



# 周立 Kubernetes 开源书

书栈(BookStack.CN)

# 目 录

致谢

Docker与Kubernetes开源书

Introduction

01-什么是Kubernetes

02-安装单机版Kubernetes

03-使用Kubespray部署生产可用的Kubernetes集群 (1.11.2)

04-K8s组件

05-Kubernetes API

06-理解K8s对象

07-Name

08-Namespace

09-Label和Selector

10-Annotation

11-K8s架构及基本概念

12-Master与Node的通信

13-Node

14-Pod

15-Replica Set

16-Deployment

17-StatefulSet

18-Daemon Set

19-配置最佳实践

20-管理容器的计算资源

21-Kubernetes资源分配

22-将Pod分配到Node

23-容忍与污点

24-Secret

25-Pod优先级和抢占

26-Service

27-Ingress Resources

28-动态水平扩容

29-实战：使用K8s编排Wordpress博客

# 致谢

当前文档《周立 Kubernetes 开源书》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-03-14。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：周立 <https://github.com/itmuch/docker-book/tree/master/kubernetes>

文档地址：<http://www.bookstack.cn/books/itmuch-kubernetes>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# Docker与Kubernetes开源书

---

我的第二本开源书。本书地址：

- Gitee: <https://gitee.com/itmuch/docker-book>
- GitHub: <https://github.com/itmuch/docker-book>

之前写的《Spring Cloud开源书》，有兴趣的可详见：

- Gitee: <https://www.gitee.com/itmuch/spring-cloud-book>
- GitHub: <https://www.github.com/eacdy/spring-cloud-book>

本开源书，包含两部分：

## Docker

---

Docker部分，包括：

- 入门
- Dockerfile详解
- 镜像管理
- 工具
- 持久化
- 网络
- Docker Compose

七大主题，涵盖Docker常用命令、Dockerfile常用命令、网络、存储、Docker Compose等常见知识点，知识体系应该还是比较完备的。如果学习完，你应该具备如下能力：

- 常用的命令信手拈来
- Dockerfile编写无压力
- 能用Docker Compose快速构建容器环境
- 理解Docker网络、存储等知识点是怎么回事。

详见 [docker](#) 目录。

## Kubernetes

---

Kubernetes部分，是个人学习Kubernetes时，对官方文档的翻译。在官方翻译的基础上，结合自己的理解，做了一些批注。

由于是SOLO翻译，精力有限，无法翻译全部文档，而且翻译本身也是为自己学习服务的，不是闲的蛋疼翻译玩，又或者有什么功利心。从知识体系上来看，可能不是那么的完备.....不过其实常见的知识点在我的文档里也都包含了。

详见 [kubernetes](#) 目录。

TIPS: 就翻译质量来看, 吹个牛, 似乎目前还没有找到比我这个更好的。

## 如何使用

---

- 方法一、懒人用法: 直接下载根目录的 `Docker.pdf` 、 `Kubernetes.pdf` 阅读;
- 方法二、将代码clone到本地后, 使用Typora或Atom等Markdown阅读软件进行阅读;
- 方法三、前往<http://www.itmuch.com/categories/Docker/> 阅读

ENJOY IT!

## 广告时间

---

- 个人博客: <http://www.itmuch.com>
- 个人微信: `jumping_me`



干货小程序



干货公众号



# 简介

---

Kubernetes开源书。不啰嗦了，JUST READ IT.

# 什么是kubernetes

---

Kubernetes是一个旨在自动部署、扩展和运行应用容器的开源平台。

使用Kubernetes，您可以快速有效地回应客户需求：

- 快速、可预测地部署应用。
- 动态缩放您的应用。
- 无缝地推出新功能。
- 仅对需要的资源限制硬件的使用

我们的目标是构建一个生态系统，提供组件和工具以减轻在公共和私有云中运行应用程序的负担。

## Kubernetes是

- 可移植：共有、私有、混合、多云
- 可扩展：模块化、可插拔、提供Hook、可组合
- 自愈：自动放置、自动重启、自动复制、自动缩放

Google于2014年启动了Kubernetes项目。Kubernetes建立在Google在大规模运行生产工作负载方面十几年的经验之上，并结合了社区中最佳的创意和实践。

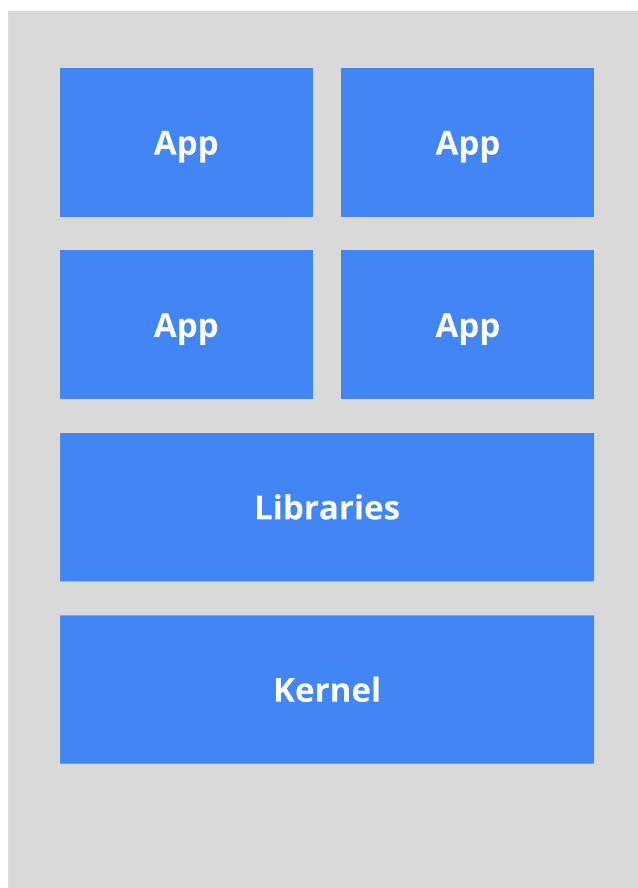
## 为什么使用容器

---

寻找你为啥要使用容器的原因？

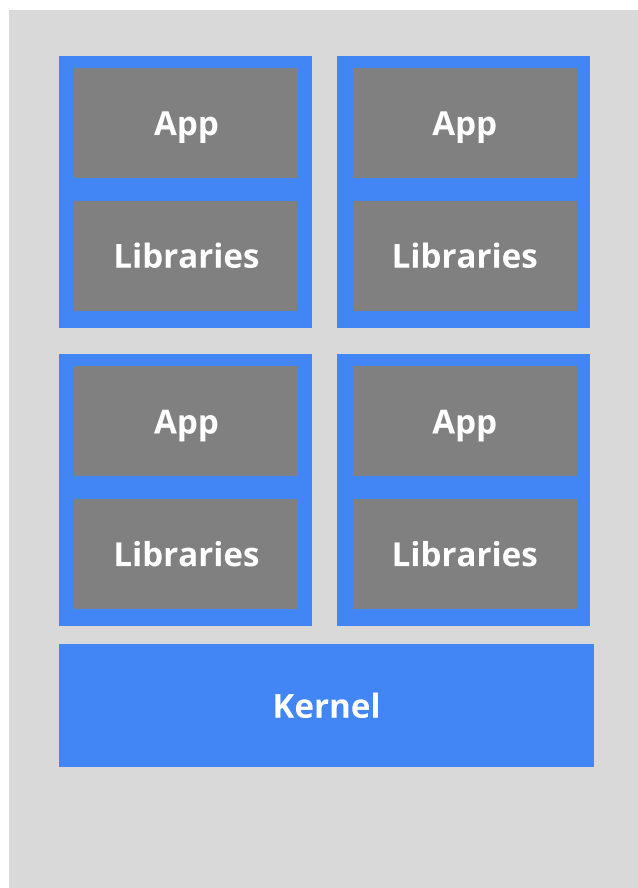


## The old way: Applications on host



*Heavyweight, non-portable  
Relies on OS package manager*

## The new way: Deploy containers



*Small and fast, portable  
Uses OS-level virtualization*

部署应用程序的旧方法是使用操作系统的软件包管理器在主机上安装应用程序。这种方式，存在可执行文件、配置、库和生命周期与操作系统相互纠缠的缺点。人们可构建不可变的虚拟机映像，从而实现可预测的升级和回滚，但VM是重量级、不可移植的。

新方法是部署容器，容器基于操作系统级别的虚拟化而不是硬件虚拟化。这些容器彼此隔离并且与宿主机隔离：它们有自己的文件系统，看不到对方的进程，并且它们的计算资源使用可以被界定。它们比VM更容易构建，并且由于它们与底层基础架构和宿主机文件系统解耦了，可实现跨云、跨操作系统的移植。

由于容器小而快，因此可在每个容器镜像中包装一个应用程序。这种一对一的应用到镜像关系解锁了容器的全部优势。使用容器，可以在构建/发布期间（而非部署期间）创建不可变的容器镜像，因为每个应用程序无需与其余的应用程序栈组合，也无需与生产基础架构环境结合。在构建/发布期间生成容器镜像使得从开发到生产都能够保持一致的环境。同样，容器比虚拟机更加透明、便于监控和管理——特别是当容器进程的生命周期由基础架构管理而非容器内隐藏的进程监控程序管理时。最后，通过在每个容器中使用单个应用程序的方式，管理容器无异于管理应用程序的部署。

容器好处概要：

- 灵活的应用创建和部署：与VM映像相比，容器镜像的创建更加容易、有效率。
- 持续开发，集成和部署：通过快速轻松的回滚（由于镜像的不可变性）提供可靠且频繁的容器镜像构建和部署。
- **Dev和Ops分离问题**：在构建/发布期间而非部署期间创建镜像，从而将应用程序与基础架构分离。
- 开发、测试和生产环境一致：在笔记本电脑运行与云中一样。

- 云和操作系统可移植性：可运行在Ubuntu、RHEL、CoreOS、内部部署，Google Container Engine以及任何其他地方。
- 以应用为中心的管理：从在虚拟硬件上运行操作系统的抽象级别，提升到使用逻辑资源在操作系统上运行应用程序的级别。
- 松耦合，分布式，弹性，解放的微服务：应用程序分为更小、独立的部件，可动态部署和管理——而不是一个运行在一个大型机上的单体。
- 资源隔离：可预测的应用程序性能。
- 资源利用：效率高，密度高。

## 为什么我需要Kubernetes，它能干啥？

最基本的功能：Kubernetes可在物理机或虚拟机集群上调度和运行应用容器。然而，Kubernetes还允许开发人员将物理机以及虚拟机“从主机为中心的基础设施转移到以容器为中心的基础设施”，从而提供容器固有的全部优势。Kubernetes提供了构建以容器为中心的开发环境的基础架构。

Kubernetes满足了在生产中运行的应用程序的一些常见需求，例如：

- [Co-locating helper processes](#)，促进组合应用程序和保留“一个应用程序的每个容器”模型
- [Mounting storage systems](#)
- [Distributing secrets](#)
- [Checking application health](#)
- [Replicating application instances](#)
- [Using Horizontal Pod Autoscaling](#)
- [Naming and discovering](#)
- [Balancing loads](#)
- [Rolling updates](#)
- [Monitoring resources](#)
- [Accessing and ingesting logs](#)
- [Debugging applications](#)
- [Providing authentication and authorization](#)

这提供了PaaS的简单性，并具有IaaS的灵活性，并促进了跨基础架构提供商的可移植性。

## Kubernetes是一个怎样的平台？

尽管Kubernetes提供了大量功能，但总有新的场景从新功能中受益。应用程序特定的工作流程可被简化，从而加快开发人员的速度。可接受的特别编排最初常常需要大规模的自动化。这就是为什么Kubernetes也被设计为提供构建组件和工具的生态系统，使其更容易部署，扩展和管理应用程序。

[Label](#) 允许用户随心所欲地组织他们的资源。[Annotation](#) 允许用户使用自定义信息来装饰资源以方便他们的工作流程，并为管理工具提供检查点状态的简单方法。

此外，[Kubernetes control plane](#) 所用的API与开发人员和用户可用的API相同。用户可以使用 [their own API](#) 编写自己的控制器，例如 [scheduler](#)，这些API可由通用 [command-line tool](#) 定位。

这种 [design](#) 使得许多其他系统可以构建在Kubernetes上。

## Kubernetes不是什么？

Kubernetes不是一个传统的，全面的PaaS系统。 它保留了用户的重要选择。

Kubernetes：

- 不限制支持的应用类型。不规定应用框架（例如 [Wildfly](#) ），不限制支持的语言运行时（例如Java，Python，Ruby），不局限于 [12-factor applications](#) ，也不区分应用程序和服务 。 Kubernetes旨在支持各种各样的工作负载，包括无状态、有状态以及数据处理工作负载。 如果应用程序可在容器中运行，那么它应该能够很好地在Kubernetes上运行。
- 不提供中间件（例如消息总线）、数据处理框架（例如Spark）、数据库（例如MySQL），也不提供分布式存储系统（例如Ceph）作为内置服务。 这些应用可在Kubernetes上运行。
- 没有点击部署的服务市场。
- 不部署源代码，并且不构建应用。持续集成（CI）工作流是一个不同用户/项目有不同需求/偏好的领域，因此它支持在Kubernetes上运行CI工作流，而不强制工作流如何工作。
- 允许用户选择其日志记录、监视和警报系统。（它提供了一些集成。）
- 不提供/授权一个全面的应用配置语言/系统（例如 [jsonnet](#) ）。
- 不提供/不采用任何综合的机器配置、维护、管理或自愈系统。

另一方面，一些PaaS系统可运行在 Kubernetes上，例如 [OpenShift](#) 、 [Deis](#) 、 [Eldarion](#) 等。 您也可实现自己的定制PaaS，与您选择的CI系统集成，或者仅使用Kubernetes部署容器。

由于Kubernetes在应用层面而非硬件层面上运行，因此它提供了PaaS产品通用的功能，例如部署，扩展，负载均衡，日志和监控。然而，Kubernetes并不是一个单体，这些默认解决方案是可选、可插拔的。

另外Kubernetes不仅仅是一个编制系统 。实际上，它消除了编制的需要。编制的技术定义，就是执行定义的工作流：首先执行A，然后B，然后执行C。相反，**Kubernetes**由一组独立、可组合的控制进程组成，这些控制进程可将当前状态持续地驱动到所需的状态。 如何从A到C不要紧，集中控制也不需要；这种做法更类似于编排 。 这使系统更易用、更强大，更具弹性和可扩展性。

译者按：编排和编制：<https://wenku.baidu.com/view/ad063ef2f61fb7360b4c65cd.html>

## Kubernetes的含义是什么？K8S呢？

**Kubernetes**源自希腊语，意思是舵手或飞行员，是 *governor*（掌舵人）和 *cybernetic*（控制论）的根源。  
*K8s*是将8个字母“ubernete”替换为“8”的缩写。

译者按：控制论简介（讲解了什么是governor&cybernetic）：<https://wenku.baidu.com/view/1d97762c0066f5335a812157.html>

## 原文

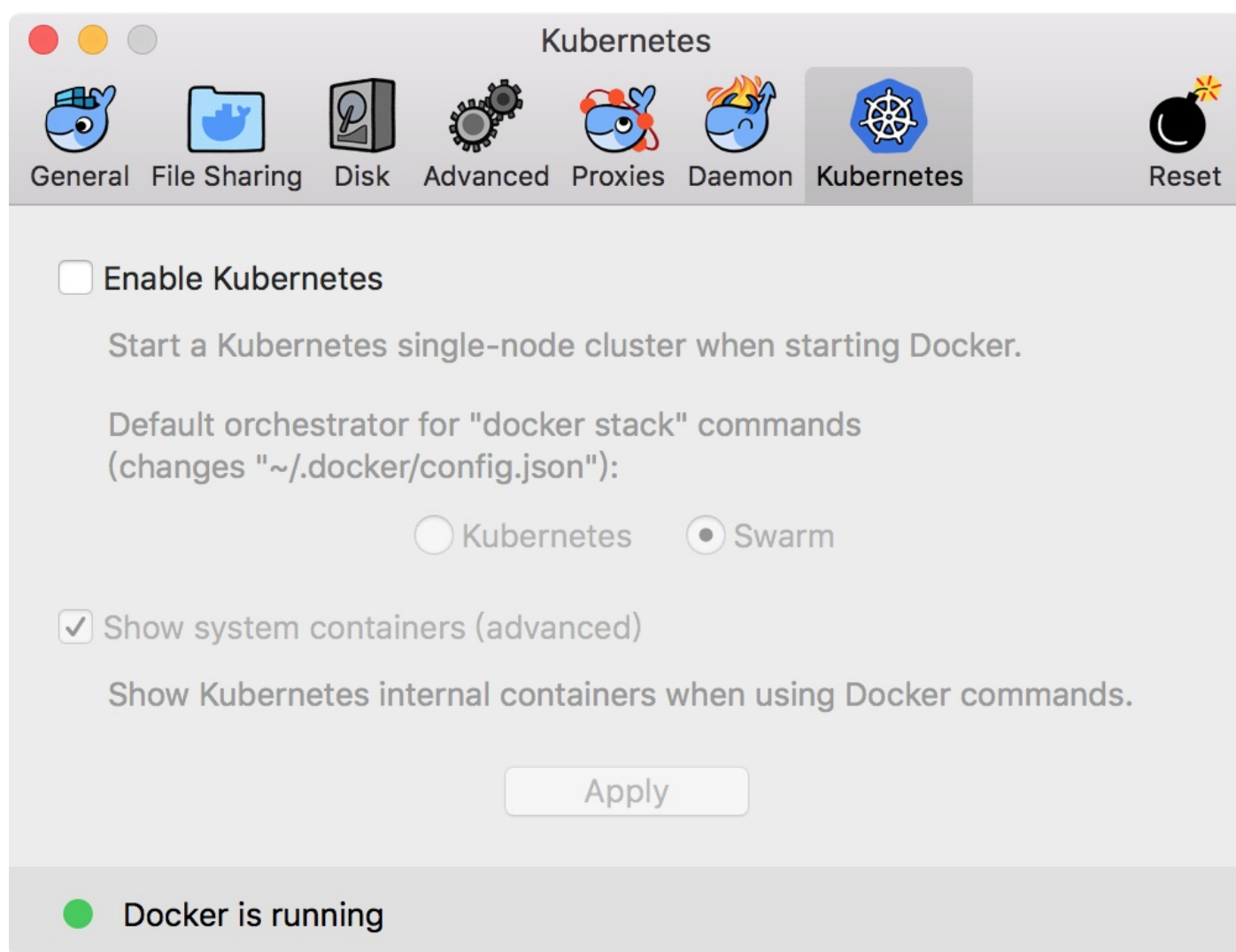
<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

# 安装Kubernetes（单机）

## 对于Mac/Windows 10

- 前提：保持网络畅通
- 系统版本满足要求

对于macOS或者Windows 10，Docker已经原生支持了Kubernetes。你所要做的只是启用Kubernetes即可，如下图：



## Minikube

一些场景下，安装Minikube是个不错的选择。该方式适用于Windows 10、Linux、macOS


- 官方安装说明文档：<https://github.com/kubernetes/minikube>
- 如何在Windows 10上运行Docker和Kubernetes？：<http://dockone.io/article/8136>

## 启用Kubernetes Dashboard

执行：

```
1. kubectl proxy
```

访问：

/proxy/#!/overview?namespace=default">http://localhost:8001/api/v1/namespaces/kube-  
system/services/https  /proxy/#!/overview?namespace=default

参考：

<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

# 使用Kubespray部署生产可用的Kubernetes集群（1.11.2）

前提：科学上网，或自行将gcr.io的镜像转成其他镜像仓库的镜像。

Kubernetes的安装部署是难中之难，每个版本安装方式都略有区别。笔者一直想找一种 支持多平台、相对简单、适用于生产环境 的部署方案。经过一段时间的调研，有如下几种解决方案进入笔者视野：

部署方案	优点	缺点
Kubeadm	官方出品	部署较麻烦、不够透明
Kubespray	官方出品、部署较简单、懂Ansible就能上手	不够透明
RKE	部署较简单、需要花一些时间了解RKE的cluster.yml配置文件	不够透明
手动部署 <a href="#">第三方操作文档</a>	完全透明、可配置、便于理解K8s各组件之间的关系	部署非常麻烦，容易出错

其他诸如Kops之类的方案，由于无法跨平台，或者其他因素，被我pass了。

最终，笔者决定使用Kubespray部署Kubernetes集群。也希望大家能够一起讨论，总结出更加好的部署方案。

废话不多说，以下是操作步骤。

注：撰写本文时，笔者临时租赁了几台海外阿里云机器，实现了科学上网。如果您的机器在国内，请：

- 考虑科学上网
- 或修改Kubespray中的gcr地址，改为其他仓库地址，例如阿里云镜像地址。

## 主机规划

IP	作用
172.20.0.87	ansible-client
172.20.0.88	master,node
172.20.0.89	master,node
172.20.0.90	node
172.20.0.91	node
172.20.0.92	node

## 准备工作

## 关闭selinux

所有机器都必须关闭selinux，执行如下命令即可。

```
1. ~]# setenforce 0
2. ~]# sed -i --follow-symlinks 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux
```

## 网络配置

### 在master机器上

```
1. ~]# firewall-cmd --permanent --add-port=6443/tcp
2. ~]# firewall-cmd --permanent --add-port=2379-2380/tcp
3. ~]# firewall-cmd --permanent --add-port=10250/tcp
4. ~]# firewall-cmd --permanent --add-port=10251/tcp
5. ~]# firewall-cmd --permanent --add-port=10252/tcp
6. ~]# firewall-cmd --permanent --add-port=10255/tcp
7. ~]# firewall-cmd --reload
8. ~]# modprobe br_netfilter
9. ~]# echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
10. ~]# sysctl -w net.ipv4.ip_forward=1
```

如果关闭了防火墙，则只需执行最下面三行。

### 在node机器上

```
1. ~]# firewall-cmd --permanent --add-port=10250/tcp
2. ~]# firewall-cmd --permanent --add-port=10255/tcp
3. ~]# firewall-cmd --permanent --add-port=30000-32767/tcp
4. ~]# firewall-cmd --permanent --add-port=6783/tcp
5. ~]# firewall-cmd --reload
6. ~]# echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
7. ~]# sysctl -w net.ipv4.ip_forward=1
```

如果关闭了防火墙，则只需执行最下面两行。

### 【可选】关闭防火墙

```
1. systemctl stop firewalld
```

## 在ansible-client机器上安装ansible

### 安装ansible

```
1. ~]# sudo yum install epel-release
2. ~]# sudo yum install ansible
```

## 安装jinja2

```
1. ~]# easy_install pip
2. ~]# pip2 install jinja2 --upgrade
```

如果执行 `pip2 install jinja2 --upgrade` 出现类似如下的提示：

```
1. You are using pip version 9.0.1, however version 18.0 is available.
2. You should consider upgrading via the 'pip install --upgrade pip' command.
```

则执行 `pip install --upgrade pip` 升级pip，再执行 `pip2 install jinja2 --upgrade`

## 安装Python 3.6

```
1. ~]# sudo yum install python36 -y
```

## 在ansible-client机器上配置免密登录

### 生成ssh公钥和私钥

在ansible-client机器上执行：

```
1. ~]# ssh-keygen
```

然后三次回车，生成ssh公钥和私钥。

### 建立ssh单向通道

在ansible-client机器上执行：

```
1. ~]# ssh-copy-id root@172.20.0.88      #将公钥分发给88机器
2. ~]# ssh-copy-id root@172.20.0.89
3. ~]# ssh-copy-id root@172.20.0.90
4. ~]# ssh-copy-id root@172.20.0.91
5. ~]# ssh-copy-id root@172.20.0.92
```

## 在ansible-client机器上安装kubespray

- 下载kubespray

**TIPS:** 本文下载的是master分支，如果大家要部署到线上环境，建议下载RELEASE分支。笔者撰写本文时，最新的RELEASE是2.6.0，RELEASE版本下载地址：<https://github.com/kubernetes-incubator/kubespray/releases> )



```
1. ~]# git clone https://github.com/kubernetes-incubator/kubespray.git
```

- 安装kubespray需要的包：

```
1. ~]# cd kubespray
2. ~]# sudo pip install -r requirements.txt
```

- 拷贝 `inventory/sample` ，命名为 `inventory/mycluster` ，mycluster可以改为其他你喜欢的名字

```
1. cp -r inventory/sample inventory/mycluster
```

- 使用inventory\_builder，初始化inventory文件

```
1. ~]# declare -a IPS=(172.20.0.88 172.20.0.89 172.20.0.90 172.20.0.91 172.20.0.92)
2. ~]# CONFIG_FILE=inventory/mycluster/hosts.ini python3 contrib/inventory_builder/inventory.py ${IPS[@]}
```

此时，会看到 `inventory/mycluster/host.ini` 文件内容类似如下：

```
1. [k8s-cluster:children]
2. kube-master
3. kube-node
4.
5. [all]
6. node1    ansible_host=172.20.0.88 ip=172.20.0.88
7. node2    ansible_host=172.20.0.89 ip=172.20.0.89
8. node3    ansible_host=172.20.0.90 ip=172.20.0.90
9. node4    ansible_host=172.20.0.91 ip=172.20.0.91
10. node5    ansible_host=172.20.0.92 ip=172.20.0.92
11.
12. [kube-master]
13. node1
14. node2
15.
16. [kube-node]
17. node1
18. node2
19. node3
20. node4
21. node5
22.
23. [etcd]
24. node1
25. node2
26. node3
27.
28. [calico-rr]
29.
30. [vault]
31. node1
```

```
32. node2
33. node3
```

- 使用ansible playbook部署kubespray

```
1. ~]# ansible-playbook -i inventory/mycluster/hosts.ini cluster.yml
```

- 大概20分钟左右，Kubernetes即可安装完毕。

## 验证

### 验证1：查看Node状态

```
1. ]# kubectl get nodes
2. NAME          STATUS    ROLES          AGE      VERSION
3. node1         Ready    master,node    2m       v1.11.2
4. node2         Ready    master,node    2m       v1.11.2
5. node3         Ready    node           2m       v1.11.2
6. node4         Ready    node           2m       v1.11.2
7. node5         Ready    node           2m       v1.11.2
```

每个node都是ready的，说明OK。

### 验证2：部署一个NGINX

```
1. # 启动一个单节点nginx
2. ]# kubectl run nginx --image=nginx:1.7.9 --port=80
3.
4. # 为“nginx”服务暴露端口
5. ]# kubectl expose deployment nginx --type=NodePort
6.
7. # 查看nginx服务详情
8. ]# kubectl get svc nginx
9. NAME          TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
10. nginx         NodePort      10.233.29.96    <none>           80:32345/TCP     14s
11.
12. # 访问测试，如果能够正常返回NGINX首页，说明正常
13. ]# curl localhost:32345
```

## 卸载

```
1. ]# ansible-playbook -i inventory/mycluster/hosts.ini reset.yml
```

## 遇到的问题

Calico网络插件部署失效。这是Calico 3.2所带来的问题，原因详见：<https://github.com/kubernetes-incubator/kubespray/issues/3223>

解决方

法：<https://github.com/wilmardo/kubespray/commit/1c87a49d1443bcdd237500a714f1a60d680c1ad8>，即：将Calico降级到3.1.3。

## 参考文档：

- Kubespray – 10 Simple Steps for Installing a Production-Ready, Multi-Master HA Kubernetes Cluster: <https://dzone.com/articles/kubespray-10-simple-steps-for-installing-a-product>

**TIPS:** 主要的参考文档，里面还讲解了Kubespray的一些配置，与可能会遇到的问题及解决方案。

- 使用Kubespray 部署kubernetes 高可用集群: <https://yq.aliyun.com/articles/505382>
- kubespray(ansible)自动化安装k8s集群: <https://www.cnblogs.com/iiiiher/p/8128184.html>

**TIPS:** 里面有将如何替换gcr镜像为国内镜像

- Installing Kubernetes On-premises/Cloud Providers with Kubespray:<https://kubernetes.io/docs/setup/custom-cloud/kubespray/>

# K8s组件

本文概述了Kubernetes集群中所需的各种组件。

## Master组件

Master组件提供K8s集群的控制面板。Master对集群进行全局决策（例如调度），以及检测和响应集群事件（例如：当replication controller所设置的 `replicas` 不够时，启动一个新的Pod）。

Master可在集群中的任意节点上运行。然而，简单起见，设置脚本通常在同一个VM上启动所有Master组件，并且不会在该VM上运行用户的容器。请阅读 [Building High-Availability Clusters](#) 以实现多主机VM配置。

## kube-apiserver

`kube-apiserver` 暴露Kubernetes的API。它是Kubernetes控制能力的前端。它被设计为可水平扩展——也就是通过部署更多实例来实现扩容。详见 [Building High-Availability Clusters](#) 。

## etcd

`etcd` 用作Kubernetes的后端存储。集群的所有数据都存储在此。请为你Kubernetes集群的etcd数据提供备份计划。

## kube-controller-manager

`kube-controller-manager` 运行Controller，它们是处理集群中常规任务的后台线程。逻辑上来讲，每个Controller都是一个单独的进程，但为了降低复杂性，它们都被编译成独立的二进制文件并运行在一个进程中。

这些控制器包括：

- Node Controller：当节点挂掉时，负责响应。
- Replication Controller：负责维护系统中每个replication controller对象具有正确数量的Pod。
- Endpoints Controller：填充Endpoint对象（即：连接Service&Pod）。
- Service Account & Token Controllers：为新的namespace创建默认帐户和API access tokens。

## cloud-controller-manager

cloud-controller-manager运行着与底层云提供商交互的Controller。cloud-controller-manager是在Kubernetes 1.6版中引入的，处于Alpha阶段。

cloud-controller-manager仅运行云提供商特定的Controller循环。您必须在kube-controller-manager中禁用这些Controller循环。可在启动kube-controller-manager时将 `--cloud-provider` 标志设为 `external` 来禁用控制器循环。

cloud-controller-manager允许云供应商代码和Kubernetes内核独立发展。在以前的版本中，核心的Kubernetes代码依赖于特定云提供商的功能代码。在未来的版本中，云供应商的特定代码应由云供应商自己维护，

并在运行Kubernetes时与cloud-controller-manager相关联。

以下控制器存在云提供商依赖：

- Node Controller：用于检查云提供商，从而确定Node在停止响应后从云中删除
- Route Controller：用于在底层云基础设施中设置路由
- Service Controller：用于创建、更新以及删除云提供商负载均衡器
- Volume Controller：用于创建、连接和装载Volume，并与云提供商进行交互，从而协调Volume

## kube-scheduler

`kube-scheduler` 监视新创建的、还没分配Node的Pod，并选择一个Node供这些Pod运行。

## addons（插件）

Addon是实现集群功能的Pod和Service。Pod可由Deployment、ReplicationController等进行管理。Namespace的插件对象则是在 `kube-system` 这个namespace中被创建的。

Addon manager创建并维护addon的资源。详见这里：[here](#)。

## DNS

虽然其他Addon不是严格要求的，但所有Kubernetes集群都应该有 `cluster DNS`，许多用例都依赖于它。

Cluster DNS是一个DNS服务器，它为Kubernetes服务提供DNS记录。

Kubernetes启动的容器会自动将该DNS服务器包含在DNS搜索中。

## Web UI（Dashboard）

`Dashboard` 是一个Kubernetes集群通用、基于Web的UI。它允许用户管理/排错集群中应用程序以及集群本身。

## Container Resource Monitoring（容器资源监控）

`Container Resource Monitoring` 将容器的通用时序指标记录到一个中心化的数据库中，并提供一个UI以便于浏览该数据。

## Cluster-level Logging（集群级别的日志）

`Cluster-level logging` 机制负责将容器的日志存储到具有搜索/浏览界面的中央日志存储中去。

## Node组件

Node组件在每个Node上运行，维护运行的Pod并提供Kubernetes运行时环境。

## kubelet

`kubelet` 是主要的Node代理。它监视已分配到你Node上的Pod（通过apiserver或本地配置文件）和：

- 装载Pod所需的Volume。
- 下载Pod的secret。
- 通过Docker（或实验时使用rkt）运行Pod的容器。
- 定期执行任何被请求容器的活动探针（liveness probes）。
- 在必要时创建*mirror pod*，从而将pod的状态报告回系统的其余部分。
- 将节点的状态报告回系统的其余部分。

## kube-proxy

`kube-proxy` 在主机上维护网络规则并执行连接转发，从而来实现Kubernetes服务抽象。

## docker

`docker` 用于运行容器。

## rkt

`rkt` 是一个Docker的替代品，支持在实验中运行容器

## supervisord

`supervisord` 是一个轻量级的进程监控/控制系统，可用于保持kubelet和docker运行。

## fluentd

`fluentd` 是一个守护进程，利用它可实现 `cluster-level logging`。

# Kubernetes API

[API conventions doc](#) 中描述了API的总体规范。

[API Reference](#) 中描述了API端点、资源类型和样本。

[access doc](#) 讨论了API的远程访问。

Kubernetes API也是系统声明式配置模式的基础。[Kubectl](#) 命令行工具可用于创建、更新、删除以及获取API对象。

Kubernetes也会存储其API资源方面的序列化状态（目前存在 [etcd](#) 中）。

Kubernetes本身被分解成了多个组件，通过其API进行交互。

## API更改

根据我们的经验，任何成功的系统都需要随着新用例的出现或现有的变化而发展和变化。因此，我们预计Kubernetes API将会不断变化和发展。但是，在很长一段时间内并不会破坏与现有客户端的兼容性。一般来说，新的API资源和新的资源字段通常可被频繁添加。消除资源或字段将需遵循 [API deprecation policy](#) 。

[API change document](#) 详细介绍了兼容更改以及如何更改API的内容。

## OpenAPI与Swagger定义

完整的API详情使用 [Swagger v1.2](#) 和 [OpenAPI](#) 记录。Kubernetes apiserver（又名“master”）公开了一个路径是 `/swaggerapi` API，该API使用Swagger v1.2 Kubernetes API规范。您也可以通过将 `--enable-swagger-ui=true` 标志传递给apiserver，从而浏览 `/swagger-ui` 的启用UI的API文档。

从Kubernetes 1.4开始，OpenAPI规范也可在 `/swagger.json` 。当我们从Swagger v1.2转换到OpenAPI（又名Swagger v2.0）时，一些工具（例如：`kubectl`和`swagger-ui`）仍在使用的v1.2规范。OpenAPI规范在Kubernetes 1.5中，进入Beta阶段。

Kubernetes为主要用于集群内通信的API实现了另一种基于Protobuf的序列化格式，在 [design proposal](#) 有记录，每个schema的IDL文件都存放在定义该API对象的Go语言包中。

## API版本

为了更容易地消除字段或重组资源表示，Kubernetes支持多种API版本，每种API版本都有不同的API路径，例如 `/api/v1` 或 `/apis/extensions/v1beta1` 。

我们选择在API级别，而非资源级别/字段级别使用版本控制，从而确保API提供清晰、一致的系統资源和行为视图，以及控制对终极API/实验API的访问。JSON和Protobuf序列化schema遵循相同的schema更改准则——以下所有描述都涵盖了两种格式。

请注意，API版本控制和软件版本控制仅仅是间接相关的关系。[API and release versioning proposal](#) (API

和[版本发布提案](#) ) 1 描述了API版本控制和软件版本控制之间的关系。

不同的API版本意味着不同程度的稳定性和支持。[API Changes documentation](#) 详细描述了每个级别的标准。概括如下：

- Alpha级别：
  - 版本名称包含 `alpha` （例如 `v1alpha1` ）
  - 可能有一些bug，启用该功能可能会显示错误。 默认禁用
  - 一些功能可能随时会被废弃，恕不另行通知
  - API可能会以不兼容的方式更改，恕不另行通知
  - 建议仅在短期测试集群中使用，因为增加了bug带来的风险，而且缺乏长期支持
- Beta级别：
  - 版本名称包含 `beta` （例如 `v2beta3` ）
  - 代码经过了良好的测试。启用该功能被认为是安全的。 默认启用
  - 整体功能不会被删除，尽管细节可能会改变
  - 对象的schema/语义可能会在后续的beta版/稳定版本中以不兼容的方式发生变化。发生这种情况时，我们将提供迁移到下一个版本的说明。 这可能需要删除、编辑和重新创建API对象。编辑进程可能需要一些思考。依赖该功能的应用程序可能需要停机。
  - 推荐仅用于非关键业务用途，因为后续版本中可能会发生不兼容的更改。如果您有多个可独立升级的集群，则可放宽此限制。
  - 请尝试我们的**beta**功能并给他们反馈！一旦他们退出**beta**状态，我们就不会做更多的改变。
- Stable等级：
  - 版本名称为 `vx` ，其中 `x` 为整数。
  - 稳定版本的功能将会出现在许多后续版本中。

## API组

为了使Kubernetes API扩展更容易，我们实现了 [API groups](#) 。 API组可在序列化对象的 `apiVersion` 字段中使用一个REST路径指定。

目前可使用的几个API组：

1. “core”（由于没有明确的组名称，通常称为“legacy”）组，它的REST路径是 `/api/v1` 。例如 `apiVersion: v1` 。
2. 命名组是REST路径 `/apis/$GROUP_NAME/$VERSION` ，并使用 `apiVersion: $GROUP_NAME/$VERSION` （例如 `apiVersion: batch/v1` ）。支持的API组的完整列表可详见：[Kubernetes API reference](#) 。

使用 [custom resources](#) 扩展API有两个支持的路径：

1. [CustomResourceDefinition](#) 适用于非常基本的CRUD需求的用户。
2. 即将推出：用户需要完整的Kubernetes API语法，这些语法可实现自己的apiserver，并使用 [aggregator](#) 无缝连接客户端。

## 启用API组

默认情况下，某些资源和API组已被启用。可通过在apiserver上设置 `--runtime-config` 来启用或禁用它们。 `--`



`runtime-config` 接受逗号分隔的值。例如：要禁用 `batch / v1`，请设置 `--runtime-config=batch/v1=false`；想启用 `batch/v2alpha1`，可设置 `--runtime-config=batch/v2alpha1`。该标志接受逗号分隔的一组键值对，键值对描述了apiserver的运行时配置。

重要信息：启用或禁用组或资源，需要重启apiserver和controller-manager来获取 `--runtime-config` 的更改。

## 启用组中的资源

默认情况下，DaemonSets、Deployments、HorizontalPodAutoscalers、Ingress、Jobs和ReplicaSets都被启用。可通过在apiserver上设置 `--runtime-config` 来启用其他扩展资源。`--runtime-config` 接受逗号分隔值。例如：要禁用Deployments和Ingress，可设置

```
1. --runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/ingress=false
```

## 原文

<https://kubernetes.io/docs/concepts/overview/kubernetes-api/>

# 理解K8s对象

这个页面描述了Kubernetes对象在Kubernetes API中的表现，以及如何用 `.yaml` 格式表达它们。

## 理解Kubernetes对象

Kubernetes对象是Kubernetes系统中的持久实体。Kubernetes使用这些实体来表示集群的状态。具体来说，它们可描述：

- 容器化的应用程序正在运行什么（以及在哪些Node上运行）
- 这些应用可用的资源
- 这些应用的策略，例如重启策略，升级和容错

Kubernetes对象是一种“意图记录”——一旦您创建对象，Kubernetes系统将持续工作以确保对象存在。通过创建一个对象，您可以有效地告诉Kubernetes系统您希望集群的工作负载是怎样的？这是您集群的期望状态。

要使用Kubernetes对象——无论是创建、修改还是删除它们，您都需要使用 [Kubernetes API](#)。例如，当您使用 `kubectl` 命令行时，CLI会为您提供必要的Kubernetes API调用；您也可直接在自己的程序中使用Kubernetes API。Kubernetes目前提供了一个 `golang client library`，并且正在开发其他语言的客户端库（如 `Python`）。

## 对象Spec和Status（规格和状态）

每个Kubernetes对象都包含两个嵌套的对象字段，它们控制着对象的配置：对象`spec`和对象`status`。您必须提供`spec`，它描述了对象所期望的状态——您希望对象所具有的特性。`status`描述对象的实际状态，由Kubernetes系统提供和更新。在任何时候，Kubernetes Control Plane都会主动管理对象的实际状态，从而让其匹配您所期望的状态。

例如，Kubernetes Deployment是一个表示在集群上运行的应用程序的对象。在创建Deployment时，可设置Deployment spec，例如指定有三个应用程序的replicas（副本）正在运行。这样，Kubernetes系统就会读取Deployment spec，并启动您想要的、应用程序的三个实例——根据您的spec更新status。如果任何一个实例失败（status发生改变），Kubernetes系统就会响应spec和status之间的差异，并进行更正——在这本例中，会启动一个替换实例。

有关对象spec，status和metadata的更多信息，详见：[Kubernetes API Conventions](#)。

## 描述Kubernetes对象

在Kubernetes中创建对象时，必须提供对象的spec，spec描述对象所期望的状态，以及一些关于对象的基本信息（如名称）。当您使用Kubernetes API创建对象（直接或通过 `kubectl`）时，该API请求必须在请求体中包含该信息（以JSON格式）。最常见的，可在一个`.yaml`文件中向`kubectl`提供信息。在进行API请求时，`kubectl`会将信息转换为JSON。

如下是一个示例 `.yaml` 文件，显示Kubernetes Deployment所需的字段和对象spec：

```
1. apiVersion: apps/v1beta1
2. kind: Deployment
3. metadata:
4.   name: nginx-deployment
5. spec:
6.   replicas: 3
7.   template:
8.     metadata:
9.       labels:
10.        app: nginx
11.     spec:
12.       containers:
13.       - name: nginx
14.         image: nginx:1.7.9
15.         ports:
16.         - containerPort: 80
```

使用该 `.yaml` 文件创建Deployment的一种方法是在 `kubectl` 命令行界面中使用 `kubectl create` 命令，将 `.yaml` 文件作为参数传递。 例如：

```
1. $ kubectl create -f docs/user-guide/nginx-deployment.yaml --record
```

将会输出类似如下的内容：

```
1. deployment "nginx-deployment" created
```

## 必填字段

在Kubernetes对象的 `.yaml` 文件中，您需要为以下字段设置值：

- `apiVersion` —指定Kubernetes API的版本
- `kind` —你想创建什么类型的对象
- `metadata` —有助于唯一标识对象的数据，包括 `name` 字符串，UID和可选的 `namespace`

您还需要提供 `spec` 字段。对于不同的Kubernetes对象来说，`spec` 的格式都是不同的。 [Kubernetes API reference](#) 可帮助您找到所有对象的spec格式。

## 原文

<https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

# 名称

---

Kubernetes REST API中的所有对象都会被Name和UID明确标识。

对于用户提供非唯一属性，Kubernetes提供 [labels](#) 和 [annotations](#) 来标识。

## 名称（Name）

---

名称通常由客户提供。对于给定类型的对象，一次只有一个对象可以有一个给定的名称（即：它们在空间上是唯一的）。但是，如果您删除一个对象，则可使用相同的名称创建一个新对象。名称用于引用资源URL中的对象，例如 `/api/v1/pods/some-name`。按照惯例，Kubernetes资源的名称最多可达253个字符，由小写字母、数字、`-` 和 `.` 组成，但某些资源有更具体的限制。有关名称的精确语法规则，详见：[identifiers design doc](#)。

## UID

---

UID由Kubernetes生成。在Kubernetes集群的整个生命周期中创建的每个对象都有不同的UID（即：它们在空间和时间上是唯一的）。

# Namespace（命名空间）

Kubernetes支持在同一物理集群中创建多个虚拟集群。 这些虚拟集群被称为Namespace。

## 使用多个Namespace的场景

Namespace旨在用于这种环境：有许多的用户，这些用户分布在多个团队/项目中。对于只有几个或几十个用户的集群，您根本不需要创建或考虑使用Namespace。 当您需要使用Namespace提供的功能时，再考虑使用Namespace。

Namespace为Name（名称）提供了范围。在Namespace中，资源的名称必须唯一，但不能跨Namespace。

Namespace是一种在多种用途之间划分集群资源的方法（通过 `resource quota` ）。

在未来的Kubernetes版本中，同一Namespace中的对象默认有相同的访问控制策略。

没有必要使用多个Namespace来分隔稍微不同的资源，例如同一软件的不同版本，可使用 `labels` 来区分同一Namespace中的资源。

## 使用Namespace

Namespace的创建和删除在 [Admin Guide documentation for namespaces](#) 有描述。

## 查看Namespace

可使用如下命令列出集群中当前的Namespace：

```
1. $ kubectl get namespaces
2. NAME                STATUS    AGE
3. default             Active   1d
4. kube-system         Active   1d
```

Kubernetes初始有两个Namespace：

- `default` ：对于没有其他Namespace的对象的默认Namespace
- `kube-system` ：由Kubernetes系统所创建的对象的Namespace

## 为请求设置Namespace

要临时设置请求的Namespace，可使用 `--namespace` 标志。

例如：

```
1. $ kubectl --namespace=<insert-namespace-name-here> run nginx --image=nginx
2. $ kubectl --namespace=<insert-namespace-name-here> get pods
```

## 设置Namespace首选项

可在上下文中永久保存所有后续 `kubectl` 命令的Namespace。

```
1. $ kubectl config set-context $(kubectl config current-context) --namespace=<insert-namespace-name-here>
2. # Validate it
3. $ kubectl config view | grep namespace:
```

## Namespace和DNS

当你创建 `Service` 时，会创建一个相应的 `DNS entry`。此条目的形式为 `<service-name>.<namespace-name>.svc.cluster.local`，这意味着如果容器只使用 `<service-name>`，则它将解析为Namespace本地的服务。这对在多个Namespace（例如Development、Staging以及Production）中使用相同的配置的场景非常有用。如果要跨越Namespace，请使用完全限定域名（FQDN）。

译者按：FQDN是fully qualified domain name的缩写，即：完全限定域名

## 并非所有对象都在Namespace中

大多数Kubernetes资源（例如：Pod、Service、Replication Controllers等）都在某些Namespace中。但Namespace资源本身并不在Namespace中。低级资源（例如：`nodes` 和 `persistentVolumes`）也不在任何Namespace中。事件是一个例外：它们可能有也可能没有Namespace，具体取决于事件的对象。

# Label和Selector（Label和选择器）

Label是附加到对象（如Pod）的键值对。Label旨在用于指定对用户有意义的对象的识别属性，但不直接表示核心系统的语义。Label可用于组织和选择对象的子集。Label可在创建时附加到对象，也可在创建后随时添加和修改。每个对象都可定义一组Label。对于给定的对象，Key必须唯一。

```
1. "labels" : {
2.   "key1"  : "value1" ,
3.   "key2"  : "value2"
4. }
```

我们最终会对Label进行索引和反向索引，以便于高效的查询、watch、排序、分组等操作。不要使用非标识的、大型的结构化数据污染Label。对于非标识的信息应使用非标识，特别是大型和/或结构化数据来污染Label。非识别信息应使用 `annotation` 记录。

## 动机

Label使用户能够以松耦合的方式，将自己的组织结构映射到系统对象上，客户端无需存储这些映射。

服务部署和批处理流水线通常是多维实体（例如：多个分区或部署、多个发布轨道、多个层、每层有多个微服务）。管理往往需要跨部门才能进行，这打破了严格层级表现的封装，特别是由基础设施而非用户确定的刚性层次结构。

示例Label：

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

以上只是常用Label示例。可自由定制。请记住，给定对象的Label的key必须唯一。

## 语法和字符集

Label是键值对的形式。

有效Label的key分为两段：

- 名称段和前缀段，以 `/` 分隔。
- 名称段是必需的，不超过63个字符，以字母或数字（`[a-z0-9A-Z]`）开头和结尾，中间可包含 `-`、`_`、`.`、字母或数字等字符。
- 前缀段是可选的。必须是DNS子域：一系列由 `.` 分隔的DNS Label，总共不超过253个字符，后跟 `/`。
- 如省略前缀，则假定Label的Key对用户是私有的。
- 为最终用户对象添加Label的自动化系统组件（例如，`kube-scheduler`、`kube-controller-manager`、`kube-apiserver`、`kubectl` 或其他第三方自动化组件）必须指定前缀。`kubernetes.io/` 前缀保留给

Kubernetes核心组件。

有效的Label值必须：

- 不超过63个字符
- 为空，或者：以字母或数字（`[a-z0-9A-Z]`）开头和结尾，中间包含 `-`、`_`、`.`、字母或数字等字符。

## Label选择器

不同于 `names and UIDs`，Label不提供唯一性。一般来说，多个可对象携带相同的Label。

通过Label选择器，客户端/用户可识别一组对象。Label选择器是Kubernetes中的核心分组API。

API目前支持两种类型的选择器：*equality-based* 和 *set-based*。Label选择器可由逗号分隔的多个需求组成。在多重需求的情况下，必须满足所有需求，因此逗号作为AND逻辑运算符。

一个empty Label选择器（即一个零需求的选择器）选择集合中的每个对象。

null Label选择器（仅用于可选的选择器字段）不选择任何对象。

注意：两个Controller的Label选择器不能在命名空间内重叠，否则它们将会产生冲突。

## Equality-based requirement

*Equality- or inequality-based* requirement允许通过LabelKey和Value进行过滤。匹配对象必须满足所有的Label约束，尽管对象可能还有其他Label。允许使用三种运算符：`=`、`==`、`!=`。前两个表示相等（只是同义），而后者则表示不相等。例如：

```
1. environment = production
2. tier != frontend
```

前者选择所有与key = `environment` 并且value = `production` 的资源。后者选择key = `tier` 并且value 不等于 `frontend`，以及key不等于 `tier` 的所有资源。可使用 `,` 过滤是生产环境中（`production`）非前端（`frontend`）的资源：`environment=production,tier!=frontend`。

## Set-based requirement

*Set-based* label requirement允许根据一组Value过滤Key。支持三种运算符：`in`、`notin`和 `exists`（只有Key标识符）。例如：

```
1. environment in (production, qa)
2. tier notin (frontend, backend)
3. partition
4. !partition
```

第一个示例选择Key等于 `environment`，Value等于 `production` 或 `qa` 的所有资源。



第二个示例选择Key等于 `tier`，Value不等于 `frontend` 或 `backend`，以及所有Key不等于 `tier` 的所有资源。

第三个示例选择Key包含 `partition` 所有资源；不检查Value的值。

第四个示例选择Key不包含 `partition` 的所有资源，不检查任何值。

类似地，逗号可用作AND运算符。因此可用 `partition,environment notin (qa)` 过滤 Key = `partition`（无论值）并且 `environment != qa` 的资源。基于集合的Label选择器，也可表示基于等式的Label选择。例如，`environment=production` 等同于 `environment in (production)`；同样，`!=` 等同于 `notin`。

Set-based requirement可以与equality-based requirement混合使用。例如：`partition in (customerA, customerB),environment!=qa`。

## API

### LIST与WATCH过滤

LIST和WATCH操作可指定Label选择器，这样就可以使用查询参数过滤返回的对象集。两种requirement都是允许的（在这里表示，它们将显示在URL查询字符串中）：

- equality-based requirement: `?labelSelector=environment%3Dproduction,tier%3Dfrontend`
- set-based requirements: `?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29`

两种Label选择器都可用于在REST客户端中LIST或WATCH资源。例如，使用 `kubectl` 定位 `apiserver` 并使用 equality-based 的方式可写为：

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

或使用set-based requirements：

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

如上所示，set-based requirement表达能力更强。例如，他们可以对Value执行OR运算符：

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

或通过exists操作符，实现restricting negative matching：

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

### 在API对象中设置引用

一些Kubernetes对象（如 `services` 和 `replicationcontrollers`）也可使用Label选择器来指定其他资源（例如 `pods`）。

## Service 和 ReplicationController

使用Label选择器定义 `service` 指向的一组Pod。类似地， `replicationcontroller` 所管理的Pod总数也可用Label选择器定义。

两个对象的Label选择器都使用map在 `json` 或 `yaml` 文件中定义，只支持`equality-based requirement`选择器：

```
1. "selector": {
2.     "component" : "redis",
3. }
```

或：

```
1. selector:
2.     component: redis
```

此选择器（分别以 `json` 或 `yaml` 格式）等价于 `component=redis` 或 `component in (redis)` 。

## 支持set-based requirement的资源

较新的资源（例如 `Job` 、 `Deployment` 、 `Replica Set` 以及 `Daemon Set` ）也支持 `set-based requirement`。

```
1. selector:
2.     matchLabels:
3.         component: redis
4.     matchExpressions:
5.         - {key: tier, operator: In, values: [cache]}
6.         - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` 是 `{ key,value }` 的映射。`matchLabels` 映射中的单个 `{ key,value }` 等价于 `matchExpressions` 一个元素，其 `key` 为“key”，`operator` 为“In”，`values` 数组仅包含“value”。`matchExpressions` 是一个Pod选择器需求列表。有效运算符包括In、NotIn、Exists以及DoesNotExist。 当使用In和NotIn时，Value必须非空。所有来自 `matchLabels` 和 `matchExpressions` 的需求使用AND串联，所有需求都满足才能匹配。

## 选择Node列表

使用Label选择Node的一个用例是约束一个Pod能被调度到哪些Node上。有关详细信息，请参阅有关 [node selection](#) 的文档。

# Annotation（注解）

---

可使用Kubernetes annotation将任意的非识别元数据附加到对象。 客户端可取得此元数据。

## 将元数据附加到对象

---

您可以使用标签或Annotation将元数据附加到Kubernetes对象。标签可用于选择对象并查找满足某些条件的对象集合。相比之下，Annotation不用于标识和选择对象。Annotation中的元数据可大可小，结构化或非结构化，并且可包括标签所不允许的字符。

Annotation类似于标签，是键值对映射：

```
1. "annotations": {  
2.   "key1" : "value1",  
3.   "key2" : "value2"  
4. }
```

类似以下信息可记录到Annotation中：

- 由declarative configuration layer管理的字段。将这些字段附加为Annotation，可将它们与客户端或服务器设置的默认值、自动生成的字段或以及auto-sizing或auto-scaling的系统所设置的字段区分开。
- 构建信息、发布信息或镜像信息，如时间戳、release ID、git分支、PR编号、镜像哈希以及注册表地址。
- 指向日志、监控、分析或审计仓库。
- 用于调试的客户端库或工具的信息：例如名称、版本和构建信息。
- 用户或工具/系统来源信息，例如来自其他生态系统组件的相关对象的URL。
- 轻量级升级工具的元数据：例如配置或检查点。
- 负责人的电话或寻呼机号码，或指定能找到该信息的条目，例如团队网站。

如果不使用注释，则可将这种类型的信息存储在外部数据库或目录中，但这将会使生产共享客户端库/工具（用于部署、管理、内省）变得更加困难。

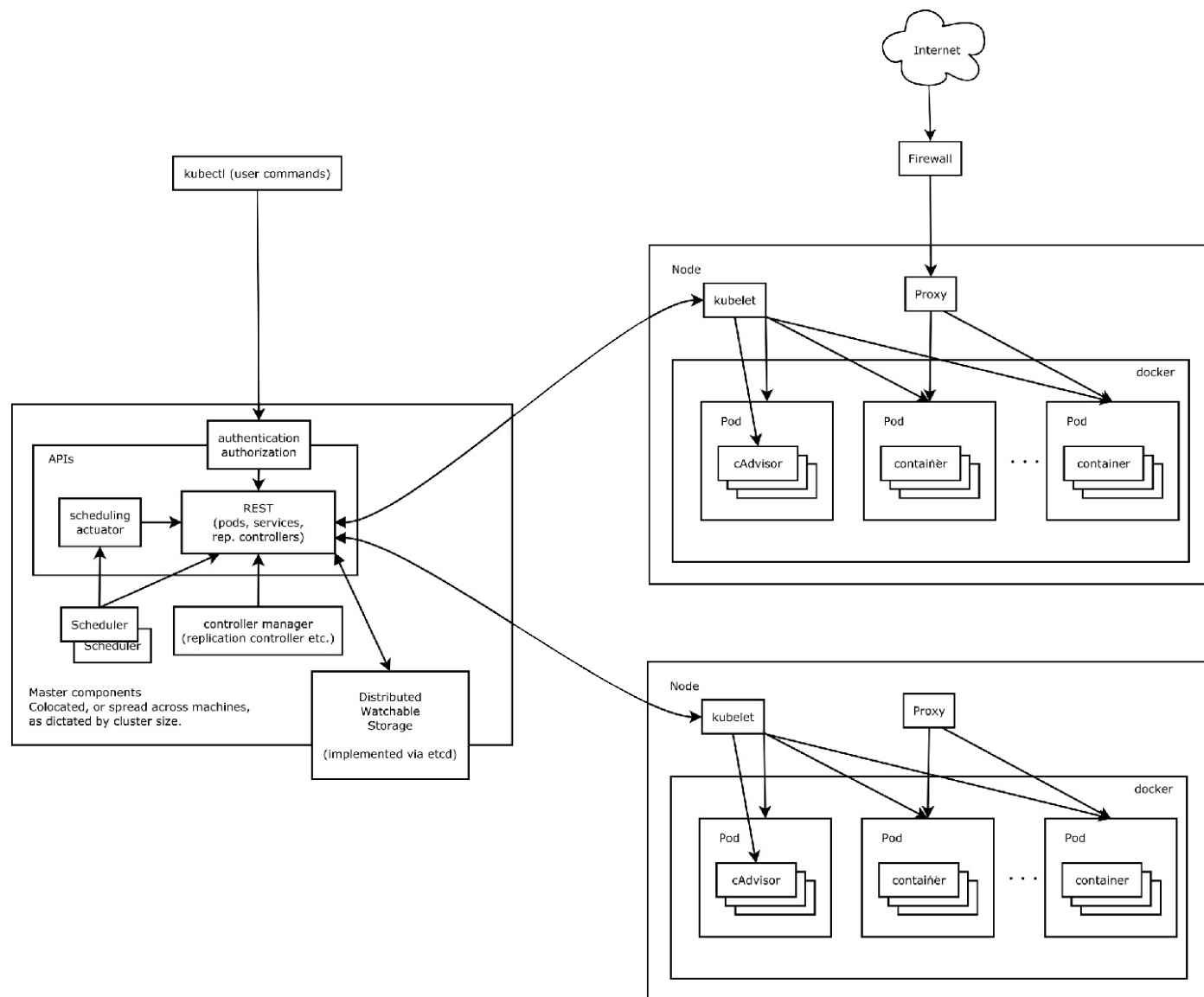
## 原文

---

<https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/>

# K8s架构及基本概念

## 架构图



## Master（主节点）

K8s里的Master指的是集群控制节点，一个K8s集群需要有一个Master节点来负责整个集群的管理和控制，一般来说，K8s的所有控制命令都是发送给Master，然后由Master来负责具体的执行过程。

Master通常会部署在一个独立的服务器或虚拟机上，它是整个集群的首脑，如果Master宕机或不可用，那么我们所有的控制命令都将会失效。

Master节点上运行着如下的关键进程：

- **API Server:** K8s里所有资源增删改查等操作的对外入口，也是集群控制的入口进程，它提供了HTTP RESTful API接口给客户端以及其他组件调用。

- **Controller Manager**: Controller Manager是K8s里所有对象的自动化控制中心。顾名思义，它负责管理“Controller”，主要有：
  - endpoint-controller: 刷新服务和pod 的关联信息
  - replication-controller: 维护某个 pod 的副本数为配置的数值
- **Scheduler**: 负责资源调度（Pod调度），将Pod分配到某个节点上。
- [可能有]etcd: 资源对象存储中心，K8s的所有资源对象数据都存储在此。

## Node（工作节点）

在K8s集群中，除Master以外的其他机器被称为Node节点。与Master一样，Node节点也可以是一台物理机或虚拟机。

由于Node节点才是K8s集群中的工作

- **kubelet**: 负责Pod所对应的容器的生命周期管理，例如容器的创建、启停等。根据从etcd中获取的信息来管理容器、上报Pod运行状态等。
- **kube-proxy**: 实现K8s Service的通信与负载均衡机制。
- **docker**: 你懂的

## Pod（容器组）

Pod是由若干容器组成的容器组，同一个Pod内的所有容器运行在同一主机上，这些容器使用相同的网络命名空间、IP地址和端口，相互之间能通过localhost来发现和通信。

不仅如此，同一个Pod内的所有容器还共享存储卷，这个存储卷也被称为Pod Volume。

在k8s中创建，调度和管理的最小单位就是**Pod**，而非容器，Pod通过提供更高层次的抽象，提供了更加灵活的部署和管理模式。

## Replication Controller（RC）

RC是用来管理Pod的工具，每个RC由一个或多个Pod组成。

- 在RC被创建之后，系统将会保持RC中可用的Pod个数与创建RC时定义的Pod个数一致。
  - 如果Pod个数小于定义的个数，RC会启动新的Pod
  - 反之则会杀死多余的Pod。

我们常会使用YAML定义一个RC，例如：

```
1. apiVersion: v1                                # API版本
2. kind: ReplicationController                    # 定义一个RC
3. metadata:
4.   name: mysql                                  # RC名称，全局唯一
5. spec:
6.   replicas: 1
7.   selector:
8.     app: mysql                                  # RC的POD标签选择器，即：监控和管理拥有这些标签的POD实例，确保当前集群中有且只有
replicas个POD实例在运行
```

```

9.   template:
10.     metadata:
11.       labels:                # 指定该POD的标签
12.         app: mysql           # POD副本拥有的标签，需要与RC的selector一致
13.     spec:
14.       containers:
15.         - name: mysql
16.           image: mysql
17.           ports:
18.             - containerPort: 3306
19.           env:
20.             - name: MYSQL_ROOT_PASSWORD
21.               value: "123456"

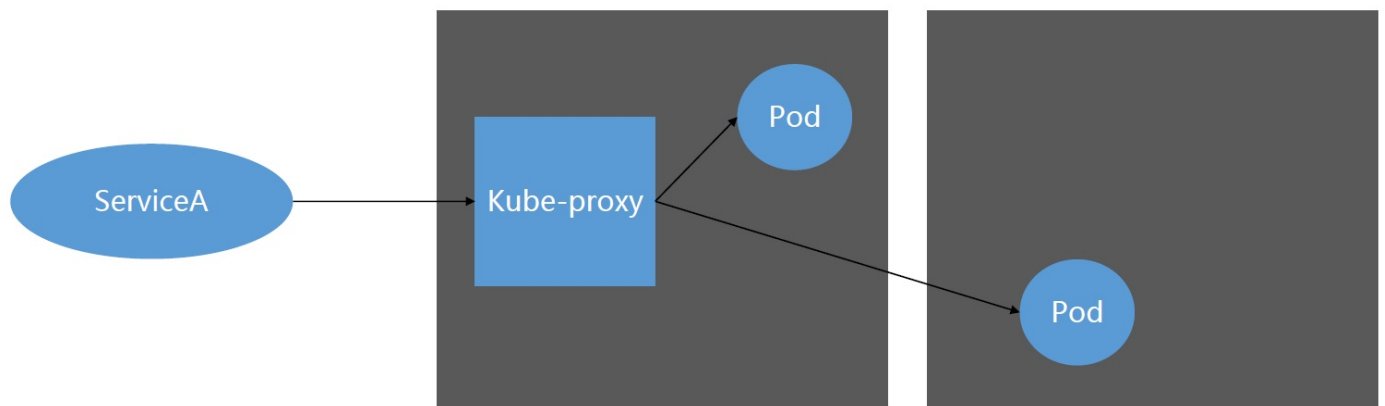
```

如上所示，一个RC的定义一般包含了如下三部分：

- Pod所期待的副本数
- 筛选Pod的标签选择器：RC通过Label来关联对应的Pod
- Pod模板：当集群中Pod的副本数量小于（大于）期望值时，K8s就会使用该模板去创建（删除）新的Pod。

## Service

Service可以简单理解成一组提供相同服务的Pod的对外访问入口。Service与Pod之间通过Selector来绑定。下图简单说明了Service与Pod之间的关系



提问：RC、Pod、Service三者之间的关系是怎样的？

## 参考文档

容器Docker与kubernetes: <http://www.cnblogs.com/stonehat/p/5148455.html>

Kubernetes扫盲: [http://blog.csdn.net/frank\\_zhu\\_bj/article/details/51824697](http://blog.csdn.net/frank_zhu_bj/article/details/51824697)

Kubernetes微服务架构应用实践: <http://www.chinacloud.cn/show.aspx?id=24091&cid=12>

# Master与Node的通信

## 概要

本文列出了Master（其实是apiserver）和Kubernetes集群之间的通信路径。目的是允许用户自定义其安装，从而加强网络配置，以便集群可在不受信任的网络（或云提供商上的公共IP）上运行。

## Cluster -> Master

从集群到Master的所有通信路径终止于apiserver（其他Master组件都不是设计来暴露远程服务的）。在典型的部署中，我们会为apiserver配置监听启用了一种或多形式的客户端 [authentication](#) 的安全HTTPS端口（443）。应启用一种或多 [authorization](#) 形式，特别是允许 [anonymous requests](#) 或 [service account tokens](#) 的情况下

应为Node配置集群的公共根证书，以便安全地连接到apiserver。例如，在默认的GCE部署中，提供给kubelet的客户端凭证采用客户端证书的形式。请参阅用于自动配置kubelet客户端证书的 [kubelet TLS bootstrapping](#)。

希望连接到apiserver的Pod可通过服务帐户安全地进行，这样，Kubernetes就会在实例化时，自动将公共根证书和有效的承载令牌注入到该Pod中。 `kubernetes` 服务（在所有namespace中）配置了一个虚拟IP地址（通过 kube-proxy）重定向到apiserver上的HTTPS端点。

Master组件通过不安全（未加密或验证）端口与集群apiserver通信。该端口通常仅在Master机器上的localhost接口上公开，以便所有运行在同一台机器上的Master组件可与集群的apiserver进行通信。随着时间的推移，Master组件将被迁移以使用安全端口进行认证和授权（参见 [#13598](#)）。

因此，默认情况下，来自集群（Node，以及Node上运行的Pod）到Master的连接默认操作模式将被保护，并且可在不可信和/公共网络上运行。

## Master -> Cluster

从Master（apiserver）到集群主要有两个通信路径。一是从apiserver到集群中的每个Node上运行的kubelet进程。二是通过apiserver的proxy功能，从apiserver到任意Node、Pod、或服务。

## apiserver -> kubelet

从apiserver到kubelet的连接用于：

- 获取Pod的日志。
- 通过kubectl连接到运行的Pod。
- 提供kubelet的端口转发功能。

这些连接终止于kubelet的HTTPS端点。默认情况下，apiserver不验证kubelet的证书，这使得连接可能会受到中间人的攻击，并且在不可信/公共网络上运行是不安全的。

要验证此连接，请使用 `--kubelet-certificate-authority` 标志，为apiserver提供一个根证书包，用于验证kubelet的证书。

如果不能这样做，如果需要，请在apiserver和kubelet之间使用 [SSH tunneling](#)，以避免通过不可信或公共网络进行连接。

最后，应启用 [Kubelet authentication and/or authorization](#) 来保护kubelet API。

## apiserver -> nodes, pods, and services

从apiserver到Node、Pod或服务连接默认为纯HTTP连接，因此不会被认证或加密。可通过将 `https:` 前缀添加到API URL中的Node、Pod、Service名称，从而运行安全的HTTPS连接，但不会验证HTTPS端点提供的证书，也不会提供客户端凭据——因此，尽管连接将被加密，它不会提供任何诚信保证。这些连接在不受信任/公共网络上运行并不安全。

## SSH隧道

[Google Container Engine](#) 使用SSH隧道来保护Master -> 集群的通信路径。在此配置中，apiserver会启动一个SSH隧道到集群中的每个Node（连接到监听端口22的ssh服务器），并通过隧道传递目标为kubelet，Node，Pod或服务的所有流量。该隧道确保流量不会暴露到私有GCE网络以外。

## 原文

---

<https://kubernetes.io/docs/concepts/architecture/master-node-communication/>



# Node（节点）

## 什么是Node？

`node` 是Kubernetes中的工作机器（worker），以前被称为 `minion`。集群中的Node可以是VM或物理机。每个Node上都有运行 `pods` 所必要的服务，并由Master组件管理。Node上的服务包括Docker、kubelet和kube-proxy。有关更多详细信息，请参阅架构设计文档中的 [The Kubernetes Node](#) 部分。

## Node状态

Node状态包含如下信息：

- Addresses
- Phase 【弃用】
- Condition
- Capacity
- Info

下面详解每个部分：

## Addresses（地址）

这些字段的用法取决于你云提供商或裸机的配置。

- HostName：Node内核报告的hostname。可通过kubelet的 `--hostname-override` 参数配置。
- ExternalIP：通常是可外部路由的Node IP（可从群集外部获得）。
- InternalIP：通常只能在集群内进行路由的Node IP。

## Phase（阶段）

已弃用，Node phase不再使用。

## Condition（状况）

`conditions` 字段描述所有 `Running` Node的状态。

Node Condition	Description
<code>OutOfDisk</code>	如果节点没有足够的可用空间来添加新的Pod，则为 <code>True</code> ，否则为 <code>False</code>
<code>Ready</code>	如果节点健康并准备好接受Pod，则为 <code>True</code> ；如果节点不健康且不接受Pod，则为 <code>False</code> ，如果node controller与Node失联40秒以上，则为“ <code>Unknown</code> ”
<code>MemoryPressure</code>	如果node的内存存在压力，则为 <code>True</code> ——即node内存容量低；否则为 <code>False</code>
<code>DiskPressure</code>	如果磁盘存在压力，则为 <code>True</code> ——即磁盘容量低；否则为 <code>False</code>
<code>NetworkUnavailable</code>	如果Node的网络未正确配置，则为 <code>True</code> ，否则为 <code>False</code>

node condition使用JSON对象来表示。 例如，以下描述了一个健康的Node。

```
1. "conditions": [
2.   {
3.     "kind": "Ready",
4.     "status": "True"
5.   }
6. ]
```

如果 `Ready` condition的状态为“Unknown”或“False”，并且持续超过 `pod-eviction-timeout`，则会将一个参数传递给 `kube-controller-manager`，并且Node上的所有Pod都会被Node Controller驱逐。默认驱逐的超时时间为五分钟。在某些情况下，当Node不可访问时，apiserver无法与其上的kubelet进行通信。在与apiserver恢复通信之前，删除Pod的指令无法传达到kubelet。同时，计划删除的Pod可能会继续在该Node上运行。

在Kubernetes 1.5之前，Node Controller将强制从apiserver中 `force delete` 这些不可达的pod。但在1.5及更高版本中，Node Controller不会强制删除Pod，直到确认它们已停止运行。

这些不可达Node上运行的Pod会处于“Terminating”或“Unknown”状态。如果Kubernetes无法从底层基础设施推断出Node已永久离开集群，集群管理员可能需要手动删除Node对象。从Kubernetes删除Node对象会导致运行在其上的所有Pod对象从apiserver中删除，并释放其名称。

Kubernetes 1.8引入了一个自动创建代表condition的 `taints` 功能（目前处于Alpha状态）。要启用此特性，请向API server、controller manager和scheduler传递标志 `--feature-gates=...,TaintNodesByCondition=true`。一旦启用 `TaintNodesByCondition`，scheduler将会忽略选择Node时的condition；而是看Node的taint（污点）和Pod的toleration（容忍度）。

译者按：

1. “`taints and tolerations`”的功能是允许你标注（taint）Node，那样Pod就不会调度到这个Node上，除非Pod明确“tolerates”这个“taint”。
2. K8s高级调度特性：<http://blog.csdn.net/jetty/article/details/69500150>

现在用户可在旧调度模型和新的更灵活的调度模型之间选择。没有toleration（容忍度）的Pod根据旧的模型进行调度。但是，对特定Node能够容忍污点（tolerates the taints）的Pod可被调度到该Node。

请注意，由于延迟时间小，通常少于1秒，在观察condition和产生污点的时间段内，启用此功能可能会稍微增加成功调度但被kubelet拒绝的Pod的数量。

## Capacity（容量）

描述Node上可用的资源：CPU、内存，以及可调度到该Node的最大Pod数。

## Info（信息）

关于Node的一般信息，如内核版本、Kubernetes版本（kubelet和kube-proxy版本）、Docker版本（如果使用了Docker的话）、OS名称。信息由Kubelet从Node收集。

## Management（管理）

与 `pods` 、 `services` 不同，Node不是由Kubernetes创建的：它是由Google Compute Engine等云提供商在外部创建的，或存在于物理机或虚拟机池中。这意味着当Kubernetes创建一个Node时，它只是创建一个表示Node的对象。创建后，Kubernetes将检查Node是否有效。例如，如果您尝试从以下内容创建一个Node：

```
1. {
2.   "kind": "Node",
3.   "apiVersion": "v1",
4.   "metadata": {
5.     "name": "10.240.79.157",
6.     "labels": {
7.       "name": "my-first-k8s-node"
8.     }
9.   }
10. }
```

Kubernetes将在内部创建一个Node对象（用来表示Node），并通过基于 `metadata.name` 字段的健康检查来验证Node（我们假设 `metadata.name` 可以被解析）。如果Node通过验证，即：所有必需的服务都处于运行状态，则它有资格运行Pod；否则，它将被忽略，直到通过验证。 请注意，Kubernetes将保留无效Node的对象，并继续检查它是否有效，除非它被客户端明确删除。

目前，有三个组件与Kubernetes Node接口进行交互：`node controller`、`kubelet`和`kubect1`。

## Node Controller

Node Controller是一个Kubernetes Master组件，管理Node的各个方面。

Node Controller在Node的生命周期中具有多个角色。首先，是在注册时将CIDR块分配给Node（如果CIDR分配已打开）。

译者按：CIDR（无类别域间路由）：<http://blog.csdn.net/xinianbuxiu/article/details/53560417>

其次，是使Node Controller的内部列表与云提供商的可用机器列表同步。在云环境中，每当Node不健康时，Node Controller就会请求云提供商，查询该Node的VM是否依然可用。如果不可用，Node Controller就会从其Node列表中删除该Node。

第三，是监视Node的健康状况。当Node变得不可达时，Node Controller负责将NodeStatus的NodeReady condition更新为ConditionUnknown（即：Node Controller由于某些原因停止接收心跳，例如由于Node关闭）。如果Node依然无法访问，那么就会从该Node中驱逐所有Pod（使用优雅关闭的方式）。（Node Controller与Node失联超过40秒，就会报告ConditionUnknown，5分钟后开始驱逐Pod）。Node Controller每隔 `--node-monitor-period` 检查一次Node状态。

在Kubernetes 1.4中，我们更新了Node Controller的逻辑，从而更好地处理大量Node无法连接Master的情况（例如，由于Master有网络问题）。从1.4开始，Node Controller在做关于Pod驱逐的决定时，会查看集群中所有Node的状态。

在大多数情况下，Node Controller将驱逐速率限制为 `--node-eviction-rate`（默认为0.1）每秒，这意味着它不会1个Node驱逐Pod花费的时间不会超过10秒。

当给定可用区 (availability zone) 中的Node变得不健康时, Node驱逐的行为就会发生变化。Node Controller同时也会检查区域中有多少百分比的Node是不正常的 (NodeReady condition是ConditionUnknown或ConditionFalse)。如果不健康Node的阈值达到 `--unhealthy-zone-threshold` (默认为0.55), 那么驱逐速率将被降低: 如果集群很小 (即小于或等于 `--large-cluster-size-threshold` 个Node, 默认50), 那么驱逐就会停止; 否则驱逐速率降低到 `--secondary-node-eviction-rate` (默认0.01) 每秒。每个可用区实施这些策略的原因, 是因为每个可用区都可能会从Master断开, 而另一个可用区仍然保持连接。如果您的集群不会跨越多个云提供商的可用区, 那么就只有一个可用区 (整个群集)。

译者按: 可用区举例: AWS可用区: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>

在可用区之间传播Node的一个关键原因是: 当整个区域停止时, 工作负载可以转移到健康区域。因此, 当一个区域中的所有Node都不健康时, 那么Node Controller就以正常速率 `--node-eviction-rate` 驱逐。当所有区域都不健康时 (即集群中没有健康的Node), Node Controller就会假定Master的连接有问题, 并停止所有驱逐, 直到连接恢复。

从Kubernetes 1.6开始, NodeController还负责驱逐运行在“NoExecute taint”的Node上的Pod, 当Pod不能忍受这些taint时。另外, 作为默认禁用的Alpha功能, NodeController负责添加taint, 这些taint与诸如Node可达或未准备好等问题相关。有关NoExecute taint和Alpha功能的详细信息, 请参阅 [this documentation](#)。

从1.8版开始, Node Controller可负责创建表示节点condition的taint。这是1.8版的Alpha功能。

## Node的自注册

当kubelet标志 `--register-node` 为true (默认值) 时, kubelet将会尝试向API server注册自己。这是大多数版本所使用的首选模式。

对于自注册, kubelet会使用如下的选项启动:

- `--kubeconfig` : 凭证向apiserver进行身份验证的路径。
- `--cloud-provider` : 如何与云提供商进行会话, 从而获取自身的元数据。
- `--register-node` : 自动向API server注册。
- `--register-with-taints` : 注册具有给定taint列表的Node (逗号分隔的 `<key>=<value>:<effect>`)。如果 `register-node` 为false, 则为No-op (空操作, 啥都不干)。
- `--node-ip` : Node的IP。
- `--node-labels` : 向集群注册Node时添加的标签。
- `--node-status-update-frequency` : 指定kubelet将Node状态发送到Master的频率。

目前, 任何kubelet都被授权创建/修改任何Node资源, 但实际上只能创建/修改其自身的资源。(将来, 我们计划只允许一个kubelet修改自己的Node资源。)

## 手动管理Node

集群管理员可创建和修改Node对象。

如果管理员希望手动创建Node对象, 可设置kubelet标志 `--register-node=false`。

管理员可修改Node资源 (忽视 `--register-node` 的设置)。修改操作包括: 在Node上设置标签, 并将其标记为不可

调度。

Node上的Label可与Pod上的Node selector（Node选择器）一起使用，从而控制调度——例如，限制一个Pod只能在指定的节点列表上运行。

将Node标记为不可调度，将会阻止新的Pod被调度到该Node，但不会影响Node上的现有的Pod。这对于做Node重启之前的准备工作很有用。例如，要将node标记为不可调度，可使用如下命令：

```
1. kubectl cordon $NODENAME
```

请注意，由DaemonSet Controller创建的Pod会绕过Kubernetes调度程序，并且不遵循节点上的unschedulable属性。 因为，我们假设daemon进程属于机器，即使在准备重启时正被耗尽。

## Node容量

Node的容量（CPU数量和内存大小）是Node对象的一部分。 通常来说，Node在创建Node对象时注册自身，并报告其容量。如果您正在进行 [manual node administration](#)，则需要在添加Node时设置Node容量。

Kubernetes调度程序可确保Node上的所有pod都有足够的资源。它会检查节点上容器的请求总和不大于Node容量。它包括由kubelet启动的所有容器，但不包括由Docker直接启动的容器，也不包含那些不运行在容器中的进程。

如果要明确保留非Pod进程的资源，可创建一个“placeholder pod（占位Pod）”。使用以下模板：

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: resource-reserver
5. spec:
6.   containers:
7.   - name: sleep-forever
8.     image: gcr.io/google_containers/pause:0.8.0
9.     resources:
10.      requests:
11.        cpu: 100m
12.        memory: 100Mi
```

将 `cpu` 和 `memory` 值设置为您要保留的资源量。将该文件放在清单目录中（kubelet的 `--config=DIR` 标志）。 在想要预留资源的每个kubelet上执行此操作。

## API对象

Node是Kubernetes REST API中的顶级资源。有关API对象的更多详细信息，可详见：[Node API object](#)。

## 原文

<https://kubernetes.io/docs/concepts/architecture/nodes/>

# 从Pod说起

- Pod是Kubernetes中可以创建和管理的最小部署单元。是K8s种最小的调度单位
- 可以认为是一个“虚拟机”
- 是一个逻辑概念

拓展与提问：如果没有Pod，而把容器作为最小的调度单位，由于容器是“单进程模型”，会出现什么问题？

举个例子：rsyslogd包括三个模块（进程）：imklog、imuxsock、rsyslogd。三个进程必须运行在一台机器上，否则基于Socket的通信和文件交换会出问题。于是，我们把三个模块分别制作成三个不同的容器，并未三个容器分别分配1G的内存。

如果没有Pod，那么为了让三个容器运行在一台node上，我们可能会这么做：首先启用imklog，让它调度到一台服务器（记为Node1）上，然后将imuxsock、rsyslogd也调度到imklog所在的机器上（一般通过设置亲和性的方式）。但假如Node1总共只有2G的内存，那么就势必调度失败；但是呢，由于设置了亲和性，又意味着这三个容器必须调度到Node1上。

这是一个非常经典的调度问题。不同的容器编排框架采用不同的方式去解决以上问题。例如：Mesos采用资源囤积的方式解决这个问题（即：当且仅当所有设置亲和性约束的任务都达到时，才统一调度）

Kubernetes采用Pod的方式，也良好地解决了该问题。

## 什么是Pod

Pod（像鲸鱼群或豌豆荚）由一个或多个容器（例如Docker容器）组成的容器组，组内容器具有共享存储、网络，以及容器的运行规范。Pod的内容始终是被同时调度。Pod可被认为是运行特定应用程序的“逻辑主机”——它包含一个或多个相对紧密耦合的应用容器——在容器启动之前，应用容器总是会被调度到在相同的Node上。

尽管Kubernetes比Docker支持更多的容器运行时，但Docker是最常见的运行时，这样有助于使用Docker术语中描述Pod。

Pod的共享上下文是一组Linux命名空间、cgroups和潜在的其他方面的隔离机制——这一点与Docker容器的隔离机制一致。在Pod的上下文中，各个应用程序可能会有更小的子隔离环境。

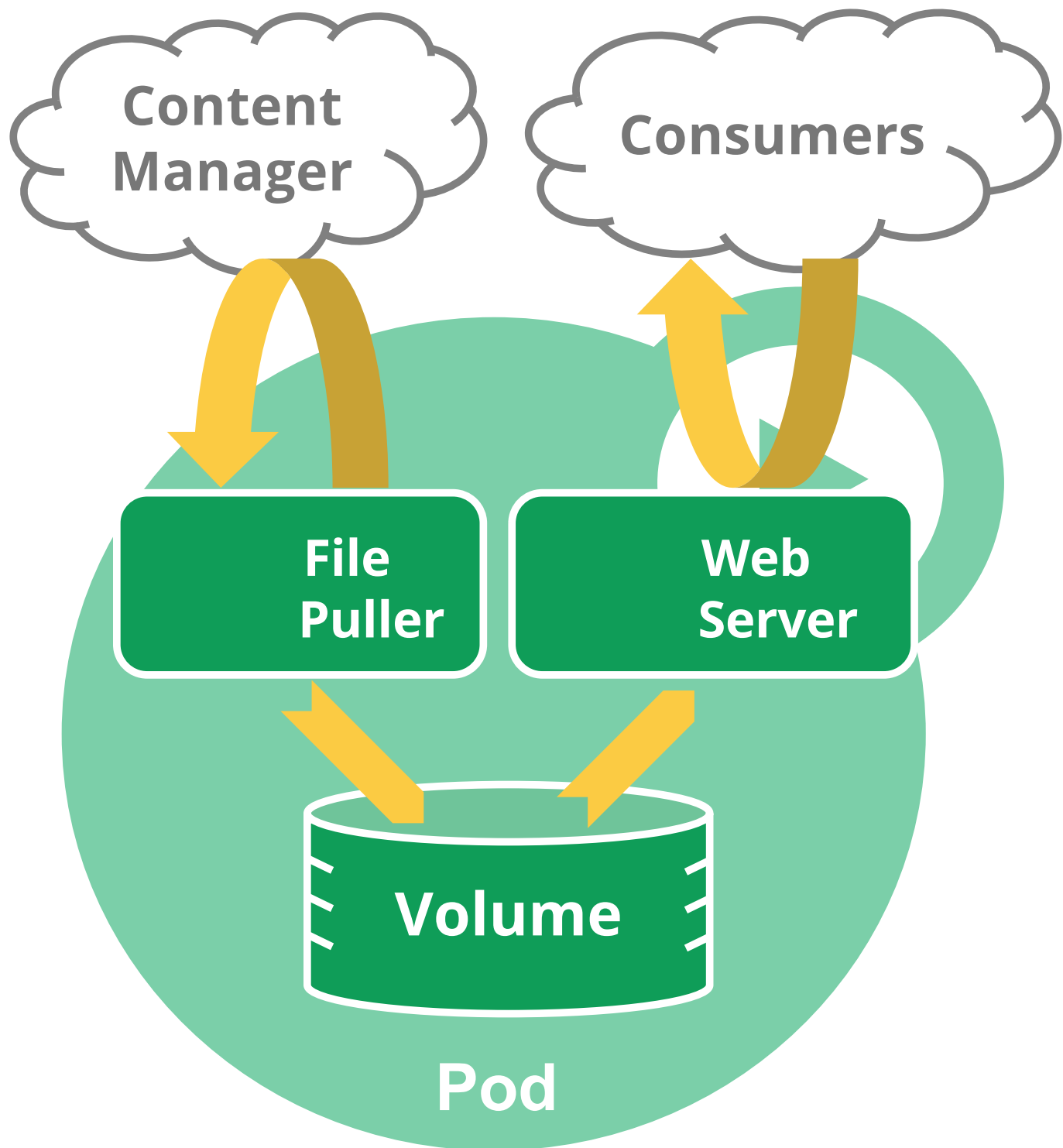
Pod中的容器共享IP地址和端口，并且可通过 `localhost` 找到对方。它们还可使用标准的进程间通信（IPC）（例如SystemV信号量或POSIX共享内存）进行通信。不同Pod中的容器有不同的IP地址，不能通过IPC进行通信。

Pod中的应用程序还可以访问共享卷，这些卷定义为Pod的一部分并挂载到每个应用程序的文件系统中。

根据Docker的结构，Pod被建模为一组具有共享namespace和**共享卷**的Docker容器。暂不支持PID namespace的共享。

和应用容器一样，Pod被认为是相对短暂的（非持久）实体。如在[Pod 生命周期](#)所讨论的，创建Pod后，会分配唯一的ID（UID），并将其调度到Node上，直到终止（根据重启策略）或删除。如果一个Node挂掉，那么在经过超时时间后，调度到该节点的Pod将被删除。一个给定的（例如UID定义的）Pod不会“重新调度”到新的Node上；而是被相同的Pod取代。如有需要，甚至可使用相同的名称，但会使用新的UID（有关更多详细信息，请参阅[replication controller](#)）。（将来，会有更高级别的API将支持Pod迁移。）

当某些东西与一个Pod的生命周期（例如Volume）相同时，这意味着只要该Pod（具有该UID）存在就存在。如果由于任何原因删除了该Pod，那么，即使创建了相同的Pod（以替代之前的Pod），相关内容（例如Volume）也会被销毁并重新创建。



## Pod的动机

### 管理

Pod是一个服务中多个进程的聚合单位，通过更高级别的抽象来简化应用程序的部署和管理。在K8s中，Pod是部署，横向缩放和复制的单位。Pod中的容器可自动处理托管（共同调度）、共享命运（例如终止）、协调复制，资源共享和依赖关系管理。

### 资源共享和通信



Pods中的成员之间可进行数据共享和通信。

Pod中的应用程序都使用相同的网络namespace（相同的IP和端口），因此可使用 `localhost` 互相发现。因此，Pod中的应用程序必须协调其端口的使用。每个Pod都会在一个扁平的共享网络空间中分配一个IP地址，从而与其他物理机以及Pod通信。

对于Pod中的容器，主机名会被设为Pod的名称。 [更多相关网络的内容](#)。

除定义应用程序容器之外，Pod还指定了一组共享Volume。Volume允许数据在容器重启之后不丢失，并在Pod中的应用程序之间共享。

## Pod的使用

Pods可用于托管垂直集成的应用栈（例如LAMP），但其主要动机是支持共同协作、共同管理的工作程序，例如：

- 内容管理系统，文件和数据加载器，本地缓存管理等
- 日志和检查点备份，压缩，旋转，快照等
- 数据更改观察者，日志分配器，日志记录和监视适配器，事件发布者等
- 代理，桥接器和适配器
- 控制器，管理器，配置器和更新器

一般来说，单个Pod不会运行同一应用的多个实例。

详情请看[The Distributed System ToolKit: Patterns for Composite Containers](#)

## 考虑替代方案

为什么不在单个（*Docker*）容器中运行多个程序？

1. 透明。使Pod中的容器对基础设施可见，以便基础设施为这些容器提供服务，例如进程管理和资源监控。这为用户提供了一些便利。
2. 解耦软件依赖。各个容器可以独立进行版本管理、重建和重新部署。未来，Kubernetes甚至有可能支持单个容器的实时更新。
3. 使用方便。用户无需运行自己的进程管理器，担心信号以及退出代码传播等。
4. 效率。因为基础设施承担起更多的责任，容器更加轻量级。

为什么不支持基于亲和性部署的容器协同调度？

这种方法将会提供协同定位，但无法提供Pod的大部分优势，例如资源共享，IPC（进程间通信），保证命运共享和简化管理。

## Pod的持久性（或缺乏持久性）

Pod不能被视为持久的实体。它们不会因调度失败，节点故障或其他问题而生存，例如由于缺乏资源，或者在节点维护的情况下。

一般来说，用户无需直接创建Pod。他们几乎总是使用Controller（例如 [Deployment](#)），即使是创建单个Pod



时。Controller提供集群范围的自我修复、复制和升级管理。

集体API作为面向用户的语言的方式，在在集群调度系统中相对比较常见，包括 [Borg](#) 、 [Marathon](#) 、 [Aurora](#) 以及 [Tupperware](#) 都采用这种方式。

Pod被暴露为原始API，以便于：

- 调度程序和控制器可插拔性
- 支持Pod级操作，而无需通过控制器API“代理”它们
- 将Pod生命周期与控制器生命周期解耦
- 控制器和服务的解耦——端点控制器秩序观察Pod
- 清晰地将Kubelet级的功能与集群级的功能组合——Kubelet实际上是“Pod控制器”
- 高可用性应用，Pod可在其终止或删除之前被替换，例如在计划的驱逐，图像预取或Pod实时移植的情况下 [#3949](#)

[StatefulSet](#) 控制器（目前处于Beta测试阶段），支持有状态Pod。该功能在1.4中是alpha测试阶段，被称为PetSet。对于先前版本的Kubernetes，有状态Pod的最佳做法是创建一个replication controller，其 `replicas` 等于 `1` 并提供相应的服务，详见[this MySQL deployment example](#)。

## Pod的终止

因为Pod代表集群中Node上运行的进程，所以当不再需要这些进程时，允许这些进程正常终止（比起使用KILL信号这种暴力的方式）是非常重要的。用户应该能够请求删除并知道进程何时终止，但也能够确保删除最终完成。当用户请求删除Pod时，系统会在Pod被强制杀死之前记录预期的优雅关闭时间，并且TERM信号被发送到每个容器的主进程。一旦超过优雅关闭时间，就会向这些进程发送KILL信号，然后从API Server中删除该Pod。如果在等待进程终止的过程中Kubelet或Container Manager重启了，那么，在重启后仍会重试完整优雅关闭。

示例流程：

1. 用户发送删除Pod的命令，默认优雅关闭时间是30s
2. 随着时间的推移，API Server中的Pod状态会被更新，Pod会被标记为“dead”，并开始进入优雅关闭时间
3. 当在客户端命令行中列出Pod时，状态显示为“Terminating”
4. （与3同时）当Kubelet看到Pod已被标记为Terminating，因为2中的时间已经设置，它将开始Pod关闭进程。
5. 如果Pod定义了[preStop hook](#)，它将在Pod中被调用。如果 `preStop` Hook在优雅关闭时间到期后仍在运行，则会在第二步中增加2秒的优雅关闭时间。
6. Pod中的进程发送TERM信号。
7. （与3同时），Pod将从该服务的端点列表中删除，并且不再被认为是replication controllers正在运行的Pod的一部分。缓慢关闭的Pod可以继续接收从load balancer转发过来的流量，直到load balancer将其从可用列表中移除。
8. 当优雅关闭时间到期时，仍在Pod中运行的任何进程都会被SIGKILL杀死。
9. 通过设置优雅关闭时间为0（立即删除），Kubelet将在API Server上完成Pod的删除。Pod从API消失，并且在客户端中也不可见。

默认情况下，优雅删除时间是30秒。`kubectl delete` 命令支持 `--grace-period=<seconds>` 选项，允许用户覆盖默认值并指定自己的值。如果设置为 `0`，则表示 [强制删除](#) Pod。当kubectl version >= 1.5时，您必须同时使用标志 `--force` 与 `--grace-period=0` 来强制删除Pod。

## 强制删除Pod

一个Pod的强制删除被定义为从群集状态和etcd立即删除一个Pod。 当执行强制删除时，API Server不等待来自Kubelet的确认，而是直接在其运行的Node上终止该Pod。它立即删除API Server中的Pod，以便创建一个新的同名Pod。在Node上，被设为立即终止的Pod在被强制杀死之前仍然会有一个较小的优雅关闭时间。

强制删除对于某些Pod可能是危险的，应慎用。在StatefulSet Pod的情况下，请参阅[deleting Pods from a StatefulSet](#)。

## Pod phase

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

Pod的phase是Pod在其生命周期中的简单、高级的概述。phase并不是对容器或Pod状态的综合汇总，也不是作为为了做为状态机。

phase值的数量和含义被严格保护。除了这里记录的内容，不应该假定 `phase` 有其他值。

以下是 `phase` 可能的取值：

- Pending：Pod已被Kubernetes系统接受，但一个或多个容器镜像尚未创建。该时间调度Pod所花费的时间以及通过网络下载镜像的时间，这可能需要一段时间。
- Running：Pod已绑定到一个节点，并且所有的容器都已创建。至少有一个容器仍在运行，或正在启动或重新启动。
- Succeeded：Pod中的所有容器已成功终止，不会重新启动。
- Failed：Pod中的所有容器都已终止，并且至少有一个容器已终止失败。也就是说，容器以非零状态退出或被系统终止。
- Unknown：由于某种原因，无法获得Pod状态，通常是由于与Pod所在主机通信时出现错误。

## Pod Condition

Pod有一个PodStatus，它有一个PodConditions 数组。 PodCondition数组的每个元素都有一个 `type` 字段和一个 `status` 字段。 `type` 字段是一个字符串，可能的值为 `PodScheduled`、`Ready`、`Initialized`以及 `Unschedulable` 。 `status` 字段是一个字符串，可能的值为 `True`、`False`和`Unknown` 。

## 容器探针

`Probe` 是由kubernetes 对容器定期执行的诊断。要执行诊断，Kubelet调用由Container实现的Handler 。 有三种类型的handler：

- `ExecAction`：在Container中执行指定的命令。 如果命令退出的状态码为0，则认为诊断成功。
- `TCPSocketAction`：对容器IP的指定端口执行TCP检查。如果端口打开，则认为诊断成功。
- `HTTPGetAction`：对容器IP的指定端口和路径执行HTTP Get请求。如果响应的状态码大于等于200且小于400，则认为诊断成功。

每个探针都有三个结果之一：

- **Success**：容器已通过诊断。
- **Failure**：容器未通过诊断。
- **Unknown**：诊断失败，因此不会采取任何行动。

kubelet可以选择对运行容器上的两种探针执行和反应：

- **livenessProbe**（活动探针）：指示容器是否正在运行。如活动探测失败，那么kubelet就会杀死容器，并且容器将受到其 **重启策略** 的影响。如容器不提供活动探针，则默认状态为 **Success**。
- **readinessProbe**（就绪探针）：指示容器是否准备好服务请求。如就绪探测失败，Endpoint Controller将会从与Pod匹配的所有Service的端点中删除该Pod的IP地址。初始延迟之前的就绪状态默认为 **Failure**。如果容器不提供就绪探针，则默认状态为 **Success**。

## 什么时候应该使用活动探针或就绪探针？

如果容器中的进程能够在遇到问题或不健康的情况下自行崩溃，则不一定需要活动探针；kubelet将根据Pod的 **restartPolicy** 自动执行正确的操作。

如果您希望容器在探测失败时被杀死并重新启动，那么请指定一个活动探针，并指定 **restartPolicy** 为Always或OnFailure。

如果要仅在探测成功时才开始向Pod发送流量，请指定就绪探针。在这种情况下，就绪探针可能与活动探针相同，但是spec中存在就绪探针，就意味着Pod会在没有接收到任何流量的情况下启动，并且只在探针探测成功后才开始接收流量。

如果您希望容器能够自己维护，可指定一个就绪探针，该探针检查的端点与活动探针不同。

请注意，如果您只想在Pod被删除时排除请求，则不一定需要就绪探针；在删除时，Pod会自动将自身置于未完成状态，无论就绪探针是否存在。在等待Pod中的容器停止的过程中，Pod仍处于未完成状态。

## Pod及容器状态

有关Pod容器状态的详细信息，请参阅 [PodStatus](#) 和 [ContainerStatus](#)。请注意，作为Pod状态报告的信息取决于当前的 [ContainerState](#)。

## 重启策略

PodSpec有一个 **restartPolicy** 字段，可能的值为Always，OnFailure和Never，默认值为Always。

**restartPolicy** 适用于Pod中的所有容器。**restartPolicy** 仅指同一Node上kubelet重启容器时所使用的策略。失败的容器由kubelet重启，以五分钟上限的指数退避延迟（10秒，20秒，40秒...），并在成功执行十分钟后重置。如 [Pods document](#) 中所述，一旦绑定到一个Node，Pod将永远不会重新绑定到另一个Node。

## Pod的寿命

一般来说，Pod不会消失，直到有人销毁它们——可能是人工或Controller去销毁Pod。这个规则的唯一例外是Pod的 **phase** 成功或失败超过一段时间（由Master确定）的Pod将过期并被自动销毁。

有三种类型的Controller可用：

- **Job** ，Pod预期会终止，例如批量计算。Job仅适用于 `restartPolicy` 为OnFailure或Never的Pod。
- **ReplicationController** 、 **ReplicaSet** 以及 **Deployment** ，Pod预期不会终止，例如Web服务器。ReplicationController仅适用于 `restartPolicy` 为Always的Pod。
- **DaemonSet** ，每台机器都需要运行一个Pod，因为它们提供特定于机器的系统服务。

以上三种类型的Controller都包含一个PodTemplate。建议创建适当的Controller，让Controller创建Pod，而非直接创建Pod。这是因为单独的Pod在机器故障发生时无法自我修复，而Controller可以。

如果Node挂掉或与集群断开，那么Kubernetes应用一种策略，将故障Node上的所有Pod的 `phase` 设为Failed。

## 示例

### 高级活动探针示例

活动探测由kubelet执行，因此所有请求都会在kubelet网络命名空间中进行。

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   labels:
5.     test: liveness
6.   name: liveness-http
7. spec:
8.   containers:
9.     - args:
10.       - /server
11.       image: gcr.io/google_containers/liveness
12.       livenessProbe:
13.         httpGet:
14.           # when "host" is not defined, "PodIP" will be used
15.           # host: my-host
16.           # when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and "HTTPS" are allowed
17.           # scheme: HTTPS
18.           path: /healthz
19.           port: 8080
20.           httpHeaders:
21.             - name: X-Custom-Header
22.               value: Awesome
23.           initialDelaySeconds: 15
24.           timeoutSeconds: 1
25.       name: liveness

```

容器的源码：

```

1. http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
2.     duration := time.Now().Sub(started)
3.     if duration.Seconds() > 10 {

```

```

4.     w.WriteHeader(500)
5.     w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
6. } else {
7.     w.WriteHeader(200)
8.     w.Write([]byte("ok"))
9. } })

```

参考: <http://blog.51cto.com/kusorz/2069412>

## 状态示例

- Pod正在运行，并只有一个容器。容器退出成功。
  - 记录完成事件
  - 如果 `restartPolicy` 是：
    - Always: 重启容器，Pod的 `phase` 保持Running
    - OnFailure: Pod的 `phase` 变成Succeeded.
    - Never: Pod的 `phase` 变成Succeeded.
- Pod正在运行，并只有一个容器。容器退出失败。
  - 记录失败事件
  - 如果 `restartPolicy` 是：
    - Always: 重启容器，Pod的 `phase` 保持Running.
    - OnFailure: 重启容器，Pod的 `phase` 保持Running.
    - Never: Pod的 `phase` 变成Failed.
- Pod正在运行，并且有两个容器，Container 1退出失败。
  - 记录失败事件
  - 如果 `restartPolicy` 是：
    - Always: 重启容器，Pod的 `phase` 保持Running.
    - OnFailure: 重启容器；Pod的 `phase` 保持Running.
    - Never: 不会重启容器，Pod的 `phase` 保持Running.
  - 如果Container 1不在运行，Container退出
    - 记录失败事件
    - 如果 `restartPolicy` 是：
      - Always: 重启容器，Pod的 `phase` 保持Running.
      - OnFailure: 重启容器，Pod的 `phase` 保持Running.
      - Never: Pod的 `phase` 变成Failed.
- Pod正在运行，并且只有一个容器，容器内存溢出：
  - 容器失败并终止
  - 记录OOM事件
  - 如果 `restartPolicy` 是：
    - Always: 重启容器，Pod的 `phase` 保持Running.
    - OnFailure: 重启容器，Pod `phase` 保持Running.
    - Never: 记录失败事件，Pod `phase` 变成Failed.
- Pod正在运行，磁盘损坏：
  - 杀死所有容器
  - 记录适当事件
  - Pod的 `phase` 变成Failed
  - 如果Pod是用Controller创建的，Pod将在别处重建

- Pod正在运行，Node被分段
  - Node Controller等待，直到超时
  - Node Controller将Pod的 `phase` 设为Failed
  - 如果Pod是用Controller创建的，Pod将在别处重建

# Replica Set

**ReplicaSet**是下一代**Replication Controller**。**\*ReplicaSet**和 **[Replication Controller\*]** (<https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/>) 之间的唯一区别就是选择器支持。ReplicaSet支持 [labels user guide](#) 描述的新的set-based selector requirement, 而Replication Controller仅支持equality-based selector requirement。

关于Label Selector: <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/> , 本文的示例种也有用到两种selector。

## 如何使用ReplicaSet

支持Replication Controller的大多数 `kubectl` 命令也支持ReplicaSets。 `rolling-update` 命令是一个例外。如果您想要滚动更新功能, 请考虑使用Deployment。此外, `rolling-update` 命令是必不可少的, 而Deployment是声明式的, 因此我们建议您通过 `rollout` 命令使用Deployment。

虽然ReplicaSet可独立使用, 但是目前它主要被 **Deployment** 作为编排Pod创建、删除和更新的机制。当您使用Deployment时, 您不必担心如何管理Deployment创建的ReplicaSet。Deployment拥有并管理其ReplicaSet。

## 何时使用ReplicaSet

ReplicaSet确保在任何时间都会运行指定数量的Pod副本。但是, Deployment是一个更高层次的概念——它管理ReplicaSet, 并提供对Pod的声明性更新以及许多其他有用的功能。因此, 我们建议您使用Deployment而不是直接使用ReplicaSet, 除非您需要自定义更新编排或根本不需要更新。

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

## 示例

```
1. apiVersion: apps/v1beta2 # for versions before 1.6.0 use extensions/v1beta1
2. kind: ReplicaSet
3. metadata:
4.   name: frontend
5.   labels:
6.     app: guestbook
7.     tier: frontend
8. spec:
9.   # this replicas value is default
10.  # modify it according to your case
11.  replicas: 3
12.  selector:
13.    # 下面的是equality-based selector requirement
14.    matchLabels:
15.      tier: frontend
```

```

16.     # 下面是set-based selector requirement
17.     matchExpressions:
18.       - {key: tier, operator: In, values: [frontend]}
19.   template:
20.     metadata:
21.       labels:
22.         app: guestbook
23.         tier: frontend
24.     spec:
25.       containers:
26.       - name: php-redis
27.         image: gcr.io/google_samples/gb-frontend:v3
28.         resources:
29.           requests:
30.             cpu: 100m
31.             memory: 100Mi
32.         env:
33.         - name: GET_HOSTS_FROM
34.           value: dns
35.           # If your cluster config does not include a dns service, then to
36.           # instead access environment variables to find service host
37.           # info, comment out the 'value: dns' line above, and uncomment the
38.           # line below.
39.           # value: env
40.       ports:
41.       - containerPort: 80

```

将此清单保存为 `frontend.yaml`，并将其提交给Kubernetes集群，即可创建你所定义的ReplicaSet以及ReplicaSet管理的Pod。

```

1. $ kubectl create -f frontend.yaml
2. replicaset "frontend" created
3. $ kubectl describe rs/frontend
4. Name:          frontend
5. Namespace:     default
6. Selector:      tier=frontend,tier in (frontend)
7. Labels:        app=guestbook
8.                tier=frontend
9. Annotations:   <none>
10. Replicas:      3 current / 3 desired
11. Pods Status:   3 Running / 0 Waiting / 0 Succeeded / 0 Failed
12. Pod Template:
13.   Labels:       app=guestbook
14.                tier=frontend
15.   Containers:
16.   php-redis:
17.     Image:       gcr.io/google_samples/gb-frontend:v3
18.     Port:        80/TCP
19.     Requests:
20.       cpu:       100m
21.       memory:    100Mi
22.     Environment:

```



```

23.      GET_HOSTS_FROM:    dns
24.      Mounts:             <none>
25.      Volumes:            <none>
26.  Events:
27.    FirstSeen    LastSeen    Count   From              SubobjectPath   Type        Reason          Message
28.    -
29.    1m           1m          1       {replicaset-controller }   Normal        SuccessfulCreate Created
    pod: frontend-qhloh
30.    1m           1m          1       {replicaset-controller }   Normal        SuccessfulCreate Created
    pod: frontend-dnjpy
31.    1m           1m          1       {replicaset-controller }   Normal        SuccessfulCreate Created
    pod: frontend-9si5l
32. $ kubectl get pods
33. NAME                READY    STATUS    RESTARTS   AGE
34. frontend-9si5l       1/1     Running   0           1m
35. frontend-dnjpy       1/1     Running   0           1m
36. frontend-qhloh       1/1     Running   0           1m

```

## 编写ReplicaSet的spec

与所有其他Kubernetes API对象一样，ReplicaSet需要 `apiVersion` 、 `kind` 和 `metadata` 等字段。关于使用清单的一般信息，请参阅 [here](#) 、 [here](#) 和 [here](#) 。

ReplicaSet还需要一个 `.spec` [section](#) 。

## Pod模板

`.spec.template` 是 `.spec` 唯一必需的字段。`.spec.template` 是一个 `pod template` 。它与 `pod` 有完全相同的模式—除了是嵌套的，没有 `apiVersion` 以及 `kind` 以外。

除Pod必需的字段外，ReplicaSet中的Pod模板必须指定适当的标签和适当的重新启动策略。

对于标签，请确保不会与其他Controller重叠。有关更多信息，请参阅 [pod selector](#) 。

对于 `restart policy` ， `.spec.template.spec.restartPolicy` 的唯一允许的值是 `Always` ，这是默认值。

对于本地容器重新启动，ReplicaSet将委托给Node上的代理，例如 [Kubelet](#) 或Docker。

## Pod选择器

`.spec.selector` 字段是一个 `label selector` 。一个ReplicaSet管理所有与标签选择器相匹配的Pod。它不区分其创建或删除的Pod，也不区分另一个人或进程创建或删除的Pod。这允许我们在不影响正在运行的Pod的前提下替换ReplicaSet。

`.spec.template.metadata.labels` 必须与 `.spec.selector` 匹配，否则将被API拒绝。

在Kubernetes 1.8中，ReplicaSet类型上当前的API版本是 `apps/v1beta2` ，默认启用。API版本 `extensions/v1beta1` 已被弃用。在API版本 `apps/v1beta2` 中，如果没有设置，则 `.spec.selector` 和 `.metadata.labels` 默认不再与 `.spec.template.metadata.labels` 一致。因此，必须明确设定这些字段。另外，在API版本 `apps/v1beta2` 中，一旦ReplicaSet创建，`.spec.selector` 是不可变的。

此外，您通常不应创建任何标签与此选择器匹配的Pod，不管是直接创建、使用ReplicaSet或其他Controller（例如Deployment）。如果这样做，ReplicaSet会认为它创建了其他Pod。Kubernetes并不会阻止你这样做。

如果您最终使用具有重叠选择器的多个Controller，则必须自行管理删除。

## ReplicaSet上的标签

ReplicaSet本身可以有标签（`.metadata.labels`）。通常，您应该将这些该字段设置为与`.spec.template.metadata.labels`相同。但是，也允许不同，`.metadata.labels`不会影响ReplicaSet的行为。

## Replicas（副本）

您可以通过设置`.spec.replicas`来指定同时运行多少个Pod。任何时间运行的Pod个数都可能会更高或更低，例如，如果replica刚刚增加或减少；或者如果Pod优雅关闭，而替换提前启动。

如果不指定`.spec.replicas`，则默认为1。

## 使用ReplicaSet

### 删除ReplicaSet和其Pod

可使用`kubectl delete`删除ReplicaSet及其所有pod。Kubectl将ReplicaSet缩放为零，并等待它删除每个Pod，然后再删除ReplicaSet本身。如果这个kubectl命令被中断，可以重启。

当使用REST API或Go语言客户端库时，需要明确执行这些步骤（将副本缩放为0，等待Pod删除，然后删除ReplicaSet）。

### 只删除ReplicaSet

可以只删除ReplicaSet，而不影响ReplicaSet的任何Pod，使用`kubectl delete`和`--cascade=false`选项即可。

使用REST API或Go语言客户端库时，只需删除ReplicaSet对象即可。

原始的ReplicaSet被删除后，您可以创建一个新的ReplicaSet来替换它。只要新旧ReplicaSet的`.spec.selector`相同，那么新ReplicaSet就会使用旧ReplicaSet的Pod。然而，它不会努力使已存在的Pod去匹配一个新的、不同的Pod模板。要想以可控的方式将Pod更新为新的spec，请使用`rolling update`。

### 从ReplicaSet隔离Pod

可以通过更改其标签的方式，从ReplicaSet中删除Pod。此技术可用于从Service中删除Pod，从而进行debug、数据恢复等。以这种方式删除的Pod将被自动替换（假设副本的数量也不会改变）。

### ReplicaSet伸缩

只需更新 `.spec.replicas` 字段即可轻松缩放ReplicaSet。ReplicaSet controller会确保集群中有指定数量的Pod可用并可操作。

## ReplicaSet作为Horizontal Pod Autoscaler (HPA) 目标

ReplicaSet也可以是 [Horizontal Pod Autoscalers \(HPA\)](#) 的目标。也就是说，ReplicaSet可以由HPA自动伸缩。以下是一个HPA指向我们在上一个示例中创建的ReplicaSet的示例。

```
1. apiVersion: autoscaling/v1
2. kind: HorizontalPodAutoscaler
3. metadata:
4.   name: frontend-scaler
5. spec:
6.   scaleTargetRef:
7.     kind: ReplicaSet
8.     name: frontend
9.   minReplicas: 3
10.  maxReplicas: 10
11.  targetCPUUtilizationPercentage: 50
```

将此清单保存为 `hpa-rs.yaml` 并将其提交到Kubernetes集群，这样就会创建一个HPA，根据Pod副本的CPU使用率自动调整目标ReplicaSet。

```
1. kubectl create -f hpa-rs.yaml
```

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

```
1. kubectl autoscale rs frontend
```

## ReplicaSet的替代方案

### Deployment（推荐）

`Deployment` 是一种更高级的API对象，它以与 `kubectl rolling-update` 类似的方式，更新其底层的ReplicaSet及其Pod。如果您想要滚动更新的功能，则建议使用Deployment，因为与 `kubectl rolling-update` 不同，它们是声明式、服务器端的，并且具有其他特性。有关使用Deployment运行无状态应用的更多信息，请阅读 [Run a Stateless Application Using a Deployment](#)。

### Bare Pod（裸Pod）

与用户直接创建Pod的情况不同，ReplicaSet会替换由于任何原因被删除或终止的pod，例如在Node故障或Node中断维护（例如内核升级）的情况下。因此，我们建议您使用ReplicaSet，即使您的应用程序只需要一个Pod。与process supervisor（进程管理器）类似，ReplicaSet只能监视多个Node上的多个Pod，而不是单个Node上的单个进程。ReplicaSet将本地容器的重启委托给Node上的某个代理（例如，Kubelet或Docker）。

## Job（作业）

对于可预期会终止的Pod（即批处理作业），可以使用 `Job` 而非ReplicaSet。

## DaemonSet

对于提供机器级功能（例如机器监控或日志）的Pod，请使用 `DaemonSet` 而非ReplicaSet。这些Pod的生命周期与机器的生命周期相关：在其他Pod启动之前，这些Pod需要在机器上运行；当机器准备重启/关闭时，可安全终止这些Pod。

## 原文

---

<https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>

# Deployment

- RC只支持基于等式的selector ( env=dev或environment!=qa )，但Replica Set还支持新的，基于集合的selector ( version in (v1.0, v2.0)或env notin (dev, qa) )，这对复杂的运维管理很方便。
- 使用Deployment升级Pod，只需要定义Pod的最终状态，k8s会为你执行必要的操作，虽然能够使用命令 `# kubectl rolling-update` 完成升级，但它是在客户端与服务端多次交互控制RC完成的，所以REST API中并没有rolling-update的接口，这为定制自己的管理系统带来了一些麻烦。
- Deployment拥有更加灵活强大的升级、回滚功能。

Deployment Controller为 Pod 和 ReplicaSet 提供声明式的更新。

只需在Deployment对象中描述所期望的状态，Deployment Controller就会以受控的速率将实际状态逐步转变为你所期望的状态。您可以定义Deployment以创建新的ReplicaSet，也可删除现有Deployment并让新的Deployment采用其所有资源。

注意：您不应该管理Deployment所拥有的ReplicaSet。而应该操作Deployment对象从而管理ReplicaSet。如果你认为有必要直接管理Deployment所拥有的ReplicaSet的场景，请考虑在Kubernetes repo中提Issue。

## 用例

以下是Deployment的典型用例：

- [Create a Deployment to rollout a ReplicaSet](#)（创建Deployment从而升级ReplicaSet）。ReplicaSet在后台创建Pod。检查升级的状态，看是否成功。
- 通过更新Deployment的PodTemplateSpec来 [Declare the new state of the Pods](#)（声明Pod的新状态）。这样，会创建一个新的ReplicaSet，并且Deployment会以受控的速率，将Pod从旧ReplicaSet移到新的ReplicaSet。每个新的ReplicaSet都会更新Deployment的修订版本。
- 如果的Deployment当前状态不稳定，则 [Rollback to an earlier Deployment revision](#)（回滚到之前的Deployment修订版本）。每次回滚都会更新Deployment的修订版本。
- [Scale up the Deployment to facilitate more load](#)（扩展Deployment，以便更多的负载）
- [Pause the Deployment](#)（暂停Deployment），从而将多个补丁应用于其PodTemplateSpec，然后恢复它，开始新的升级。
- [Use the status of the Deployment](#)（使用Deployment的状态）作为升级卡住的指示器。
- 清理您不再需要的 [Clean up older ReplicaSets](#)（清理旧的ReplicaSet）

## 创建Deployment

以下是Deployment的示例。它创建一个包含三个 `nginx` Pod的ReplicaSet：

```
1. apiVersion: apps/v1
2. kind: Deployment
3. metadata:
4.   name: nginx-deployment
5.   labels:
6.     app: nginx
7. spec:
```

```

8.   replicas: 3
9.   selector:
10.    matchLabels:
11.      app: nginx
12.   template:
13.     metadata:
14.       labels:
15.         app: nginx
16.     spec:
17.       containers:
18.       - name: nginx
19.         image: nginx:1.7.9
20.         ports:
21.         - containerPort: 80

```

在本例中：

- 将创建名为 `nginx-deployment` 的Deployment，由 `metadata: name` 字段定义。
- Deployment创建三个Pod副本，由 `replicas` 字段定义。
- Pod模板的规范，即：`template: spec` 字段定义Pod运行一个 `nginx` 容器，它运行1.7.9版的 `nginx` [Docker Hub](#) 镜像。
- Deployment打开80端口供Pod使用。

`template` 字段包含以下说明：

- Pod被打上了 `app: nginx` 的标签
- 创建一个名为 `nginx` 的容器。
- 运行 `nginx 1.7.9` 镜像。
- 打开端口 `80`，以便容器可以发送和接收流量。

要创建此Deployment，请运行以下命令：

```

1. kubectl create -f
   https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/concepts/workloads/controllers/nginx-deployment.yaml

```

注意：您可以将 `--record` 附加到此命令以在资源的annotation中记录当前命令。这对将来的审查（review）很有用，例如调查在每个Deployment修订版中执行了哪些命令。

接下来，运行 `kubectl get deployments`，将会输出类似如下内容：

1. NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
2. nginx-deployment	3	0	0	0	1s

当您inspect集群中的“Deployment”时，将会显示以下字段：

- `NAME` 列出了集群中Deployment的名称。
- `DESIRED` 显示您创建Deployment时所定义的期望副本数。这是所需的状态。

- **CURRENT** 显示当前正在运行的副本数。
- **UP-TO-DATE** 显示已更新，从而实现期望状态的副本数。
  - 译者按：该字段常用于在滚动升级时，显示有多少个Pod副本已成功更新到最新版本。
- **AVAILABLE** 显示当前有可用副本的数量。
- **AGE** 显示应用程序已经运行了多久。

注意每个字段中的值如何对应于Deployment规范中的值：

- 根据 `spec.replicas` 字段，期望的副本数量是3。
- 根据 `.status.replicas` 字段，当前副本的数量为0。
- 根据 `.status.updatedReplicas` 字段，最新副本的数量为0。
- 根据 `.status.availableReplicas` 字段，可用副本的数量为0。

要查看Deployment升级的状态，请运行 `kubectl rollout status deployment/nginx-deployment`。此命令将会返回类似如下的输出：

```
1. Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
2. deployment "nginx-deployment" successfully rolled out
```

几秒钟后再次运行 `kubectl get deployments`：

```
1. NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
2. nginx-deployment    3         3         3            3           18s
```

由结果可知，Deployment已创建三个Pod副本，并且所有副本都是最新的（它们包含最新的Pod模板）并且可用（Pod状态为Ready的持续时间至少得达到 `.spec.minReadySeconds` 字段定义的值）。

编者按：拓展阅读：<http://blog.csdn.net/WaltonWang/article/details/77461697>

要查看Deployment创建的ReplicaSet（`rs`），可运行 `kubectl get rs`：

```
1. NAME                                DESIRED   CURRENT   READY   AGE
2. nginx-deployment-2035384211         3         3         3       18s
```

请注意，ReplicaSet的名称的格式始终为 `[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]`。创建Deployment时会自动生成该hash。

要查看为每个Pod自动生成的label，请运行 `kubectl get pods --show-labels`。可返回类似以下输出：

```
1. NAME                                READY     STATUS    RESTARTS   AGE       LABELS
2. nginx-deployment-2035384211-7ci7o   1/1       Running   0          18s       app=nginx,pod-template-hash=2035384211
3. nginx-deployment-2035384211-kzszej  1/1       Running   0          18s       app=nginx,pod-template-hash=2035384211
4. nginx-deployment-2035384211-qqcnn   1/1       Running   0          18s       app=nginx,pod-template-hash=2035384211
```

ReplicaSet会确保在任何时候都有三个 `nginx` Pod运行。

注意：您必须在Deployment中指定适当的选择器和Pod模板标签（在本例中为 `app: nginx` ）。不要与其他Controller（包括其他Deployment和StatefulSet）所使用的标签或选择器重叠。Kubernetes不会阻止您重叠，如果多个Controller选择器发生重叠，那么这些Controller可能会出现冲突并且出现意外。

## Pod-template-hash标签

注意：请勿修改此标签。

`pod-template-hash label` 由Deployment Controller添加到其创建或采用的每个ReplicaSet上。

此标签可确保Deployment的子ReplicaSet不重叠。它通过将ReplicaSet的 `PodTemplate` 进行hash，并将生成的hash值作为标签，添加到ReplicaSet选择器、Pod模板标签、ReplicaSet所拥有的任何现有Pod中。

## 升级Deployment

注意：当且仅当Deployment的Pod模板（即 `.spec.template`）发生变化时，Deployment才会发生升级。例如，如果模板的标签或容器镜像被更新，则会触发Deployment的更新。其他更新，例如对Deployment伸缩，不会触发更新。

假设我们现在想要升级nginx Pod，让其使用 `nginx:1.9.1` 镜像，而非 `nginx:1.7.9` 镜像。

```
1. $ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
2. deployment "nginx-deployment" image updated
```

也可 `edit` Deployment，并将 `.spec.template.spec.containers[0].image` 从 `nginx:1.7.9` 改为 `nginx:1.9.1`：

```
1. $ kubectl edit deployment/nginx-deployment
2. deployment "nginx-deployment" edited
```

要想查看升级的状态，可运行：

```
1. $ kubectl rollout status deployment/nginx-deployment
2. Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
3. deployment "nginx-deployment" successfully rolled out
```

升级成功后，您可能希望 `get` Deployment：

```
1. $ kubectl get deployments
2. NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
3. nginx-deployment 3         3         3            3           36s
```

- up-to-date：表示Deployment已升级为最新配置的副本数量



- **current**: 表示此Deployment管理的副本总数
- **available**: 表示当前可用副本的数量。

运行 `kubectl get rs` 即可看到，Deployment通过创建一个新的ReplicaSet并将其扩展到3个副本，并将旧ReplicaSet减少到0个副本的方式更新Pod。

```
1. $ kubectl get rs
2. NAME                                DESIRED   CURRENT   READY   AGE
3. nginx-deployment-1564180365        3         3         3       6s
4. nginx-deployment-2035384211        0         0         0       36s
```

运行 `get pods` ，则只会显示新的Pod：

```
1. $ kubectl get pods
2. NAME                                READY     STATUS    RESTARTS   AGE
3. nginx-deployment-1564180365-khku8   1/1       Running   0           14s
4. nginx-deployment-1564180365-nacti   1/1       Running   0           14s
5. nginx-deployment-1564180365-z9gth   1/1       Running   0           14s
```

当下次我们要更新这些Pod时，只需再次更新Deployment的Pod模板。

Deployment可确保在升级时，只有一定数量的Pod会被关闭。默认情况下，它确保至少少要有1个期望数量的Pod运行（最多1个不可用）。

Deployment还可确保在期望数量的Pod上，只能创建一定数量的Pod。默认情况下，它确保最多有多于1个期望数量的Pod运行（最多1个波动）。

译者注：以上两段表示：如果期望的Pod数目是3，那么在升级的过程中，保持运行状态的Pod最小是2，最大是4。

未来，默认值将从1-1变为25% - 25%。

例如，如果您仔细观察上述Deployment，您将看到它首先创建了一个新Pod，然后删除一些旧Pod并创建新Pod。直到新Pod的数量已经足够，它才会杀死旧Pod；直到足够数量的老Pod被杀死才创建新Pod。它确保可用的Pod数量至少为2，并且Pod总数量至多为4。

```
1. $ kubectl describe deployments
2. Name:                               nginx-deployment
3. Namespace:                           default
4. CreationTimestamp: Tue, 15 Mar 2016 12:01:06 -0700
5. Labels:                               app=nginx
6. Annotations: deployment.kubernetes.io/revision=2
7. Selector:                             app=nginx
8. Replicas:                             3 desired | 3 updated | 3 total | 3 available | 0 unavailable
9. StrategyType:                         RollingUpdate
10. MinReadySeconds:                      0
11. RollingUpdateStrategy: 1 max unavailable, 1 max surge
12. Pod Template:
13.   Labels:                             app=nginx
14.   Containers:
15.     nginx:
```

```

16.   Image:           nginx:1.9.1
17.   Port:            80/TCP
18.   Environment:     <none>
19.   Mounts:           <none>
20.   Volumes:          <none>
21. Conditions:
22.   Type             Status Reason
23.   ----             -
24.   Available        True   MinimumReplicasAvailable
25.   Progressing      True   NewReplicaSetAvailable
26. OldReplicaSets:    <none>
27. NewReplicaSet:     nginx-deployment-1564180365 (3/3 replicas created)
28. Events:
29.   FirstSeen LastSeen   Count   From                                     SubobjectPath   Type       Reason
30.   -----
31.   36s        36s        1       {deployment-controller }               Normal       ScalingReplicaSet
32.   Scaled up replica set nginx-deployment-2035384211 to 3
33.   23s        23s        1       {deployment-controller }               Normal       ScalingReplicaSet
34.   Scaled up replica set nginx-deployment-1564180365 to 1
35.   23s        23s        1       {deployment-controller }               Normal       ScalingReplicaSet
36.   Scaled down replica set nginx-deployment-2035384211 to 2
37.   23s        23s        1       {deployment-controller }               Normal       ScalingReplicaSet
38.   Scaled up replica set nginx-deployment-1564180365 to 2
39.   21s        21s        1       {deployment-controller }               Normal       ScalingReplicaSet
40.   Scaled down replica set nginx-deployment-2035384211 to 0
41.   21s        21s        1       {deployment-controller }               Normal       ScalingReplicaSet
42.   Scaled up replica set nginx-deployment-1564180365 to 3

```

由上可知，当我们第一次创建Deployment时，它创建了一个ReplicaSet（nginx-deployment-2035384211），并直接将其扩容到3个副本。当我们升级Deployment时，它创建了一个新ReplicaSet（nginx-deployment-1564180365），并将其扩容到1，然后将旧ReplicaSet缩容到2。这样，在任何时候，至少有2个Pod可用，并且至多创建4个Pod。然后，继续按照相同的滚动更新策略对新旧ReplicaSet进行扩容/缩容。最后，新ReplicaSet中将会有3个可用副本，旧ReplicaSet副本缩小到0。

## Rollover（翻转）（也称为multiple updates in-flight[不停机多更新、多个升级并行]）

每次Deployment Controller观察到新的Deployment对象时，则会创建一个ReplicaSet来启动所期望的Pod（如果没有现有的ReplicaSet执行此操作）。现有ReplicaSet控制那些标签与 `.spec.selector` 匹配，但其模板不与 `.spec.template` 匹配的对象缩容。最终，新ReplicaSet将缩放到 `.spec.replicas`，所有旧ReplicaSets将被缩放到0。

如果在更新Deployment时，另一个更新正在进行中，那么Deployment就会为每个更新创建一个新的ReplicaSet，并开始扩容，并将滚动以前扩容的ReplicaSet——它会将其添加到旧ReplicaSet列表中，并开始缩容它。

例如，假设您创建一个Deployment，让它创建5个 `nginx:1.7.9` 副本，当仅3个 `nginx:1.7.9` 副本完成创建时，你开始更新Deployment，让它创建5个 `nginx:1.9.1` 副本。在这种情况下，Deployment将会立即开始杀死已创建的3个 `nginx:1.7.9` Pod，并将开始创建 `nginx:1.9.1` Pod。它不会等待5个副本的 `nginx:1.7.9` 都创建完成后才开始创建1.9.1的Pod。

## 标签选择器更新

通常不鼓励更新标签选择器，建议您预先规划好您的选择器。无论如何，如果您需要更新标签选择器，请务必谨慎，并确保您已经掌握了所有的含义。

注意：在API版本 `apps/v1beta2` 中，Deployment的标签选择器创建后不可变。

- 添加选择器要求Deployment规范中的Pod模板标签也更新为新标签，否则将会返回验证错误。此更改是不重叠的，这意味着新选择器不会选择使用旧选择器所创建的ReplicaSet和Pod，也就是说，所有旧版本的ReplicaSet都会被丢弃，并创建新ReplicaSet。
- 更新选择器——即，更改选择器中key中的现有value，会导致与添加相同的行为。
- 删除选择器——即从Deployment选择器中删除现有key——不需要对Pod模板标签进行任何更改。现有的ReplicaSet不会被孤立，也不会创建新的ReplicaSet，但请注意，删除的标签仍然存在于任何现有的Pod和ReplicaSet中。

## 回滚Deployment

有时您可能想要回滚Deployment；例如，当Deployment不稳定时，例如循环崩溃。默认情况下，所有Deployment的升级历史记录都保留在系统中，以便能随时回滚（可通过修改“版本历史记录限制”进行更改）。

注意：当Deployment的升级被触发时，会创建Deployment的修订版本。这意味着当且仅当更改Deployment的Pod模板（`.spec.template`）时，才会创建新版本，例如，模板的标签或容器镜像发生改变。其他更新（例如伸缩Deployment）不会创建Deployment修订版本，以便我们可以方便同时进行手动或自动缩放。这意味着，当您回滚到较早的版本时，对于一个Deployment，只有Pod模板部分会被回滚。

假设我们在更新Deployment时写错了字，将镜像名称写成了 `nginx:1.91` 而非 `nginx:1.9.1`：

```
1. $ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
2. deployment "nginx-deployment" image updated
```

升级将被卡住。

```
1. $ kubectl rollout status deployments nginx-deployment
2. Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

按下Ctrl-C，即可停止查阅上述状态。有关升级卡住的更多信息，请 [read more here](#)。

您还将看到旧副本（nginx-deployment-1564180365和nginx-deployment-2035384211）和新副本（nginx-deployment-3066724191）的数量都是2。

```
1. $ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1564180365	2	2	0	25s
nginx-deployment-2035384211	0	0	0	36s
nginx-deployment-3066724191	2	2	2	6s

查看创建的Pod，您将看到由新ReplicaSet创建的2个Pod卡在拉取镜像的过程中。

```

1. $ kubectl get pods
2. NAME                                READY   STATUS    RESTARTS   AGE
3. nginx-deployment-1564180365-70iae   1/1     Running   0           25s
4. nginx-deployment-1564180365-jbqqo   1/1     Running   0           25s
5. nginx-deployment-3066724191-08mng   0/1     ImagePullBackOff  0           6s
6. nginx-deployment-3066724191-eocby   0/1     ImagePullBackOff  0           6s

```

注意: Deployment Controller将自动停止不良的升级, 并将停止扩容新的ReplicaSet。这取决于您指定的rollingUpdate参数(具体为 `maxUnavailable`)。默认情况下, Kubernetes将maxUnavailable设为1, 而spec.replicas也为1, 因此, 如果您没有关注过这些参数设置, 则默认情况下, 您的Deployment可能100%不可用! 这将在未来版本的Kubernetes中修复。

```

1. $ kubectl describe deployment
2. Name:                               nginx-deployment
3. Namespace:                           default
4. CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
5. Labels:                               app=nginx
6. Selector:                             app=nginx
7. Replicas:                             2 updated | 3 total | 2 available | 2 unavailable
8. StrategyType:                         RollingUpdate
9. MinReadySeconds:                       0
10. RollingUpdateStrategy: 1 max unavailable, 1 max surge
11. OldReplicaSets:                       nginx-deployment-1564180365 (2/2 replicas created)
12. NewReplicaSet:                       nginx-deployment-3066724191 (2/2 replicas created)
13. Events:
14.   FirstSeen LastSeen   Count   From                                SubobjectPath   Type       Reason
15.   Message
16.   -----
16.   1m          1m          1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
17.   up replica set nginx-deployment-2035384211 to 3
17.   22s         22s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
18.   up replica set nginx-deployment-1564180365 to 1
18.   22s         22s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
19.   down replica set nginx-deployment-2035384211 to 2
19.   22s         22s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
20.   up replica set nginx-deployment-1564180365 to 2
20.   21s         21s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
21.   down replica set nginx-deployment-2035384211 to 0
21.   21s         21s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
22.   up replica set nginx-deployment-1564180365 to 3
22.   13s         13s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
23.   up replica set nginx-deployment-3066724191 to 1
23.   13s         13s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
24.   down replica set nginx-deployment-1564180365 to 2
24.   13s         13s         1       {deployment-controller }           Normal      ScalingReplicaSet Scaled
25.   up replica set nginx-deployment-3066724191 to 2

```

为解决以上问题, 我们需要回滚到以前稳定版本的Deployment。

## 检查Deployment的升级历史记录

首先, 检查此Deployment的修订版本:

```

1. $ kubectl rollout history deployment/nginx-deployment
2. deployments "nginx-deployment"
3. REVISION      CHANGE-CAUSE
4. 1             kubectl create -f docs/user-guide/nginx-deployment.yaml --record
5. 2             kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
6. 3             kubectl set image deployment/nginx-deployment nginx=nginx:1.91

```

因为我们在创建此Deployment时，使用 `--record` 记录了命令，所以我们能够轻松看到我们在每个版本中所做的更改。

要进一步查看每个版本的详细信息，请运行：

```

1. $ kubectl rollout history deployment/nginx-deployment --revision=2
2. deployments "nginx-deployment" revision 2
3.   Labels:      app=nginx
4.             pod-template-hash=1159050644
5.   Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
6.   Containers:
7.     nginx:
8.       Image:      nginx:1.9.1
9.       Port:       80/TCP
10.      QoS Tier:
11.        cpu:      BestEffort
12.        memory:   BestEffort
13.   Environment Variables:  <none>
14.   No volumes.

```

## 回滚到以前的版本

现在我们决定：撤消当前的升级并回滚到以前的版本：

```

1. $ kubectl rollout undo deployment/nginx-deployment
2. deployment "nginx-deployment" rolled back

```

或者，可通过 `--to-revision` 参数指定回滚到特定修订版本：

```

1. $ kubectl rollout undo deployment/nginx-deployment --to-revision=2
2. deployment "nginx-deployment" rolled back

```

有关回滚命令相关的信息，详见 `kubectl rollout` 。

现在，Deployment就会回滚到以前的稳定版本。如下可知，Deployment Controller会生成 `DeploymentRollback` 事件。

```

1. $ kubectl get deployment
2. NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
3. nginx-deployment    3         3         3            3           30m
4.

```

```

5. $ kubectl describe deployment
6. Name:          nginx-deployment
7. Namespace:     default
8. CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
9. Labels:        app=nginx
10. Selector:      app=nginx
11. Replicas:      3 updated | 3 total | 3 available | 0 unavailable
12. StrategyType:  RollingUpdate
13. MinReadySeconds: 0
14. RollingUpdateStrategy: 1 max unavailable, 1 max surge
15. OldReplicaSets: <none>
16. NewReplicaSet:   nginx-deployment-1564180365 (3/3 replicas created)
17. Events:
18.   FirstSeen LastSeen   Count   From              SubobjectPath  Type      Reason
19.   -----
20.   30m         30m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
21.   up replica set nginx-deployment-2035384211 to 3
22.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
23.   up replica set nginx-deployment-1564180365 to 1
24.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
25.   down replica set nginx-deployment-2035384211 to 2
26.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
27.   up replica set nginx-deployment-1564180365 to 2
28.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
29.   down replica set nginx-deployment-2035384211 to 0
30.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
31.   up replica set nginx-deployment-3066724191 to 2
32.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
33.   up replica set nginx-deployment-3066724191 to 1
34.   29m         29m         1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
35.   down replica set nginx-deployment-1564180365 to 2
36.   2m          2m          1       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
37.   down replica set nginx-deployment-3066724191 to 0
38.   2m          2m          1       {deployment-controller }      Normal    DeploymentRollback  Rolled
39.   back deployment "nginx-deployment" to revision 2
40.   29m         2m          2       {deployment-controller }      Normal    ScalingReplicaSet  Scaled
41.   up replica set nginx-deployment-1564180365 to 3

```

## 伸缩Deployment

可使用如下命令伸缩Deployment：

```

1. $ kubectl scale deployment nginx-deployment --replicas=10
2. deployment "nginx-deployment" scaled

```

假设您的群集启用了 [horizontal pod autoscaling](#) 功能，您可以为Deployment设置一个autoscaler，并根据现有Pod的CPU利用率，选择要运行的Pod的最小和最大个数。

```

1. $ kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80
2. deployment "nginx-deployment" autoscaled

```

## Proportional scaling (比例缩放)

RollingUpdate Deployment支持同时运行一个应用程序的多个版本。当您或autoscaler伸缩一个正处于升级中（正在进行或暂停）的RollingUpdate Deployment时，Deployment Controller就会平衡现有的、正在活动的ReplicaSet（ReplicaSet with Pod）中新增的副本，以减轻风险。这称为比例缩放。

例如，您正在运行一个具有10个副本的Deployment，`maxSurge=3`，`maxUnavailable=2`。

```
1. $ kubectl get deploy
2. NAME                                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
3. nginx-deployment                   10         10         10            10          50s
```

你更新到一个新的镜像，该镜像在集群内部无法找到。

```
1. $ kubectl set image deploy/nginx-deployment nginx=nginx:sometag
2. deployment "nginx-deployment" image updated
```

镜像使用ReplicaSet `nginx-deployment-1989198191`开始升级，但是由于上面设置了`maxUnavailable=2`，升级将被阻塞。

```
1. $ kubectl get rs
2. NAME                                DESIRED    CURRENT    READY    AGE
3. nginx-deployment-1989198191         5          5          0        9s
4. nginx-deployment-618515232         8          8          8        1m
```

然后，发起一个新的Deployment扩容请求。autoscaler将Deployment副本增加到15个。Deployment Controller需要决定在哪里添加这个5个新副本。如果我们不使用比例缩放，那么5个副本都将会被添加到新的ReplicaSet中。使用比例缩放，则新添加的副本将传播到所有ReplicaSet中。较大比例会被加入到有更多的副本的ReplicaSet，较低比例会被加入到较少的副本的ReplicaSet。剩余部分将添加到具有最多副本的ReplicaSet中。具有零个副本的ReplicaSet不会被扩容。

在上面的示例中，3个副本将被添加到旧ReplicaSet中，2个副本将添加到新ReplicaSet中。升级进程最终会将所有副本移动到新的ReplicaSet中，假设新副本变为健康状态。

```
1. $ kubectl get deploy
2. NAME                                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
3. nginx-deployment                   15         18         7             8           7m
4. $ kubectl get rs
5. NAME                                DESIRED    CURRENT    READY    AGE
6. nginx-deployment-1989198191         7          7          0        7m
7. nginx-deployment-618515232         11         11         11        7m
```

## 暂停与恢复Deployment

在触发一个或多个更新之前，您可以暂停Deployment，然后恢复。这将允许您在暂停和恢复之间应用多个补丁，而不会触发不必要的升级。

例如，使用刚刚创建的Deployment：

```
1. $ kubectl get deploy
2. NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
3. nginx         3         3         3            3           1m
4. $ kubectl get rs
5. NAME          DESIRED   CURRENT   READY   AGE
6. nginx-2142116321 3         3         3       1m
```

通过运行以下命令暂停：

```
1. $ kubectl rollout pause deployment/nginx-deployment
2. deployment "nginx-deployment" paused
```

然后升级Deployment的镜像：

```
1. $ kubectl set image deploy/nginx-deployment nginx=nginx:1.9.1
2. deployment "nginx-deployment" image updated
```

请注意，并不会产生新的ReplicaSet：

```
1. $ kubectl rollout history deploy/nginx-deployment
2. deployments "nginx"
3. REVISION  CHANGE-CAUSE
4. 1         <none>
5.
6. $ kubectl get rs
7. NAME          DESIRED   CURRENT   READY   AGE
8. nginx-2142116321 3         3         3       2m
```

您可以根据需要进行多次更新，例如，更新将要使用的资源：

```
1. $ kubectl set resources deployment nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi
2. deployment "nginx-deployment" resource requirements updated
```

在暂停之前，Deployment的初始状态将继续执行其功能；只要Deployment暂停，Deployment的更新将不会有任何影响。

最终，恢复Deployment并观察一个新的ReplicaSet提供了所有新的更新：

```
1. $ kubectl rollout resume deploy/nginx-deployment
2. deployment "nginx" resumed
3. $ kubectl get rs -w
4. NAME          DESIRED   CURRENT   READY   AGE
5. nginx-2142116321 2         2         2       2m
6. nginx-3926361531 2         2         0       6s
7. nginx-3926361531 2         2         1       18s
8. nginx-2142116321 1         2         2       2m
```



```

 9. nginx-2142116321 1      2      2      2m
10. nginx-3926361531 3      2      1      18s
11. nginx-3926361531 3      2      1      18s
12. nginx-2142116321 1      1      1      2m
13. nginx-3926361531 3      3      1      18s
14. nginx-3926361531 3      3      2      19s
15. nginx-2142116321 0      1      1      2m
16. nginx-2142116321 0      1      1      2m
17. nginx-2142116321 0      0      0      2m
18. nginx-3926361531 3      3      3      20s
19. ^C
20. $ kubectl get rs
21. NAME                DESIRED   CURRENT   READY   AGE
22. nginx-2142116321    0         0         0       2m
23. nginx-3926361531    3         3         3       28s

```

注意：恢复暂停的Deployment之前，您无法回滚。

## Deployment状态

Deployment在其生命周期中有各种状态。在升级新ReplicaSet时是 `progressing`，也可以是 `complete` 或 `fail to progress` 状态。

### Progressing Deployment（进行中的Deployment）

当执行以下任务之一时，Kubernetes将Deployment标记为`progressing`：

- Deployment创建一个新ReplicaSet。
- 该Deployment正在扩容其最新ReplicaSet。
- Deployment正在缩容其旧版ReplicaSet。
- 新Pod已准备就绪或可用（Ready状态至少持续了 `MinReadySeconds` 时间）。

您可使用 `kubectl rollout status` 监视部署的进度。

### Complete Deployment（完成的Deployment）

Deployment具有以下特点时候，Kubernetes将Deployment标记为`complete`：

- 与Deployment相关联的所有副本都已被更新为您所指定的最新版本，也就是说您所请求的任何更新都已完成。
- 与Deployment相关联的所有副本都可用。
- Deployment的旧副本都不运行。

可使用 `kubectl rollout status` 来检查Deployment是否已经完成。如果升级成功，则 `kubectl rollout status` 将会返回一个为零的退出代码。

```

1. $ kubectl rollout status deploy/nginx-deployment
2. Waiting for rollout to finish: 2 of 3 updated replicas are available...
3. deployment "nginx" successfully rolled out
4. $ echo $?

```

## Failed Deployment (失败的Deployment)

您的Deployment在尝试部署最新ReplicaSet的过程中可能会阻塞，永远也无法完成。这可能是由于以下一些因素造成的：

- 配额不足
- 就绪探针探测失败
- 镜像拉取错误
- 权限不足
- 范围限制
- 应用程序运行时配置错误

您可以检测到这种情况的一种方法，是在Deployment spec中指定一个期限参数：

( `spec.progressDeadlineSeconds` )。 `spec.progressDeadlineSeconds` 表示Deployment Controller等待多少秒后认为Deployment进程已停止。

以下 `kubectl` 命令使用 `progressDeadlineSeconds` 设置spec，使Controller在10分钟后缺少Deployment进度：

```
1. $ kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'
```

```
2. "nginx-deployment" patched
```

一旦超过截止时间，Deployment Controller就会将一个DeploymentCondition添加到Deployment的 `status.conditions` ，DeploymentCondition包含以下属性：

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

有关状态条件的更多信息，请参阅 [Kubernetes API conventions](#) 。

注意：除报告 `Reason=ProgressDeadlineExceeded` 以外，Kubernetes不会对停滞的Deployment采取任何行动。更高级别的协调者，可利用它并采取相应的行动，例如，将Deployment恢复到之前的版本。

注意：如果暂停Deployment，Kubernetes不会根据您的截止时间检查进度。您可以在升级过程中安全地暂停Deployment，然后再恢复，这样不会触发超过截止时间的条件。

您的Deployments可能会遇到短暂的错误，这可能是由于您设置的超时时间偏低，或者可能是由于可被视为“短暂”的其他类型的错误。例如，配额不足。如果您describe Deployment，将可看到以下部分的内容：

```
1. $ kubectl describe deployment nginx-deployment
```

```
2. <...>
```

```
3. Conditions:
```

Type	Status	Reason
Available	True	MinimumReplicasAvailable
Progressing	True	ReplicaSetUpdated

```

8.   ReplicaFailure   True     FailedCreate
9.   <...>

```

如果运行 `kubectl get deployment nginx-deployment -o yaml`，则Deployment状态可能如下所示：

```

1. status:
2.   availableReplicas: 2
3.   conditions:
4.     - lastTransitionTime: 2016-10-04T12:25:39Z
5.       lastUpdateTime: 2016-10-04T12:25:39Z
6.       message: Replica set "nginx-deployment-4262182780" is progressing.
7.       reason: ReplicaSetUpdated
8.       status: "True"
9.       type: Progressing
10.    - lastTransitionTime: 2016-10-04T12:25:42Z
11.      lastUpdateTime: 2016-10-04T12:25:42Z
12.      message: Deployment has minimum availability.
13.      reason: MinimumReplicasAvailable
14.      status: "True"
15.      type: Available
16.    - lastTransitionTime: 2016-10-04T12:25:39Z
17.      lastUpdateTime: 2016-10-04T12:25:39Z
18.      message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded quota:
19.        object-counts, requested: pods=1, used: pods=3, limited: pods=2'
20.      reason: FailedCreate
21.      status: "True"
22.      type: ReplicaFailure
23.   observedGeneration: 3
24.   replicas: 2
25.   unavailableReplicas: 2

```

最终，一旦Deployment进度超过了截止时间，Kubernetes将会更新状态以及导致Progressing的原因：

```

1. Conditions:
2.   Type           Status   Reason
3.   ----           -
4.   Available       True     MinimumReplicasAvailable
5.   Progressing     False    ProgressDeadlineExceeded
6.   ReplicaFailure  True     FailedCreate

```

可扩容Deployment、扩容正在运行的其他Controller或通过增加namespace中的配额，从而解决配额不足的问题。当您满足配额条件后，Deployment Controller就会完成升级，您将看到Deployment的状态更新为成功（`Status=True` 和 `Reason=NewReplicaSetAvailable`）。

```

1. Conditions:
2.   Type           Status   Reason
3.   ----           -
4.   Available       True     MinimumReplicasAvailable
5.   Progressing     True     NewReplicaSetAvailable

```

`Type=Available`，并且 `Status=True` 表示您的Deployment具有最低可用性。最低可用性由部署策略中指定的参数决定。`Type=Progressing`，并且 `Status=True` 表示您的Deployment正在升级中；正处于progressing状态；抑或已成功完成其进度，并且达到所需的最小可用新副本个数（请查看特定状态的原因——本例中，`Reason=NewReplicaSetAvailable` 意味着Deployment完成）。

可使用 `kubectl rollout status` 来检查Deployment进程是否失败。如果Deployment已超过截止之间，则 `kubectl rollout status` 将会返回非零的退出代码。

```
1. $ kubectl rollout status deploy/nginx-deployment
2. Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
3. error: deployment "nginx" exceeded its progress deadline
4. $ echo $?
5. 1
```

## 操作失败的Deployment

应用于完成的Deployment的所有操作也适用于失败的Deployment。如果需要在Deployment的Pod模板中应用多个调整，您可以进行扩容/缩容、回滚到先前的版本，甚至是暂停。

## 清理策略

可在Deployment中设置 `.spec.revisionHistoryLimit` 字段，从而指定该Deployment要保留多少个旧版本ReplicaSet。其余的将在后台垃圾收集。默认情况下，所有修订历史都将被保留。在将来的版本中，默认是2。

注意：明确将此字段设置为0将会导致清除Deployment的所有历史记录，这会导致Deployment无法回滚。

## 用例

### Canary Deployment（金丝雀部署）

如果要使用Deployment向一部分用户或服务器发布版本，则可以按照 [managing resources](#) 描述的金丝雀模式，创建多个Deployment，每个Deployment对应各自的版本。

## 编写Deployment Spec

与所有其他Kubernetes配置一样，Deployment需要 `apiVersion`、`kind` 和 `metadata` 等字段。有关使用配置文件的一般信息，请参阅 [deploying applications](#)，配置容器以及 [using kubectl to manage resources](#) 文档。

Deployment还需要一个 `.spec` section。

## Pod Template

`.spec.template` 是 `.spec` 唯一必需的字段。

`.spec.template` 是一个 `pod template` 。它与 `Pod` 有完全相同的schema，除了它是嵌套的，并且没有 `apiVersion` 或 `kind` 。

除了Pod必需的字段之外，Deployment中的Pod模板必须指定适当的标签和重启策略。对于标签，请确保不与其他Controller重叠。详见 `selector` )。

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

## 副本

`.spec.replicas` 是一个可选字段，用于指定期望的 `.spec.replicas` 的数量，默认为1。

## 选择器

`.spec.selector` 是一个可选字段，指定此Deployment所关联的Pod的 `label selector` 。

`.spec.selector` 必须与 `.spec.template.metadata.labels` 相匹配，否则将被API拒绝。

在API版本 `apps/v1beta2` 中，如果未经设置，则 `.spec.selector` 和 `.metadata.labels` 不再默认与 `.spec.template.metadata.labels` 相同。 所以，必须明确设定这些字段。 另请注意，在 `apps/v1beta2` 中， `.spec.selector` 在Deployment创建后是不可变的。

对于模板与 `.spec.template` 不同，抑或副本总数超过 `.spec.replicas` 定义的Pod，Deployment可能会终止这些Pod。如果Pod的数量小于所需的数量，它将会使用 `.spec.template` 的定义启动新Pod。

译者按：“模板与 `.spec.template` 不同”的场景：Deployment先创建，然后修改YAML定义文件，升级镜像的版本。此时，旧Pod的镜像字段就与 `.spec.template` 不同了

注意：你不应该建立其他与该选择器相匹配的Pod，无论是直接创建，还是通过另一个Deployment创建，抑或通过另一个Controller创建（例如ReplicaSet或ReplicationController）。如果这样做，第一个Deployment将会认为是它创造了这些Pod。Kubernetes并不会阻止你这么 做。

如果多个Controller的选择器发生重叠，Controller会发生冲突，并导致不正常的行为。

## 策略

`.spec.strategy` 指定使用新Pod代替旧Pod的策略。 `.spec.strategy.type` 可有“Recreate”或“RollingUpdate” 两种取值。默认为“RollingUpdate”。

### Recreate Deployment

当 `.spec.strategy.type==Recreate` 时，创建新Pod前，会先杀死所有现有的Pod。

### Rolling Update Deployment（滚动更新Deployment）

当 `.spec.strategy.type==RollingUpdate` 时，Deployment以 `rolling update` 的方式更新Pod。可指定 `maxUnavailable` 和 `maxSurge` 控制滚动更新的过程。

Max Unavailable

`.spec.strategy.rollingUpdate.maxUnavailable` 是一个可选字段，用于指定在更新的过程中，不可用Pod的最大数量。该值可以是一个绝对值（例如5），也可以是期望Pod数量的百分比（例如10%）。通过百分比计算出来的绝对值会向下取整。如果 `.spec.strategy.rollingUpdate.maxSurge` 是0，那么该值不能为0。默认值是25%。

例如，此值被设置为30%，当滚动更新开始时，旧ReplicaSet会立即缩容到期望Pod数量的70%。一旦新的Pod进入Ready状态，老ReplicaSet可进一步缩容，然后扩容新ReplicaSet，确保在更新过程中，任何时候都会有至少70%的可用Pod。

### Max Surge

`.spec.strategy.rollingUpdate.maxSurge` 是一个可选字段，用于指定在更新的过程中，超过Pod期望数量的最大数量。该值可以是一个绝对值（例如5），也可以是期望Pod数量的百分比（例如10%）。如果 `MaxUnavailable` 为0，该值不能为0。通过百分比计算出来的绝对值会绝对会向上取整。默认值是25%。

例如，此值被设置为30%，当滚动更新开始时，新ReplicaSet会立即扩容，新旧Pod的总数不超过期望Pod数量的130%。一旦老Pod已被杀死，新ReplicaSet可进一步扩容，确保在更新过程中，任何时候运行的Pod总数不超过期望Pod数量的130%。

## Progress Deadline Seconds

`.spec.progressDeadlineSeconds` 是一个可选字段，用于指定表示Deployment Controller等待多少秒后认为Deployment进程已 **failed progressing** —表现为在资源状态中有：`Type=Progressing`、`Status=False`，以及 `Reason=ProgressDeadlineExceeded`。Deployment Controller将继续重试该Deployment。在未来，一旦实现自动回滚，Deployment Controller观察到这种状况时，就会尽快回滚 Deployment。

如需设置本字段，值必须大于 `.spec.minReadySeconds`。

## Min Ready Seconds

`.spec.minReadySeconds` 是一个可选字段，用于指定新创建的Pod进入Ready状态（Pod的容器持续多少秒不发生崩溃，就被认为可用）的最小秒数。默认为0（Pod在Ready后就会被认为是可用状态）。要了解什么时候Pod会被认为已Ready，详见 [Container Probes](#)。

## Rollback To

在API版本 `extensions/v1beta1` 和 `apps/v1beta1` 中，`.spec.rollbackTo` 已被弃用，并且在API版本 `apps/v1beta2` 中不再支持该字段。取而代之的是，建议使用 `kubectl rollout undo`，详见 [Rolling Back to a Previous Revision](#)。

## Revision History Limit（修订历史限制）

Deployment的修订历史记录存储在它所控制的ReplicaSet内。

`.spec.revisionHistoryLimit` 是一个可选字段，用于指定要保留的ReplicaSet数量，以便回滚。它的理想取值取决于新Deployment的频率和稳定性。默认情况下，所有老ReplicaSet都被保存，将资源存储在 `etcd` 中，使用 `kubectl get rs` 查询ReplicaSet信息。每个Deployment修订的配置都被存储在其ReplicaSet；因此，一旦旧ReplicaSet被删除，Deployment将无法回滚到那个修订版本。

更具体地讲，将该字段设为零，意味着所有0副本的旧ReplicaSet将被清理。在这种情况下，一个新的Deployment无法回滚，因为其修订历史都被清除了。

## Paused

`.spec.paused` 是一个可选的布尔类型的字段，用于暂停和恢复Deployment。Deployment暂停和未暂停之间唯一的区别是：对暂停Deployment的PodTemplateSpec所做的任何更改，不会触发新的升级。当Deployment创建后，默认情况下不会暂停。

## Deployment的替代方案

---

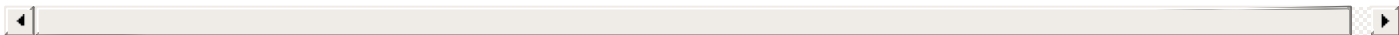
### kubectl滚动更新

`kubectl rolling update` 以类似的方式更新Pod和ReplicationControllers。但是建议使用Deployment，因为是声明式、服务器端的，并有额外的功能，例如滚动更新完成后可回滚到历史版本。

## 原文

---

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>





# StatefulSet

StatefulSet有两个维度的抽象：

- 拓扑状态：有时候多个实例之间并非对等关系，可能需要按顺序启动，又或者，某个Pod挂掉后，新创建的Pod必须和原先Pod的网络标识一样，这样，原先的访问者才可能使用访问原先Pod的方法，访问到新Pod。
- 存储状态：即使Pod挂掉被重建，访问的存储应该是同一份。
- 本质：StatefulSet是一种特殊的Deployment，只不过这个Deployment的Pod名称+编号固定，编号的顺序固定了Pod之间的拓扑关系；而编号对应的Persistent Volume，绑定了Pod与持久化存储之间的关系。因此，Pod即使被重建，状态也不会改变。

**StatefulSet**是用于管理有状态应用的工作负载API对象。**StatefulSet**在1.8中是beta版本。

管理Deployment以及一组 **Pods** 的伸缩，并为这些Pod的排序和唯一性提供保证。

像 **Deployment** 一样，StatefulSet管理基于相同容器spec的Pod。与Deployment不同，StatefulSet会为每个Pod保留粘性身份。这些Pod是从相同的spec创建的，但不可互换：每个Pod都有一个持久化的标识符，即使重新调度也仍然会使用该标识符。

StatefulSet与任何其他Controller的模式相同。您可在StatefulSet对象中定义期望状态，StatefulSet Controller会从当前状态进行必要的更新，从而达到期望状态。

## Using StatefulSets (使用StatefulSet)

StatefulSet适用于以下场景：

- 稳定、唯一的网络标识。
- 稳定、持久化的存储。
- 有序、优雅的部署和缩放（即：部署有顺序、扩展也有顺序）。
- 有序、优雅的删除和终止。
- 有序、自动的滚动更新。

在上文中，“稳定”是Pod调度（重新调度）持久化的代名词。如果应用程序不需要任何稳定的标识符或有序部署、删除或缩放，则应该使用提供无状态副本的Controller部署应用。 诸如 **Deployment** 或者 **ReplicaSet** 可能更适合您的无状态需求。

## Limitations (限制)

- StatefulSet是一个beta资源（处于Beta阶段的资源），在1.5之前的Kubernetes版本中不可用。
- 与所有其他alpha/beta资源一样，可传给apiserver一个 `--runtime-config` 选项来禁用StatefulSet。
- 给定Pod的存储必须由 **PersistentVolume Provisioner** 根据请求的 `storage class` 提供，或由管理员预先设置。
- 删除/缩容StatefulSet将不会删除与StatefulSet关联的Volume。这样做是为了确保数据安全性，这通常比自动清除StatefulSet所有相关资源更有价值。
- StatefulSet目前需要一个 **Headless Service** 负责Pod的网络身份。您有责任创建此Service。

## Components (组件)



下面的示例演示了StatefulSet的组件。

- 名为nginx的Headless Service，用于控制网络域名。
- 名为web的StatefulSet，它有一个Spec，表示有3个nginx副本。
- volumeClaimTemplates将使用PersistentVolume Provisioner提供的 [PersistentVolumes](#)，从而提供稳定的存储。

```

1. kind: PersistentVolume
2. apiVersion: v1
3. metadata:
4.   name: pv-volume
5.   labels:
6.     type: local
7. spec:
8.   storageClassName: manual
9.   capacity:
10.    storage: 10Gi
11.   accessModes:
12.    - ReadWriteOnce
13.   hostPath:
14.    path: "/mnt/data"
15. ---
16. kind: PersistentVolumeClaim
17. apiVersion: v1
18. metadata:
19.   name: pv-claim
20. spec:
21.   storageClassName: manual
22.   accessModes:
23.    - ReadWriteOnce
24.   resources:
25.    requests:
26.     storage: 3Gi
27. ---
28. apiVersion: v1
29. kind: Service
30. metadata:
31.   name: nginx
32.   labels:
33.    app: nginx
34. spec:
35.   ports:
36.    - port: 80
37.     name: web
38.   clusterIP: None
39.   selector:
40.    app: nginx
41. ---
42. apiVersion: apps/v1
43. kind: StatefulSet
44. metadata:
45.   name: web

```

```

46. spec:
47.   selector:
48.     matchLabels:
49.       app: nginx # has to match .spec.template.metadata.labels
50.   serviceName: "nginx"
51.   replicas: 3 # by default is 1
52.   template:
53.     metadata:
54.       labels:
55.         app: nginx # has to match .spec.selector.matchLabels
56.     spec:
57.       terminationGracePeriodSeconds: 10
58.       containers:
59.       - name: nginx
60.         image: nginx
61.         ports:
62.         - containerPort: 80
63.           name: web
64.         volumeMounts:
65.         - name: task-pv-storage
66.           mountPath: /usr/share/nginx/html
67.       volumes:
68.       - name: task-pv-storage
69.         persistentVolumeClaim:
70.           claimName: task-pv-claim

```

## Pod Selector (Pod选择器)

您必须设置StatefulSet的 `spec.selector` 字段以匹配其 `.spec.template.metadata.labels` 的Label。在Kubernetes 1.8之前，`spec.selector` 字段被默认为省略。在1.8及更高版本中，如未指定匹配的Pod Selector将会导致在StatefulSet创建过程中验证错误。

## Pod Identity (Pod身份)

StatefulSet Pod有唯一的身份，包括序数 (ordinal)、稳定的网络标识和稳定的存储。身份会绑定到Pod (具有粘性)，不管Pod被调度到哪个Node上。

## Ordinal Index (有序的索引)

对于一个有N个副本的StatefulSet，StatefulSet中的每个Pod将被分配一个整数序号，范围[0, N)，并且唯一。

## Stable Network ID (稳定的网络ID)

StatefulSet中的每个Pod，从StatefulSet的名称和Pod的序数派生其主机名。构造的主机名的模式是

`$(statefulset name)-$(ordinal)`。上面的例子中，将会创建名为 `web-0,web-1,web-2` 的Pod。StatefulSet可以使用 [Headless Service](#) 来控制其Pod的域名。此Service管理的域名的格式为：`$(service name).$(namespace).svc.cluster.local`，其中“cluster.local”是集群域名。在创建每个Pod时，它将会获得一个匹

配的DNS子域，采用以下形式：`$(podname).$(governing service domain)`，其中governing service由StatefulSet上的 `serviceName` 字段定义。

以下是Cluster Domain、Service名称、StatefulSet名称以及如何影响StatefulSet的Pod的DNS名称的一些示例。

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-1}.r
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-1}.r
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-1}.r

请注意，除非 `otherwise configured`，Cluster Domain将被设为 `cluster.local`。

## Stable Storage (稳定的存储)

Kubernetes为每个VolumeClaimTemplate创建一个 `PersistentVolume`。在上面的nginx示例中，每个Pod将收到一个PersistentVolume，其中包含名为 `my-storage-class` 的StorageClass和1 Gib的存储。如果未指定StorageClass，则将使用默认StorageClass。当一个Pod被重新调度到一个Node上时，它的 `volumeMounts` 将挂载与其PersistentVolume Claims相关联的PersistentVolume。请注意，当Pod或StatefulSet被删除时，与Pod的PersistentVolume Claims相关联的PersistentVolumes不会被删除。这必须手动完成。

## Deployment and Scaling Guarantees (部署和缩放保证)

- 对于具有N个副本的StatefulSet，当部署Pod时，它们将按从{0..N-1}的顺序创建。
- 当Pod被删除时，它们按从{N-1..0}的顺序终止。
- 在将扩容操作应用于Pod之前，它的所有“前辈”必须Running and Ready。
- 在Pod终止之前，所有的“后辈”必须完全关闭。

StatefulSet不应该将 `pod.Spec.TerminationGracePeriodSeconds` 设为0。这种做法是不安全，强烈不建议您这么做。有关进一步说明，请参阅 `force deleting StatefulSet Pods`。

当创建上述的nginx示例时，将以web-0、web-1、web-2的顺序部署三个Pod。在web-0正在 `Running and Ready`，web-1将不被部署，而在web-1 Running and Ready之前web-2将不被部署。如果web-0失败，在web-1 Running and Ready之后，但在启动web-2之前，web-2将不会启动，直到web-0成功重启并Running and Ready。

如果用户通过patch StatefulSet来scale部署的示例，例如设置 `replicas=1`，则web-2将首先被终止。在web-2完全关闭和删除之前，web-1不会被终止。如果web-0失败发生在web-2终止并完全关闭之后、web-1终止之前，web-1将不会终止，除非web-0已经Running and Ready。

## Pod Management Policies (Pod管理策略)

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

## OrderedReady Pod Management (OrderedReady的Pod管理)

`OrderedReady` Pod管理是StatefulSet的默认值。它实现了[上述](#) 行为。

## Parallel Pod Management (并行 Pod管理)

`Parallel` Pod管理告诉StatefulSet Controller 并行启动或终止所有Pod，并且不要等待Pod在启动或终止另一个Pod之前变为“Running”和“Ready”或完全终止。

## Update Strategies (更新策略)

在Kubernetes 1.7及更高版本中，StatefulSet的 `.spec.updateStrategy` 字段允许您配置和禁用StatefulSet中Pod的容器、标签、资源最小要求/最大限制、Annotation的滚动更新。

### On Delete

`OnDelete` 更新策略实现了遗留（1.6和以前）的行为。当 `spec.updateStrategy` 未指定时，这是默认策略。当StatefulSet的 `.spec.updateStrategy.type` 设置为 `OnDelete`，StatefulSet Controller将不会自动更新StatefulSet中的Pod。用户必须手动删除Pod以使Controller创建新的Pod，以反映对StatefulSet的 `.spec.template` 进行的修改。

### Rolling Updates

`RollingUpdate` 更新策略为在StatefulSet中的Pod实现自动的滚动更新。当StatefulSet的 `.spec.updateStrategy.type` 设置为 `RollingUpdate` 时，StatefulSet Controller将在StatefulSet中删除并重新创建每个Pod。它将按照Pod终止（从最大序号到最小）的顺序进行，每次更新一个Pod。在更新其“前辈”之前，它将等待正在更新的Pod进入Running and Ready状态。

### Partitions (分区)

可以通过指定 `.spec.updateStrategy.rollingUpdate.partition` 来对 `RollingUpdate` 更新策略进行分区。如果指定了分区，则当StatefulSet的 `.spec.template` 更新时，具有大于或等于分区的序数的所有 `.spec.template` 被更新。具有小于分区的序数的所有Pods将不会更新，即使删除它们，也会使用以前的版本重新创建。如果StatefulSet的 `.spec.updateStrategy.rollingUpdate.partition` 大于其 `.spec.replicas`，则 `.spec.replicas` 更新将不会传播到其Pod。在大多数情况下，您不需要使用分区，但如果您想要进行阶段更新、金丝雀部署或执行分阶段发布，分区将非常有用。

## What's next (下一步)

- Follow an example of [deploying a stateful application](#).
- Follow an example of [deploying Cassandra with Stateful Sets](#).

## 原文

---

<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>

## 很赞的博客

---

在K8s中创建StatefulSet: <http://www.cnblogs.com/puyangsky/p/6677308.html>



# Daemon Set

## What is a DaemonSet? (什么是DaemonSet?)

*DaemonSet* 确保所有（或一些）Node 运行 Pod 的副本。随着 Node 被添加到集群，Pod 被添加到 Node 上。随着从集群中删除 Node，这些 Pods 被垃圾回收。删除 *DaemonSet* 将清理它创建的 Pod。

*DaemonSet* 的典型用途是：

- 在每个 Node 上运行集群存储 daemon，例如在每个 Node 上运行 `glusterd` 、 `ceph` 。
- 在每个 Node 上运行日志收集 daemon，例如 `fluentd` 或 `logstash` 。
- 在每个 Node 上运行 Node 监控 daemon，如 `Prometheus Node Exporter` 、 `collectd` 、 `Datadog Agent` ， `New Relic Agent` 或 `Ganglia` `gmond` 。

在简单的情况下，一个 *DaemonSet*，覆盖所有 Node，将用于每种类型的 daemon。更复杂的设置可能对单一类型的 daemon 使用多个 *DaemonSet*，但对不同硬件类型使用不同标志和/或不同内存和 cpu 最小需求。

## Writing a DaemonSet Spec (编写DaemonSet Spec)

### Create a DaemonSet (创建DaemonSet)

您可以在 YAML 文件中描述 *DaemonSet*。例如，下面的 `daemonset.yaml` 文件描述了运行 `fluentd-elasticsearch` Docker 镜像的 *DaemonSet*：

```

1. apiVersion: apps/v1beta2
2. kind: DaemonSet
3. metadata:
4.   name: fluentd-elasticsearch
5.   namespace: kube-system
6.   labels:
7.     k8s-app: fluentd-logging
8. spec:
9.   selector:
10.    matchLabels:
11.      name: fluentd-elasticsearch
12.   template:
13.     metadata:
14.       labels:
15.         name: fluentd-elasticsearch
16.     spec:
17.       containers:
18.         - name: fluentd-elasticsearch
19.           image: gcr.io/google-containers/fluentd-elasticsearch:1.20
20.           resources:
21.             limits:
22.               memory: 200Mi

```

```

23.         requests:
24.             cpu: 100m
25.             memory: 200Mi
26.         volumeMounts:
27.             - name: varlog
28.               mountPath: /var/log
29.             - name: varlibdockercontainers
30.               mountPath: /var/lib/docker/containers
31.               readOnly: true
32.         terminationGracePeriodSeconds: 30
33.         volumes:
34.             - name: varlog
35.               hostPath:
36.                 path: /var/log
37.             - name: varlibdockercontainers
38.               hostPath:
39.                 path: /var/lib/docker/containers

```

- 基于YAML文件创建DaemonSet: `kubectl create -f daemonset.yaml`

## Required Fields (必填字段)

与所有其他Kubernetes配置一样，DaemonSet需要 `apiVersion`，`kind` 和 `metadata` 字段。有关使用配置文件的信息，请参阅 [deploying applications](#)、[configuring containers](#) 以及 [working with resources](#)。

DemonSet还需要一个 `.spec` 部分。

## Pod Template (Pod模板)

`.spec.template` 是 `.spec` 中必需的字段之一。

`.spec.template` 是一个 `pod template`。它与 `Pod` 具有完全相同的模式，只不过它是嵌套的，没有 `apiVersion` 或 `kind`。

除Pod的必需字段外，DaemonSet中的Pod模板必须指定适当的标签（请参阅 [pod selector](#)）。

DemonSet中的Pod模板必须具有等于 `Always` 的 `RestartPolicy`，也可留空，默认为 `Always`。

## Pod Selector (Pod选择器)

`.spec.selector` 字段是一个pod选择器。它的作用与 `Job` 的 `.spec.selector` 相同。

从Kubernetes 1.8起，您必须指定与 `.spec.template` 的Label相匹配的Pod选择器。如果该字段留空，Pod Selector将不再有默认值。Selector默认与 `kubectl apply` 不兼容。另外，一旦DaemonSet被创建，它的 `spec.selector` 就不可变。改变Pod Selector可能会导致Pod的孤立，从而引起用户的困惑。

`spec.selector` 是由两个字段组成的对象：

- `matchLabels` - 与 `ReplicationController` 的 `.spec.selector` 相同。

- `matchExpressions` - 允许通过指定key列表、value列表、以及与key/value相关联的运算符的方式，来构建更复杂的选择器。

当同时指定以上两个字段时，两者之间AND关系。

如果指定了 `.spec.selector`，它必须与 `.spec.template.metadata.labels` 匹配。如果未指定，则默认为相等。不匹配的配置将会被API拒绝。

此外，您通常不应创建其标签与此选择器相匹配的任何Pod，不管是通过另一个DaemonSet还是通过其他Controller（如ReplicaSet）进行匹配。否则，DaemonSet Controller会认为那些Pod是由它创建的。Kubernetes不会阻止你这样做。您可能想要执行此操作的一种情况，是手动在一个Node上创建具有不同值的Pod进行测试。

如果您尝试创建一个DaemonSet，那么

```
// TODO
```

## Running Pods on Only Some Nodes ( 只在一些Node上运行Pod )

如果指定了 `.spec.template.spec.nodeSelector`，则DaemonSet Controller将在与该 `node selector` 匹配的Node上创建Pod。同样，如果您指定了 `.spec.template.spec.affinity`，则DaemonSet控制器将在与该 `node affinity` 匹配的Node上创建Pod。如果不指定，则DaemonSet Controller将会在所有Node上创建Pod。

译者按：拓展阅读：《 k8s nodeSelector&affinity 》：<http://blog.csdn.net/yevvzi/article/details/54585686>

## How Daemon Pods are Scheduled ( 如何调度Daemon Pods )

通常，Pod在哪台机器上运行是由Kubernetes Scheduler选择的。但是，由DaemonSet Controller创建的Pod已经事先确定在哪台机器上运行（在创建Pod时指定了 `.spec.nodeName`，因此Scheduler将忽略调度这些Pod）。因此：

- Node的 `unschedulable` 字段不受DaemonSet Controller的约束。
- 即使Scheduler尚未启动，DaemonSet Controller也可以创建Pod，这样设计对集群启动是有帮助的。

Daemon Pod 关注所设置的 `taints and tolerations` 规则，但它们会为如下未指定 `tolerationSeconds` 的 taint创建 `NoExecute` toleration。

- `node.kubernetes.io/not-ready`
- `node.alpha.kubernetes.io/unreachable`

这样可以确保在启用 `TaintBasedEvictions` alpha功能时，当Node出现问题（如网络分区）时，它们将不被驱逐。（当 `TaintBasedEvictions` 功能未启用时，它们在这些情况下也不会被驱逐，但会NodeController的硬编码行为而被驱逐，而会因为toleration被驱逐）。

它们也tolerate以下的 `NoSchedule` taints：



- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`

当启用对critical Pod的支持时，并且DaemonSet中的Pod被标记为critical时，将会创建Daemon pod，Daemon Pod会为 `node.kubernetes.io/out-of-disk` 的taint创建额外 `NoSchedule` 的toleration。

请注意，如果启用了Alpha功能 `TaintNodesByCondition`，上述 `NoSchedule` Taints仅在1.8或更高版本中创建。

## Communicating with Daemon Pods (与Daemon Pod通信)

与DaemonSet中的Pod通信的方式有：

- **Push**：配置DaemonSet中的Pod，将更新发送到其他Service服务，例如统计数据库。它们没有客户端。
- **NodeIP and Known Port**：DaemonSet中的Pod可以使用 `hostPort`，以便通过Node的IP访问Pod。客户能通过某种方式知道Node IP的列表，并知道端口。
- **DNS**：使用相同的Pod选择器创建 `headless service`，然后使用 `endpoints` 资源或从DNS查询多个A记录发现DaemonSet。
- **Service**：使用相同的Pod选择器创建Service，并使用该Service来访问随机Node上的daemon。（无法到达特定Node）

## Updating a DaemonSet (更新DaemonSet)

如果Node的Label被更改，DaemonSet会立即将Pods添加到新匹配的Node，并从不匹配Node删除Pod。

您可以修改DaemonSet创建的Pod。但是，Pod不允许更新所有字段。此外，在下次创建一个Node（即使使用相同的名称）时，DaemonSet Controller依然会使用原始模板。

您可以删除DaemonSet。如果使用 `kubectl` 指定 `--cascade=false`，那么Pod将保留在Node上。然后，您可以使用不同的模板创建一个新的DaemonSet。新DaemonSet将识别所有具有匹配的标签的、已存在的Pod。尽管Pod模板并不匹配，但它不会修改或删除这些Pod。您将需要通过删除Pod或删除Node的方式，强制创建新的Pod。

在Kubernetes 1.6及更高版本中，您可以在DaemonSet上 `perform a rolling update`。

未来的Kubernetes版本将支持Node的受控更新。

## Alternatives to DaemonSet (DaemonSet的替代方案)

### Init Scripts (初始化脚本)

通过在Node上直接启动它们（例如使用 `init`、`upstartd` 或 `systemd`），可以运行daemon进程。这是非常好的。但是，通过DaemonSet运行这些进程有几个优点：

- 能够监控和管理daemon进程的日志，就像应用程序一样。
- 相同的配置语言和工具（如Pod模板，`kubectl`）用于daemon和应用程序。

- 未来版本的Kubernetes可能支持 DaemonSet创建的Pod 与 Node升级工作流 之间的集成。
- 在具有资源限制的容器中运行daemon会增强应用容器和daemon隔离性。 当然，也可通过在容器中运行而不在Pod中运行daemon的方式，实现该目标（例如，直接通过Docker启动）。

## Bare Pods（裸Pod）

可以直接创建Pods，指定Pod运行在特定的Node上。但是，DaemonSet会替换由于任何原因而被删除或终止的Pod，例如在Node故障或Node被中断维护（例如内核升级）的情况下。因此，您应该使用DaemonSet而非单独创建Pod。

## Static Pods（静态Pod）

可通过将文件写入由Kubelet监视的某个目录来的方式来创建Pods。 这些被称为 `static pods` 。 与DaemonSet不同，静态Pod无法使用kubectl或其他Kubernetes API客户端进行管理。静态Pod不依赖于apiserver，这使其在集群启动的情况下很有用。 此外，静态Pod可能在将来会被废弃。

## Deployments

DaemonSets类似于 `Deployments` ，因为它们都创建了Pods，并且这些Pod都有不期望终止的进程（例如web服务器，存储服务器）。

为无状态服务使用Deployment，例如前端，缩放副本数量和滚动升级比控制Pod运行在哪个主机上要重要得多。 当一个Pod副本始终在所有或某些主机上运行，并需要先于其他Pod启动时，可使用DaemonSet。

## 原文

---

<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

# Configuration Best Practices

本文档强调并整合了整个用户指南、入门文档和示例中引入的配置最佳实践。

这是一个“活”的文件。如果你想到的东西不在这个名单上，但可能对他人有用，请不要犹豫，提交issue或提交PR (pull request)。

## General Config Tips (常规配置Tips)

- 定义配置时，指定最新的稳定API版本（当前为v1）。
- 配置文件应该被推送到集群之前，应存储在版本控制软件中。这样，如果需要，可以快速回滚配置。如果需要，还可以帮助集群重新创建和恢复。
- 使用YAML而非JSON编写配置文件。尽管这些格式在几乎所有情况下可以互换使用，但YAML往往对用户更加友好。
- 只要有意义，将相关对象组合成单个文件。一个文件通常比多个文件更容易管理。请参阅 [guestbook-all-in-one.yaml](#) 文件作为此语法的示例。

还要注意，可以在目录上调用许多 `kubectl` 命令，因此您还可以在配置文件所在目录中调用 `kubectl create`。请参阅下面的更多细节。

- 不要指定不必要的默认值 - 简单和最小的配置将减少错误。
- 将对象描述放在Annotation中。

## “Naked” Pods vs Replication Controllers and Jobs (“裸”Pod vs Replication Controllers/Job)

- 如果有一个可行的方案替代裸Pod（换句话说：Pod没有绑定到 `replication controller`），请使用替代方案。在Node发生故障的情况下，裸Pod将不会被重新调度。

除了某些明确的 `restartPolicy: Never` 方案外，Replication Controller总是比直接创建Pod更为可取。一个 `Job` 对象（目前处于Beta状态）也可能适合你。

## Services

- 通常最好在相应的 `replication controllers` 之前创建一个 `service`。这使得scheduler能够分散构成Service的Pod。

您还可以使用此过程来确保至少一个副本在创建许多副本之前起作用：

- i. 创建Replication Controller而不指定副本（这将会设置 `replicas = 1`）；
- ii. 创建Service

iii. 然后扩容Replication Controller。

- 除非绝对必要，否则不要使用 `hostPort`（例如：对于node daemon）。它指定要在Node上暴露的端口号。当您将Pod绑定到 `hostPort`，由于端口冲突，调度Pod的数量有限 - 您只能调度与Kubernetes集群中的Node一样多的Pod。

如果只需访问端口进行调试，可以使用 `kubectl proxy and apiserver proxy` 或 `kubectl port-forward`。您可以使用 `Service` 对象进行外部服务访问。

如果您需要在特定主机上暴露Pod的端口，请优先使用 `NodePort` Service，再考虑使用 `hostPort`。

- 由于与 `hostPort` 相同的原因，避免使用 `hostNetwork`。
- 当您不需要kube-proxy负载均衡时，使用 `headless services` 轻松发现服务。见 `headless services`。

## Using Labels（标签的使用）

- 定义、使用能够标识应用程序或部署的语义属性的 `labels`。例如，不要将Label附加到一组Pod来显式表示某些Service（例如 `service: myservice`），或者显式地表示管理Pod的replication controller（例如 `controller: mycontroller`），你应该附加标识语义的Label属性，例如 `{ app: myapp, tier: frontend, phase: test, deployment: v3 }`。这将允许您选择适用于上下文的对象组—例如，所有标记了 `tier: frontend` 的Pod的Service；或“myapp”应用的所有“test”阶段的组件。有关此方法的示例，请参阅 `guestbook` 应用程序。

可通过简单的、从其选择器中省略特定于发行版的标签，而非更新Service的选择器以完全Replication Controller的选择器的方式，来实现跨越多个部署的Service，例如 `rolling updates`。

- 为方便滚动更新，请在Replication Controller名称中包含版本信息，例如作为名称的后缀。设置 `version` 标签也很有用。滚动更新创建一个新的Controller，而不是修改现有的Controller。因此，版本无关的Controller名称可能会有问题。有关更多详细信息，请参阅rolling-update命令中的 `documentation`。

请注意，`Deployment` 对象避免了管理Replication Controller `version names` 的需要。对象的期望状态由Deployment描述，如果应用对该spec的更改，则Deployment Controller将以可控的速率，来将实际状态更改为期望状态。（Deployment对象当前是 `extensions API Group` 的一部分。）

- 您可以操作Label进行调试。由于Kubernetes Replication Controller和Service使用Label来匹配Pod，因此可通过删除相关Label来删除Controller或Service的Pod。如果您删除现有Pod的Label，其Controller将会创建一个新的Pod来取代它。这是一个有用的方法，用来调试隔离环境中之前“活着”的Pod。请参阅 `kubectl label` 命令。

## Container Images（容器镜像）

- `default container image pull policy` 是 `IfNotPresent`，如果镜像已经存在，则会导致 `Kubelet` 不会拉取镜像的问题。如果您想始终强制拉取镜像，则必须在.yaml文件中指定 `Always` 的拉取策略（`imagePullPolicy: Always`），或在镜像上指定 `:latest` 标签。

也就是说，如果您指定的镜像不是 `:latest` 标签，例如 `myimage:v1`，并且更新了有相同标签的镜像，

则Kubelet将不会拉取新的镜像。 可通过更新镜像标签时同时更新镜像的标签（例如改为 `myimage:v2` ），并确保您的配置指向正确的版本，从而解决此问题。

注意：在生产中部署容器时，应避免使用 `:latest` 标签，因为这很难跟踪正在运行哪个版本的镜像并且很难回滚。

- 要使用特定版本的镜像，可使用其摘要（SHA256）来指定镜像。这种方法保证镜像永远不会更新。有关使用镜像摘要的详细信息，请参阅 [the Docker documentation](#) 。

## Using kubectl ( kubectl的使用 )

- 在可能的情况下使用 `kubectl create -f <directory>` 。这将会 `<directory>` 中查找所有 `.yaml` ，`.yml` 和 `.json` 文件中查找配置对象，并将其传给 `create` 命令。
- 使用 `kubectl delete` 而不是 `stop` 。 `delete` 具有 `stop` 功能的超集，`stop` 已被弃用。
- 使用kubectl批量操作（通过文件和/或Label）进行get和delete操作。 请参阅 [label selectors and using labels effectively](#) 。
- 使用 `kubectl run` 和 `expose` 快速创建和暴露单个容器部署。有关示例，请参阅[quick start guide](#) 。

## 原文

<https://kubernetes.io/docs/concepts/configuration/overview/>

# Managing Compute Resources for Containers (管理容器的计算资源)

译者按：本节中，笔者将request翻译成最小需求，limit翻译成最大限制。由于出现的次数太多，故而绝大多数地方直接不翻译了，大家可以当做术语来阅读。

指定 `Pod` 时，可选择指定每个容器需要多少CPU和内存（RAM）。当容器指定了最小资源需求时，Scheduler可对Pod调度到哪个Node上进行更好的决策。当容器具有指定的资源限制时，可以指定的方式，处理Node上资源的争抢。有关资源的最小需求和最大限制之间的差异的更多信息，请参阅 [Resource QoS](#)。

## Resource types (资源类型)

CPU和内存都是资源类型。资源类型有基本单元。CPU以核心为单位指定，内存以字节为单位指定。

CPU和内存统称为计算资源，也可称为资源。计算资源是可以请求、分配和消费的，可测量的数量。它们与 [API resources](#)。API资源（如Pods和 [Services](#) 是可通过Kubernetes API Server读取和修改的对象。

## Resource requests and limits of Pod and Container (Pod和容器资源的最小需求与最大限制)

Pod的每个容器可指定以下一个或多个：

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`

尽管只能在每个容器上指定request和limit，但这样既可方便地算出Pod资源的request和limit。特定资源类型的Pod resource request/limit是Pod中每个容器该类型资源的request/limit的总和。

## Meaning of CPU (CPU的含义)

CPU资源的request和limit以cpu为单位。在Kubernetes中，一个cpu相当于：

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 *Hyperthread* on a bare-metal Intel processor with Hyperthreading

允许小数。具有 `spec.containers[].resources.requests.cpu=0.5` 的容器，保证其所需的CPU资源是需要 `1cpu` 容器资源的一半。表达式 `0.1` 等价于表达式 `100m`，可看作“100millicpu”。有些人说“100 millicore”，表达的也是一个意思。具有小数点的请求（如 `0.1`，会由API转换为 `100m`，精度不超过 `1m`。

CPU始终被要求作为绝对数量，从不作为相对数量；0.1在单核、双核或48核机器中，表示的是相同数量的CPU。

## Meaning of memory (内存的含义)

`memory` 的request和limit以字节为单位。可使用整数或定点整数来表示内存，并使用如下后缀之一：E、P、T、G、M、K；也可使用：Ei, Pi, Ti, Gi, Mi, Ki。 例如，以下代表大致相同的值：

```
1. 128974848, 129e6, 129M, 123Mi
```

如下是一个例子。如下Pod有两个容器。每个容器都有0.25 cpu和64MiB (226字节) 内存的request。 每个容器的内存限制为0.5 cpu和128MiB。你可以说Pod有0.5 cpu和128 MiB内存的request，有1 cpu和256MiB内存的limit。

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: frontend
5. spec:
6.   containers:
7.     - name: db
8.       image: mysql
9.       resources:
10.        requests:
11.          memory: "64Mi"
12.          cpu: "250m"
13.        limits:
14.          memory: "128Mi"
15.          cpu: "500m"
16.     - name: wp
17.       image: wordpress
18.       resources:
19.        requests:
20.          memory: "64Mi"
21.          cpu: "250m"
22.        limits:
23.          memory: "128Mi"
24.          cpu: "500m"
```

## How Pods with resource requests are scheduled (如何调度带有request的Pods)

当您创建一个Pod时，Kubernetes Scheduler将为Pod选择一个Node。对于各种资源类型，每个Node都有最大容量：可为Pod提供的CPU和内存量。Scheduler确保对于每种资源类型，调度到该Node的所有容器的request之和小于该Node的容量。请注意，尽管Node上的实际内存或CPU资源使用量非常低，但如果容量检查失败，那么Scheduler仍会拒绝在该Node上放置一个Pod。这样可在资源使用稍后增加时，例如在请求的高峰期，防止Node上的资源短缺。

## How Pods with resource limits are run (带有资源



## limit的Pod是如何运行的)

当kubelet启动Pod的容器时，它将CPU和内存限制传递到容器运行时。

使用Docker时：

- `spec.containers[].resources.requests.cpu` 转换为其核心值，该值可能是小数，乘以1024。该数字中的较大值或2用作 `docker run` 命令中 `--cpu-shares` 的值。
- `spec.containers[].resources.limits.cpu` 转换为其millicore值并乘以100。结果值是容器每100ms可以使用的CPU时间总量。在此间隔期间，容器不能占用超过其CPU时间的份额。

注意：默认配额期限为100ms。CPU配额的最小分辨率为1ms。

- `spec.containers[].resources.limits.memory` 会被转换为一个整数，并用作 `docker run` 命令中 `--memory` 标志的值。

如果容器超出其内存limit，则可能会被终止。如果容器能够重新启动，则与所有其他类型的运行时故障一样，kubelet将重新启动它。

如果一个容器超出其内存request，那么当Node内存不满足要求时，Pod可能会被逐出。

容器可能被允许或不允许长时间超过其CPU limit。然而，即使CPU使用量过大，容器也不会被杀死。

要确定容器是否由于资源limit而无法调度或被杀死，请参阅 [Troubleshooting](#) 部分。

## 监控计算资源使用情况 (Monitoring compute resource usage)

Pod的资源使用情况被报告为Pod status的一部分。

如果为集群配置了 [optional monitoring](#)，那么即可从监控系统查询Pod资源的使用情况。

## Troubleshooting (故障排查)

### My Pods are pending with event message failedScheduling

如果Scheduler找不到任何Pod能够匹配的Node，则Pod将保持unscheduled状态。每当调度程序找不到地方调度Pod时，会产生一个事件，如下所示：

```
1. $ kubectl describe pod frontend | grep -A 3 Events
2. Events:
3.   FirstSeen LastSeen  Count From          Subobject    PathReason    Message
4.   36s      5s        6     {scheduler }   FailedScheduling Failed for reason PodExceedsFreeCPU and possibly others
```



在上述示例中，由于Node上的CPU资源不足，名为“frontend”的Pod无法调度。 如果内存不足，也可能导致失败，并提示类似的错误消息（PodExceedsFreeMemory）。一般来说，如果一个Pod处于pending状态，并带有这种类型的消息，有几件事情要尝试：

- 向集群添加更多Node。
- 终止不需要的Pod，为处于pending的Pod腾出空间。
- 检查Pod是否不大于所有Node。例如，如果所有Node的容量为 `cpu: 1`，那么request = `cpu: 1.1` 的Pod将永远不会被调度。

可使用 `kubectl describe nodes` 命令检查Node的容量和数量。 例如：

```
1. $ kubectl describe nodes e2e-test-minion-group-4lw4
2. Name: e2e-test-minion-group-4lw4
3. [ ... lines removed for clarity ...]
4. Capacity:
5.   alpha.kubernetes.io/nvidia-gpu: 0
6.   cpu: 2
7.   memory: 7679792Ki
8.   pods: 110
9. Allocatable:
10.  alpha.kubernetes.io/nvidia-gpu: 0
11.  cpu: 1800m
12.  memory: 7474992Ki
13.  pods: 110
14. [ ... lines removed for clarity ...]
15. Non-terminated Pods: (5 in total)
16.   Namespace   Name                                CPU Requests  CPU Limits  Memory Requests  Memory Limits
17.   -----   -
18.   kube-system fluentd-gcp-v1.38-28bv1        100m (5%)    0 (0%)     200Mi (2%)     200Mi (2%)
19.   kube-system kube-dns-3297075139-61lj3        260m (13%)   0 (0%)     100Mi (1%)     170Mi (2%)
20.   kube-system kube-proxy-e2e-test-...        100m (5%)    0 (0%)     0 (0%)         0 (0%)
21.   kube-system monitoring-influxdb-grafana-v4-z1m12  200m (10%)   200m (10%)  600Mi (8%)     600Mi (8%)
22.   kube-system node-problem-detector-v0.1-fj7m3    20m (1%)     200m (10%)  20Mi (0%)      100Mi (1%)
23. Allocated resources:
24.   (Total limits may be over 100 percent, i.e., overcommitted.)
25.   CPU Requests  CPU Limits  Memory Requests  Memory Limits
26.   -----
27.   680m (34%)    400m (20%)  920Mi (12%)     1070Mi (14%)
```

由如上输出可知，如果一个Pod的request超过1120mCPU或6.23Gi内存，它将不适合该Node。

通过查看 `Pods` 部分，可查看哪些Pod占用Node上的空间。

译者按：CPU 1120m是这么算的：1800m (Allocatable) - 680m (Allocated)。同理，内存是7474992Ki - 1070Mi

Pods所用的资源量必须小于Node容量，因为系统守护程序需要使用一部分资源。 `allocatable` 字段 `NodeStatus` 给出了Pod可用的资源量。有关更多信息，请参阅 [Node Allocatable Resources](#)。

可配置 `resource quota` 功能，从而限制能够使用的资源总量。 如果与Namespace一起使用，则可防止一个团队占用所有资源。

# My Container is terminated

由于资源不足，容器可能会被终止。要查看容器是否因为资源限制而被杀死，请在感兴趣的Pod上调用

```
kubectl
```

```
describe pod :
```

```
1. [12:54:41] $ kubectl describe pod simmemleak-hra99
2. Name: simmemleak-hra99
3. Namespace: default
4. Image(s): saadali/simmemleak
5. Node: kubernetes-node-tf0f/10.240.216.66
6. Labels: name=simmemleak
7. Status: Running
8. Reason:
9. Message:
10. IP: 10.244.2.75
11. Replication Controllers: simmemleak (1/1 replicas created)
12. Containers:
13.   simmemleak:
14.     Image: saadali/simmemleak
15.     Limits:
16.       cpu: 100m
17.       memory: 50Mi
18.     State: Running
19.       Started: Tue, 07 Jul 2015 12:54:41 -0700
20.     Last Termination State: Terminated
21.       Exit Code: 1
22.       Started: Fri, 07 Jul 2015 12:54:30 -0700
23.       Finished: Fri, 07 Jul 2015 12:54:33 -0700
24.     Ready: False
25.     Restart Count: 5
26. Conditions:
27.   Type      Status
28.   Ready     False
29. Events:
30.   FirstSeen          LastSeen          Count   From
31.   SubobjectPath      Reason           Message
32.   Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700   1      {scheduler }
33.   scheduled   Successfully assigned simmemleak-hra99 to kubernetes-node-tf0f
34.   Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700   1      {kubelet kubernetes-node-tf0f}
35.   implicitly required container POD   pulled          Pod container image "gcr.io/google_containers/pause:0.8.0"
36.   already present on machine
37.   Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700   1      {kubelet kubernetes-node-tf0f}
38.   implicitly required container POD   created         Created with docker id 6a41280f516d
39.   Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700   1      {kubelet kubernetes-node-tf0f}
40.   implicitly required container POD   started         Started with docker id 6a41280f516d
41.   Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700   1      {kubelet kubernetes-node-tf0f}
42.   spec.containers{simmemleak}        created         Created with docker id 87348f12526a
```

在上述示例中， `Restart Count: 5` 表示Pod中的 `simmemleak` 容器已终止并重启了5次。

可使用 `kubectl get pod` 的 `-o go-template=...` 选项来获取先前终止的Containers的状态：

```
1. [13:59:01] $ kubectl get pod -o go-template='{{range .status.containerStatuses}}{{"Container Name: "}}{{.name}}
```

```

    {"r\nLastState: "}}{.lastState}}{end}}'  simmemleak-hra99
2. Container Name: simmemleak
3. LastState: map[terminated:map[exitCode:137 reason:OOM Killed startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z containerID:docker://0e4095bba1feccdfef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22fad8b2]]

```

您可以看到容器由于 `reason:OOM Killed` 而终止，其中 `OOM` 代表Out Of Memory。

## Local ephemeral storage (alpha feature) (ephemeral-storage, 本地临时存储 (Alpha功能))

Kubernetes 1.8版本引入了一种新的资源，用于管理本地临时存储的ephemeral-storage。 在每个Kubernetes Node中，kubelet的根目录（默认 `/var/lib/kubelet`）和日志目录（`/var/log`）存储在Node的根分区上。 此分区也可由Pod通过EmptyDir Volume、容器日志、镜像层以及容器可写层等进行共享和使用。

该分区是“短暂的”，应用程序不能对此分区的性能SLA（例如磁盘IOPS）有期望。 Local ephemeral storage管理仅适用于根分区；镜像层和可写层的可选分区超出了Local ephemeral storage的范围。

注意：如果使用可选的运行时分区，根分区将不会保存任何镜像层或可写层。

译者按：

SLA: <https://baike.baidu.com/item/SLA/2957862>

IOPS: <https://baike.baidu.com/item/IOPS/3105194>

系统SLA和监控流程: <http://www.doc88.com/p-9082091179407.html>

## Requests and limits setting for local ephemeral storage (local ephemeral storage的request和limit设置)

Pod的每个容器可指定以下一个或多个：

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

`ephemeral-storage` 的request和limit以字节为单位。可使用整数或定点整数来表示内存，并使用如下后缀之一：E、P、T、G、M、K。也可使用：Ei、Pi、Ti，Gi、Mi、Ki。 例如，以下代表大致相同的值：

```
1. 128974848, 129e6, 129M, 123Mi
```

例如，以下Pod有两个容器。每个容器有一个2GiB的local ephemeral storage的request。每个容器的local ephemeral storage的limit是4GiB。因此，Pod有4GiB的local ephemeral storage的request，limit为8GiB。

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: frontend
5. spec:

```

```

6.   containers:
7.     - name: db
8.       image: mysql
9.       resources:
10.        requests:
11.          ephemeral-storage: "2Gi"
12.        limits:
13.          ephemeral-storage: "4Gi"
14.     - name: wp
15.       image: wordpress
16.       resources:
17.        requests:
18.          ephemeral-storage: "2Gi"
19.        limits:
20.          ephemeral-storage: "4Gi"

```

## How Pods with ephemeral-storage requests are scheduled ( 如何调度设置了ephemeral-storage request的Pod )

当您创建一个Pod时，Kubernetes Scheduler将为Pod选择一个Node。每个Node具有能够为Pod提供的local ephemeral storage最大量值。（有关详细信息，请参见 [“Node Allocatable”](#)）。Scheduler确保调度的容器的资源需求总和小于Node的容量。

## How Pods with ephemeral-storage limits run ( 如何运行设置了ephemeral-storage limit的Pod )

对于容器级别的隔离，如果容器可写层和日志的使用超出其存储限制，则该Pod将被驱逐。对于Pod级别的隔离，如果所有容器的local ephemeral storage使用量的综合超过限制，则Pod将被驱逐，同理，Pod的EmptyDir也是如此。

## Opaque integer resources (alpha feature) ( 不透明的整数资源 (alpha特征) )

废弃通知：从 [Kubernetes v1.8](#) 开始，该特性已被 [deprecated](#)。

既已废弃，就没有翻译的必要了。多抱半小时老婆吧。该功能的替代品是Extended Resources。

## Extended Resources ( 扩展资源 )

Kubernetes 1.8版引入了Extended Resources。Extended Resources是 [kubernetes.io](#) 域名之外的完全合格的资源名称。Extended Resources允许集群运营商发布新的Node级别的资源，否则系统将无法识别这些资源。Extended Resources数量必须是整数，不能过大。

用户可像CPU和内存一样使用Pod spec中的Extended Resources。Scheduler负责资源计算，以便分配给Pod

的资源部超过可用的资源量。

API Server将Extended Resources的数量限制为整数，例如 `3Ki` 和 `3Ki` 是有效的，`0.5` 和 `1500m` 是无效的。

注意：扩展资源替代 [Opaque Integer Resources](#) 。 用户可使用 `kubernetes.io/` 域名之外的任何域名前缀，而非以前的 `pod.alpha.kubernetes.io/opaque-int-resource-` 前缀。

使用Extended Resources需要两步。首先，集群操作员必须在一个或多个Node上发布per-node Extended Resource。第二，用户必须在Pod中请求Extended Resource。

要发布新的Extended Resource，集群操作员应向API Server提交 `PATCH` HTTP请求，从而指定集群中Node的 `status.capacity` 。在此操作之后，Node的 `status.capacity` 将包含一个新的资源。`status.allocatable` 字段由kubelet异步地使用新资源自动更新。请注意，由于Scheduler在评估Pod适应度时，会使用Node的 `status.allocatable` 值，所以在 使用新资源PATCH到Node容量 和 第一个Pod请求该Node上资源 之间可能会有短暂的延迟。

示例：

如下是一个示例，显示如何使用 `curl` 构建一个HTTP请求，该请求在Node `k8s-node-1` （Master是 `k8s-master` ）上发布了5个“example.com/foo”资源。

```
1. curl --header "Content-Type: application/json-patch+json" \
2. --request PATCH \
3. --data ' [{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"} ]' \
4. http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

注意：在上述请求中，`~1` 是PATCH路径中字符 `/` 的编码。JSON-Patch中的操作路径值被拦截为JSON指针。有关更多详细信息，请参阅 [IETF RFC 6901, section 3](#) 。

要在Pod中使用Extended Resource，请将资源名称作为 `spec.containers[].resources.requests` map中key。

注意：Extended resources不能提交过大的值，因此如果request和limit都存在于容器spec中，则两者必须相等。

TODO：这是什么意思？

只有当所有资源的request都满足时（包括cpu、内存和任何Extended Resources），Pod才会被调度。只要资源的request无法被任何Node满足，Pod将保持在 `PENDING` 状态。

示例：

下面的Pod有如下request：2 cpus和1“example.com/foo”（extended resource）。

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: my-pod
5. spec:
6.   containers:
7.     - name: my-container
```

```
8.     image: myimage
9.     resources:
10.        requests:
11.            cpu: 2
12.            example.com/foo: 1
```

## Planned Improvements (计划改进)

Kubernetes 1.5仅允许在容器上指定资源量。计划对Pod中所有容器共享资源的计费进行改进，例如 [emptyDir volumes](#)。

Kubernetes 1.5仅支持容器级别的CPU/内存的request/limit。计划添加新的资源类型，包括node disk space resource和用于添加自定义 [resource types](#) 的框架。

Kubernetes通过支持多层的 [Quality of Service](#) 支持overcommitment of resources。

overcommitment of resources: 笔者理解就是资源超售。

Quality of Service在部分K8s文档上也被简写成QoS。

在Kubernetes 1.5中，对于不同云提供商，或对于同一个云提供商中的不同机器类型，一个CPU单位表达的是不同的意思。例如，在AWS上，Node的容量在 [ECUs](#) 中报告，而在GCE中报告为逻辑内核。我们计划修改cpu资源的定义，从而使得在提供商和平台之间更一致。

## What's next

- 掌握 [assigning Memory resources to containers and pods](#) 的实践经验。
- 掌握 [assigning CPU resources to containers and pods](#) 的实践经验。
- [Container](#)
- [ResourceRequirements](#)

## 原文

<https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/>

# K8s资源分配

## 准备工作

本节中的示例需要安装Heapster。因此，需要先安装Heapster，对于Minikube，执行如下操作即可。

```
1. minikube addons enable heapster
```

查看Heapster Service是否已在运行：

```
1. kubectl get services --namespace=kube-system
```

## TIPS

按理，对于Minikube，启用Heapster addon后，即可在Dashboard上看到各种资源的CPU占用等指标。然而，对于 `minikube 0.22.x`，在Dashboard上无法看到，可使用如下命令，直接查询Grafana监控界面：

```
1. minikube addons open heapster
```

相关Issue: <https://github.com/kubernetes/minikube/issues/2013>

## 限制CPU分配

### 示例

- `cpu-request-limit.yaml` :

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: cpu-demo
5. spec:
6.   containers:
7.   - name: cpu-demo-ctr
8.     image: vish/stress
9.     resources:
10.      # 最多使用1个CPU单位
11.      limits:
12.        cpu: "1"
13.      # 至少保证0.5个CPU单位
14.      requests:
15.        cpu: "0.5"
16.      # 容器启动时的参数
17.      args:
```

```

18.      # 使用2个CPU单位
19.      - -cpus
20.      - "2"

```

- 创建Pod：

```
1. kubectl create -f cpu-request-limit.yaml
```

- 查看Pod状态：

```
1. kubectl get pod cpu-demo -o yaml
```

- 启动 `proxy` ，并查询配额信息：

```

1. kubectl proxy
2. # 另起一个终端，输入：
3. curl http://localhost:8001/api/v1/proxy/namespaces/kube-
    system/services/heapster/api/v1/models/namespaces/default/pods/cpu-demo/metrics/cpu/usage_rate

```

即可看到监控信息。

在本例中，尽管容器启动时，尝试使用2个CPU单位，但由于配置了只允许使用1个CPU单位，因此，最终最多只能使用1个CPU单位。

## CPU单位

CPU资源以`cpu`为单位。 在Kubernetes，一个cpu相当于：

- 1 AWS vCPU
- 1个GCP核心
- 1 Azure vCore
- 1个在裸机Intel处理器上的超线程

允许小数值。你可以使用m后缀来表示“毫”。例如100m cpu，100millicpu和0.1cpu表达的含义其实是相同的。精度不允许超过1m。

精度不允许超过1m的意思是，你不能指定有500.88m个CPU单位，精度最小是毫，就像人民币中最小的单位是分一样。

CPU配额是一个绝对值，而非相对值；“0.1 CPU”在单核、双核或48核机器表示的配额是相同的。

## 配额超过任何Node的容量

CPU最低要求和最大限制与容器相关联，但将Pod视为具有CPU最小配合和最大限制是有用的。Pod的CPU最小配需求是Pod中所有容器的CPU请求的总和。同样，Pod的CPU最大限制是Pod中所有容器的CPU最大限制的总和。

Pod的调度是基于最低要求的。只有当Node具有足够可用的CPU资源时，Pod才会在Node上运行。

在本练习中，您将创建一个非常大的CPU最低要求，以至于它超出了集群中任何Node的容量。以下是具有一个容器的



Pod的配置文件。容器请求100 cpu，这可能超过集群中任何Node的容量。

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: cpu-demo-2
5. spec:
6.   containers:
7.   - name: cpu-demo-ctr-2
8.     image: vish/stress
9.     resources:
10.    limits:
11.      cpu: "100"
12.    requests:
13.      cpu: "100"
14.    args:
15.    - -cpus
16.    - "2"

```

创建该Pod后，使用 `kubect1 get pod cpu-demo-2` 查看Pod状态，即可看到类似如下的结果。由输出可知，Pod的状态为Pending。也就是说，Pod并没有被调度到任何Node上运行，它将无限期地保持在Pending状态。

```

1. kubect1 get pod cpu-demo-2
2. NAME          READY    STATUS    RESTARTS   AGE
3. cpu-demo-2    0/1     Pending   0           7m

```

使用 `kubect1 describe pod cpu-demo-2` 命令查看Pod详情，即可看到类似如下的结果。由输出可看到无法调度的具体原因。

```

1. Events:
2.   Reason          Message
3.   -----
4.   FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient cpu (3).

```

## 限制内存分配

### 示例

- `memory-request-limit.yaml`

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: memory-demo
5. spec:
6.   containers:
7.   - name: memory-demo-ctr

```

```

8.     image: vish/stress
9.     resources:
10.      # 最大允许使用200Mi内存
11.      limits:
12.        memory: "200Mi"
13.      # 至少保证200Mi的内存资源
14.      requests:
15.        memory: "100Mi"
16.     args:
17.      # 尝试分配150Mi内存给容器
18.      - -mem-total
19.      - 150Mi
20.      - -mem-alloc-size
21.      - 10Mi
22.      - -mem-alloc-sleep
23.      - 1s

```

- 查看Pod状态：

```
1. kubectl get pod memory-demo -o yaml
```

可看到类似如下结果：

```

1. ...
2. resources:
3.   limits:
4.     memory: 200Mi
5.   requests:
6.     memory: 100Mi
7. ...

```

- 启动 `proxy` 并查询配额信息：

```

1. kubectl proxy
2. curl http://localhost:8001/api/v1/proxy/namespaces/kube-
   system/services/heapster/api/v1/model/namespaces/default/pods/memory-demo/metrics/memory/usage

```

可看到如下结果：

```

1. {
2.   "timestamp": "2017-06-20T18:54:00Z",
3.   "value": 162856960
4. }

```

由结果可知，Pod正使用大约162,900,000的内存，大约为150 MiB。这比Pod的100 MiB要求大，但在Pod的200MiB限制之内。

## 超出容器的内存限制

示例：

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: memory-demo-2
5. spec:
6.   containers:
7.     - name: memory-demo-2-ctr
8.       image: vish/stress
9.       resources:
10.        requests:
11.          memory: 50Mi
12.        limits:
13.          memory: "100Mi"
14.      args:
15.        - -mem-total
16.        - 250Mi
17.        - -mem-alloc-size
18.        - 10Mi
19.        - -mem-alloc-sleep
20.        - 1s

```

此时，容器将不能运行。Container会因为内存不足（OOM）而被杀死。

类似的，如果指定Pod的内存超过K8s集群中任何Node的内存，Pod也无法运行。

## 内存单位

内存资源以字节为单位。使用一个整数或一个定点整数跟上以下后缀，来描述内存：E，P，T，G，M，K，Ei，Pi，Ti，Gi，Mi，Ki。 例如，以下代表大致相同的值：

```

1. 128974848, 129e6, 129M , 123Mi

```

## 参考文档

- 官方文档：《Assign CPU Resources to Containers and Pods》：<https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/>
- 官方文档：《Assign Memory Resources to Containers and Pods》：<https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/#memory-units>
- k8s的资源分配：<https://segmentfault.com/a/1190000003506106>

# Assigning Pods to Nodes (将Pod分配到Node)

您可以约束一个 `pod`，让其只能在特定 `nodes` 上运行，或者更倾向于在特定Node上运行。有几种方法能做到这点，他们都使用 `label selectors` 进行选择。通常这种约束不是必要的，因为Scheduler会自动执行合理的放置（例如：在Node之间传播您的Pod，而不是将Pod放在一个没有足够资源的Node上等），但在某些情况下，您可能需要控制一个Pod落在特定的Node上，例如：确保一个Pod在一个有SSD的机器上运行，或者将来自两个不同的服务的Pod放到相同的可用区域。

您可以在 [in our docs repo here](#) 找到这些示例的所有文件。

## nodeSelector

`nodeSelector` 是最简单的约束形式。`nodeSelector` 是PodSpec的一个字段。它指定键值对的映射。为了使Pod能够在Node上运行，Node必须将每个指定的键值对作为标签（也可有其他标签）。最常见的用法是使用一个键值对。

下面，我们来看看如何使用 `nodeSelector`。

## Step Zero: Prerequisites (先决条件)

这个例子假设你对Kubernetes Pod有一个基本的了解，并已经 [turned up a Kubernetes cluster](#)。

## Step One: Attach label to the node (将Label附加到Node)

运行 `kubectl get nodes`，获取集群Node的名称。选择要添加Label的Node，然后运行 `kubectl label nodes <node-name> <label-key>=<label-value>`，向您所选的Node添加Label。例如，如果我的Node名称是 `kubernetes-foo-node-1.ca-robinson.internal`，而我想要的标签是 `disktype=ssd`，那么可使用 `kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd`。

如果此命令出现“invalid command”的错误，那么你可能使用的是旧版本的kubectl，它没有 `label` 命令。在这种情况下，有关如何在Node上手动设置Label的说明，请参阅本指南的 [previous version](#)。

另外请注意，Label的key必须采用DNS标签的格式（如 [identifiers doc](#) 所述），这意味着key不允许包含大写字母。

您可以通过重新运行 `kubectl get nodes --show-labels` 并检查Node是否已经有你所设的标签标签，来验证Label是否成功添加。

## Step Two: Add a nodeSelector field to your pod configuration (将nodeSelector字段添加到您的Pod配置)

在任意一个你想运行的Pod的配置文件中添加nodeSelector部分，如下所示。例如，如果我的Pod配置如下：

```
1. apiVersion: v1
```

```

2.  kind: Pod
3.  metadata:
4.    name: nginx
5.    labels:
6.      env: test
7.  spec:
8.    containers:
9.      - name: nginx
10.       image: nginx

```

需要像这样添加一个nodeSelector：

```

1.  apiVersion: v1
2.  kind: Pod
3.  metadata:
4.    name: nginx
5.    labels:
6.      env: test
7.  spec:
8.    containers:
9.      - name: nginx
10.       image: nginx
11.       imagePullPolicy: IfNotPresent
12.    nodeSelector:
13.      disktype: ssd

```

当您运行 `kubectl create -f pod.yaml` 时，该Pod将在您拥有以上标签的Node上调度！您可以通过运行 `kubectl get pods -o wide` 查看该Pod所在的“NODE”，来验证是否正常工作。

## Interlude: built-in node labels (Interlude: Node的内置标签)

除你 `attach` 的Label以外，Node还预设了一组标准的Label。从Kubernetes v1.4起，这些Label是：

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `beta.kubernetes.io/os`
- `beta.kubernetes.io/arch`

## Affinity and anti-affinity (亲和与反亲和)

`nodeSelector` 提供了一种非常简单的方式，从而将Pod约束到具有特定Label的Node。目前处于beta阶段的Affinity/Anti-affinity特性极大地扩展了您可以表达的约束类型。关键的改进是：

1. 语言更具表现力（不仅仅是“使用AND、完全匹配”）
2. 可指定“soft”/“preference”规则，而非影响要求，因此如果Scheduler不能满足你的要求，则该Pod仍将

被安排

3. 您可以限制在Node（或其他拓扑域）上运行的其他Pod的标签，而非针对Node本身上的标签，这允许设置规则，指定哪些Pod可以并且不能放到一起。

Affinity特性由两种affinity组成：“node affinity”和“inter-pod affinity/anti-affinity”。Node affinity类似于现有的节点 `nodeSelector`（但具有上面列出的前两个优点）；而inter-pod affinity/anti-affinity约束Pod的Label而非Node的Label，此特性除具有上面列出的前两项性质之外，还有如上述第三项中所述的性质。

`nodeSelector` 继续像往常一样工作，但最终将会被废弃，因为Node Affinity可表示 `nodeSelector` 所能表达的所有内容。

## Node affinity (beta feature) (Node亲和性 (beta特性))

Node Affinity在Kubernetes 1.2中作为alpha功能引入。Node Affinity在概念上类似于 `nodeSelector`——它允许您根据Node上的Label来约束您的Pod可被调度哪些Node。

目前有两种类型的Node Affinity，称为 `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`。您可以将它们分别认为是“hard”和“soft”，前者规定了要将Pod调度到Node上时，必须满足的规则（就像 `nodeSelector`，但使用更具表现力的语法），而后者指定调度程序将尝试强制调度但不能保证的首选项。名称中的“IgnoredDuringExecution”部分表示，与 `nodeSelector` 工作方式类似，如果Node上的Label在运行时更改，导致不再满足Pod上的Affinity规则，则该Pod仍将继续在该Node上运行。在未来，我们计划提供 `requiredDuringSchedulingRequiredDuringExecution`，它和 `requiredDuringSchedulingIgnoredDuringExecution` 一样，只是它会从不再满足Pod的Node Affinity要求的Node中驱逐Pod。

因此，`requiredDuringSchedulingIgnoredDuringExecution` 的一个示例是“仅在在有Intel CPU的Node上运行Pod”，并且一个 `preferredDuringSchedulingIgnoredDuringExecution` 的一个示例是“尝试在可用区XYZ中运行此组Pod，但如果无法做到，则允许在其他地方运行”。

Node Affinity被指定 `nodeAffinity` 字段，它是 `PodSpec` 中 `affinity` 字段的子字段。

以下是一个使用Node Affinity的Pod的示例：

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: with-node-affinity
5. spec:
6.   affinity:
7.     nodeAffinity:
8.       requiredDuringSchedulingIgnoredDuringExecution:
9.         nodeSelectorTerms:
10.        - matchExpressions:
11.          - key: kubernetes.io/e2e-az-name
12.            operator: In
13.            values:
14.              - e2e-az1
15.              - e2e-az2
16.       preferredDuringSchedulingIgnoredDuringExecution:
```

```

17.     - weight: 1
18.       preference:
19.         matchExpressions:
20.           - key: another-node-label-key
21.             operator: In
22.             values:
23.               - another-node-label-value
24.   containers:
25.   - name: with-node-affinity
26.     image: gcr.io/google_containers/pause:2.0

```

该Node Affinity规则表示，该Pod只能放在带有 `kubernetes.io/e2e-az-name` 的Label的Node上，其值为 `e2e-az1` 或 `e2e-az2`。另外，在符合该标准的Node中，优先使用具有 `another-node-label-key`，值为 `another-node-label-value` 的Node。

在本例中可以看到 `In` 操作符。新Node Affinity语法支持以下操作符：`In`、`NotIn`、`Exists`、`DoesNotExist`、`Gt`、`Lt`。没有明确的“node anti-affinity”概念，但 `NotIn` 和 `DoesNotExist` 提供了这种行为。

如果同时指定 `nodeSelector` 和 `nodeAffinity`，则必须同时满足，才能将Pod调度到候选Node上。

如果指定与 `nodeAffinity` 类型相关联的多个 `nodeSelectorTerms`，那么如果满足 `nodeSelectorTerms` 之一，即可将Pod调度到节点上。

如果指定与 `matchExpressions` 相关联的多个 `matchExpressions`，则只有满足所有 `matchExpressions` 才能将该Pod调度到Node上。

如果删除或更改了调度Pod的Node的Label，则该Pod不会被删除。换句话说，Affinity选择仅在调度Pod时起作用。

有关Node Affinity的更多信息，请参见 [设计文档](#)。

## Inter-pod affinity and anti-affinity (beta feature) (Pod内的亲和性与反亲和性 (Beta特性))

Kubernetes 1.4引入了Inter-pod affinity 和 anti-affinity。Inter-pod affinity和anti-affinity允许您根据已在Node上运行的Pod 上的Label，而非基于Node的Label来约束您的Pod能被调度到哪些Node。规则的形式是“如果X已经运行了一个或多个满足规则Y的Pod，则该Pod应该（或者在anti-affinity的情况下不应该）运行在X中”。Y表示一个与Namespace列表相关联（或“所有”命名空间）的LabelSelector；与Node不同，因为Pod是在Namespace中的（因此Pod上的Label是隐含Namespace的），Pod Label上的Label Selector必须指定选择器要应用的Namespace。概念上，X是一个拓扑域，如Node、机架、云提供商Zone、云提供商Region等。您可以使用 `topologyKey` 表示它，该key是系统用来表示拓扑域的Node标签，例如参见上面 [Interlude: built-in node labels](#) 中列出的键。

与Node Affinity一样，目前有两种类型的pod affinity和anti-affinity，称为

`requiredDuringSchedulingIgnoredDuringExecution` 和 `preferredDuringSchedulingIgnoredDuringExecution`，表示“hard”对“soft”需求。请参阅上文Node Affinity部分中的说明。一个

`requiredDuringSchedulingIgnoredDuringExecution` 的例子是“将同一个Zone中的Service A和Service B的Pod放到一起，因为它们彼此通信很多”，而 `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity表示“将该

Service的Pod跨Zone“（硬性要求没有意义，因为你可能有比Zone更多的Pod）。

Inter-pod affinity用 `podAffinity` 字段指定，它是PodSpec中 `affinity` 字段的子字段。inter-pod anti-affinity用 `podAntiAffinity` 指定，它是 `affinity` 字段的子字段。

An example of a pod that uses pod affinity:

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: with-pod-affinity
5. spec:
6.   affinity:
7.     podAffinity:
8.       requiredDuringSchedulingIgnoredDuringExecution:
9.         - labelSelector:
10.            matchExpressions:
11.              - key: security
12.                operator: In
13.              values:
14.                - S1
15.            topologyKey: failure-domain.beta.kubernetes.io/zone
16.     podAntiAffinity:
17.       preferredDuringSchedulingIgnoredDuringExecution:
18.         - weight: 100
19.           podAffinityTerm:
20.             labelSelector:
21.               matchExpressions:
22.                 - key: security
23.                   operator: In
24.                 values:
25.                   - S2
26.             topologyKey: kubernetes.io/hostname
27.   containers:
28.   - name: with-pod-affinity
29.     image: gcr.io/google_containers/pause:2.0

```

本例中，Pod的Affinity定义了一个Pod Affinity规则和一个Pod anti-affinity规则。在此示例中，`podAffinity` 在是 `requiredDuringSchedulingIgnoredDuringExecution`，而 `podAntiAffinity` 是 `preferredDuringSchedulingIgnoredDuringExecution`。Pod Affinity规则表示，只有当相同Zone中的某个Node至少有一个已经运行的、具有key=security、value=S1的Label的Pod时，该Pod才能调度到Node上。（更准确地说，Pod会运行在这样的Node N上：Node N具有带有 `failure-domain.beta.kubernetes.io/zone` 的Label key和某些value V，即：集群中至少有一个带有key= `failure-domain.beta.kubernetes.io/zone` 以及value=V的Node，其上运行了key=security并且value=S1标签的Pod）。pod anti-affinity规则表示该Pod更倾向于不往那些已经运行着Label key=security且value=S2的Pod的Node上调度。（如果 `topologyKey` 是 `failure-domain.beta.kubernetes.io/zone`，那么这意味着如果相同Zone中的Node中有Pod的key=security并且value=S2，则该Pod不能调度到该Node上”。对于pod affinity以及anti-affinity的更多例子，详见 [design doc](#)。

pod affinity and anti-affinity的合法操作符是 `In`、`NotIn`、`Exists`、`DoesNotExist`。



原则上，`topologyKey` 可以是任何合法的标签key。然而，出于性能和安全的原因，对于`topologyKey`有一些限制：

1. 对于Affinity以及 `RequiredDuringScheduling` pod anti-affinity,, 不允许使用空 (empty) 的 `topologyKey` 。
2. 对于 `RequiredDuringScheduling` pod anti-affinity, 引入了admission controller `LimitPodHardAntiAffinityTopology` , 从而限制到 `kubernetes.io/hostname` 的 `topologyKey` 。如果要使其可用于自定义拓扑, 您可以修改admission controller, 或者简单地禁用它。
3. 对于 `PreferredDuringScheduling` pod anti-affinity, 空 (empty) `topologyKey` `kubernetes.io/hostname` 被解释为“所有拓扑”(“所有拓扑”在这里仅限于 `kubernetes.io/hostname` 和 `failure-domain.beta.kubernetes.io/region` 的组合)。
4. 除上述情况外, `topologyKey` 可以是任何合法的标签key。

除 `labelSelector` 和 `topologyKey` 外, 还可以选择指定 `labelSelector` 应该匹配的Namespace列表 (这与 `labelSelector` 和 `topologyKey` 的定义相同)。如果省略, 默认为: 定义affinity/anti-affinity了的Pod的Namespace。如果定义为空 (empty), 则表示“所有Namespace”。

所有与 `requiredDuringSchedulingIgnoredDuringExecution` affinity以及anti-affinity相关联的 `matchExpressions` 必须满足, 才会将Pod调度到Node上。

## More Practical Use-cases (更多实用的用例)

Interpod Affinity和AnitAffinity在与更高级别的集合 (例如ReplicaSets、Statefulsets、Deployments等) 一起使用时可能更为有用。可轻松配置工作负载应当统统位于同一个拓扑中, 例如, 同一个Node。

Always co-located in the same node (始终位于同一个Node)

在一个有3个Node的集群中, web应用有诸如redis的缓存。我们希望web服务器尽可能地与缓存共存。这是一个简单的redis Deployment的yaml片段, 包含3个副本和选择器标签 `app=store` 。

```
1. apiVersion: apps/v1beta1 # for versions before 1.6.0 use extensions/v1beta1
2. kind: Deployment
3. metadata:
4.   name: redis-cache
5. spec:
6.   replicas: 3
7.   template:
8.     metadata:
9.       labels:
10.        app: store
11.     spec:
12.       containers:
13.       - name: redis-server
14.         image: redis:3.2-alpine
```

在webserver Deployment的yaml代码片段下面配置了 `podAffinity` , 它通知scheduler其所有副本将与具有选择器标签 `app=store` 的Pod共同定位

```
1. apiVersion: apps/v1beta1 # for versions before 1.6.0 use extensions/v1beta1
```

```
2. kind: Deployment
3. metadata:
4.   name: web-server
5. spec:
6.   replicas: 3
7.   template:
8.     metadata:
9.       labels:
10.        app: web-store
11.     spec:
12.       affinity:
13.         podAffinity:
14.           requiredDuringSchedulingIgnoredDuringExecution:
15.             - labelSelector:
16.                 matchExpressions:
17.                   - key: app
18.                     operator: In
19.                     values:
20.                       - store
21.             topologyKey: "kubernetes.io/hostname"
22.     containers:
23.       - name: web-app
```

如果我们创建上述两个Deployment，我们的三节点集群可能如下所示。

node-1	node-2	node-3
webserver-1	webserver-2	webserver-3
cache-1	cache-2	cache-3

如您所见， `web-server` 3个副本将按预期自动与缓存共同定位。

```
1. $kubectl get pods -o wide
2. NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
3. redis-cache-1450370735-6dzlj        1/1     Running   0           8m    10.192.4.2    kube-node-3
4. redis-cache-1450370735-j2j96        1/1     Running   0           8m    10.192.2.2    kube-node-1
5. redis-cache-1450370735-z73mh        1/1     Running   0           8m    10.192.3.1    kube-node-2
6. web-server-1287567482-5d4dz         1/1     Running   0           7m    10.192.2.3    kube-node-1
7. web-server-1287567482-6f7v5         1/1     Running   0           7m    10.192.4.3    kube-node-3
8. web-server-1287567482-s330j         1/1     Running   0           7m    10.192.3.2    kube-node-2
```

最佳实践是配置这些高可用的有状态工作负载，例如有AntiAffinity规则的redis，以保证更好的扩展，我们将在下一节中看到。

Never co-located in the same node (永远不位于同一个Node)

高可用数据库StatefulSet有一主三从，可能不希望数据库实例都位于同一Node中。

node-1	node-2	node-3	node-4
DB-MASTER	DB-REPLICA-1	DB-REPLICA-2	DB-REPLICA-3

[Here](#) 是一个Zookeeper StatefulSet的例子，为高可用配置了anti-affinity的。

有关inter-pod affinity/anti-affinity的更多信息，请参阅 [here](#) 的设计文档。

您也可以检查 [Taints](#) ，这样可让Node排斥一组Pod。

## 原文

---

<https://kubernetes.io/docs/concepts/configuration/assign-pod-node/>

## 参考文档

---

<http://www.php230.com/1491134522.html>



# Taints and Tolerations

译者按：

- Taints: 在本文中根据上下文，有的地方直接叫Taints，有的地方翻译成“污点”或者“污染”。
- Tolerations: 在本文中根据上下文，有的地方直接叫Tolerations，有的地方翻译成“容忍”或“容忍度”。

[here](#) 描述的节点亲和性是一个Pod属性，使用它可将Pod吸引到一组Node（作为优选或硬性要求）。Taint功能相反，它允许Node排斥一组Pod。

Taints和Tolerations一起工作，以确保Pod不被安排在不适当的Node上。一个或多个Taints应用于Node；这标志着Node不应该接受任何不能容忍这些Taints的Pod。Tolerations应用于Pod，并允许（但不要求）Pod可以调度到具有匹配Taints的Node上。

## Concepts（概念）

您可以使用 `kubectl taint` 向节点添加Taint。例如，

```
1. kubectl taint nodes node1 key=value:NoSchedule
```

在 `node1` 上设置了一个Taint。Taint具有key、value以及污点效果 `NoSchedule`。这意味着没有Pod能够调度到 `node1`，除非它具有匹配的Toleration。可在PodSpec中为Pod指定Toleration。以下两种Toleration都与上述 `kubectl taint` 所造成的Taint“匹配”，因此这些Pod将能够调度到 `node1` 上：

```
1. tolerations:
2. - key: "key"
3.   operator: "Equal"
4.   value: "value"
5.   effect: "NoSchedule"
```

```
1. tolerations:
2. - key: "key"
3.   operator: "Exists"
4.   effect: "NoSchedule"
```

如果key相同，并且效果相同，那么Toleration匹配Taint，并且：

- `operator` 为 `Exists`（在这种情况下不应指定 `value`），或
- `operator` 为 `Equal` 并且 `value` 相等

如不指定，则 `Operator` 默认为 `Equal`。

注意：有两种特殊情况：

- `key` 留空配合操作符 `Exists` 可匹配所有key、value和效果，这意味着容忍一切。

```
1. tolerations:
2. - operator: "Exists"
```

- `effect` 留空匹配所有带有该 `key` 效果。

```
1. tolerations:
2. - key: "key"
3. operator: "Exists"
```

上述示例使用 `NoSchedule` 这个 `effect`。或者，可使用 `PreferNoSchedule` 这个 `effect`。它是 `NoSchedule` 的“偏好”或“软”版本—系统将尽量避免放置无法容忍Node上Taint的Pod，但不是必需的。第三种 `effect` 是 `NoExecute`，稍后描述。

您可以在同一个Node上设置多个Taint，并在同一个Pod上设置多个Toleration。Kubernetes处理多个Taint和Toleration的方式就像过滤器：从Node的所有Taint开始，然后忽略具有匹配Toleration的那些Pod；剩下的不被忽视的Taints对Pod有明显影响。尤其是，

- 如果至少有一个无法忽视的Taint，效果为 `NoSchedule`，那么Kubernetes不会将Pod调度到该Node上；
- 如果没有不受忽视的Taint，效果为 `NoSchedule`；但至少有一个无法忽视的Taint，效果为 `PreferNoSchedule`，那么Kubernetes会将尝试不将Pod调度到该Node上；
- 如果至少有一个不受忽视的污点，效果为 `NoExecute`，则该Pod将从Node中驱逐（如果已经在Node上运行），并且不会被调度到该Node上（如果尚未运行在该Node上）。

例如，假设你污染了这样一个Node：

```
1. kubectl taint nodes node1 key1=value1:NoSchedule
2. kubectl taint nodes node1 key1=value1:NoExecute
3. kubectl taint nodes node1 key2=value2:NoSchedule
```

一个Pod有两个Tolerations：

```
1. tolerations:
2. - key: "key1"
3. operator: "Equal"
4. value: "value1"
5. effect: "NoSchedule"
6. - key: "key1"
7. operator: "Equal"
8. value: "value1"
9. effect: "NoExecute"
```

在本例中，由于没有符合第三个Taint的Toleration，该Pod将无法调度到 `node1` 上。但如果在添加Taint时已经在该Node上运行，则能继续运行。

通常情况下，如果一个Node添加了一个带有 `NoExecute` 效果的Taint，那么任何不能Toleration该Taint的Pod将立即被驱逐，任何Toleration该Taint的Pod都不会被驱逐。但是，使用 `NoExecute` 效果的Toleration可指定一个可选的 `tolerationSeconds` 字段，该字段表示在添加Taint后，Pod停驻在该Node的时间。例如，

```

1. tolerations:
2. - key: "key1"
3.   operator: "Equal"
4.   value: "value1"
5.   effect: "NoExecute"
6.   tolerationSeconds: 3600

```

意味着如果该Pod正在运行，并且Taint被添加到Node，则Pod将在该Node上停驻3600秒，然后被驱逐。如果在该时间内删除了Taint，则Pod不会被驱逐。

译者按：`tolerationSeconds` 可以理解为：容忍多少秒，过了这么多秒后，就不容忍了，然后就会被驱逐。

## Example Use Cases（使用案例示例）

Taints和Tolerations是一种能让将Pod“远离”Node或驱逐不应该运行的Pod的方式，该方式非常灵活。一些用例是

- **Dedicated Nodes（专用Node）**：如果要将一组Node专用于一组特定的用户，您可以向这些Node添加一个污点（例如，`kubectl taint nodes nodename dedicated=groupName:NoSchedule`），然后将相应的Toleration添加到它们的Pod（这可以通过编写自定义 `admission controller` 来轻松完成）。然后具有Tolerations的Pod将被允许使用污染（专用）Node以及集群中的任何其他Node。如果要将Node专用于它们，并确保它们只使用专用Node，那么您应该额外添加标签，这些标签类似于同一组Node（例如 `dedicated=groupName`）上的Taint，并且Admission Controller应该另外添加一个Node的亲合性要求，Pod只能调度到标有 `dedicated=groupName` 的Node上。
- **Nodes with Special Hardware（具有特殊硬件的Node）**：在一小部分Node具有专用硬件（例如GPU）的集群中，希望将不需要专用硬件的Pod原理这些Node，从而为需要使用专用硬件的Pod留出空间。这可以通过污染具有专门硬件的Node（例如，`kubectl taint nodes nodename special=true:NoSchedule` 或 `kubectl taint nodes nodename special=true:PreferNoSchedule`）来完成，并将相应的Toleration添加到需使用特殊硬件的Pod。在专用Node的用例中，使用自定义 `admission controller` 应用Tolerations可能是最简单的）。例如，Admission Controller可使用Pod的一些特性来确定应该允许该Pod使用特殊Node。为确保需要特殊硬件的Pod只被调度到具有特殊硬件的Node上，您将需要一些额外的机制，例如您可以使用 `opaque integer resources` 来表示特殊资源，并将其作为PodSpec中的资源最小需求，或您可以标记具有特殊硬件的Node，并在需要硬件的Pod上使用Node Affinity。
- **Taint based Evictions（alpha feature）（基于Taint的驱逐（alpha功能））**：当Node出现问题时，per-pod-configurable的驱逐行为，这将在下一节中描述。

## Taint based Evictions（基于Taint的驱逐）

之前我们提到了 `NoExecute` taint的效果，它会影响已经在Node上运行的Pod，如下所示

- 不能容忍污点的Pod被立即驱逐
- 容忍污点的Pod，而不指定 `tolerationSeconds` 将永远被绑定
- 容许污点的Pod，指定 `tolerationSeconds`，在指定的时间内保持绑定

上述行为是一个beta功能。此外，Kubernetes 1.6具有表示Node问题的alpha支持。换句话说，当某些条件为真

时，Node Controller会自动给Node添加Taint。 目前内置的Taint包括：

- `node.kubernetes.io/not-ready` : Node尚未准备就绪。 这对应于NodeCondition `Ready` 字段为 `False` 。
- `node.alpha.kubernetes.io/unreachable` : Node无法从Node Controller访问。这对应于NodeCondition `Ready` 字段为 `Unknown` 。
- `node.kubernetes.io/out-of-disk` : Node磁盘不可用。
- `node.kubernetes.io/memory-pressure` : Node内存有压力。
- `node.kubernetes.io/disk-pressure` : Node磁盘有压力。
- `node.kubernetes.io/network-unavailable` : Node的网络不可用。
- `node.cloudprovider.kubernetes.io/uninitialized` : 当kubelet以外部cloud provider启动时，它会为Node设置一个Taint，将其标记为未使用。当来自cloud-controller-manager的Controller初始化此Node时，kubelet将删除此Taint。

当启用 `TaintBasedEvictions` alpha功能时（您可以通过在Kubernetes controller manager的 `--feature-gates` 包含 `TaintBasedEvictions=true` 来实现此功能，例如 `--feature-gates=FooBar=true,TaintBasedEvictions=true`），这样，NodeController（或kubelet）将会自动添加Taint，并且基于Ready NodeCondition从Node驱出Pod的逻辑将会被禁用。（注意：为保持由于Node问题而导致的现有 `rate limiting` 行为，系统实际上以rate-limited的方式添加Taints，从而防止在Master节点在发生网络分区故障的场景中有大量的Pod被驱逐。 该alpha特征与 `tolerationSeconds` 相结合，从而允许Pod指定应该保持绑定到具有这些问题中的Node的时长。

例如，具有很多本地状态的应用可能希望在网络分区故障发生的情况下长时间保持绑定到Node，希望分区恢复时避免该驱逐。 在这种情况下，Pod将使用的Toleration情况看起来像：

```
1. tolerations:
2. - key: "node.alpha.kubernetes.io/unreachable"
3.   operator: "Exists"
4.   effect: "NoExecute"
5.   tolerationSeconds: 6000
```

请注意，除非用户提供的Pod配置已经具有 `node.kubernetes.io/not-ready` 的Toleration，否则Kubernetes会自动为 `node.kubernetes.io/not-ready` 添加 `tolerationSeconds=300` 的Toleration。 同样地，对于 `node.alpha.kubernetes.io/unreachable`，也会有 `tolerationSeconds=300` 用户自行设置。

这些自动添加的Toleration确保在检测到这些问题之一后默认保持绑定5分钟。是 `DefaultTolerationSeconds` `admission controller` 添加了这两个默认Toleration。

对于以下没有未设置 `tolerationSeconds` 的Taint，`DaemonSet` Pod是通过 `NoExecute` Toleration来创建的：

- `node.alpha.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

这确保了DaemonSet Pod对这些问题容忍而永远不会被驱逐，这与禁用此功能时的行为相匹配。

## Taint Nodes by Condition (使用Condition为Node添加Taint)

1.8版引入了一个Alpha功能，导致Node Controller创建与Node状态相对应的Taint。启用此功能时（可以通过在Scheduler的 `--feature-gates` 命令中添加 `TaintNodesByCondition=true` 来启用该功能，例如 `--feature-gates=FooBar=true,TaintNodesByCondition=true` ），Scheduler不会检查Node的Condition，而是检查Taint。这样可确保Node Condition不会影响调度到该Node上的内容。用户可通过添加适当的Pod Tolerantion来选择忽略Node的一些问题（在Node Condition中显示）。

为了确保打开此功能不会破坏DaemonSet的特性，从1.8版本开始，DaemonSet Controller会自动将以下 `NoSchedule` 的Tolerantion添加到所有daemon中：

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk` (*only for critical pods*)

上述设置确保向后兼容性，但是我们了解，它们可能无法适应所有用户的需求，这就是为什么集群管理员可能需要向DaemonSet添加Arbitrary Tolerantion的原因。

## 原文

---

<https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/>



# Secrets

`secret` 类型的对象旨在保存敏感信息，例如密码、OAuth token和ssh key。将这些信息放在 `secret` 中比放在 `pod` 定义或Docker镜像中更加安全、灵活。有关更多信息，请参阅 [Secrets design document](#)。

## Overview of Secrets (Secret概述)

Secret是包含少量敏感数据（如密码、token或key）的对象。这样的信息可能会被放在Pod spec或镜像中；将其放在一个Secret对象中能够更好地控制如何使用它，并降低意外暴露的风险。

用户可以创建Secret，系统也会创建一些Secret。

要使用Secret，Pod需要引用Secret。Secret与Pod一起使用有两种方式：作为mount到一个或多个容器上的 `volume` 的文件，或在为Pod拉取镜像时由kubelet使用。

## Built-in Secrets (内置的Secret)

### Service Accounts Automatically Create and Attach Secrets with API Credentials (服务帐户自动使用API Credential创建和附加Secret)

Kubernetes自动创建包含访问API的凭据的Secret，并自动修改您的Pod，让其使用此类型的Secret。

如果需要，自动创建和API Credential的使用可禁用或覆盖。但是，如果你只是想安全访问apiserver，默认方式是推荐的工作流程。

参阅 [Service Account](#) 文档查看其如何工作的更多信息。

## Creating your own Secrets (创建自己的Secret)

### Creating a Secret Using kubectl create secret (使用kubectl create secret命令创建Secret)

假设Pod需要访问数据库。Pod所使用的用户名和密码在本地机器上的 `./username.txt` 和 `./password.txt` 文件中。

```
1. # Create files needed for rest of example.
2. $ echo -n "admin" > ./username.txt
3. $ echo -n "1f2d1e2e67df" > ./password.txt
```

`kubectl create secret` 命令可将这些文件打包成一个Secret，并在Apiserver上创建对象。

```
1. $ kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
2. secret "db-user-pass" created
```

你可以检查这个Secret是如何创建的：

```

1. $ kubectl get secrets
2. NAME                                TYPE                                DATA    AGE
3. db-user-pass                        Opaque                             2        51s
4.
5. $ kubectl describe secrets/db-user-pass
6. Name:                                db-user-pass
7. Namespace:                           default
8. Labels:                               <none>
9. Annotations:                         <none>
10.
11. Type:                                Opaque
12.
13. Data
14. ===
15. password.txt: 12 bytes
16. username.txt: 5 bytes

```

请注意，默认情况下，`get` 和 `describe` 都不会显示文件的内容。这是为了保护Secret不被意外地暴露。

请参阅 [decoding a secret](#) ，了解如何查看内容。

## Creating a Secret Manually (手动创建Secret)

您也可以先以json或yaml格式在文件中创建一个Secret对象，然后再创建该对象。

每一项都必须以base64进行编码：

```

1. $ echo -n "admin" | base64
2. YWRtaW4=
3. $ echo -n "1f2d1e2e67df" | base64
4. MWYyZDFMmU2N2Rm

```

现在写一个如下所示的Secret对象：

```

1. apiVersion: v1
2. kind: Secret
3. metadata:
4.   name: mysecret
5. type: Opaque
6. data:
7.   username: YWRtaW4=
8.   password: MWYyZDFMmU2N2Rm

```

data字段是一个map。它的key必须符合 `DNS_SUBDOMAIN` ，只是也允许leading dots。value是任意数据，使用base64进行编码。

使用 `kubectl create` 创建Secret：

```
1. $ kubectl create -f ./secret.yaml
2. secret "mysecret" created
```

**Encoding Note:** Secret数据的值在序列化JSON或YAML后，被编码为base64字符串。换行符在这些字符串中无效，必须省略。当在Darwin/OS X上使用 `base64` utility时，用户应避免使用 `-b` 选项来拆分非常长的行。相反，Linux用户应该添加选项 `-w 0` 到 `base64` 命令；或者管道 `base64 | tr -d '\n'`，如果 `-w` 选项不可用。

## Decoding a Secret (解码Secret)

Secret可通过 `kubectl get secret` 命令查看。例如，要查看在上一节中创建的Secret：

```
1. $ kubectl get secret mysecret -o yaml
2. apiVersion: v1
3. data:
4.   username: YWRtaW4=
5.   password: MWYyZDF1MmU2N2Rm
6. kind: Secret
7. metadata:
8.   creationTimestamp: 2016-01-22T18:41:56Z
9.   name: mysecret
10.  namespace: default
11.  resourceVersion: "164619"
12.  selfLink: /api/v1/namespaces/default/secrets/mysecret
13.  uid: cfee02d6-c137-11e5-8d73-42010af00002
14. type: Opaque
```

解码password字段：

```
1. $ echo "MWYyZDF1MmU2N2Rm" | base64 --decode
2. 1f2d1e2e67df
```

## Using Secrets (使用Secret)

Secret可作为数据volume被mount，或作为环境变量暴露，以供Pod中的容器使用。它们也可被系统的其他部分使用，而不会直接暴露在Pod内。例如，它们可保存系统其他部分应该使用的凭据，从而代表你外部系统进行交互。

## Using Secrets as Files from a Pod (使用Secret作为来自Pod的文件)

在Pod中的volume中使用Secret：

1. 创建一个secret或使用现有的secret。多个Pod可引用相同的secret。
2. 修改您的Pod定义，在 `spec.volumes[]` 下添加一个Volume。为Volume任意起个名字，并设置一个 `spec.volumes[].secret.secretName` 字段等于Secret对象的名称。
3. 将 `spec.containers[].volumeMounts[]` 添加到需要使用Secret的每个容器。设置 `spec.containers[].volumeMounts[].readOnly = true`，设置 `spec.containers[].volumeMounts[].mountPath` 为您希望显示Secret的未使用的目录名称。

4. 修改您的镜像和/或命令行，以便程序查找该目录中的文件。 `Secret` `data` map中的每个key都将成为 `mountPath` 下的文件名。

以下是一个在Volume挂载Secret的Pod：

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: mypod
5. spec:
6.   containers:
7.     - name: mypod
8.       image: redis
9.       volumeMounts:
10.      - name: foo
11.        mountPath: "/etc/foo"
12.        readOnly: true
13.   volumes:
14.     - name: foo
15.       secret:
16.         secretName: mysecret
```

你想使用的每个Secret都需要在 `spec.volumes` 中引用。

如果Pod中有多个容器，则每个容器都需要自己的 `volumeMounts` ，但每个Secret只需要一个 `spec.volumes` 。

您可以将多个文件打包成一个Secret，或使用多个Secret，怎么方便怎么玩。

### 将Secret key投影到特定路径

我们还可控制投影Secret key的Volume。可使用 `spec.volumes[].secret.items` 字段来更改每个key的目标路径：

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: mypod
5. spec:
6.   containers:
7.     - name: mypod
8.       image: redis
9.       volumeMounts:
10.      - name: foo
11.        mountPath: "/etc/foo"
12.        readOnly: true
13.   volumes:
14.     - name: foo
15.       secret:
16.         secretName: mysecret
17.         items:
18.           - key: username
19.             path: my-group/my-username
```

这样：

- `username` 这个Secret将会存储在 `/etc/foo/my-group/my-username` 文件中，而非 `/etc/foo/username` 。
- `password` 不被投影

如使用 `spec.volumes[].secret.items`，则只会投影 `items` 中指定的key。如果要投影Secret中的所有key，那么所有key都必须列在 `items` 字段中。所有列出的key必须存在于相应的Secret中。否则，Volume不会被创建。

### Secret files permissions（Secret文件权限）

你也可以指定一个Secret的权限模式位。如不指定，默认使用 `0644`。可指定整个Secret Volume的默认模式，并根据需要覆盖每个key。

例如，你可以指定一个像这样的默认模式：

```

1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: mypod
5. spec:
6.   containers:
7.     - name: mypod
8.       image: redis
9.       volumeMounts:
10.      - name: foo
11.        mountPath: "/etc/foo"
12.   volumes:
13.     - name: foo
14.       secret:
15.         secretName: mysecret
16.         defaultMode: 256

```

// TODO

Then, the secret will be mounted on `/etc/foo` and all the files created by the secret volume mount will have permission `0400` .

Note that the JSON spec doesn't support octal notation, so use the value 256 for 0400 permissions. If you use yaml instead of json for the pod, you can use octal notation to specify permissions in a more natural way.

You can also use mapping, as in the previous example, and specify different permission for different files like this:

## 原文

<https://kubernetes.io/docs/concepts/configuration/secret/>

# Pod Priority and Preemption (Pod优先级和抢占)

本节把priority翻译成优先级，Preemption翻译成抢占。

特性状态：`Kubernetes v1.8` alpha

在Kubernetes 1.8或更高版本中，`Pods` 有priority的概念。priority表示某个Pod相对于其他Pod的重要性。当一个Pod不能被调度时，Scheduler试图抢占（驱逐）较低priority的Pod，从而使调度处于pending状态的Pod成为可能。在未来的Kubernetes版本中，priority还将影响Node上资源的驱逐排序。

注意：抢占不遵守PodDisruptionBudget；有关详细信息，请参阅 [the limitations section](#)。

## How to use priority and preemption (如何使用优先级和抢占)

要在Kubernetes 1.8中使用priority和preemption，请按照如下步骤操作：

1. 启用该功能。
2. 添加一个或多个PriorityClasses。
3. 创建Pod，并将 `PriorityClassName` 设为你所添加的PriorityClasses之一。当然，您无需直接创建Pod；通常可将 `PriorityClassName` 添加到集合对象的Pod模板（如Deployment）。

以下部分提供有关这些步骤的更多信息。

## Enabling priority and preemption (启用优先级和抢占)

默认情况下，Kubernetes 1.8中的Pod priority和preemption功能是禁用的。要启用该功能，请为API Server和Scheduler设置此命令行标志：

```
1. --feature-gates=PodPriority=true
```

并为API Server设置此标志：

```
1. --runtime-config=scheduling.k8s.io/v1alpha1=true
```

启用该功能后，您可以创建 `PriorityClasses` 并创建具有 `PriorityClassName` 设置的Pods。

如果您尝试过该功能后，决定禁用它，那么您必须删除PodPriority命令行标志或将其设置为false，然后重新启动API Server和Scheduler。禁用该功能后，现有的Pod将保留其priority字段，但禁用preemption，priority字段将被忽略；并且，您不能在新Pod中设置PriorityClassName。

# PriorityClass

PriorityClass是一个non-namespaced对象，它定义了从priority class到priority整数值映射。该名称在PriorityClass对象元数据的 `name` 字段中指定。该值在必需的 `value` 字段中指定。value值越大，优先级越高。

PriorityClass对象可以有小于或等于10亿的、任意32位整数值。较大的数字保留给关键系统Pod，这些Pod通常不应该被抢占或被驱逐。 集群管理员应为每个映射创建一个PriorityClass对象。

PriorityClass还有两个可选字段： `globalDefault` 和 `description` 。对于未设置 `PriorityClassName` 的Pod， `globalDefault` 字段表示该PriorityClass的值。只有一个 `globalDefault=true` 的PriorityClass能够存在于系统中。如果没有设置了 `globalDefault` 的PriorityClass，那么，未设置 `PriorityClassName` 的Pod的优先级为零。

`description` 字段是一个任意字符串。这是为了告诉集群用户，他们什么时候该使用这个PriorityClass。

**注1**：如果升级现有集群并启用此功能，则现有Pod的优先级为零。

**注2**：为Pod动态添加 `globalDefault=true` 的PriorityClass，不会更改现有Pod的优先级。此类PriorityClass的值仅用于添加PriorityClass后创建的Pod。

TODO 注2 原文有点歧义，待测试、改进。

**注3**：如果您删除了PriorityClass，则引用该PriorityClass名称的现有Pod将保持不变，但您无法继续创建引用该PriorityClass名称的Pod。

## Example PriorityClass

```
1. apiVersion: scheduling.k8s.io/v1alpha1
2. kind: PriorityClass
3. metadata:
4.   name: high-priority
5. value: 1000000
6. globalDefault: false
7. description: "This priority class should be used for XYZ service pods only."
```

## Pod priority

创建一个或多个PriorityClass之后，可创建Pod，并在其spec中指定其中一个PriorityClass名称。priority admission controller使用 `priorityClassName` 字段并填充该字段的整数值。如果未找到priority class，则Pod被拒绝。

如下YAML是一个Pod的定义，该Pod使用上述示例中所创建的PriorityClass。priority admission controller检查spec，并将Pod的priority设为1000000。

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
```

```

4.   name: nginx
5.   labels:
6.     env: test
7. spec:
8.   containers:
9.   - name: nginx
10.    image: nginx
11.    imagePullPolicy: IfNotPresent
12.    priorityClassName: high-priority

```

## Preemption ( 抢占 )

当Pods被创建时，它们会进入队列并等待被调度。Scheduler从队列中选择一个Pod，并尝试将其调度到Node上。如果未能找到满足Pod所有要求的Node，则为处于pending状态的Pod触发preemption逻辑。下面我们称这个处于pending状态的Pod为P。抢占逻辑尝试找到一个Node，在删除一个或多个优先级低于P的Pod后，就能在该Node上调度P。如果能找到这样的Node，则会从Node中删除一个或多个优先级较低的Pod。在其删除后，P就能在Node上调度。

## Limitations of preemption (alpha version)

### Starvation of preempting Pod

当Pod被抢占时，受害者获得了 [graceful termination period](#) 。他们有graceful termination period所定义的时长完成工作并退出。如果无法优雅关闭，就会被杀死。这个graceful termination period会在scheduler抢占Pod的时间点 与 能在Node (N) 上调度pending Pod (P) 的时间点 之间创建时间间隔。在此期间，Scheduler会调度其他pending的Pod。当受害者退出或终止时，Scheduler会尝试调度在待处理队列中的Pod，并且在Scheduler将P调度到N之前，可能会将其中的一个或多个考虑并调度到N。在这种情况下，所有的受害者退出，Pod P不再适用于Node N。因此，Scheduler将必须抢占Node N或其他Node上的其他Pod，以便能够调度P。对于第二次和后续的抢占，这种情况可能会再次重复，P可能会在一定时间内无法得到调度。这种情况可能会导致各种集群中的问题，在Pod创建速率较高的集群中尤其成问题。

我们将在Pod preemption的Beta版本中解决这个问题。 [provided here \( 在这里提供了 \)](#) 我们计划实施的解决方案。

## PodDisruptionBudget is not supported ( 不支持 PodDisruptionBudget )

[Pod Disruption Budget \(PDB\)](#) 允许应用程序所有者限制从voluntary disruptions中资源下降的Pod的数量。

然而，在选择preemption的受害者时，alpha版本的preemption并不遵守PDB规则。我们计划在beta版本中添加PDB支持，但即使在beta版中，也只是尽可能地遵守PDB。Scheduler将尝试寻找那些 PDB不会被preemption违反的 受害者，但如果未发现这样的受害者，preemption仍然会发生，优先级较低的Pod将被删除，尽管这会违反PDB。

译者按：PDB简介：<http://blog.csdn.net/horsefoot/article/details/76496496>



## Inter-Pod affinity on lower-priority Pods (低优先级Pod的Inter-Pod 亲和性)

在版本1.8中，只有当该问题的答案为yes时，才将Node视为可抢占：“如果从Node中删除所有优先级低于pending Pod的Pod，就能在该Node上调度pending Pod吗？”

注意：抢占并不一定会清除所有优先级较低的Pod。如果删除部分优先级较低的Pod就能调度pending Pod，那么只有一部分优先级较低的Pod会被删除。即使如此，上述问题的答案也必须是yes。否则，则Node不被视为可抢占。

如果pending Pod具有与 Node上一个或多个较低优先级Pod 的inter-pod affinity，则在那些较低优先级Pod 缺失的情况下，不能满足 inter-Pod affinity规则。在这种情况下，Scheduler不会抢占Node上的任何Pod。相反，它寻找另一个Node。 Scheduler可能会找到一个合适的Node，又或者它找不到合适的Node。不能保证 pending Pod能够被调度。

我们可能会在将来的版本中解决这个问题，但还没有一个明确的计划。我们不会认为它是Beta或GA的阻止者。部分原因是找到满足所有inter-Pod affinity规则的较低优先级的Pod集合，计算代价比较高，并且对抢占逻辑增加了相当大的复杂性。此外，即使preemption保留优先级较低的Pod来满足inter-Pod affinity，较低优先级的Pod也可能被其他Pod稍后抢占，这消除了遵守inter-Pod affinity的复杂逻辑的好处。

我们针对此问题的建议解决方案是仅在相同或更高优先级的Pod上创建inter-Pod affinity。

## Cross node preemption (跨Node抢占)

假设Node N正被考虑用于抢占，以便可在N上调度pending Pod P。只有当另一个Node上的Pod被抢占时，P才可能在N上可行。这里有一个例子：

- Node N正在考虑Pod P。
- Pod Q在与Node N相同zone中的另一个Node上运行。
- Pod P与Pod Q具有anti-affinity。
- Pod P和其他Pod之间没有anti-affinity的情况。
- 为在Node N上安排Pod P，Pod Q应该被抢占，但是Scheduler不执行跨节点抢占。 所以，Pod P在Node N上被视为unschedulable的。

如果Pod Q从其Node中删除，那么久不违反anti-affinity了，并且Pod P可能会在Node N上进行调度。

如果我们找到某种性能合理的算法，可能会考虑在将来的版本中添加跨Node抢占。在这一点上我们不能作任何承诺，并且跨Node抢占不会被认为是Beta或GA的阻止者。

## 原文

<https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/>

# Service

Kubernetes `Pods` 是有生命周期的，它们产生和死亡，一旦死亡就不会复活。`ReplicationControllers` 能动态创建和销毁 `Pod`（例如，当扩容、缩容或在 `rolling updates` 时）。尽管每个 `Pod` 有自己的IP，但这些IP可能是会变化的。这导致一个问题：如果一些 `Pod`（称为backend）为Kubernetes群集中的其他 `Pod`（称为frontend）提供功能，那么frontend如何找到，并一直能找到backend呢？

译者注：下面将backend译为后端，frontend译为前端。

可用 `Services` 来解决该问题。

Kubernetes `Service` 是一个抽象，它定义了一组逻辑 `Pod` 和一个访问它们的策略——有时称为微服务。`Service` 所关联的一组 `Pod`（通常）由 `Label Selector` 决定（详见下文，为什么您可能需要没有选择器的 `Service`）。

例如，假设一个图片处理的后端有三个正在运行的副本。这些副本是可替代的——前端不关心他们使用哪个后端副本。虽然构成后端的实际 `Pod` 可能会改变，但前端客户端不应该也没必要跟踪自己的后端列表（例如：知道后端副本列表的地址等待）。`Service` 抽象可实现这种解耦。

对于Kubernetes-native应用程序，Kubernetes提供了一个简单的 `Endpoint` API，只要 `Service` 中的 `Pod` 发生变化，它将被更新。对于non-native应用程序，Kubernetes为Service提供了一个基于虚拟IP的网桥，该网桥可重定向到后端 `Pod`。

## 定义Service

Kubernetes中的 `Service` 是一个REST对象，类似于 `Pod`。像所有的REST对象一样，`Service` 定义可被POST到apiserver，从而创建一个新的实例。例如，假设您有一组 `Pod`，每个 `Pod` 都暴露端口9376，并携带标签 `"app=MyApp"`。

```
1. kind: Service
2. apiVersion: v1
3. metadata:
4.   name: my-service
5. spec:
6.   selector:
7.     app: MyApp
8.   ports:
9.     - protocol: TCP
10.      port: 80
11.      targetPort: 9376
```

使用此文件，即可创建一个名为“my-service”的新 `Service` 对象，该对象会代理那些使用TCP端口9376，并且有 `"app=MyApp"` 标签的Pod，该 `Service` 还会被分配一个IP地址（有时称为“Cluster IP”），它会被服务的代理使用（见下文）。`Service` 的选择器将会被持续评估，处理的结果会被POST到一个名为“my-service”的 `Endpoint` 对象。

请注意，`Service` 可将传入端口 `port` 映射到任意 `targetPort` 端口。默认情况下，`targetPort` 将被设置

为与 `port` 字段相同的值。也许更有趣的是，`targetPort` 可以是字符串，指向到是后端 `Pod` 端口的名称。拥有该名称的端口的实际端口在每个后端 `Pod` 中可能并不相同。这为 `Services` 提供了很大的灵活性。例如，您可以在后端软件的下一个版本中更改该Pod所暴露的端口，而不会影响客户端的调用。

Kubernetes `Service` 支持 `TCP` 和 `UDP` 协议。默认值为 `TCP`。

## Services without selectors (不带选择器的Service)

Service通常抽象了 `Pod` 的访问，但也可抽象其他类型的后端。例如：

- 您希望在生产环境中使用外部数据库集群，但在测试中，您将使用自己的数据库。
- 您希望将Service指向另一个 `Namespace` 的Service或其他集群中的Service。
- 您正在将工作负载迁移到Kubernetes，部分后端运行在Kubernetes之外。

在以上这些情况下，您可以定义不带选择器的Service：

```
1. kind: Service
2. apiVersion: v1
3. metadata:
4.   name: my-service
5. spec:
6.   ports:
7.     - protocol: TCP
8.       port: 80
9.       targetPort: 9376
```

因为该Service没有选择器，所以不会创建相应的 `Endpoint` 对象。可手动将Service映射到指定的 `Endpoint`：

```
1. kind: Endpoints
2. apiVersion: v1
3. metadata:
4.   name: my-service
5. subsets:
6.   - addresses:
7.     - ip: 1.2.3.4
8.     ports:
9.       - port: 9376
```

注意：Endpoint IP不能是loopback（127.0.0.0/8），link-local（169.254.0.0/16）或link-local multicast（224.0.0.0/24）。

译者按：loopback、link-local、link-local multicast拓展阅读：<https://4sysops.com/archives/ipv6-tutorial-part-6-site-local-addresses-and-link-local-addresses/>

访问不带选择器的 `Service` 与访问有选择器Service相同。流量将被路由到用户定义的端点（在本例中为 `1.2.3.4:9376`）。

ExternalName Service一种特殊的Service：它不带选择器，也不定义任何端口或端点。相反，它可以作为将别名返回到外部Service的一种方式。

```

1. kind: Service
2. apiVersion: v1
3. metadata:
4.   name: my-service
5.   namespace: prod
6. spec:
7.   type: ExternalName
8.   externalName: my.database.example.com

```

当查找主机 `my-service.prod.svc.CLUSTER` 时，集群DNS服务将会返回一条值为 `my.database.example.com` 的 `CNAME` 记录。访问这样的Service与访问其他Service的工作方式相同，唯一的区别是重定向发生在DNS级别，并且不会发生代理或转发。如果您以后决定将数据库移动到集群中，可启动对应的Pod，添加适当的选择器或Endpoint并更改服务的 `type`。

## Virtual IPs and service proxies (虚拟IP与Service代理)

Kubernetes集群中的每个Node都运行着一个 `kube-proxy`。`kube-proxy` 负责为 `Service` 实现一个虚拟IP形式，`ExternalName` 类型的Service除外。在Kubernetes v1.0中，使用userspace模式进行代理。在Kubernetes v1.1中，添加了iptables代理，但不是默认的操作模式。从Kubernetes v1.2开始，默认使用iptables代理。

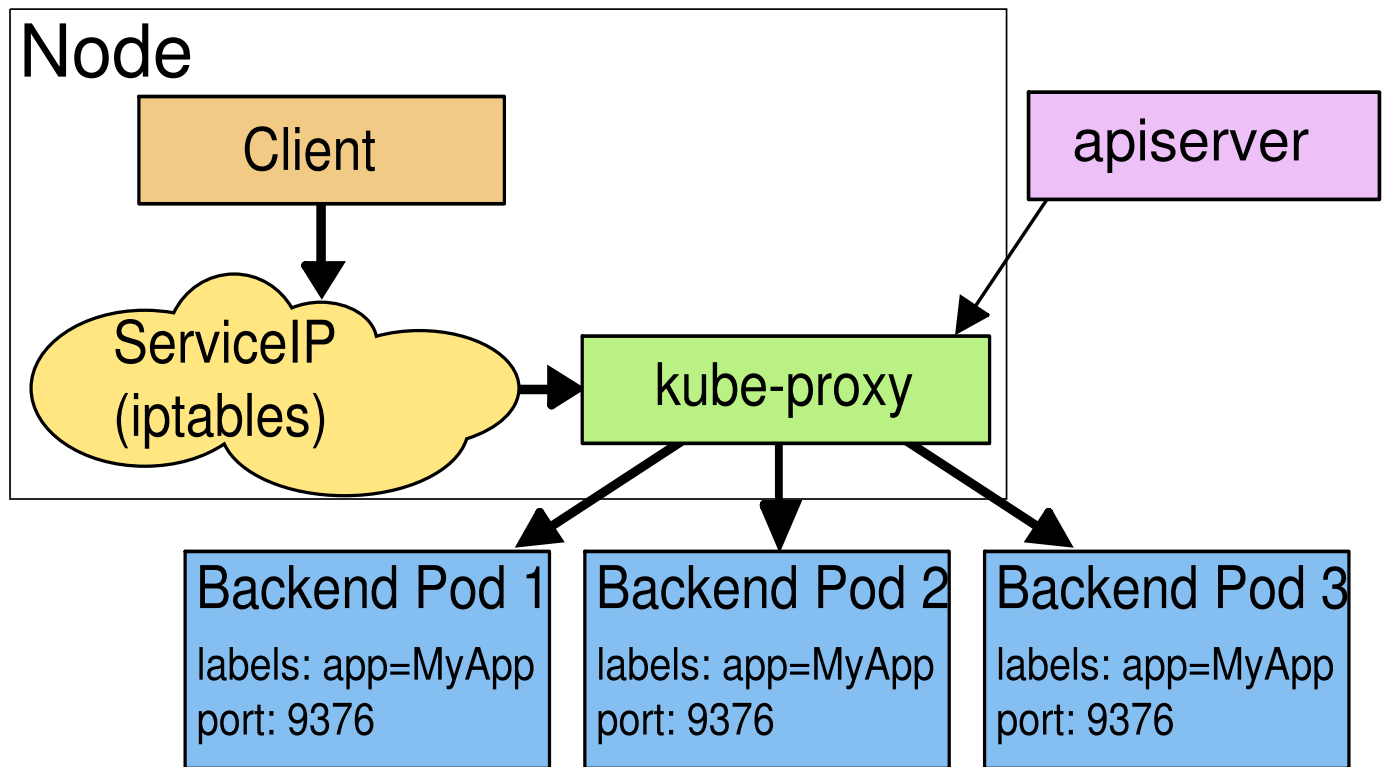
从Kubernetes v1.0起，`Services` 是“4层”(TCP / UDP over IP)结构。在Kubernetes v1.1中，添加了 `Ingress` API (beta) 来表示“7层”(HTTP)服务。

## Proxy-mode: userspace (代理模式: userspace)

在这种模式下，`kube-proxy`会监视Kubernetes Master对 `Service` 和 `Endpoints` 对象的添加和删除。对于每个 `Service`，它将在本地Node上打开一个端口（随机选择），记为“代理端口”。与该“代理端口”的任何连接将被代理到 `Service` 的其中一个后端 `Pod`（由 `Endpoint` 报告）。使用哪个后端 `Pod` 是根据Service的 `SessionAffinity` 决定的。最后，它安装iptables规则，捕获到 `Service` 的 `clusterIP:Port` 的流量，并将流量重定向到代理端口，代理端口再代理后端 `Pod`。

这样，任何绑定 `Service IP:端口` 的流量都被代理到合适的后端，而客户端不需要知道有关Kubernetes或 `Service` 或 `Pod` 的任何内容。

默认情况下，后端使用算法轮询选择Pod。要想实现基于客户端IP的会话亲和性 (Client-IP based session affinity)，可将 `service.spec.sessionAffinity` 设置为 `"ClientIP"`（默认为 `"None"`）；此时，可通过 `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` 字段设置最大会话粘性时间（默认值为“10800”）

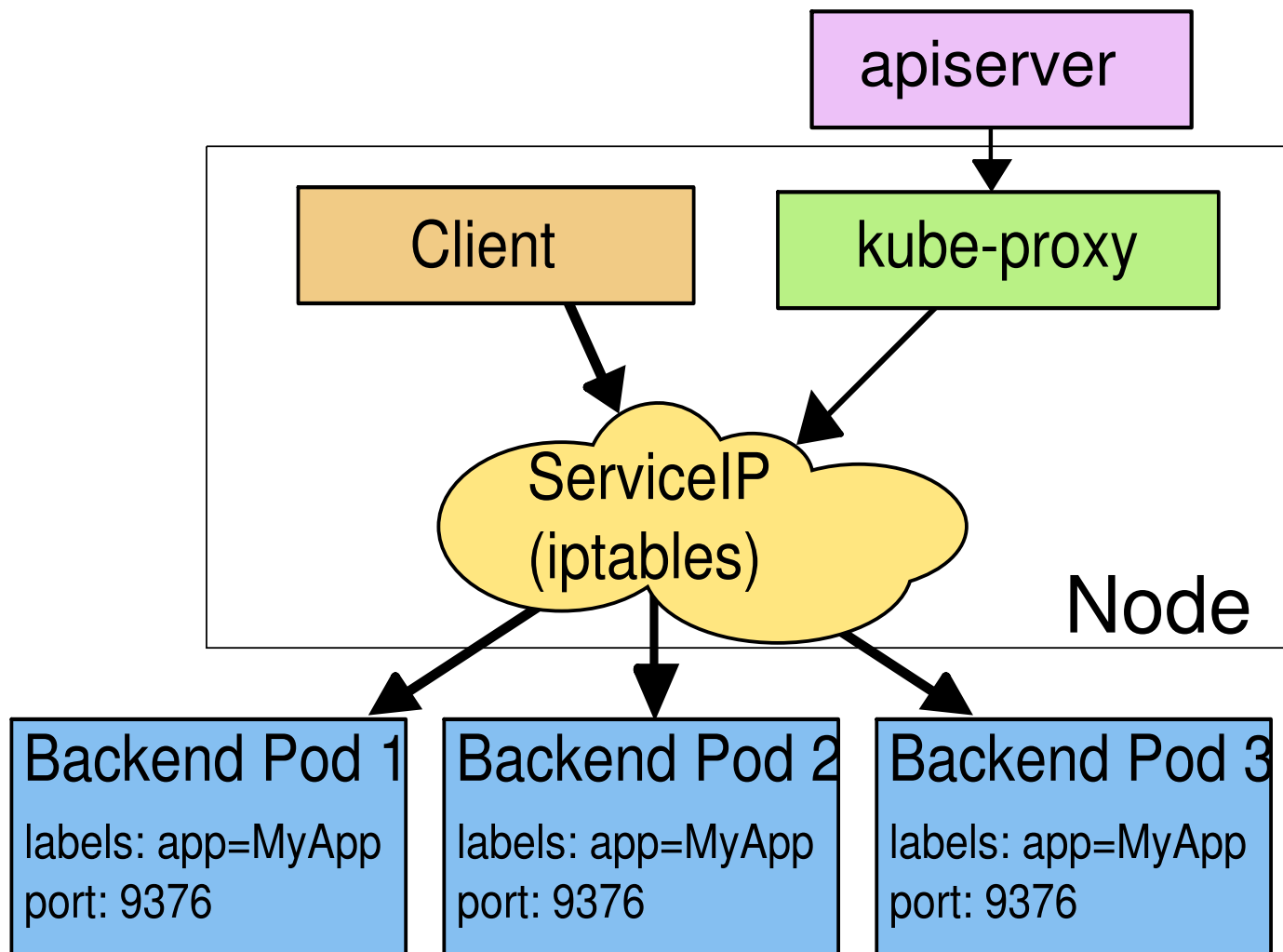


## Proxy-mode: iptables (代理模式: iptables)

在这种模式下, kube-proxy 会监视 Kubernetes Master 对 `Service` 和 `Endpoints` 对象的添加和删除。它将对每个 `Service` 安装 iptables 规则, 从而捕获到 `Service` 的 `clusterIP:Port` 的流量, 并将流量重定向到 `Service` 的其中一个后端 Pod。对于每个 `Endpoint` 对象, 它会安装选择后端 `Pod` 的 iptables 规则。

默认情况下, 后端使用随机算法的选择 Pod。要想实现基于客户端 IP 的会话亲和性 (Client-IP based session affinity), 可将 `service.spec.sessionAffinity` 设为 `"ClientIP"` (默认为 `"None"`) ; 此时, 可通过 `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` 字段设置最大会话粘性时间 (默认值为 `"10800"`) 。

与用户 `userspace proxy` 模式一样, 最终, 任何绑定 `Service IP:端口` 的流量都被代理到合适的后端, 而客户端无需有关 Kubernetes 或 `Services` 或 `Pods` 的任何内容。这种模式应该比 `userspace proxy` 模式更快更可靠。但是, 与 `userspace proxy` 模式不同, 如果最初选择的 Pod 不响应, iptables 代理不能自动重试请求另一个 Pod, 因此它依赖 `readiness probes` 。



译者按：

1. 简而言之，就是userspace方式，使用kubeproxy来转发；而iptables模式中，kube-proxy只生成相应的iptables规则，转发由iptables实现。
2. 拓展阅读：<http://blog.csdn.net/liyingke112/article/details/76022267>

## Multi-Port Services (多端口Service)

某些 `Service` 可能需要暴露多个端口。对于这种情况，Kubernetes支持 `Service` 对象上定义多个端口。当使用多个端口时，您必须提供所有端口名称，防止Endpoint产生歧义。 例如：

```

1. kind: Service
2. apiVersion: v1
3. metadata:
4.   name: my-service
5. spec:
6.   selector:
7.     app: MyApp
8.   ports:
9.   - name: http
10.    protocol: TCP
11.    port: 80
12.    targetPort: 9376
  
```

```

13.   - name: https
14.     protocol: TCP
15.     port: 443
16.     targetPort: 9377

```

## Choosing your own IP address ( 选择自己的IP地址 )

**Service** 创建时，可指定自己的Cluster IP。可通过 `spec.clusterIP` 字段设置Cluster IP。例如，如果您想替换一条现有DNS条目，或者有一个配置了固定IP、很难重新配置的遗留系统。用户选择的IP地址必须是合法的IP地址，并且该IP在 `service-cluster-ip-range` CIDR范围内，该范围由API Server的标识指定。如果IP不合法，apiserver将返回一个422 HTTP状态码，表示该值不合法。

## Why not use round-robin DNS? ( 为什么不使用轮询DNS ? )

为什么要使用虚拟IP，而非标准的round-robin DNS呢？有几个原因：

- 长久以来，DNS库都未能实现DNS TTL并缓存域名查询结果。
- 许多应用只执行一次DNS查询，并缓存结果。
- 即使应用程序和DNS库都进行了适当的重新解析，客户端重新解析DNS造成的负载难以管理。

我们试图阻止用户吃力不讨好的事情。也就是说，如果有足够的人要求该功能，我们也可实现该方案作为替代方案。

## Discovering services ( 服务发现 )

Kubernetes主要支持两种方式找到 **Service** ：环境变量和DNS。

## Environment variables ( 环境变量 )

当 **Pod** 在 **Node** 上运行时，kubelet会为每个活动的 **Service** 添加一组环境变量。它支持 **Docker links compatible** 变量（请参阅 `makeLinkVariables` ）和更简单的 `{SVCNAME}_SERVICE_HOST` 以及 `{SVCNAME}_SERVICE_PORT` 变量，其中服务名称是大写的，中划线将会转换为下划线。

例如，暴露TCP端口6379，并已分配 Cluster IP地址 `10.0.0.11` 的服务 `redis-master` ，将会产生以下环境变量：

```

1. REDIS_MASTER_SERVICE_HOST=10.0.0.11
2. REDIS_MASTER_SERVICE_PORT=6379
3. REDIS_MASTER_PORT=tcp://10.0.0.11:6379
4. REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
5. REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
6. REDIS_MASTER_PORT_6379_TCP_PORT=6379
7. REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11

```

这意味着有顺序的需求 — **Pod** 想要访问的任何 **Service** 必须在该 **Pod** 之前创建，否则环境变量将不会被赋值。DNS没有这个限制。



## DNS

DNS server是一个可选的（虽然强烈推荐）`cluster add-on`。DNS server监视创建新 `Services` 的 Kubernetes API，并为每个Service创建一组DNS记录。如果在整个集群中启用了DNS，那么所有的 `Pod` 都应该能够自动进行 `Service` 名称解析。

例如，如果在Kubernetes `my-ns` 这个 `Namespace` 中，有一个名为 `my-service` 的Service，就会创建 `my-service.my-ns` 的DNS记录。存在于 `"my-ns"` 这个 `Namespace` 中的 `Pod` 可通过 `my-service` 这个名称查找来找到这个Service。存在于其他 `Namespaces` 的 `Pod` 将必须使用 `my-service.my-ns`。使用服务名称查询的结果是Cluster IP。

Kubernetes还支持命名端口的DNS SRV (Service) 记录。如果 `my-service.my-ns` 这个 `Service` 有一个名为 `http`、协议为 `TCP` 的端口，则可以使用 `_http._tcp.my-service.my-ns` 的DNS SRV查询来发现 `http` 这个端口的端口号。

Kubernetes DNS Server是访问 `ExternalName` 类型Service的唯一方法。[DNS Pods and Services](#) 提供了更多信息。

## Headless Service

有时你不需要或不想使用负载均衡，以及一个单独的Service IP。在这种情况下，您可以通过将Cluster IP (`spec.clusterIP`) 指定 `None` 来创建Headless Service。

此选项允许开发人员通过允许由他们实现自己的服务发现方式，从而减少与Kubernetes系统的耦合。应用程序仍可使用自注册的模式，并且可轻松地将此适配器用于其他的服务发现系统。

对于这类 `Services`，不会分配Cluster IP，kube-proxy也不会处理它们，并且平台也不会对它们进行负载均衡或代理。如何自动配置DNS，取决于Service是否定义了选择器。

### With selectors (有选择器)

对于定义了选择器的Headless Services，Endpoint Controller会在API中创建 `Endpoint` 记录，并修改DNS配置以返回 `Pods` A记录（地址），A记录直接指向Service后端的Pod。

### Without selectors (无选择器)

对于没有选择器的Headless Service，Endpoint Controller不会创建 `Endpoint` 记录。但是，DNS系统会寻找并配置：

- `ExternalName` 类型Service的CNAME记录。
- 所有与Service共享名称的 `Endpoints` 的记录，以及其他类型。

## Publishing services - service types (暴露 Service—服务类型)

对于应用程序的某些部分（例如前端），您可能希望将服务暴露到一个外部（集群外部）IP中。



Kubernetes `ServiceTypes` 允许您指定所需的Service类型。默认是 `ClusterIP`。

`Type` 值及其行为如下：

- `ClusterIP`：通过集群内部的IP暴露Service，Service只能从集群内部访问。这是默认的 `ServiceType`。
- `NodePort`：在每个Node的IP的静态端口上暴露该Service。NodePort Service会路由到 `ClusterIP` Service。NodePort类型的集群外的请求可通过 `<NodeIP>:<NodePort>` 与访问一个 `NodePort` 类型的Service。
- `LoadBalancer`：使用云提供商提供的负载均衡器外部暴露服务。负载均衡器将会路由 `NodePort` 和 `ClusterIP` 类型的Service。
- `ExternalName`：通过返回带有其值的 `CNAME` 记录，将Service映射到 `externalName` 字段的内容（例如 `foo.bar.example.com`）。没有任何代理设置。这需要 `kube-dns` 1.7或更高版本。

## Type NodePort (NodePort类型)

如果将 `type` 字段设为 `NodePort`，则Kubernetes Master将会从给定的配置范围（默认30000-32767）分配一个端口，并且每个Node都会将该端口（每个Node上的同一端口）代理到 `Service`。该端口将在 `Service` 的 `spec.ports[*].nodePort` 字段中报告。

如果您想要一个特定的端口号，可在 `nodePort` 字段的值，系统会为您分配该端口，否则API事务将会失败（即：您需要自己关心可能的端口冲突）。您指定的值必须在Node端口的配置范围内（默认30000-32767）。

这使开发人员可自由设置自己的负载均衡器，配置Kubernetes不完全支持的环境，甚至直接暴露一个或多个Node的IP。

请注意，此Service将在 `<NodeIP>:spec.ports[*].nodePort` 和 `spec.clusterIp:spec.ports[*].port` 可见。

## Type LoadBalancer (LoadBalancer类型)

略

## External IPs (外部IP)

如果外部IP路由到一个或多个集群Node上，则可在这些 `externalIPs` 上暴露Kubernetes Service。使用外部IP：端口（作为目标IP）进入集群的流量，将被路由到其中一个Service Endpoint。`externalIPs` 不由Kubernetes管理，这属于集群管理员的职责。

在ServiceSpec中，`ServiceTypes` 的Service都可指定 `externalIPs`。如下示例中，`my-service` 可通过 `80.11.12.10:80`访问（`externalIP:port`）

```
1. kind: Service
2. apiVersion: v1
3. metadata:
4.   name: my-service
5. spec:
6.   selector:
7.     app: MyApp
8.   ports:
```

```

9.   - name: http
10.   protocol: TCP
11.   port: 80
12.   targetPort: 9376
13.   externalIPs:
14.   - 80.11.12.10

```

## Shortcomings ( 缺点 )

使用VIP的userspace proxy可在中小规模的情况下工作，但无法扩容到有数千个Service的非常大的集群。有关详细信息，请参阅 [the original design proposal for portals \( 门户网站的原始设计方案 \)](#) 。

使用userspace proxy会隐藏访问 `Service` 的数据包的源IP。这使得某些防火墙无法实现。iptables proxier不会隐藏群集内的源IP，但它仍会影响到使用负载均衡器或node-port的客户端。

译者按：但它仍会影响到使用负载均衡器或node-port的客户端是什么意思？

`Type` 字段支持嵌套——每个级别添加之前级别的功能。不会严格要求所有云提供商这么玩（例如Google Compute Engine不需要分配 `NodePort` 来使 `LoadBalancer` 工作，但AWS），但当前API是强制要求的。

## Future work ( 未来的工作 )

将来，代理策略可能会比简单的round-robin均衡策略更加的细致，例如Master选举或分片。`Services` 也将会拥有“真正的”负载均衡器，在这种情况下，VIP只会将数据包传输到负载均衡器上。

我们打算改进对L7（HTTP）`Service` 的支持。

我们打算为 `Service` 提供更灵活的入口模式，包括当前的 `ClusterIP`、`NodePort` 和 `LoadBalancer` 模式等。

## The gory details of virtual IPs ( 虚拟IP血腥的细节 )

对于那些只想使用 `Service` 的人来说，之前的信息应该已经足够。不过，幕后还有很多值得我们理解的事情。

## Avoiding collisions ( 避免冲突 )

Kubernetes的主要哲学之一是，用户不应被暴露在那些“可能会导致他们操作失败，但又不是他们的过错”的场景。在这种情况下，我们来检查网络端口——用户不应该必须选择端口，如果该选择可能与其他用户发生冲突。那就是“隔离失败”。

译者按：笔者理解的“隔离失败”，指的是尽管两个用户之间是隔离的，但仍可能发生失败。

为了允许用户为其 `Service` 选择端口，我们必须确保两个 `Service` 不会相冲突。我们通过为每个 `Service` 分配自己的IP地址的方式来实现。

为了确保每个Service接收到唯一的IP，内部分配器将在创建每个Service之前，以原子方式更新etcd中的全局分

配映射表。映射表对象必须存在于服务的注册表中，从而获取IP，否则创建将失败，并显示无法分配IP的消息。后台Controller负责创建该映射表（老版本使用的是内存锁），并检查由于管理员干预而造成的无效分配，并清理那些已经分配但尚无Service使用的IP。

## IPs and VIPs

与实际路由到固定目的地的 Pod IP地址不同，Service IP实际上不由单个主机来应答。相反，我们使用 iptables（Linux中的数据包处理逻辑）定义虚拟IP，从而根据需要透明地进行重定向。当客户端连接到VIP时，其流量将自动传输到适当的端点。Service的环境变量和DNS实际上是根据 Service 的VIP和端口填写的。

我们支持两种代理模式—userspace和iptables，运行方式略有不同。

## Userspace

作为示例，考虑上面提到的图像处理应用。创建后端 Service，Kubernetes Master分配虚拟IP地址，例如 10.0.0.1。假设 Service 端口为1234，Service 由集群中的所有 kube-proxy 实例观察。当kube-proxy看到一个新的 Service，它会随机打开一个新的端口，建立从VIP到这个新端口的iptables重定向，然后开始接受连接。

当客户端连接到VIP时，iptables规则开始起作用，并将数据包重定向到 Service proxy 的端口。Service proxy 选择一个后端节点，并开始代理从客户端到后端的流量。

这意味着 Service 所有者可选择他们想要的任何端口，没有冲突的风险。客户端可简单地连接到IP和端口，而无需知道它们正在访问哪个 Pod。

## Iptables

再次考虑上述图像处理应用。创建后端 Service，Kubernetes Master分配虚拟IP地址，例如 10.0.0.1。假设 Service 端口为1234，Service 由集群中的所有 kube-proxy 实例观察。当代理看到一个新的 Service 时，它会安装一系列从VIP重定向到per-Service 的iptables规则。per-Service 的规则链链接到per-Endpoint 规则，per-Endpoint 规则会重定向（目标NAT）到后端。

当客户端连接到VIP时，iptables规则就会起作用。选择一个后端（基于会话亲和性或随机），并将数据包重定向到后端。与userspace proxy不同，数据包不会复制到userspace，因此无需运行kube-proxy才能使VIP工作，客户端IP也不会被更改。

当流量通过node-port或负载均衡器进入时，同样的基本流程就会执行，尽管在这种情况下，客户端IP会被改变。

## API Object

Service是Kubernetes REST API中的顶级资源。有关API对象的更多详细信息，请访问：[Service API object](#)。

## For More Information

阅读 [Connecting a Front End to a Back End Using a Service](#)。

## 原文

---

<https://kubernetes.io/docs/concepts/services-networking/service/>

# Ingress Resources

## 术语

在本文中，您将看到有些术语，这些术语在其他地方往往被交叉使用。这可能会导致混淆。本节试图澄清它们。

- **Node**：Kubernetes集群中的单个虚拟机或物理机。
- **Cluster**：一组位于互联网防火墙之后的Node，这是Kubernetes管理的主要计算资源。
- **Edge router**：为集群强制执行防火墙策略的路由器。这可能是由cloud provider或物理硬件管理的网关。
- **Cluster network**：一组逻辑或物理链接，可根据 [Kubernetes networking model](#) 实现集群内的通信。集群网络的示例包括诸如 [flannel](#) 的Overlay网络或诸如 [OVS](#) 的SDN网络。
- **Service**：Kubernetes [Service](#) 使用Label Selector识别一组Pod。除非另有说明，否则Service假定在集群网络内仅可通过虚拟IP进行路由。

## What is Ingress?

通常，Service和Pod的IP只能在集群网络内部访问。所有到达edge router的所有流量会被丢弃或转发到其他地方。在概念上，这可能看起来像：

```

1.      internet
2.      |
3.  -----
4.  [ Services ]
```

Ingress是允许入站连接到达集群Service的规则集合。

```

1.      internet
2.      |
3.  [ Ingress ]
4.  --|-----|--
5.  [ Services ]
```

可配置Ingress，从而提供外部可访问的URL、负载均衡流量、SSL、提供基于名称的虚拟主机等。用户通过POST Ingress资源到API Server的方式来请求Ingress。 [Ingress controller](#) 负责实现Ingress，通常使用负载均衡器，也可配置edge router或其他前端，这有助于以HA方式处理流量。

## Prerequisites（先决条件）

在开始使用Ingress资源之前，您应该了解一些事情。 Ingress是一个Beta资源，在1.1之前的任何Kubernetes版本中都不可用。您需要一个Ingress Controller才能满足Ingress，单纯创建Ingress不起作用。

GCE/GKE会在Master上部署一个Ingress Controller。您可以在Pod中部署任意数量的自定义Ingress Controller。您必须使用适当的类去为每个Ingress添加Annotation，如此 [here](#) 和 [here](#) 所示。

确保您看过此Controller的 [beta limitations](#) 。在GCE/GKE以外的环境中，您需要 [deploy a](#)

`controller` 为pod。

## The Ingress Resource

最简单的Ingress可能如下所示：

```
1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: test-ingress
5.   annotations:
6.     ingress.kubernetes.io/rewrite-target: /
7. spec:
8.   rules:
9.   - http:
10.     paths:
11.     - path: /testpath
12.       backend:
13.         serviceName: test
14.         servicePort: 80
```

如果您尚未配置*Ingress Controller*，那么将其发送到API Server将不起作用。

**1-6行**：与所有其他Kubernetes配置一样，Ingress需要 `apiVersion`、`kind` 和 `metadata` 字段。有关使用配置文件的信息，请参阅 [deploying applications](#)、[configuring containers](#)、[managing resources](#) 以及 [ingress configuration rewrite](#)。

**7-9行**：Ingress `spec` 有配置负载均衡器或代理服务器所需的所有信息。最重要的是，它包含了一个匹配所有入站请求的rule列表。目前，Ingress资源只支持http rule。

**10-11行**：每个http rule包含以下信息：host（例如：foo.bar.com，本例中默认为\*）；path列表（例如：/testpath），每个path都有关联的backend（比如test:80）。在负载均衡器将流量导入到backend之前，host和path都必须都与入站请求的内容匹配。

**行12-14**：backend是 [services doc](#) 描述的 `service:port` 组合。Ingress流量通常直接发送到与backend匹配的端点。

**全局参数**：简单起见，Ingress示例没有全局参数，要查看资源的完整定义，请阅读 [API reference](#)。可指定全局缺省backend，在所有请求都无法与spec中path匹配的情况下，会发送给缺省backend。

## Ingress controllers

为使Ingress资源正常工作，集群必须运行Ingress Controller。这与其他类型的Controller不同，它们通常作为 `kube-controller-manager` 二进制文件的一部分运行，通常作为集群创建的一部分自动启动。您需要选择最适合您集群的Ingress Controller实现，或自己实现一个。我们目前支持和维护 [GCE](#) 和 [nginx](#) Controller。

## Before you begin

以下文档描述了通过Ingress资源暴露的一组跨平台功能。理想情况下，所有Ingress Controller都应符合此规范，但还没有实现。GCE和Nginx Controller的文档分别 [here](#) 和 [here](#)。确保您查看**Controller**的文档，以便您了解每个文档的注意事项。

## Types of Ingress

### Single Service Ingress

现有的Kubernetes概念允许您暴露单个Service（请参阅 [alternatives](#)），但是您也可通过Ingress来暴露，通过指定不带rule的默认backend来实现。

```
1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: test-ingress
5. spec:
6.   backend:
7.     serviceName: testsvc
8.     servicePort: 80
```

使用 `kubectl create -f` 创建它，即可看到：

```
1. $ kubectl get ing
2. NAME                RULE          BACKEND          ADDRESS
3. test-ingress        -             testsvc:80       107.178.254.228
```

其中，`107.178.254.228` 是由Ingress Controller为此Ingress分配的IP。`RULE` 列显示发送到IP的所有流量都被转发到在BACKEND列所列出的Kubernetes服务。

### Simple fanout

如上文所述，kubernetes Pod的IP只能在集群网络可见，所以我们需要一些边缘组件，边缘接受Ingress流量并将其代理到正确的端点。该组件通常是高可用的负载均衡器。Ingress允许您将负载均衡器的数量降至最低，例如：如果你想创建类似如下的配置：

```
1. foo.bar.com -> 178.91.123.132 -> / foo    s1:80
2.                                     / bar    s2:80
```

那么将需要一个Ingress，例如：

```
1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: test
5.   annotations:
6.     ingress.kubernetes.io/rewrite-target: /
7. spec:
```

```

8.   rules:
9.     - host: foo.bar.com
10.    http:
11.      paths:
12.        - path: /foo
13.        backend:
14.          serviceName: s1
15.          servicePort: 80
16.        - path: /bar
17.        backend:
18.          serviceName: s2
19.          servicePort: 80

```

当您使用 `kubectl create -f` 创建Ingress时:

```

1. $ kubectl get ing
2. NAME      RULE      BACKEND    ADDRESS
3. test      -
4.           foo.bar.com
5.           /foo      s1:80
6.           /bar      s2:80

```

只要存在Service (s1, s2), Ingress Controller就会提供满足Ingress的特定负载均衡器的实现。完成这个步骤后, 您将在Ingress的最后一列看到负载均衡器的地址。

## Name based virtual hosting (基于名称的虚拟主机)

Name-based virtual hosts为相同IP使用多个主机名。

```

1. foo.bar.com --|                               |-> foo.bar.com s1:80
2.               | 178.91.123.132 |
3. bar.foo.com  --|                               |-> bar.foo.com s2:80

```

以下Ingress告诉后端负载均衡器根据 `Host header` 进行路由请求。

```

1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: test
5. spec:
6.   rules:
7.     - host: foo.bar.com
8.     http:
9.       paths:
10.        - backend:
11.          serviceName: s1
12.          servicePort: 80
13.     - host: bar.foo.com
14.     http:
15.       paths:

```



```

16.         - backend:
17.             serviceName: s2
18.             servicePort: 80

```

**默认后端**：一个没有rule的Ingress，如上一节所示，所有流量都会发送到一个默认的backend。您可以使用相同的技术，通过指定一组rule和默认backend，来告诉负载均衡器找到您网站的404页面。如果您的Ingress中的所有Host都无法与请求头中的Host匹配，或者，没有path与请求的URL匹配，则流量将路由到您的默认backend。

## TLS

您可以通过指定包含TLS私钥和证书的 `secret` 来加密Ingress。目前，Ingress仅支持单个TLS端口443，并假定 TLS termination (TLS终止)。如果Ingress中的TLS配置部分指定了不同的host，那么它们将根据通过SNI TLS扩展指定的主机名复用同一端口（如果你提供的Ingress Controller支持SNI）。TLS secret必须包含名为 `tls.crt` 和 `tls.key` 的密钥，其中包含用于TLS的证书和私钥，例如：

```

1. apiVersion: v1
2. data:
3.   tls.crt: base64 encoded cert
4.   tls.key: base64 encoded key
5. kind: Secret
6. metadata:
7.   name: testsecret
8.   namespace: default
9. type: Opaque

```

在Ingress中引用这个secret会告诉Ingress Controller，使用TLS加密从客户端到负载均衡器器的channel：

```

1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: no-rules-map
5. spec:
6.   tls:
7.     - secretName: testsecret
8.   backend:
9.     serviceName: s1
10.    servicePort: 80

```

请注意，不同Ingress Controller支持的TLS功能存在差距。请参阅有关 [nginx](#)、[GCE](#) 或任何其他平台特定的Ingress Controller的文档，以了解TLS在您的环境中的工作原理。

## Loadbalancing

Ingress Controller启动时，会带上适用于所有Ingress的负载均衡策略设置，例如负载均衡算法、后端权重方案等。更高级的负载均衡概念（例如：持续会话、动态权重）尚未通过Ingress公开。您仍然可以通过 `service loadbalancer` 获得这些功能。随着时间的推移，我们计划将跨平台适用的负载均衡模式提取到Ingress资源中。

另外，尽管健康检查不直接通过Ingress暴露，但是在Kubernetes中存在parallel（并行）的概念，例如

[readiness probes](#)，可让您实现相同的最终结果。请查看特定Controller的文档，以了解他们如何处理健康检查（[nginx](#)、[GCE](#)）。

## Updating an Ingress

如果您想将新的Host添加到现有Ingress中，可通过编辑资源进行更新：

```
1. $ kubectl get ing
2. NAME      RULE      BACKEND  ADDRESS
3. test      -         178.91.123.132
4.          foo.bar.com
5.          /foo      s1:80
6. $ kubectl edit ing test
```

将会弹出一个编辑器，让你修改现有yaml，修改它，添加新的Host：

```
1. spec:
2.   rules:
3.   - host: foo.bar.com
4.     http:
5.       paths:
6.       - backend:
7.           serviceName: s1
8.           servicePort: 80
9.         path: /foo
10.  - host: bar.baz.com
11.    http:
12.      paths:
13.      - backend:
14.          serviceName: s2
15.          servicePort: 80
16.        path: /foo
17.  ..
```

保存yaml，就会更新API Server中的资源，这将会告诉Ingress Controller重新配置负载均衡器。

```
1. $ kubectl get ing
2. NAME      RULE      BACKEND  ADDRESS
3. test      -         178.91.123.132
4.          foo.bar.com
5.          /foo      s1:80
6.          bar.baz.com
7.          /foo      s2:80
```

在一个被修改过的Ingress yaml文件上，调用 `kubectl replace -f` 也可实现相同的效果。

## Failing across availability zones (跨可用区的故障)

不同cloud provider之间，跨故障域流量传播技术有所不同。有关详细信息，请查看Ingress Controller相关的文档。 有关在federated cluster中部署Ingress的详细信息，请参阅[federation doc](#)。

## Future Work

---

- 各种模式的HTTPS/TLS支持（例如：SNI、re-encryption）
- 通过声明请求IP或主机名
- 结合L4和L7 Ingress
- 更多Ingress Controllers

请追踪 [L7 and Ingress proposal](#)（L7和Ingress提案），了解有关资源演进的更多细节，以及 [Ingress repository](#)，了解有关各种Ingress Controller演进的更多细节。

## Alternatives（备选方案）

---

有多种方式暴露Service，而无需直接涉及Ingress资源：

- 使用 [Service.Type=LoadBalancer](#)
- 使用 [Service.Type=NodePort](#)
- 使用 [Port Proxy](#)
- 部署 [Service loadbalancer](#)。这允许您在多个Service之间共享一个IP，并通过Service Annotation实现更高级的负载均衡。

## 原文

---

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

# Horizontal Pod Autoscaling (HPA)

---

本文描述了Kubernetes中Horizontal Pod Autoscaling的当前状态。

## What is Horizontal Pod Autoscaling? (什么是HPA)

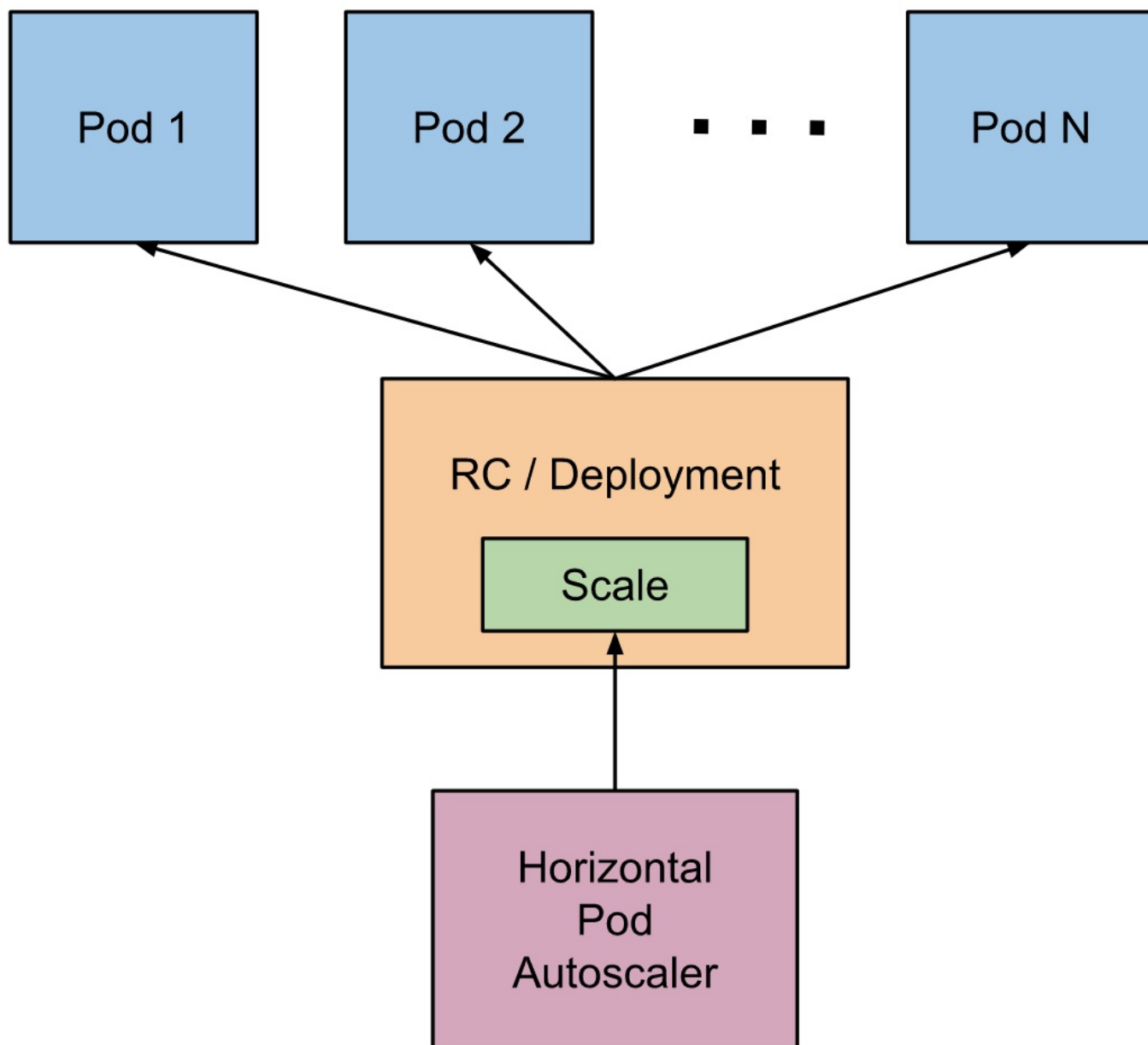
---

使用HPA，Kubernetes会根据观察到的CPU利用率（或根据在其他应用提供的度量标准，Beta状态）自动缩放Replication Controller、Deployment或ReplicaSet中的Pod个数。请注意，HPA不适用于无法缩放的对象，例如DaemonSet。

Horizontal Pod Autoscaler实现为Kubernetes API资源和Controller。资源决定了Controller的行为。Controller定期调整Replication Controller或Deployment中的副本数，从而将观察到的平均CPU利用率与用户指定的目标相匹配。

## How does the Horizontal Pod Autoscaler work? (Horizontal Pod Autoscaler如何工作)

---



Horizontal Pod Autoscaler被实现为一个控制回路，由Controller Manager的 `--horizontal-pod-autoscaler-sync-period` 标志（默认值为30秒）控制循环周期。

在每个周期内，Controller Manager根据每个HorizontalPodAutoscaler定义中指定的指标来查询资源利用。Controller Manager从资源指标API（针对per-pod的资源指标）或自定义指标API（所有其他指标）中获取指标。

- 对于per-pod的资源指标（如CPU），Controller从每个pod的资源指标API中获取指标，这些Pod是由HorizontalPodAutoscaler定位的。然后，如果设置了目标利用率值，则Controller计算利用率的值，跟每个pod中的容器上的资源请求的百分比相同。如果设置了目标原始值，则直接使用原始指标值。然后，Controller在所有目标pod中，获取利用率或原始值（取决于指定的目标类型）的平均值，并产生用于缩放期望的副本数量的比率。

请注意，如果某些pod的容器未设置相应的资源请求，则不会定义pod的CPU利用率，并且autoscaler不会对该指标采取任何操作。有关自动伸缩算法如何工作的更多信息，阅读 [autoscaling algorithm design document](#)。

- 对于per-pod的自定义指标，Controller的功能类似于per-pod的资源指标，除了它适用于原始值而非利用率值。
- 对于对象的指标，获取单个指标（描述所讨论的对象），并与目标值进行比较，从而产生如上所述的比率。

HorizontalPodAutoscaler Controller可以两种不同的方式获取指标：直接使用Heapster访问与使用REST客户端访问。

当直接使用Heapster访问时，HorizontalPodAutoscaler直接通过API Server的service proxy subresource查询Heapster。需在集群上部署Heapster并在kube-system这个namespace中运行。

有关使用REST客户端访问的详细信息，请参阅 [Support for custom metrics](#) 。

autoscaler通过 scale sub-resource访问相应的Replication Controller、Deployment或ReplicaSet。Scale是一个允许您动态设置副本数，并检查其当前状态的接口。有关scale sub-resource的更多细节可在 [here](#) 找到。

译者按：Heapster是一款容器集群监控及性能分析工具，拓展阅读：<https://segmentfault.com/a/1190000007708162>

## API Object

Horizontal Pod Autoscaler是Kubernetes `autoscaling` API组中的API资源。当前稳定版本中，只包括对CPU自动缩放的支持，可在 `autoscaling/v1` API版本中找到。

Beta版本包括对内存以及自定义指标的支持，可以在 `autoscaling/v2beta1` 找到。在 `autoscaling/v2beta1` 中引入的新字段，在使用 `autoscaling/v1` 时被保留为注释。

有关API对象的更多详细信息，请参见 [HorizontalPodAutoscaler Object](#) 。

## Support for Horizontal Pod Autoscaler in kubectl (kubectl对Horizontal Pod Autoscaler的支持)

和所有API资源一样，Horizontal Pod Autoscaler以 `kubectl` 的标准方式支持。可使用 `kubectl create` 命令创建一个新的autoscaler；通过 `kubectl get hpa` 列出autoscaler；通过 `kubectl describe hpa` 获得详细的描述；通过 `kubectl delete hpa` 删除autoscaler。

此外，还有一个特殊的 `kubectl autoscale` 命令，使用它可轻松创建一个Horizontal Pod Autoscaler。例如，执行 `kubectl autoscale rc foo --min=2 --max=5 --cpu-percent=80` 将为foo这个Replication Controller创建一个autoscaler，目标CPU利用率设置为 `80%`，副本数量介于2和5之间。可在 [here](#) 找到 `kubectl autoscale` 的详细文档。

## Autoscaling during rolling update (滚动更新期间的自动缩放)

目前在Kubernetes中，可以通过直接管理Replication Controller或使用Deployment对象来执行 [rolling](#)

`update`，当使用Deployment时，Deployment对象为您管理底层Replication Controller。Horizontal Pod Autoscaler仅支持后一种方法：Horizontal Pod Autoscaler被绑定到Deployment对象，它设置Deployment对象的大小，Deployment负责设置底层Replication Controller的大小。

当直接使用Replication Controller时，Horizontal Pod Autoscaler不能与滚动更新一起工作，即不能将Horizontal Pod Autoscaler绑定到Replication Controller并执行滚动更新（例如使用 `kubectl rolling-update`）。不行的原因是：当滚动更新创建一个新的Replication Controller时，Horizontal Pod Autoscaler将不会绑定到新的Replication Controller。

## Support for multiple metrics（对多个指标的支持）

Kubernetes 1.6添加了对“基于多个指标缩放”的支持。可使用 `autoscaling/v2beta1` API版本，指定Horizontal Pod Autoscaler扩展的多个指标。然后，Horizontal Pod Autoscaler Controller将对每个指标进行评估，并根据该指标提出新的规模。最大建议的规模将被用作新的规模。

## 对自定义指标的支持

注意：Kubernetes 1.2根据使用特殊Annotation的application-specific metrics（特定于应用程序的指标），增加了对缩放的Alpha支持。在Kubernetes 1.6中删除了对这些Annotation的支持，而是使用新的自动缩放API。虽然，用于收集自定义指标的旧方法仍然可用，但是这些指标将不可被Horizontal Pod Autoscaler使用，并且，用于指定要缩放的自定义指标的Annotation，不再由Horizontal Pod Autoscaler Controller执行。

Kubernetes 1.6增加了在Horizontal Pod Autoscaler中使用自定义指标的支持。您可在 `autoscaling/v2beta1` API中，添加Horizontal Pod Autoscaler要使用的自定义指标。这样，Kubernetes就会查询新的自定义指标API来获取适当的自定义指标的值。

## Requirements（需要的条件）

要使用Horizontal Pod Autoscaler的自定义指标，您必须在部署集群时设置必要的配置：

- [Enable the API aggregation layer](#)（启用API聚合层）
- 使用API aggregation layer注册资源指标API和自定义指标API。这两个API server必须在您的集群上运行。
  - 资源指标API：可使用Heapster的资源指标API实现，方法是运行Heapster时将其 `--api-server` 标志设为true。
  - 自定义指标API：这必须由单独的组件提供。样板代码，请参阅 [kubernetes-incubator/custom-metrics-apiserver](#) 以及 [k8s.io/metrics](#)。
- 为kube-controller-manager设置相应的标志：
  - `--horizontal-pod-autoscaler-use-rest-clients` 应设为true。
  - `--kubeconfig <path-to-kubeconfig>` 或 `--master <ip-address-of-apiserver>`

请注意，可使用 `--master` 或 `--kubeconfig` 标志；`--master` 将会覆盖 `--kubeconfig`，如果

两者都被指定。这些标志指定API aggregation layer的位置，允许Controller Manager与 API server通信。

在Kubernetes 1.7中，Kubernetes提供的标准聚合层与kube-apiserver一起运行，因此可使用

```
kubectl get pods --selector k8s-app=kube-apiserver --namespace kube-system -o  
jsonpath='{.items[0].status.podIP}'
```

 找到目标IP。

## Further reading ( 进一步阅读 )

---

- 设计文档： [Horizontal Pod Autoscaling](#).
- kubectl autoscale 命令： [kubectl autoscale](#).
- 使用示例： [Horizontal Pod Autoscaler](#).

## 原文

---

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>



# 标题

## MySQL

### 创建deployment

```
1. apiVersion: extensions/v1beta1
2. kind: Deployment                                # 定义一个deployment
3. metadata:
4.   name: mysql                                  # deployment名称, 全局唯一
5.   labels:
6.     app: mysql123
7.     release: stable
8. spec:
9.   replicas: 1
10.  selector:
11.    matchLabels:
12.      app: mysql123                            # deployment的POD标签选择器, 即: 监控和管理拥有这些标签的POD实例, 确保当前集群中有
且只有replicas个POD实例在运行
13.  template:
14.    metadata:
15.      labels:                                    # 指定该POD的标签
16.        app: mysql123                            # POD副本拥有的标签, 需要与deployment的selector一致
17.    spec:
18.      containers:
19.        - name: mysql
20.          image: mysql
21.          ports:
22.            - containerPort: 3306
23.          env:
24.            - name: MYSQL_ROOT_PASSWORD
25.              value: "123456"
```

### 创建SVC

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: mysql                                  # Service名称, 全局唯一
5.   labels:
6.     app: mysql123
7. spec:
8.   ports:
9.     - port: 3306                                # Service提供服务的端口号
10.  selector:
11.    app: mysql123                                # 选择器
```

# Wordpress

## 创建deployment

```

1. apiVersion: extensions/v1beta1
2. kind: Deployment
3. metadata:
4.   name: wordpress
5.   labels:
6.     app: wordpress
7. spec:
8.   replicas: 1
9.   selector:
10.    matchLabels:
11.      app: wordpress
12.   template:
13.     metadata:
14.       labels:
15.         app: wordpress
16.     spec:
17.       containers:
18.         - name: wordpress
19.           image: wordpress:4.8-apache
20.           ports:
21.             - containerPort: 80
22.           env:
23.             - name: WORDPRESS_DB_HOST
24.               value: "mysql"
25.             - name: WORDPRESS_DB_PASSWORD
26.               value: "123456"

```

## 创建SVC

```

1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: wordpress
5.   labels:
6.     app: wordpress
7. spec:
8.   type: NodePort    # 为该Service开启NodePort方式的外网访问模式
9.   ports:
10.    - port: 80        # Service提供服务的端口号
11.      targetPort: 80  # 将Service的80端口转发到Pod中容器的80端口上
12.      nodePort: 32001 # 在k8s集群外访问的端口，如果设置了NodePort类型，但没设置nodePort，将会随机映射一个端口，可使用kubectl
get svc wordpress看到
13.   selector:
14.     app: wordpress

```

# 删除

---

1. `kubectl delete deployment,svc -l app=mysql123`
2. `kubectl delete deployment,svc -l app=wordpress`