire service at the same time. See [the rolling update design document](#) and [the example of rolling update](#) for more information.

Let's say you were running version 1.7.9 of nginx:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

To update to version 1.9.1, you can use `kubectl rolling-update --image`:

```
$ kubectl rolling-update my-nginx --image=nginx:1.9.1
Creating my-nginx-ccba8fdb8cc8160970f63f9a2696fc46
```

In another window, you can see that kubectl added a deployment label to the pods, whose value is a hash of the configuration, to distinguish the new pods from the old:

```
$ kubectl get pods -lapp=nginx -Ldeployment
NAME                               READY   STATUS    RESTARTS   AGE
my-nginx-1jgkf                     1/1    Running   0          1h
my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-k156z 1/1    Running   0          1m
my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-v95yh 1/1    Running   0          35s
my-nginx-divi2                      1/1    Running   0          2h
my-nginx-o0ef1                      1/1    Running   0          2h
my-nginx-q6all
```

kubectl rolling-update reports progress as it progresses:

```
Updating my-nginx replicas: 4, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas: 1
At end of loop: my-nginx replicas: 4, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas:
At beginning of loop: my-nginx replicas: 3, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 rep
Updating my-nginx replicas: 3, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas: 2
At end of loop: my-nginx replicas: 3, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas:
At beginning of loop: my-nginx replicas: 2, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 rep
Updating my-nginx replicas: 2, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas: 3
At end of loop: my-nginx replicas: 2, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas:
At beginning of loop: my-nginx replicas: 1, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 rep
Updating my-nginx replicas: 1, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas: 4
At end of loop: my-nginx replicas: 1, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas:
At beginning of loop: my-nginx replicas: 0, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 rep
Updating my-nginx replicas: 0, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas: 5
At end of loop: my-nginx replicas: 0, my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 replicas:
Update succeeded. Deleting old controller: my-nginx
Renaming my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 to my-nginx
my-nginx
```

If you encounter a problem, you can stop the rolling update midway and revert to the previous version using --rollback:

```
$ kubectl rolling-update my-nginx --image=nginx:1.9.1 --rollback
Found existing update in progress (my-nginx-ccba8fdb8cc8160970f63f9a2696fc46), resuming.
Found desired replicas. Continuing update with existing controller my-nginx.
Stopping my-nginx-02ca3e87d8685813dbe1f8c164a46f02 replicas: 1 -> 0
Update succeeded. Deleting my-nginx-ccba8fdb8cc8160970f63f9a2696fc46
my-nginx
```

This is one example where the immutability of containers is a huge asset.

If you need to update more than just the image (e.g., command arguments, environment variables), you can create a new replication controller, with a new name and distinguishing label value, such as:

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx-v4
spec:
  replicas: 5
  selector:
    app: nginx
    deployment: v4
  template:
    metadata:
      labels:
        app: nginx
        deployment: v4
    spec:
      containers:
        - name: nginx
          image: nginx:1.9.2
          args: ["nginx", "-T"]
          ports:
            - containerPort: 80

```

and roll it out:

```

$ kubectl rolling-update my-nginx -f ./nginx-rc.yaml
Creating my-nginx-v4
At beginning of loop: my-nginx replicas: 4, my-nginx-v4 replicas: 1
Updating my-nginx replicas: 4, my-nginx-v4 replicas: 1
At end of loop: my-nginx replicas: 4, my-nginx-v4 replicas: 1
At beginning of loop: my-nginx replicas: 3, my-nginx-v4 replicas: 2
Updating my-nginx replicas: 3, my-nginx-v4 replicas: 2
At end of loop: my-nginx replicas: 3, my-nginx-v4 replicas: 2
At beginning of loop: my-nginx replicas: 2, my-nginx-v4 replicas: 3
Updating my-nginx replicas: 2, my-nginx-v4 replicas: 3
At end of loop: my-nginx replicas: 2, my-nginx-v4 replicas: 3
At beginning of loop: my-nginx replicas: 1, my-nginx-v4 replicas: 4
Updating my-nginx replicas: 1, my-nginx-v4 replicas: 4
At end of loop: my-nginx replicas: 1, my-nginx-v4 replicas: 4
At beginning of loop: my-nginx replicas: 0, my-nginx-v4 replicas: 5
Updating my-nginx replicas: 0, my-nginx-v4 replicas: 5
At end of loop: my-nginx replicas: 0, my-nginx-v4 replicas: 5
Update succeeded. Deleting my-nginx
my-nginx-v4

```

You can also run the [update demo](#) to see a visual representation of the rolling update process.

In-place updates of resources

Sometimes it's necessary to make narrow, non-disruptive updates to resources you've created. For instance, you might want to add an [annotation](#) with a description of your object. That's easiest to do with kubectl patch:

```
$ kubectl patch rc my-nginx-v4 -p '{"metadata": {"annotations": {"description": "my front my-nginx-v4\n\napiVersion: v1\nkind: ReplicationController\nmetadata:\n  annotations:\n    description: my frontend running nginx\n...'}}}'\n\n
```

The patch is specified using json.

For more significant changes, you can get the resource, edit it, and then replace the resource with the updated version:

```
$ kubectl get rc my-nginx-v4 -o yaml > /tmp/nginx.yaml\n$ vi /tmp/nginx.yaml\n$ kubectl replace -f /tmp/nginx.yaml\nreplicationcontrollers/my-nginx-v4\n$ rm $TMP
```

The system ensures that you don't clobber changes made by other users or components by confirming that the resourceVersion doesn't differ from the version you edited. If you want to update regardless of other changes, remove the resourceVersion field when you edit the resource. However, if you do this, don't use your original configuration file as the source since additional fields most likely were set in the live state.

Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may just want to make a recursive change immediately, such as to fix broken pods created by a replication controller. To change such fields, use replace --force, which deletes and re-creates the resource. In this case, you can simply modify your original configuration file:

```
$ kubectl replace -f ./nginx-rc.yaml --force\nreplicationcontrollers/my-nginx-v4\nreplicationcontrollers/my-nginx-v4
```

What's next?

- Learn about how to use kubectl for application introspection and debugging.
- Tips and tricks when working with config

Kubernetes UI

译者：李昂 校对：无

Kubernetes有一个基于web的用户界面，它可以图表化显示当前集群状态。

访问UI

Kubernetes界面默认是作为集群插件部署的。要访问它需要进入 `https://<kubernetes-master>/ui` 这个地址，之后会重定向到 `https://<kubernetes-master>/api/v1/proxy/namespaces/kube-system/services/kube-ui/#/dashboard/`。

如果你发现你无法访问UI，那有可能是Kubernetes UI服务在你的集群上还没有启动。如果是这样，你可以手动开启UI服务：

```
kubectl create -f cluster/addons/kube-ui/kube-ui-rc.yaml --namespace=kube-system
kubectl create -f cluster/addons/kube-ui/kube-ui-svc.yaml --namespace=kube-system
```

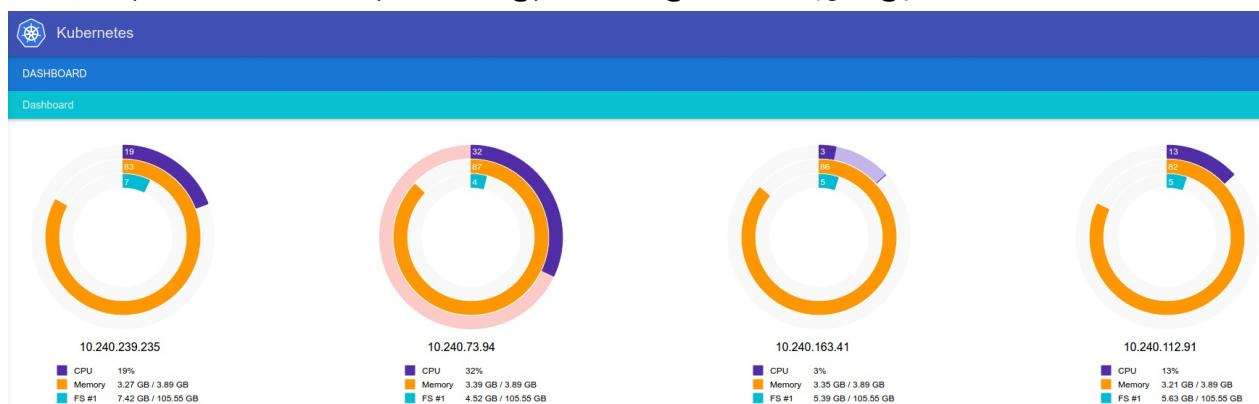
通常，这些应该通过 `kube-addons.sh` 脚本自动地被运行在主节点上。

使用UI

Kubernetes UI可以被用于监控你当前的集群，例如查看资源利用率或者检查错误信息。但是你不能用UI修改集群。

节点资源使用

访问Kubernetes UI后，你可以看到主页动态列出的你当前集群的所有节点，而且还会有关信息列出，包括内部IP地址，CPU状态，内存状态和文件系统状态。



Dashboard视图

在页面的右上方点击“Views”按钮可以看到其他可用的视图，包括Explore, Pods, Nodes, Replication Controllers, Services和Events。

Explore视图

Explore视图允许你轻松看到当前集群的pods, replication controller和services。

The screenshot shows the Kubernetes Explore view. At the top, there's a navigation bar with tabs for DASHBOARD, Dashboard > Explore, and Views (dropdown). Below the navigation is a search bar with placeholder text "Search for resources". A "Group by" dropdown menu is open, showing options like "type", "name", and "label selector". The main content area is divided into sections based on resource type:

- Type: pod**: Shows a list of pods including elasticsearch-logging-v1-8hw8, elasticsearch-logging-v1-nkfv2, fluentd-elasticsearch-kubernetes-minion-0whl, fluentd-elasticsearch-kubernetes-minion-4jdf, fluentd-elasticsearch-kubernetes-minion-epbe, fluentd-elasticsearch-kubernetes-minion-ju5z, frontend-pv6w6, frontend-pxaza, kibana-logging-v1-pdfsk, kube-dns-v3-ip8g, monitoring-heapster-v3-hp6du, monitoring-influx-grafana-v1-2zhup, redis-master-ic771, redis-slave-45e11, and redis-slave-mvgjb.
- Type: replicationController**: Shows a list of replication controllers including elasticsearch-logging-v1, frontend, kibana-logging-v1, kube-dns-v3, monitoring-heapster-v3, monitoring-influx-grafana-v1, redis-master, and redis-slave.
- Type: service**: Shows a list of services including elasticsearch-logging, kibana-logging, kube-dns, kubernetes, monitoring-grafana, monitoring-heapster, monitoring-influxdb, redis-master, and redis-slave.

“Group by”下拉列表允许你用类型、名称、主机等条件对资源分组。

This screenshot shows the same Explore view as above, but with the "Group by" dropdown menu open, revealing more detailed grouping options. The menu includes:

- Type: **pod**: Options include component (selected), provider, and version.
- Type: **replicationController**: Options include host (selected) and component.
- Type: **service**: Options include host and component.

你也可以点击资源列表的下拉三角来生成一个过滤器，有多重过滤类型可供选择。

The screenshot shows the Kubernetes Dashboard interface. At the top, there's a navigation bar with the Kubernetes logo and the word "Dashboard". Below it, a breadcrumb navigation shows "Dashboard > Explore". A dropdown menu "Group by: type" is open, showing options for "pod", "replicationController", and "service". The "pod" section is expanded, showing a list of pods including "elasticsearch-logging-v1-8hwI8", "elasticsearch-logging-v1-n", "frontend-pv6w6", "frontend-pxaza", and "kibana-logging-v1". To the right of the list is a "FILTER" sidebar with fields for "host", "k8s-app", "kubernetes.io/cluster-service", "type", and "version". The "replicationController" and "service" sections are also partially visible below.

若是要看每项资源的更多细节，就在上面点击。

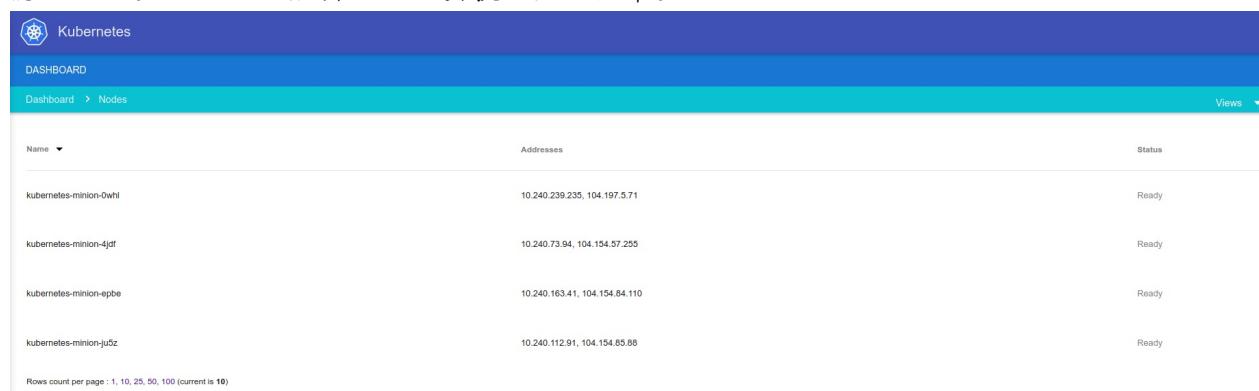
The screenshot shows the Kubernetes Dashboard interface, specifically the "Pod" view for the pod named "elasticsearch-logging-v1-8hwI8". At the top, there's a navigation bar with the Kubernetes logo and the word "Dashboard". Below it, a breadcrumb navigation shows "Dashboard > Pod". A back button "BACK" is visible on the left. The main content area displays the pod details:

- Name:** elasticsearch-logging-v1-8hwI8
- Status:** Running on [Host IP]
- Created:** Jul 1, 2015 3:33:36 PM
- Host Networking:** /10.240.239.235
- Pod Networking:** 10.244.0.9 es-port: 9200, es-transport-port: 9300
- Labels:**
 - k8s-app: elasticsearch-logging
 - kubernetes.io/cluster-service: true
 - version: v1
- Containers:**

	Name	Image	Ready	Restarts	State
	elasticsearch-logging	gcr.io/google_containers/elasticsearch:1.4	true	0	Running
					Started: Jul 1, 2015 3:33:48 PM

其他视图

其他视图（Pods, Nodes, Replication Controllers, Services, and Events）简单列出了每种资源的信息。你也可以点击其他种类获取更多细节。



The screenshot shows the Kubernetes UI interface. At the top, there's a dark blue header with the Kubernetes logo and the word "Kubernetes". Below it is a blue navigation bar with the text "DASHBOARD". Underneath is a teal header bar with the text "Dashboard > Nodes". On the right side of the teal bar, there's a "Views" dropdown menu. The main content area is a table with three columns: "Name", "Addresses", and "Status". There are four rows of data:

Name	Addresses	Status
kubernetes-minion-Owhl	10.240.239.235, 104.197.5.71	Ready
kubernetes-minion-4jdf	10.240.73.94, 104.154.57.255	Ready
kubernetes-minion-epbe	10.240.163.41, 104.154.84.110	Ready

At the bottom left of the table, it says "Rows count per page : 1, 10, 25, 50, 100 (current is 10)".

更多信息

想了解更多信息，访问[Kubernetes UI development document](#)在web目录。

使用kubectl exec检查容器中的环境变量

译者：李昂 校对：无

Kubernetes通过环境变量来暴露[services](#)。使用kubectl exec去检查环境变量会很方便。

首先我们创建一个pod和一个service,

```
$ kubectl create -f examples/guestbook/redis-master-controller.yaml  
$ kubectl create -f examples/guestbook/redis-master-service.yaml
```

等到pod的状态为Running和Ready,

```
$ kubectl get pod  
NAME             READY   REASON      RESTARTS   AGE  
redis-master-ft9ex  1/1     Running     0          12s
```

然后我们就可以检查pod的环境变量了,

```
$ kubectl exec redis-master-ft9ex env  
...  
REDIS_MASTER_SERVICE_PORT=6379  
REDIS_MASTER_SERVICE_HOST=10.0.0.219  
...
```

使用kubectl exec检查挂载的数据卷 (volume)

使用kubectl exec检查你希望挂载的数据卷 (volume) 同样很方便。首先还是创建一个pod并且挂载一个数据卷到这个pod的/data/redis目录,

```
kubectl create -f docs/user-guide/walkthrough/pod-redis.yaml
```

等待pod状态为Running和Ready,

```
$ kubectl get pods  
NAME    READY   REASON      RESTARTS   AGE  
storage  1/1     Running     0          1m
```

然后我们就可以使用 `kubectl exec` 去验证数据卷已经挂载到了`/data/redis`目录，

```
$ kubectl exec storage ls /data  
redis
```

使用**kubectl exec**在**pod**中打开一个**bash**终端

在pod中打开一个终端毕竟才是最直接检查pod的方式。假设pod已经在运行，

```
$ kubectl exec -ti storage -- bash  
root@storage:/data#
```

如上所做，你就可以打开pod的终端了。

应用：kubectl proxy和apiserver proxy

译者：李昂 校对：无

你已经看过了 kubectl proxy 和 apiserver proxy 的[基本知识](#)。本页将为你展示如何一起使用它们去访问运行在你的工作站上的Kubernetes集群提供的服务（[kube-ui](#)）。

获取kube-ui的apiserver proxy URL

kube-ui是作为一个集群的插件进行部署的。想要找到他的apiserver proxy URL：

```
$ kubectl cluster-info | grep "KubeUI"  
KubeUI is running at https://173.255.119.104/api/v1/proxy/namespaces/kube-system/services
```

如果这条命令不能找到URL，试试[这些步骤](#)。

从你的本地工作站连接到kube-ui服务

上面说的proxy URL是由apiserver提供来访问kube-ui服务的。你要是在本地想访问它仍然需要用apiserver认证。[kubectl proxy](#) 可以处理认证。

```
$ kubectl proxy --port=8001  
Starting to serve on localhost:8001
```

现在你就可以在你的本地工作站访问kube-ui服务了，使用如下地址

<http://localhost:8001/api/v1/proxy/namespaces/kube-system/services/kube-ui>

应用 : kubectl port-forward

译者：李昂 校对：无

kubectl port-forward命令转发了本地端口到pod端口的连接。它的[手册](#)现在这里可以查看。相比于[kubectl proxy](#), `kubectl port-forward`也可以转发TCP流量而 `kubectl proxy` 只能转发HTTP流量。本页阐述了如何使用 `kubectl port-forward` 去连接Redis数据库，这在数据库查错中很有用。

创建一个Redis主服务

```
$ kubectl create examples/redis/redis-master.yaml
pods/redis-master
```

等Redis主服务的pod状态变为Running和Ready。

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
redis-master 2/2     Running   0          41s
```

连接到Redis主服务

Redis主服务监听在端口6379，使用下面命令可以验证，

```
$ kubectl get pods redis-master -t='{{(index (index .spec.containers 0).ports 0).containerPort}}'
6379
```

然后我们转发redis主服务pod的端口6379到本地6379端口，

```
$ kubectl port-forward -p redis-master 6379:6379
I0710 14:43:38.274550    3655 portforward.go:225] Forwarding from 127.0.0.1:6379 -> 6379
I0710 14:43:38.274797    3655 portforward.go:225] Forwarding from [::1]:6379 -> 6379
```

想要验证连接成功，我们可以在本地启动一个redis客户端，

```
$ redis-cli  
127.0.0.1:6379> ping  
PONG
```

现在我们就可以从本地来检查数据库了。

Kubernetes集群管理员手册

译者：李昂 校对：无

管理员手册适合于创建了或者正在管理Kubernetes集群的人。阅读管理员手册最好已经熟悉[用户手册](#)中的一些概念。

规划一个集群

如何部署创建一个Kubernetes集群有很多不同的例子可以参考。很多例子都在这个[列表](#)中列出了。我们称在列表中每个组合为一个发行版。

在选择需要的手册之前，有些事情需要提前考虑：

- 你只是想在你的笔记本上试一下Kubernetes还是创建一个多节点的高可用集群？这两种模式都是支持的，但是有些发行版只适合一种情况或其他情况。
- 你是否会使用主机模式的Kubernetes集群，像是[GKE](#)这种或是你自己启动的。
- 你的集群在机房中或者云服务器上（IaaS）吗？Kubernetes不直接支持混合集群。我们推荐建立多个集群而不是跨越遥远的地点。
- 你要在现实主机或者虚拟机上运行Kubernetes吗？Kubernetes都是支持的，请看不同的发行版。
- 你只是想要运行一个集群还是想要为Kubernetes项目做开发？如果是后者，最好选择一个其他开发者都使用的发行版。某些发行版只有二进制可执行文件但却提供了多种选择。
- 不是所有的发行版都被很好的维护着。最近版本的Kubernetes列出的一些只是为了测试。
- 如果你在机房配置Kubernetes，你需要考虑最适合的[网络模型](#)。
- 如果你正在设计高可用集群，你可以看看[运行在多个地点的集群](#)。
- 你可以运行多种[组件](#)构成的集群来使你自己变成更加熟练。

创建一个集群

从[列表](#)中挑选一个入门手册然后按它指导。如果没有入门指南适合你，你也可以参考入门手册中的一部分指导。

一种可选的网络模型就是OpenVSwitch GRE/VxLAN 网络（[ovs-networking.md](#)），使用OpenVSwitch设置的网络可以使Kubernetes不同节点之间的pod通信。

如果你正在使用Salt修改已经存在指南，这篇文档会告诉你[Salt如何应用于Kubernetes项目](#)。

管理一个集群以及升级

[管理集群](#)

管理节点

[管理节点](#)

可选的集群服务

使用SkyDNS ([dns.md](#)) 整合DNS：为Kubernetes服务解析DNS名字。 使用[Kibana](#)记录日志。

多租户支持

[资源配置](#) ([resource-quota.md](#))

安全

Kubernetes容器环境 ([docs/user-guide/container-environment.md](#)) :阐述了Kubelet在管理Kubernetes节点上的容器时的环境。

安全访问API Server：[访问api](#)

认证：[authentication](#)

授权：[authorization](#)

接纳控制：[admission_controllers](#)

Kubernetes架构

译者 : kz 校对 : 无

一个运行着的Kubernetes集群包含node agents(即kubelet)和主要的组件（API, 调度器等等），其构建于分布式存储的基础之上。这个图显示了我们的理想的最终状态，尽管我们还在对一些部分做改动，如让kubelet自身运行在容器之内，和让调度器百分之百的插件式。[!p 架构图](#)

Kubernetes Node (Kubernetes节点)

当研究系统的架构的时候，我们会将其划分成运行子worker节点上的service和那些组成集群级别控制面板的service。

Kubernetes Node拥有能运行应用容器的必要服务和从主系统管理的service。

每一个node当然运行着Docker。Docker负责处理下载镜像，运行容器的细节。

kubelet

kubelet管理着pod和它们的容器，它们的镜像，它们的卷，等等。

kube-proxy

每一个node也运行着一个简单的网络代理和负载均衡器。这反映出定义在Kubernetes API中的每个节点的service能做简单的TCP和UDP流转发（轮流的方式）到一组后端。

service的端点现在是通过DNS或者通过环境变量来发现的。这些变量解析到有service代理管理着的端口。

Kubernetes控制面板

Kubernetes控制面板被分成一组组件。目前，它们都运行在一个单一的master节点上，但是这很快就会改变，以支持高可用的集群。这些节点一起工作提供了一个集群统一的视图。

etcd

所有持久化的master状态保存在一个etcd的节点中。这给保存配置数据稳定性提供了很好的存储。有watch的支持，协作的组件可以在更改时很快的被通知到。

Kubernetes API Server

apiserver服务着Kubernetes API。其目标是作为简单的CRUD式的服务，然后绝大多数/所有的业务逻辑都在单独的组件或者插件里面实现。他主要处理REST操作，做数据验证，并更新对应的etcd里面的对象（然后最终其他的存储）。

Scheduler

调度器通过binding API将没有调度的pod绑定到node。调度器是插件式的，我们期望支持多种集群的调度器，最终在未来甚至支持用户提供的调度器。

Kubernetes Controller Manager Server

所有的其他集群级别的功能目前是通过Controller Manager来执行的。例如Endpoint对象通过endpoint控制器偶来创建和更新，node是通过node控制器来发现管理和监控的。这些可能会最终被拆分到单独的组件，以让他们可以独立的插件化。

replication控制器是一个在简单API的上层的机制。我们最终计划将其移植成可插入式的机制，一旦有一个被实现了。

Kubernetes集群管理指南：集群组件

译者：Nancy 校对：无

本文概述了各种二进制组件，这些组件的运行可以提供一个有效的Kubernetes集群。

主组件

主组件可以提供集群的控制面板。例如，主组件负责对有关集群做全局性的决策（如，调度），以及探测和对集群事件的响应（如，当Replication Controller的'replicas'字段不满足时，启动一个新的Pod）。

理论上，主组件可以运行在集群的任何节点上。但是，为了简化，当前的设置脚本通常在同样的VM上启动所有的主组件，而且不会在该VM上运行用户容器。参考[high-availability.md](#)，一个Multi-Master-VM设置的实例。

即便在未来，如果Kubernetes完全自托管，将只允许主组件调度节点的子集，以限制和运行用户Pod的合作运行，减少节点损害安全漏洞的可能范围。

Kube-apiserver

[Kube-apiserver](#)暴漏出Kubernetes API；它是Kubernetes控制面的前端，可以横向扩展（如，可以通过运行多个API检测一个apiserver-- [high-availability.md](#)）。

etcd

[etcd](#)作为Kubernetes的后备存储。所有的集群数据都存储在这里。一个Kubernetes集群的适当管理包括对ETCD的数据备份计划。

Kube控制器管理者

[Kube控制器管理者](#)是运行控制器的一个二进制文件，处理集群中日常事务的后端进程。逻辑上，每个控制器是独立的进程，但是为了减少系统中移动片的数量，它们都被编译成一个独立的二进制，并且运行在一个单一的进程中。

这些控制器包括：

- Node Controller
 - 当节点异常时，负责查看和响应。
- Replication Controller

- 负责对系统中的每一个控制器对象，保持Pod的正确值。
- Endpoints Controller
 - 填充端点对象（即加入Service & Pod）
- Service Account & Token Controllers
 - 为新的Namespace创建默认账户和API接入的Token。
- ...等等。

Kube调度

[Kube调度](#)观察新创建、还没有分配节点的Pod，并且从中选择一个节点运行。

扩展插件

扩展插件属于Pod和Service，实现了集群功能。它们不能在主VM上运行，但是目前能够调用API创建这些Pod和Service的默认启动脚本可以在主VM上运行。可参考[kube-master-addons](#)

扩展插件对象创建在“kube-system”空间。

扩展插件实例：

- [DNS](#)提供集群本地DNS。
- [kube-ui](#)为集群提供图形用户界面。
- [fluentd-elasticsearch](#)提供日志存储。参考[gcp version](#)。
- [cluster-monitoring](#)监控集群。

Node组件

Node组件运行在每一个节点上，维护运行Pod，并给它们提供Kubernetes运行环境。

Kubelet

[Kubelet](#)是主节点代理。它：

- 监督已经分配到节点的Pod（或者被apiserver分配，或者通过本地配置文件）：
 - 挂载Pod需要的Volume
 - 下载Pod的Secret
 - 通过Docker（或，rkt）运行Pod容器
 - 定期执行任何请求的容器活性探针（container liveness probes）
 - 给系统的其它部分报告Pod的状态，如果有必要，可以创建“mirror pod”
- 返回节点的状态给系统的其它部分。

Kube代理

[Kube代理](#)使得Kubernetes服务抽象化，在主机上维护网络规则，并且执行连接转发。

Docker

Docker主要用于运行容器。

Rkt

Rkt，实验证明，可以作为Docker的替代者。

Monit

Monit是一个轻量级进程维护系统，可以保持Kubelet和Docker正常运行。

Kube-API Server

译者：Nancy 校对：无

概要

Kubernetes API服务器为API对象验证和配置数据，这些对象包含Pod, Service, Replication Controller等等。API Server提供REST操作以及前端到集群的共享状态，所有其它组件可以通过这些共享状态交互。

kube-apiserver

选项

```
--admission-control="AlwaysAdmit": 集群中资源的Admission Controller的插件的有序列表，分别使用逗号分隔。
--admission-control-config-file"": Admission Controller配置文件。
--advertise-address=<nil>: 广播API Server给所有集群成员的IP地址。其它集群都可以访问该IP地址，如果设置为<nil>，则广播给所有集群成员的IP地址。
--allow-privileged[=false]: true, 表示允许特权容器。
--authorization-mode="AlwaysAllow": 安全端口授权插件的有序列表，分别以逗号分隔，AlwaysAllow, AlwaysDeny。
--authorization-policy-file"": 授权策略的CSV文件，使用于--authorization-mode=ABAC模式的配置。
--basic-auth-file"": 如果配置该选项，该文件会通过HTTP基本认证允许API Server安全端口的请求。
--bind-address=0.0.0.0: 服务--read-only-port和--secure-port端口的IP地址。相关接口必须是其它集群的IP地址。
--cert-dir="/var/run/kubernetes": TLS证书的目录（默认/var/run/kubernetes）。如果配置--tls-cert-dir，该值将被忽略。
--client-ca-file"": 如果设置，任何提交客户端证书的请求都会验证与相关客户端证书的CommonName的身份。
--cloud-config"": 云提供商配置文件的路径，空表示没有该配置文件。
--cloud-provider"": 云服务的提供商，空表示没有该提供商。
--cluster-name="kubernetes": 集群实例的前缀。
--cors-allowed-origins=[]: CORS的允许起源（allowed origins, 翻译待考虑）的列表，用逗号分隔。一个或多个URL。
--etcd-config"": ETCD客户端的配置文件，与--etcd-servers配置项互斥。
--etcd-prefix="/registry": ETCD中所有资源路径的前缀。
--etcd-servers=[]: ETCD服务器（http://ip:port）列表，以逗号分隔。与--etcd-config配置项互斥。
--etcd-servers-overrides=[]: 每个资源ETCD服务器覆盖文件，以逗号分隔。独立覆盖格式，group/resource。
--event-ttl=1h0m0s: 保留事件的时间值，默认1小时。
--experimental-keystone-url"": 如果Passed，激活Keystone认证插件。
--external-hostname"": 为Master生成外部URLs使用的主机名。
--google-json-key"": 用户Google Cloud Platform Service Account JSON Key认证。
--insecure-bind-address=127.0.0.1: 非安全端口（所有接口都设置为0.0.0.0）的服务IP地址。默认是本地IP。
--insecure-port=8080: 不安全且没有认证的进程访问端口，默认8080。假设防火墙规则设置该端口从集群外部访问。
--kubelet-certificate-authority"": 证书路径。证书授权文件。
--kubelet-client-certificate"": TLS客户端证书文件路径。
--kubelet-client-key"": TLS客户端秘钥文件路径。
--kubelet-https[=true]: 使用https建立Kubelet连接。
--kubelet-port=10250: Kubelet端口。
```

```
--kubelet-timeout=5s: Kubelet操作Timeout值。
--log-flush-frequency=5s: 日志缓冲秒数的最大值。
--long-running-request-regexp="/|^(watch|proxy)(/$)|(logs?|portforward|exec|attach)"
--master-service-namespace="default": Namespace, 该Namespace的Kubernetes主服务应该注入Pod。
--max-connection-bytes-per-sec=0: 如果非零, 表示每个用户连接的最大值, 字节数/秒, 当前只适用于长时
--max-requests-inflight=400: 给定时间内运行的请求的最大值。如果超过最大值, 该请求就会被拒绝。零表示
--min-request-timeout=1800: 这是个可选字段, 表示一个请求处理的最短时间, 单位是秒。在超时之前, 这个
--oidc-ca-file"": 如果设置该选项, Oidc-ca-file中的相关机构会验证OpenID服务的证书。否则, 会使用主
--oidc-client-id"": 如果设置了oidc-issuer-url字段, 该字段, OpenID连接客户端的客户ID也必须设置。
--oidc-issuer-url"": OpenID发行的URL, 只接受HTTPS协议。如果设置该字段, 将被用来验证OIDC JSON W
--oidc-username-claim="sub": 。默认值之外的那些值, 可能是不唯一的, 可变的。这个标志还在尝试中, 详情
--profiling[=true]: 通过web接口进行分析 host:port/debug/pprof/
--runtime-config=: key=value键值对集, 描述运行时配置, 也会回传输到apiserver。apis/<groupVersion>
--secure-port=6443: 用于HTTPS的认证和授权。0表示不支持HTTPS服务。
--service-account-key-file"": 该文件包含RPM-encoded x509 RSA的私钥和公钥, 用于验证ServiceAcc
--service-account-lookup[=false]: true, 表示验证Service Account的Token做为Authentication
--service-cluster-ip-range=<nil>: CIDR标记的IP范围, 从中分配IP给服务集群。该范围不能与分配给Pod
--service-node-port-range=: NodePort可见性服务的端口范围, 包含范围的两端。如'30000-32767', 包含
--ssh-keyfile"": 如果非空, 使用安全SSH代理到该节点, 用该秘钥文件。
--ssh-user"": 如果非空, 使用安全SSH代理到该节点, 用该用户名。
--storage-versions="extensions/v1beta1,v1": 存储资源的版本。不同的组存储在不同的版本里面, 指定格
--tls-cert-file"": 该文件包含HTTPS的x509证书。(CA证书, 如果存在, 连接在服务器证书之后)。如果支持
--tls-private-key-file"": 该文件包含x509私钥匹配项--tls-cert-file.
--token-auth-file"": 该文件使用Token验证保护API Server的安全端口。
--watch-cache[=true]: 可以在API Server查看缓存。
```

授权插件

译者：Nancy 校对：无

授权是Kubernetes认证中的一个独立部分，参考[认证部分](#)的文档。

授权应用在主（安全）API服务端口的所有HTTP访问请求。

任何请求的授权检查都会通过访问策略比较该请求上下文的属性，（比如用户，资源和Namespace）。API的调用必须符合一些规则，按顺序执行。

下面的选项都是可行的，可通过标志位选择：

- --authorization-mode=AlwaysDeny
- --authorization-mode=AlwaysAllow
- --authorization-mode=ABAC

AlwaysDeny会阻止所有的请求（测试中使用的）。AlwaysAllow允许所有请求，如果不需要授权，可以使用这个参数。ABAC（Attribute-Based Access Control）用于已经配置的用户授权规则。

ABAC模式

请求属性

一个授权请求可配置五个参数：

- 用户（用户是否是用户串）
- 组（用户所属组名的列表）
- 请求只读性(GETs是只读的)
- 资源权限
 - 仅适用于API端点，例如/api/v1/namespaces/default/pods。对于杂端点（miscellaneous endpoints，翻译有待考虑），如/version，是空串。
- 可访问对象的Namespace，空串的Namespace，该空串端点不支持命名对象。

我们期望增加更多的属性，允许更细粒度的访问控制，并协助策略管理。

策略文件格式

ABAC模式，也可以指定参数：--authorization-policy-file=SOME_FILENAME

该文件格式每行有一个JSON对象，不会有封闭列表或者映射，而仅仅每行有一个映射。

每行是一个“策略对象”。一个策略对象是包含以下属性的一个映射：

- user, 字符型；来自--token-auth-file, 如果指定用户，必须匹配认证用户的用户名。
- group, 字符型；如果指定用户组，必须匹配认证用户所属组的其中一个。
- readonly, 布尔型, true表示该策略仅适用于GET操作。
- resource, 字符型；一个资源来自一个URL, 比如Pod。
- namespace, 字符型；一个Namespace字符串。

未设置的属性同类型设置为zero的属性（如，空串，0, false），含义是相同的。但是，没设置的属性首选是可读性。

在未来，策略可以展现在JSON格式中，并通过REST接口进行管理。

授权算法

一个请求属性和一个策略对象的属性是相关的。

一个请求被接受时，其属性是确定的。未知属性默认设置为0, 如空串, 0, false。

未设置的属性将匹配相应属性的任何值。

检查属性的元组在每一个策略文件中每个策略的匹配正确性。至少有一行匹配到该请求属性，则授权该请求（但之后的验证也许会失败）。

为了让每个用户都做些事情，编写一个策略用于用户未设置的属性。（To permit any user to do something, write a policy with the user property unset. To permit an action Policy with an unset namespace applies regardless of namespace. 翻译有待考虑）

示例

1. Alice能够做任何事情：{"user":"alice"}
2. Kubelet能够读任何pods : {"user":"kubelet", "resource": "pods", "readonly": true}
3. Kubelet能够读写事件 : {"user": "kubelet", "resource": "events"}
4. Bob只能在"projectCaribou"命名空间中读pods : {"user":"bob", "resource": "pods", "readonly": true, "ns": "projectCaribou"}

[完整文件示例](#)

服务账户的快速标记

一个服务账户会自动生成一个用户。该用户的名称生成是根据如下命名规范：

```
system:serviceaccount:<namespace>:<serviceaccountname>
```

创建一个新的Namespace，也会附带创建一个新的服务账户，格式如下：

```
system:serviceaccount:<namespace>:default
```

例如，如果你想要在Kube系统授予API默认账户的所有权限，你需要在规则文件中添加如下一行：

```
{"user":"system:serviceaccount:kube-system:default"}
```

重启apiserver使新添加的规则生效。

插件开发

其余实现的开发相对容易，API服务会调用Authorizer接口：

```
type Authorizer interface {
    Authorize(a Attributes) error
}
```

确认是否允许每一个API行为。

一个授权插件是实现该接口的一个模块。授权插件源码路径：

pkg/auth/authorization/\$MODULENAME.

一个授权模块完全是用Go语言实现的，或者可以调用一个远程授权服务。授权模块有自己的缓存，减少对相同或相似参数重复授权的成本。开发人员应该考虑缓存和权限撤销之间的相互交互。

认证插件

译者：Nancy 校对：无

Kubernetes使用客户端证书，令牌，或者HTTP基本身份验证用户的API调用。

在API服务器中配置—client-ca-file=SOMEFILE选项，就会启动客户端证书认证。引用文件必须包含一个或多个认证机制，通过认证机制验证传给API服务器的客户端证书。当一个客户端证书通过认证，该证书主题的名字就被作为该请求的用户名。

在API服务器中配置选项：--token-auth-file=SOMEFILE，启动Token认证。目前，Token没有有效期，必须重启API服务，Token列表的更改才会生效。

令牌文件格式路径：plugin/pkg/auth/authenticator/token/tokenfile/...，该文件是一个CSV文件，含有三行：Token，用户名，用户uid。

当http客户端使用Token认证，apiserver需要含有Bearer Sometoken值的一个Authorization头。

OpenID Connect ID Token，传递下面的参数给apiserver：

- --oidc-issuer-url (必须) API Server连接到OpenID提供者的URL，只接受HTTPS协议。
- --oidc-client-id (必须) API Server用于验证Token用户，合法的ID Token在它的aud参数(aud claims 翻译待考虑)中包含该client-id。
- --oidc-ca-file (可选) API Server用于和OpenID提供者建立和验证安全连接。
- --oidc-username-claim (可选，实验性参数) 指定用户名对应的OpenID。默认设置为sub参数，在指定域中是唯一的，不可变的。集群管理员可以选择其它参数如email，作为用户名，但不保证其唯一性和不变性。

请注意，这个标志仍然处于试验阶段，如果我们可以处理更多关于OpenID用户和Kubernetes用户的映射关系，便可以开始使用。因此，未来的变化还是很有可能的。

目前，该ID Token会通过一些第三方应用程序获取。这意味着应用程序必须和API Server共享该配置--oidc-client-id。

如Token文件，当从HTTP客户端使用Token认证方式，API Server希望在Authorization头添加一个Bearer SOMETOKEN的值。

启动基本认证，需要在API Server配置选项—basic_auth_file=SOMEFILE。当前，基本认证凭据是无限期的，而且重启API Server，密码的修改才会生效。需要注意，基本认证方式是更安全的模式，更容易使用，更通用。

基本认证文件格式， plugin/pkg/auth/authenticator/password/passwordfile/...， 该文件是一个 CSV文件， 含有三个值， 密码， 用户名和用户 id。 如果在HTTP客户端使用基本认证， API Server需要一个值是Basic BASE64ENCODEDUSER:PASSWORD的Authorization头。

Keystone认证会在API Server启动的时候把--experimental-keystone-url='AuthURL'参数传给 API Server， 该认证就会生效。 该插件在

plugin/pkg/auth/authenticator/request/keystone/keystone.go文件中实现。 有关如何使用 Keystone去管理项目和用户的详细信息，请参考[Keystone文档](#)。 请注意， 该插件还处于试验阶段， 很可能还会变化。 请参考有关该插件的[讨论和计划](#)了解更多细节。

插件开发

我们计划给Kubernetes API Server解决Token问题。 使用“bedrock”认证用户， 外部提供者给 Kubernetes。 我们计划使Kubernetes和一个Bedrock认证提供者（如github.com, google.com, Enterprise Directory, Kerberos等等）之间的接口开发更容易。

附录

创建证书

客户端证书认证， 用户可以手动产生证书， 也可以使用已经存在的脚本部署。

部署脚本路径在cluster/saltbase/salt/generate-cert/make-ca-cert.sh。 执行该脚本需要两个参数， 一个是API Server的IP地址， 另一个是IP：或者DNS：主题备用名称的列表。 该脚本会产生三个文件， ca.crt, server.crt和server.key。 最后， 添加下面的参数作为API Server的启动参数， --client-ca-file=/srv/kubernetes/ca.crt, --tls-cert-file=/srv/kubernetes/server.cert, --tls-private-key-file=/srv/kubernetes/server.key。

Easyrsa可以用来为你的集群手动生成证书。

1. 下载， 解压， 初始化Easyrsa的补丁版本。

```
curl -L -o https://storage.googleapis.com/kubernetes-release/easy-rsa/easy-rsa.tar.gz
tar xzf easy-rsa.tar.gz
cd easy-rsa-master/easyrsa3
./easyrsa init-pki
```

2. 产生一个CA. (--batch设置自动模式。 --req-cn使用默认CN。)

```
./easyrsa --batch "--req-cn=${MASTER_IP}@date +%s" build-ca nopass
```

3. 产生服务器证书和秘钥。 (build-server-full [文件名] : 给客户端和服务器生成一个本地的

秘钥对和信号)

```
./easyrsa --subject-alt-name="IP:${MASTER_IP}" build-server-full kubernetes-master no
```

4. 复制pki/ca.crt, pki/issued/kubernetes-master.crt, pki/private/kubernetes-master.key到你的目录。
5. 记得填写参数--client-ca-file=/yourdirectory/ca.crt, --tls-cert-file=/yourdirectory/server.cert, --tls-private-key-file=/yourdirectory/server.key, 并作为API Server的启动参数。

Openssl也可以用来给你的集群手动生成证书。

1. 使用2048bit生成ca.key : openssl genrsa -out ca.key 2048
2. 根据ca.key生成ca.crt。(-days设置证书的有效时间)。

```
openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER_IP}" -days 10000 -out c
```

3. 使用2048bit生成server.key : openssl genrsa -out server.key 2048
4. 根据server.key生成server.csr。

```
openssl req -new -key server.key -subj "/CN=${MASTER_IP}" -out server.csr
```

5. 根据ca.key, ca.crt和server.csr生成server.crt。

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server
```

6. 查看证书 : openssl x509 -noout -text -in ./server.crt。最后, 记得填写参数, 并作为API Server的启动参数。

API Server端口配置

译者：Nancy 校对：无

本文档介绍Kubernetes apiserver服务的端口以及如何访问这些端口。观众都是集群管理员，都想定制他们自己的集群或者了解细节。

有关集群访问的更多问题，在[Accessing the cluster](#)都有覆盖。

服务的Ports和IPs

Kubernets API Server进程提供Kuvernetes API。通常情况下，有一个进程运行在单一 kubernetes-master节点上。

默认情况，Kubernetes API Server提供HTTP的两个端口：

1.本地主机端口

- HTTP服务
- 默认端口8080，修改标识--insecure-port
- 默认IP是本地主机，修改标识—`insecure-bind-address`
- 在HTTP中没有认证和授权检查
- 主机访问受保护

2.Secure Port

- 默认端口6443，修改标识—`secure-port`
- 默认IP是首个非本地主机的网络接口，修改标识—`bind-address`
- HTTPS服务。设置证书和秘钥的标识，`--tls-cert-file`, `--tls-private-key-file`
- [认证方式](#)，令牌文件或者客户端证书
- 使用基于策略的[授权方式](#)

3.移除：只读端口

- 基于安全考虑，会移除只读端口，使用[Service Account](#)代替。

代理和防火墙规则

此外，在某些配置文件中有一个代理（nginx）作为API Server进程运行在同一台机器上。该代理是HTTPS服务，认证端口是443，访问API Server是本地主机8080端口。在这些配置文件里，Secure Port通常设置为6443。

防火墙规则，通常配置运行外部HTTPS通过443端口访问。

上面的都是默认配置，反应了Kubernetes使用kube-up.sh如何部署到Google Compute Engine。其它的云提供商可能会有所不同。

用例和IP:Ports

有关服务端口，有三种不同的配置，有各自的应用场景。

1. Kubernetes集群之外的客户端，例如在台式机上运行kubectl命令的人员。目前，通过运行在kubernetes-master机器上面的代理（nginx）访问本地主机端口。该代理可以使用证书认证或者Token认证方式。
2. 运行在Kubernetes的Container里面的进程需要从API Server中读取。目前，这些进程都是用[Service Account](#)
3. 调度器和Controller管理进程，需要对API做读写操作。目前，这些都必须运行在API Server同样的主机上面，使用本地主机。未来，这些进程将会使用Service Account服务，避免共存的必要。
4. Kubelets，需要对API做读写操作，并且同API Server相比，它必须运行在不同的机器上面。Kubelet使用Secure Port获取Pod，发现Pod可以看到的服务，并且记录这些事件。在集群启动事件内，分布设置Kubelet凭证。Kubelet和Kube-proxy可以使用证书认证和Token认证方式。

预期变化

- Policy会限制Kubelet通过身份认证端口实行的一些操作。
- 调度器和Controller管理也会使用Secure Port。他们可以运行在不同的机器上。

Admission Controller

译者：Nancy 校对：无

目录

- Admission Controller 『参见以下内容』
 - Admission Controller是什么？『参见下文“什么是Admission Controller”』
 - 为什么使用Admission Controller？『参见下文“为什么使用Admission Controller”』
 - 如何使用接入控制插件？『参见下文“如何接入该插件”』
 - 每个插件的功能『参见下文“每个插件的功能是什么”』
 - AlwaysAdmit 『参见下文“AlwaysAdmin插件”』
 - AlwaysDeny 『参见下文“AlwaysDeny插件”』
 - DenyExecOnPrivileged (废弃) 『参见下文“DenyExecOnPrivileged (废弃)插件”』
 - DenyEscalatingExec 『参见下文“DenyEscalatingExec插件”』
 - ServiceAccount 『参见下文“ServiceAccount插件”』
 - SecurityContextDeny 『参见下文“SecurityContextDeny插件”』
 - ResourceQuota 『参见下文“ResourceQuota插件”』
 - LimitRanger 『参见下文“LimitRanger插件”』
 - NamespaceExists (废弃) 『参见下文“NamespaceExists (废弃) 插件”』
 - NamespaceAutoProvision (废弃) 『参见下文“NamespaceAutoProvision (废弃)插件”』
 - NamespaceLifecycle 『参见下文“NamespaceLifecycle插件”』
 - 是否有推荐的插件集合？『参见下文“是否有推荐的插件集合”』

什么是Admission Controller？

Admission Controller插件是一段代码，其拦截Kubernetes API服务的请求早于对象的持久性，但是在请求的认证和授权之后。插件代码位于API服务进程中，会编译成二进制以便此时使用。

集群在接受一个请求之前，每一个Admission Controller插件都会按序运行。如果这个序列中的某个插件拒绝该请求，则整个的请求都会被立刻拒绝，返回一个错误给用户。

Admission Controller插件在某些情况下也许会改变传进来的对象，配置系统默认值。此外，Admission Controller插件也许会改变请求处理中的部分相关资源去做些事情，比如增量配额的使用。

为什么使用Admission Controller？

Kubernetes中许多高级功能需要激活Admission Controller插件，以便更好的支持该功能。总之，没有正确配置Admission Controller插件的Kubernetes API服务是不完整的服务，很多用户期望的服务是不支持的。

如何接入该插件？

Kubernetes API 服务器提供了一个参数，admission-control，用逗号分隔，在集群中修改对象之前，调用许可控制选项的有序列表。

每个插件的功能是什么？

AlwaysAdmin插件**

使用插件本身处理所有请求。

AlwaysDeny插件**

拒绝所有请求，主要用于测试。

DenyExecOnPrivileged (废弃)插件**

如果一个Pod有一个特权Container，该插件就会拦截所有的请求，在该Pod中执行一个命令。

如果你的集群支持特权Container，而且你想要限制终端用户在那些Container中执行命令的权限，我们强烈建议使用该插件。

该功能已经合并到DenyEscalatingExec插件『参见下文“DenyEscalatingExec插件”』)

DenyEscalatingExec插件**

该插件拒绝执行和附加令到允许主机访问的且有升级特权的Pod。包含含有运行特权的Pod，有访问主机PID Namespace的权限。

如果你的集群支持含有升级特权的Container，而且你想要限制终端用户在这些Container中执行命令的能力，我们强烈建议使用该插件。

ServiceAccount插件**

这个插件实现了[serviceAccounts](#)的自动化。如果你打算使用Kubernetes ServiceAccount对象，我们强烈建议使用该插件。

SecurityContextDeny插件**

[SecurityContext](#)定义了一些不适用于Container的选项，这个插件将会拒绝任何含有该SecurityContext的Pod。

ResourceQuota插件**

该插件会检查传入的请求，确保其不违反任何Namespace中ResourceQuota对象枚举的约束条件。如果你在Kubernetes开发中正在使用ResourceQuota对象，你必须使用该插件实现配额约束条件。

查看[resourceQuota设计文档](#)和[Resource Quota示例](#)了解更多细节。

强烈建议配置该插件在Admission Controller插件的序列中。This is so that quota is not prematurely incremented only for the request to be rejected later in admission control。（该句话翻译有待考虑）

LimitRanger插件**

该插件会检查传入的请求，确保其不违反任何Namespace中LimitRange对象枚举的约束条件。如果你在Kubernetes开发中正在使用LimitRange对象，你必须使用该插件实现约束条件。LimitRange也经常用于Pod中默认资源请求，不会指定哪一个请求。目前，默认LimitRange，在默认的Namespace中对所有的Pod，需要0.1CPU。

查阅[LimitRange设计文档](#)和[用例](#)，了解更多详细信息。

NamespaceExists（废弃）插件**

该插件会检查所有传入的请求，尝试在Kubernetes Namespace中创建资源，如果该Namespace不是当前创建的，该插件会拒绝这个请求。我们强烈建议使用该插件，确保数据的完整性。

Admission Controller的该功能已经并入NamespaceLifecycle插件。

NamespaceAutoProvision（废弃）插件**

该插件会检查所有传入的请求，尝试在Kubernetes Namespace中创建资源，如果Namespace不存在，会创建一个新的Namespace。

我们强烈建议NamespaceExists插件优先级高于NamespaceAutoProvision插件。

NamespaceLifecycle插件**

如果Namespace已经终止，则不能在其中创建新的Namespace，该插件会强制该操作。

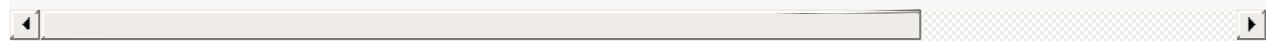
删除一个Namespace会终止一系列操作，移除该Namespace中所有对象（Pod，服务等等）。为了加强该过程的完整性，我们建议使用该插件。

是否有推荐的插件集合？

是的。

Kubernetes 1.0，我们强烈建议使用如下的许可控制插件集合（按顺序排列）：

```
--admission_control=NamespaceLifecycle, NamespaceExists, LimitRanger, SecurityContextDeny, Se
```



Service Accounts集群管理指南

译者：Nancy 校对：无

这是对Service Accounts的集群管理指南，详情[Service Accounts用户指南](#)。

对用户授权和用户账户的支持正在计划中，还没有完全完成。有时候，不完整的一些特性，可以更好的描述Service Accounts。

User Accounts和Service Accounts

基于以下原因，Kubernetes区分了User Accounts和Service Accounts：

- User Accounts针对人，Service Accounts针对运行在Pod的进程；
- User Accounts是全局的，其名字必须在一个集群的所有Namespace中是唯一的。未来的用户资源将不被命令，但是Service Accounts是可以被命名的。
- 通常情况下，集群的User Accounts可以从一个企业数据库同步。在企业数据库中，新建的账户需要特殊权限，而且绑定到复杂业务流程。新建Service Accounts可以更加轻量级，允许集群用户为特殊任务创建Service Accounts（比如，最小权限规则）。
- 对人类和Service Accounts的审核注意事项是不同的。
- 复杂系统的配置包含对系统组件的各种Service Accounts的定义。因为Service Accounts可以创建ad-hoc，可以命名，配置是便携式的。

Service Accounts自动化

三个独立的组件合作实现对Service Accounts的自动化。

- Service账户Admission Controller
- Token控制器
- Service Accounts控制器

Service Accounts Admission Controller

Pod的修改是Admission Controller插件实现的，该插件是apiserver的一部分。插件的创建和更新，会同步修改Pod。当插件状态是active（大多版本中，默认是active），创建或者修改Pod会遵循以下流程：

1. 如果该Pod没有ServiceAccount集，将ServiceAccount设为default
2. ServiceAccount必须有存在的Pod引用，否则拒绝该ServiceAccount

3. 如果该Pod不包含任何ImagePullSecrets，然后该ServiceAccount的ImagesPullSecrets会被加入Pod。
4. 添加一个volume到该Pod，包含API访问的令牌。
5. 添加一个volume到该Pod的每一个容器，挂载在/var/run/secrets/kubernetes.io/serviceaccount

Token Controller

TokenController做为controller-manager的一部分异步运行。

- 检查serviceAccount的创建，并且创建一个关联的Secret，允许API访问。
- 检查serviceAccount的删除，并且删除所有相关ServiceAccountToken Secrets
- 检查额外Secret，确保引用的ServiceAccount的存在，并且如果有必要则添加一个Token到该Secret。
- 检查Secret的删除，如果有必要，从相关的ServiceAccount中移除参考信息。

创建额外的API Token

一个控制循环要确保每一个Service Accounts存在一个API Token。为一个Service Accounts创建一个额外的API Token，类型是ServiceAccountToken，含有一个注释去引用到对应的个Service Accounts。该控制器使用如下的Token去更新：

```
secret.json:
{
  "kind": "Secret",
  "apiVersion": "v1",
  "metadata": {
    "name": "mysecretname",
    "annotations": {
      "kubernetes.io/service-account.name": "myserviceaccount"
    }
  },
  "type": "kubernetes.io/service-account-token"
}

kubectl create -f ./secret.json
kubectl describe secret mysecretname
```

删除、废弃一个个Service Accounts Token

```
kubectl delete secret mysecretname
```

服务账户控制器(**Service Account Controller**) (考虑可以不翻译)

Service Account Controller管理Namespace中的ServiceAccount， 确保每一个“default”的ServiceAccount存在于每一个活动空间中。

Kube调度器

译者：Nancy 校对：无

概要

Kubernetes调度器是一个函数，policy-rich, topology-aware和workload-specific（三个形容词，翻译待考虑），显著影响可用性、性能和容量。该调度器需要考虑个人和集体的资源需求，服务质量需求，硬件、软件、策略约束，亲和力和非亲和力规范，数据局部性，内部工作负载干扰，截止日期等等。特定工作负载需求在必要时会通过API暴露。

kube-scheduler

选项

```
--address=127.0.0.1: 服务的IP地址（所有的接口，设置为0.0.0.0）  
--algorithm-provider="DefaultProvider": 提供的调度算法，DefaultProvider是其中一种。  
--bind-pods-burst=100: 爆发期间每秒允许绑定的调度数量。  
--bind-pods-qps=50: 可以继续工作的每秒允许绑定的调度数量。  
--google-json-key"": 用户认证的Google Cloud Platform Service Account JSON Key。  
--kubeconfig"": 含有授权和主位置信息的Kube配置文件路径。  
--log-flush-frequency=5s: 日志缓冲最大值，单位是秒。  
--master"": Kubernetes API服务器地址（Kube配置文件里都有说明）  
--policy-config-file"": 含有调度策略的配置文件。  
--port=10251: 调度的HTTP协议服务端口。  
--profiling[=true]: 通过Web接口host:port/debug/pprof/，激活Profiling（翻译待考虑）。
```

网络

译者：何炜 校对：无

Kubernetes网络需要解决下面四类不同的问题：

1. 高耦合容器和容器间通信
2. Pod和Pod间通信
3. Pod和Service间通信
4. 外部和内部通信

模型和动机

Kubernetes从默认的Docker网络模型中脱离出来（尽管和Docker 1.8的网络插件已经很接近了）。在一个共享网络命名空间的平面内，目标是一个pod一个IP，每个pod都可以和其它物理机或者容器实现跨网络通信。每个pod一个IP创建了一个简洁的，后向兼容的模型，在这个模型里，无论从哪个角度来看，包括端口分配，组网，命名，服务发现，负载均衡，应用配置和迁移，pod都可以被当作虚拟机或者物理主机。

另一方面，动态端口分配，要求同时支持静态端口（如对外部可接入的服务）和动态分配的端口；需要对中心型分配和本地获取动态端口做区分，复杂的调度（应为端口是稀缺资源）对用户来说是非常不方便的；复杂的应用配置，造成用户会被端口冲突、重用和耗尽所困扰；要求非标准的方式来命名（如consul和etcd而不是DNS）；需要代理或者重定向才能让程序使用标准的命名和寻址技术（如web浏览器）；如果还希望监控整个组成员的变化，并且阻碍容器或pod迁移（使用CRIU），就需要监控和缓存无效的地址和端口变化。其它问题中，NAT分隔地址空间引入了额外的复杂度，打破了自注册机制。

容器到容器

所有容器在一个pod中表现为它们在同一个主机上并且不受网络限制。它们可以通过端口在本地实现互相通信。这提供了简单（已知静态端口），安全（端口绑定在localhost，可以被pod中其他容器发现但不会被外部看到），还有性能提升。这同样减少了物理机获虚拟机上非容器化应用的迁移。人们运行应用都堆砌到统一个主机上，它已经解决了如何让端口不冲突和已经安排了如何让客户端去找到它。

这个方法确实减少了一个pod中容器的隔离性（端口可能冲突），并且可以没有容器私有化端口，但这些看起来都是未来才会面对的相对比较小的问题。另外，本地化pod是容器在pod中共享同样的资源（如volumes，cpu，内存等），所以希望并且可以容忍隔离性的减少。通常情况，用户可以控制哪些容器属于同一个pod，而不能控制哪些pod一起运行在一个主机上。

Pod到Pod

因为每个pod都有一个“真”（非机器私有的）的IP地址，pods间可以相互通信，不依赖于代理和转换。Pod可以使用一个常见的端口号，还可以防止更高层次的服务发现系统像DNS-SD，Consul或者Etcd。

当任何容器调用ioctl (SIOCGIFADDR) (获取接口的地址)，可以看到相同的IP，任何同级别的容器都可以看到这个IP，就是说每一个pod有自己的IP地址，并且这个IP地址其他pods也可以知道。在容器内外通过生成相同IP地址和端口，我们创造了一个非NAT模式，扁平化的地址空间。运行 `ip addr show` 应该可以看到期待的值。这会让所有现在的命名或发现机制到盒子外面去实现，包括自注册机制和应用分发IP地址。我们应该对pod间网络通信持乐观态度。在一个pod内，容器间更像是通过volumes (如tmpfs) 或者IPC来通信。

这种方式和标准的Docker模式有所不同。在Docker模式下，每一个容器有一个IP，IP是在172.x.x.x的网段内，并且只能从SIOCGIFADDR看到172.x.x.x的地址。如果这些容器连接了另一个容器，那同级别的容器会看到这个连接来自于一个不同的IP而不是容器自己知道的IP。简而言之，永远不能自注册同一个容器内的任何东西，因为一个容器不能被自己私有的IP获取到。

另一个解决方案我们考虑增加一个寻址层：每个容器一个中心化pod的IP。每个容器都有自己本地的IP地址，这个IP只在pod内可见。这种方案可能会让在物理机或虚拟机上的容器化应用移动到pod中更容易，但会使实现变得更复杂（如需要每个pod一个桥，水平分割的DNS）。造成额外的地址转换，而且会破坏自注册和IP分配机制。

像Docker，端口仍可以向主机节点的接口公开，但这样的需求应该从根本上减少。

实现

For the Google Compute Engine cluster configuration scripts, we use advanced routing rules and ip-forwarding-enabled VMs so that each VM has an extra 256 IP addresses that get routed to it. This is in addition to the 'main' IP address assigned to the VM that is NAT-ed for Internet access. The container bridge (called cbr0 to differentiate it from docker0) is set up outside of Docker proper.

例如，GCE的高级路由规则：

```
gcloud compute routes add "${MINION_NAMES[$i]}" \
--project "${PROJECT}" \
--destination-range "${MINION_IP_RANGES[$i]}" \
--network "${NETWORK}" \
--next-hop-instance "${MINION_NAMES[$i]}" \
--next-hop-instance-zone "${ZONE}" &
```

GCE itself does not know anything about these IPs, though. This means that when a pod tries to egress beyond GCE's project the packets must be SNAT'ed (masqueraded) to the VM's IP, which GCE recognizes and allows.

其他实现

With the primary aim of providing IP-per-pod-model, other implementations exist to serve the purpose outside of GCE.

OpenVSwitch with GRE/VxLAN Flannel L2 networks ("With Linux Bridge devices" section)

Weave is yet another way to build an overlay network, primarily aiming at Docker integration.

Calico uses BGP to enable real container IPs.

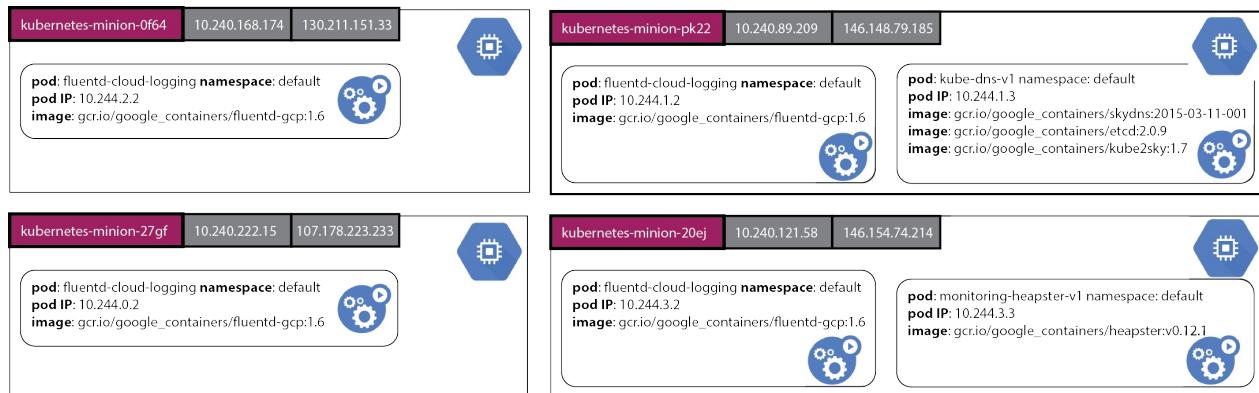
从集群级别日志到Google云日志平台

译者：White 校对：无

通常一个忙碌的Kubernetes集群运行着很多系统和应用级别的pods。那么系统管理员如何才能收集，管理和查看这些系统pods的日志呢？那么用户应该如何查询这些由pods组成的应用日志呢？这些pods可能会被重新启动或者是自动的被Kubernetes系统创建出来。这些问题可以通过Kubernetes集群日志服务来解决。

集群级别的日志服务能够让我们收集来自于运行在pod中的容器镜像或者是pod，甚至是整个集群的日志，这些日志持久性记录了超出他们运行生命周期的全部状态。这篇文章中，假设我们已经创建了Kubernetes集群，并且支持向谷歌云日志服务发送集群级别日志。集群一旦创建，你会拥有一个系统pods的集合，这些pods运行在 `kube-system` 命名空间中，它们支持收集监控、日志记录以及Kubernetes服务的域名名称解析信息。

```
$ kubectl get pods --namespace=kube-system
NAME                               READY   REASON   RESTARTS   AGE
fluentd-cloud-logging-kubernetes-minion-0f64   1/1     Running   0          32m
fluentd-cloud-logging-kubernetes-minion-27gf    1/1     Running   0          32m
fluentd-cloud-logging-kubernetes-minion-pk22    1/1     Running   0          31m
fluentd-cloud-logging-kubernetes-minion-20ej    1/1     Running   0          31m
kube-dns-v3-pk22                      3/3     Running   0          32m
monitoring-heapster-v1-20ej           0/1     Running   9          32m
```



图片显示GCE上创建了4个节点，紫色背景显示的是每个节点的虚拟节点名称。灰色方块中显示的分别是内网和外网IP地址，绿色方块显示的是运行在节点中的pods。每一个方框中显示了pod的名称，运行的命名空间，ip地址以及执行的镜像名称。我们可以看到每个节点都运行一个叫做fluentd-cloud-logging的pod,用来收集节点容器的运行日志并且发送到Google云日志服务。其中一个节点上运行着一个[集群域名解析](#)的pod,另外一个节点上运行中提供监控服务的pod。

为了帮助我们解释集群级别日志服务是如何工作的，让我们从一个合成日志产生器开始，这个pod的定义[counter-pod.yaml](#):

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
  namespace: default
spec:
  containers:
  - name: count
    image: ubuntu:14.04
    args: [bash, -c,
           'for ((i = 0; ; i++)); do echo "$i: $(date)"; sleep 1; done']
```

这个pod规范定义了一个容器，这个容器一旦创建就会运行一个bash脚本。这个脚本简单的输出一个计数器的值，每秒运行一次，无限运行下去。我们在默认的命名空间中创建这个pod。

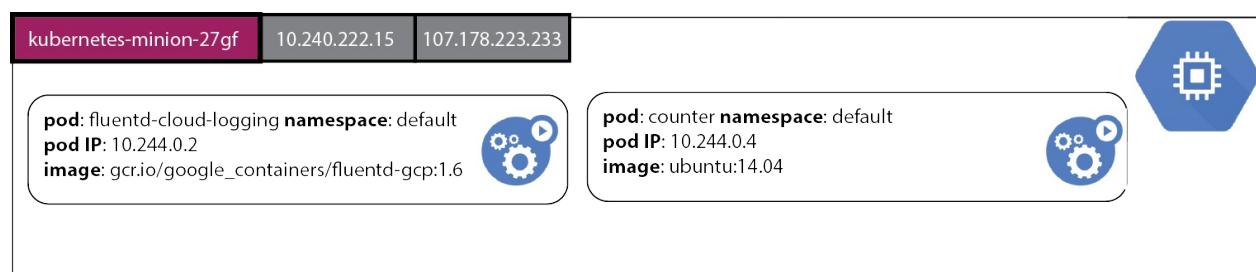
```
$ kubectl create -f examples/blog-logging/counter-pod.yaml
pods/counter
```

我们可以查看这个运行的pod:

NAME	READY	STATUS	RESTARTS	AGE
counter	1/1	Running	0	5m

这一步可能需要几分钟来下载Ubuntu:14.04图像，pod的状态显示为 Pending。

计数器pod运行在其中一个节点上：



当pod状态变成 Running 后，我们可以用kubectl logs命令来查看计数器pod的输出。

```
$ kubectl logs counter
0: Tue Jun  2 21:37:31 UTC 2015
1: Tue Jun  2 21:37:32 UTC 2015
2: Tue Jun  2 21:37:33 UTC 2015
3: Tue Jun  2 21:37:34 UTC 2015
4: Tue Jun  2 21:37:35 UTC 2015
5: Tue Jun  2 21:37:36 UTC 2015
...
```

这条命令用于从一个运行在容器中镜像的Docker日志文件中读取文本日志。我们可以连到这个运行的容器，观察计数器bash脚本运行情况。

```
$ kubectl exec -i counter bash
ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        1  0.0  0.0  17976  2888 ?        Ss   00:02  0:00 bash -c for ((i = 0; ; i
root      468  0.0  0.0  17968  2904 ?        Ss   00:05  0:00 bash
root      479  0.0  0.0   4348   812 ?        S    00:05  0:00 sleep 1
root      480  0.0  0.0  15572  2212 ?        R    00:05  0:00 ps aux
```

如果任何原因导致运行在pod中的镜像被杀掉了，然后又被Kubernetes重新启动，那么会发生什么呢？我们仍然可以看到，重新启动的容器日志会接着之前调用容器输出的日志行后面输出吗？或者是原来容器的执行日志会丢失，只能看到新容器的日志行输出吗？我们来看一看，首先停止当前正在运行的计数器。

```
$ kubectl stop pod counter
pods/counter
```

现在我们重启这个计数器。

```
$ kubectl create -f examples/blog-logging/counter-pod.yaml
pods/counter
```

等容器重新启动后重新获取日志。

```
$ kubectl logs counter
0: Tue Jun  2 21:51:40 UTC 2015
1: Tue Jun  2 21:51:41 UTC 2015
2: Tue Jun  2 21:51:42 UTC 2015
3: Tue Jun  2 21:51:43 UTC 2015
4: Tue Jun  2 21:51:44 UTC 2015
5: Tue Jun  2 21:51:45 UTC 2015
6: Tue Jun  2 21:51:46 UTC 2015
7: Tue Jun  2 21:51:47 UTC 2015
8: Tue Jun  2 21:51:48 UTC 2015
```

这个pod容器中的首次调用日志丢失了！理想情况是，我们会运行在每个pod中的每一个容器的每一次调用日志行。此外，即使pod重新启动，我们仍然希望保留所有的pod中容器发出的所有日志行。但是不用担心，这仅仅是Kubernetes提供的集群级别的日志功能。当集群创建后，每一个容器的标准输出和标准错误输出都可以被运行在每个节点中的Fluentd打到[Google云日志平台](#)或者是打到Elasticsearch中用Kibana来查看。

当Kubernetes集群创建后开启了记录到Google云日志平台的功能，系统会在集群每个节点上创建一个叫做 fluentd-cloud-logging 的pod来收集Docker容器日志。在这篇博客一开始我们就可以看到如何获取这些pods的命令。

这个日志收集的pod规范文件，看起来像这样[fluentd-gcp.yaml](#):

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
spec:
  containers:
    - name: fluentd-cloud-logging
      image: gcr.io/google_containers/fluentd-gcp:1.11
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
      env:
        - name: FLUENTD_ARGS
          value: -qq
      volumeMounts:
        - name: containers
          mountPath: /var/lib/docker/containers
  volumes:
    - name: containers
      hostPath:
        path: /var/lib/docker/containers
```

这个pod的规范将包含Docker日志文件路径，/var/lib/docker/containers，映射到容器中的相同路径。这个pod运行一个叫做gcr.io/google_containers/fluentd-gcp:1.6的镜像，配置为从日志文件路径收集Docker日志，打到Google云日志平台。集群中的每个节点都会运行这个pod实例。如果这个pod失败退出，Kubernetes会自动重启它。

我们可以在Google开发者控制台的监控页面中点击日志项，选择叫做kubernetes.counter_default_count的日志统计项。这个日志收集器标识了这个pod(计数器)，命名空间（默认）以及容器的名称（计数）。通过名称，我们可以通过名称从下拉菜单中选择计数器容器的日志。

The screenshot shows the Google Cloud Platform Logs interface. The top navigation bar has 'Logs' selected. Below it is a search bar labeled 'Filter by label or text search'. The main area displays log entries for a specific pod. The pod name is 'kubernetes.counter_default_count'. The log entries are timestamped and show various log types: 'activity_log', 'kubelet', and logs from the fluentd container. The fluentd logs are in JSON format, showing log entries like 'log': "37: Wed Jun 10 21:50:10 UTC 2015\n", "stream": "stdout", "time": "2015-06-10T21:50:10.7131...".

在开发者控制台中我们可以看到所有容器调用的日志。

The screenshot shows the Google Cloud Platform Logs interface. The top navigation bar has 'Logs' selected. Below it is a search bar labeled 'Filter by label or text search'. The main area displays log entries for a specific pod. The pod name is 'kubernetes.counter_default_count'. The log entries are timestamped and show various log types: 'activity_log', 'kubelet', and logs from the fluentd container. The fluentd logs are in JSON format, showing log entries like 'log': "106: Tue Jun 2 21:51:37 UTC 2015\n", "stream": "stdout", "time": "2015-06-02T21:51:37.000000000Z...".

注意到第一个容器计数到108就被终止了。当下一个容器启动后，计数进程复位到0。类似的，如果我们删除这个pod并重启它，当pod运行时，我们能够捕获到运行在这个pod中所有容器实例的日志。

导入到Google云日志平台中的日志可能会被导出到不同的系统中，包括[Google云存储](#)和[BigQuery](#)。使用云日志控制台中的导出选项页来指定日志导出到哪里。你还可以按照此链接去往[设置选项卡](#)。

我们可以通过SQL语句从BigQuery中查询导入的计数日志行，最新的日志行先显示。

```
SELECT metadata.timestamp, structPayload.log
FROM [mylogs.kubernetes_counter_default_count_20150611]
ORDER BY metadata.timestamp DESC
```

下面是一些示例输出：

New Query

```
1 SELECT metadata.timestamp, structPayload.log FROM [music.kubernetes_counter_default_count_20150611] order by metadata.timestamp desc
```

RUN QUERY Save Query Save View Format Query Show Options Query complete (0.9s elapsed, 656 KB processed) ✓

Query Results 3:38pm, 11 Jun 2015

Row	metadata_timestamp	structPayload_log
1	2015-06-11 22:38:44 UTC	123: Thu Jun 11 22:38:44 UTC 2015
2	2015-06-11 22:38:43 UTC	122: Thu Jun 11 22:38:43 UTC 2015
3	2015-06-11 22:38:42 UTC	121: Thu Jun 11 22:38:42 UTC 2015
4	2015-06-11 22:38:41 UTC	120: Thu Jun 11 22:38:41 UTC 2015
5	2015-06-11 22:38:40 UTC	119: Thu Jun 11 22:38:40 UTC 2015
6	2015-06-11 22:38:39 UTC	118: Thu Jun 11 22:38:39 UTC 2015

Download as CSV Save as Table Chart View

我们还可以从Google云存储中把这些日志拉到desktop或者laptop上来本地化搜索。下面的命令可以从一个叫作 myproject 的Computer Engine项目中获取集群中运行的pod计数器日志。只搜了2015-06-11的日志。

```
$ gsutil -m cp -r gs://myproject/kubernetes.counter_default_count/2015/06/11 .
```

现在我们在导入的日志中执行查询。下面的示例用jq程序来抽取日志行。

```
$ cat 21\:00\:00_21\:59\:59_S0.json | jq '.structPayload.log'
"0: Thu Jun 11 21:39:38 UTC 2015\n"
"1: Thu Jun 11 21:39:39 UTC 2015\n"
"2: Thu Jun 11 21:39:40 UTC 2015\n"
"3: Thu Jun 11 21:39:41 UTC 2015\n"
"4: Thu Jun 11 21:39:42 UTC 2015\n"
"5: Thu Jun 11 21:39:43 UTC 2015\n"
"6: Thu Jun 11 21:39:44 UTC 2015\n"
"7: Thu Jun 11 21:39:45 UTC 2015\n"
...
```

这个页面已经简要介绍了如何在一个Kubernetes部署集群中收集日志的底层机制。这个方法仅仅适用于收集运行在pod中的容器的标准输出和标准错误输出日志。要想收集存储在文件中的日志，参照这篇文件中的描述[使用Fluentd在容器中收集日志文件](#)，可以使用一个sidecar容器来收集需要的文件，并且发送到Google云日志服务。

本节中的一些材料也出现在这篇博客文章中[Cluster Level Logging with Kubernetes](#)。

使用Kubernetes在云上原生部署cassandra

译者：卢文泉 校对：无

下面的文档描述了在Kubernetes上部署*cloud native Cassandra*的开发过程。当我们讲*cloud native*时，我们是指这样一款应用，它运行在集群管理器中，并利用集群管理器的基础设施来实现应用的功能。特别的是，在本章介绍的实例中，一个定制好的Cassandra SeedProvider*可以确保Cassandra能够动态发现新的Cassandra节点并把它添加到集群中。

这篇文档也在努力描述Kubernetes中的核心概念：*Pods, Services和Replication Controllers*。

预备知识

下面的例子假设你已经安装和运行一个Kubernetes集群，并且你也下载好kubectl命令行工具以及配置在你的path中。请先阅读[入门指南](#)获取在你平台上安装Kubernetes的指令。

本文中的例子还需要一些代码和配置文件。为了避免重复输入（代码和配置文件），你可以使用git clone命令从Kubernetes仓库中克隆文件到本地计算机。

对于不耐心的友善提醒

这是一个有点长的教程，如果你想直接跳到“马上操作”的命令，可以到文末看[这些](#)

简单的单个Pod Cassandra节点

在Kubernetes中，应用的原子单元（【译者注】原子单元指不可再分的最小单元）是Pod。一个Pod指必须被部署在同一个主机的一个或者多个容器。同一个pod中的所有容器共享一个网络名称空间，以及可选地共享挂载卷。在这个简单的例子中，我们在pod中定义一个运行Cassandra的容器：

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  containers:
    - args:
        - /run.sh
      resources:
        limits:
          cpu: "0.5"
      image: gcr.io/google_containers/cassandra:v5
      name: cassandra
      ports:
        - name: cql
          containerPort: 9042
        - name: thrift
          containerPort: 9160
      volumeMounts:
        - name: data
          mountPath: /cassandra_data
    env:
      - name: MAX_HEAP_SIZE
        value: 512M
      - name: HEAP_NEWSIZE
        value: 100M
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
  volumes:
    - name: data

```

上面的描述中有几点需要注意。首先我们运行的镜像是`kubernetes/cassandra`。这是一个标准的安装在Debian上的Cassandra镜像。但是它还给Cassandra添加一个定制的[SeedProvider](#)，在Cassandra中，`SeedProvider`设置一个gossip协议用来发现其它Cassandra节点。**KubernetesSeedProvider**使用内置的Kubernetes发现服务找到KubernetesAPI服务器，然后利用Kubernetes API发现新的节点。（稍后将详细介绍）

也许你注意到（在上面的脚本中）我们设置了一些Cassandra参数（`MAX_HEAP_SIZE` 和 `HEAP_NEWSIZE`）以及添加了关于[名称空间](#)的信息。我们还告诉Kubernetes容器暴露了CQL和Thrift的API端口。最后，我们告诉集群管理器我们需要0.5个CPU（0.5个内核core）

理论上会立刻创建一个Cassandra pod，但是在Cassandra部署过程中，**KubernetesSeedProvider**需要知道现在哪个节点上创建服务（【译者注】这会消耗一些时间）

Cassandra服务

在Kubernetes中一个[服务](#)是指执行相同任务的一系列pods集合（【译者注】一个任务可能由多个pod协同完成）。例如，Cassandra集群中的pods集合可以是Kubernetes服务，甚至仅仅是一个pod也可以。服务的一个重要用途是在pods成员间分发流量实现负载均衡。服务另外一个重要用途也可以用作长期查询，查询调用Kubernetes API使得动态改变pod（或者是上面创建的一个pod）成为可能。这就是我们最初使用Cassandra服务的两种方式。

下面是服务的描述：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  ports:
    - port: 9042
  selector:
    name: cassandra
```

这里对于节点最重要的是选择器，（选择器）是建立在标签上的查询，标签则标记了服务中包含的pods集合。在这个例子中选择器是`name=cassandra`。如果你现在回头看Pod的描述，你会发现pod有和它对应的标签。所以这个pod会被选中成为服务中的成员。

通过下面命令创建服务：

```
$ kubectl create -f examples/cassandra/cassandra-service.yaml
```

现在服务运行起来了，我们可以使用前面提到的说明来创建第一个Cassandra。

```
$ kubectl create -f examples/cassandra/cassandra.yaml
```

不久，你会看到pod也运行了，以及pod中运行的单个容器：

```
$ kubectl get pods cassandra
NAME      READY     STATUS    RESTARTS   AGE
cassandra 1/1      Running   0          55s
```

你也可以查询服务的端点来检查pod是否被正确选中：

```
$ kubectl get endpoints cassandra -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2015-06-21T22:34:12Z
  labels:
    name: cassandra
    name: cassandra
  namespace: default
  resourceVersion: "944373"
  selfLink: /api/v1/namespaces/default/endpoints/cassandra
  uid: a3d6c25f-1865-11e5-a34e-42010af01bcc
subsets:
- addresses:
  - ip: 10.244.3.15
    targetRef:
      kind: Pod
      name: cassandra
      namespace: default
      resourceVersion: "944372"
      uid: 9ef9895d-1865-11e5-a34e-42010af01bcc
  ports:
- port: 9042
  protocol: TCP
```

添加复制节点

当然，单节点集群并不特别有趣。Kubernetes和Cassandra的真正能量在于可以很方便创建一个副本，使得Cassandra集群可扩展。

在Kubernetes中一个[复制控制器](#)负责复制相同的pods。复制控制器的作用和服务的相同点在于复制控制器有选择器能查询集合中的指定成员。与服务不同的是复制控制器有预期数量的副本，所以它能创建或删除Pods确保pod数量达到预期状态。

复制控制器会“接受”已经存在并且符合查询条件的pods，现在我们会创建一个只有一个副本的复制控制器，用它来接收已经存在的Cassandra pod：

```

apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  replicas: 1
  selector:
    name: cassandra
  template:
    metadata:
      labels:
        name: cassandra
    spec:
      containers:
        - command:
          - /run.sh
        resources:
          limits:
            cpu: 0.5
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        image: gcr.io/google_containers/cassandra:v5
        name: cassandra
        ports:
          - containerPort: 9042
            name: cql
          - containerPort: 9160
            name: thrift
        volumeMounts:
          - mountPath: /cassandra_data
            name: data
      volumes:
        - name: data
          emptyDir: {}

```

大部分复制控制器的定义和上面的Cassandra pod的定义类似，定义简单地为复制控制器提供一个模板，创建新的Cassandra pod时可以很方便拿来使用。不同的部分是选择器属性包含控制器的选择器查询以及副本属性中的副本的预期数量，在这个例子中是1。

创建控制器的指令：

```
$ kubectl create -f examples/cassandra/cassandra-controller.yaml
```

现在事实上已经不那么无趣了，因为之前没做新的事情。现在它会变得有趣。

现在我们把集群数量扩展到2：

```
$ kubectl scale rc cassandra --replicas=2
```

现在如果你列出集群中的pods，查询符合标签**name=cassandra**的pods，你会看到两个cassandra pod：

```
$ kubectl get pods -l="name=cassandra"
NAME          READY   STATUS    RESTARTS   AGE
cassandra     1/1     Running   0          3m
cassandra-af6h5 1/1     Running   0          28s
```

注意其中一个pod的名字是人们可读的**cassandra**，它是我们在配置文件中指定的。另一个是复制控制器随机生成的字符串。

为了证明所有的pods都正常工作，你可以使用**nodetool**命令检查集群的状态。使用**kubectl exec**命令在Cassandra Pod中运行**nodetool**。

```
$ kubectl exec -ti cassandra -- nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load      Tokens  Owns (effective)  Host ID
UN  10.244.0.5  74.09 KB    256      100.0%          86fed0f-f070-4a5b-bda1-2eeb0ad08b77
UN  10.244.3.3  51.28 KB    256      100.0%          dafe3154-1d67-42e1-ac1d-78e7e80dce2b
```

现在扩展集群到4个节点：

```
kubectl scale rc cassandra --replicas=4
```

一会后再次检查状态：

```
$ kubectl exec -ti cassandra -- nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
--  Address      Load      Tokens  Owns (effective)  Host ID
UN  10.244.2.3  57.61 KB   256     49.1%           9d560d8e-dafb-4a88-8e2f-f554379c21c3
UN  10.244.1.7  41.1  KB   256     50.2%           68b8cc9c-2b76-44a4-b033-31402a77b839
UN  10.244.0.5  74.09 KB   256     49.7%           86feda0f-f070-4a5b-bda1-2eeb0ad08b77
UN  10.244.3.3  51.28 KB   256     51.0%           dafe3154-1d67-42e1-ac1d-78e7e80dce2b
```

tl; dr;

下面是给那些不耐心的小伙伴的，下面是以上命令的总结：

```
# create a service to track all cassandra nodes
kubectl create -f examples/cassandra/cassandra-service.yaml

# create a single cassandra node
kubectl create -f examples/cassandra/cassandra.yaml

# create a replication controller to replicate cassandra nodes
kubectl create -f examples/cassandra/cassandra-controller.yaml

# scale up to 2 nodes
kubectl scale rc cassandra --replicas=2

# validate the cluster
kubectl exec -ti cassandra -- nodetool status

# scale up to 4 nodes
kubectl scale rc cassandra --replicas=4
```

Seed Provider Source

[看这里](#)

原文地址

[原文档地址](#)

Spark例子

译者：卢文泉 校对：无

在下面的例子中，你会学习使用Kubernetes和Docker创建一个能够使用的阿帕奇Spark集群。我们使用Spark的[单例模式](#)创建一个Spark master节点服务和一系列Spark workers节点。

对于不耐心的专家，直接跳到[这个部分](#)

资源

Docker镜像很重，基于<https://github.com/mattf/docker-spark>

步骤0:预备知识

下面的例子假设你已经安装和运行一个Kubernetes集群，并且你也下载好[kubectl命令行工具](#)以及配置在你的path中。请先阅读[入门指南](#)获取在你平台上安装Kubernetes的指令。

步骤一：启动Master服务

Master服务是一个Spark集群的主服务（或头服务）。

使用[examples/spark/spark-master.json](#)文件创建运行在Master服务中的pod。

```
$ kubectl create -f examples/spark/spark-master.json
```

然后使用[examples/spark/spark-master-service.json](#)文件创建一个逻辑服务端点供Spark workers节点使用连接Matser pod。

```
$ kubectl create -f examples/spark/spark-master-service.json
```

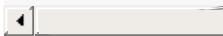
检查Master节点使用运行并且能够连接

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
[...]
spark-master                        1/1     Running   0          25s
```

检查日志查看master节点的状态：

```
$ kubectl logs spark-master

starting org.apache.spark.deploy.master.Master, logging to /opt/spark-1.4.0-bin-hadoop2.6
Spark Command: /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java -cp /opt/spark-1.4.0-bin-ha
=====
15/06/26 14:01:49 INFO Master: Registered signal handlers for [TERM, HUP, INT]
15/06/26 14:01:50 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl
15/06/26 14:01:51 INFO SecurityManager: Changing view acls to: root
15/06/26 14:01:51 INFO SecurityManager: Changing modify acls to: root
15/06/26 14:01:51 INFO SecurityManager: SecurityManager: authentication disabled; ui acls
15/06/26 14:01:51 INFO Slf4jLogger: Slf4jLogger started
15/06/26 14:01:51 INFO Remoting: Starting remoting
15/06/26 14:01:52 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sp
15/06/26 14:01:52 INFO Utils: Successfully started service 'sparkMaster' on port 7077.
15/06/26 14:01:52 INFO Utils: Successfully started service on port 6066.
15/06/26 14:01:52 INFO StandaloneRestServer: Started REST server for submitting applicati
15/06/26 14:01:52 INFO Master: Starting Spark master at spark://spark-master:7077
15/06/26 14:01:52 INFO Master: Running Spark version 1.4.0
15/06/26 14:01:52 INFO Utils: Successfully started service 'MasterUI' on port 8080.
15/06/26 14:01:52 INFO MasterWebUI: Started MasterWebUI at http://10.244.2.34:8080
15/06/26 14:01:53 INFO Master: I have been elected leader! New state: ALIVE
```



步骤二：启动**Spark workers**

在Spark集群中Spark workers做繁重的工作。它们为你的程序提供计算资源和数据缓存的能力。

Spark workers需要Master服务支持才能运行。

使用[examples/spark/spark-worker-controller.json](#)文件创建复制控制器管理workers pod。

```
$ kubectl create -f examples/spark/spark-worker-controller.json
```

检查**workers**节点是否运行

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
[...]
spark-master                         1/1     Running   0          14m
spark-worker-controller-hifwi        1/1     Running   0          33s
spark-worker-controller-u40r2         1/1     Running   0          33s
spark-worker-controller-vpgyg        1/1     Running   0          33s

$ kubectl logs spark-master
[...]
15/06/26 14:15:43 INFO Master: Registering worker 10.244.2.35:46199 with 1 cores, 2.6 GB
15/06/26 14:15:55 INFO Master: Registering worker 10.244.1.15:44839 with 1 cores, 2.6 GB
15/06/26 14:15:55 INFO Master: Registering worker 10.244.0.19:60970 with 1 cores, 2.6 GB
```

步骤三：使用集群做点什么

获取Master服务的地址和端口。

```
$ kubectl get service spark-master
NAME      LABELS           SELECTOR           IP(S)      PORT(S)
spark-master   name=spark-master   name=spark-master   10.0.204.187   7077/TCP
```

使用SSH连接集群中的一个节点，在GCE/GKE（【译者注】指云服务谷歌计算引擎和谷歌Kubernetes引擎）上，你可以使用[开发者控制台](#)（更多细节点击[这里](#)）或者运行[gcloud compute ssh](#)，name可以通过[kubectl get nodes](#)命令获得（更多细节[在这](#)）。

```
$ kubectl get nodes
NAME           LABELS           STATUS
kubernetes-minion-5jvu   kubernetes.io/hostname=kubernetes-minion-5jvu   Ready
kubernetes-minion-6fbi   kubernetes.io/hostname=kubernetes-minion-6fbi   Ready
kubernetes-minion-8y2v   kubernetes.io/hostname=kubernetes-minion-8y2v   Ready
kubernetes-minion-h0tr   kubernetes.io/hostname=kubernetes-minion-h0tr   Ready

$ gcloud compute ssh kubernetes-minion-5jvu --zone=us-central1-b
Linux kubernetes-minion-5jvu 3.16.0-0.bpo.4-amd64 #1 SMP Debian 3.16.7-ckt9-3~deb8u1~bpo7

==== GCE Kubernetes node setup complete ===

me@kubernetes-minion-5jvu:~$
```

一旦登陆成功就可以使用Spark基础镜像了。在镜像中有一个脚本用来设置基于Master的IP和端口环境。

结果

现在你已经为你的Spark master节点和Spark workers节点创建了服务、复制控制器和pods。你可以在下一篇文档中继续使用这个例子以及使用这个Spark集群。点击[Spark文档](#)获取更多信息。

tl;dr

```
kubectl create -f spark-master.json  
  
kubectl create -f spark-master-service.json  
  
Make sure the Master Pod is running (use: kubectl get pods).  
  
kubectl create -f spark-worker-controller.json
```

原文连接

Storm 示例

译者：White 校对：无

接下来的例子中，你将会使用Kubernetes和Docker来创建一个多功能的Apache Storm集群。

你将会设置一个Apache ZooKeeper服务，一个Storm master服务（又名Nimbus主机），以及一个Storm工作者集合（又名监管者）。

如果你已经熟悉这个部分，请直接跳到tl;dr章节。

资源

可以自由获取这些资源文件：

- Docker镜像 - <https://github.com/mattf/docker-storm>
- Docker受信的构建文件 - <https://registry.hub.docker.com/search?q=mattf/storm>

步骤零：前期准备

这个示例假设你已经安装运行了一个Kubernetes集群，环境路径下已经安装了kubectl命令行工具。请查看不同平台的安装说明[开始](#)。

步骤一：启动ZooKeeper服务

ZooKeeper是一个分布式协调者服务，Storm使用它来作为引导程序和存储运行转态数据。

使用这个[examples/storm/zookeeper.json](#)文件来创建一个运行ZooKeeper服务的pod。

```
$ kubectl create -f examples/storm/zookeeper.json
```

然后使用[examples/storm/zookeeper-service.json](#)文件创建一个逻辑服务终端节点用来给Storm访问ZooKeeper pod。

```
$ kubectl create -f examples/storm/zookeeper-service.json
```

在这之前，你需要确保ZooKeeper pod处于运行态并且可以被访问。

查看ZooKeeper是否运行

```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
zookeeper  1/1     Running   0          43s
```

查看ZooKeeper是否可以访问

```
$ kubectl get services
NAME           LABELS
kubernetes     component=apiserver,provider=kubernetes
zookeeper      name=zookeeper
                selector
                <none>
                name=zookeeper
                ip(s)
                10.254.
                10.254.

$ echo ruok | nc 10.254.139.141 2181; echo
imok
```

步骤二：启动Nimbus服务

Nimbus服务是Storm集群的主节点(或者首要)服务。Nimbus依赖于多种功能的ZooKeeper服务。

使用[examples/storm/storm-nimbus.json](#)文件创建一个运行Nimbus服务的pod。

```
$ kubectl create -f examples/storm/storm-nimbus.json
```

然后使用[examples/storm/storm-nimbus-service.json](#)文件创建一个逻辑服务终端节点用来给Storm工作者访问Nimbus pod。

```
$ kubectl create -f examples/storm/storm-nimbus-service.json
```

确保Nimbus服务运行正常。

查看Nimbus节点是否运行以及可以访问

```
$ kubectl get services
NAME           LABELS
kubernetes     component=apiserver,provider=kubernetes
zookeeper      name=zookeeper
nimbus         name=nimbus
SELECTOR
<none>
name=zookeeper
name=nimbus
IP(S)
10.254.
10.254.
10.254.

$ sudo docker run -it -w /opt/apache-storm mattf/storm-base sh -c '/configure.sh 10.254.1
...
No topologies running.
```

步骤三：启动Storm工作者

在Storm集群中，Storm工作者（或者监督者）用来完成繁重的工作。Nimbus服务管理这些运行流处理拓扑应用的工人。

Storm工作者需要保证ZooKeeper和Nimbus服务处于运行态。

使用[examples/storm/storm-worker-controller.json](#)文件来创建副本控制器来管理工作者pods。

```
$ kubectl create -f examples/storm/storm-worker-controller.json
```

查看工作者们是否在运行

一种查看工作者信息的方式是，通过ZooKeeper服务查看有多少客户端在运行。

```
$ echo stat | nc 10.254.139.141 2181; echo
Zookeeper version: 3.4.6--1, built on 10/23/2014 14:18 GMT
Clients:
/192.168.48.0:44187[0](queued=0,recved=1,sent=0)
/192.168.45.0:39568[1](queued=0,recved=14072,sent=14072)
/192.168.86.1:57591[1](queued=0,recved=34,sent=34)
/192.168.8.0:50375[1](queued=0,recved=34,sent=34)
/192.168.45.0:39576[1](queued=0,recved=34,sent=34)

Latency min/avg/max: 0/2/2570
Received: 23199
Sent: 23198
Connections: 5
Outstanding: 0
Zxid: 0xa39
Mode: standalone
Node count: 13
```

Nimbus服务和每个工作者都对应着一个客户端。理想情况下，应该可以在副本控制器创建前后从ZooKeeper获取 stat 输出。

(欢迎提交pull requests使用不同的方式来验证workers)

tl;dr

```
kubectl create -f zookeeper.json
```

```
kubectl create -f zookeeper-service.json
```

确保ZooKeeper Pod正在运行（使用：`kubectl get pods`）。

```
kubectl create -f storm-nimbus.json
```

```
kubectl create -f storm-nimbus-service.json
```

确保Nimbus Pod正在运行。

```
kubectl create -f storm-worker-controller.json
```

示例：分布式任务队列 **Celery, RabbitMQ和Flower**

译者：White 校对：无

介绍

Celery是基于分布式消息传递的异步任务队列。它可以用来创建一些执行单元（例如，一个任务），这些任务可以同步，异步的在一个或者多个工作节点执行。

Celery基于Python实现。

因为**Celery**基于消息传递，需要一些叫作消息代理的中间件（他们用来在发送者和接受者之间处理传递的消息）。**RabbitMQ**是一种和**Celery**联合使用的消息中间件。

下面的示例将向你展示，如何使用Kubernetes来建立一个基于**Celery**作为任务队列，**RabbitMQ**作为消息代理的分布式任务队列系统。同时，还要展示如何建立一个基于**Flower**的任务监控前端。

目标

在例子的最后，我们可以看到：

- 3个pods:
 - 一个**Celery**任务队列
 - 一个**RabbitMQ**消息代理
 - **Flower**前端
- 一个提供访问消息代理的服务
- 可以传递给工作节点的级别**Celery**任务

先决条件

你应该已经拥有一个Kubernetes集群。要完成大部分的例子，确保Kubernetes创建一个以上的节点（例如，通过设置 `NUM_MINIONS` 环境变量为2或者更多）。

第一步：启动**RabbitMQ**服务

Celery任务队列需要连接到RabbitMQ代理。RabbitMQ队列最终会出现在一个独立的pod上，但是，由于pod是短暂存在的，需要一个服务来透明的路由请求到RabbitMQ。

使用这个文件 `examples/celery-rabbitmq/rabbitmq-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: rabbitmq
    name: rabbitmq-service
spec:
  ports:
  - port: 5672
  selector:
    app: taskQueue
    component: rabbitmq
```

这样运行一个服务：

```
$ kubectl create -f examples/celery-rabbitmq/rabbitmq-service.yaml
```

这个服务允许其他pods连接到rabbitmq。对于它们可以使用5672端口，服务也会将流量路由到容器（也通过5672端口）。

第二步：启动**RabbitMQ**

RabbitMQ代理可以通过这个文件启动 `examples/celery-rabbitmq/rabbitmq-controller.yaml`：

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: rabbitmq
    name: rabbitmq-controller
spec:
  replicas: 1
  selector:
    component: rabbitmq
  template:
    metadata:
      labels:
        app: taskQueue
        component: rabbitmq
    spec:
      containers:
        - image: rabbitmq
          name: rabbitmq
          ports:
            - containerPort: 5672
          resources:
            limits:
              cpu: 100m
```

运行 `$ kubectl create -f examples/celery-rabbitmq/rabbitmq-controller.yaml` 这个命令来创建副本控制器，确保当一个RabbitMQ实例运行时，一个pod已经存在。

请注意创建这个pod需要一些时间来拉取一个docker镜像。在这个例子中，这些操作也适用于其他pods。

第三步：启动**Celery**

通过 `$ kubectl create -f examples/celery-rabbitmq/celery-controller.yaml` 来创建一个celery worker,文件如下：

```

apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: celery
    name: celery-controller
spec:
  replicas: 1
  selector:
    component: celery
  template:
    metadata:
      labels:
        app: taskQueue
        component: celery
    spec:
      containers:
        - image: endocode/celery-app-add
          name: celery
          ports:
            - containerPort: 5672
          resources:
            limits:
              cpu: 100m

```

一些地方需要指出...:

像RabbitMQ控制器，需要确保总是有一个pod运行着Celery worker实例。

celery-app-add这个镜像是对标准的Celery Docker镜像的扩展。Dockerfile如下：

```

FROM library/celery

ADD celery_conf.py /data/celery_conf.py
ADD run_tasks.py /data/run_tasks.py
ADD run.sh /usr/local/bin/run.sh

ENV C_FORCE_ROOT 1

CMD ["/bin/bash", "/usr/local/bin/run.sh"]

```

celery_conf.py文件包含了一个简单的celery加法运算任务。最后一行启动Celery worker。

注意：`ENV C_FORCE_ROOT 1` 用来确保Celery以root用户身份运行，不提倡在生产环境中采用这种方式。

celery_conf.py文件的内容如下：

```

import os

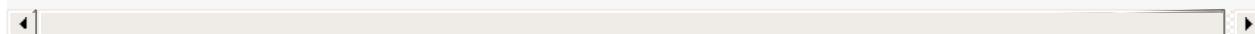
from celery import Celery

# Get Kubernetes-provided address of the broker service
broker_service_host = os.environ.get('RABBITMQ_SERVICE_SERVICE_HOST')

app = Celery('tasks', broker='amqp://guest@%s//' % broker_service_host, backend='amqp')

@app.task
def add(x, y):
    return x + y

```



假设你已经熟悉Celery的运行机制，除了这

个 `os.environ.get('RABBITMQ_SERVICE_SERVICE_HOST')` 部分。第一步创建的RabbitMQ服务的IP地址已经在环境变量中设置。Kubernetes会自动对所有定义了名为RabbitMQ服务的应用程序标签的容器提供环境变量（这个例子中叫“任务队列”）。上面那段Python代码，会在pod运行时自动填充代理地址。

第二个python脚本(`run_tasks.py`)，会以五秒为间隔，周期性的执行add随机数字的任务。

现在的问题是，你怎么看发生了什么？

第四步：弄一个前端

Flower是一个基于web的工具，用来监控和管理Celery集群。通过连接到一个包含Celery的节点，你可以实时看到所有worker以及他们的任务的工作情况。

首先，通过 `$ kubectl create -f examples/celery-rabbitmq/flower-service.yaml` 命令来启动一个Flower服务。这个服务定义如下：

```

apiVersion: v1
kind: Service
metadata:
  labels:
    name: flower
    name: flower-service
spec:
  ports:
  - port: 5555
  selector:
    app: taskQueue
    component: flower
  type: LoadBalancer

```

它会被标记为一个外部负载均衡器。然而，在许多平台上，必须添加一个明确的防火墙规则，打开5555端口。GCE上可以这么操作：

```
$ gcloud compute firewall-rules create --allow=tcp:5555 --target-tags=kubernetes-minion
```

请记住在运行完这个例子后删除这条规则 (on GCE: `$ gcloud compute firewall-rules delete kubernetes-minion-5555`)。

运行下面命令来启动pods, `$ kubectl create -f examples/celery-rabbitmq/flower-controller.yaml`。这个控制器是这么定义的：

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: flower
    name: flower-controller
spec:
  replicas: 1
  selector:
    component: flower
  template:
    metadata:
      labels:
        app: taskQueue
        component: flower
    spec:
      containers:
        - image: endocode/flower
          name: flower
        resources:
          limits:
            cpu: 100m
```

这将创建一个新的pod,这个pod安装了Flower，服务节点会暴露5555端口（Flower的默认端口）。这个镜像使用下面命令启动Flower:

```
flower --broker=amqp://guest:guest@${RABBITMQ_SERVICE_SERVICE_HOST}:localhost:5672//
```

同样，它使用Kubernetes提供的环境变量来获取RabbitMQ服务的IP地址。

一旦所有的pods启动并且运行，运行 `kubectl get pods` 命令会显示下面内容：

NAME	READY	REASON	RESTARTS	AGE
celery-controller-wqkz1	1/1	Running	0	8m
flower-controller-7bglc	1/1	Running	0	7m
rabbitmq-controller-5eb21	1/1	Running	0	13m

`kubectl get service flower-service` 命令会帮助你获取Flower服务的外部IP地址。

NAME	LABELS	SELECTOR	IP(S)	PORT(S)
flower-service	name=flower	app=taskQueue,component=flower	10.0.44.166 162.222.181.180	5555/TCP

在你的网页浏览器中输入正确的flower服务地址和5555端口，（在这个例子中，<http://162.222.181.180:5555>）。如果你点击叫作“任务”的标签，你应该会看到一个不断增长的名为"celery_conf.add"的任务表单，它显示了 `run_tasks.py` 脚本的调度情况。

Kubernetes在Hazelcast平台上部署原生云应用

译者：贾澜鹏 校对：无

这篇文档主要是描述Kubernetes在Hazelcast平台上部署原生云应用的方法。当我们提到原生云应用时，意味着我们的应用程序是运行在一个集群之上，同时使用这个集群的基础设施实现这个应用程序。值得注意的是，在此情况下，一个定制化的Hazelcast 引导程序 被用来使 Hazelcast可以动态的发现已经加入集群的Hazelcast节点。

当拓扑结构发生变化时，需要Hazelcast节点自身进行交流和处理。

本文档同样也尝试去解释Kubernetes的核心组件：Pods、Service和ReplicationController。

前提

下面的例子假定你已经安装了Kubernetes集群并且可以运行。同时你也已经在你的路径下安装了kubectl命令行工具。可以从“[开始](#)”这个小节中得到相应平台的安装指令。

给急性子的备注

下面的介绍会有点长，如果你想跳过直接开始的话，请看结尾处的“太长不读版”。

资源

免费的资源如下：

- Hazelcast节点发现 - <https://github.com/pires/hazelcast-kubernetes-bootstrapper>
- Dockerfile文件 -<https://github.com/pires/hazelcast-kubernetes>
- 官方的Docker镜像 - <https://quay.io/repository/pires/hazelcast-kubernetes>

简单的单调度单元的Hazelcast节点

在Kubernetes中，最小的应用单元就是Pod。一个Pod就是同一个主机调度下的一个或者多个容器。在一个Pod中的所有容器共享同一个网络命名空间，同时可以有选择性的共享同一个数据卷。

在这种情况下，我们不能单独运行一个Hazelcast Pod，因为它的发现机制依赖于Service的定义。

添加一个Hazelcast 服务

在Hazelcast中，一个Service被描述为执行同一任务的Pods集合。比如，一个Hazelcast集群中的节点集合。Service的一个重要用途就是通过建立一个均衡负载器将流量均匀的分到集合中的每一个成员。此外，Service还可以作为一个标准的查询器，使动态变化的Pod集合提供有效通过Kubernetes的API。实际上，这个就是探索机制的工作原理，就是在service的基础上去发现Hazelcast Pods。下面是对Service的描述：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: hazelcast
  name: hazelcast
spec:
  ports:
    - port: 5701
  selector:
    name: hazelcast
```

这里值得注意的是selector（选择器）。在标签的上层有一个查询器，它标识了被Service所覆盖的Pods集合。在这种情况下，selector就是代码中 name: hazelcast。在接下来的Replication Controller说明书中，你会看到Pods中有对应的标签，那么它就会被这个Service中对应的成员变量所选中。

创建该Service的命令如下：

```
$ kubectl create -f examples/hazelcast/hazelcast-service.yaml
```

添加一个拷贝节点

Kubernetes和Hazelcast真正强大的地方在于它们可以轻松的建立一个可拷贝的、大小可调的Hazelcast集群。

在Kubernetes中，存在一个叫做Replication Controller的管理器，专门用来管理相同Pods的拷贝集合。和Service一样，它也存在一个在集合成员变量中定义的选择查询器。和Service不同的是，它对拷贝的个数有要求，会通过创建或者删除Pods来确保当前Pods的数量符合要求。

Replication Controllers会通过匹配响应的选择查询器来确认要接收的Pods，下面我们将创建一个单拷贝的Replication Controller去接收已经存在的Hazelcast Pod。

```

apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: hazelcast
  name: hazelcast
spec:
  replicas: 1
  selector:
    name: hazelcast
  template:
    metadata:
      labels:
        name: hazelcast
    spec:
      containers:
        - resources:
            limits:
              cpu: 0.1
          image: quay.io/pires/hazelcast-kubernetes:0.5
          name: hazelcast
        env:
          - name: "DNS_DOMAIN"
            value: "cluster.local"
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        ports:
          - containerPort: 5701
            name: hazelcast

```

在这段代码中，有一些需要注意的东西。首先要注意的是，我们运行的是`quay.io/pires/hazelcast-kubernetes` image, tag 0.5。这个busybox安装在JRE8上。尽管如此，它还是添加了一个用户端的应用程序，从而可以发现集群中的Hazelcast节点并且引导一个Hazelcast实例。`HazelcastDiscoveryController`通过内置的搜索服务器来探索Kubernetes API Server，之后用Kubernetes API来发现新的节点。

你可能已经注意到了，我们会告知Kubernetes，容器会暴露Hazelcast端口。最终，我们需要告诉集群的管理器，我们需要一个CPU核。

对于Hazelcast Pod而言，Replication Controller块的配置基本相同，以上就是声明，它只是给管理器提供一种简单的建立新节点的方法。其它的部分就是包含了Controller选择条件的 `selector`，以及配置Pod数量的 `replicas`，在这个例子中数量为1。

最后，我们需要根据你的Kubernetes集群的DNS配置来设置 `DNS_DOMAIN` 的环境变量。

创建该控制器的命令：

```
$ kubectl create -f examples/hazelcast/hazelcast-controller.yaml
```

当控制器成功的准备好后，你就可以查询服务端点：

```
$ kubectl get endpoints hazelcast -o json
{
  "kind": "Endpoints",
  "apiVersion": "v1",
  "metadata": {
    "name": "hazelcast",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/endpoints/hazelcast",
    "uid": "094e507a-2700-11e5-abbc-080027eae546",
    "resourceVersion": "4094",
    "creationTimestamp": "2015-07-10T12:34:41Z",
    "labels": {
      "name": "hazelcast"
    }
  },
  "subsets": [
    {
      "addresses": [
        {
          "ip": "10.244.37.3",
          "targetRef": {
            "kind": "Pod",
            "namespace": "default",
            "name": "hazelcast-nsyzn",
            "uid": "f57eb6b0-2706-11e5-abbc-080027eae546",
            "resourceVersion": "4093"
          }
        }
      ],
      "ports": [
        {
          "port": 5701,
          "protocol": "TCP"
        }
      ]
    }
  ]
}
```

你可以看到Service发现那些被Replication Controller建立的Pods。

这下变的更加有趣了。让我们把集群提高到2个Pod。

```
$ kubectl scale rc hazelcast --replicas=2
```

现在，如果你去列出集群中的Pods,你应该会看到2个Hazelcast Pods:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hazelcast-nanfb 1/1     Running   0          40s
hazelcast-nsyzn 1/1     Running   0          2m
kube-dns-xudrp 3/3     Running   0          1h
```

如果想确保每一个Pods都在工作，你可以通过`log`命令来进行日志检查，如下：

```
$ kubectl log hazelcast-nanfb hazelcast
2015-07-10 13:26:34.443 INFO 5 --- [           main] com.github.pires.hazelcast.Applicat
2015-07-10 13:26:34.535 INFO 5 --- [           main] s.c.a.AnnotationConfigApplicationCo
2015-07-10 13:26:35.888 INFO 5 --- [           main] o.s.j.e.a.AnnotationMBeanExporter
2015-07-10 13:26:35.924 INFO 5 --- [           main] c.g.p.h.HazelcastDiscoveryControlle
2015-07-10 13:26:37.259 INFO 5 --- [           main] c.g.p.h.HazelcastDiscoveryControlle
2015-07-10 13:26:37.404 INFO 5 --- [           main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.405 INFO 5 --- [           main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.415 INFO 5 --- [           main] c.h.instance.DefaultAddressPicker
2015-07-10 13:26:37.852 INFO 5 --- [           main] com.hazelcast.spi.OperationService
2015-07-10 13:26:37.879 INFO 5 --- [           main] c.h.s.i.o.c.ClassicOperationExecuto
2015-07-10 13:26:38.531 INFO 5 --- [           main] com.hazelcast.system
2015-07-10 13:26:38.532 INFO 5 --- [           main] com.hazelcast.system
2015-07-10 13:26:38.533 INFO 5 --- [           main] com.hazelcast.instance.Node
2015-07-10 13:26:38.534 INFO 5 --- [           main] com.hazelcast.core.LifecycleService
2015-07-10 13:26:38.672 INFO 5 --- [           cached1] com.hazelcast.nio.tcp.SocketConnect
2015-07-10 13:26:38.683 INFO 5 --- [           cached1] c.h.nio.tcp.TcpIpConnectionManager
2015-07-10 13:26:45.699 INFO 5 --- [ration.thread-1] com.hazelcast.cluster.ClusterServic

Members [2] {
    Member [10.244.37.3]:5701
    Member [10.244.77.3]:5701 this
}

2015-07-10 13:26:47.722 INFO 5 --- [           main] com.hazelcast.core.LifecycleService
2015-07-10 13:26:47.723 INFO 5 --- [           main] com.github.pires.hazelcast.Applicat
```

接着是4个Pods:

```
$ kubectl scale rc hazelcast --replicas=4
```

然后通过刚才的操作去检查这4个成员是否连接。

太长不读版

对于那些急性子，下面是这一章所用到的所有命令：

```
# 建立一个service去跟踪所有的Hazelcast Nodes
kubectl create -f examples/hazelcast/hazelcast-service.yaml

# 建立一个Replication Controller去拷贝Hazelcast Nodes
kubectl create -f examples/hazelcast/hazelcast-controller.yaml

# 升级成2个节点
kubectl scale rc hazelcast --replicas=2

# 升级成4个节点
kubectl scale rc hazelcast --replicas=4
```

Hazelcast的发现机制源代码

[点击这里](#)

Meteor on Kuberenetes

译者：贾润鹏 校对：无

这个例子将会向你展示如何在Kubernetes上打包运行一个[Meteor app](#)。

从谷歌的计算引擎开始

Meteor使用MongoDB，并且我们使用GCEPersistentDisk类型的卷作为永久存储介质。所以，这个实例只使用于[谷歌的计算引擎](#)。如果想选用其它的方式，可以去查看一下[卷使用文档](#)。

首先，你需要完成如下操作：

1. 建立一个[谷歌云平台](#)的项目。
2. 启用Google的[付费API](#)？？
3. 安装[谷歌云的SDK](#)。

认证谷歌云并且将谷歌云的默认项目名称指向你希望部署Kubernetes集群的项目：

```
gcloud auth login  
gcloud config set project <project-name>
```

之后，开启一个Kubernetes集群

```
wget -q -O - https://get.k8s.io | bash
```

所有细节和其它方式下启动集群的方法，请参考[谷歌GCE平台的Kubernetes入门指导](#)。

为你的Meteor APP创建一个容器

为了使一个Meteor APP能运行再Kubernetes上，你首先需要建立一个Docker容器。在建立之前你需要安装[DOCKER](#)。一旦这些都具备了，你需要将 `Dockerfile` 和 `.dockerignore` 这2个文件添加到当前的Meteor工程中。

Dockerfile文件中应该包含下面的语句。其中的 `ROOT_URL` 你应该替换为当前APP的主机名。

```
FROM cheeses/meteor-kubernetes
ENV ROOT_URL http://myawesomeapp.com
```

对于 `.dockerignore` 文件，应该包含下面的语句。它是为了告诉Docker，再建立你的容器时忽略以下指定路径的文件。

```
.meteor/local
packages/*/.build*
```

你可以在下面的链接中看到一个已经建立起来的Meteor工程：[meteor-gke-example](#)。你可以随意的使用这个例子中的APP。如果你已经在你的Meteor工程中添加了移动平台，那么下面的步骤将不起作用。所以替换你的平台为 `meteor list-platforms`。

现在，你就可以在你的Meteor工程中运行下面的语句来建立一个容器：

```
docker build -t my-meteor .
```

推送一个注册表

对于[Docker Hub](#)，你需要利用下面的命令向Hub推送一个携带了你的用户名的APP图片，请注意替换 `<username>` 为你对应的Hub用户名。命令如下：

```
docker tag my-meteor <username>/my-meteor
docker push <username>/my-meteor
```

对于[Google Container Registry](#)，你需要向GCR推送一个携带你的工程ID的APP图片，同时，请注意替换 `<project>` 为你的对应工程ID。

```
docker tag my-meteor gcr.io/<project>/my-meteor
gcloud docker push gcr.io/<project>/my-meteor
```

运行

现在，你已经容器化了你的Meteor APP，是时候开始建立你的集群了。编辑 `meteor-controller.json` 同时确保 `image:` 与你刚刚推送到Docker Hub或者GCR的容器是相对应的。我们将需要一个MongoDB作为一个永久的Kubernetes卷来存放数据。相关的选项可以参考[volumes documentation](#)。我们将利用谷歌的计算引擎永久磁盘。创建MongoDB磁盘的命令如下：

```
gcloud compute disks create --size=200GB mongo-disk
```

你可以开启Mongo去使用那些磁盘， 命令如下：

```
kubectl create -f examples/meteor/mongo-pod.json
kubectl create -f examples/meteor/mongo-service.json
```

等待Mongo完全启动，之后你就可以开启自己的Meteor APP:

```
kubectl create -f examples/meteor/meteor-service.json
kubectl create -f examples/meteor/meteor-controller.json
```

需要注意的是，`meteor-service.json` 创建了一个负载均衡器，所以你的APP要求能在Meteor Pods启动时，有效的通过负载均衡器的IP。我们会再建立RC之前建立一个服务，该服务会提供反向关联（或者其它的什么）用以帮助调度程序去排列pod，从而帮助调度程序匹配对应的pods。你可以通过下面的命令获得你的负载均衡器的IP：

```
kubectl get service meteor --template="{{range .status.loadBalancer.ingress}} {{.ip}} {{end}}
```

之后，你需要在你的环境上开启你的80号端口。如果是使用谷歌计算引擎的用户，你需要运行下面的命令：

```
gcloud compute firewall-rules create meteor-80 --allow=tcp:80 --target-tags kubernetes-mi
```

接下来呢？

首先，在`Dockerfile`中的`FROM chees/meteor-kubernetes`语句会指定Meteor APP的基础镜像。这个镜像的构建代码放置在示例代码工程的`dockerbase/`子目录下。你可以通过阅读`Dockerfile`文件中的代码来了解`docker build`的步骤。这个镜像是基于Node.js官方镜像构建的。它会安装Meteor并将用户的程序拷贝进去。其中的最后一行的命令指出了你的app需要通过怎样的命令在容器中运行起来。

```
ENTRYPOINT MONGO_URL=mongodb://$MONGO_SERVICE_HOST:$MONGO_SERVICE_PORT /usr/local/bin/nod
```

在上面的命令中，我们能看到传递到Meteor App的MongoDB主机和端口信息。`MONGO_SERVICE...` 这个环境变量是由Kubernetes设置的，同时，由 `mongo-service.json` 指定了名为 `mongo` 的服务器的详细信息。更多细节可以参考[environment documentation](#)。

也许你已经了解，Meteor需要使用TCP长连接，以及持续性的Session（Sticky Sessions）。通过Kubernetes用户能够轻松的在集群Scale out时保持节点的Session相关性。在`meteor-service.json`文件中所包含的 `"sessionAffinity": "clientIP"`，就向我们提供了这些。更多的信息请参考 [service documentation](#)。

就向之前所提到的，`mongo`容器使用了一个被Kubernetes映射到永久磁盘的卷。在 `mongo-pod.json` 中，容器对指定的卷进行了划分：

```
{  
  "volumeMounts": [  
    {  
      "name": "mongo-disk",  
      "mountPath": "/data/db"  
    }  
  ]}
```

`mongo-disk`是指超出容器范围的卷：

```
{  
  "volumes": [  
    {  
      "name": "mongo-disk",  
      "gcePersistentDisk": {  
        "pdName": "mongo-disk",  
        "fsType": "ext4"  
      }  
    }  
  ],  
  "containers": [  
    {  
      "name": "meteor",  
      "image": "gcr.io/meteorapp/meteor:0.14.0",  
      "volumeMounts": [  
        {  
          "name": "mongo-disk",  
          "mountPath": "/data/db"  
        }  
      ]  
    }  
  ]}
```

配置文件使用入门

译者：钟健鑫 校对：无

除了其他地方所描述的命令行式风格的命令以外，Kubernetes还支持以YAML或者JSON格式的配置方式，很多时候，配置文件是优于纯命令方式的，一旦他们能够被签入版本管理，文件的变化也能够像代码文件一样被回顾和管理，就能生产出更健壮，可靠和可存档系统。

通过pod的配置文件来运行容器

```
$ cd kubernetes  
$ kubectl create -f ./pod.yaml  
Where pod.yaml contains something like:  
  
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx  
  labels:  
    app: nginx  
spec:  
  containers:  
  - name: nginx  
    image: nginx  
    ports:  
    - containerPort: 80
```

通过以下命令来查看集群中pod的信息：

```
$ kubectl get pods  
and delete the pod you just created:  
  
$ kubectl delete pods nginx
```

通过配置文件来运行容器的复制集

要让被复制的容器运行，你需要一个Replication Controller。一个Replication Controller负责保证规定数量的pod会一直存在于所在集群当中

```
$ cd kubernetes  
$ kubectl create -f ./replication.yaml
```

一般replication.yaml会包含:

```
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
```

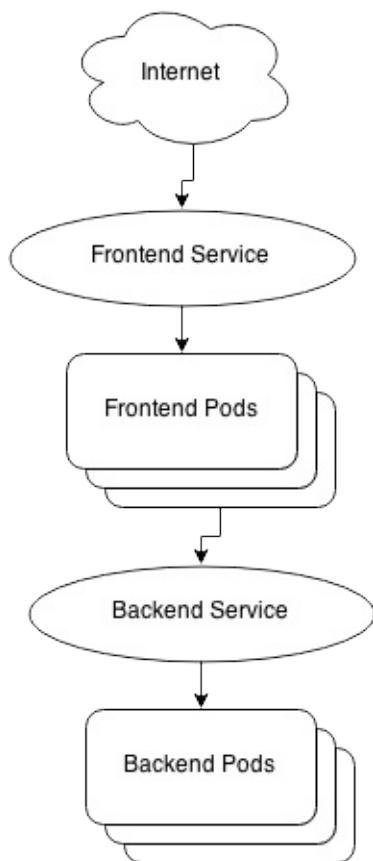
删除Replication Controller品（还包括她所创建的pod）

```
$ kubectl delete rc nginx
```

环境向导示例

译者：丁麒玮 校对：无

这个示例展示了pod,replication controller 和 service是如何运行的。示例给出了两种pod:前端pod和后端pod，在这两种pod的前面都有service与之连接。访问前端pod将返回他自己的环境变量信息，以及该pod通过service可以访问的后端pod。这个示例主要目的是说明在k8s集群里运行的容器时，可利用的环境变量元数据。k8s环境变量文档在[这里](#)。



前提条件

这个示例假设你已经安装了可运行的k8s集群，并且安装了kubectl命令行工具且已经添加到了环境变量。安装k8s环境请阅读[getting started](#)文档。

可选择：构建自己的容器

容器的代码在这里 [containers/](#)。

开始运行

```
kubectl create -f ./backend-rc.yaml
kubectl create -f ./backend-srv.yaml
kubectl create -f ./show-rc.yaml
kubectl create -f ./show-srv.yaml
```

查询service

用 `kubectl describe service show-srv` 来确认你的service 的public ip。

注意：如果你的平台不支持外网负载均衡，你需要在内网开一个合适的端口，并且直接连内网ip，用以上的命令来输出给前端的service。

运行 `curl <public ip>:80` 来查询service。你应该得到返回信息：

```
Pod Name: show-rc-xxu6i
Pod Namespace: default
USER_VAR: important information

Kubernetes environment variables
BACKEND_SRV_SERVICE_HOST = 10.147.252.185
BACKEND_SRV_SERVICE_PORT = 5000
KUBERNETES_RO_SERVICE_HOST = 10.147.240.1
KUBERNETES_RO_SERVICE_PORT = 80
KUBERNETES_SERVICE_HOST = 10.147.240.2
KUBERNETES_SERVICE_PORT = 443
KUBE_DNS_SERVICE_HOST = 10.147.240.10
KUBE_DNS_SERVICE_PORT = 53

Found backend ip: 10.147.252.185 port: 5000
Response from backend
Backend Container
Backend Pod Name: backend-rc-6qiya
Backend Namespace: default
```

首先，打印前端pod的信息，其name和 namespace是从 Downward API检索出来的。其次，`USER_VAR` 是我们定义在pod里的环境变量名字，然后，扫描动态的k8s环境变量并且打印，这些都是用来找出名字是 `backend-srv` 的后端的service。最后前端的pod向后端service请求并打印返回的信息。然后后端的pod返回他的pod的name和namespace。

尝试多次运行 `curl` 命令，并且注意变化。例如：`watch -n 1 curl -s <ip>` 首先，前端的Service每次将你的请求分配到不同的前端pod，前端的pod通过后端的service与后端的pod连接，因此，每个的后端pod都可以处理各个请求。

清除工作

```
kubectl delete rc,service -l type=show-type  
kubectl delete rc,service -l type=backend-type
```

Downward API 范例

译者：钟健鑫 校对：无

接下来的例子，你会创建一个pod，它包含了一个通过访问[downward API](#)来使用这个pod名字和命名空间的容器

步骤 0: 前提条件

这个例子会假设你有一个安装好并正在运行的Kubernetes集群，也已经在系统的某个路径下安装好了kubectl命令行工具。具体安装步骤请在[安装入门](#)中找到与你平台对应的安装说明。

步骤 1: 创建一个pod

容器可以通过环境变量来消费downward API，而且downward API允许容器通过被注入的方式来使用所在pod的name和namespace等信息

下面我们使用examples/downward-api/dapi-pod.yaml来创建一个pod，并让它其中的容器通过downward API获取了所属pod的信息：

```
$ kubectl create -f docs/user-guide/downward-api/dapi-pod.yaml
```

检查日志

这个pod在一个容器中运行env命令，来调用downward API。接下来你就能通过过滤pod的日志来看到这个pod被注入的确切的值：

```
$ kubectl logs dapi-test-pod | grep POD_
2015-04-30T20:22:18.568024817Z POD_NAME=dapi-test-pod
2015-04-30T20:22:18.568087688Z POD_NAMESPACE=default
```

在Kubernetes上运行你的第一个容器

译者：钟健鑫 校对：无

好了，如果你已经开始了任何一个入门指南，并且启动了一个Kubernetes集群。那么接下来呢？这个片指南会帮助你正对Kubernetes，在其集群上运行第一个容器。

运行一个容器（简单版）

从这时开始，我假设你已经根据其它入门指南安装了kubectl。

下面这行kubectl命令会穿件两个监听80端口的nginx pod. 还会创建一名为my-nginx个replication controller,用来保证始终会有两个pod在运行。

```
kubectl run my-nginx --image=nginx --replicas=2 --port=80
```

一旦这些pod被创建好了，你可以列出他们并查看他们的启动和运行。

```
kubectl get pods
```

你也能够看见replication controller被创建了：

```
kubectl get rc
```

To stop the two replicated containers, stop the replication controller: 如果要停止这两个被复制的容器，你可以通过停止replication controller

```
kubectl stop rc my-nginx
```

让你的的pod可以被外网方位.

在一些平台上（例如Google Compute Engine），kubectl命令能够集成云端提供的API来给pod条件公有IP地址，可以通过以下命令来实现：

```
kubectl expose rc my-nginx --port=80 --type=LoadBalancer
```

这个命令会打印出被创建了的service,以及一个外部IP地址映射到service. 对外的IP地址根你实际运行环境有关。例如，对于Google Compute Engine的外部IP地址会被列为新创建的服务的一部分，还可以通过在运行时检索。

```
kubectl get services
```

为了访问你的nginx初始页面,你还不得不保证通过外部IP的通信是被允许的。那么就要通过让防火墙允许80端口通信才可以做到

接下来: 配置文件

Most people will eventually want to use declarative configuration files for creating/modifying their applications. A simplified introduction is given in a different document.

大多数人最终都会使用声明式的配置文件来创建或修改他们的应用程勋。另外一个文档给出了一个[简单介绍](#)。

Kubernetes DNS 实例实战

译者：钟健鑫 校对：无

这是一个好玩的例子来演示如何使用kubernetes DNS

第0步：前提条件

当前例子会假设你已经从仓库获取并启动了一个Kubernetes集群。请确保已经在设置中启动了DNS, 具体请移步[DNS 文档](#):

```
$ cd kubernetes  
$ hack/dev-build-and-up.sh
```

第一步：创建两个namespace

接下来我们会明白集群的DNS如何在跨多个namespace的情况下工作，首先我们需要创建两个namespace:

```
$ kubectl create -f examples/cluster-dns/namespace-dev.yaml  
$ kubectl create -f examples/cluster-dns/namespace-prod.yaml
```

现在我们列出所有的 namespaces:

```
$ kubectl get namespaces  
NAME      LABELS      STATUS  
default    <none>     Active  
development  name=development  Active  
production   name=production  Active
```

For kubectl client to work with each namespace, we define two contexts: 为了让kubectl客户端操作每一个不同的namespace, 我们定义了两个上下文环境：

```
$ kubectl config set-context dev --namespace=development --cluster=${CLUSTER_NAME} --user  
$ kubectl config set-context prod --namespace=production --cluster=${CLUSTER_NAME} --user
```

You can view your cluster name and user name in kubernetes config at `~/.kube/config`.

Step Two: Create backend replication controller in each namespace

Use the file examples/cluster-dns/dns-backend-rc.yaml to create a backend server replication controller in each namespace.

```
$ kubectl config use-context dev  
$ kubectl create -f examples/cluster-dns/dns-backend-rc.yaml
```

Once that's up you can list the pod in the cluster:

```
$ kubectl get rc  
CONTROLLER      CONTAINER(S)        IMAGE(S)          SELECTOR          REPLICAS  
dns-backend     dns-backend        ddysher/dns-backend   name=dns-backend    1
```

Now repeat the above commands to create a replication controller in prod namespace:

```
$ kubectl config use-context prod  
$ kubectl create -f examples/cluster-dns/dns-backend-rc.yaml  
$ kubectl get rc  
CONTROLLER      CONTAINER(S)        IMAGE(S)          SELECTOR          REPLICAS  
dns-backend     dns-backend        ddysher/dns-backend   name=dns-backend    1
```

Step Three: Create backend service

Use the file examples/cluster-dns/dns-backend-service.yaml to create a service for the backend server.

```
$ kubectl config use-context dev  
$ kubectl create -f  
examples/cluster-dns/dns-backend-service.yaml
```

Once that's up you can list the service in the cluster:

```
$ kubectl get service dns-backend  
NAME      LABELS      SELECTOR          IP(S)      PORT(S)  
dns-backend <none>    name=dns-backend  10.0.236.129  8000/TCP
```

Again, repeat the same process for prod namespace:

```
$ kubectl config use-context prod
$ kubectl create -f examples/cluster-dns/dns-backend-service.yaml
$ kubectl get service dns-backend
NAME      LABELS      SELECTOR      IP(S)      PORT(S)
dns-backend  <none>  name=dns-backend  10.0.35.246  8000/TCP
```

Step Four: Create client pod in one namespace

Use the file `examples/cluster-dns/dns-frontend-pod.yaml` to create a client pod in dev namespace. The client pod will make a connection to backend and exit. Specifically, it tries to connect to address <http://dns-backend.development.cluster.local:8000>.

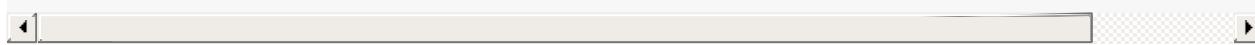
```
$ kubectl config use-context dev
$ kubectl create -f
examples/cluster-dns/dns-frontend-pod.yaml
```

Once that's up you can list the pod in the cluster:

```
$ kubectl get pods dns-frontend
NAME      READY      STATUS      RESTARTS      AGE
dns-frontend  0/1      ExitCode:0  0            1m
```

Wait until the pod succeeds, then we can see the output from the client pod:

```
$ kubectl logs dns-frontend
2015-05-07T20:13:54.147664936Z 10.0.236.129
2015-05-07T20:13:54.147721290Z Send request to: http://dns-backend.development.cluster.lo
2015-05-07T20:13:54.147733438Z <Response [200]>
2015-05-07T20:13:54.147738295Z Hello World!
```



Please refer to the source code about the log. First line prints out the ip address associated with the service in dev namespace; remaining lines print out our request and server response.

If we switch to prod namespace with the same pod config, we'll see the same result, i.e. dns will resolve across namespace.

```
$ kubectl config use-context prod
$ kubectl create -f examples/cluster-dns/dns-frontend-pod.yaml
$ kubectl logs dns-frontend
2015-05-07T20:13:54.147664936Z 10.0.236.129
2015-05-07T20:13:54.147721290Z Send request to: http://dns-backend.development.cluster.lo
2015-05-07T20:13:54.147733438Z <Response [200]>
2015-05-07T20:13:54.147738295Z Hello World!
```

Note about default namespace If you prefer not using namespace, then all your services can be addressed using default namespace, e.g. <http://dns-backend.default.cluster.local:8000>, or shorthand version <http://dns-backend:8000>

tl; dr;

For those of you who are impatient, here is the summary of the commands we ran in this tutorial. Remember to set first \$CLUSTER_NAME and \$USER_NAME to the values found in `~/.kube/config`.

```
# create dev and prod namespaces
kubectl create -f examples/cluster-dns/namespace-dev.yaml
kubectl create -f examples/cluster-dns/namespace-prod.yaml

# create two contexts
kubectl config set-context dev --namespace=development --cluster=${CLUSTER_NAME} --user=$
kubectl config set-context prod --namespace=production --cluster=${CLUSTER_NAME} --user=$

# create two backend replication controllers
kubectl config use-context dev
kubectl create -f examples/cluster-dns/dns-backend-rc.yaml
kubectl config use-context prod
kubectl create -f examples/cluster-dns/dns-backend-rc.yaml

# create backend services
kubectl config use-context dev
kubectl create -f examples/cluster-dns/dns-backend-service.yaml
kubectl config use-context prod
kubectl create -f examples/cluster-dns/dns-backend-service.yaml

# create a pod in each namespace and get its output
kubectl config use-context dev
kubectl create -f examples/cluster-dns/dns-frontend-pod.yaml
kubectl logs dns-frontend

kubectl config use-context prod
kubectl create -f examples/cluster-dns/dns-frontend-pod.yaml
kubectl logs dns-frontend
```



滚动升级示例

译者：赵帅龙 校对：无

本例展示了如何使用Kubernetes对一组正在运行的pods做滚动升级 rolling update。如果你还不知道为什么需要滚动升级，或者什么时候做滚动升级，请查看[这里](#)。更多信息查看[滚动升级设计文档](#)。

第一步：前提条件

本例子假设你已经fork了一份Kubernetes代码，并且创建了一个Kubernetes集群：

```
$ cd kubernetes  
$ ./cluster/kube-up.sh
```

Step One: Turn up the UX for the demo

You can use bash job control to run this in the background (note that you must use the default port -- 8001 -- for the following demonstration to work properly). This can sometimes spew to the output so you could also run it in a different terminal. You have to run `kubectl proxy` in the root of the Kubernetes repository. Otherwise you will get "404 page not found" errors as the paths will not match. You can find more information about `kubectl proxy` [here](#).

```
$ kubectl proxy --www=docs/user-guide/update-demo/local/ &  
I0218 15:18:31.623279 67480 proxy.go:36] Starting to serve on localhost:8001
```

Now visit the the [demo website](#). You won't see anything much quite yet.

Step Two: Run the replication controller

Now we will turn up two replicas of an [image](#). They all serve on internal port 80.

```
$ kubectl create -f docs/user-guide/update-demo/nautilus-rc.yaml
```

After pulling the image from the Docker Hub to your worker nodes (which may take a minute or so) you'll see a couple of squares in the UI detailing the pods that are running along with the image that they are serving up. A cute little nautilus.

Step Three: Try scaling the replication controller

Now we will increase the number of replicas from two to four:

```
$ kubectl scale rc update-demo-nautilus --replicas=4
```

If you go back to the [demo website](#) you should eventually see four boxes, one for each pod.

Step Four: Update the docker image

We will now update the docker image to serve a different image by doing a rolling update to a new Docker image.

```
$ kubectl rolling-update update-demo-nautilus --update-period=10s -f docs/user-guide/upda
```

The rolling-update command in kubectl will do 2 things:

1. Create a new [replication controller](#) with a pod template that uses the new image
(`gcr.io/google_containers/update-demo:kitten`)
2. Scale the old and new replication controllers until the new controller replaces the old.
This will kill the current pods one at a time, spinning up new ones to replace them.

Watch the [demo website](#), it will update one pod every 10 seconds until all of the pods have the new image. Note that the new replication controller definition does not include the replica count, so the current replica count of the old replication controller is preserved. But if the replica count had been specified, the final replica count of the new replication controller will be equal this number.

Step Five: Bring down the pods

```
$ kubectl delete rc update-demo-kitten
```

This first stops the replication controller by turning the target number of replicas to 0 and then deletes the controller.

Step Six: Cleanup

To turn down a Kubernetes cluster:

```
$ ./cluster/kube-down.sh
```

Kill the proxy running in the background: After you are done running this demo make sure to kill it:

```
$ jobs  
[1]+  Running                  ./kubectl proxy --www=local/ &  
$ kill %1  
[1]+  Terminated: 15            ./kubectl proxy --www=local/
```

Updating the Docker images

If you want to build your own docker images, you can set `$DOCKER_HUB_USER` to your Docker user id and run the included shell script. It can take a few minutes to download/upload stuff.

```
$ export DOCKER_HUB_USER=my-docker-id  
$ ./docs/user-guide/update-demo/build-images.sh
```

To use your custom docker image in the above examples, you will need to change the image name in `docs/user-guide/update-demo/nautilus-rc.yaml` and `docs/user-guide/update-demo/kitten-rc.yaml`.

Image Copyright

Note that the images included here are public domain.

- kitten
- nautilus

explorer

译者：赵帅龙 校对：无

Explorer is a little container for examining the runtime environment kubernetes produces for your pods.

The intended use is to substitute gcr.io/google_containers/explorer for your intended container, and then visit it via the proxy.

Currently, you can look at:

- The environment variables to make sure kubernetes is doing what you expect.
- The filesystem to make sure the mounted volumes and files are also what you expect.
- Perform DNS lookups, to see how DNS works.

`pod.yaml` is supplied as an example. You can control the port it serves on with the `-port` flag.

Example from command line (the DNS lookup looks better from a web browser):

```
$ kubectl create -f examples/explorer/pod.yaml
$ kubectl proxy &
Starting to serve on localhost:8001

$ curl localhost:8001/api/v1/proxy/namespaces/default/pods/explorer:8080/vars/
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=explorer
KIBANA_LOGGING_PORT_5601_TCP_PORT=5601
KUBERNETES_SERVICE_HOST=10.0.0.2
MONITORING_GRAFANA_PORT_80_TCP_PROTO=tcp
MONITORING_INFLUXDB_UI_PORT_80_TCP_PROTO=tcp
KIBANA_LOGGING_SERVICE_PORT=5601
MONITORING_HEAPSTER_PORT_80_TCP_PORT=80
MONITORING_INFLUXDB_UI_PORT_80_TCP_PORT=80
KIBANA_LOGGING_SERVICE_HOST=10.0.204.206
KIBANA_LOGGING_PORT_5601_TCP=tcp://10.0.204.206:5601
KUBERNETES_PORT=tcp://10.0.0.2:443
MONITORING_INFLUXDB_PORT=tcp://10.0.2.30:80
MONITORING_INFLUXDB_PORT_80_TCP_PROTO=tcp
MONITORING_INFLUXDB_UI_PORT=tcp://10.0.36.78:80
KUBE_DNS_PORT_53_UDP=udp://10.0.0.10:53
MONITORING_INFLUXDB_SERVICE_HOST=10.0.2.30
ELASTICSEARCH_LOGGING_PORT=tcp://10.0.48.200:9200
ELASTICSEARCH_LOGGING_PORT_9200_TCP_PORT=9200
KUBERNETES_PORT_443_TCP=tcp://10.0.0.2:443
ELASTICSEARCH_LOGGING_PORT_9200_TCP_PROTO=tcp
KIBANA_LOGGING_PORT_5601_TCP_ADDR=10.0.204.206
KUBE_DNS_PORT_53_UDP_ADDR=10.0.0.10
```

```

MONITORING_HEAPSTER_PORT_80_TCP_PROTO=tcp
MONITORING_INFLUXDB_PORT_80_TCP_ADDR=10.0.2.30
KIBANA_LOGGING_PORT=tcp://10.0.204.206:5601
MONITORING_GRAFANA_SERVICE_PORT=80
MONITORING_HEAPSTER_SERVICE_PORT=80
MONITORING_HEAPSTER_PORT_80_TCP=tcp://10.0.150.238:80
ELASTICSEARCH_LOGGING_PORT_9200_TCP=tcp://10.0.48.200:9200
ELASTICSEARCH_LOGGING_PORT_9200_TCP_ADDR=10.0.48.200
MONITORING_GRAFANA_PORT_80_TCP_PORT=80
MONITORING_HEAPSTER_PORT=tcp://10.0.150.238:80
MONITORING_INFLUXDB_PORT_80_TCP=tcp://10.0.2.30:80
KUBE_DNS_SERVICE_PORT=53
KUBE_DNS_PORT_53_UDP_PORT=53
MONITORING_GRAFANA_PORT_80_TCP_ADDR=10.0.100.174
MONITORING_INFLUXDB_UI_SERVICE_HOST=10.0.36.78
KIBANA_LOGGING_PORT_5601_TCP_PROTO=tcp
MONITORING_GRAFANA_PORT=tcp://10.0.100.174:80
MONITORING_INFLUXDB_UI_PORT_80_TCP_ADDR=10.0.36.78
KUBE_DNS_SERVICE_HOST=10.0.0.10
KUBERNETES_PORT_443_TCP_PORT=443
MONITORING_HEAPSTER_PORT_80_TCP_ADDR=10.0.150.238
MONITORING_INFLUXDB_UI_SERVICE_PORT=80
KUBE_DNS_PORT=udp://10.0.0.10:53
ELASTICSEARCH_LOGGING_SERVICE_HOST=10.0.48.200
KUBERNETES_SERVICE_PORT=443
MONITORING_HEAPSTER_SERVICE_HOST=10.0.150.238
MONITORING_INFLUXDB_SERVICE_PORT=80
MONITORING_INFLUXDB_PORT_80_TCP_PORT=80
KUBE_DNS_PORT_53_UDP_PROTO=udp
MONITORING_GRAFANA_PORT_80_TCP=tcp://10.0.100.174:80
ELASTICSEARCH_LOGGING_SERVICE_PORT=9200
MONITORING_GRAFANA_SERVICE_HOST=10.0.100.174
MONITORING_INFLUXDB_UI_PORT_80_TCP=tcp://10.0.36.78:80
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.2
HOME=/

$ curl localhost:8001/api/v1/proxy/namespaces/default/pods/explorer:8080/fs/
mount/
var/
.dockerenv
etc/
dev/
proc/
.dockerinit
sys/
README.md
explorer

$ curl localhost:8001/api/v1/proxy/namespaces/default/pods/explorer:8080/dns?q=elasticsea

```

```
elasticsearch-logging
```

```
Lookup
```

```
LookupNS(elasticsearch-logging):
Result: ([]*net.NS)
Error: <*>lookup elasticsearch-logging: no such host

LookupTXT(elasticsearch-logging):
Result: ([]string)
Error: <*>lookup elasticsearch-logging: no such host

LookupSRV("", "", elasticsearch-logging):
cname: elasticsearch-logging.default.svc.cluster.local.
Result: ([]*net.SRV)[<*>{Target:(string)elasticsearch-logging.default.svc.cluster.loc
Error:

LookupHost(elasticsearch-logging):
Result: ([]string)[10.0.60.245]
Error:

LookupIP(elasticsearch-logging):
Result: ([]net.IP)[10.0.60.245]
Error:

LookupMX(elasticsearch-logging):
Result: ([]*net.MX)
Error: <*>lookup elasticsearch-logging: no such host
```



Kubernetes API

[用户指南](#)记录了主系统和API概念。

[API规范](#)文档描述了API整体规范。

Kubernetes通过[Swagger](#)记录API所有细节。Kubernetes apiserver (aka "master")提供了一个API接口用于获取Kubernetes API的[Swagger spec](#)，默认在路径 /swaggerapi 下，/swagger-ui 是可以使用浏览器查看API文档的UI。我们会定期更新[静态生成UI](#)。

远程访问API在[访问文档](#)有讨论。

Kubernetes API是系统描述性配置的基础。[Kubectl](#)命令行工具被用于创建，更新，删除，获取API对象。

Kubernetes通过API资源存储自己序列化状态(现在存储在[etcd](#))。

Kubernetes被分成多个组件，各部分通过API相互交互。

API更改

根据经验，任何成功的系统都需要成长和改变，可能是一个新的示例出现，也可能是对已有系统进行更改。因此，我们希望Kubernetes API也可以持续的改变和成长。在较长一段时间内，我们不打算中断已有客户端。一般情况，经常会有新的API资源和新的资源字段增加。删除资源或者字段会有一个跟踪废弃流程。删除功能有一个小的废弃策略叫做TBD，但是Kubernetes到达1.0版本时，将会有个详细的策略代替。

如何构成一个兼容性的改变以及如何更改API的详细信息都在[API变化文档](#)

API版本

为了使删除字段或者重构资源表示更加容易，Kubernetes支持多个API版本。每一个版本都在不同API路径下，例如 /api/v1 或者 /apis/extensions/v1beta1 .

我们选择版本时根据API而不是根据资源或者字段来确认的，API为系统资源和行为提供了一个清晰的，一致性的视图，并且可以控制访问生命周期结束和／或测试的APIs。

注意API版本控制和软件版本控制只能间接的相关联。文档[API and release versioning proposal](#) 描述了API版本控制和软件版本控制两者之间的关系。

不同的API版本会有不同级别的稳定性和支持。每个级别的详细描述包含在文档[API Changes documentation](#)中。 内容主要概括如下：

- Alpha 级别:
 - 版本名称包含了 `alpha` (e.g. `v1alpha1`).
 - 可能是有问题的。实现的功能可能隐含问题，功能默认是不可用的。
 - 支持的功能可能在没有通知的情况下随时删除。
 - API的更改可能存在冲突，但是在后续的软件发布中不会有任何通知。
 - 由于bugs风险的增加和缺乏长期的支持，推荐在短暂的集群测试中使用。
- Beta 级别:
 - 版本名称包含了 `beta` (e.g. `v2beta3`).
 - 代码已经测试过。实现的功能认为是安全的，功能默认是可用的。
 - 所有已支持的功能不会被删除，细节可能会发生变化。
 - 对象的模式和／或定义在随后的beta版本或者稳定版本可能以不兼容的方式改变。
在这种情况下，Kubernetes会提供合并到下一个版本的指南。这一过程中可能会要求删除，编译和重新创建API对象。编译过程中可能会要求一些选择。根据不同的功能可能会需要一些应用的下载时间。
 - 在随后的发布中存在一些非兼容性的更改，所已推荐在不重要的非商业化的情况下使用。如果你已经有多个集群，并且可以单独更新，你可以放宽这个限制。
 - 请尝试我们的**beta**版本功能并且给出反馈！因为已经是**beta**版本，对于我们来说太多的更改可能不太实用
- Stable 级别:
 - 版本名称中包含 `vX` 这里的X是一个整数.
 - 稳定版本的功能在后续的版本发布中会一直存在。

API群组

为了使扩展Kubernetes API更加简单，我们正在实现[API groups](#). API群组是一些可以读和／或更改相同基础资源的简单的不同的接口。API群组被定义在一个REST路径下，在 `apiVersion` 的一个序列化对象的字段。

当前有两个API群组在使用：

1. "core"群组，在REST路径 `/api/v1` 下，但不是`apiVersion`字段的一部分。e.g.
`apiVersion: v1` 。
2. "extensions"群组，在REST路径 `/apis/extensions/$VERSION` 下，并且使用 `apiVersion: extensions/$VERSION` (e.g. 当前是 `apiVersion: extensions/v1beta1`).

未来我们希望有更多的API群组，并且所有的群组都在REST路径 `/apis/$API_GROUP` 下，并且使用 `apiVersion: $API_GROUP/$VERSION` 。我们希望将来有一种方式支持三方可以创建自己的[API groups](#)，并且可以避免命名冲突。

扩展资源

默认情况下Jobs, Ingress和HorizontalPodAutoscalers都是可用的。其他的扩展资源通过设置apiserver的runtime-config使其可用。 runtime-config多个值可通过逗号分隔。例如：设置部署可用，jobs不可用命令，`--runtime-config=extensions/v1beta1/deployments=true,extensions/v1beta1/jobs=false`

v1beta1, v1beta2, and v1beta3已经废弃；请转到v1 ASAP

2015年6月4日，Kubernetes v1 API已经默认可以使用。v1beta1和v1beta2 APIs在2015年6月1日删除。v1beta3计划会在2015年6月6日删除。

v1转换技巧(从v1beta3)

我们已经将所有文档和例子转换成v1版本。并且写了一个简单的[API转换工具](#)来进行简单的转换。使用`kubectl create --validate`命令可以使你的json或者yaml按照Swagger spec规范有效。

v1beta3和v1版本之间最重要的不同是服务的改变。

- `service.spec.portalIP` 属性重命名为 `service.spec.clusterIP`。
- `service.spec.createExternalLoadBalancer` 已经被删除。定义了 `service.spec.type: "LoadBalancer"` 来代替，用于创建外部负载均衡。
- `service.spec.publicIPs` 属性已经被废弃，现在叫做 `service.spec.deprecatedPublicIPs`。当v1beta3删除时，这个属性也会被完全删除。这个字段的多数用户使用这个字段来公开nodes上的services端口。现在这些用户应该使用 `service.spec.type: "NodePort"` 来代替。获取更多信息可阅读[External Services](#)。如果这些还是不能满足你的需求，请提交issue或者联系@thockin。

v1beta3和v1版本之间其他的不同：

- `pod.spec.containers[*].privileged` 和 `pod.spec.containers[*].capabilities` 属性已经合并到 `pod.spec.containers[*].securityContext` 属性。可以查看文档[Security Contexts](#)。
- `pod.spec.host` 属性被重命名为 `pod.spec.nodeName`。
- `endpoints.subsets[*].addresses.IP` 被重命名为 `endpoints.subsets[*].addresses.ip`。
- `pod.status.containerStatuses[*].state.termination` 和 `pod.status.containerStatuses[*].lastState.termination` 属性被分别重命名
为 `pod.status.containerStatuses[*].state.terminated` 和 `pod.status.containerStatuses[*].lastState.terminated`。
- `pod.status.Condition` 属性被重命名为 `pod.status.conditions`。

- `status.details.id` 属性被重命名为 `status.details.name`。

v1beta3转换技巧(从v1beta1/2)

v1beta1/2和v1beta3重要不同点:

- 资源 `id` 现在称作``name``。
- `name`, `labels`, `annotations`, 和其他元数据现在内嵌在一个称作 `metadata` 的map中
- `desiredState` 现在称作 `spec`, `currentState` 现在称作 `status`
- `/minions` 已经被移到 `/nodes`, 且资源包括各种 `Node`
- namesapce是必要条件(对于所有资源namesapce)且已经从一个URL参数改为路径 : `/api/v1beta3/namespaces/{namespace}/{resource_collection}/{resource_name}` . 如果之前你没有使用过namespace, 在这可以使用 `default` .
- 所有资源收集器的名字已经小写 - 已经废弃 `replicationControllers`, 使用 ``replicationcontrollers` 来代替。`
- 查看某个资源的改变, 可打开一个HTTP或者Websocket连接查询, 并且提供 ?`watch=true` 请求参数和 `resourceVersion` 参数进行查询。
- `labels` 查询参数已经被重命名为 `labelSelector` 。
- `fields` 查询参数已经被重命名为 `fieldSelector` 。
- Container中的 `entrypoint` 已经被重命名为 `command`, 而 `command` 被重命名为 `args` .
- Container, volume, and node资源的定义已经被内嵌到一个maps (e.g., `resources{cpu:1}`)并没有作为独立的字段, 并且资源数值支持后缀而不是固定的(e.g., milli-cores)。
- 重新启动策略简单的表示为字串(e.g., `"Always"`)而不是作为一个内嵌的map(`always{}`).
- Pull策略从 `PullAlways`, `PullNever`, 和 `PullIfNotPresent` 变成了 `Always`, `Never`, 和 `IfNotPresent` 。
- `volume source` 被内联到 `volume` 而非嵌套。
- 宿主机volumes已经从 `hostDir` 变成了 `hostPath`, 这样做的好处是, 挂载的可以是一个文件也可以是一个路径。

译者：安雪艳 校对：无

资源管理

本篇文章的目的是为了那些已经使用过一些示例并且想学习更多关于kubectl管理资源pod或者services的用户。想直接通过REST API访问的用户和想扩展Kubernetes API的开发者都应该参考文档 [api conventions](#)和[api document](#).

资源自动更新

当创建一个资源例如pod，随后会收到被创建和资源已添加的多个字段。可以参考下面的工作示例：

```
$ cat > /tmp/original.yaml <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: busybox
  restartPolicy: Never
EOF
$ kubectl create -f /tmp/original.yaml
pods/original
$ kubectl get pods/original -o yaml > /tmp/current.yaml
pods/original
$ wc -l /tmp/original.yaml /tmp/current.yaml
  51 /tmp/current.yaml
   9 /tmp/original.yaml
  60 total
```

我们提交的资源只有9行，但是我们收到了51行。如果你执行 `diff -u /tmp/original.yaml /tmp/current.yaml`，你可以看到pod增加的字段。系统通过如下几种方式增加字段：

- 一些字段时在资源创建时同步设置的，一些异步设置的。
 - 例如：`metadata.uid` 同步设置。(获取更多信息请看 [metadata](#)).
 - 例如，`status.hostIP` 是在pod被调度后设置。这些谁都发生的比较快。但是你可以注意到还没有设置的pods。这叫做初始化。(获取更多信息请看[status](#)和[late initialization](#)).
- 一些字段被设置了默认值。一些默认值是根据cluster设置，一些默认值是API特定版本固定的。(获取更多信息请看 [defaulting](#)).

- 例如，`spec.containers[0].imagePullPolicy` 在api v1版本的默认值一直是`IfNotPresent`。
- 例如，`spec.containers[0].resources.limits.cpu` 在一些cluster可能默认设置为`100m`，其他cluster中是其他的值，并且不是在所有的cluster中都默认设置。API一般不会改变你已经设置的字段；它只会设置那些你没有定义的字段。

资源文档

你可以在[project website](#)或者[github](#)上浏览自动生成的API文档.

译者：安雪艳 校对：无

Kubernetes API参考

使用这些参考文档可学习如何通过REST API与Kubernetes交互。

你可以通过文档[API versioning](#)看到关于扩展API的详细信息.

内容列表:

- [Operations](#)
- [Definitions](#)
- Extensions API:
 - [Extensions: Operations](#)
 - [Extensions: Definitions](#)

运维API

译者：丁麒玮 校对：无

<!DOCTYPE html>

操作

获取可用资源

GET /api/v1

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- application/json

返回类型

- application/json

标签

- apiv1

列出组件状态信息

GET /api/v1/componentstatuses

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.ComponentStatusList

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的组件状态信息

```
GET /api/v1/componentstatuses/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	name	ComponentStatus 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ComponentStatus

传入参数

- /

Kubernetes中文文档

返回类型

- application/json

标签

- apiv1

列出或监听**Endpoint**对象

GET /api/v1/endpoints

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.EndpointsList

- /

返回类型

- application/json

标签

- apiv1

列出或监听**Event**对象

GET /api/v1/events

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.EventList

-
- /

返回类型

- application/json

标签

- apiv1

列出或监听**LimitRange**对象

GET /api/v1/limitranges

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.LimitRangeList

• /

返回类型

- application/json

标签

- apiv1

列出或监听**Namespace**对象

GET /api/v1/namespaces

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.NamespaceList

- /

返回类型

- application/json

标签

- apiv1

创建一个Namespace

POST /api/v1/namespaces

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Namespace	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- /

- application/json

标签

- apiv1

创建一个绑定

POST /api/v1/namespaces/{namespace}/bindings

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Binding	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Binding

传入参数

• /

Kubernetes中文文档

返回类型

- application/json

标签

- apiv1

列出或监听**Endpoints**对象

GET /api/v1/namespaces/{namespace}/endpoints

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码 Kubernetes中文文档	描述	结构
200	成功	v1.EndpointsList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Endpoint

POST /api/v1/namespaces/{namespace}/endpoints

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Endpoints	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Endpoints

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的Endpoint

```
GET /api/v1/namespaces/{namespace}/endpoints/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Endpoint 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Endpoints

传入参数

- `/`

返回类型

- `application/json`

标签

- `api/v1`

替换指定的Endpoint

```
PUT /api/v1/namespaces/{namespace}/endpoints/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Endpoints	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Endpoint 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Endpoints

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个Endpoint

Kubernetes中文文档

DELETE /api/v1/namespaces/{namespace}/endpoints/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Endpoint 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

- [api/v1](#)

部分更新指定的Endpoint

PATCH /api/v1/namespaces/{namespace}/endpoints/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Endpoint 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Endpoints

传入参数

- `application/json-patch+json`

- application/merge-patch+json

Kubernetes中文文档

- application/strategic-merge-patch+json
-

返回类型

- application/json

标签

- apiv1

列出或监听**Event**对象

GET /api/v1/namespaces/{namespace}/events

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
Kubernetes中文文档 200	成功	v1.EventList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Event

POST /api/v1/namespaces/{namespace}/events

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Event	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Event

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的Event

```
GET /api/v1/namespaces/{namespace}/events/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Event名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Event

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的Event

```
PUT /api/v1/namespaces/{namespace}/events/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Event	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Event名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Event

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个Event

Kubernetes中文文档

DELETE /api/v1/namespaces/{namespace}/events/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Event名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

• /

返回类型

- application/json

Kubernetes中文文档

标签

- apiv1

部分更新指定的Event

PATCH /api/v1/namespaces/{namespace}/events/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Event名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Event

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

列出或监听**LimitRange**对象

GET /api/v1/namespaces/{namespace}/limitranges

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
Kubernetes中文文档 200	成功	v1.LimitRangeList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个LimitRange

POST /api/v1/namespaces/{namespace}/limitranges

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.LimitRange	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.LimitRange

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的**LimitRange**

```
GET /api/v1/namespaces/{namespace}/limitranges/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	LimitRange 名字	是	string	

HTTP 状态码	描述	结构
200	成功	v1.LimitRange

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的**LimitRange**

```
PUT /api/v1/namespaces/{namespace}/limitranges/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.LimitRange	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	LimitRange 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.LimitRange

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个LimitRange

```
DELETE /api/v1/namespaces/{namespace}/limitranges/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化 格式打印。	不	string	
Body 参 数	body		是	v1.DeleteOptions	
Path 参 数	namespace	对象名字和 认证范围, 比如团队和 项目。	是	string	
Path 参 数	name	LimitRange 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的**LimitRange**

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	LimitRange 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.LimitRange

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

- [api/v1](#)

列出或监听**PersistentVolumeClaim**对象

GET /api/v1/namespaces/{namespace}/persistentvolumeclaims

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaimList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个**PersistentVolumeClaim**

```
POST /api/v1/namespaces/{namespace}/persistentvolumeclaims
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格 式打 印。	不	string	
Body 参数	body		是	v1.PersistentVolumeClaim	
Path 参 数	namespace	对象名 字和认 证范 围，比 如团队 和项 目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaim

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的PersistentVolumeClaim

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PersistentVolumeClaim 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaim

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的PersistentVolumeClaim

PUT /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}

参数

类别	名字	描述	必须	结构
Query 参数	pretty	如果为 <code>true</code> , 美化格式 打印。	不	string
Body 参数	body		是	v1.PersistentVolumeClaim
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string
Path 参数	name	PersistentVolumeClaim 名字	是	string

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaim

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个 PersistentVolumeClaim

```
DELETE /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PersistentVolumeClaim 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的**PersistentVolumeClaim**

PATCH /api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PersistentVolumeClaim 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaim

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

替换指定的**PersistentVolumeClaim**的状态

参数

类别	名字	描述	必须	结构
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string
Body 参数	body		是	v1.PersistentVolumeClaim
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string
Path 参数	name	PersistentVolumeClaim 名字	是	string

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaim

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听Pod对象

GET /api/v1/namespaces/{namespace}/pods

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
Kubernetes中文文档 200	成功	v1.PodList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Pod

POST /api/v1/namespaces/{namespace}/pods

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Pod	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的Pod

```
GET /api/v1/namespaces/{namespace}/pods/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的Pod

```
PUT /api/v1/namespaces/{namespace}/pods/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Pod	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个Pod

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

- [api/v1](#)

部分更新指定的Pod

PATCH /api/v1/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- application/json-patch+json
- Kubernetes中文文档
-
- application/merge-patch+json
 - application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

connect GET requests to attach of Pod

GET /api/v1/namespaces/{namespace}/pods/{name}/attach

参数

类别	名字	描述	必须	结构	默认
Query 参数	stdin	如果Stdin为true, 为这次请求重定向这个pod的标准输入流。默认为false。	不	boolean	
Query 参数	stdout	如果Stdout为true, 表明标准输出被重定向到这次连接请求。默认为true。	不	boolean	

Kubernetes 文档	Query 参数	stderr	如果 Stderr 为 true, 表明错误将被重定向到该连接请求。Stderr 默认值为 true。	不	boolean	
Query 参数	tty	TTY 如果为 true, 表明一个终端将被分配给该连接请求。这种请求贯穿整个容器的运行过程, 所以终端是在容器运行时分配到工作节点上。TTY 默认值为 false。		不	boolean	
Query 参数	container	执行命令的容器。如果 pod 只有一个容器, 默认认为这个容器		不	string	
Path 参数	namespace			是	string	

Kubernetes中文文档		对象名字 和认证范 围，比如 团队和项 目。			
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

connect POST requests to attach of Pod

POST /api/v1/namespaces/{namespace}/pods/{name}/attach

参数

类别	名字	描述	必须	结构	默认
Query 参 数	stdin	如果Stdin 为true, 为这次请 求重定向	不	boolean	

		这个pod 的标准输 入流。默 认为 false。			
Query 参 数	stdout	如果 Stdout为 true，表 明标准输 出被重定 向到这次 连接请 求。默认 为true。	不	boolean	
Query 参 数	stderr	如果 Stderr为 true，表 明错误将 被重定向 到该连接 请求。 Stderr默 认值为 true。	不	boolean	
Query 参 数	tty	TTY如果 为ture， 表明一个 终端将被 分配给该 连接请 求。这种 请求贯穿 整个容器 的运行过 程，所以 终端是在 容器运行	不	boolean	

Kubernetes中文文档		时分配到工作节点上。TTY默认值为false。		
Query 参数	container	执行命令的容器。如果pod只有一个容器，默认认为这个容器	不	string
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string
Path 参数	name	Pod名字	是	string

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- [api/v1](#)

Kubernetes中文文档

创建一个Binding的绑定

POST /api/v1/namespaces/{namespace}/pods/{name}/binding

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Binding	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Binding名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Binding

传入参数

- /

返回类型

标签

- apiv1

连接一个GET请求到执行exec命令的Pod

GET /api/v1/namespaces/{namespace}/pods/{name}/exec

参数

类别	名字	描述	必须	结构	默认
Query 参数	stdin	为这次访问重定向pod的标准输入流， 默认为false。	不	boolean	
Query 参数	stdout	为这次访问重定向pod的标准输出流， 默认为true。	不	boolean	
Query 参数	stderr	为这次访问重定向pod的标准错误流， 默认为true。	不	boolean	
Query 参数	tty	如果TTY为true， 表示将为这次exec请求分配	不	boolean	

		终端，默 认为 <code>false</code> 。			
Query 参数	container	执行命令 的容器。 如果pod 只有一个 容器，默 认为这个 容器。	不	string	
Query 参数	command	command 是一个远 程执行的 命令，参 数数组， 不是用 shell执行 的。	不	string	
Path 参数	namespace	对象名字 和认证范 围，比如 团队和项 目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

• /

标签

- apiv1

连接一个POST请求到执行exec命令的Pod

POST /api/v1/namespaces/{namespace}/pods/{name}/exec

参数

类别	名字	描述	必须	结构	默认
Query 参数	stdin	为这次访问重定向 pod 的标准输入流， 默认为 false。	不	boolean	
Query 参数	stdout	为这次访问重定向 pod 的标准输出流， 默认为 true。	不	boolean	
Query 参数	stderr	为这次访问重定向 pod 的标准错误流， 默认为 true。	不	boolean	
Query 参数	tty	如果 TTY 为 true， 表示将为	不	boolean	

Kubernetes中文文档		这次exec 请求分配 终端，默 认为 false。		
Query 参数	container	执行命令 的容器。 如果pod 只有一个 容器，默 认为这个 容器。	不	string
Query 参数	command	command 是一个远 程执行的 命令，参 数数组， 不是用 shell执行 的。	不	string
Path 参数	namespace	对象名字 和认证范 围，比如 团队和项 目。	是	string
Path 参数	name	Pod名字	是	string

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

• /

Kubernetes中文文档

返回类型

• /

标签

- apiv1

读取指定Pod的日志

GET /api/v1/namespaces/{namespace}/pods/{name}/log

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	container	流式返回log的容器, 如果pod只有一个容器, 默认为这个容器。	不	string	
Query 参数	follow	跟踪pod的流式log, 默认为 <code>false</code> 。	不	boolean	
Query 参数	previous	返回先前终止的容器日志, 默认为 <code>false</code> 。	不	boolean	
Query 参数	sinceSeconds	一个比当前时间早的时间值来展示日志。如果值提前于这个pod的创建	不	integer (int32)	

		时间，只返回从pod创建时间开始的日志。如果值在将来，没有日志返回。只有指定的一段特定的时间内。		
Query 参数	sinceTime	从一个RFC3339时间戳的时间来展示日志，如果值提前于这个pod的创建时间，只返回从pod创建时间开始的日志。如果值在将来，没有日志返回。只有指定的一段特定的时间。	不	string
Query 参数	timestamps	如果为true，在log输出每行首加上RFC3339或者RFC3339Nano时间戳，默认为false。	不	boolean
Query 参数	tailLines	如果设置，输入从日志尾起的数条日志，如果没指定，日志将从创建容器或者一段	不	integer (int32)

Kubernetes中文文档		时间以前或者一个日期之后起输出。		
Query 参数	limitBytes	如果设置，将会在终止日志输出前输出一定字节。这将不会完全展示最后一行日志，并且轻微的大于或小于指定限制。	不	integer (int32)
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string
Path 参数	name	Pod名字	是	string

响应

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- /

返回类型

- application/json

标签

- apiv1

连接一个GET请求到端口转发的Pod

Kubernetes中文文档

GET /api/v1/namespaces/{namespace}/pods/{name}/portforward

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个POST请求到端口转发的Pod

POST /api/v1/namespaces/{namespace}/pods/{name}/portforward

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个GET请求到代理的Pod

```
GET /api/v1/namespaces/{namespace}/pods/{name}/proxy
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个PUT请求到代理的Pod

```
PUT /api/v1/namespaces/{namespace}/pods/{name}/proxy
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个**DELETE**请求到代理的Pod

```
DELETE /api/v1/namespaces/{namespace}/pods/{name}/proxy
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个**POST**请求到代理的Pod

```
POST /api/v1/namespaces/{namespace}/pods/{name}/proxy
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接一个GET请求到代理的Pod

```
GET /api/v1/namespaces/{namespace}/pods/{name}/proxy/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接PUT请求到代理的Pod

```
PUT /api/v1/namespaces/{namespace}/pods/{name}/proxy/{path: *}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接DELETE请求到代理的Pod

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

连接POST请求到代理的Pod

Kubernetes中文文档

POST /api/v1/namespaces/{namespace}/pods/{name}/proxy/{path: *}

参数

类别	名字	描述	必须	结构	默认
Query 参数	path	path是把请求代理到pod的url路径。	不	string	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- [api/v1](#)

Kubernetes中文文档

替换指定的Pod的状态

PUT /api/v1/namespaces/{namespace}/pods/{name}/status

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Pod	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Pod

传入参数

- /

返回类型

- application/json

Kubernetes中文文档

标签

- apiv1

列出或监听**PodTemplate**对象

GET /api/v1/namespaces/{namespace}/podtemplates

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplateList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个PodTemplate

POST /api/v1/namespaces/{namespace}/podtemplates

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.PodTemplate	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplate

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的PodTemplate

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PodTemplate名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplate

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的PodTemplate

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.PodTemplate	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PodTemplate名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplate

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个PodTemplate

Kubernetes中文文档

DELETE /api/v1/namespaces/{namespace}/podtemplates/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	PodTemplate名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- `api/v1`

Kubernetes中文文档

部分更新指定的PodTemplate

`PATCH /api/v1/namespaces/{namespace}/podtemplates/{name}`

参数

类别	名字	描述	必须	结构	默认
Query 参数	<code>pretty</code>	如果为 <code>true</code> , 美化格式打印。	不	<code>string</code>	
Body 参数	<code>body</code>		是	unversioned.Patch	
Path 参数	<code>namespace</code>	对象名字和认证范围, 比如团队和项目。	是	<code>string</code>	
Path 参数	<code>name</code>	PodTemplate名字	是	<code>string</code>	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplate

传入参数

- `application/json-patch+json`
- `application/merge-patch+json`
- `application/strategic-merge-patch+json`

-
- application/json

标签

- apiv1

列出或监听**ReplicationController**对象

```
GET /api/v1/namespaces/{namespace}/replicationcontrollers
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationControllerList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个**ReplicationController**

```
POST /api/v1/namespaces/{namespace}/replicationcontrollers
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ReplicationController	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationController

传入参数

- `/`

返回类型

- `application/json`

标签

- `apiVersion`

读取指定的ReplicationController

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ReplicationController 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationController

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的ReplicationController

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ReplicationController	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ReplicationController 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationController

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个ReplicationController

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ReplicationController 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- api/v1

部分更新指定的ReplicationController

Kubernetes中文文档

PATCH /api/v1/namespaces/{namespace}/replicationcontrollers/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ReplicationController 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationController

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

- [api/v1](#)

替换指定的ReplicationController的状态

PUT /api/v1/namespaces/{namespace}/replicationcontrollers/{name}/status

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ReplicationController	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ReplicationController 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ReplicationController

传入参数

- [/](#)

返回类型

- application/json

Kubernetes中文文档

标签

- apiv1

列出或监听**ResourceQuota**对象

GET /api/v1/namespaces/{namespace}/resourcequotas

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuotaList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个**ResourceQuota**

POST /api/v1/namespaces/{namespace}/resourcequotas

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ResourceQuota	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuota

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的ResourceQuota

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ResourceQuota 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuota

传入参数

- /

返回类型

- application/json

标签

- api/v1

替换指定的ResourceQuota

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ResourceQuota	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ResourceQuota 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuota

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个ResourceQuota

```
DELETE /api/v1/namespaces/{namespace}/resourcequotas/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ResourceQuota 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的ResourceQuota

```
PATCH /api/v1/namespaces/{namespace}/resourcequotas/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ResourceQuota 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuota

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

替换指定的ResourceQuota的状态

Kubernetes中文文档

PUT /api/v1/namespaces/{namespace}/resourcequotas/{name}/status

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ResourceQuota	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ResourceQuota名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuota

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听Secret对象

Kubernetes中文文档

GET /api/v1/namespaces/{namespace}/secrets

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
Kubernetes中文文档 200	成功	v1.SecretList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Secret

POST /api/v1/namespaces/{namespace}/secrets

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Secret	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Secret

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的Secret

```
GET /api/v1/namespaces/{namespace}/secrets/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Secret名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Secret

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的Secret

```
PUT /api/v1/namespaces/{namespace}/secrets/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Secret	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Secret名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Secret

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个Secret

Kubernetes中文文档

DELETE /api/v1/namespaces/{namespace}/secrets/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Secret名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

- [api/v1](#)

部分更新指定的Secret

PATCH /api/v1/namespaces/{namespace}/secrets/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Secret名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Secret

传入参数

- application/json-patch+json
- Kubernetes中文文档
-
- application/merge-patch+json
 - application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

列出或监听**ServiceAccount**对象

GET /api/v1/namespaces/{namespace}/serviceaccounts

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccountList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个**ServiceAccount**

POST /api/v1/namespaces/{namespace}/serviceaccounts

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ServiceAccount	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccount

传入参数

- `/`

返回类型

- `application/json`

标签

- `apiVersion`

读取指定的**ServiceAccount**

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ServiceAccount 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccount

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的ServiceAccount

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.ServiceAccount	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ServiceAccount 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccount

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个ServiceAccount

```
DELETE /api/v1/namespaces/{namespace}/serviceaccounts/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ServiceAccount 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的ServiceAccount

```
PATCH /api/v1/namespaces/{namespace}/serviceaccounts/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	ServiceAccount 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccount

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

列出或监听**Service**对象

Kubernetes中文文档

GET /api/v1/namespaces/{namespace}/services

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
Kubernetes中文文档 200	成功	v1.ServiceList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Service

POST /api/v1/namespaces/{namespace}/services

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Service	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

HTTP 状态码	描述	结构
200	成功	v1.Service

传入参数

- /

返回类型

- application/json

标签

- apiv1

读取指定的**Service**

```
GET /api/v1/namespaces/{namespace}/services/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Service

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的Service

```
PUT /api/v1/namespaces/{namespace}/services/{name}
```

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Service	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Service

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个Service

Kubernetes中文文档

DELETE /api/v1/namespaces/{namespace}/services/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- [apiv1](#)

Kubernetes中文文档

部分更新指定的Service

PATCH /api/v1/namespaces/{namespace}/services/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	
Path 参数	name	Service 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Service

传入参数

- `application/json-patch+json`
- `application/merge-patch+json`

- application/strategic-merge-patch+json

Kubernetes中文文档

返回类型

- application/json

标签

- apiv1

读取指定的Namespace

GET /api/v1/namespaces/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- /

返回类型

- application/json

- [api/v1](#)

替换指定的Namespace

PUT /api/v1/namespaces/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Namespace	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- [/](#)

返回类型

- application/json

标签

- [api/v1](#)

Kubernetes中文文档

删除一个Namespace

`DELETE /api/v1/namespaces/{name}`

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- `/`

返回类型

- `application/json`

标签

- [api/v1](#)

部分更新指定的Namespace

Kubernetes中文文档

PATCH /api/v1/namespaces/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

替换指定的Namespace为最终版

Kubernetes中文文档

PUT /api/v1/namespaces/{name}/finalize

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Namespace	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- /

返回类型

- application/json

标签

- apiV1

替换指定的Namespace的状态

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Namespace	
Path 参数	name	Namespace 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Namespace

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听**Node**对象

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.NodeList

传入参数

- /

返回类型

- application/json

标签

- apiv1

创建一个Node

POST /api/v1/nodes

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <i>true</i> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Node	

响应

HTTP 状态码	描述	结构
200	成功	v1.Node

- /

返回类型

- application/json

标签

- apiv1

读取指定的Node

GET /api/v1/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Node

传入参数

- /

返回类型

- application/json

Kubernetes中文文档

标签

- apiv1

替换指定的Node

PUT /api/v1/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Node	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Node

传入参数

- /

返回类型

- application/json

标签

- apiv1

Kubernetes中文文档

删除一个Node

DELETE /api/v1/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	<code>v1.DeleteOptions</code>	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	<code>unversioned.Status</code>

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的Node

Kubernetes中文文档

PATCH /api/v1/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Node

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- [apiv1](#)

Kubernetes中文文档

替换指定的**Node**的状态

`PUT /api/v1/nodes/{name}/status`

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.Node	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.Node

传入参数

- `/`

返回类型

- `application/json`

标签

- [apiv1](#)

列出或监听**PersistentVolumeClaim**对象

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeClaimList

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听**PersistentVolume**对象

```
GET /api/v1/persistentvolumes
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolumeList

- /

返回类型

- application/json

标签

- apiv1

创建一个**PersistentVolume**

POST /api/v1/persistentvolumes

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.PersistentVolume	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolume

传入参数

- /

- application/json

标签

- apiv1

读取指定的PersistentVolume

GET /api/v1/persistentvolumes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolume

传入参数

- /

返回类型

- application/json

标签

- apiv1

替换指定的PersistentVolume

Kubernetes中文文档

PUT /api/v1/persistentvolumes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.PersistentVolume	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolume

传入参数

- /

返回类型

- application/json

标签

- apiv1

删除一个PersistentVolume

DELETE /api/v1/persistentvolumes/{name}

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.DeleteOptions	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	unversioned.Status

传入参数

- /

返回类型

- application/json

标签

- apiv1

部分更新指定的**PersistentVolume**

PATCH /api/v1/persistentvolumes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	unversioned.Patch	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolume

传入参数

- application/json-patch+json
- application/merge-patch+json
- application/strategic-merge-patch+json

返回类型

- application/json

标签

- apiv1

替换指定的**PersistentVolume**的状态

PUT /api/v1/persistentvolumes/{name}/status

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Body 参数	body		是	v1.PersistentVolume	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	v1.PersistentVolume

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听Pod对象

GET /api/v1/pods

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodList

• /

返回类型

- application/json

标签

- apiv1

列出或监听**PodTemplate**对象

GET /api/v1/podtemplates

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.PodTemplateList

- /

返回类型

- application/json

标签

- apiv1

代理GET请求到Pod

GET /api/v1/proxy/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

• /

标签

• apiv1

代理PUT请求到Pod

PUT /api/v1/proxy/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

• /

返回类型

• /

- apiv1

代理DELETE请求到Pod

DELETE /api/v1/proxy/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理POST请求到Pod

Kubernetes中文文档

POST /api/v1/proxy/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理GET请求到Pod

GET /api/v1/proxy/namespaces/{namespace}/pods/{name}/{path: *}

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理PUT请求到Pod

```
PUT /api/v1/proxy/namespaces/{namespace}/pods/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理**DELETE**请求到Pod

```
DELETE /api/v1/proxy/namespaces/{namespace}/pods/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理POST请求到Pod

```
POST /api/v1/proxy/namespaces/{namespace}/pods/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理GET请求到Service

```
GET /api/v1/proxy/namespaces/{namespace}/services/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理PUT请求到Service

```
PUT /api/v1/proxy/namespaces/{namespace}/services/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理**DELETE**请求到**Service**

```
DELETE /api/v1/proxy/namespaces/{namespace}/services/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理**POST**请求到**Service**

```
POST /api/v1/proxy/namespaces/{namespace}/services/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理GET请求到Service

```
GET /api/v1/proxy/namespaces/{namespace}/services/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理PUT请求到Service

```
PUT /api/v1/proxy/namespaces/{namespace}/services/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理**DELETE**请求到**Service**

```
DELETE /api/v1/proxy/namespaces/{namespace}/services/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理POST请求到Service

```
POST /api/v1/proxy/namespaces/{namespace}/services/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理GET请求到Node

```
GET /api/v1/proxy/nodes/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理PUT请求到Node

```
PUT /api/v1/proxy/nodes/{name}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

- /

返回类型

- /

标签

- apiv1

代理**DELETE**请求到**Node**

`DELETE /api/v1/proxy/nodes/{name}`

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理POST请求到Node

Kubernetes中文文档

POST /api/v1/proxy/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理GET请求到Node

GET /api/v1/proxy/nodes/{name}/{path: *}

参数

Kubernetes中文文档

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理PUT请求到Node

```
PUT /api/v1/proxy/nodes/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

代理**DELETE**请求到**Node**

```
DELETE /api/v1/proxy/nodes/{name}/{path: *}
```

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

- /

标签

- apiv1

代理POST请求到Node

POST /api/v1/proxy/nodes/{name}/{path: *}

参数

类别	名字	描述	必须	结构	默认
Path 参数	name	Node名字	是	string	
Path 参数	path	资源路径	是	string	

响应

HTTP 状态码	描述	结构
default	成功	string

传入参数

- /

返回类型

- /

标签

- apiv1

列出或监听ReplicationController对象

Kubernetes中文文档

GET /api/v1/replicationcontrollers

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

HTTP 状态码	描述	结构
200	成功	v1.ReplicationControllerList

传入参数

- /

返回类型

- application/json

标签

- apiv1

列出或监听**ResourceQuota**对象

GET /api/v1/resourcequotas

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.ResourceQuotaList

• /

返回类型

- application/json

标签

- apiv1

列出或监听**Secret**对象

GET /api/v1/secrets

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.SecretList

-
- /

返回类型

- application/json

标签

- apiv1

列出或监听**ServiceAccount**对象

GET /api/v1/serviceaccounts

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceAccountList

• /

返回类型

- application/json

标签

- apiv1

列出或监听**Service**对象

GET /api/v1/services

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	v1.ServiceList

• /

返回类型

- application/json

标签

- apiv1

监听一列**Endpoints**里面的个别变化

GET /api/v1/watch/endpoints

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一列**Event**里面的个别变化

GET /api/v1/watch/events

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一列**LimitRange**里面的个别变化

GET /api/v1/watch/limitranges

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一列**Namespace**里面的个别变化

GET /api/v1/watch/namespaces

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一列**Endpoints**里面的个别变化

GET /api/v1/watch/namespaces/{namespace}/endpoints

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**Endpoints**对象的变化

GET /api/v1/watch/namespaces/{namespace}/endpoints/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Endpoint名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一列**Event**里面的个别变化

GET /api/v1/watch/namespaces/{namespace}/events

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**Event**对象的变化

GET /api/v1/watch/namespaces/{namespace}/events/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Event名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- apiv1

监听一列**LimitRange**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/limitranges
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**LimitRange**对象的变化

GET /api/v1/watch/namespaces/{namespace}/limitranges/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	LimitRange名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一列**PersistentVolumeClaim**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/persistentvolumeclaims
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**PersistentVolumeClaim**对象的变化

```
GET /api/v1/watch/namespaces/{namespace}/persistentvolumeclaims/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	PersistentVolumeClaim名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一列**Pod**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/pods
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个Pod对象的变化

GET /api/v1/watch/namespaces/{namespace}/pods/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, labelSelector默认值为everything	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, fieldSelector默认值为everything	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Pod名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一列**PodTemplate**里面的个别变化

GET /api/v1/watch/namespaces/{namespace}/podtemplates

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**PodTemplate**对象的变化

GET /api/v1/watch/namespaces/{namespace}/podtemplates/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	PodTemplate名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- apiv1

监听一列**ReplicationController**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/replicationcontrollers
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码 Kubernetes中文文档	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**ReplicationController**对象的变化

GET /api/v1/watch/namespaces/{namespace}/replicationcontrollers/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer(int32)	
Path参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path参数	name	ReplicationController名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- apiv1

监听一列**ResourceQuota**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/resourcequotas
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**ResourceQuota**对象的变化

GET /api/v1/watch/namespaces/{namespace}/resourcequotas/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	ResourceQuota名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- apiv1

监听一列**Secret**里面的个别变化

GET /api/v1/watch/namespaces/{namespace}/secrets

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**Secret**对象的变化

GET /api/v1/watch/namespaces/{namespace}/secrets/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Secret名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一列**ServiceAccount**里面的个别变化

GET /api/v1/watch/namespaces/{namespace}/serviceaccounts

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**ServiceAccount**对象的变化

GET /api/v1/watch/namespaces/{namespace}/serviceaccounts/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	ServiceAccount名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一列**Service**里面的个别变化

```
GET /api/v1/watch/namespaces/{namespace}/services
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围, 比如团队和项目。	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一个**Service**对象的变化

GET /api/v1/watch/namespaces/{namespace}/services/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch		不	boolean	

Kubernetes中文文档		监听指定资源的变化，并且以流的形式返回添加、更新、删除等变化的通知。需指定resourceVersion。			
Query 参数	resourceVersion	当指定一个监听函数调用时，它会显示资源在某个特定版本之后的变化。默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	namespace	对象名字和认证范围，比如团队和项目。	是	string	
Path 参数	name	Service名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

- [api/v1](#)

监听一个**Namespace**对象的变化

GET /api/v1/watch/namespaces/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	name	Namespace名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一列**Node**里面的个别变化

GET /api/v1/watch/nodes

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

-
- /

返回类型

- application/json

标签

- apiv1

监听一个**Node**对象的变化

GET /api/v1/watch/nodes/{name}

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	
Path 参数	name	Node名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一列**PersistentVolumeClaim**里面的个别变化

GET /api/v1/watch/persistentvolumeclaims

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

监听一列**PersistentVolume**里面的个别变化

GET /api/v1/watch/persistentvolumes

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一个**PersistentVolume**对象的变化

```
GET /api/v1/watch/persistentvolumes/{name}
```

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	
Path 参数	name	PersistentVolume 名字	是	string	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

传入参数

- /

返回类型

- application/json

标签

- apiv1

监听一列**Pod**里面的个别变化

GET /api/v1/watch/pods

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

- /

返回类型

- application/json

标签

- apiv1

监听一列**PodTemplate**里面的个别变化

GET /api/v1/watch/podtemplates

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

监听一列**ReplicationController**里面的个别变化

GET /api/v1/watch/replicationcontrollers

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

监听一列**ResourceQuota**里面的个别变化

GET /api/v1/watch/resourcequotas

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

监听一列**Secret**里面的个别变化

GET /api/v1/watch/secrets

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	list/watch 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

监听一列**ServiceAccount**里面的个别变化

GET /api/v1/watch/serviceaccounts

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

-
- /

返回类型

- application/json

标签

- apiv1

监听一列**Service**里面的个别变化

GET /api/v1/watch/services

参数

类别	名字	描述	必须	结构	默认
Query 参数	pretty	如果为 <code>true</code> , 美化格式打印。	不	string	
Query 参数	labelSelector	选择器通过label来约束返回结果, <code>labelSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	fieldSelector	选择器通过field来约束返回结果, <code>fieldSelector</code> 默认值为 <code>everything</code>	不	string	
Query 参数	watch	监听指定资源的变化, 并且以流的形式返回添加、更新、删除等变化的通知。需指定 <code>resourceVersion</code> 。	不	boolean	
Query 参数	resourceVersion	当指定一个监听函数调用时, 它会显示资源在某个特定版本之后的变化。 默认为与起始版本相比的变化。	不	string	
Query 参数	timeoutSeconds	<code>list/watch</code> 请求的超时时间。	不	integer (int32)	

响应

HTTP 状态码	描述	结构
200	成功	json.WatchEvent

• /

返回类型

- application/json

标签

- apiv1

Last updated 2015-11-06 18:46:00 UTC

参数说明

译者：赵帅龙 校对：无

- [v1.Pod](#)
- [v1.PodList](#)
- [v1.PodTemplate](#)
- [v1.PodTemplateList](#)
- [v1.ReplicationController](#)
- [v1.ReplicationControllerList](#)
- [v1.Service](#)
- [v1.ServiceList](#)
- [v1.Endpoints](#)
- [v1.EndpointsList](#)
- [v1.Node](#)
- [v1.NodeList](#)
- [v1.Binding](#)
- [v1.Event](#)
- [v1.EventList](#)
- [v1.LimitRange](#)
- [v1.LimitRangeList](#)
- [v1.ResourceQuota](#)
- [v1.ResourceQuotaList](#)
- [v1.Namespace](#)
- [v1.NamespaceList](#)
- [v1.Secret](#)

- [v1.SecretList](#)

Kubernetes中文文档

-
- [v1.ServiceAccount](#)

- [v1.ServiceAccountList](#)

- [v1.PersistentVolume](#)

- [v1.PersistentVolumeList](#)

- [v1.PersistentVolumeClaim](#)

- [v1.PersistentVolumeClaimList](#)

- [v1.DeleteOptions](#)

- [v1.ComponentStatus](#)

- [v1.ComponentStatusList](#)
-

Definitions

v1.Node

Node，也叫minion，在kubernetes中是工作节点。在缓存中（比如etcd），每个节点是唯一的。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind (类型)	Kind 是指当前对象代表的REST资源类型。 client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息， apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象的元数据(metadata)。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
spec	Spec定义了一个节点的行为。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	v1.NodeSpec
status	当前节点的最新状态。由系统生成，只读。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	v1.NodeStatus

v1.PersistentVolumeClaimList

PersistentVolumeClaimList是一个列表，包含多个 PersistentVolumeClaim条目。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	<p>一组持久存储卷声明。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistentvolumeclaims</p>	true	v1.PersistentVolumeClaim array

v1.ObjectFieldSelector

ObjectFieldSelector 用来选择对象的某个APIVersioned字段。

Name	Kubernetes中文文档	描述	是否必须	类型	默认值
apiVersion		我们用某个类型去呈现 FieldPath, apiVersion指类型的版本号, 默认是"v1"。	false	string	
fieldPath		针对某个特定的版本的 API, 要选择的字段的路径。	true	string	

v1.SELinuxOptions

SELinuxOptions是指要赋给容器的标签(labels)。

Name	描述	是否必须	类型	默认值
user	User是指要赋给容器的SELinux用户标签。	false	string	
role	Role是指要赋给容器的SELinux角色标签。	false	string	
type	Type是指要赋给容器的SELinux类型标签。	false	string	
level	Level是指要赋给容器的SELinux级别标签。	false	string	

v1.ContainerStateRunning

ContainerStateRunning指容器"运行中"的状态

Name	描述	是否必须	类型	默认值
startedAt	容器上一次启动或者重启的时间	false	string	

v1.VolumeMount

VolumeMount用来描述一个容器数据卷的挂载。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档				
name	这个字段必须与数据卷的名字相匹配。	true	string	
readOnly	如果挂载时设置为只读，则该字段为 true，否则为可读写(false 或者 unspecified)。默认值为 false。	false	boolean	false
mountPath(挂载路径)	指定数据卷在容器内的挂载路径。	true	string	

v1.PersistentVolumeClaimSpec

PersistentVolumeClaimSpec用来描述存储设备的常见属性，允许针对某个特定供应商的设备自定义属性。

Name Kubernetes中文文档	描述	是否 必须	类型
accessModes (访问模式)	AccessModes 包含了数据卷应具备的访问模式。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#access-modes-1	false	v1.PersistentVolumeAccessModes array
resources (资源)	Resources代表存储卷支持的最小资源。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#resources	false	v1.ResourceRequirements
volumeName (数据卷名字)	VolumeName是持久存储卷的一个引用。	false	string

v1.CephFSType

CephFSType表示一次Ceph文件系统挂载，其生命周期与挂载到的pod一致。

Name	描述	是否必须
monitors (监控器)	必须字段。monitors是Ceph monitor的集合。详情查看： http://releases.k8s.io/HEAD/examples/cephfs/README.md#how-to-use-it	true
user	可选字段。User是一个rados用户名称，默认为admin。详情查看： http://releases.k8s.io/HEAD/examples/cephfs/README.md#how-to-use-it	false
secretFile	可选字段。SecretFile是用户秘钥文件的路径，默认为/etc/ceph/user.secret。详情查看： http://releases.k8s.io/HEAD/examples/cephfs/README.md#how-to-use-it	false
secretRef	可选字段。SecretRef是用户身份认证的秘钥，默认为空。详情查看： http://releases.k8s.io/HEAD/examples/cephfs/README.md#how-to-use-it	false
readOnly	可选字段。默认为false (read/write)。该选项保证挂载数据卷时将其设置为只读。详情查看： http://releases.k8s.io/HEAD/examples/cephfs/README.md#how-to-use-it	false

v1.DownwardAPIVolumeSource

DownwardAPIVolumeSource代表一个包含了下行API信息的数据卷。

Name	描述	是否必须	类型	默认值
items	Items是下行API数据卷文件的列表。	false	v1.DownwardAPIVolumeFile array	

unversioned.StatusCause

StatusCause提供api.Status的失败信息，其中包括一些同时出现多种错误的情况。

Name	描述	是否必须	类型	默认值
reason	机器可识别的错误原因描述。如果该字段的值为空，表示没有可用的错误信息。	false	string	
message	人可识别的错误原因描述。从人的角度出发，该字段具有很大的可读性。	false	string	
field	<p>造成当前错误的资源的字段，一般以JSON方式序列化后的字符串命名。对于嵌套结构，可能会包含点(dot)和后缀表示法。数组下标从0开始计数。如果某个字段造成多个错误，那么该字段可能会在数组中出现多次。该字段为可选字段。</p> <p>举个栗子：</p> <p>"name" - 当前资源的"name"字段</p> <p>"items[0].name" - 数组"items"字段的第一个值的"name"字段。</p>	false	string	

v1.GCEPersistentDiskVolumeSource

Kubernetes中文文档

GCEPersistentDiskVolumeSource 代表Google Compute Engine上的一块持久存储盘。

GCE持久存储盘(简称GCE PD)必须存在，且在被挂载到容器之前必须首先被格式化。该存储盘必须和kubelet存在于同一个GCE项目和区域中。GCE PD以可读/可写模式只能被挂载一次。

Name	描述	是否必须	类型	默认值
pdName	GCE中PD资源的唯一名字，用来识别GCE中的磁盘。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk	true	string	
fsType	要挂载的存储卷的文件类型。提示：必须保证宿主操作系统支持存储卷的文件类型，比如ext4, xfs, ntfs。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk	true	string	
partition	你要挂载的存储卷分区。默认根据存储卷名称挂载。比如，对于存储卷/dev/sda1，你指定了分区为"1"。类似的，存储卷/dev/sda的对应分区为"0"。你也可以将该字段留空。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk	false	integer (int32)	
readOnly	该选项保证挂载数据卷时将其设置为只读。默认为false (read/write)。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk	false	boolean	false

v1.ResourceQuotaSpec

ResourceQuotaSpec定义了资源限额上的硬性限制。

Name	描述
hard	Hard 是指对于每一个已命名资源的硬性限制。详情查看： http://releases.k8s.io/HEAD/docs/design/admission_control_resource_quota.md#admissionplugin-resourcequota

v1.NamespaceStatus

NamespaceStatus指一个命名空间的当前状态信息。

Name	描述	是否必须
phase	Phase是指命名空间在整个生命周期中所处的阶段。详情查看： http://releases.k8s.io/HEAD/docs/design/namespaces.md#phases	false

v1.NamespaceSpec

NamespaceSpec描述了一个命名空间的属性。

Name	描述	
finalizers	Finalizers是一组变量，如果一个对象要从存储中删除，那么对应的变量必须被先置为空。详情查看： http://releases.k8s.io/HEAD/docs/design/namespaces.md#finalizers	

v1.PersistentVolume

PersistentVolume (PV)是由管理员设置的存储资源，这些存储资源对于不同的节点来说并没有太大差别(跨节点)。详情查看：<http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md>

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。 client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象的元数据。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
spec	Spec 定义了集群拥有的持久存储卷的规格，是由管理员提供的。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistent-volumes	false	v1.PersistentVolumeSpec
status	Status 表示持久存储卷的当前状态和信息。它是由系统生成的，是只读字段。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistent-volumes	false	v1.PersistentVolumeStatus

v1.PersistentVolumeStatus

Kubernetes中文文档

PersistentVolumeStatus表示一个持久存储卷的当前状态。

Name	描述	是否必须	类型	默认值
phase	Phase表示存储卷处于哪个阶段。比如"可用 (available)"、"绑定到一个持久存储卷声明 (bound to a claim)"和"已释放 (released by a claim)"。详情查 看 : http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#phase	false	string	
message	人可读写的一段信息，用于显示储存卷之所以处于当前状态的原因。	false	string	
reason	Reason是一个简单的驼峰式字符串，用于显示在命令行中，它用于描述出现的任何错误。程序可解析该字段的值。	false	string	

v1.EndpointsList

EndpointsList是一组endpoints。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
items	endpoint列表。	true	v1.Endpoints array

v1.GitRepoVolumeSource

GitRepoVolumeSource是pod创建时从git仓库下载的一个存储卷。

Name	描述	是否必须	类型	默认值
repository	Repository URL	true	string	
revision	特定版本代码提交的hash值。	true	string	

v1.Capabilities

从运行中的容器添加和删除POSIX capabilities。

Name	描述	是否必须	类型	默认值
add	已添加的capabilities	false	v1.Capability array	
drop	已删除的capabilities	false	v1.Capability array	

v1.PodTemplateList

PodTemplateList是一组Pod模板 (PodTemplates)。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	一组Pod模板	true	v1.PodTemplate array

v1.NodeCondition

NodeCondition表示一个节点的状态信息。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
type	节点状态信息的类型，目前只有 Ready	true	string	
status	节点状态的状态值，与状态类型 (type)一一对应，可以是 True, False 或 Unknown。	true	string	
lastHeartbeatTime	最后一次状态更新的时间。	false	string	
lastTransitionTime	最后一次状态切换的时间。	false	string	
reason	(简单表述)最后一次状态切换的原因。	false	string	
message	人可阅读的一段信息，显示最后一次状态切换的详细信息。	false	string	

v1.LocalObjectReference

LocalObjectReference包含了一些寻址信息，通过这个字段的值你可以在相同的
Kubernetes中文文档 namespace下找到它指向的对象。

Name	描述	是否必须	类型	默认值
name	被指向对象的名字。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#names	false	string	

v1.ResourceQuotaStatus

ResourceQuotaStatus定义了资源的硬性限制和使用率。

Name	描述
hard	Hard是一组应用到每个已命名资源的硬性限制。详情查 看： http://releases.k8s.io/HEAD/docs/design/admission_control_resource_quota.md#admission-plugin-resourcequota
used	Used是当前命名空间下总的资源使用率。

v1.ExecAction

ExecAction描述了一个"run in container"的动作。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档	command	Command是一个要在容器中运行的命令，运行的当前目录为容器文件系统的根目录(/)。命令并不是在shell中执行的，所以传统的shell命令可能不好用。如果需要的话，你必须显式地调用shell。返回值0表示live/healthy，非零表示unhealthy。	false	string array

v1.ObjectMeta

ObjectMeta表示所有持久资源必须有的元数据，它包含了用户必须创建的所有对象。

Name	描述	是否必须
name	Name在一个命名空间内必须是唯一的。虽然在创建资源时，有些允许客户端请求自动生成名字，但也必须填写这个字段。Name最初是为了保证创建操作的幂等性和保证配置可定义设计的。它一旦确定便无法更改。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#names	false
generateName		false

	<p>GenerateName是一个可选的前缀名字，当Name字段没有填写时，server使用它生成一个唯一的Name。如果使用这个字段，客户端传过来的名字将会失效，因为server会返回一个不同的Name。这个值也会和唯一的后缀一起使用。这个字段的检查规则与Name字段一致，由于suffix的长度限制，可能被截断，以保证在server端的唯一性</p> <p>如果这个字段已经被指定，generated name存在，server不会返回409错误，而是返回201 created或者500 server timeout。500错误意味着在允许的时间内无法确定唯一名字，client端应该重试(这是一个可选操作，只有在Retry-After header中设置了时间才会触发)。</p> <p>只有没有指定Name字段时，该字段才有效。详情查看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#idempotency</p>	
namespace	<p>Namespace定义了一个空间，在该空间内，名字(Name)必须是唯一的。一个空("")的命名空间和"default"命名空间是等价的，然而"default"命名空间是标准命名空间，并不是所有的对象都被必须放到该命名空间内。 "default"命名空间中的对象的该字段为空。</p> <p>必须是DNS_LABEL。该字段一旦确定不能修改。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/namespaces.md</p>	false
selfLink	<p>SelfLink是一个代表这个对象本身的URL。该字段由系统自动生成，是一个只读字段。</p>	false
uid	<p>UID表示当前对象在任何时间和空间中都是唯一的。该字段在资源创建成功后由系统自动生成，PUT操作不会改变该字段的值。</p>	false

	<p>该字段由系统生成的，是一个只读字段。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#uids</p>	
resourceVersion	<p>用来表示当前对象的内部版本号，client端根据这个字段来判断对象是否已经发生变化。它也能用在optimistic concurrency，变化检测和监控外界对一个或者一组资源做了哪些操作。client端必须显式地对待这些值，并且不做任何修改地传递给server端。这个字段可能只对特定资源有效。</p> <p>该字段由系统生成的，是一个只读字段。客户端必须显式地(opaque)对待该字段(tbd)。详情查看：http://releases.k8s.io/HEAD/docs-devel/api-conventions.md#concurrency-control-and-consistency</p>	false
generation	<p>一个序列号，它表示期望状态的一次生成(generation)。目前只有replication controllers支持。该字段由系统生成的，是一个只读字段。</p>	false
creationTimestamp	<p>CreationTimestamp代表该对象在server端生成的时间。前后独立的两次创建操作并不能保证该字段也是有序的。client端不能设置这个字段。显示格式遵循RFC3339规范，以UTC方式显示。</p> <p>该字段由系统生成的，是一个只读字段。对于列表类型，该字段为空。详情查看：http://releases.k8s.io/HEAD/docs-devel/api-conventions.md#metadata</p>	false
deletionTimestamp	<p>DeletionTimestamp是只资源(要)被删除的时间，由遵循RFC3339格式。用户请求删除资源时，服务器端生成该字段，生成以后client端不能直接修改。到该字段指定的时间以后，资源会被删除，删除以后 资源列表中将看不到该资源，也无法通过名字检索该资源。该字段一旦设置，该字段只能被修改为已指定时间之前的某个时间，</p>	false

	<p>也可以在已指定时间之前删除该资源，而不能取消或者修改为已指定时间之后的时间。举个例子，一个用户请求在30s内删除一个pod，对应的kubelet会向pod内的所有容器发送一个优雅的termination信号，一旦pod资源被删除，kubelet会发送强制的termination信号。如果不设置该字段，将不向容器发送termination信号而是直接终止。</p> <p>当server接收到优雅删除的请求时，系统会自动生成该字段。该字段是只读的。详情查看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	
deletionGracePeriodSeconds	在被系统删除之前，该对象允许被优雅删除的时间，以秒为单位。只有设置deletionTimestamp字段后，该字段才生效。该字段只能被缩短，不能取消设置或者延长。该字段是只读字段。	false
labels	labels用来对对象进行组织和分类，类型为map[string]string。它与replication controller和services的选择器(selectors)检查匹配。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/labels.md	false
annotations	Annotations是一种非结构化的map[string]string，可以通过kubernetes系统外的工具对其进行修改和读取。不能根据这个字段执行查询操作，在修改对象时，也必须带上该字段。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/annotations.md	false

v1.LimitRangeSpec

LimitRangeSpec定义了一类资源的限制，包括最小和最大可用范围。

Name	描述	是否必须	类型	默认值
limits	Limits是一组 LimitRangeItem 对象。	true	v1.LimitRangeItem array	

v1.ISCSIVolumeSource

ISCSIVolumeSource 是一种iSCSI磁盘，它可以以read/write模式挂载，但只能挂载一次

Name	描述	是否必须	类型	默认值
targetPortal	iSCSI目标地址，可以是IP地址，如果端口号不是默认端口号（比如tcp port860和3260），也可以是ip_addr:port。	true	string	
iqn	目标iSCSI磁盘的名字	true	string	
lun	目标iSCSI磁盘的逻辑单元号。	true	integer (int32)	
fsType	要挂载的磁盘的文件系统类型。注意：必须保证磁盘的文件系统与操作系统的文件系统相兼容，比如采用"ext4", "xfs", "ntfs"格式。 详情查 看：http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#iscsi	true	string	
readOnly	该选项保证挂载数据卷时将其设置为只读。 默认为false。	false	boolean	false

v1.EmptyDirVolumeSource

EmptyDirVolumeSource是一个临时文件目录，生命周期与pod一致。

Name	描述	是否必须	类型	默认值
medium	<p>Kubernetes中文文档</p> <p>该目录之后运行的存储介质的类型， 默认为""， 即使用节点的存储介质类型。该字段必须为空字符串(默认)或者Memory。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#emptydir</p>	false	string	

v1.NodeList

NodeList只是注册到master上的所有节点的列表。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	一组节点	true	v1.Node array

unversioned.Patch

Patch定义了Kubernetes PATCH请求的具体名字和类型。

v1.PersistentVolumeClaim

PersistentVolumeClaim是用户对持久存储卷的请求和请求声明。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Spec 定义了 pod 创建者请求的数据卷的属性 (characteristics)。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistentvolumeclaims</p>	false	v1.PersistentVolumeClaim
status	<p>Status 代表当前持久存储卷声明的当前状态和信息。该字段是只读的。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistentvolumeclaims</p>	false	v1.PersistentVolumeClaim

v1.NamespaceList

Kubernetes中文文档

NamespaceList是一组命名空间。

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
items	Items是一组命名空间对象。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/namespaces.md	true	v1.Namespace array

v1.ServiceAccount

ServiceAccount将一组字段绑定到一起，这些字段分别是：用户或外围系统可辨识的Name(用于身份识别)、能够被认证或者授权的原则、一组secrets。

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。 client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。 详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象元数据。 详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
secrets	Secrets是一组secret，使用当前ServiceAccount的pods有权使用Secrets。 详情查看： http://releases.k8s.io/HEAD/docs/user-guide/secrets.md	false	v1.ObjectReference
imagePullSecrets	ImagePullSecrets是同一namespace中一组secret的引用，使用当前ServiceAccount的pod有权使用它下载镜像。ImagePullSecrets与Secrets不同，Secrets可以被挂载到pod中，而	false	v1.LocalObjectReference

ImagePullSecrets只能被kubelet使用。详情查看：<http://releases.k8s.io/HEAD/docs/user-guide/secrets.md#manually-specifying-an-imagepullsecret>

v1.NodeAddress

NodeAddress包含了节点地址的信息。

Name	描述	是否必须	类型	默认值
type	节点地址类型，可以是域名、外网ip或者内网ip。	true	string	
address	节点地址	true	string	

v1.Namespace

Namespace是为名字 (Name) 定义了域，可以同时使用多个Namespace。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Spec 定义了 Namespace 的行为。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.NamespaceSpec
status	<p>Status 描述了 Namespace 的当前状态。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.NamespaceStatus

v1.FlockerVolumeSource

FlockerVolumeSource 代表一个 Flock 存储设备，它是由 Flocker agent 挂载的。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档	datasetName 必填字段： 存储卷的名字。依据 Flocker负载 情况，这个 值被存储在 meta中。	true	string	

v1.PersistentVolumeClaimVolumeSource

PersistentVolumeClaimVolumeSource是同一个命名空间下用户持久存储卷声明 (PVC) 的引用。通过这个字段可以找到绑定的持久存储卷 (PV)，并将该PV挂载到指定的pod上。从本质上来说，PersistentVolumeClaimVolumeSource是对其他用户(或系统)拥有的存储卷类型的一个封装。

Name	描述	是否必须	类型	默认值
claimName	ClaimName是Pod使用存储卷时，对应的 PersistentVolumeClaim的名字。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistentvolumeclaims	true	string	
readOnly	在挂载时强制将存储卷设置为只读。默认为 false。	false	boolean	false

unversioned.ListMeta

ListMeta定义了综合资源的元数据，包括列表或各种状态对象。一个资源只能有 ObjectMeta和ListMeta中的一个。

Name Kubernetes中文文档	描述	是否 必须	类型
selfLink	SelfLink是当前对象的URL。该字段由系统生成的，是一个只读字段。	false	string
resourceVersion	<p>用来表示当前对象的内部版本号，client端根据这个字段来判断对象是否已经发生变化。它也能用在optimistic concurrency，变化检测和监控外界对一个或者一组资源做了哪些操作。client端必须显式地对待这些值，并且不做任何修改地传递给server端。这个字段可能只对特定资源有效。该字段由系统生成的，是一个只读字段。</p> <p>详情查 看 : http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#concurrency-control-and-consistency</p>	false	string

v1.ResourceQuotaList

ResourceQuotaList is a list of ResourceQuota items.

Name	描述
Kubernetes中文文档	
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不同的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则忽略该字段。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resource-version
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#kinds
items	Items is a list of ResourceQuota objects. 详情查看： http://releases.k8s.io/HEAD/docs/design/admission_control_resource_quota.md#plugin-resourcequota

v1.PersistentVolumeClaimStatus

PersistentVolumeClaimStatus是一个持久存储卷声明的当前状态。

Name	描述	是否必须	类型
Kubernetes中文文档			
phase	Phase代表一个PersistentVolumeClaim处于哪个阶段。	false	string
accessModes	AccessModes包含了支持当前PVC声明的存储卷的访问模式。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#access-modes-1	false	v1.PersistentVolumeAccessModes array
capacity	底层存储卷的实际资源量。	false	any

v1.EndpointSubset

EndpointSubset是一组带有端口号的地址。扩展endpoint集是由地址和端口号构成的坐标集。举个例子：

```
{
  Addresses: [{"ip": "10.10.1.1"}, {"ip": "10.10.2.2"}],
  Ports: [{"name": "a", "port": 8675}, {"name": "b", "port": 309}]
}
```

上面的endpoint集最终可以写成如下形式：

```
a: [ 10.10.1.1:8675, 10.10.2.2:8675 ],
b: [ 10.10.1.1:309, 10.10.2.2:309 ]
```

Name	描述	是否必须	类型	默认值
addresses	拥有端口号的IP地址，ip和对应的端口号构成可访问的endpoints (系统标记为ready)，这些endpoints可以用来做	false	v1.EndpointAddress array	

	负载均衡，以供client端使用。		
notReadyAddresses	拥有端口号的IP地址，但是IP和对应的端口号构成的endpoints由于某些原因而没有被系统标记为ready。可能的原因有：尚未初始化成功、未能通过最近一次的可用性检查(readiness check)，或者未能通过最近一次的活性检查(liveness check)。	false	v1.EndpointAddress array
ports	端口号，与相关联的IP地址可构成可用的endpoints。	false	v1.EndpointPort array

v1.SecretVolumeSource

SecretVolumeSource会修改Secret使其适用于VolumeSource。详情查看：<http://releases.k8s.io/HEAD/docs/design/secrets.md>

Name	Kubernetes中文文档	描述	是否必须	类型	默认值
secretName		SecretName是在当前pod的命名空间中一个secret的名字。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#secrets	true	string	

v1.EnvVarSource

EnvVarSource表示环境变量(EnvVar)值的来源。

Name	描述	是否必须	类型	默认值
fieldRef	用于加密pod的一个字段，目前只支持加密Name和Namespace。	true	v1.ObjectFieldSelector	

v1.LoadBalancerIngress

LoadBalancerIngress表示负载均衡器的入口状态，对应服务的所有流量都应该被发往该入口。

Name	Kubernetes中文文档	描述	是否必须	类型	默认值
ip		如果负载均衡器入口是基于IP的，比如GCE或者openstack的均衡器，那么会设置IP字段。	false	string	
hostname		如果负载均衡器入口是局域DNS的，比如AWS的均衡器，那么会设置hostname字段。	false	string	

v1.Service

Service是对软件服务(比如mysql)的命名抽象，它是由两部分组成：1. 代理监听的本地端口(比如3306)；2. 用于确定哪些pods应答请求的选择器(selector)。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查</p> <p>看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Spec定义了一个service的行为。http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ServiceSpec
status	<p>最近一次观察到的service的状态。该字段由系统生成的，是一个只读字段。详情查</p> <p>看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ServiceStatus

v1.ServiceAccountList

ServiceAccountList是一组ServiceAccount对象。
Kubernetes中文文档

Name	描述
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds
items	ServiceAccount列表。详情查看： http://releases.k8s.io/HEAD/docs/design/service_accounts.md#service-accounts

v1.LimitRangeList

LimitRangeList一组LimitRange条目。

Name	描述
Kubernetes中文文档	
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs-devel/api-conventions.md#types-kinds
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs-devel/api-conventions.md#resources
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs-devel/api-conventions.md#types-kinds
items	一组LimitRange对象。详情查看： http://releases.k8s.io/HEAD/docs/design/admission_control_limit_range.md

v1.Endpoints

Endpoints用来支撑service的endpoints集合，实际的请求都是要通过service分发到不同的endpoints去处理。例如：

```

Name: "mysvc",
Subsets: [
{
  Addresses: [{"ip": "10.10.1.1"}, {"ip": "10.10.2.2"}],
  Ports: [{"name": "a", "port": 8675}, {"name": "b", "port": 309}]
},
{
  Addresses: [{"ip": "10.10.3.3"}],
  Ports: [{"name": "a", "port": 93}, {"name": "b", "port": 76}]
}
]
```

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
subsets	<p>所有endpoints子集合(subset)的并集。不同的地址(addresses)根据他们共享的IP被放到不同的子集合中。一个具有多个端口的地址(address)，如果一些端口是ready，而另一些由于来自于不同的容器而not ready，可能会在多个子集合中显示，但是会带有不同的端口号。任何一个地址，都不可能在同一个子集合的Addresses和NotReadyAddresses字段中同时存在。地址和端口号的集合组成一个service。</p>	true	v1.EndpointSubset array

v1.DeleteOptions

DeleteOptions可能会在删除一个API对象时提供。
Kubernetes中文文档

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
gracePeriodSeconds	定义了对象将在多少秒后删除，必须是非负值。该字段为0意味着立即删除。如果该字段为空(nil)，将采用默认值。如果没有指定默认值，那么将采用该对象的值。	true	integer (int64)

v1.Volume

Volume表示pod中的一个已命名存储卷，该存储卷可以被pod中的所有容器访问。

Name	描述

Kubernetes中文文档	存储卷名字。必须是一个DNS) LABEL，并且在一个pod中必须是唯一的。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#names
hostPath	HostPath是指存储卷在宿主机上的文件/目录路径，该路径被直接暴露给容器。只有系统级Agent或者享有特权的进程才允许查看宿主机文件系统，因此大多数容器不需要这个字段。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#hostpath
emptyDir	EmptyDir表示一个临时目录，其生命周期与所在的pod一致。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#emptydir
gcePersistentDisk	GCEPersistentDisk表示GCE磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk
awsElasticBlockStore	AWSVolume代表AWS磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#awselasticblockstore
gitRepo	GitRepo表示一个特定版本的git仓库。
secret	Secret是存在于存储卷中的secret。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#secrets
nfs	NFS代表一次宿主机上的nfs磁盘挂载，随pod产生而挂载，随pod删除卸载。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#nfs
iscsi	iSCSI代表iSCSI磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。详情查看： http://releases.k8s.io/HEAD/examples/iscsi/README.md
glusterfs	

Kubernetes中文文档	Glusterfs代表一次宿主机上的Glusterfs磁盘挂载，随pod产生而挂载，随pod删除卸载。详情查 看： http://releases.k8s.io/HEAD/examples/glusterfs/README.md
persistentVolumeClaim	PersistentVolumeClaimVolumeSource表示指向同一个命名空间下一个PersistentVolumeClaim的索引。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#persistentvolumeclaims
rbd	RBD代表一次宿主机上的Rados块设备挂载，随pod产生而挂载，随pod删除卸载。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md
cinder	Cinder表示一个cinder存储卷，它被挂载到运行kubelet的宿主机上。详情查看： http://releases.k8s.io/HEAD/examples/mysql-cinder-pd/README.md
cephfs	CephFS代表一次宿主机上的Ceph文件系统挂载，随pod产生而挂载，随pod删除卸载。
flocker	Flocker表示一个Flocker存储卷，它被挂载到运行着kubelet的宿主机上，依赖于Flocker control service的正常运行。
downwardAPI	DownwardAPI代表使用该存储卷的pod的下行Api。
fc	FC表示一个Fibre Channel资源，它被挂载到运行着kubelet的宿主机上，直接暴露给pod。

integer

v1.Probe

Probe (探针)描述了一个将对容器执行的健康状态检查，以确定该容器在运行中并且可以处理请求。

Name Kubernetes中文文档	描述	是否 必须	类型
exec	Exec定义了要执行的动作，必须且只能指定下面的一个动作(tbd)。	false	v1.ExecAction
httpGet	HTTPGet定义了一个将要发送的http请求。	false	v1.HTTPGetAction
tcpSocket	TCPSocket定义了一个将要发送的tcp请求，目前尚不支持tcp钩子。	false	v1.TCPSocketAction
initialDelaySeconds	容器启动以后，需要等待一段时间才去初始化探针。该字段表示等待的秒数。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#container-probes	false	integer (int64)
timeoutSeconds	探针运行的超时时间，默认为1秒，最小值为1。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#container-probes	false	integer (int64)
periodSeconds	运行探针程序的周期（以秒为单位），默认是10秒，最小值为1。	false	integer (int64)
successThreshold	探针运行失败系统认定容器不可用以后，至少连续运行次成功才判定容器是可用的。默认是1次，最小值为1。	false	integer (int32)
failureThreshold	探针运行成功系统认定容器可用以后，至少连续运行次失败才判定容器是不可用的。默认是3，最小值是1。	false	integer (int32)

v1.ReplicationController

ReplicationController表示一个replication controller的配置。

Name	描述	是否 必须	类型

	Kubernetes中文文档		
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>如果ReplicationController的labels为空，metadata被默认认为与该replication controller管理的pod是一致的。标准对象元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Spec定义了replication controller期望行为的规格。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ReplicationControllerSpec
status	<p>Status是最近一次观察到的replication controller的状态。由于窗口期(window time),改状态可能与实际状态不符。该字段由系统生成的，是一个只读字段。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ReplicationControllerStatus

v1.LimitRange

LimitRange设定了一个Namespace中每种资源的使用率限制。

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST 资源类型。 client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST 资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息， apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
spec	Spec定义了要实施的资源限制。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	v1.LimitRangeSpec

v1.DownwardAPIVolumeFile

DownwardAPIVolumeFile表示包含pod字段的文件的信息。
Kubernetes中文文档

Name	描述	是否必须	类型	默认值
path	必填字段。表示要创建的文件的相对路径，不能包含任何绝对路径或者".."，必须采用utf8编码。该字段的第一部分不能以".."开头。	true	string	
fieldRef	必填字段。可以选择pod的某个字段，目前支持 annotations, labels, name, namespace。	true	v1.ObjectFieldSelector	

v1.PodStatus

PodStatus代表一个pod的状态信息。PodStatus会跟踪系统的实际运行状态。

Name Kubernetes中文文档	描述	是否 必须	类型
phase	pod当前运行在哪个阶段(pending, running, etc.)。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#pod-phase	false	string
conditions	pod当前的服务状态。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#pod-conditions	false	v1.PodCondition array
message	人可读的信息，包含了Pod处于当前状态原因详情。	false	string
reason	以驼峰式显示的简单信息，包含了Pod处于当前状态的原因，比如"OutOfDisk"	false	string
hostIP	pod所在的宿主机的ip地址。如果pod还没有被scheduled，那么该字段为空。	false	string
podIP	Pod的IP地址，在集群内可访问。如果还没有分配，则该字段为空。	false	string
startTime	遵循RFC3339格式的时间，表示kubelet认为该对象的时间，该时间一般在kubelet为Pod拉取镜像之前。	false	string
containerStatuses	在Manifest中，每个容器都有一个条目。该字段是一个条目。这些条目和 docker inspect 的输出一致。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#container-statuses	false	v1.ContainerStatus array

v1.PodSpec

PodSpec是pod的描述。

--	--

Name Kubernetes中文文档	描述
volumes	一组可被pod内容器挂载的存储卷。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md
containers	一组隶属于当前pod的容器。目前尚不支持pod内容器的增加和删除。每个Pod内必须至少有一个容器。该字段不能被更改。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/containers.md
restartPolicy	pod内所有容器的重启策略，目前有三种：Always (总是), OnFailure (失败时重启), Never(不重启)。默认值为Always。详情看： http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#restartpolicy
terminationGracePeriodSeconds	需要优雅地终止pod时，可选的时间间隔(单位：秒)。在client端发送删除请求时，该字段可以被缩减，但必须是非负值。如果值为0，则意味着立即删除。如果该字段为空(nil)，将使用默认值。Grace Period(缓冲期)是指pod中运行的进程从接收到一个termination信号到接收到kill信号被强制杀死之间的一段时间，单是秒。考虑到进程进行自我清理的时间，terminationGracePeriodSeconds要设置得稍微长一些。默认值为30s。
activeDeadlineSeconds	(tbd) Optional duration in seconds the pod may be active on the node relative to StartTime before the system will actively try to mark it failed and kill associated containers. Value must be a positive integer.
dnsPolicy	pod内部容器的DNS策略，可以是： <i>ClusterFirst</i> 或 <i>Default</i> 用"ClusterFirst"。
nodeSelector	NodeSelector是一个选择器，pod通过它选择分配到哪些节点上。只有当pod的该字段与节点的labels匹配时，才会被分配。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/node-selection/README.md
serviceAccountName	ServiceAccountName是用来运行当前pod的ServiceAccount。详查看： http://releases.k8s.io/HEAD/docs/design/service_accounts.md

Kubernetes中文文档 serviceAccount	DeprecatedServiceAccount是ServiceAccountName的别名，目前已作废，请使用ServiceAccountName。
nodeName	.nodeName请求将当前pod分配到特定节点。如果该字段非空，调度器会假设该节点满足pod的资源需求，而简单地把当前pod分到指定的节点。
hostNetwork	为当前pod请求到的宿主机网络，使用宿主机网络的命名空间。如果该选项设置为true，则必须指定pod要使用的端口号。默认为false。
hostPID	使用宿主机的pid命名空间。该字段是可选字段，默认值为false。
hostIPC	使用宿主机的ipc命名空间。该字段是可选字段，默认值为false。
securityContext	SecurityContext保存了pod级别的安全属性和常用的容器设置。该字段是可选字段，默认为空。每个字段的默认值会有所不同，具体查阅字段对应的type description。
imagePullSecrets	ImagePullSecrets是一组可选的引用，它们只想同意命名空间下的secret，PodSpec使用这些secret拉取镜像。如果指定了该值，这些secrets将会被发送到每一个独立的镜像拉取对象。例如，对于docker而言，只有DockerConfig类型的secret才有效。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/images.md#specifying-imagepullsecrets-on-a-pod

v1.ContainerPort

ContainerPort代表了单个容器的网络端口。

Name	描述	是否必须	类型	默认值
name	如果指定改名字，那么必须是一个IANA_SVC_NAME，并且在pod内是唯一的。pod内的每个已命名port对象都必须有一个唯一的name。services可以通过name来定位该port对象。	false	string	
hostPort	映射到主机上的端口号。如果指定它的值，则必须是一个有效的端口号(0~65535)。如果指定了HostNetwork字段，那么它必须与ContainerPort字段匹配。大多数容器不需要使用该字段。	false	integer (int32)	
containerPort	映射到pod IP地址上的端口号，0~65535。	true	integer (int32)	
protocol	该端口对象使用的协议，必须是UDP或TCP。默认值是"TCP"。	false	string	
hostIP	将external port绑定到的IP。	false	string	

v1.ResourceQuota

ResourceQuota设置了每个命名空间的总计资源限制。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查 看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Spec 定义了期望的配额。http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ResourceQuotaSpec
status	<p>Status 表示实际的资源配额和当前的使用率。http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.ResourceQuotaStatus

v1.EventList

EventList是一组事件(event)。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
items	一组事件对象列表。	true	v1.Event array

v1.Lifecycle

Lifecycle描述了管理系统应对容器生命周期事件所采取的一系列动作。对于PostStart和PreStop动作会阻塞容器操作直到动作完成。有一个情况例外：容器运行失败时，处理函数被丢弃。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档				
postStart	<p>PostStart动作会在容器创建以后立即被执行。如果动作执行失败，容器会被终止，并根据重启策略重启。在当前动作结束之前，对于该容器的任何其他操作都会被阻塞。详情查 看：http://releases.k8s.io/HEAD/docs/user-guide/container-environment.md#hook-details</p>	false	v1.Handler	
preStop	<p>PreStop动作会在容器终止以后立即被执行，该动作执行完成以后容器即终止。不管该动作执行结果如何，容器最终都会被终止。在当前动作结束之前，对于该容器的任何其他操作都会被阻塞。详情查 看：http://releases.k8s.io/HEAD/docs/user-guide/container-environment.md#hook-details</p>	false	v1.Handler	

v1.ReplicationControllerSpec

ReplicationControllerSpec描述了replication controller的规格。

Name	描述	是否必须	类型
Kubernetes中文文档			
replicas	<p>Replicas指定了期望的副本(replica)数量。这是一个指针类型，可以区分0和nil。默认值为1。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/replication-controller.md#what-is-a-replication-controller</p>	false	integer (int32)
selector	<p>Selector是一个针对pod的label查询规格，查询结果应该与replicas字段的值一致。如果该字段为空，则默认采用pod模板中的labels。Label的键和值都必须与该Selector匹配，replication controller才能操作关联的pods。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/labels.md#label-selectors</p>	false	any
template	<p>Template用来检测replica数量，如果不满足replicas字段定义的值，那么会创建新的pod。如果TemplateRef字段也被设置，那么优先使用本字段。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/replication-controller.md#pod-template</p>	false	v1.PodTemplateSpec

v1.NodeStatus

NodeStatus指节点的当前状态信息。

Name Kubernetes中文文档	描述	是否必须
capacity	Capacity表示节点的可用资源。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#capacity for more details.	false
phase	NodePhase表示节点处于生命周期中的那个阶段。详情查看： http://releases.k8s.io/HEAD/docs/admin/node.md#node-phase	false
conditions	Conditions指一组最近探测到的节点状态信息。详情查看： http://releases.k8s.io/HEAD/docs/admin/node.md#node-condition	false
addresses	一组可访问的地址，如果有的话，可以从云服务提供商获取到。详情查看： http://releases.k8s.io/HEAD/docs/admin/node.md#node-addresses	false
daemonEndpoints	运行在该节点上的后台程序暴露出来的Endpoints。	false
nodeInfo	用来唯一识别一个节点的ids/uuids集合。详情查看： http://releases.k8s.io/HEAD/docs/admin/node.md#node-info	false

v1.GlusterfsVolumeSource

GlusterfsVolumeSource表示一次Glusterfs磁盘挂载，随pod创建挂载，随pod删除卸载。

Name	描述
Kubernetes中文文档	
endpoints	EndpointsName即endpoint名字， 通过这个endpoint可以读取Glusterfs的拓扑结构。详情查 看： http://releases.k8s.io/HEAD/examples/glusterfs/README.md#create-a-pod
path	Path是当前Glusterfs存储卷的路径。详情查 看： http://releases.k8s.io/HEAD/examples/glusterfs/README.md#create-a-pod
readOnly	ReadOnly字段用来保证挂载Glusterfs存储卷时采用只读模式。默认值为false。详情查 看： http://releases.k8s.io/HEAD/examples/glusterfs/README.md#create-a-pod

v1.Handler

Handler定义了一个动作(action)。

Name	Kubernetes中文文档	描述	是否必须	类型	默认值
exec		Exec定义了要执行的动作，必须且只能指定下面的一个动作(tbd)。	false	v1.ExecAction	
httpGet		HTTPGet定义了一个将要发送的http请求。	false	v1.HTTPGetAction	
tcpSocket		TCPSocket定义了一个将要发送的tcp请求，目前尚不支持tcp钩子。	false	v1.TCPSocketAction	

v1.EventSource

EventSource包含了一个事件的信息。

Name	描述	是否必须	类型	默认值
component	Component是产生event的源。	false	string	
host	Host表示event是在哪台机器上产生的。	false	string	

v1.PodCondition

PodCondition包含了pod的当前状态详情。

Name Kubernetes中文文档	描述	是否 必须	类型	
type	Type是conditon的类型， 目前只有 Ready。 详情查 看 : http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#pod-conditions	true	string	
status	Status是condition的状态， 可以为True、False或Unknown。 详情查 看 : http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#pod-conditions	true	string	
lastProbeTime	最近一次获取condition信息的时间。	false	string	
lastTransitionTime	最后一次condition状态发生变化的时间。	false	string	
reason	最后一次condition状态发生变化的时间。 特点：唯一性、单个词语描述、驼峰式显示。	false	string	
message	最后一次condition状态变化的详情，以人可读的方式显示。	false	string	

v1.RBDVolumeSource

RBDVolumeSource表示一次Rados块设备挂载，随pod创建挂载，随pod删除卸载。

Name	描述	是否必须
Kubernetes中文文档		
monitors	一组Ceph monitor。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	true
image	rados镜像名。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	true
fsType	要挂载的存储卷的文件类型。提示：必须保证宿主操作系统支持存储卷的文件类型，比如ext4, xfs, ntfs。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#rbd	false
pool	rados pool名， 默认为rbd。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it.	true
user	rados user名， 默认为admin。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	true
keyring	Keyring是RBDUser key ring (tbd)文件的路径， 默认为/etc/ceph/keyring。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	true
secretRef	SecretRef是RBDUser用来认证的secret的名字， 它会覆盖keyring字段的设置。默认为空。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	true
readOnly	该选项保证挂载数据卷时将其设置为只读。 Defaults to false. 详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md#how-to-use-it	false

v1.PodTemplate

Kubernetes中文文档

PodTemplate定义了一个用于创建pod副本的模板。

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
template	Template表明pod将会基于这个模板创建。 http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	v1.PodTemplateSpec

v1.ServiceStatus

ServiceStatus表示service的当前状态。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
loadBalancer	LoadBalancer是指负载均衡器的当前状态。前提是设置了负载均衡器。	false	v1.LoadBalancerStatus	

v1.NFSVolumeSource

NFSVolumeSource表示一次NFS磁盘挂载。随pod创建挂载，随pod删除卸载。

Name	描述	是否必须	类型	默认值
server	Server表示NFS服务器的域名或者ip地址。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#nfs	true	string	
path	Path是指NFS设置的exports路径。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#nfs	true	string	
readOnly	ReadOnly设置是否以只读方式挂载NFS export, 默认值是false。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#nfs	false	boolean	false

v1.FCVolumeSource

只能以read/write模式挂载一次的光纤磁盘。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
targetWWNs	必填字段：FC目标磁盘的全局名称(WWN)。	true	string array	
lun	必填字段：FC目标磁盘的逻辑单元号。(tbd)	true	integer (int32)	
fsType	必填字段：磁盘的文件系统类型，必须是宿主机操作系统支持的类型，比如"ext4", "xfs", "ntfs"。	true	string	
readOnly	可选字段， 默认为false(read/write)。该选项保证挂载数据卷时将其设置为只读。	false	boolean	false

v1.EndpointPort

EndpointPort是描述单个port对象的元组。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档				
name	port对象的名字，与ServicePort.Name对应。必须是DNS_LABEL。如果已经定义了port字段的值，该字段是可选的。	false	string	
port	endpoint的端口号	true	integer (int32)	
protocol	该port对象使用的协议，必须是UDP或者TCP。默认值为TCP。	false	string	

v1.TCPSocketAction

TCPSocketAction定义了对socket的动作。

Name	描述	是否必须	类型	默认值
port	访问容器需要的port对象的端口号或名字。端口号必须是1~65535，名字必须是IANA_SVC_NAME。	true	string	

unversioned.StatusDetails

StatusDetails是一组附加属性，server端返回响应时，可能会添加。Reason字段会说明包含了哪些附加字段。client端可以忽略不匹配的属性，并且默认所有附加属性均为empty、invalid或者undefined。

Name Kubernetes中文文档	描述	是否必须	类型
name	StatusReason相关的资源的name属性。前提是该资源只有一个名字。	false	string
kind	StatusReason相关的资源的kind属性。对于某些操作，这个字段的值可能会与请求到的资源kind不同。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
causes	status为failure时，StatusReason的详细信息。并不是所有的StatusReasons都会提供详细的原因为。	false	unversioned.StatusReason array
retryAfterSeconds	如果设置了该字段，(tbd) If specified, the time in seconds before the operation should be retried.	false	integer (int32)

v1.HTTPGetAction

HTTPGetAction定义了对HTTP Get请求的动作。

Name	描述	是否必须	类型	默认值
path	访问HTTP server的路径。	false	string	
port	port对象的名字或者端口号，可以用来访问容器。端口号必须是1~65535，名字必须是IANA_SVC_NAME。	true	string	
host	要连接的域名，默认是pod IP。	false	string	
scheme	连接host使用的协议(Scheme)，默认是HTTP。	false	string	

v1.LoadBalancerStatus

LoadBalancerStatus表示一个负载均衡器的状态。

Name	描述	是否必须	类型	默认值
ingress	Ingress是负载均衡器的一组入口。发往service的流量应该被发送到这些入口。	false	v1.LoadBalancerIngress array	

v1.SecretList

SecretList是一组secret。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	<p>Items是一组secret对象。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/secrets.md</p>	true	v1.Secret array

v1.Container

要运行在pod中的单应用容器。

Name	描述

	Kubernetes中文文档
name	容器名字，会被用作DNS_LABEL。pod中的每个容器都必须有一个唯一的名字(DNS_LABEL)。名字一旦确定就不能更改。
image	Docker镜像名。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/images.md
command	指令入口数组。注意：这些指令不在shell中执行。如果未设置该字段，那么使用docker镜像自带的指令入口(CMD)。使用环境变量时，变量名\$(VAR_NAME)会被展开。如果使用了不存在的环境变量，那么输入字符串不会改变(即不展开环境变量名)。T可以使用\$\$对环境变量中的\$进行转义。不管环境变量是否存在，转义的环境变量名都不会被展开。环境变量一旦设置，就无法更改。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/containers.md#containers-and-commands
args	指令入口参数列表。如果未设置该字段，那么使用docker镜像自带的指令入口(CMD)。使用环境变量时，变量名\$(VAR_NAME)会被展开。如果使用了不存在的环境变量，那么输入字符串不会改变(即不展开环境变量名)。可以使用\$\$对环境变量中的\$进行转义。不管环境变量是否存在，转义的环境变量名都不会被展开。环境变量一旦设置，就无法更改。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/containers.md#containers-and-commands
workingDir	容器运行的当前目录，与docker默认目录一致，即docker镜像中设置的目录。一旦设置，就无法更改。
ports	从容器暴露出来的一组port对象。该字段不能更改。
env	容器中设置的一组环境变量。该字段不能更改。
resources	计算出该容器需要的资源。该字段不能更改。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#resources
volumeMounts	挂载到容器文件系统的Pod存储卷。该字段不能更改。

livenessProbe Kubernetes中文文档	<p>周期性探测容器活性的探针。如果探测失败，系统将会重启容器。该字段不能更改。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#container-probes</p>
readinessProbe	<p>Periodic probe of container service readiness. Container will be removed from service endpoints if the probe fails. Cannot be updated. 详情查看：http://releases.k8s.io/HEAD/docs/user-guide/pod-states.md#container-probes</p>
lifecycle	<p>系统对容器事件采取的动作。该字段不能更改。</p>
terminationMessagePath	<p>可选字段。容器终止时，要把终止信息写入到容器文件系统的某个路径下。写入的信息一般是一个简短的最终状态表述，比如 assertion failure。默认路径是/dev/termination-log。该字段不能更改。</p>
imagePullPolicy	<p>Image拉取策略，有以下几种：Always, Never, IfNotPresent。如果指定镜像tag为:latest，那么默认是Always；否则，默认是IfNotPresent。该字段不能更改。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/images.md#updating-images</p>
securityContext	<p>Pod运行时携带的安全配置项。详情查看：http://releases.k8s.io/HEAD/docs/design/security_context.md</p>
stdin	<p>容器是否在运行时给stdin分配了buffer。如果该字段没有设置，从容器的stdin读取数据会返回EOF。该字段默认值为false。</p>
stdinOnce	<p>容器运行时是否在一个client读取stdin以后，就关闭stdin防止其它的client读取。如果stdin字段被设置为true，那么stdin会允许多个client去读取。如果stdinOnce被设置为true，那么stdin会在容器启动时开放，第一个连接它的client会写入数据，断开以后，stdin也会关闭。除非容器重启，stdin才会打开。如果设置成false，容器中读取stdin的进程永远都不会接收到EOF。该字段默认为false。</p>
tty	<p>容器是否给自己分配一个tty。如果分配的话，需要设置stdin true。该字段默认值是false。</p>

v1.PersistentVolumeSpec

Kubernetes中文文档

PersistentVolumeSpec表示持久存储卷的规格。

Name	描述
capacity	持久存储卷资源和容量。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#capacity
gcePersistentDisk	GCEPersistentDisk表示GCE磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。该磁盘由管理员配置。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#gcepersistentdisk
awsElasticBlockStore	AWSElasticBlockStore表示AWS磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#awselasticblockstore
hostPath	HostPath表示宿主机上的一个目录，由开发者或测试人员配置。 这只是用作单节点的开发和测试，不能用作生产环境。该模式不支持磁盘挂载，也不能再多节点集群中使用。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#hostpath
glusterfs	Glusterfs表示Glusterfs磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。该磁盘由管理员设置。详情查 看： http://releases.k8s.io/HEAD/examples/glusterfs/README.md
nfs	NFS表示挂载到宿主机上的nfs磁盘，由管理员设置。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#nfs
rbd	RBD 表示Rados块设备，它被挂载到运行kubelet的宿主机上，直接暴露给pod。详情查 看： http://releases.k8s.io/HEAD/examples/rbd/README.md

iscsi Kubernetes中文文档	ISCSI表示ISCSI磁盘资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。该磁盘由管理员设置。
cinder	Cinder表示Cinder存储卷，它被挂载到运行kubelet的宿主机上，直接暴露给pod。该磁盘由管理员设置。详情查看： http://releases.k8s.io/HEAD/examples/mysql-cinder-pd/README.md
cephfs	CephFS表示Ceph文件系统存储卷，它被挂载到运行kubelet的宿主机上，直接暴露给pod。
fc	FC表示Fibre Channel资源，它被挂载到运行kubelet的宿主机上，直接暴露给pod。
flocker	Flocker表示一个Flocker存储卷，它被挂载到运行着kubelet的宿主机上，依赖于Flocker control service的正常运行。
accessModes	AccessModes包含了所有数据卷挂载方式。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#access-modes
claimRef	ClaimRef是PersistentVolume和PersistentVolumeClaim双向绑定的一部分。绑定时，该值non-nil. claim.VolumeName是比较权威(tbd)的PV和PVC绑定。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#binding
persistentVolumeReclaimPolicy	定义了一个持久存储卷被对应的PVC释放时要做的操作。可选项有两个：Retain(default)和Recycle。Recycle(回收)操作必须通过持久存储卷底层插件来提供支持。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md#recycling-policy

v1.PodSecurityContext

PodSecurityContext包含了pod级别的安全属性和常见的容器配置。
container.securityContext的优先级要高于PodSecurityContext。

Name	描述	是否	类型	默认

Kubernetes中文文档		必须		值
seLinuxOptions	会应用到该宿主机上所有容器的 SELinux配置。如果不指定该字段，容器运行时会随机为容器分配一个 SELinux配置。也可以在 SecurityContext中设置该选项。如果在SecurityContext 和 PodSecurityContext 中均设置了该选项，优先采用 SecurityContext中的配置。	false	v1.SELinuxOptions	
runAsUser	运行容器进程CMD 的用户id(UID)。如果不指定该字段，则默认采用镜像 metadata中的UID 置。也可以在 SecurityContext中设置该选项。如果在SecurityContext 和 PodSecurityContext 中均设置了该选项，优先采用 SecurityContext中的配置。	false	integer (int64)	
runAsNonRoot	容器是否必须在非root用户下运行。如果该字段为true, kubelet将会在运行	false	boolean	false

	<p>时验证镜像，以保证容器不是作为root(UID=0)用户启动的，如果镜像验证不通过，那么将无法启动容器。如果不设置该字段，或者设置为false，那么kubelet将不执行类似的镜像检查。也可以在SecurityContext中设置该选项。如果在SecurityContext和PodSecurityContext中均设置了该选项，优先采用SecurityContext中的配置。</p>			
supplementalGroups	<p>与容器的Group类似，但除容器的GID之外，应用到容器第一个进程的组。如果不指定该字段，没有任何组会被添加到容器中。</p>	false	[integer] array	
fsGroup	<p>一个应用到pod中所有容器的特殊的组。一些存储卷类型允许Kubelet改变pod拥有的对应存储卷：</p> <ol style="list-style-type: none"> 1. 所有者的GID是FSGroup 2. 将设置setgid位(该存储卷中新创建的文件也 	false	integer (int64)	

归FSGroup所有) 3.
permission位为rw-
rw

v1.ReplicationControllerStatus

ReplicationControllerStatus represents the current status of a replication controller.

Name	描述	是否必须	类型
replicas	Replicas是最近探测到的pod副本的数量。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/replication-controller.md#what-is-a-replication-controller	true	integer (int32)
observedGeneration	ObservedGeneration反应了最近观察到的replication controller的generation(tbd)。	false	integer (int64)

v1.FinalizerName

v1.ServicePort

ServicePort包含了service的端口信息。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档				
name	当前端口对象的名字，必须是DNS_LABEL。一个ServiceSpec对象中的所有port对象都必须有唯一的名字。该字段会映射到EndpointPort对象的Name字段。如果在一个Service中有且只有一个ServicePort已经定义，那么该字段是可选的。	false	string	
protocol	本port对象使用的IP协议，支持"TCP"和"UDP"，默认是TCP。	false	string	
port	当前service对外暴露的端口号	true	integer (int32)	
targetPort	当前service用于访问pods的端口号或者端口名字。如果是端口号，则必须是1~65535.如果是名字，则必须是IANA_SVC_NAME。如果该字段是一个字符串，系统将会在目标pod中的容器中的所有port对象中查找该端口。如果不指定该字段，系统将使用Port对象的值(一组端口映射)。该字段的默认是为service的端口号。 详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#defineing-a-service	false	string	
nodePort	设置type=NodePort或者LoadBalancer时，service暴露到宿主机上的端口。通常是由系统分配的。如果指定了该端口，如果该端口号没有被占用，将会分配给该服务；如果宿主机上该端口被占用，那么将分配失败。默认情况下，如果ServiceType需要的话，系统会自动分配一个。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#type—nodeport	false	integer (int32)	

v1.ComponentCondition

Kubernetes中文文档

组件的状态信息。

Name	描述	是否必须	类型	默认值
type	组件的 condition 类型，目前有效值："Healthy"	true	string	
status	组件的 condition 状态。目前有效值为："Healthy": "True", "False", or "Unknown"。	true	string	
message	组件的 condition 信息，比如说：一次健康状态检查返回的信息。	false	string	
error	组件的 Condition 错误码，比如一次健康状态检查返回的错误码。	false	string	

v1.ComponentStatusList

组件的所有condition的状态，类型是ComponentStatus列表。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	一组ComponentStatus对象。	true	v1.ComponentStatus array

v1.HostPathVolumeSource

HostPathVolumeSource表示宿主机的目录。

Name	描述	是否必须	类型	默认值
path	宿主机上目录的路径。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#hostpath	true	string	

json.WatchEvent

Name	描述	是否必须	类型	默认值
type	要监控的事件的类型，可以是：ADDED, MODIFIED, DELETED, 或 ERROR。	false	string	
object	被监控的对象，它的类型与资源 endpoint 的类型相匹配。如果类型为 ERROR，该对象是一个 Status 对象。	false	string	

v1.Binding

Binding 将两个对象绑定在一起。比如，pod 与 node 通过 scheduler(调度器) 绑定在一起。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准对象元数据。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
target	要绑定到当前对象的目标对象。	true	v1.ObjectReference

v1.ContainerStateTerminated

ContainerStateTerminated表示容器的终止状态。

Name	Kubernetes中文文档	描述	是否必须	类型	默认值
exitCode		容器终止时的退出状态(返回值)。	true	integer (int32)	
signal		容器终止时接收到的信号。	false	integer (int32)	
reason		容器终止的原因。	false	string	
message		容器的终止信息。 =	false	string	
startedAt		最近一次容器启动的时间。	false	string	
finishedAt		最近一次容器终止的时间。	false	string	
containerID		容器的ID, 以 <i>docker://<container_id></i> 的格式显示。	false	string	

v1.CinderVolumeSource

CinderVolumeSource表示Openstack中的一个cinder存储卷。 Cinder存储卷必须在挂载到容器之前就存在， 必须和kubelet所在的地域相同。

Name	描述	是否必须	类型	
volumeID	唯一表示一个cinder存储卷的ID。详情查 看： http://releases.k8s.io/HEAD/examples/mysql-cinder-pd/README.md	true	string	
fsType	必填字段：磁盘的文件系统类型，必须是宿主机操作系统支持的类型，目前只支持"ext3"和"ext4"。 详情查 看： http://releases.k8s.io/HEAD/examples/mysql-cinder-pd/README.md	false	string	
readOnly	可选字段，默認為false(read/write)。该选项保证挂载数据卷时将其设置为只读。详情查 看： http://releases.k8s.io/HEAD/examples/mysql-cinder-pd/README.md	false	boolean	

v1.SecurityContext

SecurityContext是容器的安全配置。一些配置选项在SecurityContext和PodSecurityContext都会出现，如果两个地方均有设置，那么优先采用SecurityContext中的配置。

Name	描述	是否必须	类型	默认值
capabilities	容器运行时是否可以add/drop安全配置。默认采用容器运行环境的配置。	false	v1.Capabilities	
privileged	是否在特权模式下运行容器。特权模式下运行的容器，其中的进程权限与宿主机上root用户的权限一样大。该字段默认值为false。	false	boolean	false

seLinuxOptions Kubernetes中文文档	容器的SELinux选项。如果不指定该选项，容器运行外环境会随机指定一个SELinux选项。该选项也可以在PodSecurityContext中设置，如果在SecurityContext和PodSecurityContext均有设置，优先采用SecurityContext中的设置。	false	v1.SELinuxOptions	
runAsUser	运行容器进程CMD的用户ID(UID)。如果不指定，则默认采用容器镜像中的设置。该选项也可以在PodSecurityContext设置，如果在SecurityContext和PodSecurityContext均有设置，优先采用SecurityContext中的设置。	false	integer (int64)	
runAsNonRoot	容器是否必须非root用户下运行。如果该字段为true，Kubelet会校验镜像以确保容器不在root用户(UID 0)下运行，一旦检测到，容器就会启动失败。如果不设置该字段或者将其设置为false，那么就不会执行校验操作。	false	boolean	false

该选项也可以在 PodSecurityContext 设置，如果在 SecurityContext 和 PodSecurityContext 均有设置，优先采用 SecurityContext 中的设置。

v1.ContainerState

ContainerState表示容器可能所处的状态。只能指定下面字段中的一个，如果不指定，则默认是ContainerStateWaiting。

Name	描述	是否必须	类型	默认值
waiting	容器的等待状态信息	false	v1.ContainerStateWaiting	
running	容器运行状态信息	false	v1.ContainerStateRunning	
terminated	容器终止状态信息	false	v1.ContainerStateTerminated	

v1.AWSVolumeSource

表示AWS上的持久存储卷。

Amazon Elastic Block Store (EBS)在被kubelet挂载之前，必须提前被创建、格式化，并且和kubelet在同一个可用区中。注意：AWS EBS存储卷一次只能被挂载到一个实例上。

Name	描述	是否必须	类型	默认值
Kubernetes中文文档				
volumeID	AWS EBS存储卷的唯一标识ID。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#awselasticblockstore	true	string	
fsType	要挂载的存储卷的文件类型。提示：必须保证宿主操作系统支持存储卷的文件类型，比如ext4, xfs, ntfs。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#awselasticblockstore	true	string	
partition	指定你要挂载的是存储卷的那个分区。如果不设置，默认通过存储卷名字挂载。例如，对于存储卷/dev/dsa1，你指定了分区号为"1"，类似地，存储卷分区/dev/sda的分区号为"0"(也可以置空)。	false	integer (int32)	
readOnly	设置为true时，会将VolumeMounts下的ReadOnly属性置为true。如果不设置，默认为false。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/volumes.md#awselasticblockstore	false	boolean	false

v1.ContainerStatus

ContainerStatus包含容器当前状态的详情。

Name	Kubernetes中文文档 描述	是否 必须	类型
name	必须是DNS_LABEL，同一个pod中的容器名字必须是唯一的，一旦创建无法更改。	true	string
state	容器当前状态详情。	false	v1.ContainerState
lastState	容器的终止状态详情。	false	v1.ContainerState
ready	容器是否已经传入了活性检测探针。	true	boolean
restartCount	容器被重启的次数，计数是基于已经死掉且没有被删除的容器个数。注意：由于垃圾回收机制，该值不会大于5。	true	integer (int32)
image	容器使用的镜像。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/images.md	true	string
imageID	容器使用的镜像的ID。	true	string
containerID	容器的ID，以 <code>docker://<container_id></code> 的格式显示。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/container-environment.md#container-information	false	string

v1.ReplicationControllerList

ReplicationControllerList是replication controller的集合。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
items	一组replication controllers对象。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/replication-controller.md	true	v1.ReplicationController array

v1.NodeDaemonEndpoints

NodeDaemonEndpoints是运行在节点上的后台程序监听的所有端口。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
kubeletEndpoint	Kubelet监听的Endpoint	false	v1.DaemonEndpoint	

v1.Secret

Secret保存了特定类型的secret数据。该字段的长度必须必MaxSecretSize小(单位：字节)。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查 看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
data	<p>Data包含了secret数据。数据里的key必须是一个有效的DNS_SUBDOMAIN或者 ("."+DNS_SUBDOMAIN)。secret数据以base64方式序列化，因此可以显示任意格式的数据而不限于string类型。详情查 看：https://tools.ietf.org/html/rfc4648#section-4</p>	false	any
type	用来方便对secret数据进行程序化处理。	false	string

v1.Event

Event用于报告集群中发生的事件。
Kubernetes中文文档

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准对象元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	true	v1.ObjectMeta
involvedObject	与事件相关的对象。	true	v1.ObjectReference
reason	对象状态转变为当前状态的原因，以简短的、机器可读的字符串表示。	false	string
message	当前状态的描述，以人可读的方式显示。	false	string
source	报告该事件的组件，以简短的、机器可读的字符串表示。	false	v1.EventSource

firstTimestamp Kubernetes中文文档	该事件第一次被记录的时间，server接收到到事件的时间保存在TypeMeta中。	false	string
lastTimestamp	该事件最近一次被记录的时间。	false	string
count	该事件出现的次数。	false	integer (int32)

v1.EnvVar

EnvVar表示容器中的环境变量。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
name	环境变量的名字，必须是C_IDENTIFIER。	true	string	
value	在容器或者service中预定义了环境变量\$(VAR_NAME)时，变量名\$(VAR_NAME)会被展开。如果使用了不存在的环境变量，那么输入字符串不会改变(即不展开环境变量名)。可以使用\$\$对环境变量中的\$进行转义。不管环境变量是否存在，转义的环境变量名都不会被展开。环境变量一旦设置，就无法更改。该字段默认值为空字符串""。	false	string	
valueFrom	环境变量值的来源。如果该字段不为空，则不可用Source for the environment variable's value. Cannot be used if value is not empty. (tbd)	false	v1.EnvVarSource	

v1.ResourceRequirements

Kubernetes中文文档

ResourceRequirements定义了计算资源需求。

Name	描述	是否必须
limits	允许使用的计算资源的上限。详情查 看： http://releases.k8s.io/HEAD/docs/design/resources.md#resource-specifications	false
requests	需要的计算资源的下限。如果不设置容器的requests，默認為显式定义的limits。如果没有显式定义limits，则requests默認為实际值(implementation-defined value)。详情查 看： http://releases.k8s.io/HEAD/docs/design/resources.md#resource-specifications	false

v1.PersistentVolumeAccessMode

v1.ComponentStatus

ComponentStatus(也叫ComponentStatusList)保存了集群的校验信息。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准对象元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
conditions	一组组件状态对象。	false	v1.ComponentCondition array

v1.LimitRangeItem

LimitRangeItem定义了每种类型资源的使用量的上下限。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
type	limit应用到的资源的类型。	false	string	
max	对于这类资源的最大使用限制。	false	any	
min	对于这类资源的最小使用限制。	false	any	
default	如果limit没有设置，则limit采用当前字段定义的值。	false	any	
defaultRequest	如果request没有设置，则request采用当前字段定义的值。	false	any	
maxLimitRequestRatio	如果设置了MaxLimitRequestRatio，则对应的资源必须设置非0值的request和limit，并且limit/request<=enumerated value="">(tbd)。当前字段表示对应资源的最大使用率。	false	any	

v1.PodTemplateSpec

PodTemplateSpec表示根据模板创建pod时携带的数据。

Name	描述	是否必须	类型
Kubernetes中文文档			
metadata	标准对象元数据。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata	false	v1.ObjectMeta
spec	pod期望行为的规格。详情查 看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	v1.PodSpec

v1.PodList

PodList is a list of Pods.

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	<p>一组pod对象。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/pods.md</p>	true	v1.Pod array

v1.ServiceList

ServiceList表示一组services。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
items	一组service对象。	true	v1.Service array

v1.PersistentVolumeList

PersistentVolumeList是一组持久存储卷(PersistentVolume)。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准列表的元数据。详情查看：</p> <p>http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	unversioned.ListMeta
items	<p>一组持久存储卷对象。详情查看：http://releases.k8s.io/HEAD/docs/user-guide/persistent-volumes.md</p>	true	v1.PersistentVolume array

v1.ObjectReference

ObjectReference包含了被引用对象的信息，通过这些信息您可以探测或者修改被引用对象。

Name Kubernetes中文文档	描述	是否 必须	类型
kind	被引用对象的类型(kind)。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
namespace	被引用对象的命名空间。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/namespaces.md	false	string
name	被引用对象的名字。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#names	false	string
uid	被引用对象的UID。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/identifiers.md#uids	false	string
apiVersion	被引用对象的API version。	false	string
resourceVersion	创建当前引用时，生成的resourceVersion。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#concurrency-control-and-consistency	false	string
fieldPath	如果当前引用指向的是某个对象的一部分，而不是整个对象，fieldPath是一个有效的JSON或Go语言对象，比如desiredState.manifest.containers[2]。举个例子，如果被引用对象是pod中的一个容器，当前字段的值就像这样："spec.containers{name}"，这里"name"是指触发事件的容器的名字；如果没有容器名字，是这样："spec.containers[2]"(这里是指pod中index为2的容器)。这种语法只能用于引用定义规范的对象。	false	string

unversioned.Status

Kubernetes中文文档

Status是API调用的返回值。如果API调用不返回其它对象，则会返回该值。

Name	描述	是否必须	类型
kind	Kind 是指当前对象代表的REST资源类型。 client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。 详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	string
apiVersion	APIVersion定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources	false	string
metadata	标准列表的元数据。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds	false	unversioned.ListMeta
status	操作的状态，是"Success"或"Failure"。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status	false	string
message	当前操作的状态，以人可读的方式表示。	false	string
reason		false	string

	<p>Kubernetes中文文档</p> <p>操作状态处于"Failure"时，以机器可读的方式表示的原因信息。该字段为空意味着没有可用信息。Reason比HTTP状态码更清晰，但并不会覆盖HTTP状态码。</p>	
details	基于reason字段的扩展信息。每个reason字段都有对相应的扩展信息。该字段是可选的，返回的数据符合reason类型定义，但不保证符合其他规格。	false
code	当前状态对应的的HTTP返回值。0意味着没有设置。	false

v1.ContainerStateWaiting

ContainerStateWaiting容器的等待状态。

Name	描述	是否必须	类型	默认值
reason	简短的原因，描述为什么容器不是running状态。	false	string	
message	较为详细的原因，描述为什么容器不是running状态。	false	string	

v1.NodeSystemInfo

NodeSystemInfo 是一组ids/uuids，以唯一地表示一个节点。

Name Kubernetes中文文档	描述	是否必须	类型	默认值
machineID	机器ID	true	string	
systemUUID	System UUID	true	string	
bootID	Boot ID	true	string	
kernelVersion	内核版本，通过 <code>uname -r</code> 命令获取(e.g. 3.16.0-0.bpo.4- amd64).	true	string	
osImage	操作系统发行版，从 <code>/etc/os-release</code> 获取。(e.g. Debian GNU/Linux 7 (wheezy)).	true	string	
containerRuntimeVersion	容器运行环境版本，通过运行时远程API调用获取。(e.g. docker://1.5.0).	true	string	
kubeletVersion	Kubelet版本。	true	string	
kubeProxyVersion	KubeProxy版本。	true	string	

v1.ServiceSpec

ServiceSpec描述了用户创建的service的属性。

Name	描述	是否	类型

Kubernetes中文文档		必须	
ports	service暴露出的ports。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#virtual-ips-and-service-proxies	true	v1.ServicePort array
selector	service会将流量转发给label与该selector匹配的pods。Pod的label key和value都必须匹配，才能接收到service的流量。如果将该字段置空，那么所有的pod都会被选中；如果不设置该字段，那么需要手动设置service的endpoint。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#overview	false	any
clusterIP	ClusterIP一般是master赋给service的IP地址。如果指定了该值，并且尚未被其他service占用，那么会把该值分配给当前service；如果其它service已经使用了该值，那么创建service将会失败。该字段有效的值可以是：None, 空字符串(""), 或者一个有效的IP地址。None通常用于不需要proxy的无头service。该字段一旦创建，不能更改。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#virtual-ips-and-service-proxies	false	string
type	暴露出的service的类型。必须是ClusterIP, NodePort, 或LoadBalancer。默认值为ClusterIP。详情查 看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#external-services	false	string
externallPs	externallPs是一组外网ip，集群中的节点会使用这些ip接收请求。这些IP不是由Kubernetes管理的，因此用户必须为打到这些节点上的流量负责。常用的例子是把外部	false	string array

	<p>的负载均衡器(haproxy,nginx)接入到Kubernetes系统。在v1之前,deprecatedPublicIPs字段实现了类似的功能。使用这个字段时, 调用者应该清除deprecatedPublicIPs字段。</p>		
deprecatedPublicIPs	deprecatedPublicIPs是一个过时的字段, 目前已经被externalIPs取代。为了保持向下兼容, 介于2016年8月20日之前, 这个字段仍然会出现在v1的API中。任何新的API修改都会丢弃这个字段。如果你同时设置了deprecatedPublicIPs 和externalIPs, 系统将使用deprecatedPublicIPs。	false	string array
sessionAffinity	目前支持"ClientIP"和"None"。它用来维持会话亲和性, 目前仅支持基于client端IP的方式。该字段默认值为None (round-robin方式选择endpoint)。详情查看： http://releases.k8s.io/HEAD/docs/user-guide/services.md#virtual-ips-and-service-proxies	false	string
loadBalancerIP	只有当ServiceType为LoadBalancer时, 此字段才生效。系统将依据该字段指定的IP创建负载均衡器。创建负载均衡器的操作还要依赖云服务提供商底层的网络支持。如果云服务供应商底层不支持该特性, 那么该字段会被忽略。	false	string

v1.Pod

Pod是存在于一个宿主机上容器的集合, 由client端创建, 然后由server端分配到宿主机上。

Name	描述	是否必须	类型
Kubernetes中文文档			
kind	<p>Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。</p> <p>详情查看：Kind 是指当前对象代表的REST资源类型。client端向endpoint提交请求时，server端可以根据client提交到的endpoint来推断出这个字段的值。该字段的值是不可变的，以驼峰风格显示。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#types-kinds</p>	false	string
apiVersion	<p>APIVersion 定义了表征方式的版本信息，apiVersion不同意味着当前对象表述出来可能是不一样的。server端会自动识别该字段并将其转换为内部可识别的类型，如果无法识别该字段，则可能拒绝接收该对象。详情查看： http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#resources</p>	false	string
metadata	<p>标准对象元数据。详情查 看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#metadata</p>	false	v1.ObjectMeta
spec	<p>Pod期望行为的规格。详情查 看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.PodSpec
status	<p>最近一次观察到的Pod的状态，得到的数据可能与当前的实际情况不一致。该字段由系统生成的，是一个只读字段。详情查 看：http://releases.k8s.io/HEAD/docs/devel/api-conventions.md#spec-and-status</p>	false	v1.PodStatus

v1.NodeSpec

NodeSpec描述了节点创建时拥有的属性。
Kubernetes中文文档

Name	描述	是否必须
podCIDR	PodCIDR表示分配到该节点上的Pod IP地址段。	false
externalID	通过外部数据库给节点设置的外部ID (比如云服务提供商)。该字段已废弃。	false
providerID	云服务供应商赋予的节点ID，以<ProviderName>://<ProviderSpecificNodeID>的格式显示。	false
unschedulable	当前节点是否不可调度，即是否可以将新pod分配到该节点上。 默认情况下，当前节点是可调度的。详情查看： http://releases.k8s.io/HEAD/docs/admin/node.md#manual-node-administration "	false

v1.EndpointAddress

EndpointAddress是描述单个IP地址对象的元组。

Name	描述	是否必须	类型	默认值
ip	endpoint的IP, 不能是环路 loopback (127.0.0.0/8), link-local (169.254.0.0/16), 和 link-local multicast ((224.0.0.0/24)。	true	string	
targetRef	指向一个对象的引用，该对象提供endpoint。	false	v1.ObjectReference	

v1.DaemonEndpoint

DaemonEndpoint包含了单个后台endpoint的信息。
Kubernetes中文文档

Name	描述	是否必须	类型	默认值
Port	特定endpoint的端口号。	true	integer (int32)	

any

代表无类型的JSON map, 详情查看关于此类对象的结构。

Last updated 2015-11-06 18:46:00 UTC

kubectl

译者：hurf 校对：无

使用kubectl来管理Kubernetes集群。

摘要

使用kubectl来管理Kubernetes集群。

可以在<https://github.com/kubernetes/kubernetes>找到更多的信息。

kubectl

选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP连接不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl annotate - 更新资源的注解。](#)

- [kubectl api-versions](#) - 以“组/版本”的格式输出服务端支持的API版本。
- [kubectl apply](#) - 通过文件名或控制台输入，对资源进行配置。
- [kubectl attach](#) - 连接到一个正在运行的容器。
- [kubectl autoscale](#) - 对replication controller进行自动伸缩。
- [kubectl cluster-info](#) - 输出集群信息。
- [kubectl config](#) - 修改kubeconfig配置文件。
- [kubectl create](#) - 通过文件名或控制台输入，创建资源。
- [kubectl delete](#) - 通过文件名、控制台输入、资源名或者label selector删除资源。
- [kubectl describe](#) - 输出指定的一个/多个资源的详细信息。
- [kubectl edit](#) - 编辑服务端的资源。
- [kubectl exec](#) - 在容器内部执行命令。
- [kubectl expose](#) - 输入replication controller, service或者pod，并将其暴露为新的kubernetes service。
- [kubectl get](#) - 输出一个/多个资源。
- [kubectl label](#) - 更新资源的label。
- [kubectl logs](#) - 输出pod中一个容器的日志。
- [kubectl namespace](#) - (已停用) 设置或查看当前使用的namespace。
- [kubectl patch](#) - 通过控制台输入更新资源中的字段。
- [kubectl port-forward](#) - 将本地端口转发到Pod。
- [kubectl proxy](#) - 为Kubernetes API server启动代理服务器。
- [kubectl replace](#) - 通过文件名或控制台输入替换资源。
- [kubectl rolling-update](#) - 对指定的replication controller执行滚动升级。
- [kubectl run](#) - 在集群中使用指定镜像启动容器。
- [kubectl scale](#) - 为replication controller设置新的副本数。
- [kubectl stop](#) - (已停用) 通过资源名或控制台输入安全删除资源。
- [kubectl version](#) - 输出服务端和客户端的版本信息。

kubectl annotate

译者：hurf 校对：无

更新某个资源的注解

摘要

更新一个或多个资源的注解。注解是一个键值对，它可以包含比label更多的信息，并且可能是机读数据。注解用来存储那些辅助的、非区分性的信息，特别是那些为外部工具或系统扩展插件使用的数据。如果--overwrite设为true，将会覆盖现有的注解，否则试图修改一个注解的值将会抛出错误。如果设置了--resource-version，那么将会使用指定的这个版本，否则将使用当前版本。

支持的资源包括但不限于（大小写不限）：pods (po)、services (svc)、replicationcontrollers (rc)、nodes (no)、events (ev)、componentstatuses (cs)、limitranges (limits)、persistentvolumes (pv)、persistentvolumeclaims (pvc)、resourcequotas (quota)和secrets。

```
kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--r
```

示例

```
# 更新pod "foo"，设置其注解description的值为my frontend。
# 如果同一个注解被赋值了多次，只保存最后一次设置的值。
$ kubectl annotate pods foo description='my frontend'

# 更新“pod.json”文件中type和name字段指定的pod的注解。
$ kubectl annotate -f pod.json description='my frontend'

# 更新pod “foo”，设置其注解description的值为my frontend running nginx，已有的值将被覆盖。
$ kubectl annotate --overwrite pods foo description='my frontend running nginx'

# 更新同一namespace下所有的pod。
$ kubectl annotate pods --all description='my frontend running nginx'

# 仅当pod “foo”当前版本为1时，更新其注解
$ kubectl annotate pods foo description='my frontend running nginx' --resource-version=1

# 更新pod “foo”，删除其注解description。
# 不需要--override选项。
$ kubectl annotate pods foo description-
```

选项

```
--all[=false]: 选择namespace中所有指定类型的资源。
-f, --filename=[]: 用来指定待升级资源的文件名, 目录名或者URL。
--overwrite[=false]: 如果设置为true, 允许覆盖更新注解, 否则拒绝更新已存在的注解。
--resource-version"": 如果不为空, 仅当资源当前版本和指定版本相同时才能更新注解。仅当更新单个资
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的H
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl api-versions

译者：hurf 校对：无

以“组/版本”的格式输出服务端支持的API版本。

摘要

以“组/版本”的格式输出服务端支持的API版本。

```
kubectl api-versions
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl apply

译者：hurff 校对：无

通过文件名或控制台输入，对资源进行配置。

摘要

通过文件名或控制台输入，对资源进行配置。

接受JSON和YAML格式的描述文件。

```
kubectl apply -f FILENAME
```

示例

```
# 将pod.json中的配置应用到pod
$ kubectl apply -f ./pod.json

# 将控制台输入的JSON配置应用到Pod
$ cat pod.json | kubectl apply -f -
```

选项

```
-f, --filename=[]: 包含配置信息的文件名，目录名或者URL。
-o, --output="": 输出格式，使用"-o name"来输出简短格式（资源类型/资源名）。
--schema-cache-dir="/tmp/kubectl.schema": 如果不为空，将API schema缓存为指定文件，默认缓存。
--validate[=true]: 如果为true，在发送到服务端前先使用schema来验证输入。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl attach

译者：hurff 校对：无

连接到一个正在运行的容器。

摘要

连接到现有容器中一个正在运行的进程。

```
kubectl attach POD -c CONTAINER
```

示例

```
# 获取正在运行中的pod 123456-7890的输出， 默认连接到第一个容器
$ kubectl attach 123456-7890

# 获取pod 123456-7890中ruby-container的输出
$ kubectl attach 123456-7890 -c ruby-container date

# 切换到终端模式， 将控制台输入发送到pod 123456-7890的ruby-container的“bash”命令，并将其输出到控制台。
# 错误控制台的信息发送回客户端。
$ kubectl attach 123456-7890 -c ruby-container -i -t
```

选项

```
-c, --container"": 容器名。
-i, --stdin[=false]: 将控制台输入发送到容器。
-t, --tty[=false]: 将标准输入控制台作为容器的控制台输入。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl cluster-info

译者：hurf 校对：无

显示集群信息。

摘要

显示master节点的地址和包含kubernetes.io/cluster-service=true的service（系统service）。

```
kubectl cluster-info
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true，将不会检查服务器凭证的有效性，这会导致你的HTTP请求使用明文。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时，忽略堆栈信息。
--log-dir"": 如果不为空，将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台，不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空，命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块，格式如下：pattern=N，使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl config

译者：hurff 校对：无

修改kubeconfig配置文件。

摘要

使用config的子命令修改kubeconfig配置文件，如“kubectl config set current-context my-context”。

配置文件的读取遵循如下规则：

The loading order follows these rules:

1. 如果指定了--kubeconfig选项，那么只有指定的文件被加载。此选项只能被设置一次，并且不会合并其他文件。If
2. 如果设置了\$KUBECONFIG环境变量，将同时使用此环境变量指定的所有文件列表（使用操作系统默认的顺序），所有
3. 如果前两项都没有设置，将使用 \${HOME}/.kube/config，并且不会合并其他文件。

kubectl config SUBCOMMAND

选项

--kubeconfig=""： 使用特定的配置文件。

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。
- [kubectl config set](#) - 在kubeconfig配置文件中设置一个单独的值。
- [kubectl config set-cluster](#) - 在kubeconfig配置文件中设置一个集群项。
- [kubectl config set-context](#) - 在kubeconfig配置文件中设置一个环境项。
- [kubectl config set-credentials](#) - 在kubeconfig配置文件中设置一个用户项。
- [kubectl config unset](#) - 在kubeconfig配置文件中清除一个单独的值。
- [kubectl config use-context](#) - 使用kubeconfig中的一个环境项作为当前配置。
- [kubectl config view](#) - 显示合并后的kubeconfig设置, 或者一个指定的kubeconfig配置文件。

kubectl config set-cluster

译者：hurf 校对：无

在kubeconfig配置文件中设置一个集群项。

摘要

在kubeconfig配置文件中设置一个集群项。如果指定了一个已存在的名字，将合并新字段并覆盖旧字段。

```
kubectl config set-cluster NAME [--server=server] [--certificate-authority=path/to/certfi
```

示例

```
# 仅设置e2e集群项中的server字段，不影响其他字段
$ kubectl config set-cluster e2e --server=https://1.2.3.4

# 向e2e集群项中添加认证鉴权数据
$ kubectl config set-cluster e2e --certificate-authority=~/.kube/e2e/kubernetes.ca.crt

# 取消dev集群项中的证书检查
$ kubectl config set-cluster e2e --insecure-skip-tls-verify=true
```

选项

```
--api-version"": 设置kuebconfig配置文件中集群选项中的api-version。
--certificate-authority"": 设置kuebconfig配置文件中集群选项中的certificate-authority路径。
--embed-certs=false: 设置kuebconfig配置文件中集群选项中的embed-certs开关。
--insecure-skip-tls-verify=false: 设置kuebconfig配置文件中集群选项中的insecure-skip-tls-verify开关。
--server"": 设置kuebconfig配置文件中集群选项中的server。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl config set-context

译者：hurff 校对：无

在kubeconfig配置文件中设置一个环境项。

摘要

在kubeconfig配置文件中设置一个环境项。如果指定了一个已存在的名字，将合并新字段并覆盖旧字段。

```
kubectl config set-context NAME [--cluster=clusterNickname] [--user=userNickname] [--na
```

示例

```
# 设置gce环境项中的user字段，不影响其他字段。  
$ kubectl config set-context gce --user=cluster-admin
```

选项

```
--cluster="": 设置kuebconfig配置文件中环境选项中的集群。  
--namespace="": 设置kuebconfig配置文件中环境选项中的命名空间。  
--user="": 设置kuebconfig配置文件中环境选项中的用户。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl config set-credentials

译者：hurf 校对：无

在kubeconfig配置文件中设置一个用户项。

摘要

在kubeconfig配置文件中设置一个用户项。如果指定了一个已存在的名字，将合并新字段并覆盖旧字段。

客户端证书设置：`--client-certificate=certfile --client-key=keyfile`

不记名令牌设置：`--token=bearer_token`

基础认证设置：`--username=basic_user --password=basic_password`

不记名令牌和基础认证不能同时使用。

```
kubectl config set-credentials NAME [--client-certificate=path/to/certfile] [--client-key=
```

示例

```
# 仅设置cluster-admin用户项下的client-key字段，不影响其他值
$ kubectl config set-credentials cluster-admin --client-key=~/.kube/admin.key

# 为cluster-admin用户项设置基础认证选项
$ kubectl config set-credentials cluster-admin --username=admin --password=uXFGweU9l35qci

# 为cluster-admin用户项开启证书验证并设置证书文件路径
$ kubectl config set-credentials cluster-admin --client-certificate=~/.kube/admin.crt --e
```

选项

- `--client-certificate="":` 设置kubeconfig配置文件中用户选项中的证书文件路径。
- `--client-key="":` 设置kubeconfig配置文件中用户选项中的证书密钥路径。
- `--embed-certs=false:` 设置kubeconfig配置文件中用户选项中的embed-certs开关。
- `--password="":` 设置kubeconfig配置文件中用户选项中的密码。
- `--token="":` 设置kubeconfig配置文件中用户选项中的令牌。
- `--username="":` 设置kubeconfig配置文件中用户选项中的用户名。

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。

kubectl config set

译者：hurf 校对：无

在kubeconfig配置文件中设置一个单独的值。

摘要

在kubeconfig配置文件中设置一个单独的值

`PROPERTY_NAME` 使用“.”进行分隔，每段代表一个属性名或者map的键，`map`的键不能包含“.”。`PROPERTY_VALUE` 需要设置的新值。

```
kubectl config set PROPERTY_NAME PROPERTY_VALUE
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```



参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。

kubectl config unset

译者：hurf 校对：无

在kubeconfig配置文件中清除一个单独的值。

摘要

在kubeconfig配置文件中清除一个单独的值。 PROPERTY_NAME 使用“.”进行分隔，每段代表一个属性名或者map的键，map的键不能包含“.”。

```
kubectl config unset PROPERTY_NAME
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端文件路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl config use-context

译者：hurf 校对：无

使用kubeconfig中的一个环境项作为当前配置。

摘要

使用kubeconfig中的一个环境项作为当前配置。

```
kubectl config use-context CONTEXT_NAME
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl config view

译者：hurf 校对：无

显示合并后的kubeconfig设置，或者一个指定的kubeconfig配置文件。

摘要

显示合并后的kubeconfig设置，或者一个指定的kubeconfig配置文件。用户可使用--output=template --template=TEMPLATE来选择输出指定的值。

```
kubectl config view
```

示例

```
# 显示合并后的kubeconfig设置
$ kubectl config view

# 获取e2e用户的密码
$ kubectl config view -o template --template='{{range .users}}{{ if eq .name "e2e" }}{{ i
```

选项

```
--flatten[=false]: 将读取的kubeconfig配置文件扁平输出为自包含的结构（对创建可迁移的kubeconfig有用）。
--merge=true: 按照继承关系合并所有的kubeconfig配置文件。
--minify[=false]: 如果为true，不显示目前环境未使用到的任何信息。
--no-headers[=false]: 当使用默认输出格式时不打印标题栏。
-o, --output"": 输出格式，只能使用json|yaml|wide|name|go-template=...|go-template-file=...
--output-version"": 输出资源使用的API版本（默认使用api-version）。
--raw[=false]: 显示未经格式化的字节信息。
-a, --show-all[=false]: 打印输出时，显示所有的资源（默认隐藏状态为terminated的pod）。
--sort-by"": 如果不为空，对输出的多个结果根据指定字段进行排序。该字段使用jsonpath表达式（如"obj.metadata.name"）。
--template"": 当指定了-o=go-template或-o=go-template-file时使用的模板字符串或者模板文件。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl create

译者：hurff 校对：无

通过文件名或控制台输入，创建资源。

摘要

通过文件名或控制台输入，创建资源。

接受JSON和YAML格式的描述文件。

```
kubectl create -f FILENAME
```

示例

```
# 使用pod.json文件创建一个pod
$ kubectl create -f ./pod.json

# 通过控制台输入的JSON创建一个pod
$ cat pod.json | kubectl create -f -
```

选项

```
-f, --filename=[]: 用以创建资源的文件名，目录名或者URL。
-o, --output="": 输出格式，使用"-o name"来输出简短格式（资源类型/资源名）。
--schema-cache-dir="/tmp/kubectl.schema": 如果不为空，将API schema缓存为指定文件，默认缓存。
--validate[=true]: 如果为true，在发送到服务端前先使用schema来验证输入。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl delete

译者：hurff 校对：无

通过文件名、控制台输入、资源名或者label selector删除资源。

摘要

通过文件名、控制台输入、资源名或者label selector删除资源。接受JSON和YAML格式的描述文件。

只能指定以下参数类型中的一种：文件名、资源类型和名称、资源类型和label selector。注意：delete命令不检查资源版本，如果有人在你进行删除操作的同时进行更新操作，他所做的更新将随资源同时被删除。

```
kubectl delete ([ -f FILENAME ] | TYPE [(NAME | -l label | --all)])
```

示例

```
# 通过pod.json文件中指定的资源类型和名称删除一个pod
$ kubectl delete -f ./pod.json

# 通过控制台输入的JSON所指定的资源类型和名称删除一个pod
$ cat pod.json | kubectl delete -f -

# 删除所有名为“baz”和“foo”的pod和service
$ kubectl delete pod,service baz foo

# 删除所有带有label name=myLabel的pod和service
$ kubectl delete pods,services -l name=myLabel

# 删除UID为1234-56-7890-234234-456456的pod
$ kubectl delete pod 1234-56-7890-234234-456456

# 删除所有的pod
$ kubectl delete pods --all
```

选项

```
--all[=false]: 使用[-all]选择所有指定的资源。
--cascade[=true]: 如果为true, 级联删除指定资源所管理的其他资源(例如: 被replication controller管理的Pod)。
-f, --filename=[]: 用以指定待删除资源的文件名, 目录名或者URL。
--grace-period=-1: 安全删除资源前等待的秒数。如果为负值则忽略该选项。
--ignore-not-found[=false]: 当待删除资源未找到时, 也认为删除成功。如果设置了--all选项, 则默认为true。
-o, --output="": 输出格式, 使用"-o name"来输出简短格式(资源类型/资源名)。
-l, --selector="": 用于过滤资源的Label。
--timeout=0: 删除资源的超时设置, 0表示根据待删除资源的大小由系统决定。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version="": 和服务端交互使用的API版本。
--certificate-authority="": 用以进行认证授权的.cert文件路径。
--client-certificate="": TLS使用的客户端证书路径。
--client-key="": TLS使用的客户端密钥路径。
--cluster="": 指定使用的kubeconfig配置文件中的集群名。
--context="": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig="": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir="": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace="": 如果不为空, 命令将使用此namespace。
--password="": API Server进行简单认证使用的密码。
-s, --server="": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token="": 认证到API Server使用的令牌。
--user="": 指定使用的kubeconfig配置文件中的用户名。
--username="": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl describe

译者：hurff 校对：无

输出指定的一个/多个资源的详细信息。

摘要

输出指定的一个/多个资源的详细信息。

此命令组合调用多条API，输出指定的一个或者一组资源的详细描述。

```
$ kubectl describe TYPE NAME_PREFIX
```

首先检查是否有精确匹配TYPE和NAME_PREFIX的资源，如果没有，将会输出所有名称以NAME_PREFIX开头的资源详细信息。

支持的资源包括但不限于（大小写不限）：pods (po)、services (svc)、replicationcontrollers (rc)、nodes (no)、events (ev)、componentstatuses (cs)、limitranges (limits)、persistentvolumes (pv)、persistentvolumeclaims (pvc)、resourcequotas (quota)和secrets。

```
kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | -l label] | TYPE/NAME)
```

示例

```
# 描述一个node
$ kubectl describe nodes kubernetes-minion-emt8.c.myproject.internal

# 描述一个pod
$ kubectl describe pods/nginx

# 描述pod.json中的资源类型和名称指定的pod
$ kubectl describe -f pod.json

# 描述所有的pod
$ kubectl describe pods

# 描述所有包含label name=myLabel的pod
$ kubectl describe po -l name=myLabel

# 描述所有被replication controller "frontend"管理的pod (rc创建的pod都以rc的名字作为前缀)
$ kubectl describe pods frontend
```

选项

```
-f, --filename=[]: 用来指定待描述资源的文件名, 目录名或者URL。  
-l, --selector"": 用于过滤资源的Label。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。  
--api-version"": 和服务端交互使用的API版本。  
--certificate-authority"": 用以进行认证授权的.cert文件路径。  
--client-certificate"": TLS使用的客户端证书路径。  
--client-key"": TLS使用的客户端密钥路径。  
--cluster"": 指定使用的kubeconfig配置文件中的集群名。  
--context"": 指定使用的kubeconfig配置文件中的环境名。  
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。  
--kubeconfig"": 命令行请求使用的配置文件路径。  
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。  
--log-dir"": 如果不为空, 将日志文件写入此目录。  
--log-flush-frequency=5s: 刷新日志的最大时间间隔。  
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。  
--match-server-version[=false]: 要求服务端和客户端版本匹配。  
--namespace"": 如果不为空, 命令将使用此namespace。  
--password"": API Server进行简单认证使用的密码。  
-s, --server"": Kubernetes API Server的地址和端口号。  
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。  
--token"": 认证到API Server使用的令牌。  
--user"": 指定使用的kubeconfig配置文件中的用户名。  
--username"": API Server进行简单认证使用的用户名。  
--v=0: 指定输出日志的级别。  
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl edit

译者：hurf 校对：无

编辑服务端的资源。

摘要

使用系统默认编辑器编辑服务端的资源。

edit命令允许你直接编辑使用命令行工具获取的任何资源。此命令将打开你通过KUBE_EDITOR, GIT_EDITOR 或者EDITOR环境变量定义的编辑器，或者直接使用“vi”。你可以同时编辑多个资源，但所有的变更都只会一次性提交。除了命令行参数外，此命令也接受文件名，但所直指定的文件必须使用早于当前的资源版本。

所编辑的文件将使用默认的API版本输出，或者通过--output-version选项显式指定。默认的输出格式是YAML，如果你需要使用JSON格式编辑，使用-o json选项。

如果当更新资源的时候出现错误，将会在磁盘上创建一个临时文件，记录未成功的更新。更新资源时最常见的错误是其他人也在更新服务端的资源。当这种情况发生时，你需要将你所作的更改应用到最新版本的资源上，或者编辑保存的临时文件，使用最新的资源版本。

```
kubectl edit (RESOURCE/NAME | -f FILENAME)
```

示例

```
# 编辑名为“docker-registry”的service
$ kubectl edit svc/docker-registry

# 使用一个不同的编辑器
$ KUBE_EDITOR="nano" kubectl edit svc/docker-registry

# 编辑名为“docker-registry”的service，使用JSON格式、v1 API版本
$ kubectl edit svc/docker-registry --output-version=v1 -o json
```

选项

```
-f, --filename=[]: 用来指定待编辑资源的文件名，目录名或者URL。
-o, --output="yaml": 输出格式，可选yaml或者json中的一种。
--output-version="": 输出资源使用的API版本（默认使用api-version）。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。

kubectl exec

译者：hurff 校对：无

在容器内部执行命令。

摘要

在容器内部执行命令。

```
kubectl exec POD [-c CONTAINER] -- COMMAND [args...]
```

示例

```
# 默认在pod 123456-7890的第一个容器中运行“date”并获取输出
$ kubectl exec 123456-7890 date

# 在pod 123456-7890的容器ruby-container中运行“date”并获取输出
$ kubectl exec 123456-7890 -c ruby-container date

# 切换到终端模式，将控制台输入发送到pod 123456-7890的ruby-container的“bash”命令，并将其输出到控制台。
# 错误控制台的信息发送回客户端。
$ kubectl exec 123456-7890 -c ruby-container -i -t -- bash -il
```

选项

- c, --container="": 容器名。如果未指定，使用pod中的一个容器。
- p, --pod="": Pod名。
- i, --stdin[=false]: 将控制台输入发送到容器。
- t, --tty[=false]: 将标准输入控制台作为容器的控制台输入。

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl - 使用kubectl来管理Kubernetes集群。](#)

kubectl logs

译者：hurff 校对：无

输出pod中一个容器的日志。

摘要

输出pod中一个容器的日志。如果pod只包含一个容器则可以省略容器名。

```
kubectl logs [-f] [-p] POD [-c CONTAINER]
```

示例

```
# 返回仅包含一个容器的pod nginx的日志快照
$ kubectl logs nginx

# 返回pod ruby中已经停止的容器web-1的日志快照
$ kubectl logs -p -c ruby web-1

# 持续输出pod ruby中的容器web-1的日志
$ kubectl logs -f -c ruby web-1

# 仅输出pod nginx中最近的20条日志
$ kubectl logs --tail=20 nginx

# 输出pod nginx中最近一小时内产生的所有日志
$ kubectl logs --since=1h nginx
```

选项

```
-c, --container"": 容器名。
-f, --follow[=false]: 指定是否持续输出日志。
--interactive[=true]: 如果为true, 当需要时提示用户进行输入。默认为true。
--limit-bytes=0: 输出日志的最大字节数。默认无限制。
-p, --previous[=false]: 如果为true, 输出pod中曾经运行过, 但目前已终止的容器的日志。
--since=0: 仅返回相对时间范围, 如5s、2m或3h, 之内的日志。默认返回所有日志。只能同时使用since和since-time。
--since-time"": 仅返回指定时间 (RFC3339格式) 之后的日志。默认返回所有日志。只能同时使用since和since-time。
--tail=-1: 要显示的最新的日志条数。默认为-1, 显示所有的日志。
--timestamps[=false]: 在日志中包含时间戳。
```

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。

kubectl version

译者：hurff 校对：无

输出服务端和客户端的版本信息。

摘要

输出服务端和客户端的版本信息。

```
kubectl version
```

选项

-c, --client[=false]: 仅输出客户端版本（无需连接服务器）。

继承自父命令的选项

```
--alsologtostderr[=false]: 同时输出日志到标准错误控制台和文件。
--api-version"": 和服务端交互使用的API版本。
--certificate-authority"": 用以进行认证授权的.cert文件路径。
--client-certificate"": TLS使用的客户端证书路径。
--client-key"": TLS使用的客户端密钥路径。
--cluster"": 指定使用的kubeconfig配置文件中的集群名。
--context"": 指定使用的kubeconfig配置文件中的环境名。
--insecure-skip-tls-verify[=false]: 如果为true, 将不会检查服务器凭证的有效性, 这会导致你的HTTP请求不安全。
--kubeconfig"": 命令行请求使用的配置文件路径。
--log-backtrace-at=:0: 当日志长度超过定义的行数时, 忽略堆栈信息。
--log-dir"": 如果不为空, 将日志文件写入此目录。
--log-flush-frequency=5s: 刷新日志的最大时间间隔。
--logtostderr[=true]: 输出日志到标准错误控制台, 不输出到文件。
--match-server-version[=false]: 要求服务端和客户端版本匹配。
--namespace"": 如果不为空, 命令将使用此namespace。
--password"": API Server进行简单认证使用的密码。
-s, --server"": Kubernetes API Server的地址和端口号。
--stderrthreshold=2: 高于此级别的日志将被输出到错误控制台。
--token"": 认证到API Server使用的令牌。
--user"": 指定使用的kubeconfig配置文件中的用户名。
--username"": API Server进行简单认证使用的用户名。
--v=0: 指定输出日志的级别。
--vmodule=: 指定输出日志的模块, 格式如下: pattern=N, 使用逗号分隔。
```

参见

- [kubectl](#) - 使用kubectl来管理Kubernetes集群。

故障排查

译者：林帆 校对：无

在使用Kubernetes时，用户常常会遇到一些错误和迷惑，这个小节的内容将针对常见的问题进行解答。我们将这些问题分为以下两个部分：

- 应用程序相关的故障排查：当遇到将代码部署到Kubernetes时发生的各类错误时，请参考这部分内容。
- 集群相关的故障排查：当集群系统的管理者发现Kubernetes的集群工作不正常时，请参考这部分内容。

用户也可以查看[已知问题列表](#)来了解当前所使用的版本中有哪些我们已经发现的系统问题。

寻求帮助

如果以上的故障排查文档不能解决你所遇到的问题，还可以通过下面这几种方式寻求社区的帮助。

疑问

如果你对Kubernetes还不熟悉，文档的“用户指引”部分也许能帮助到你。

我们总结了一些用户经常遇到的问题，并分成下面几个方面记录下来了。

- 日常使用的常见问题
- 开发和调试的常见问题
- 服务相关的常见问题

此外，在Stack Overflow网站上也有相关的主题通道可供提问和参考。

- [Kubernetes相关的提问](#)
- [谷歌容器引擎GKE相关的提问](#)

更多的帮助内容

如果已经提到的这些途径还不足以解答你所遇到的问题，下面提供了其他的一些获取答案的途径。

Stack Overflow

社区中的其他人也许遇到过和你相同的问题，Kubernetes的开发者小组会订阅并定期的解答Stack Overflow上[包含有kubernetes标签的提问](#)。如果所有的这些提问都没有包含你所遇到的问题，你可以通过[这个链接](#)把问题告诉我们。

Slack

Kubernetes开发小组的成员常常会出没在Slack的 `#kubernetes-users` 频道中，用户可以在这里直接将问题抛给第一线的开发者，并进行交流。Kubernetes的Slack开发者频道入口地址是[这里](#)，第一次进入频道前，用户需要先到[这里](#)进行注册。我们欢迎倾听到各种不同的声音和问题。

邮件组

谷歌容器引擎小组有个专用的邮件群组，地址是：google-containers@googlegroups.com

Bug和需求

如果你发现的问题看起来像是一个潜在的Bug，或者你希望为Kubernetes的开发提出一些需求，请直接提交到Kubernetes在GitHub的[问题追踪页面](#)里。

需要注意的是，在你到这个页面提出任何问题之前，请先在页面中搜索一下，确认没有其他人已经提过类似的问题。

当提出Bug时，请务必包含以下信息，以便我们能够尽快的重现并定位故障：

- Kubernetes的版本：`kubectl version` 命令会告诉你所需的信息
- 运行的环境（云服务商名称），操作系统发行版类型和版本，网络配置以及所用容器（例如Docker）的版本
- 重现故障的步骤

应用程序相关的故障排查

译者：林帆 校对：无

这个小节将帮助读者调试部署在Kubernetes的应用程序运行不正常的情况。本篇并不会包含如何调试集群组建过程中出现错误的问题，这些内容在文档的下一节中进行介绍。

内容提要

- 应用程序故障排查
 - 常见问题
 - 故障诊断
 - 排查Pods的故障
 - Pod始终处于Pending状态
 - Pod始终处于Waiting状态
 - Pod一直崩溃或运行不正常
 - Pod在运行但没有如预期工作
 - 排查Replication Controllers的故障
 - 排查Services的故障
 - 服务没有端点信息
 - 网络流量没有正确的转发
 - 其他问题

常见问题

一些经常被提到的问题都更新在Kubernetes项目GitHub的[常见问题](#)页面。

故障诊断

故障排查的第一步是对故障进行分类。一般来说，应用程序的故障可以分为以下几个方面：

- Pods的故障
- ReplicationControllers的故障
- Services的故障

排查Pods的故障

检查Pod的问题首先应该了解Pod所处的状况。下面这个命令能够获得Pod当前的状态和近期的事件列表：

```
$ kubectl describe pods ${POD_NAME}
```

确认清楚在Pod以及其中每一个容器的状态，是否都处于 `Running`？通过观察容器的已运行时间判断它是否刚刚才重新启动过？

根据不同的运行状态，用户应该采取不同的调查措施。

Pod始终处于Pending状态

如果Pod保持在 `Pending` 的状态，这意味着它无法被正常的调度到一个节点上。通常来说，这是由于某种系统资源无法满足Pod运行的需求。观察刚才 `kubectl describe` 命令的输出内容，其中应该包括了能够判断错误原因的消息。常见的原因有以下这些：

- 系统没有足够的资源：你也许已经用尽了集群中所有的CPU或内存资源。对于这种情况，你需要清理一些不在需要的Pod，调整它们所需的资源量，或者向集群中增加新的节点。在[计算资源文档](#)有更详细的说明。
- 用户指定了 `hostPort`：通过 `hostPort` 用户能够将服务暴露到指定的主机端口上，但这样会限制Pod能够被调度运行的节点。在大多数情况下，`hostPort` 配置都是没有必要 的，用户应该采用Service的方式暴露其对外的服务。如果用户确实必须使用 `hostPort` 的功能，那么此Pod最多只能部署到和集群节点相同的数目。

Pod始终处于Waiting状态

Pod处在 `Waiting` 的状态，说明它已经被调度分配到了一个工作节点，然而它无法在那个节点上运行。同样的，在刚才 `kubectl describe` 命令的输出内容中，应该包含有更详细的错误信息。最经常导致Pod始终 `Waiting` 的原因是无法下载所需的镜像（译者注：由于国内特殊的网络环境，这类问题出现得特别普遍）。用户可以从下面三个方面进行排查：

- 请确保正确书写了镜像的名称
- 请检查所需镜像是否已经推送到了仓库中
- 手工的在节点上运行一次 `docker pull <镜像名称>` 检测镜像能否被正确的拉取下来

Pod一直崩溃或运行不正常

首先，查看正在运行容器的日志。

```
$ kubectl logs <Pod名称> <Pod中的容器名称>
```

如果容器之前已经崩溃过，通过以下命令可以获得容器前一次运行的日志内容。

```
$ kubectl logs --previous <Pod名称> <Pod中的容器名称>
```

此外，还可以使用 `exec` 命令在指定的容器中运行任意的调试命令。

```
$ kubectl exec <Pod名称> -c <Pod中的容器名称> -- <任意命令> <命令参数列表...>
```

值得指出的是，对于只有一个容器的Pod情况，`-c <Pod中的容器名称>` 这个参数是可以省略的。

例如查看一个运行中的Cassandra日志文件内容，可以参考下面这个命令：

```
$ kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

要是上面的这些办法都不奏效，你也可以找到正在运行该Pod的主机地址，然后使用SSH登陆进去检测。但这是在确实迫不得已的情况下才会采用的措施，通常使用Kubernetes暴露的API应该足够获得所需的环境信息。因此，如果当你发现自己不得不登陆到主机上去获取必要的信息数据时，不妨到Kubernetes的GitHub页面中给我们提一个功能需求（Feature Request），在需求中附上详细的使用场景以及为什么当前Kubernetes所提供的工具不能够满足需要。

Pod在运行但没有如预期工作

如果Pod没有按照预期的功能运行，有可能是由于在Pod描述文件中（例如你本地机器的 `mypod.yaml` 文件）存在一些错误，这些配置中的错误在Pod时创建并没有引起致命的故障。这些错误通常包括Pod描述的某些元素嵌套层级不正确，或是属性的名称书写有误（这些错误属性在运行时会被忽略掉）。举例来说，如果你把 `command` 属性误写为了 `commnd`，Pod 仍然会启动，但用户所期待运行的命令则不会被执行。

对于这种情况，首先应该尝试删掉正在运行的Pod，然后使用 `--validate` 参数重新运行一次。继续之前的例子，当执行 `kubectl create --validate -f mypod.yaml` 命令时，被误写为 `commnd` 的 `command` 指令会导致下面这样的错误：

```
I0805 10:43:25.129850    46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973    46757 schema.go:129] this may be a false alarm, see https://github.com/kubernetes/kubernetes/pulls/mypod
```

下一件事是检查当前apiserver运行Pod所使用的Pod描述文件内容是否与你想要创建的Pod内容（用户本地主机的那个yaml文件）一致。比如，执行 `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml` 命令将正在运行的Pod描述文件内容导出来保存为 `mypod-on-apiserver.yaml`，并将它和用户的Pod描述文件 `mypod.yaml` 进行对比。由于apiserver会

尝试自动补全一些缺失的Pod属性，在apiserver导出的Pod描述文件中有可能比本地的Pod描述文件多出若干行，这是正常的，但反之如果本地的Pod描述文件比apiserver导出的Pod描述文件多出了新的内容，则很可能暗示着当前运行的Pod使用了不正确的内容。

排查 Replication Controllers 的故障

Replication Controllers的逻辑十分直白，它的作用只是创建新的Pod副本，仅仅可能出现的错误便是它无法创建正确的Pod，对于这种情况，应该参考上面的『排查Pods的故障』部分进行检查。

也可以使用 `kubectl describe rc <控制器名称>` 来显示与指定Replication Controllers相关的事件信息。

排查 Services 的故障

Service为一系列后端Pod提供负载均衡的功能。有些十分常见的故障都可能导致Service无法正常工作，以下将提供对调试Service相关问题的参考。

首先，检查Service连接的服务提供端点（endpoint），对于任意一个Service对象，apiserver都会为其创建一个端点资源（译者注：即提供服务的IP地址和端口号）。

这个命令可以查看到Service的端口资源：

```
$ kubectl get endpoints <Service名称>
```

请检查这个命令输出端点信息中的端口号与实际容器提供服务的端口号是否一致。例如，如果你的Service使用了三个Nginx容器的副本（replicas），这个命令应该输出三个不同的IP地址的端点信息。

服务没有端点信息

如果刚刚的命令显示Service没有端点信息，请尝试通过Service的选择器找到具有相应标签的所有Pod。假设你的Service描述选择器内容如下：

```
...
spec:
  - selector:
      name: nginx
      type: frontend
```

可以使用以下命令找出相应的Pod：

```
$ kubectl get pods --selector=name=nginx,type=frontend
```

找到了符合标签的Pod后，首先确认这些被选中的Pod是正确，有无错选、漏选的情况。

如果被选中的Pod没有问题，则问题很可能出在这些Pod暴露的端口没有被正确的配置好。要是一个Service指定了 `containerPort`，但被选中的Pod并没有在配置中列出相应的端口，它们就不会出现在端点列表中。

请确保所用Pod的 `containerPort` 与Service的 `containerPort` 配置信息是一致的。

网络流量没有正确的转发

如果你能够连接到Service，但每次连接上就立即被断开，同时Service的端点列表内容是正确的，很可能是因为Kubernetes的kube-proxy服务无法连接到相应的Pod。

请检查以下几个方面：

- Pod是否在正常工作？从每个Pod的自动重启动次数可以作为有用的参考信息，前面介绍过的Pod错误排查也介绍了更详细的方法
- 能够直接连接到Pod提供的服务端口上吗？不妨获取到Pod的IP地址，然后尝试直接连接它，以验证Pod本身十分运行正确。
- 容器中的应用程序是否监听在Pod和Service中配置的那个端口？Kubernetes不会自动的映射端口号，因此如果应用程序监听在8080端口，务必保证Service和Pod的 `containerPort` 都配置为了8080。

其他问题

如果上述的这些步骤还不足以解答你所遇到的问题，也就是说你已经确认了相应的Service正在运行，并且具有恰当的端点资源，相应的Pod能够提供正确的服务，集群的DNS域名服务没有问题，IPtable的防火墙配置没有问题，kube-proxy服务也都运转正常。

请参看本文档故障排除部分的其他小节，以获取更加全面的故障处理信息。

集群相关的故障排查

译者：林帆 校对：无

日常使用的常见问题

译者：林帆 校对：无

开发和调试的常见问题

译者：林帆 校对：无

服务相关的常见问题

译者：林帆 校对：无