基础入门

Introduction

Kubernetes 是谷歌开源的容器集群管理系统，是 Google 多年大规模容器管理技术 Borg 的开源版本，也是 CNCF 最重要的项目之一，主要功能包括：

- 基于容器的应用部署、维护和滚动升级
- 负载均衡和服务发现
- 跨机器和跨地区的集群调度
- 自动伸缩
- 无状态服务和有状态服务
- 广泛的 Volume 支持
- 插件机制保证扩展性

Kubernetes 发展非常迅速，已经成为容器编排领域的领导者。Kubernetes 的中文资料也非常丰富，但系统化和紧跟社区更新的则就比较少见了。
《Kubernetes 指南》开源电子书旨在整理平时在开发和使用 Kubernetes 时的参考指南和实践总结，形成一个系统化的参考指南以方便查阅。欢迎大家关注和添加完善内容。

在线阅读

- 中文：
  - Gitbook: https://kubernetes.feisky.xyz/（或者 这里）
  - Github
  - InfoQ
- English
- PDF 电子书：点击 这里 下载

项目源码

项目源码存放于 Github 上，https://github.com/feiskyer/kubernetes-handbook。

本书版本更新记录

如无特殊说明，本指南所有文档仅适用于 Kubernetes v1.6 及以上版本。详细更新记录见 CHANGELOG。

微信公众号

扫码关注微信公众号，回复关键字即可在微信中查看相关章节。

贡献者

欢迎参与贡献和完善内容，贡献方法参考 CONTRIBUTING。感谢所有的贡献者，贡献者列表见 contributors。

LICENSE

署名-非商业性使用-相同方式共享 4.0 (CC BY-NC-SA 4.0)。

Kubernetes简介

Kubernetes 是谷歌开源的容器集群管理系统，是 Google 多年大规模容器管理技术 Borg 的开源版本，主要功能包括：

- 基于容器的应用部署、维护和滚动升级
- 负载均衡和服务发现
- 跨机器和跨地区的集群调度
- 自动伸缩
- 无状态服务和有状态服务
- 广泛的 Volume 支持
- 插件机制保证扩展性

Kubernetes 发展非常迅速，已经成为容器编排领域的领导者。

Kubernetes 是一个平台

Kubernetes 提供了很多的功能，它可以简化应用程序的工作流，加快开发速度。通常，一个成功的应用编排系统需要有较强的自动化能力，这也是为什么 Kubernetes 被设计作为构建组件和工具的生态系统平台，以便更轻松地部署、扩展和管理应用程序。

用户可以使用 Label 以自己的方式组织管理资源，还可以使用 Annotation 来自定义资源的描述信息，比如为管理工具提供状态检查等。

此外，Kubernetes 控制器也是构建在跟开发人员和用户使用的相同的 API 之上。用户还可以编写自己的控制器和调度器，也可以通过各种插件机制扩展系统的功能。

这种设计使得可以方便地在 Kubernetes 之上构建各种应用系统。

Kubernetes 不是什么

Kubernetes 不是一个传统意义上，包罗万象的 PaaS (平台即服务) 系统。它给用户预留了选择的自由。

- 不限制支持的应用程序类型，它不插手应用程序框架, 也不限制支持的语言 (如 Java, Python, Ruby 等)，只要应用符合 12 因素 即可。Kubernetes 旨在支持极其多样化的工作负载，包括无状态、有状态和数据处理工作负

载。只要应用可以在容器中运行，那么它就可以很好的在 Kubernetes 上运行。
- 不提供内置的中间件 (如消息中间件)、数据处理框架 (如 Spark)、数据库 (如 mysql) 或集群存储系统 (如 Ceph) 等。这些应用直接运行在 Kubernetes 之上。
- 不提供点击即部署的服务市场。
- 不直接部署代码，也不会构建您的应用程序，但您可以在 Kubernetes 之上构建需要的持续集成 (CI) 工作流。
- 允许用户选择自己的日志、监控和告警系统。
- 不提供应用程序配置语言或系统 (如 jsonnet)。
- 不提供机器配置、维护、管理或自愈系统。

另外，已经有很多 PaaS 系统运行在 Kubernetes 之上，如 Openshift, Deis 和 Eldarion 等。 您也可以构建自己的 PaaS 系统，或者只使用 Kubernetes 管理您的容器应用。

当然了，Kubernetes 不仅仅是一个“编排系统”，它消除了编排的需要。Kubernetes 通过声明式的 API 和一系列独立、可组合的控制器保证了应用总是在期望的状态，而用户并不需要关心中间状态是如何转换的。这使得整个系统更容易使用，而且更强大、更可靠、更具弹性和可扩展性。

核心组件

Kubernetes 主要由以下几个核心组件组成：

- etcd 保存了整个集群的状态；
- apiserver 提供了资源操作的唯一入口，并提供认证、授权、访问控制、API 注册和发现等机制；
- controller manager 负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上；
- kubelet 负责维护容器的生命周期，同时也负责 Volume（CVI）和网络（CNI）的管理；
- Container runtime 负责镜像管理以及 Pod 和容器的真正运行（CRI）；
- kube-proxy 负责为 Service 提供 cluster 内部的服务发现和负载均衡

Kubernetes 版本

Kubernetes 的稳定版本在发布后会继续支持 9 个月。每个版本的支持周期为：

Kubernetes version	Release month	End-of-life-month
v1.6.x	March 2017	December 2017
v1.7.x	June 2017	March 2018
v1.8.x	September 2017	June 2018
v1.9.x	December 2017	September 2018
v1.10.x	March 2018	December 2018
v1.11.x	June 2018	March 2019

参考文档

- What is Kubernetes?
- HOW CUSTOMERS ARE REALLY USING KUBERNETES

Kubernetes基本概念

Container

Container（容器）是一种便携式、轻量级的操作系统级虚拟化技术。它使用namespace 隔离不同的软件运行环境，并通过镜像自包含软件的运行环境，从而使得容器可以很方便的在任何地方运行。

由于容器体积小且启动快，因此可以在每个容器镜像中打包一个应用程序。这种一对一的应用镜像关系拥有很多好处。使用容器，不需要与外部的基础架构环境绑定，因为每一个应用程序都不需要外部依赖，更不需要与外部的基础架构环境依赖。完美解决了从开发到生产环境的一致性问题。

容器同样比虚拟机更加透明，这有助于监测和管理。尤其是容器进程的生命周期由基础设施管理，而不是被进程管理器隐藏在容器内部。最后，每个应用程序用容器封装，管理容器部署就等同于管理应用程序部署。

其他容器的优点还包括

- 敏捷的应用程序创建和部署: 与虚拟机镜像相比，容器镜像更易用、更高效。
- 持续开发、集成和部署: 提供可靠与频繁的容器镜像构建、部署和快速简便的回滚（镜像是不可变的）。
- 开发与运维的关注分离: 在构建/发布时即创建容器镜像，从而将应用与基础架构分离。
- 开发、测试与生产环境的一致性: 在笔记本电脑上运行和云中一样。
- 可观测：不仅显示操作系统的信息和度量，还显示应用自身的信息和度量。
- 云和操作系统的分发可移植性: 可运行在 Ubuntu, RHEL, CoreOS, 物理机, GKE 以及其他任何地方。
- 以应用为中心的管理: 从传统的硬件上部署操作系统提升到操作系统中部署应用程序。
- 松耦合、分布式、弹性伸缩、微服务: 应用程序被分成更小，更独立的模块，并可以动态管理和部署 - 而不是运行在专用设备上的大型单体程序。
- 资源隔离：可预测的应用程序性能。
- 资源利用：高效率和高密度。

Pod

Kubernetes 使用 Pod 来管理容器，每个 Pod 可以包含一个或多个紧密关联的容器。

Pod 是一组紧密关联的容器集合，它们共享 PID、IPC、Network 和 UTS namespace，是 Kubernetes 调度的基本单位。Pod 内的多个容器共享网络和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。

pod

在 Kubernetes 中，所有对象都使用 manifest（yaml 或 json）来定义，比如一个简单的 nginx 服务可以定义为 nginx.yaml，它包含一个镜像为 nginx 的容器：

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

Node

Node 是 Pod 真正运行的主机，可以是物理机，也可以是虚拟机。为了管理 Pod，每个 Node 节点上至少要运行 container runtime（比如 docker 或者 rkt）、`kubelet` 和 `kube-proxy` 服务。

node

Namespace

Namespace 是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。常见的 pods, services, replication controllers 和 deployments 等都是属于某一个 namespace 的（默认是 default），而 node, persistentVolumes 等则不属于任何 namespace。

Service

Service 是应用服务的抽象，通过 labels 为应用提供负载均衡和服务发现。匹配 labels 的 Pod IP 和端口列表组成 endpoints，由 kube-proxy 负责将服务 IP 负载均衡到这些 endpoints 上。

每个 Service 都会自动分配一个 cluster IP（仅在集群内部可访问的虚拟地址）和 DNS 名，其他容器可以通过该地址或 DNS 来访问服务，而不需要了解后端容器的运行。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
  - port: 8078 # the port that this service should serve on
    name: http
    # the container on each pod to connect to, can be a name
    # (e.g. 'www') or a number (e.g. 80)
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```

Label

Label 是识别 Kubernetes 对象的标签，以 key/value 的方式附加到对象上（key 最长不能超过 63 字节，value 可以为空，也可以是不超过 253 字节的字符串）。

Label 不提供唯一性，并且实际上经常是很多对象（如 Pods）都使用相同的 label 来标志具体的应用。

Label 定义好后其他对象可以使用 Label Selector 来选择一组相同 label 的对象（比如 ReplicaSet 和 Service 用 label 来选择一组 Pod）。Label Selector 支持以下几种方式：

- 等式，如 `app=nginx` 和 `env!=production`
- 集合，如 `env in (production, qa)`
- 多个 label（它们之间是 AND 关系），如 `app=nginx,env=test`

Annotations

Annotations 是 key/value 形式附加于对象的注解。不同于 Labels 用于标志和选择对象，Annotations 则是用来记录一些附加信息，用来辅助应用部署、安全策略以及调度策略等。比如 deployment 使用 annotations 来记录 rolling update 的状态。

Kubernetes101

体验 Kubernetes 最简单的方法是跑一个 nginx 容器，然后使用 kubectl 操作该容器。Kubernetes 提供了一个类似于 `docker run` 的命令 `kubectl run`，可以方便的创建一个容器（实际上创建的是一个由 deployment 来管理的 Pod）：

```
$ kubectl run --image=nginx:alpine nginx-app --port=80
deployment "nginx-app" created
$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
nginx-app-4028413181-cnt1i   1/1       Running   0          52s
```

等到容器变成 Running 后，就可以用 `kubectl` 命令来操作它了，比如

- `kubectl get` - 类似于 `docker ps`，查询资源列表
- `kubectl describe` - 类似于 `docker inspect`，获取资源的详细信息
- `kubectl logs` - 类似于 `docker logs`，获取容器的日志
- `kubectl exec` - 类似于 `docker exec`，在容器内执行一个命令

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
nginx-app-4028413181-cnt1i    1/1       Running   0          6m

$ kubectl exec nginx-app-4028413181-cnt1i ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
root         1  0.0  0.5  31736  5108 ?        Ss   00:19   0:00
nginx        5  0.0  0.2  32124  2844 ?        S    00:19   0:00
root        18  0.0  0.2  17500  2112 ?        Rs   00:25   0:00

$ kubectl describe pod nginx-app-4028413181-cnt1i
Name:          nginx-app-4028413181-cnt1i
Namespace:         default
Node:          boot2docker/192.168.64.12
Start Time:        Tue, 06 Sep 2016 08:18:41 +0800
Labels:        pod-template-hash=4028413181
               run=nginx-app
Status:        Running
IP:            172.17.0.3
Controllers:       ReplicaSet/nginx-app-4028413181
Containers:
  nginx-app:
    Container ID:              docker://4ef989b57d0a7638ad9c5bbc2
    Image:                 nginx
    Image ID:              docker://sha256:4efb2fcdb1ab05fb03c943
    Port:                  80/TCP
    State:                 Running
      Started:             Tue, 06 Sep 2016 08:19:30 +0800
    Ready:                 True
    Restart Count:             0
    Environment Variables:
Conditions:
  Type         Status
  Initialized      True
  Ready            True
  PodScheduled     True
Volumes:
  default-token-9o8ks:
    Type:          Secret (a volume populated by a Secret)
    SecretName:    default-token-9o8ks
```

```
QoS Tier:          BestEffort
Events:
  FirstSeen        LastSeen           Count      From
  ---------        --------           -----      ----
  8m           8m             1          {default-scheduler}
  8m           8m             1          {kubelet boot2docker}
  7m           7m             1          {kubelet boot2docker}
  7m           7m             1          {kubelet boot2docker}
  7m           7m             1          {kubelet boot2docker}

$ curl http://172.17.0.3
```

Welcome to nginx!

If you see this page, the nginx web server is successfully instal
working. Further configuration is required.

For online documentation and support please refer to
"http://nginx.org/">nginx.org.

Commercial support is available at
"http://nginx.com/">nginx.com.

Thank you for using nginx.

```
$ kubectl logs nginx-app-4028413181-cnt1i
127.0.0.1 - - [06/Sep/2016:00:27:13 +0000] "GET / HTTP/1.0" 200 6
```

使用 yaml 定义 Pod

上面是通过 `kubectl run` 来启动了第一个 Pod，但是 `kubectl run` 并不支持所有的功能。在 Kubernetes 中，更经常使用 yaml 文件来定义资源，并通过 `kubectl create -f file.yaml` 来创建资源。比如，一个简单的 nginx Pod 可以定义为：

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

前面提到，`kubectl run` 并不是直接创建一个 Pod，而是先创建一个 Deployment 资源（replicas=1），再由与 Deployment 关联的 ReplicaSet 来自动创建 Pod，这等价于这样一个配置：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: nginx-app
  name: nginx-app
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: nginx-app
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: nginx-app
    spec:
      containers:
      - image: nginx
        name: nginx-app
        ports:
        - containerPort: 80
          protocol: TCP
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

使用 Volume

Pod 的生命周期通常比较短，只要出现了异常，就会创建一个新的 Pod 来代替它。那容器产生的数据呢？容器内的数据会随着 Pod 消亡而自动消失。
Volume 就是为了持久化容器数据而生，比如可以为 redis 容器指定一个 hostPath 来存储 redis 数据：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-persistent-storage
      mountPath: /data/redis
  volumes:
  - name: redis-persistent-storage
    hostPath:
      path: /data/
```

Kubernetes volume 支持非常多的插件，可以根据实际需要来选择：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo
- secret
- persistentVolumeClaim
- downwardAPI
- azureFileVolume
- vsphereVolume

使用 Service

前面虽然创建了 Pod，但是在 kubernetes 中，Pod 的 IP 地址会随着 Pod 的重启而变化，并不建议直接拿 Pod 的 IP 来交互。那如何来访问这些 Pod 提供的服务呢？使用 Service。Service 为一组 Pod（通过 labels 来选择）提供一个统一的入口，并为它们提供负载均衡和自动服务发现。比如，可以为前面的 `nginx-app` 创建一个 service：

```
$ kubectl expose deployment nginx-app --port=80 --target-port=80
service "nginx-app" exposed
$ kubectl describe service nginx-app
Name:              nginx-app
Namespace:             default
Labels:            run=nginx-app
Selector:              run=nginx-app
Type:              ClusterIP
IP:                10.0.0.66
Port:                  80/TCP
NodePort:                  30772/TCP
Endpoints:             172.17.0.3:80
Session Affinity:          None
No events.
```

这样，在 cluster 内部就可以通过 `http://10.0.0.66` 和 `http://node-ip:30772` 来访问 nginx-app。而在 cluster 外面，则只能通过 `http://node-ip:30772` 来访问。

Kubernetes201

扩展应用

通过修改 Deployment 中副本的数量（replicas），可以动态扩展或收缩应用：

scale

这些自动扩展的容器会自动加入到 service 中，而收缩回收的容器也会自动从 service 中删除。

```
$ kubectl scale --replicas=3 deployment/nginx-app
$ kubectl get deploy
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-app   3         3         3            3           10m
```

滚动升级

滚动升级（Rolling Update）通过逐个容器替代升级的方式来实现无中断的服务升级：

```
kubectl rolling-update frontend-v1 frontend-v2 --image=image:v2
```

update1

update2

update3

update4

在滚动升级的过程中，如果发现了失败或者配置错误，还可以随时回滚：

```
kubectl rolling-update frontend-v1 frontend-v2 --rollback
```

需要注意的是，`kubectl rolling-update` 只针对 ReplicationController。对于更新策略是 RollingUpdate 的 Deployment（Deployment 可以在 spec 中设置更新策略为 RollingUpdate，默认就是 RollingUpdate），更新应用后会自动滚动升级：

```
  spec:
    replicas: 3
    selector:
      matchLabels:
        run: nginx-app
    strategy:
      rollingUpdate:
        maxSurge: 1
        maxUnavailable: 1
      type: RollingUpdate
```

而更新应用的话，就可以直接用 `kubectl set` 命令：

```
kubectl set image deployment/nginx-app nginx-app=nginx:1.9.1
```

滚动升级的过程可以用 `rollout` 命令查看:

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

Deployment 也支持回滚：

```
$ kubectl rollout history deployment/nginx-app
deployments "nginx-app"
REVISION    CHANGE-CAUSE
1
2

$ kubectl rollout undo deployment/nginx-app
deployment "nginx-app" rolled back
```

资源限制

Kubernetes 通过 cgroups 提供容器资源管理的功能，可以限制每个容器的 CPU 和内存使用，比如对于刚才创建的 deployment，可以通过下面的命令限制 nginx 容器最多只用 50% 的 CPU 和 128MB 的内存：

```
$ kubectl set resources deployment nginx-app -c=nginx --limits=cp
deployment "nginx" resource requirements updated
```

这等同于在每个 Pod 中设置 resources limits：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
```

健康检查

Kubernetes 作为一个面向应用的集群管理工具，需要确保容器在部署后确实处在正常的运行状态。Kubernetes 提供了两种探针（Probe，支持 exec、tcpSocket 和 http 方式）来探测容器的状态：

- LivenessProbe：探测应用是否处于健康状态，如果不健康则删除并重新创建容器
- ReadinessProbe：探测应用是否启动完成并且处于正常服务状态，如果不正常则不会接收来自 Kubernetes Service 的流量

对于已经部署的 deployment，可以通过 `kubectl edit deployment/nginx-app` 来更新 manifest，增加健康检查部分：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx-default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: http
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          limits:
            cpu: "500m"
            memory: "128Mi"
        livenessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 15
          timeoutSeconds: 1
        readinessProbe:
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          timeoutSeconds: 1
```

Kubernetes集群

一个 Kubernetes 集群由分布式存储 etcd、控制节点 controller 以及服务节点 Node 组成。

- 控制节点主要负责整个集群的管理，比如容器的调度、维护资源的状态、自动扩展以及滚动更新等
- 服务节点是真正运行容器的主机，负责管理镜像和容器以及 cluster 内的服务发现和负载均衡
- etcd 集群保存了整个集群的状态

详细的介绍请参考 Kubernetes 架构。

集群联邦

集群联邦（Federation）用于跨可用区的 Kubernetes 集群，需要配合云服务商（如 GCE、AWS）一起实现。

详细的介绍请参考 Federation。

创建 Kubernetes 集群

可以参考 Kubernetes 部署指南 来部署一套 Kubernetes 集群。而对于初学者或者简单验证测试的用户，则可以使用以下几种更简单的方法。

minikube

创建 Kubernetes cluster（单机版）最简单的方法是 minikube:

```
$ minikube start
Starting local Kubernetes cluster...
Kubectl is now configured to use the cluster.
$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.12:8443
kubernetes-dashboard is running at https://192.168.64.12:8443/api

To further debug and diagnose cluster problems, use 'kubectl clus
```

play-with-k8s

Play with Kubernetes 提供了一个免费的 Kubernetes 体验环境，直接访问 < http://play-with-k8s.com > 就可以使用 kubeadm 来创建 Kubernetes 集群。注意，每次创建的集群最长可以使用 4 小时。

Play with Kubernetes 有个非常方便的功能：自动在页面上显示所有 NodePort 类型服务的端口，点击该端口即可访问对应的服务。

详细使用方法可以参考 Play-With-Kubernetes。

Katacoda playground

Katacoda playground也提供了一个免费的 2 节点 Kubernetes 体验环境，网络基于 WeaveNet，并且会自动部署整个集群。但要注意，刚打开 Katacoda playground 页面时集群有可能还没初始化完成，可以在 master 节点上运行 `launch.sh` 等待集群初始化完成。

部署并访问 kubernetes dashboard 的方法：

```
# 在 master node 上面运行
kubectl create -f https://raw.githubusercontent.com/kubernetes/da
kubectl proxy --address='0.0.0.0' --port=8080 --accept-hosts='^*$
```

然后点击 Terminal Host 1 右边的➕，从弹出的菜单里选择 View HTTP port 8080 on Host 1，即可打开 Kubernetes 的 API 页面。在该网址后面增加 `/ui` 即可访问 dashboard。

核心原理

核心原理

介绍 Kubernetes 架构以及核心组件，包括

- 架构原理
- 设计理念
- 核心组件
  - etcd
  - kube-apiserver

- kube-scheduler
- kube-controller-manager
- kubelet
- kube-proxy
- kube-dns
- Federation
- kubeadm
- hyperkube
- kubectl

架构原理

Kubernetes 最初源于谷歌内部的 Borg，提供了面向应用的容器集群部署和管理系统。Kubernetes 的目标旨在消除编排物理 / 虚拟计算，网络和存储基础设施的负担，并使应用程序运营商和开发人员完全将重点放在以容器为中心的原语上进行自助运营。Kubernetes 也提供稳定、兼容的基础（平台），用于构建定制化的 workflows 和更高级的自动化任务。 Kubernetes 具备完善的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和在线扩容、可扩展的资源自动调度机制、多粒度的资源配额管理能力。 Kubernetes 还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

Borg 简介

Borg 是谷歌内部的大规模集群管理系统，负责对谷歌内部很多核心服务的调度和管理。Borg 的目的是让用户能够不必操心资源管理的问题，让他们专注于自己的核心业务，并且做到跨多个数据中心的资源利用率最大化。

Borg 主要由 BorgMaster、Borglet、borgcfg 和 Scheduler 组成，如下图所示

borg

- BorgMaster 是整个集群的大脑，负责维护整个集群的状态，并将数据持久化到 Paxos 存储中；
- Scheduer 负责任务的调度，根据应用的特点将其调度到具体的机器上去；
- Borglet 负责真正运行任务（在容器中）；
- borgcfg 是 Borg 的命令行工具，用于跟 Borg 系统交互，一般通过一个配置文件来提交任务。

Kubernetes 架构

Kubernetes 借鉴了 Borg 的设计理念，比如 Pod、Service、Labels 和单 Pod 单 IP 等。Kubernetes 的整体架构跟 Borg 非常像，如下图所示

architecture

Kubernetes 主要由以下几个核心组件组成：

- etcd 保存了整个集群的状态；
- kube-apiserver 提供了资源操作的唯一入口，并提供认证、授权、访问控制、API 注册和发现等机制；
- kube-controller-manager 负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- kube-scheduler 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上；
- kubelet 负责维持容器的生命周期，同时也负责 Volume（CVI）和网络（CNI）的管理；
- Container runtime 负责镜像管理以及 Pod 和容器的真正运行（CRI），默认的容器运行时为 Docker；
- kube-proxy 负责为 Service 提供 cluster 内部的服务发现和负载均衡；

除了核心组件，还有一些推荐的 Add-ons：

- kube-dns 负责为整个集群提供 DNS 服务
- Ingress Controller 为服务提供外网入口
- Heapster 提供资源监控
- Dashboard 提供 GUI
- Federation 提供跨可用区的集群
- Fluentd-elasticsearch 提供集群日志采集、存储与查询

分层架构

Kubernetes 设计理念和功能其实就是一个类似 Linux 的分层架构，如下图所示

- 核心层：Kubernetes 最核心的功能，对外提供 API 构建高层的应用，对内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS 解析等）

- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩展、动态 Provision 等）以及策略管理（RBAC、Quota、PSP、NetworkPolicy 等）
- 接口层：kubectl 命令行工具、客户端 SDK 以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
  - Kubernetes 外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS 应用、ChatOps 等
  - Kubernetes 内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

核心组件

核心 API

生态系统

关于分层架构，可以关注下 Kubernetes 社区正在推进的 Kubernetes architectual roadmap。

参考文档

- Kubernetes design and architecture
- http://queue.acm.org/detail.cfm?id=2898444
- http://static.googleusercontent.com/media/research.google.com/zh-CN//pubs/archive/43438.pdf
- http://thenewstack.io/kubernetes-an-overview
- Kubernetes Architecture SIG

设计理念

设计理念与分布式系统

分析和理解Kubernetes的设计理念可以使我们更深入地了解Kubernetes系统，更好地利用它管理分布式部署的云原生应用，另一方面也可以让我们借鉴其在分布式系统设计方面的经验。

API设计原则

对于云计算系统，系统API实际上处于系统设计的统领地位。Kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。理解掌握的API，就好比抓住了K8s系统的牛鼻子。Kubernetes系统API的设计有以下几条原则：

1. **所有API应该是声明式的。**声明式的操作，相对于命令式操作，对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更容易被用户使用，可以使系统向用户隐藏实现的细节，同时也保留了系统未来持续优化的可能性。此外，声明式的API还隐含了所有的API对象都是名词性质的，例如Service、Volume这些API都是名词，这些名词描述了用户所期望得到的一个目标对象。
2. **API对象是彼此互补而且可组合的。**这实际上鼓励API对象尽量实现面向对象设计时的要求，即"高内聚，松耦合"，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。
3. **高层API以操作意图为基础设计。**如何能够设计好API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对Kubernetes的高层API设计，一定是以K8s的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。
4. **低层API根据高层API的控制需要设计。**设计实现低层API的目的，是为了被高层API使用，考虑减少冗余、提高重用性的目的，低层API的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。
5. **尽量避免简单封装，不要有在外部API无法显式知道的内部隐藏的机制。**简单的封装，实际没有提供新的功能，反而增加了对所封装API的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如StatefulSet和ReplicaSet，本来就是两种Pod集合，那么Kubernetes就用不同API对象来定义它们，而不会说只用同一个ReplicaSet，内部通过特殊的算法再来区分这个ReplicaSet是有状态的还是无状态。
6. **API操作复杂度与对象数量成正比。**这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是API的操作复杂度不能超过O(N)，N是对象的数量，否则系统就不具备水平伸缩性了。
7. **API对象状态不能依赖于网络连接状态。**由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证API对象状态能应对网络的不稳定，API对象的状态就不能依赖于网络连接状态。
8. **尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。**

控制机制设计原则

- **控制逻辑应该只依赖于当前状态**。这是为了保证分布式系统的稳定可靠，对于经常出现局部错误的分布式系统，如果控制逻辑只依赖当前状态，那么就非常容易将一个暂时出现故障的系统恢复到正常状态，因为你只要将该系统重置到某个稳定状态，就可以自信的知道系统的所有控制逻辑会开始按照正常方式运行。
- **假设任何错误的可能，并做容错处理**。在一个分布式系统中出现局部和临时错误是大概率事件。错误可能来自于物理系统故障，外部系统故障也可能来自于系统自身的代码错误，依靠自己实现的代码不会出错来保证系统稳定其实也是难以实现的，因此要设计对任何可能错误的容错处理。
- **尽量避免复杂状态机，控制逻辑不要依赖无法监控的内部状态**。因为分布式系统各个子系统都是不能严格通过程序内部保持同步的，所以如果两个子系统的控制逻辑如果互相有影响，那么子系统就一定要能互相访问到影响控制逻辑的状态，否则，就等同于系统里存在不确定的控制逻辑。
- **假设任何操作都可能被任何操作对象拒绝，甚至被错误解析**。由于分布式系统的复杂性以及各子系统的相对独立性，不同子系统经常来自不同的开发团队，所以不能奢望任何操作被另一个子系统以正确的方式处理，要保证出现错误的时候，操作级别的错误不会影响到系统稳定性。
- **每个模块都可以在出错后自动恢复**。由于分布式系统中无法保证系统各个模块是始终连接的，因此每个模块要有自我修复的能力，保证不会因为连接不到其他模块而自我崩溃。
- **每个模块都可以在必要时优雅地降级服务**。所谓优雅地降级服务，是对系统鲁棒性的要求，即要求在设计实现模块时划分清楚基本功能和高级功能，保证基本功能不会依赖高级功能，这样同时就保证了不会因为高级功能出现故障而导致整个模块崩溃。根据这种理念实现的系统，也更容易快速地增加新的高级功能，因为不必担心引入高级功能影响原有的基本功能。

架构设计原则

- 只有apiserver可以直接访问etcd存储，其他服务必须通过Kubernetes API来访问集群状态
- 单节点故障不应该影响集群的状态
- 在没有新请求的情况下，所有组件应该在故障恢复后继续执行上次最后收到的请求（比如网络分区或服务重启等）
- 所有组件都应该在内存中保持所需要的状态，apiserver将状态写入etcd存储，而其他组件则通过apiserver更新并监听所有的变化
- 优先使用事件监听而不是轮询

引导（Bootstrapping）原则

- Self-hosting 是目标
- 减少依赖，特别是稳态运行的依赖
- 通过分层的原则管理依赖
- 循环依赖问题的原则
    - 同时还接受其他方式的数据输入（比如本地文件等），这样在其他服务不可用时还可以手动配置引导服务
    - 状态应该是可恢复或可重新发现的
    - 支持简单的启动临时实例来创建稳态运行所需要的状态；使用分布式锁或文件锁等来协调不同状态的切换（通常称为 `pivoting` 技术）
    - 自动重启异常退出的服务，比如副本或者进程管理器等

核心技术概念和API对象

API对象是K8s集群中的管理操作单元。K8s集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。

每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了用户期望K8s集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复本控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。

K8s中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是k8s重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Pod

K8s有很多技术概念，同时对应很多API对象，最重要的也是最基础的是微服务Pod。Pod是在K8s集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8s最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod是K8s集群中所有业务类型的基础，可以看作运行在K8s集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前K8s中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和StatefulSet，本文后面会一一介绍。

复制控制器（Replication Controller，RC）

RC是K8s集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是K8s较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的Web服务。

副本集（Replica Set，RS）

RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

部署(Deployment)

部署表示用户对K8s集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以K8s的发展方向，未来对所有长期伺服型的的业务的管理，都会通过Deployment来管理。

服务（Service）

RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在K8s集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在K8s集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是K8s集群内部的负载均衡器。它是一个分布式代理服务器，在K8s的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端使用反向代理作负载均衡，还要进一步解决反向代理的高可用问题。

任务（Job）

Job是K8s用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

后台支撑服务集（DaemonSet）

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在K8s集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支撑K8s集群运行的服务。

有状态服务集（StatefulSet）

K8s在1.3版本里发布了Alpha版的PetSet以支持有状态服务，并从1.5版本开始重命名为StatefulSet。在云原生应用的体系里，有下面两组近义词；第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC和RS主要是控制提供无状态服务的，其所控制的Pod的名字是随机设置的，一个Pod出故障了就被丢弃掉，在另一个地方重启一个新的Pod，名字变了、名字和启动在哪儿都不重要，重要的只是Pod总数；而StatefulSet是用来控制有状态服务，StatefulSet中的每个Pod的名字都是事先确定的，不能更改。StatefulSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。

对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于StatefulSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂载上原来Pod的存储继续以它的状态提供服务。

适合于StatefulSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务Zookeeper、etcd等有状态服务。StatefulSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用StatefulSet，Pod仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，StatefulSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。StatefulSet还只在Alpha阶段，后面的设计如何演变，我们还要继续观察。

集群联邦（Federation）

K8s在1.3版本里发布了beta版的Federation功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。K8s的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足K8s的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商K8s集群服务而设计的。

每个K8s Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员K8s Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子K8s Cluster都创建一份对应的API对象。在提供业务请求服务时，K8s Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体K8s Cluster的业务请求，会依照这个K8s Cluster独立提供服务时一样的调度模式去做K8s Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

所有的设计都尽量不影响K8s Cluster现有的工作机制，这样对于每个子K8s集群来说，并不需要更外层的有一个K8s Federation，也就是意味着所有现有的K8s代码和机制不需要因为Federation功能有任何变化。

存储卷（Volume）

K8s集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而K8s的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。K8s支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录hostPath和NFS。K8s还支持使用Persistent Volume Claim即PVC这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如AWS，Google或GlusterFS和Ceph），而将有关存储实际技术的配置交给存储管理员通过Persistent Volume来配置。

持久存储卷（Persistent Volume，PV）和持久存储卷声明（Persistent Volume Claim，PVC）

PV和PVC使得K8s集群具备了存储的逻辑抽象能力，使得在配置Pod的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给PV的配置者，即集群的管理者。存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由K8s集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，由K8s集群的使用者即服务的管理员来配置。

节点（Node）

K8s集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。K8s集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

密钥对象（Secret）

Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在K8s集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴露机会。

用户帐户（User Account）和服务帐户（Service Account）

顾名思义，用户帐户为人提供账户标识，而服务账户为计算机进程和K8s集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，所以用户账户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

名字空间（Namespace）

名字空间为K8s集群提供虚拟的隔离作用，K8s集群初始有两个名字空间，分别是默认名字空间default和系统名字空间kube-system，除此以外，管理员可以创建新的名字空间满足需要。

RBAC访问授权

K8s在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control，RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control，ABAC），RBAC主要是引入了角色（Role）和角色绑定（RoleBinding）的抽象概念。在ABAC中，K8s集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户在跟一个或多个角色相关联。显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和使用更容易扩展和重用。

总结

从K8s的系统架构、技术概念和设计理念，我们可以看到K8s系统最核心的两个设计理念：一个是**容错性**，一个是**易扩展性**。容错性实际是保证K8s系统稳定性和安全性的基础，易扩展性是保证K8s对变更友好，可以快速迭代增加新功能的基础。

参考文档

- Kubernetes Design Principles
- Kubernetes与云原生应用

核心组件

components

Kubernetes 主要由以下几个核心组件组成:

- etcd 保存了整个集群的状态；
- apiserver 提供了资源操作的唯一入口，并提供认证、授权、访问控制、API 注册和发现等机制；

- controller manager 负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler 负责资源的调度，按照预定的调度策略将 Pod 调度到相应的机器上；
- kubelet 负责维护容器的生命周期，同时也负责 Volume（CVI）和网络（CNI）的管理；
- Container runtime 负责镜像管理以及 Pod 和容器的真正运行（CRI）；
- kube-proxy 负责为 Service 提供 cluster 内部的服务发现和负载均衡；

组件通信

Kubernetes 多组件之间的通信原理为

- apiserver 负责 etcd 存储的所有操作，且只有 apiserver 才直接操作 etcd 集群
- apiserver 对内（集群中的其他组件）和对外（用户）提供统一的 REST API，其他组件均通过 apiserver 进行通信
  - controller manager、scheduler、kube-proxy 和 kubelet 等均通过 apiserver watch API 监测资源变化情况，并对资源作相应的操作
  - 所有需要更新资源状态的操作均通过 apiserver 的 REST API 进行
- apiserver 也会直接调用 kubelet API（如 logs, exec, attach 等），默认不校验 kubelet 证书，但可以通过 `--kubelet-certificate-authority` 开启（而 GKE 通过 SSH 隧道保护它们之间的通信）

比如典型的创建 Pod 的流程为

1. 用户通过 REST API 创建一个 Pod
2. apiserver 将其写入 etcd
3. scheduluer 检测到未绑定 Node 的 Pod，开始调度并更新 Pod 的 Node 绑定
4. kubelet 检测到有新的 Pod 调度过来，通过 container runtime 运行该 Pod
5. kubelet 通过 container runtime 取到 Pod 状态，并更新到 apiserver 中

端口号

ports

Master node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	6443*	Kubernetes API server
TCP	Inbound	8080	Kubernetes API insecure server
TCP	Inbound	2379-2380	etcd server client API
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10251	kube-scheduler healthz
TCP	Inbound	10252	kube-controller-manager healthz
TCP	Inbound	10253	cloud-controller-manager healthz
TCP	Inbound	10255	Read-only Kubelet API
TCP	Inbound	10256	kube-proxy healthz

Worker node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	4194	Kubelet cAdvisor
TCP	Inbound	10248	Kubelet healthz
TCP	Inbound	10249	kube-proxy metrics
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10255	Read-only Kubelet API
TCP	Inbound	10256	kube-proxy healthz
TCP	Inbound	30000-32767	NodePort Services**

参考文档

- Master-Node communication
- Core Kubernetes: Jazz Improv over Orchestration
- Installing kubeadm

etcd

Etcd 是 CoreOS 基于 Raft 开发的分布式 key-value 存储，可用于服务发现、共享配置以及一致性保障（如数据库选主、分布式锁等）。

Etcd 主要功能

- 基本的 key-value 存储
- 监听机制
- key 的过期及续约机制，用于监控和服务发现
- 原子 CAS 和 CAD，用于分布式锁和 leader 选举

Etcd 基于 RAFT 的一致性

选举方法

- 1) 初始启动时，节点处于 follower 状态并被设定一个 election timeout，如果在这一时间周期内没有收到来自 leader 的 heartbeat，节点将发起选举：将自己切换为 candidate 之后，向集群中其它 follower 节点发送请求，询问其是否选举自己成为 leader。
- 2) 当收到来自集群中过半数节点的接受投票后，节点即成为 leader，开始接收保存 client 的数据并向其它的 follower 节点同步日志。如果没有达成一致，则 candidate 随机选择一个等待间隔（150ms ~ 300ms）再次发起投票，得到集群中半数以上 follower 接受的 candidate 将成为 leader
- 3) leader 节点依靠定时向 follower 发送 heartbeat 来保持其地位。
- 4) 任何时候如果其它 follower 在 election timeout 期间都没有收到来自 leader 的 heartbeat，同样会将自己的状态切换为 candidate 并发起选举。每成功选举一次，新 leader 的任期（Term）都会比之前 leader 的任期大 1。

日志复制

当前 Leader 收到客户端的日志（事务请求）后先把该日志追加到本地的 Log 中，然后通过 heartbeat 把该 Entry 同步给其他 Follower，Follower 接收到日志后记录日志然后向 Leader 发送 ACK，当 Leader 收到大多数（n/2+1）Follower 的 ACK 信息后将该日志设置为已提交并追加到本地磁盘中，通知客户端并在下个 heartbeat 中 Leader 将通知所有的 Follower 将该日志存储在自己的本地磁盘中。

安全性

安全性是用于保证每个节点都执行相同序列的安全机制，如当某个 Follower 在当前 Leader commit Log 时变得不可用了，稍后可能该 Follower 又会被选举为 Leader，这时新 Leader 可能会用新的 Log 覆盖先前已 committed 的 Log，这就是导致节点执行不同序列；Safety 就是用于保证选举出来的 Leader 一定包含先前 committed Log 的机制；

- 选举安全性（Election Safety）：每个任期（Term）只能选举出一个 Leader
- Leader 完整性（Leader Completeness）：指 Leader 日志的完整性，当 Log 在任期 Term1 被 Commit 后，那么以后任期 Term2、Term3… 等的 Leader 必须包含该 Log；Raft 在选举阶段就使用 Term 的判断用于保证完整性：当请求投票的该 Candidate 的 Term 较大或 Term 相同 Index 更大则投票，否则拒绝该请求。

失效处理

- 1) Leader 失效：其他没有收到 heartbeat 的节点会发起新的选举，而当 Leader 恢复后由于步进数小会自动成为 follower（日志也会被新 leader 的日志覆盖）
- 2）follower 节点不可用：follower 节点不可用的情况相对容易解决。因为集群中的日志内容始终是从 leader 节点同步的，只要这一节点再次加入集群时重新从 leader 节点处复制日志即可。
- 3）多个 candidate：冲突后 candidate 将随机选择一个等待间隔（150ms ~ 300ms）再次发起投票，得到集群中半数以上 follower 接受的 candidate 将成为 leader

wal 日志

Etcd 实现 raft 的时候，充分利用了 go 语言 CSP 并发模型和 chan 的魔法，想更进行一步了解的可以去看源码，这里只简单分析下它的 wal 日志。

etcdv3

wal 日志是二进制的，解析出来后是以上数据结构 LogEntry。其中第一个字段 type，只有两种，一种是 0 表示 Normal，1 表示 ConfChange（ConfChange 表示 Etcd 本身的配置变更同步，比如有新的节点加入等）。第二个字段是 term，每个 term 代表一个主节点的任期，每次主节点变更 term 就会变化。第三个字段是 index，这个序号是严格有序递增的，代表变更序号。第四个字段是二进制的 data，将 raft request 对象的 pb 结构整个保存下。Etcd 源码下有个 tools/etcd-dump-logs，可以将 wal 日志 dump 成文本查看，可以协助分析 raft 协议。

raft 协议本身不关心应用数据，也就是 data 中的部分，一致性都通过同步 wal 日志来实现，每个节点将从主节点收到的 data apply 到本地的存储，raft 只关心日志的同步状态，如果本地存储实现的有 bug，比如没有正确的将 data apply 到本地，也可能会导致数据不一致。

Etcd v2 与 v3

Etcd v2 和 v3 本质上是共享同一套 raft 协议代码的两个独立的应用，接口不一样，存储不一样，数据互相隔离。也就是说如果从 Etcd v2 升级到 Etcd v3，原来 v2 的数据还是只能用 v2 的接口访问，v3 的接口创建的数据也只能访问通过 v3 的接口访问。所以我们按照 v2 和 v3 分别分析。

推荐在 Kubernetes 集群中使用 **Etcd v3**，**v2 版本已在 Kubernetes v1.11 中弃用**。

Etcd v2 存储，Watch 以及过期机制

etcdv2

Etcd v2 是个纯内存的实现，并未实时将数据写入到磁盘，持久化机制很简单，就是将 store 整合序列化成 json 写入文件。数据在内存中是一个简单的树结构。比如以下数据存储到 Etcd 中的结构就如图所示。

```
/nodes/1/name  node1
/nodes/1/ip    192.168.1.1
```

store 中有一个全局的 currentIndex，每次变更，index 会加 1. 然后每个 event 都会关联到 currentIndex.

当客户端调用 watch 接口（参数中增加 wait 参数）时，如果请求参数中有 waitIndex，并且 waitIndex 小于 currentIndex，则从 EventHistroy 表中查询 index 大于等于 waitIndex，并且和 watch key 匹配的 event，如果有数据，则直接返回。如果历史表中没有或者请求没有带 waitIndex，则放入 WatchHub 中，每个 key 会关联一个 watcher 列表。 当有变更操作时，变更生成的 event 会放入 EventHistroy 表中，同时通知和该 key 相关的 watcher。

这里有几个影响使用的细节问题：

1. EventHistroy 是有长度限制的，最长 1000。也就是说，如果你的客户端停了许久，然后重新 watch 的时候，可能和该 waitIndex 相关的 event 已经被淘汰了，这种情况下会丢失变更。
2. 如果通知 watcher 的时候，出现了阻塞（每个 watcher 的 channel 有 100 个缓冲空间），Etcd 会直接把 watcher 删除，也就是会导致 wait 请求的连接中断，客户端需要重新连接。
3. Etcd store 的每个 node 中都保存了过期时间，通过定时机制进行清理。

从而可以看出，Etcd v2 的一些限制：

1. 过期时间只能设置到每个 key 上，如果多个 key 要保证生命周期一致则比较困难。
2. watcher 只能 watch 某一个 key 以及其子节点（通过参数 recursive)，不能进行多个 watch。
3. 很难通过 watch 机制来实现完整的数据同步（有丢失变更的风险），所以当前的大多数使用方式是通过 watch 得知变更，然后通过 get 重新获取数据，并不完全依赖于 watch 的变更 event。

Etcd v3 存储，Watch 以及过期机制

etcdv3

Etcd v3 将 watch 和 store 拆开实现，我们先分析下 store 的实现。

Etcd v3 store 分为两部分，一部分是内存中的索引，kvindex，是基于 google 开源的一个 golang 的 btree 实现的，另外一部分是后端存储。按照它的设计，backend 可以对接多种存储，当前使用的 boltdb。boltdb 是一个单机的支持事务的 kv 存储，Etcd 的事务是基于 boltdb 的事务实现的。Etcd 在 boltdb 中存储的 key 是 revision，value 是 Etcd 自己的 key-value 组合，也就是说 Etcd 会在 boltdb 中把每个版本都保存下，从而实现了多版本机制。

举个例子： 用 etcdctl 通过批量接口写入两条记录：

```
etcdctl txn <<<'
put key1 "v1"
put key2 "v2"

'
```

再通过批量接口更新这两条记录：

```
etcdctl txn <<<'
put key1 "v12"
put key2 "v22"

'
```

boltdb 中其实有了 4 条数据：

```
rev={3 0}, key=key1, value="v1"
rev={3 1}, key=key2, value="v2"
rev={4 0}, key=key1, value="v12"
rev={4 1}, key=key2, value="v22"
```

revision 主要由两部分组成，第一部分 main rev，每次事务进行加一，第二部分 sub rev，同一个事务中的每次操作加一。如上示例，第一次操作的 main rev 是 3，第二次是 4。当然这种机制大家想到的第一个问题就是空间问题，所以 Etcd 提供了命令和设置选项来控制 compact，同时支持 put 操作的参数来精确控制某个 key 的历史版本数。

了解了 Etcd 的磁盘存储，可以看出如果要从 boltdb 中查询数据，必须通过 revision，但客户端都是通过 key 来查询 value，所以 Etcd 的内存 kvindex 保存的就是 key 和 revision 之前的映射关系，用来加速查询。

然后我们再分析下 watch 机制的实现。Etcd v3 的 watch 机制支持 watch 某个固定的 key，也支持 watch 一个范围（可以用于模拟目录的结构的 watch），所以 watchGroup 包含两种 watcher，一种是 key watchers，数据结构是每个 key 对应一组 watcher，另外一种是 range watchers, 数据结构是一个 IntervalTree（不熟悉的参看文文末链接），方便通过区间查找到对应的 watcher。

同时，每个 WatchableStore 包含两种 watcherGroup，一种是 synced，一种是 unsynced，前者表示该 group 的 watcher 数据都已经同步完毕，在等待新的变更，后者表示该 group 的 watcher 数据同步落后于当前最新变更，还在追赶。

当 Etcd 收到客户端的 watch 请求，如果请求携带了 revision 参数，则比较请求的 revision 和 store 当前的 revision，如果大于当前 revision，则放入 synced 组中，否则放入 unsynced 组。同时 Etcd 会启动一个后台的 goroutine 持续同步 unsynced 的 watcher，然后将其迁移到 synced 组。也就是这种机制下，Etcd v3 支持从任意版本开始 watch，没有 v2 的 1000 条历史 event 表限制的问题（当然这是指没有 compact 的情况下）。

另外我们前面提到的，Etcd v2 在通知客户端时，如果网络不好或者客户端读取比较慢，发生了阻塞，则会直接关闭当前连接，客户端需要重新发起请求。Etcd v3 为了解决这个问题，专门维护了一个推送时阻塞的 watcher 队列，在另外的 goroutine 里进行重试。

Etcd v3 对过期机制也做了改进，过期时间设置在 lease 上，然后 key 和 lease 关联。这样可以实现多个 key 关联同一个 lease id，方便设置统一的过期时间，以及实现批量续约。

相比 Etcd v2, Etcd v3 的一些主要变化：

1. 接口通过 grpc 提供 rpc 接口，放弃了 v2 的 http 接口。优势是长连接效率提升明显，缺点是使用不如以前方便，尤其对不方便维护长连接的场景。
2. 废弃了原来的目录结构，变成了纯粹的 kv，用户可以通过前缀匹配模式模拟目录。
3. 内存中不再保存 value，同样的内存可以支持存储更多的 key。
4. watch 机制更稳定，基本上可以通过 watch 机制实现数据的完全同步。
5. 提供了批量操作以及事务机制，用户可以通过批量事务请求来实现 Etcd v2 的 CAS 机制（批量事务支持 if 条件判断）。

Etcd，Zookeeper，Consul 比较

- Etcd 和 Zookeeper 提供的能力非常相似，都是通用的一致性元信息存储，都提供 watch 机制用于变更通知和分发，也都被分布式系统用来作为共享信息存储，在软件生态中所处的位置也几乎是一样的，可以互相替代的。二者除了实现细节，语言，一致性协议上的区别，最大的区别在周边生态圈。Zookeeper 是 apache 下的，用 java 写的，提供 rpc 接口，最早从 hadoop 项目中孵化出来，在分布式系统中得到广泛使用（hadoop, solr, kafka, mesos 等）。Etcd 是 coreos 公司旗下的开源产品，比较新，

以其简单好用的 rest 接口以及活跃的社区俘获了一批用户，在新的一些集群中得到使用（比如 kubernetes）。虽然 v3 为了性能也改成二进制 rpc 接口了，但其易用性上比 Zookeeper 还是好一些。

- 而 Consul 的目标则更为具体一些，Etcd 和 Zookeeper 提供的是分布式一致性存储能力，具体的业务场景需要用户自己实现，比如服务发现，比如配置变更。而 Consul 则以服务发现和配置变更为主要目标，同时附带了 kv 存储。

Etcd 的周边工具

1. **Confd**

   在分布式系统中，理想情况下是应用程序直接和 Etcd 这样的服务发现 / 配置中心交互，通过监听 Etcd 进行服务发现以及配置变更。但我们还有许多历史遗留的程序，服务发现以及配置大多都是通过变更配置文件进行的。Etcd 自己的定位是通用的 kv 存储，所以并没有像 Consul 那样提供实现配置变更的机制和工具，而 Confd 就是用来实现这个目标的工具。

   Confd 通过 watch 机制监听 Etcd 的变更，然后将数据同步到自己的一个本地存储。用户可以通过配置定义自己关注哪些 key 的变更，同时提供一个配置文件模板。Confd 一旦发现数据变更就使用最新数据渲染模板生成配置文件，如果新旧配置文件有变化，则进行替换，同时触发用户提供的 reload 脚本，让应用程序重新加载配置。

   Confd 相当于实现了部分 Consul 的 agent 以及 consul-template 的功能，作者是 kubernetes 的 Kelsey Hightower，但大神貌似很忙，没太多时间关注这个项目了，很久没有发布版本，我们着急用，所以 fork 了一份自己更新维护，主要增加了一些新的模板函数以及对 metad 后端的支持。conf d

2. **Metad**

   服务注册的实现模式一般分为两种，一种是调度系统代为注册，一种是应用程序自己注册。调度系统代为注册的情况下，应用程序启动后需要有一种机制让应用程序知道『我是谁』，然后发现自己所在的集群以及自己的配置。Metad 提供这样一种机制，客户端请求 Metad 的一个固定的接口 / self，由 Metad 告知应用程序其所属的元信息，简化了客户端的服务发现和配置变更逻辑。

Metad 通过保存一个 ip 到元信息路径的映射关系来做到这一点，当前后端支持 Etcd v3，提供简单好用的 http rest 接口。 它会把 Etcd 的数据通过 watch 机制同步到本地内存中，相当于 Etcd 的一个代理。所以也可以把它当做 Etcd 的代理来使用，适用于不方便使用 Etcd v3 的 rpc 接口或者想降低 Etcd 压力的场景。 metad

Etcd 使用注意事项

1. Etcd cluster 初始化的问题

   如果集群第一次初始化启动的时候，有一台节点未启动，通过 v3 的接口访问的时候，会报告 Error: Etcdserver: not capable 错误。这是为兼容性考虑，集群启动时默认的 API 版本是 2.3，只有当集群中的所有节点都加入了，确认所有节点都支持 v3 接口时，才提升集群版本到 v3。这个只有第一次初始化集群的时候会遇到，如果集群已经初始化完毕，再挂掉节点，或者集群关闭重启（关闭重启的时候会从持久化数据中加载集群 API 版本），都不会有影响。

2. Etcd 读请求的机制

   v2 quorum=true 的时候，读取是通过 raft 进行的，通过 cli 请求，该参数默认为 true。

   v3 --consistency="l" 的时候（默认）通过 raft 读取，否则读取本地数据。sdk 代码里则是通过是否打开：WithSerializable option 来控制。

   一致性读取的情况下，每次读取也需要走一次 raft 协议，能保证一致性，但性能有损失，如果出现网络分区，集群的少数节点是不能提供一致性读取的。但如果不设置该参数，则是直接从本地的 store 里读取，这样就损失了一致性。使用的时候需要注意根据应用场景设置这个参数，在一致性和可用性之间进行取舍。

3. Etcd 的 compact 机制

   Etcd 默认不会自动 compact，需要设置启动参数，或者通过命令进行 compact，如果变更频繁建议设置，否则会导致空间和内存的浪费以及错误。Etcd v3 的默认的 backend quota 2GB，如果不 compact，boltdb 文件大小超过这个限制后，就会报错："Error: etcdserver: mvcc: database space exceeded"，导致数据无法写入。

etcd 的问题

```
当前 Etcd 的 raft 实现保证了多个节点数据之间的同步，但明显的一个问题就是扩充节点
```

注： 部分转自 jolestar 和infoq.

参考文档

- Etcd website
- Etcd github
- Projects using etcd
- http://jolestar.com/etcd-architecture/
- etcd 从应用场景到实现原理的全方位解读

kube-apiserver

kube-apiserver 是 Kubernetes 最重要的核心组件之一，主要提供以下的功能

- 提供集群管理的 REST API 接口，包括认证授权、数据校验以及集群状态变更等
- 提供其他模块之间的数据交互和通信的枢纽（其他模块通过 API Server 查询或修改数据，只有 API Server 才直接操作 etcd）

REST API

kube-apiserver 支持同时提供 https（默认监听在 6443 端口）和 http API（默认监听在 127.0.0.1 的 8080 端口），其中 http API 是非安全接口，不做任何认证授权机制，不建议生产环境启用。两个接口提供的 REST API 格式相同，参考 Kubernetes API Reference 查看所有 API 的调用格式。

在实际使用中，通常通过 kubectl 来访问 apiserver，也可以通过 Kubernetes 各个语言的 client 库来访问 apiserver。在使用 kubectl 时，打开调试日志也可以看到每个 API 调用的格式，比如

```
$ kubectl --v=8 get pods
```

可通过 `kubectl api-versions` 和 `kubectl api-resources` 查询 Kubernetes API 支持的 API 版本以及资源对象。

```
$ kubectl api-versions
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
apps/v1
apps/v1beta1
apps/v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling/v1
autoscaling/v2beta1
batch/v1
batch/v1beta1
certificates.k8s.io/v1beta1
events.k8s.io/v1beta1
extensions/v1beta1
metrics.k8s.io/v1beta1
networking.k8s.io/v1
policy/v1beta1
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
scheduling.k8s.io/v1beta1
storage.k8s.io/v1
storage.k8s.io/v1beta1
v1

$ kubectl api-resources --api-group=storage.k8s.io
NAME                SHORTNAMES   APIGROUP          NAMESPACED   KI
storageclasses      sc           storage.k8s.io    false        St
volumeattachments                storage.k8s.io    false        Vo
```

OpenAPI 和 Swagger

通过 `/swaggerapi` 可以查看 Swagger API，`/openapi/v2` 查看 OpenAPI。

开启 `--enable-swagger-ui=true` 后还可以通过 `/swagger-ui` 访问 Swagger UI。

根据 OpenAPI 也可以生成各种语言的客户端，比如可以用下面的命令生成 Go 语言的客户端：

```
git clone https://github.com/kubernetes-client/gen /tmp/gen
cat >go.settings <# Kubernetes branch name
export KUBERNETES_BRANCH="release-1.11"

# client version for packaging and releasing.
export CLIENT_VERSION="1.0"

# Name of the release package
export PACKAGE_NAME="client-go"
EOF

/tmp/gen/openapi/go.sh ./client-go ./go.settings
```

访问控制

Kubernetes API 的每个请求都会经过多阶段的访问控制之后才会被接受，这包括认证、授权以及准入控制（Admission Control）等。

认证

开启 TLS 时，所有的请求都需要首先认证。Kubernetes 支持多种认证机制，并支持同时开启多个认证插件（只要有一个认证通过即可）。如果认证成功，则用户的 `username` 会传入授权模块做进一步授权验证；而对于认证失败的请求则返回 HTTP 401。

> **Kubernetes 不直接管理用户**
>
> 虽然 Kubernetes 认证和授权用到了 username，但 Kubernetes 并不直接管理用户，不能创建 `user` 对象，也不存储 username。

更多认证模块的使用方法可以参考 Kubernetes 认证插件。

授权

认证之后的请求就到了授权模块。跟认证类似，Kubernetes 也支持多种授权机制，并支持同时开启多个授权插件（只要有一个验证通过即可）。如果授权成功，则用户的请求会发送到准入控制模块做进一步的请求验证；而对于授权失败的请求则返回 HTTP 403.

更多授权模块的使用方法可以参考 Kubernetes 授权插件。

准入控制

准入控制（Admission Control）用来对请求做进一步的验证或添加默认参数。不同于授权和认证只关心请求的用户和操作，准入控制还处理请求的内容，并且仅对创建、更新、删除或连接（如代理）等有效，而对读操作无效。准入控制也支持同时开启多个插件，它们依次调用，只有全部插件都通过的请求才可以放过进入系统。

更多准入控制模块的使用方法可以参考 Kubernetes 准入控制。

启动 apiserver 示例

```
kube-apiserver --feature-gates=AllAlpha=true --runtime-config=api
    --requestheader-allowed-names=front-proxy-client \
    --client-ca-file=/etc/kubernetes/pki/ca.crt \
    --allow-privileged=true \
    --experimental-bootstrap-token-auth=true \
    --storage-backend=etcd3 \
    --requestheader-username-headers=X-Remote-User \
    --requestheader-extra-headers-prefix=X-Remote-Extra- \
    --service-account-key-file=/etc/kubernetes/pki/sa.pub \
    --tls-cert-file=/etc/kubernetes/pki/apiserver.crt \
    --tls-private-key-file=/etc/kubernetes/pki/apiserver.key \
    --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-ku
    --requestheader-client-ca-file=/etc/kubernetes/pki/front-prox
    --insecure-port=8080 \
    --admission-control=NamespaceLifecycle,LimitRanger,ServiceAcc
    --requestheader-group-headers=X-Remote-Group \
    --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-cl
    --secure-port=6443 \
    --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostr
    --service-cluster-ip-range=10.96.0.0/12 \
    --authorization-mode=RBAC \
    --advertise-address=192.168.0.20 --etcd-servers=http://127.0.
```

工作原理

kube-apiserver 提供了 Kubernetes 的 REST API，实现了认证、授权、准入控制等安全校验功能，同时也负责集群状态的存储操作（通过 etcd）。

API 访问

有多种方式可以访问 Kubernetes 提供的 REST API：

- kubectl 命令行工具

- SDK，支持多种语言
  - Go
  - Python
  - Javascript
  - Java
  - CSharp
  - 其他 OpenAPI 支持的语言，可以通过 gen 工具生成相应的 client

kubectl

```
kubectl get --raw /api/v1/namespaces
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes
kubectl get --raw /apis/metrics.k8s.io/v1beta1/pods
```

kubectl proxy

```
$ kubectl proxy --port=8080 &

$ curl http://localhost:8080/api/
{
  "versions": [
    "v1"
  ]
}
```

curl

```
# In Pods with service account.
$ TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)
$ CACERT=/run/secrets/kubernetes.io/serviceaccount/ca.crt
$ curl --cacert $CACERT --header "Authorization: Bearer $TOKEN"
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

```
# Outside of Pods.
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep de
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --i
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

API 参考文档

最近 4 个稳定版本的 API 参考文档为：

- v1.11 API Reference
- v1.10 API Reference
- v1.9 API Reference
- v1.8 API Reference

kube-scheduler

kube-scheduler 负责分配调度 Pod 到集群内的节点上，它监听 kube-apiserver，查询还未分配 Node 的 Pod，然后根据调度策略为这些 Pod 分配节点（更新 Pod 的 `NodeName` 字段）。

调度器需要充分考虑诸多的因素：

- 公平调度
- 资源高效利用
- QoS
- affinity 和 anti-affinity
- 数据本地化（data locality）
- 内部负载干扰（inter-workload interference）
- deadlines

指定 Node 节点调度

有三种方式指定 Pod 只运行在指定的 Node 节点上

- nodeSelector：只调度到匹配指定 label 的 Node 上
- nodeAffinity：功能更丰富的 Node 选择器，比如支持集合操作
- podAffinity：调度到满足条件的 Pod 所在的 Node 上

nodeSelector 示例

首先给 Node 打上标签

```
kubectl label nodes node-01 disktype=ssd
```

然后在 daemonset 中指定 nodeSelector 为 `disktype=ssd` ：

```
spec:
  nodeSelector:
    disktype: ssd
```

nodeAffinity 示例

nodeAffinity 目前支持两种：requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution，分别代表必须满足条件和优选条件。比如下面的例子代表调度到包含标签 `kubernetes.io/e2e-az-name` 并且值为 e2e-az1 或 e2e-az2 的 Node 上，并且优选还带有标签 `another-node-label-key=another-node-label-value` 的 Node。

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: gcr.io/google_containers/pause:2.0
```

podAffinity 示例

podAffinity 基于 Pod 的标签来选择 Node，仅调度到满足条件 Pod 所在的 Node 上，支持 podAffinity 和 podAntiAffinity。这个功能比较绕，以下面的例子为例：

- 如果一个 “Node 所在 Zone 中包含至少一个带有 `security=S1` 标签且运行中的 Pod”，那么可以调度到该 Node
- 不调度到 “包含至少一个带有 `security=S2` 标签且运行中 Pod” 的 Node 上

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: gcr.io/google_containers/pause:2.0
```

Taints 和 tolerations

Taints 和 tolerations 用于保证 Pod 不被调度到不合适的 Node 上，其中 Taint 应用于 Node 上，而 toleration 则应用于 Pod 上。

目前支持的 taint 类型

- NoSchedule：新的 Pod 不调度到该 Node 上，不影响正在运行的 Pod
- PreferNoSchedule：soft 版的 NoSchedule，尽量不调度到该 Node 上

- NoExecute：新的 Pod 不调度到该 Node 上，并且删除（evict）已在运行的 Pod。Pod 可以增加一个时间（tolerationSeconds），

然而，当 Pod 的 Tolerations 匹配 Node 的所有 Taints 的时候可以调度到该 Node 上；当 Pod 是已经运行的时候，也不会被删除（evicted）。另外对于 NoExecute，如果 Pod 增加了一个 tolerationSeconds，则会在该时间之后才删除 Pod。

比如，假设 node1 上应用以下几个 taint

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

下面的这个 Pod 由于没有 tolerate `key2=value2:NoSchedule` 无法调度到 node1 上

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

而正在运行且带有 tolerationSeconds 的 Pod 则会在 600s 之后删除

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 600
- key: "key2"
  operator: "Equal"
  value: "value2"
  effect: "NoSchedule"
```

注意，DaemonSet 创建的 Pod 会自动加上对 `node.alpha.kubernetes.io/unreachable` 和 `node.alpha.kubernetes.io/notReady` 的 NoExecute Toleration，以避免它们因此被删除。

优先级调度

从 v1.8 开始，kube-scheduler 支持定义 Pod 的优先级，从而保证高优先级的 Pod 优先调度。并从 v1.11 开始默认开启。

> 注：在 v1.8-v1.10 版本中的开启方法为
>
> - apiserver 配置 `--feature-gates=PodPriority=true` 和 `--runtime-config=scheduling.k8s.io/v1alpha1=true`
> - kube-scheduler 配置 `--feature-gates=PodPriority=true`

在指定 Pod 的优先级之前需要先定义一个 PriorityClass（非 namespace 资源），如

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service
```

其中

- `value` 为 32 位整数的优先级，该值越大，优先级越高
- `globalDefault` 用于未配置 PriorityClassName 的 Pod，整个集群中应该只有一个 PriorityClass 将其设置为 true

然后，在 PodSpec 中通过 PriorityClassName 设置 Pod 的优先级：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

多调度器

如果默认的调度器不满足要求，还可以部署自定义的调度器。并且，在整个集群中还可以同时运行多个调度器实例，通过 `podSpec.schedulerName` 来选择使用哪一个调度器（默认使用内置的调度器）。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  # 选择使用自定义调度器 my-scheduler
  schedulerName: my-scheduler
  containers:
  - name: nginx
    image: nginx:1.10
```

调度器的示例参见 这里。

调度器扩展

kube-scheduler 还支持使用 `--policy-config-file` 指定一个调度策略文件来自定义调度策略，比如

```
{
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [
    {"name" : "PodFitsHostPorts"},
    {"name" : "PodFitsResources"},
    {"name" : "NoDiskConflict"},
    {"name" : "MatchNodeSelector"},
    {"name" : "HostName"}
    ],
"priorities" : [
    {"name" : "LeastRequestedPriority", "weight" : 1},
    {"name" : "BalancedResourceAllocation", "weight" : 1},
    {"name" : "ServiceSpreadingPriority", "weight" : 1},
    {"name" : "EqualPriority", "weight" : 1}
    ],
"extenders":[
    {
        "urlPrefix": "http://127.0.0.1:12346/scheduler",
        "apiVersion": "v1beta1",
        "filterVerb": "filter",
        "prioritizeVerb": "prioritize",
        "weight": 5,
        "enableHttps": false,
        "nodeCacheCapable": false
    }
    ]
}
```

其他影响调度的因素

- 如果 Node Condition 处于 MemoryPressure，则所有 BestEffort 的新 Pod（未指定 resources limits 和 requests）不会调度到该 Node 上
- 如果 Node Condition 处于 DiskPressure，则所有新 Pod 都不会调度到该 Node 上
- 为了保证 Critical Pods 的正常运行，当它们处于异常状态时会自动重新调度。Critical Pods 是指
  - annotation 包括 `scheduler.alpha.kubernetes.io/critical-pod=''`
  - tolerations 包括 `[{"key":"CriticalAddonsOnly", "operator":"Exists"}]`
  - priorityClass 为 `system-cluster-critical` 或者 `system-node-critical`

启动 kube-scheduler 示例

```
kube-scheduler --address=127.0.0.1 --leader-elect=true --kubeconf
```

kube-scheduler 工作原理

kube-scheduler 调度原理：

```
For given pod:

    +-------------------------------------------+
    |               Schedulable nodes:          |
    |                                           |
    | +--------+    +--------+      +--------+  |
    | | node 1 |    | node 2 |      | node 3 |  |
    | +--------+    +--------+      +--------+  |
    |                                           |
    +-------------------+-----------------------+
                        |
                        |
                        v
    +-------------------+-----------------------+

    Pred. filters: node 3 doesn't have enough resource

    +-------------------+-----------------------+
                        |
                        |
                        v
    +-------------------+-----------------------+
    |             remaining nodes:              |
    |   +--------+                 +--------+   |
    |   | node 1 |                 | node 2 |   |
    |   +--------+                 +--------+   |
    |                                           |
    +-------------------+-----------------------+
                        |
                        |
                        v
    +-------------------+-----------------------+

    Priority function:    node 1: p=2
                          node 2: p=5

    +-------------------+-----------------------+
                        |
                        |
```

```
                    v
          select max{node priority} = node 2
```

kube-scheduler 调度分为两个阶段，predicate 和 priority

- predicate：过滤不符合条件的节点
- priority：优先级排序，选择优先级最高的节点

predicates 策略

- PodFitsPorts：同 PodFitsHostPorts
- PodFitsHostPorts：检查是否有 Host Ports 冲突
- PodFitsResources：检查 Node 的资源是否充足，包括允许的 Pod 数量、CPU、内存、GPU 个数以及其他的 OpaqueIntResources
- HostName：检查 `pod.Spec.NodeName` 是否与候选节点一致
- MatchNodeSelector：检查候选节点的 `pod.Spec.NodeSelector` 是否匹配
- NoVolumeZoneConflict：检查 volume zone 是否冲突
- MaxEBSVolumeCount：检查 AWS EBS Volume 数量是否过多（默认不超过 39）
- MaxGCEPDVolumeCount：检查 GCE PD Volume 数量是否过多（默认不超过 16）
- MaxAzureDiskVolumeCount：检查 Azure Disk Volume 数量是否过多（默认不超过 16）
- MatchInterPodAffinity：检查是否匹配 Pod 的亲和性要求
- NoDiskConflict：检查是否存在 Volume 冲突，仅限于 GCE PD、AWS EBS、Ceph RBD 以及 ISCSI
- GeneralPredicates：分为 noncriticalPredicates 和 EssentialPredicates。noncriticalPredicates 中包含 PodFitsResources，EssentialPredicates 中包含 PodFitsHost，PodFitsHostPorts 和 PodSelectorMatches。
- PodToleratesNodeTaints：检查 Pod 是否容忍 Node Taints
- CheckNodeMemoryPressure：检查 Pod 是否可以调度到 MemoryPressure 的节点上
- CheckNodeDiskPressure：检查 Pod 是否可以调度到 DiskPressure 的节点上
- NoVolumeNodeConflict：检查节点是否满足 Pod 所引用的 Volume 的条件

priorities 策略

- SelectorSpreadPriority：优先减少节点上属于同一个 Service 或 Replication Controller 的 Pod 数量
- InterPodAffinityPriority：优先将 Pod 调度到相同的拓扑上（如同一个节点、Rack、Zone 等）

- LeastRequestedPriority：优先调度到请求资源少的节点上
- BalancedResourceAllocation：优先平衡各节点的资源使用
- NodePreferAvoidPodsPriority：alpha.kubernetes.io/preferAvoidPods 字段判断，权重为 10000，避免其他优先级策略的影响
- NodeAffinityPriority：优先调度到匹配 NodeAffinity 的节点上
- TaintTolerationPriority：优先调度到匹配 TaintToleration 的节点上
- ServiceSpreadingPriority：尽量将同一个 service 的 Pod 分布到不同节点上，已经被 SelectorSpreadPriority 替代 [默认未使用]
- EqualPriority：将所有节点的优先级设置为 1[默认未使用]
- ImageLocalityPriority：尽量将使用大镜像的容器调度到已经下拉了该镜像的节点上 [默认未使用]
- MostRequestedPriority：尽量调度到已经使用过的 Node 上，特别适用于 cluster-autoscaler[默认未使用]

> **代码入口路径**
>
> 与 Kubernetes 其他组件的入口不同 (其他都是位于 `cmd/` 目录)，kube-schedular 的入口在 `plugin/cmd/kube-scheduler`。

参考文档

- Pod Priority and Preemption
- Configure Multiple Schedulers
- Taints and Tolerations
- Advanced Scheduling in Kubernetes

kube-controller-manager

Controller Manager 由 kube-controller-manager 和 cloud-controller-manager 组成，是 Kubernetes 的大脑，它通过 apiserver 监控整个集群的状态，并确保集群处于预期的工作状态。

kube-controller-manager 由一系列的控制器组成

- Replication Controller
- Node Controller
- CronJob Controller
- Daemon Controller
- Deployment Controller
- Endpoint Controller
- Garbage Collector

- Namespace Controller
- Job Controller
- Pod AutoScaler
- RelicaSet
- Service Controller
- ServiceAccount Controller
- StatefulSet Controller
- Volume Controller
- Resource quota Controller

cloud-controller-manager 在 Kubernetes 启用 Cloud Provider 的时候才需要，用来配合云服务提供商的控制，也包括一系列的控制器，如

- Node Controller
- Route Controller
- Service Controller

从 v1.6 开始，cloud provider 已经经历了几次重大重构，以便在不修改 Kubernetes 核心代码的同时构建自定义的云服务商支持。参考 这里 查看如何为云提供商构建新的 Cloud Provider。

Metrics

Controller manager metrics 提供了控制器内部逻辑的性能度量，如 Go 语言运行时度量、etcd 请求延时、云服务商 API 请求延时、云存储请求延时等。Controller manager metrics 默认监听在 `kube-controller-manager` 的 10252 端口，提供 Prometheus 格式的性能度量数据，可以通过 `http://localhost:10252/metrics` 来访问。

```
$ curl http://localhost:10252/metrics
...
# HELP etcd_request_cache_add_latencies_summary Latency in micros
# TYPE etcd_request_cache_add_latencies_summary summary
etcd_request_cache_add_latencies_summary{quantile="0.5"} NaN
etcd_request_cache_add_latencies_summary{quantile="0.9"} NaN
etcd_request_cache_add_latencies_summary{quantile="0.99"} NaN
etcd_request_cache_add_latencies_summary_sum 0
etcd_request_cache_add_latencies_summary_count 0
# HELP etcd_request_cache_get_latencies_summary Latency in micros
# TYPE etcd_request_cache_get_latencies_summary summary
etcd_request_cache_get_latencies_summary{quantile="0.5"} NaN
etcd_request_cache_get_latencies_summary{quantile="0.9"} NaN
etcd_request_cache_get_latencies_summary{quantile="0.99"} NaN
etcd_request_cache_get_latencies_summary_sum 0
etcd_request_cache_get_latencies_summary_count 0
...
```

kube-controller-manager 启动示例

```
ube-controller-manager \
  --enable-dynamic-provisioning=true \
  --feature-gates=AllAlpha=true \
  --horizontal-pod-autoscaler-sync-period=10s \
  --horizontal-pod-autoscaler-use-rest-clients=true \
  --node-monitor-grace-period=10s \
  --address=127.0.0.1 \
  --leader-elect=true \
  --kubeconfig=/etc/kubernetes/controller-manager.conf \
  --cluster-signing-key-file=/etc/kubernetes/pki/ca.key \
  --use-service-account-credentials=true \
  --controllers=*,bootstrapsigner,tokencleaner \
  --root-ca-file=/etc/kubernetes/pki/ca.crt \
  --service-account-private-key-file=/etc/kubernetes/pki/sa.key \
  --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt \
  --allocate-node-cidrs=true \
  --cluster-cidr=10.244.0.0/16 \
  --node-cidr-mask-size=24
```

控制器

kube-controller-manager

kube-controller-manager 由一系列的控制器组成，这些控制器可以划分为三组

1. 必须启动的控制器
    - EndpointController
    - ReplicationController：
    - PodGCController
    - ResourceQuotaController
    - NamespaceController
    - ServiceAccountController
    - GarbageCollectorController
    - DaemonSetController

    - JobController
    - DeploymentController
    - ReplicaSetController
    - HPAController
    - DisruptionController
    - StatefulSetController
    - CronJobController
    - CSRSigningController
    - CSRApprovingController
    - TTLController
2. 默认启动的可选控制器，可通过选项设置是否开启
    - TokenController
    - NodeController
    - ServiceController
    - RouteController
    - PVBinderController
    - AttachDetachController
3. 默认禁止的可选控制器，可通过选项设置是否开启
    - BootstrapSignerController
    - TokenCleanerController

cloud-controller-manager

cloud-controller-manager 在 Kubernetes 启用 Cloud Provider 的时候才需要，用来配合云服务提供商的控制，也包括一系列的控制器

- CloudNodeController
- RouteController
- ServiceController

高可用

在启动时设置 `--leader-elect=true` 后，controller manager 会使用多节点选主的方式选择主节点。只有主节点才会调用 `StartControllers()` 启动所有控制器，而其他从节点则仅执行选主算法。

多节点选主的实现方法见 leaderelection.go。它实现了两种资源锁（Endpoint 或 ConfigMap，kube-controller-manager 和 cloud-controller-manager 都使用 Endpoint 锁），通过更新资源的 Annotation（`control-plane.alpha.kubernetes.io/leader`），来确定主从关系。

高性能

从 Kubernetes 1.7 开始，所有需要监控资源变化情况的调用均推荐使用 Informer。Informer 提供了基于事件通知的只读缓存机制，可以注册资源变化的回调函数，并可以极大减少 API 的调用。

Informer 的使用方法可以参考 这里。

Node Eviction

Node 控制器在节点异常后，会按照默认的速率（`--node-eviction-rate=0.1`，即每10秒一个节点的速率）进行 Node 的驱逐。Node 控制器按照 Zone 将节点划分为不同的组，再跟进 Zone 的状态进行速率调整：

- Normal：所有节点都 Ready，默认速率驱逐。
- PartialDisruption：即超过33% 的节点 NotReady 的状态。当异常节点比例大于 `--unhealthy-zone-threshold=0.55` 时开始减慢速率：
  - 小集群（即节点数量小于 `--large-cluster-size-threshold=50`）：停止驱逐
  - 大集群，减慢速率为 `--secondary-node-eviction-rate=0.01`
- FullDisruption：所有节点都 NotReady，返回使用默认速率驱逐。但当所有 Zone 都处在 FullDisruption 时，停止驱逐。

kubelet

每个节点上都运行一个 kubelet 服务进程，默认监听 10250 端口，接收并执行 master 发来的指令，管理 Pod 及 Pod 中的容器。每个 kubelet 进程会在 API Server 上注册节点自身信息，定期向 master 节点汇报节点的资源使用情况，并通过 cAdvisor 监控节点和容器的资源。

节点管理

节点管理主要是节点自注册和节点状态更新：

- Kubelet 可以通过设置启动参数 --register-node 来确定是否向 API Server 注册自己；

- 如果 Kubelet 没有选择自注册模式，则需要用户自己配置 Node 资源信息，同时需要告知 Kubelet 集群上的 API Server 的位置；
- Kubelet 在启动时通过 API Server 注册节点信息，并定时向 API Server 发送节点新消息，API Server 在接收到新消息后，将信息写入 etcd

Pod 管理

获取 Pod 清单

Kubelet 以 PodSpec 的方式工作。PodSpec 是描述一个 Pod 的 YAML 或 JSON 对象。 kubelet 采用一组通过各种机制提供的 PodSpecs（主要通过 apiserver），并确保这些 PodSpecs 中描述的 Pod 正常健康运行。

向 Kubelet 提供节点上需要运行的 Pod 清单的方法：

- 文件：启动参数 --config 指定的配置目录下的文件 (默认 / etc/kubernetes/manifests/)。该文件每 20 秒重新检查一次（可配置）。
- HTTP endpoint (URL)：启动参数 --manifest-url 设置。每 20 秒检查一次这个端点（可配置）。
- API Server：通过 API Server 监听 etcd 目录，同步 Pod 清单。
- HTTP server：kubelet 侦听 HTTP 请求，并响应简单的 API 以提交新的 Pod 清单。

通过 API Server 获取 Pod 清单及创建 Pod 的过程

Kubelet 通过 API Server Client(Kubelet 启动时创建)使用 Watch 加 List 的方式监听 "/registry/nodes/$ 当前节点名" 和 “/registry/pods” 目录，将获取的信息同步到本地缓存中。

Kubelet 监听 etcd，所有针对 Pod 的操作都将会被 Kubelet 监听到。如果发现有新的绑定到本节点的 Pod，则按照 Pod 清单的要求创建该 Pod。

如果发现本地的 Pod 被修改，则 Kubelet 会做出相应的修改，比如删除 Pod 中某个容器时，则通过 Docker Client 删除该容器。 如果发现删除本节点的 Pod，则删除相应的 Pod，并通过 Docker Client 删除 Pod 中的容器。

Kubelet 读取监听到的信息，如果是创建和修改 Pod 任务，则执行如下处理：

- 为该 Pod 创建一个数据目录；
- 从 API Server 读取该 Pod 清单；
- 为该 Pod 挂载外部卷；
- 下载 Pod 用到的 Secret；

- 检查已经在节点上运行的 Pod，如果该 Pod 没有容器或 Pause 容器没有启动，则先停止 Pod 里所有容器的进程。如果在 Pod 中有需要删除的容器，则删除这些容器；
- 用 “kubernetes/pause” 镜像为每个 Pod 创建一个容器。Pause 容器用于接管 Pod 中所有其他容器的网络。每创建一个新的 Pod，Kubelet 都会先创建一个 Pause 容器，然后创建其他容器。
- 为 Pod 中的每个容器做如下处理：
    1. 为容器计算一个 hash 值，然后用容器的名字去 Docker 查询对应容器的 hash 值。若查找到容器，且两者 hash 值不同，则停止 Docker 中容器的进程，并停止与之关联的 Pause 容器的进程；若两者相同，则不做任何处理；
    2. 如果容器被终止了，且容器没有指定的 restartPolicy，则不做任何处理；
    3. 调用 Docker Client 下载容器镜像，调用 Docker Client 运行容器。

Static Pod

所有以非 API Server 方式创建的 Pod 都叫 Static Pod。Kubelet 将 Static Pod 的状态汇报给 API Server，API Server 为该 Static Pod 创建一个 Mirror Pod 和其相匹配。Mirror Pod 的状态将真实反映 Static Pod 的状态。当 Static Pod 被删除时，与之相对应的 Mirror Pod 也会被删除。

容器健康检查

Pod 通过两类探针检查容器的健康状态:

- (1) LivenessProbe 探针：用于判断容器是否健康，告诉 Kubelet 一个容器什么时候处于不健康的状态。如果 LivenessProbe 探针探测到容器不健康，则 Kubelet 将删除该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含 LivenessProbe 探针，那么 Kubelet 认为该容器的 LivenessProbe 探针返回的值永远是 “Success”；
- (2)ReadinessProbe：用于判断容器是否启动完成且准备接收请求。如果 ReadinessProbe 探针探测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 IP 地址的 Endpoint 条目。

Kubelet 定期调用容器中的 LivenessProbe 探针来诊断容器的健康状况。LivenessProbe 包含如下三种实现方式：

- ExecAction：在容器内部执行一个命令，如果该命令的退出状态码为 0，则表明容器健康；

- TCPSocketAction：通过容器的 IP 地址和端口号执行 TCP 检查，如果端口能被访问，则表明容器健康；
- HTTPGetAction：通过容器的 IP 地址和端口号及路径调用 HTTP GET 方法，如果响应的状态码大于等于 200 且小于 400，则认为容器状态健康。

LivenessProbe 探针包含在 Pod 定义的 spec.containers.{某个容器} 中。

cAdvisor 资源监控

Kubernetes 集群中，应用程序的执行情况可以在不同的级别上监测到，这些级别包括：容器、Pod、Service 和整个集群。Heapster 项目为 Kubernetes 提供了一个基本的监控平台，它是集群级别的监控和事件数据集成器(Aggregator)。Heapster 以 Pod 的方式运行在集群中，Heapster 通过 Kubelet 发现所有运行在集群中的节点，并查看来自这些节点的资源使用情况。Kubelet 通过 cAdvisor 获取其所在节点及容器的数据。Heapster 通过带着关联标签的 Pod 分组这些信息，这些数据将被推到一个可配置的后端，用于存储和可视化展示。支持的后端包括 InfluxDB(使用 Grafana 实现可视化) 和 Google Cloud Monitoring。

cAdvisor 是一个开源的分析容器资源使用率和性能特性的代理工具，已集成到 Kubernetes 代码中。cAdvisor 自动查找所有在其所在节点上的容器，自动采集 CPU、内存、文件系统和网络使用的统计信息。cAdvisor 通过它所在节点机的 Root 容器，采集并分析该节点机的全面使用情况。

cAdvisor 通过其所在节点机的 4194 端口暴露一个简单的 UI。

Kubelet Eviction（驱逐）

Kubelet 会监控资源的使用情况，并使用驱逐机制防止计算和存储资源耗尽。在驱逐时，Kubelet 将 Pod 的所有容器停止，并将 PodPhase 设置为 Failed。

Kubelet 定期（`housekeeping-interval`）检查系统的资源是否达到了预先配置的驱逐阈值，包括

Eviction Signal	Condition	Description
`memory.available`	MemoryPressue	`memory.available := no` `node.stats.memory.worki`
`nodefs.available`	DiskPressure	`nodefs.available := no` 及日志等）
`nodefs.inodesFree`	DiskPressure	`nodefs.inodesFree := n`
`imagefs.available`	DiskPressure	`imagefs.available := n` 像以及容器可写层等）
`imagefs.inodesFree`	DiskPressure	`imagefs.inodesFree :=`

这些驱逐阈值可以使用百分比，也可以使用绝对值，如

```
--eviction-hard=memory.available<500Mi,nodefs.available<1Gi,image
--eviction-minimum-reclaim="memory.available=0Mi,nodefs.available
--system-reserved=memory=1.5Gi
```

这些驱逐信号可以分为软驱逐和硬驱逐

- 软驱逐（Soft Eviction）：配合驱逐宽限期（eviction-soft-grace-period和eviction-max-pod-grace-period）一起使用。系统资源达到软驱逐阈值并在超过宽限期之后才会执行驱逐动作。
- 硬驱逐（Hard Eviction ）：系统资源达到硬驱逐阈值时立即执行驱逐动作。

驱逐动作包括回收节点资源和驱逐用户 Pod 两种：

- 回收节点资源
  - 配置了 imagefs 阈值时
    - 达到 nodefs 阈值：删除已停止的 Pod
    - 达到 imagefs 阈值：删除未使用的镜像
  - 未配置 imagefs 阈值时
    - 达到 nodefs阈值时，按照删除已停止的 Pod 和删除未使用镜像的顺序清理资源
- 驱逐用户 Pod
  - 驱逐顺序为：BestEffort、Burstable、Guaranteed
  - 配置了 imagefs 阈值时
    - 达到 nodefs 阈值，基于 nodefs 用量驱逐（local volume + logs）
    - 达到 imagefs 阈值，基于 imagefs 用量驱逐（容器可写层）

    - 未配置 imagefs 阈值时
        - 达到 nodefs阈值时，按照总磁盘使用驱逐（local volume + logs + 容器可写层）

容器运行时

容器运行时（Container Runtime）是 Kubernetes 最重要的组件之一，负责真正管理镜像和容器的生命周期。Kubelet 通过 Container Runtime Interface (CRI) 与容器运行时交互，以管理镜像和容器。

Container Runtime Interface（CRI）是 Kubernetes v1.5 引入的容器运行时接口，它将 Kubelet 与容器运行时解耦，将原来完全面向 Pod 级别的内部接口拆分成面向 Sandbox 和 Container 的 gRPC 接口，并将镜像管理和容器管理分离到不同的服务。

CRI 最早从从 1.4 版就开始设计讨论和开发，在 v1.5 中发布第一个测试版。在 v1.6 时已经有了很多外部容器运行时，如 frakti 和 cri-o 等。v1.7 中又新增了 cri-containerd 支持用 Containerd 来管理容器。

CRI 基于 gRPC 定义了 RuntimeService 和 ImageService 等两个 gRPC 服务，分别用于容器运行时和镜像的管理。其定义在

- v1.10-v1.11: pkg/kubelet/apis/cri/runtime/v1alpha2
- v1.7-v1.9: pkg/kubelet/apis/cri/v1alpha1/runtime
- v1.6: pkg/kubelet/api/v1alpha1/runtime

Kubelet 作为 CRI 的客户端，而容器运行时则需要实现 CRI 的服务端（即 gRPC server，通常称为 CRI shim）。容器运行时在启动 gRPC server 时需要监听在本地的 Unix Socket （Windows 使用 tcp 格式）。

目前基于 CRI 容器引擎已经比较丰富了，包括

- Docker: 核心代码依然保留在 kubelet 内部（pkg/kubelet/dockershim），是最稳定和特性支持最好的运行时
- OCI 容器运行时：
    - 社区有两个实现
        - Containerd，支持 kubernetes v1.7+
        - CRI-O，支持 Kubernetes v1.6+
    - 支持的 OCI 容器引擎包括
        - runc：OCI 标准容器引擎
        - gVisor：谷歌开源的基于用户空间内核的沙箱容器引擎
        - Clear Containers：Intel 开源的基于虚拟化的容器引擎

    - Kata Containers：基于虚拟化的容器引擎，由 Clear Containers 和 runV 合并而来
- PouchContainer：阿里巴巴开源的胖容器引擎
- Frakti：支持 Kubernetes v1.6+，提供基于 hypervisor 和 docker 的混合运行时，适用于运行非可信应用，如多租户和 NFV 等场景
- Rktlet：支持 rkt 容器引擎
- Virtlet：Mirantis 开源的虚拟机容器引擎，直接管理 libvirt 虚拟机，镜像须是 qcow2 格式
- Infranetes：直接管理 IaaS 平台虚拟机，如 GCE、AWS 等

启动 kubelet 示例

```
/usr/bin/kubelet \
  --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf \
  --kubeconfig=/etc/kubernetes/kubelet.conf \
  --pod-manifest-path=/etc/kubernetes/manifests \
  --allow-privileged=true \
  --network-plugin=cni \
  --cni-conf-dir=/etc/cni/net.d \
  --cni-bin-dir=/opt/cni/bin \
  --cluster-dns=10.96.0.10 \
  --cluster-domain=cluster.local \
  --authorization-mode=Webhook \
  --client-ca-file=/etc/kubernetes/pki/ca.crt \
  --cadvisor-port=0 \
  --rotate-certificates=true \
  --cert-dir=/var/lib/kubelet/pki
```

kubelet 工作原理

如下 kubelet 内部组件结构图所示，Kubelet 由许多内部组件构成

- Kubelet API，包括 10250 端口的认证 API、4194 端口的 cAdvisor API、10255 端口的只读 API 以及 10248 端口的健康检查 API
- syncLoop：从 API 或者 manifest 目录接收 Pod 更新，发送到 podWorkers 处理，大量使用 channel 处理来处理异步请求
- 辅助的 manager，如 cAdvisor、PLEG、Volume Manager 等，处理 syncLoop 以外的其他工作

- CRI：容器执行引擎接口，负责与 container runtime shim 通信
- 容器执行引擎，如 dockershim、rkt 等（注：rkt 暂未完成 CRI 的迁移）
- 网络插件，目前支持 CNI 和 kubenet

Pod 启动流程

Pod Start

查询 Node 汇总指标

通过 Kubelet 的 10255 端口可以查询 Node 的汇总指标。有两种访问方式

- 在集群内部可以直接访问 kubelet 的 10255 端口，比如 `http://:10255/stats/summary`
- 在集群外部可以借助 `kubectl proxy` 来访问，比如

```
kubectl proxy&
curl http://localhost:8001/api/v1/proxy/nodes/:10255/stats/summar
```

kube-proxy

每台机器上都运行一个 kube-proxy 服务，它监听 API server 中 service 和 endpoint 的变化情况，并通过 iptables 等来为服务配置负载均衡（仅支持 TCP 和 UDP）。

kube-proxy 可以直接运行在物理机上，也可以以 static pod 或者 daemonset 的方式运行。

kube-proxy 当前支持一下几种实现

- userspace：最早的负载均衡方案，它在用户空间监听一个端口，所有服务通过 iptables 转发到这个端口，然后在其内部负载均衡到实际的 Pod。该方式最主要的问题是效率低，有明显的性能瓶颈。
- iptables：目前推荐的方案，完全以 iptables 规则的方式来实现 service 负载均衡。该方式最主要的问题是在服务多的时候产生太多的 iptables 规则，非增量式更新会引入一定的时延，大规模情况下有明显的性能问题
- ipvs：为解决 iptables 模式的性能问题，v1.11 新增了 ipvs 模式（v1.8 开始支持测试版，并在 v1.11 GA），采用增量式更新，并可以保证 service 更新期间连接保持不断开
- winuserspace：同 userspace，但仅工作在 windows 节点上

注意：使用 ipvs 模式时，需要预先在每台 Node 上加载内核模块 `nf_conntrack_ipv4`，`ip_vs`，`ip_vs_rr`，`ip_vs_wrr`，`ip_vs_sh` 等。

```
# load module
modprobe -- ip_vs
modprobe -- ip_vs_rr
modprobe -- ip_vs_wrr
modprobe -- ip_vs_sh
modprobe -- nf_conntrack_ipv4

# to check loaded modules, use
lsmod | grep -e ipvs -e nf_conntrack_ipv4
# or
cut -f1 -d " "  /proc/modules | grep -e ip_vs -e nf_conntrack_ipv
```

Iptables 示例

(图片来自cilium/k8s-iptables-diagram)

```
# Masquerade
-A KUBE-MARK-DROP -j MARK --set-xmark 0x8000/0x8000
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traf

# clusterIP and publicIP
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.98.154.163/32 -p tcp -m
-A KUBE-SERVICES -d 10.98.154.163/32 -p tcp -m comment --comment
-A KUBE-SERVICES -d 12.12.12.12/32 -p tcp -m comment --comment "d

# Masq for publicIP
-A KUBE-FW-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:
-A KUBE-FW-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:
-A KUBE-FW-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:

# Masq for nodePort
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/nginx:" -m
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/nginx:" -m

# load balance for each endpoints
-A KUBE-SVC-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:
-A KUBE-SVC-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:
-A KUBE-SVC-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:

# endpoint #1
-A KUBE-SEP-6QCC2MHJZP35QQAR -s 10.244.3.4/32 -m comment --commer
-A KUBE-SEP-6QCC2MHJZP35QQAR -p tcp -m comment --comment "default

# endpoint #2
-A KUBE-SEP-TOYRWPNILILHH3OR -s 10.244.2.4/32 -m comment --commer
-A KUBE-SEP-TOYRWPNILILHH3OR -p tcp -m comment --comment "default

# endpoint #3
-A KUBE-SEP-UXHBWR5XIMVGXW3H -s 10.244.1.2/32 -m comment --commer
-A KUBE-SEP-UXHBWR5XIMVGXW3H -p tcp -m comment --comment "default
```

如果服务设置了 `externalTrafficPolicy: Local` 并且当前 Node 上面没有任何属于该服务的 Pod，那么在 `KUBE-XLB-4N57TFCL4MD7ZTDA` 中会直接丢掉从公网 IP 请求的包：

```
-A KUBE-XLB-4N57TFCL4MD7ZTDA -m comment --comment "default/nginx:
```

ipvs 示例

Kube-proxy IPVS mode 列出了各种服务在 IPVS 模式下的工作原理。

```
$ ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InAct
TCP  10.0.0.1:443 rr persistent 10800
  -> 192.168.0.1:6443             Masq    1      1          0
TCP  10.0.0.10:53 rr
  -> 172.17.0.2:53                Masq    1      0          0
UDP  10.0.0.10:53 rr
  -> 172.17.0.2:53                Masq    1      0          0
```

注意，IPVS 模式也会使用 iptables 来执行 SNAT 和 IP 伪装（MASQUERADE），并使用 ipset 来简化 iptables 规则的管理：

ipset 名	成员	用途
KUBE-CLUSTER-IP	All service IP + port	Mark-Masq for ca `e-all=true` or specified
KUBE-LOOP-BACK	All service IP + port + IP	masquerade for s purpose
KUBE-EXTERNAL-IP	service external IP + port	masquerade for p IPs
KUBE-LOAD-BALANCER	load balancer ingress IP + port	masquerade for p balancer type ser
KUBE-LOAD-BALANCER-LOCAL	LB ingress IP + port with `externalT rafficPolicy=local`	accept packages with `externalTraffi`
KUBE-LOAD-BALANCER-FW	load balancer ingress IP + port with `loadBalancerSourceRanges`	package filter for `loadBalancerSo` specified
KUBE-LOAD-BALANCER-SOURCE-CIDR	load balancer ingress IP + port + source CIDR	package filter for `loadBalancerSo` specified
KUBE-NODE-PORT-TCP	nodeport type service TCP port	masquerade for p nodePort(TCP)
KUBE-NODE-PORT-LOCAL-TCP	nodeport type service TCP port with `externalTrafficPolicy=local`	accept packages with `externalTraffi`

ipset 名	成员	用途
KUBE-NODE-PORT-UDP	nodeport type service UDP port	masquerade for p nodePort(UDP)
KUBE-NODE-PORT-LOCAL-UDP	nodeport type service UDP port with `externalTrafficPolicy=local`	accept packages with `externalTra`

启动 kube-proxy 示例

```
kube-proxy --kubeconfig=/var/lib/kubelet/kubeconfig --cluster-cid
```

kube-proxy 工作原理

kube-proxy 监听 API server 中 service 和 endpoint 的变化情况，并通过 userspace、iptables、ipvs 或 winuserspace 等 proxier 来为服务配置负载均衡（仅支持 TCP 和 UDP）。

kube-proxy 不足

kube-proxy 目前仅支持 TCP 和 UDP，不支持 HTTP 路由，并且也没有健康检查机制。这些可以通过自定义 Ingress Controller 的方法来解决。

kube-dns

DNS 是 Kubernetes 的核心功能之一，通过 kube-dns 或 CoreDNS 作为集群的必备扩展来提供命名服务。

CoreDNS

从 v1.11 开始可以使用 CoreDNS 来提供命名服务，并从 v1.13 开始成为默认 DNS 服务。CoreDNS 的特点是效率更高，资源占用率更小，推荐使用 CoreDNS 替代 kube-dns 为集群提供 DNS 服务。

从 kube-dns 升级为 CoreDNS 的步骤为：

```
$ git clone https://github.com/coredns/deployment
$ cd deployment/kubernetes
$ ./deploy.sh | kubectl apply -f -
$ kubectl delete --namespace=kube-system deployment kube-dns
```

全新部署的话，可以点击这里 查看 CoreDNS 扩展的配置方法。

支持的 DNS 格式

- Service
  - A record：生成 `my-svc.my-namespace.svc.cluster.local` ，解析 IP 分为两种情况
    - 普通 Service 解析为 Cluster IP
    - Headless Service 解析为指定的 Pod IP 列表
  - SRV record：生成 `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster.local`
- Pod
  - A record： `pod-ip-address.my-namespace.pod.cluster.local`
  - 指定 hostname 和 subdomain： `hostname.custom-subdomain.default.svc.cluster.local` ，如下所示

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

支持配置私有 DNS 服务器和上游 DNS 服务器

从 Kubernetes 1.6 开始，可以通过为 kube-dns 提供 ConfigMap 来实现对存根域以及上游名称服务器的自定义指定。例如，下面的配置插入了一个单独的私有根 DNS 服务器和两个上游 DNS 服务器。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {“acme.local”: [“1.2.3.4”]}
  upstreamNameservers: |
    [“8.8.8.8”, “8.8.4.4”]
```

使用上述特定配置，查询请求首先会被发送到 kube-dns 的 DNS 缓存层 (Dnsmasq 服务器)。Dnsmasq 服务器会先检查请求的后缀，带有集群后缀（例如：”.cluster.local”）的请求会被发往 kube-dns，拥有存根域后缀的名称

（例如：".acme.local"）将会被发送到配置的私有 DNS 服务器 ["1.2.3.4"]。最后，不满足任何这些后缀的请求将会被发送到上游 DNS ["8.8.8.8", "8.8.4.4"] 里。

kube-dns

启动 kube-dns 示例

一般通过扩展的方式部署 DNS 服务，如把 kube-dns.yaml 放到 Master 节点的 `/etc/kubernetes/addons` 目录中。当然也可以手动部署：

```
kubectl apply -f https://kubernetes.feisky.xyz/manifests/kubedns/
```

这会在 Kubernetes 中启动一个包含三个容器的 Pod，运行着 DNS 相关的三个服务：

```
# kube-dns container
kube-dns --domain=cluster.local. --dns-port=10053 --config-dir=/k

# dnsmasq container
dnsmasq-nanny -v=2 -logtostderr -configDir=/etc/k8s/dns/dnsmasq-r

# sidecar container
sidecar --v=2 --logtostderr --probe=kubedns,127.0.0.1:10053,kuber
```

Kubernetes v1.10 也支持 Beta 版的 CoreDNS，其性能较 kube-dns 更好。可以以扩展方式部署，如把 coredns.yaml 放到 Master 节点的 `/etc/kubernetes/addons` 目录中。当然也可以手动部署：

```
kubectl apply -f https://kubernetes.feisky.xyz/manifests/kubedns/
```

kube-dns 工作原理

如下图所示，kube-dns 由三个容器构成：

- kube-dns：DNS 服务的核心组件，主要由 KubeDNS 和 SkyDNS 组成
  - KubeDNS 负责监听 Service 和 Endpoint 的变化情况，并将相关的信息更新到 SkyDNS 中

    - SkyDNS 负责 DNS 解析，监听在 10053 端口 (tcp/udp)，同时也监听在 10055 端口提供 metrics
    - kube-dns 还监听了 8081 端口，以供健康检查使用
- dnsmasq-nanny：负责启动 dnsmasq，并在配置发生变化时重启 dnsmasq
    - dnsmasq 的 upstream 为 SkyDNS，即集群内部的 DNS 解析由 SkyDNS 负责
- sidecar：负责健康检查和提供 DNS metrics（监听在 10054 端口）

源码简介

kube-dns 的代码已经从 kubernetes 里面分离出来，放到了 https://github.com/kubernetes/dns。

kube-dns、dnsmasq-nanny 和 sidecar 的代码均是从 `cmd//main.go` 开始，并分别调用 `pkg/dns`、`pkg/dnsmasq` 和 `pkg/sidecar` 完成相应的功能。而最核心的 DNS 解析则是直接引用了 `github.com/skynetservices/skydns/server` 的代码，具体实现见 skynetservices/skydns。

常见问题

Ubuntu 18.04 中 DNS 无法解析的问题

Ubuntu 18.04 中默认开启了 systemd-resolved，它会在系统的 /etc/resolv.conf 中写入 `nameserver 127.0.0.53` 。由于这是一个本地地址，从而会导致 CoreDNS 或者 kube-dns 无法解析外网地址。

解决方法是替换掉 systemd-resolved 生成的 resolv.conf 文件：

```
sudo rm /etc/resolv.conf
sudo ln -s /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

或者为 DNS 服务手动指定 resolv.conf 的路径：

```
--resolv-conf=/run/systemd/resolve/resolv.conf
```

参考文档

- dns-pod-service 介绍

- coredns/coredns

Federation

在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。K8s 的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足 K8s 的调度和计算存储连接要求。而集群联邦（Federation）就是为提供跨 Region 跨服务商 K8s 集群服务而设计的。

每个 Federation 有自己的分布式存储、API Server 和 Controller Manager。用户可以通过 Federation 的 API Server 注册该 Federation 的成员 K8s Cluster。当用户通过 Federation 的 API Server 创建、更改 API 对象时，Federation API Server 会在自己所有注册的子 K8s Cluster 都创建一份对应的 API 对象。在提供业务请求服务时，K8s Federation 会先在自己的各个子 Cluster 之间做负载均衡，而对于发送到某个具体 K8s Cluster 的业务请求，会依照这个 K8s Cluster 独立提供服务时一样的调度模式去做 K8s Cluster 内部的负载均衡。而 Cluster 之间的负载均衡是通过域名服务的负载均衡来实现的。

所有的设计都尽量不影响 K8s Cluster 现有的工作机制，这样对于每个子 K8s 集群来说，并不需要更外层的有一个 K8s Federation，也就是意味着所有现有的 K8s 代码和机制不需要因为 Federation 功能有任何变化。

Federation 主要包括三个组件

- federation-apiserver：类似 kube-apiserver，但提供的是跨集群的 REST API
- federation-controller-manager：类似 kube-controller-manager，但提供多集群状态的同步机制
- kubefed：Federation 管理命令行工具

Federation 的代码维护在 https://github.com/kubernetes/federation。

Federation 部署方法

下载 kubefed 和 kubectl

kubefed 下载

```
# Linux
curl -LO https://storage.googleapis.com/kubernetes-release/releas
tar -xzvf kubernetes-client-linux-amd64.tar.gz

# OS X
curl -LO https://storage.googleapis.com/kubernetes-release/releas
tar -xzvf kubernetes-client-darwin-amd64.tar.gz

# Windows
curl -LO https://storage.googleapis.com/kubernetes-release/releas
tar -xzvf kubernetes-client-windows-amd64.tar.gz
```

kubectl 的下载可以参考 这里。

初始化主集群

选择一个已部署好的 Kubernetes 集群作为主集群，作为集群联邦的控制平面，并配置好本地的 kubeconfig。然后运行 `kubefed init` 命令来初始化主集群：

```
$ kubefed init fellowship \
    --host-cluster-context=rivendell \   # 部署集群的 kubeconfig 配置
    --dns-provider="google-clouddns" \   # DNS 服务提供商，还支持 aws
    --dns-zone-name="example.com." \     # 域名后缀，必须以. 结束
    --apiserver-enable-basic-auth=true \ # 开启 basic 认证
    --apiserver-enable-token-auth=true \ # 开启 token 认证
    --apiserver-arg-overrides="--anonymous-auth=false,--v=4" # fe
$ kubectl config use-context fellowship
```

自定义 DNS

coredns 需要先部署一套 etcd 集群，可以用 helm 来部署：

```
$ helm install --namespace my-namespace --name etcd-operator stab
$ helm upgrade --namespace my-namespace --set cluster.enabled=tru
```

然后部署 coredns

```
$ cat Values.yaml
isClusterService: false
serviceType: "LoadBalancer"
middleware:
  kubernetes:
    enabled: false
  etcd:
    enabled: true
    zones:
    - "example.com."
    endpoint: "http://etcd-cluster.my-namespace:2379"

$ helm install --namespace my-namespace --name coredns -f Values.
```

使用 coredns 时，还需要传入 coredns 的配置

```
$ cat $HOME/coredns-provider.conf
[Global]
etcd-endpoints = http://etcd-cluster.my-namespace:2379
zones = example.com.

$ kubefed init fellowship \
    --host-cluster-context=rivendell \   # 部署集群的 kubeconfig 配置
    --dns-provider="coredns" \           # DNS 服务提供商，还支持 aws
    --dns-zone-name="example.com." \     # 域名后缀，必须以. 结束
    --apiserver-enable-basic-auth=true \ # 开启 basic 认证
    --apiserver-enable-token-auth=true \ # 开启 token 认证
    --dns-provider-config="$HOME/coredns-provider.conf" \ # coredns
    --apiserver-arg-overrides="--anonymous-auth=false,--v=4" # fe
```

物理机部署

默认情况下，`kubefed init` 会创建一个 LoadBalancer 类型的 federation API server 服务，这需要 Cloud Provider 的支持。在物理机部署时，可以通过 `--api-server-service-type` 选项将其改成 NodePort：

```
$ kubefed init fellowship \
    --host-cluster-context=rivendell \   # 部署集群的 kubeconfig 配置
    --dns-provider="coredns" \           # DNS 服务提供商，还支持 aws
    --dns-zone-name="example.com." \     # 域名后缀，必须以. 结束
    --apiserver-enable-basic-auth=true \ # 开启 basic 认证
    --apiserver-enable-token-auth=true \ # 开启 token 认证
    --dns-provider-config="$HOME/coredns-provider.conf" \ # coredns
    --apiserver-arg-overrides="--anonymous-auth=false,--v=4" \ #
    --api-server-service-type="NodePort" \
    --api-server-advertise-address="10.0.10.20"
```

自定义 etcd 存储

默认情况下，`kubefed init` 通过动态创建 PV 的方式为 etcd 创建持久化存储。如果 kubernetes 集群不支持动态创建 PV，则可以预先创建 PV，注意 PV 要匹配 `kubefed` 的 PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.alpha.kubernetes.io/storage-class: "yes"
  labels:
    app: federated-cluster
  name: fellowship-federation-apiserver-etcd-claim
  namespace: federation-system
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

注册集群

除主集群外，其他 kubernetes 集群可以通过 `kubefed join` 命令加入集群联邦：

```
$ kubefed join gondor --host-cluster-context=rivendell --cluster-
```

集群查询

查询注册到 Federation 的 kubernetes 集群列表

```
$ kubectl --context=federation get clusters
```

ClusterSelector

v1.7 + 支持使用 annotation `federation.alpha.kubernetes.io/cluster-selector` 为新对象选择 kubernetes 集群。该 annotation 的值是一个 json 数组，比如

```
  metadata:
    annotations:
      federation.alpha.kubernetes.io/cluster-selector: '[{"key":"
        "In", "values": ["true"]}, {"key": "environment", "operat
        ["test"]}]'
```

每条记录包含三个键值

- key：集群的 label 名字
- operator：包括 In, NotIn, Exists, DoesNotExist, Gt, Lt
- values：集群的 label 值

策略调度

> 注：仅 v1.7 + 支持策略调度。

开启策略调度的方法

（1）创建 ConfigMap

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/ku
```

（2）编辑 federation-apiserver

```
kubectl -n federation-system edit deployment federation-apiserver
```

增加选项：

```
--admission-control=SchedulingPolicy
--admission-control-config-file=/etc/kubernetes/admission/config.
```

增加 volume：

```
- name: admission-config
  configMap:
    name: admission
```

增加 volumeMounts:

```
volumeMounts:
- name: admission-config
  mountPath: /etc/kubernetes/admission
```

（3）部署外部策略引擎，如 Open Policy Agent (OPA)

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/ku
kubectl create -f https://raw.githubusercontent.com/kubernetes/ku
```

（4）创建 namespace `kube-federation-scheduling-policy` 以供外部策略引擎使用

```
kubectl --context=federation create namespace kube-federation-sch
```

（5）创建策略

```
wget https://raw.githubusercontent.com/kubernetes/kubernetes.gith
kubectl --context=federation -n kube-federation-scheduling-policy
```

（6）验证策略

```
kubectl --context=federation annotate clusters cluster-name-1 pci
kubectl --context=federation create -f https://raw.githubusercont
kubectl --context=federation get rs nginx-pci -o jsonpath='{.meta
```

集群联邦使用

集群联邦支持以下联邦资源，这些资源会自动在所有注册的 kubernetes 集群中创建：

- Federated ConfigMap
- Federated Service
- Federated DaemonSet
- Federated Deployment
- Federated Ingress
- Federated Namespaces
- Federated ReplicaSets
- Federated Secrets
- Federated Events（仅存在 federation 控制平面）
- Federated Jobs（v1.8+）
- Federated Horizontal Pod Autoscaling (HPA，v1.8+)

比如使用 Federated Service 的方法如下：

```
# 这会在所有注册到联邦的 kubernetes 集群中创建服务
$ kubectl --context=federation-cluster create -f services/nginx.y

# 添加后端 Pod
$ for CLUSTER in asia-east1-c asia-east1-a asia-east1-b \
                         europe-west1-d europe-west1-c europe-west
                         us-central1-f us-central1-a us-central1-b
                         us-east1-d us-east1-c us-east1-b
do
  kubectl --context=$CLUSTER run nginx --image=nginx:1.11.1-alpir
done

# 查看服务状态
$ kubectl --context=federation-cluster describe services nginx
```

可以通过 DNS 来访问联邦服务，访问格式包括以下几种

- `nginx.mynamespace.myfederation.`
- `nginx.mynamespace.myfederation.svc.example.com.`
- `nginx.mynamespace.myfederation.svc.us-central1.example.com.`

删除集群

```
$ kubefed unjoin gondor --host-cluster-context=rivendell
```

删除集群联邦

集群联邦控制平面的删除功能还在开发中，目前可以通过删除 namespace `federation-system` 的方法来清理（注意 pv 不会删除）：

```
$ kubectl delete ns federation-system
```

参考文档

- Kubernetes federation
- kubefed

kubeadm

kubeadm 是 Kubernetes 主推的部署工具之一，正在快速迭代开发中。

初始化系统

所有机器都需要初始化容器执行引擎（如 docker 或 frakti 等）和 kubelet。这是因为 kubeadm 依赖 kubelet 来启动 Master 组件，比如 kube-apiserver、kube-manager-controller、kube-scheduler、kube-proxy 等。

安装 master

在初始化 master 时，只需要执行 kubeadm init 命令即可，比如

```
kubeadm init --pod-network-cidr 10.244.0.0/16 --kubernetes-versio
```

这个命令会自动

- 系统状态检查
- 生成 token
- 生成自签名 CA 和 client 端证书
- 生成 kubeconfig 用于 kubelet 连接 API server
- 为 Master 组件生成 Static Pod manifests，并放到 `/etc/kubernetes/manifests` 目录中
- 配置 RBAC 并设置 Master node 只运行控制平面组件
- 创建附加服务，比如 kube-proxy 和 kube-dns

配置 Network plugin

kubeadm 在初始化时并不关心网络插件，默认情况下，kubelet 配置使用 CNI 插件，这样就需要用户来额外初始化网络插件。

CNI bridge

```
mkdir -p /etc/cni/net.d
cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.1.0/24",
        "routes": [
            {"dst": "0.0.0.0/0"}
        ]
    }
}
EOF
cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "type": "loopback"
}
EOF
```

flannel

```
kubectl create -f https://github.com/coreos/flannel/raw/master/Dc
kubectl create -f https://github.com/coreos/flannel/raw/master/Dc
```

weave

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$
```

calico

```
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
```

添加 Node

```
token=$(kubeadm token list | grep authentication,signing | awk '{
kubeadm join --token $token ${master_ip}
```

这包括以下几个步骤

- 从 API server 下载 CA
- 创建本地证书，并请求 API Server 签名
- 最后配置 kubelet 连接到 API Server

删除安装

```
kubeadm reset
```

参考文档

- kubeadm Setup Tool

hyperkube

hyperkube是Kubernetes的allinone binary，可以用来启动多种kubernetes服务，常用在Docker镜像中。每个Kubernetes发布都会同时发布一个包含hyperkube的docker镜像，如 `gcr.io/google_containers/hyperkube:v1.6.4` 。

hyperkube支持的子命令包括

- kubelet
- apiserver

- controller-manager
- federation-apiserver
- federation-controller-manager
- kubectl
- proxy
- scheduler

kubectl

kubectl 是 Kubernetes 的命令行工具（CLI），是 Kubernetes 用户和管理员必备的管理工具。

kubectl 提供了大量的子命令，方便管理 Kubernetes 集群中的各种功能。这里不再罗列各种子命令的格式，而是介绍下如何查询命令的帮助

- `kubectl -h` 查看子命令列表
- `kubectl options` 查看全局选项
- `kubectl --help` 查看子命令的帮助
- `kubectl [command] [PARAMS] -o=` 设置输出格式（如 json、yaml、jsonpath 等）
- `kubectl explain [RESOURCE]` 查看资源的定义

配置

使用 kubectl 的第一步是配置 Kubernetes 集群以及认证方式，包括

- cluster 信息：Kubernetes server 地址
- 用户信息：用户名、密码或密钥
- Context：cluster、用户信息以及 Namespace 的组合

示例

```
kubectl config set-credentials myself --username=admin --passwor
kubectl config set-cluster local-server --server=http://localhost
kubectl config set-context default-context --cluster=local-server
kubectl config use-context default-context
kubectl config view
```

常用命令格式

- 创建：`kubectl run --image=` 或者 `kubectl create -f manifest.yaml`
- 查询：`kubectl get`
- 更新 `kubectl set` 或者 `kubectl patch`
- 删除：`kubectl delete` 或者 `kubectl delete -f manifest.yaml`
- 查询 Pod IP：`kubectl get pod -o jsonpath='{.status.podIP}'`
- 容器内执行命令：`kubectl exec -ti sh`
- 容器日志：`kubectl logs [-f]`
- 导出服务：`kubectl expose deploy --port=80`
- Base64 解码：

```
kubectl get secret SECRET -o go-template='{{ .data.KEY | base64de
```

注意，`kubectl run` 仅支持 Pod、Replication Controller、Deployment、Job 和 CronJob 等几种资源。具体的资源类型是由参数决定的，默认为 Deployment：

创建的资源类型	参数
Pod	`--restart=Never`
Replication Controller	`--generator=run/v1`
Deployment	`--restart=Always`
Job	`--restart=OnFailure`
CronJob	`--schedule=`

命令行自动补全

Linux 系统 Bash：

```
source /usr/share/bash-completion/bash_completion
source <(kubectl completion bash)
```

MacOS zsh

```
source <(kubectl completion zsh)
```

日志查看

`kubectl logs` 用于显示 pod 运行中，容器内程序输出到标准输出的内容。跟 docker 的 logs 命令类似。

```
  # Return snapshot logs from pod nginx with only one container
  kubectl logs nginx

  # Return snapshot of previous terminated ruby container logs fr
  kubectl logs -p -c ruby web-1

  # Begin streaming the logs of the ruby container in pod web-1
  kubectl logs -f -c ruby web-1
```

连接到一个正在运行的容器

`kubectl attach` 用于连接到一个正在运行的容器。跟 docker 的 attach 命令类似。

```
  # Get output from running pod 123456-7890, using the first cont
  kubectl attach 123456-7890

  # Get output from ruby-container from pod 123456-7890
  kubectl attach 123456-7890 -c ruby-container

  # Switch to raw terminal mode, sends stdin to 'bash' in ruby-co
  # and sends stdout/stderr from 'bash' back to the client
  kubectl attach 123456-7890 -c ruby-container -i -t

Options:
  -c, --container='': Container name. If omitted, the first conta
  -i, --stdin=false: Pass stdin to the container
  -t, --tty=false: Stdin is a TTY
```

在容器内部执行命令

`kubectl exec` 用于在一个正在运行的容器执行命令。跟 docker 的 exec 命令类似。

```
  # Get output from running 'date' from pod 123456-7890, using th
  kubectl exec 123456-7890 date

  # Get output from running 'date' in ruby-container from pod 123
  kubectl exec 123456-7890 -c ruby-container date

  # Switch to raw terminal mode, sends stdin to 'bash' in ruby-co
  # and sends stdout/stderr from 'bash' back to the client
  kubectl exec 123456-7890 -c ruby-container -i -t -- bash -il

Options:
  -c, --container='': Container name. If omitted, the first conta
  -p, --pod='': Pod name
  -i, --stdin=false: Pass stdin to the container
  -t, --tty=false: Stdin is a TT
```

端口转发

`kubectl port-forward` 用于将本地端口转发到指定的 Pod。

```
# Listen on ports 5000 and 6000 locally, forwarding data to/from
kubectl port-forward mypod 5000 6000

# Listen on port 8888 locally, forwarding to 5000 in the pod
kubectl port-forward mypod 8888:5000

# Listen on a random port locally, forwarding to 5000 in the pod
kubectl port-forward mypod :5000

# Listen on a random port locally, forwarding to 5000 in the pod
kubectl port-forward mypod 0:5000
```

也可以将本地端口转发到服务、复制控制器或者部署的端口。

```
# Forward to deployment
kubectl port-forward deployment/redis-master 6379:6379

# Forward to replicaSet
kubectl port-forward rs/redis-master 6379:6379

# Forward to service
kubectl port-forward svc/redis-master 6379:6379
```

API Server 代理

`kubectl proxy` 命令提供了一个 Kubernetes API 服务的 HTTP 代理。

```
$ kubectl proxy --port=8080
Starting to serve on 127.0.0.1:8080
```

可以通过代理地址 `http://localhost:8080/api/` 来直接访问 Kubernetes API，比如查询 Pod 列表

```
curl http://localhost:8080/api/v1/namespaces/default/pods
```

注意，如果通过 `--address` 指定了非 localhost 的地址，则访问 8080 端口时会报未授权的错误，可以设置 `--accept-hosts` 来避免这个问题（ **不推荐生产环境这么设置** ）：

```
kubectl proxy --address='0.0.0.0' --port=8080 --accept-hosts='^*$
```

文件拷贝

`kubectl cp` 支持从容器中拷贝，或者拷贝文件到容器中

```
  # Copy /tmp/foo_dir local directory to /tmp/bar_dir in a remote
  kubectl cp /tmp/foo_dir :/tmp/bar_dir

  # Copy /tmp/foo local file to /tmp/bar in a remote pod in a spe
  kubectl cp /tmp/foo :/tmp/bar -c

  # Copy /tmp/foo local file to /tmp/bar in a remote pod in names
  kubectl cp /tmp/foo /:/tmp/bar

  # Copy /tmp/foo from a remote pod to /tmp/bar locally
  kubectl cp /:/tmp/foo /tmp/bar

Options:
  -c, --container='': Container name. If omitted, the first conta
```

注意：文件拷贝依赖于 tar 命令，所以容器中需要能够执行 tar 命令

kubectl drain

```
kubectl drain NODE [Options]
```

- 它会删除该 NODE 上由 ReplicationController, ReplicaSet, DaemonSet, StatefulSet or Job 创建的 Pod
- 不删除 mirror pods（因为不可通过 API 删除 mirror pods）
- 如果还有其它类型的 Pod（比如不通过 RC 而直接通过 kubectl create 的 Pod）并且没有 --force 选项，该命令会直接失败
- 如果命令中增加了 --force 选项，则会强制删除这些不是通过 ReplicationController, Job 或者 DaemonSet 创建的 Pod

有的时候不需要 evict pod，只需要标记 Node 不可调用，可以用 `kubectl cordon` 命令。

恢复的话只需要运行 `kubectl uncordon NODE` 将 NODE 重新改成可调度状态。

权限检查

`kubectl auth` 提供了两个子命令用于检查用户的鉴权情况：

- `kubectl auth can-i` 检查用户是否有权限进行某个操作，比如

```
# Check to see if I can create pods in any namespace
kubectl auth can-i create pods --all-namespaces

# Check to see if I can list deployments in my current namespac
kubectl auth can-i list deployments.extensions

# Check to see if I can do everything in my current namespace (
kubectl auth can-i '*' '*'

# Check to see if I can get the job named "bar" in namespace "f
kubectl auth can-i list jobs.batch/bar -n foo
```

- `kubectl auth reconcile` 自动修复有问题的 RBAC 策略，如

```
# Reconcile rbac resources from a file
kubectl auth reconcile -f my-rbac-rules.yaml
```

kubectl 插件

kubectl 插件提供了一种扩展 kubectl 的机制，比如添加新的子命令。插件可以以任何语言编写，只需要满足以下条件即可

- 插件放在 `~/.kube/plugins` 或环境变量 `KUBECTL_PLUGINS_PATH` 指定的目录中
- 插件的格式为 `子目录 / 可执行文件或脚本` 且子目录中要包括 `plugin.yaml` 配置文件

比如

```
$ tree
.
└── hello
    └── plugin.yaml

1 directory, 1 file

$ cat hello/plugin.yaml
name: "hello"
shortDesc: "Hello kubectl plugin!"
command: "echo Hello plugins!"

$ kubectl plugin hello
Hello plugins!
```

你也可以使用 krew 来管理 kubectl 插件。

原始 URI

kubectl 也可以用来直接访问原始 URI，比如要访问 Metrics API 可以

- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/pods`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespace//pods/`

附录

kubectl 的安装方法

```
# OS X
curl -LO https://storage.googleapis.com/kubernetes-release/releas

# Linux
curl -LO https://storage.googleapis.com/kubernetes-release/releas

# Windows
curl -LO https://storage.googleapis.com/kubernetes-release/releas
```

资源对象

Kubernetes 主要概念和对象介绍。

- Autoscaling (HPA)
- ConfigMap
- CronJob
- CustomResourceDefinition
- DaemonSet
- Deployment
- Ingress
- Job
- LocalVolume
- Namespace
- NetworkPolicy
- Node
- PersistentVolume
- Pod
- PodPreset
- ReplicaSet
- Resource Quota
- Secret
- SecurityContext
- Service
- ServiceAccount
- StatefulSet
- ThirdPartyResources
- Volume

Autoscaling

Horizontal Pod Autoscaling (HPA) 可以根据 CPU 使用率或应用自定义 metrics 自动扩展 Pod 数量（支持 replication controller、deployment 和 replica set）。

- 控制管理器每隔 30s（可以通过 `--horizontal-pod-autoscaler-sync-period` 修改）查询 metrics 的资源使用情况
- 支持三种 metrics 类型
  - 预定义 metrics（比如 Pod 的 CPU）以利用率的方式计算
  - 自定义的 Pod metrics，以原始值（raw value）的方式计算
  - 自定义的 object metrics
- 支持两种 metrics 查询方式：Heapster 和自定义的 REST API
- 支持多 metrics

注意：

- 本章是关于 Pod 的自动扩展，而 Node 的自动扩展请参考 Cluster AutoScaler。
- 在使用 HPA 之前需要 **确保已部署好 metrics-server**。

API 版本对照表

Kubernetes 版本	autoscaling API 版本	支持的 metrics
v1.5+	autoscaling/v1	CPU
v1.6+	autoscaling/v2beta1	Memory及自定义

示例

```
# 创建 pod 和 service
$ kubectl run php-apache --image=k8s.gcr.io/hpa-example --request
service "php-apache" created
deployment "php-apache" created

# 创建 autoscaler
$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=
deployment "php-apache" autoscaled

$ kubectl get hpa
NAME         REFERENCE                     TARGET    MINPODS   MA
php-apache   Deployment/php-apache/scale   0% / 50%  1         10

# 增加负载
$ kubectl run -i --tty load-generator --image=busybox /bin/sh
Hit enter for command prompt
$ while true; do wget -q -O- http://php-apache.default.svc.cluste

# 过一会就可以看到负载升高了
$ kubectl get hpa
NAME         REFERENCE                     TARGET      CURRENT
php-apache   Deployment/php-apache/scale   305% / 50%  305%

# autoscaler 将这个 deployment 扩展为 7 个 pod
$ kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   7         7         7            7           19m

# 删除刚才创建的负载增加 pod 后会发现负载降低，并且 pod 数量也自动降回 1 个
$ kubectl get hpa
NAME         REFERENCE                     TARGET       MINPODS
php-apache   Deployment/php-apache/scale   0% / 50%     1

$ kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   1         1         1            1           27m
```

自定义 metrics

使用方法

- 控制管理器开启 `--horizontal-pod-autoscaler-use-rest-clients`
- 控制管理器配置的 `--master` 或者 `--kubeconfig`
- 在 API Server Aggregator 中注册自定义的 metrics API，如 https://github.com/kubernetes-incubator/custom-metrics-apiserver 和 https://github.com/kubernetes/metrics

> 注：可以参考 k8s.io/metics 开发自定义的 metrics API server。

比如 HorizontalPodAutoscaler 保证每个 Pod 占用 50% CPU、1000pps 以及 10000 请求 / s：

HPA 示例

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Pods
    pods:
      metricName: packets-per-second
      targetAverageValue: 1k
  - type: Object
    object:
      metricName: requests-per-second
      target:
        apiVersion: extensions/v1beta1
        kind: Ingress
        name: main-route
      targetValue: 10k
status:
  observedGeneration: 1
  lastScaleTime: >
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 0
```

状态条件

v1.7+ 可以在客户端中看到 Kubernetes 为 HorizontalPodAutoscaler 设置的状态条件 `status.conditions`，用来判断 HorizontalPodAutoscaler 是否可以扩展（AbleToScale）、是否开启扩展（ScalingActive）以及是否受到限制（ScalingLimitted）。

```
$ kubectl describe hpa cm-test
Name:                           cm-test
Namespace:                      prom
Labels:
Annotations:
CreationTimestamp:              Fri, 16 Jun 2017 18:09:22 +0000
Reference:                      ReplicationController/cm-test
Metrics:                        (current / target)
  "http_requests" on pods:      66m / 500m
Min replicas:                   1
Max replicas:                   4
ReplicationController pods:     1 current / 1 desired
Conditions:
  Type                  Status  Reason                  Message
  ----                  ------  ------                  -------
  AbleToScale           True    ReadyForNewScale        the last
  ScalingActive         True    ValidMetricFound        the HPA w
  ScalingLimited        False   DesiredWithinRange      the desir
Events:
```

HPA 最佳实践

- 为容器配置 CPU Requests
- HPA 目标设置恰当，如设置 70% 给容器和应用预留 30% 的余量
- 保持 Pods 和 Nodes 健康（避免 Pod 频繁重建）
- 保证用户请求的负载均衡
- 使用 `kubectl top node` 和 `kubectl top pod` 查看资源使用情况

ConfigMap

在执行应用程式或是生产环境等等, 会有许多的情况需要做变更, 而我们不希望因应每一种需求就要准备一个镜像档, 这时就可以透过 ConfigMap 来帮我们做一个配置档或是命令参数的映射, 更加弹性化使用我们的服务或是应用程式。

ConfigMap 用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。ConfigMap 跟 secret 很类似，但它可以更方便地处理不包含敏感信息的字符串。

API 版本对照表

Kubernetes 版本	Core API 版本
v1.5+	core/v1

ConfigMap 创建

可以使用 `kubectl create configmap` 从文件、目录或者 key-value 字符串创建等创建 ConfigMap。也可以通过 `kubectl create -f file` 创建。

从 key-value 字符串创建

```
$ kubectl create configmap special-config --from-literal=special.
configmap "special-config" created
$ kubectl get configmap special-config -o go-template='{{.data}}'
map[special.how:very]
```

从 env 文件创建

```
$ echo -e "a=b\nc=d" | tee config.env
a=b
c=d
$ kubectl create configmap special-config --from-env-file=config.
configmap "special-config" created
$ kubectl get configmap special-config -o go-template='{{.data}}'
map[a:b c:d]
```

从目录创建

```
$ mkdir config
$ echo a>config/a
$ echo b>config/b
$ kubectl create configmap special-config --from-file=config/
configmap "special-config" created
$ kubectl get configmap special-config -o go-template='{{.data}}'
map[a:a
 b:b
]
```

从文件 Yaml/Json 文件创建

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

```
$ kubectl create  -f  config.yaml
configmap "special-config" created
```

ConfigMap 使用

ConfigMap 可以通过三种方式在 Pod 中使用，三种分别方式为：设置环境变量、设置容器命令行参数以及在 Volume 中直接挂载文件或目录。

> **注意**
>
> - ConfigMap 必须在 Pod 引用它之前创建
> - 使用 `envFrom` 时，将会自动忽略无效的键
> - Pod 只能使用同一个命名空间内的 ConfigMap

首先创建 ConfigMap：

```
$ kubectl create configmap special-config --from-literal=special.
$ kubectl create configmap env-config --from-literal=log_level=IN
```

用作环境变量

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "-c", "env"]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

当 Pod 结束后会输出

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
log_level=INFO
```

用作命令行参数

将 ConfigMap 用作命令行参数时，需要先把 ConfigMap 的数据保存在环境变量中，然后通过 `$(VAR_NAME)` 的方式引用环境变量.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPE
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

当 Pod 结束后会输出

```
very charm
```

使用 volume 将 ConfigMap 作为文件或目录直接挂载

将创建的 ConfigMap 直接挂载至 Pod 的 / etc/config 目录下，其中每一个 key-value 键值对都会生成一个文件，key 为文件名，value 为内容

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh", "-c", "cat /etc/config/special.how"]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

当 Pod 结束后会输出

```
very
```

将创建的 ConfigMap 中 special.how 这个 key 挂载到 / etc/config 目录下的一个相对路径 / keys/special.level。如果存在同名文件，直接覆盖。其他的 key 不挂载

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh","-c","cat /etc/config/keys/special.leve
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: keys/special.level
  restartPolicy: Never
```

当 Pod 结束后会输出

```
very
```

ConfigMap 支持同一个目录下挂载多个 key 和多个目录。例如下面将 special.how 和 special.type 通过挂载到 / etc/config 下。并且还将 special.how 同时挂载到 / etc/config2 下。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["/bin/sh","-c","sleep 36000"]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
      - name: config-volume2
        mountPath: /etc/config2
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: keys/special.level
        - key: special.type
          path: keys/special.type
    - name: config-volume2
      configMap:
        name: special-config
        items:
        - key: special.how
          path: keys/special.level
  restartPolicy: Never
```

```
# ls  /etc/config/keys/
special.level  special.type
# ls  /etc/config2/keys/
special.level
# cat  /etc/config/keys/special.level
very
# cat  /etc/config/keys/special.type
charm
```

使用 subpath 将 ConfigMap 作为单独的文件挂载到目录

在一般情况下 configmap 挂载文件时，会先覆盖掉挂载目录，然后再将 congfigmap 中的内容作为文件挂载进行。如果想不对原来的文件夹下的文件造成覆盖，只是将 configmap 中的每个 key，按照文件的方式挂载到目录下，可以使用 subpath 参数。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      command: ["/bin/sh","-c","sleep 36000"]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/nginx/special.how
        subPath: special.how
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: special.how
  restartPolicy: Never
```

```
root@dapi-test-pod:/# ls /etc/nginx/
conf.d    fastcgi_params    koi-utf  koi-win  mime.types  modules
root@dapi-test-pod:/# cat /etc/nginx/special.how
very
root@dapi-test-pod:/#
```

参考文档：

- ConfigMap

CronJob

CronJob 即定时任务，就类似于 Linux 系统的 crontab，在指定的时间周期运行指定的任务。

API 版本对照表

Kubernetes 版本	Batch API 版本	默认开启
v1.5-v1.7	batch/v2alpha1	否
v1.8-v1.9	batch/v1beta1	是

注意：使用默认未开启的 API 时需要在 kube-apiserver 中配置 `--runtime-config=batch/v2alpha1` 。

CronJob Spec

- `.spec.schedule` 指定任务运行周期，格式同 Cron
- `.spec.jobTemplate` 指定需要运行的任务，格式同 Job
- `.spec.startingDeadlineSeconds` 指定任务开始的截止期限
- `.spec.concurrencyPolicy` 指定任务的并发策略，支持 Allow、Forbid 和 Replace 三个选项

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

```
$ kubectl create -f cronjob.yaml
cronjob "hello" created
```

当然，也可以用 `kubectl run` 来创建一个 CronJob：

```
kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --
```

```
$ kubectl get cronjob
NAME      SCHEDULE      SUSPEND   ACTIVE    LAST-SCHEDULE
hello     */1 * * * *   False     0
$ kubectl get jobs
NAME               DESIRED   SUCCESSFUL   AGE
hello-1202039034   1         1            49s
$ pods=$(kubectl get pods --selector=job-name=hello-1202039034 --
$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
Hello from the Kubernetes cluster

# 注意，删除 cronjob 的时候不会自动删除 job，这些 job 可以用 kubectl delet
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

参考文档

- Cron Jobs

CustomResourceDefinition

CustomResourceDefinition（CRD）是 v1.7 新增的无需改变代码就可以扩展 Kubernetes API 的机制，用来管理自定义对象。它实际上是 ThirdPartyResources（TPR） 的升级版本，而 TPR 已经在 v1.8 中删除。

API 版本对照表

Kubernetes 版本	CRD API 版本
v1.8+	apiextensions.k8s.io/v1beta1

CRD 示例

下面的例子会创建一个 `/apis/stable.example.com/v1/namespaces//crontabs/…` 的自定义 API：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: .
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis//
  group: stable.example.com
  # versions to use for REST API: /apis//
  versions:
  - name: v1beta1
    # Each version can be enabled/disabled by Served flag.
    served: true
    # One and only one version must be marked as the storage vers
    storage: true
  - name: v1
    served: true
    storage: false
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis///
    plural: crontabs
    # singular name to be used as an alias on the CLI and for dis
    singular: crontab
    # kind is normally the CamelCased singular type. Your resourc
    kind: CronTab
    # shortNames allow shorter string to match your resource on t
    shortNames:
    - ct
```

API 创建好后，就可以创建具体的 CronTab 对象了

```
$ cat my-cronjob.yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image

$ kubectl create -f my-crontab.yaml
crontab "my-new-cron-object" created

$ kubectl get crontab
NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
$ kubectl get crontab my-new-cron-object -o yaml
apiVersion: stable.example.com/v1
kind: CronTab
metadata:
  creationTimestamp: 2017-07-03T19:00:56Z
  name: my-new-cron-object
  namespace: default
  resourceVersion: "20630"
  selfLink: /apis/stable.example.com/v1/namespaces/default/cronta
  uid: 5c82083e-5fbd-11e7-a204-42010a8c0002
spec:
  cronSpec: '* * * * /5'
  image: my-awesome-cron-image
```

Finalizer

Finalizer 用于实现控制器的异步预删除钩子，可以通过 `metadata.finalizers` 来指定 Finalizer。

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
  - finalizer.stable.example.com
```

Finalizer 指定后，客户端删除对象的操作只会设置 `metadata.deletionTimestamp` 而不是直接删除。这会触发正在监听 CRD 的控制器，控制器执行一些删除前的清理操作，从列表中删除自己的 finalizer，然后再重新发起一个删除操作。此时，被删除的对象才会真正删除。

Validation

v1.8 开始新增了实验性的基于 OpenAPI v3 schema 的验证（Validation）机制，可以用来提前验证用户提交的资源是否符合规范。使用该功能需要配置 kube-apiserver 的 `--feature-gates=CustomResourceValidation=true`。

比如下面的 CRD 要求

- `spec.cronSpec` 必须是匹配正则表达式的字符串
- `spec.replicas` 必须是从 1 到 10 的整数

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
  validation:
   # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      properties:
        spec:
          properties:
            cronSpec:
              type: string
              pattern: '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+)?){4}$'
            replicas:
              type: integer
              minimum: 1
              maximum: 10
```

这样，在创建下面的 CronTab 时

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * *"
  image: my-awesome-cron-image
  replicas: 15
```

会报验证失败的错误：

```
The CronTab "my-new-cron-object" is invalid: []: Invalid value: m
validation failure list:
spec.cronSpec in body should match '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(
spec.replicas in body should be less than or equal to 10
```

Subresources

v1.10 开始 CRD 还支持 `/status` 和 `/scale` 等两个子资源（Beta），并且从 v1.11 开始默认开启。

> v1.10 版本使用前需要在 `kube-apiserver` 开启 `--feature-gates=CustomResourceSubresources=true` 。

```yaml
# resourcedefinition.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
  # subresources describes the subresources for custom resources.
  subresources:
    # status enables the status subresource.
    status: {}
    # scale enables the scale subresource.
    scale:
      # specReplicasPath defines the JSONPath inside of a custom
      specReplicasPath: .spec.replicas
      # statusReplicasPath defines the JSONPath inside of a custo
      statusReplicasPath: .status.replicas
      # labelSelectorPath defines the JSONPath inside of a custom
      labelSelectorPath: .status.labelSelector
```

```
$ kubectl create -f resourcedefinition.yaml
$ kubectl create -f- <"stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
  replicas: 3
EOF

$ kubectl scale --replicas=5 crontabs/my-new-cron-object
crontabs "my-new-cron-object" scaled

$ kubectl get crontabs my-new-cron-object -o jsonpath='{.spec.rep
5
```

Categories

Categories 用来将 CRD 对象分组，这样就可以使用 `kubectl get` 来查询属于该组的所有对象。

```
# resourcedefinition.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
    # categories is a list of grouped resources the custom resour
    categories:
    - all
```

```
# my-crontab.yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

```
$ kubectl create -f resourcedefinition.yaml
$ kubectl create -f my-crontab.yaml
$ kubectl get all
NAME                          AGE
crontabs/my-new-cron-object   3s
```

CRD 控制器

在使用 CRD 扩展 Kubernetes API 时，通常还需要实现一个新建资源的控制器，监听新资源的变化情况，并作进一步的处理。

https://github.com/kubernetes/sample-controller 提供了一个 CRD 控制器的示例，包括

- 如何注册资源 `Foo`
- 如何创建、删除和查询 `Foo` 对象
- 如何监听 `Foo` 资源对象的变化情况

Kubebuilder

从上面的实例中可以看到从头构建一个 CRD 控制器并不容易，需要对 Kubernetes 的 API 有深入了解，并且RBAC 集成、镜像构建、持续集成和部署等都需要很大工作量。

kubebuilder 正是为解决这个问题而生，为 CRD 控制器提供了一个简单易用的框架，并可直接生成镜像构建、持续集成、持续部署等所需的资源文件。

安装

```
# Install kubebuilder
VERSION=1.0.1
wget https://github.com/kubernetes-sigs/kubebuilder/releases/downr
tar zxvf kubebuilder_${VERSION}_linux_amd64.tar.gz
sudo mv kubebuilder_${VERSION}_linux_amd64 /usr/local/kubebuilder
export PATH=$PATH:/usr/local/kubebuilder/bin

# Install dep kustomize
go get -u github.com/golang/dep/cmd/dep
go get github.com/kubernetes-sigs/kustomize
```

使用方法

初始化项目

```
mkdir -p $GOPATH/src/demo
cd $GOPATH/src/demo
kubebuilder init --domain k8s.io --license apache2 --owner "The K
```

创建 API

```
kubebuilder create api --group ships --version v1beta1 --kind Slo
```

然后按照实际需要修改 `pkg/apis/ship/v1beta1/sloop_types.go` 和 `pkg/controller/sloop/sloop_controller.go` 增加业务逻辑。

本地运行测试

```
make install
make run
```

> 如果碰到错误 `ValidationError(CustomResourceDefinition.status): missing required field "storedVersions" in io.k8s.apiextensions-apiserver.pkg.apis.apiextensions.v1beta1.CustomResourceDefinitionStatus]` ，可以手动修改 `config/crds/ships_v1beta1_sloop.yaml` : ```yaml status: acceptedNames: kind: "" plural: "" conditions: [] storedVersions: []
>
> 然后运行 `kubectl apply -f config/crds` 创建 CRD。

然后就可以用 `ships.k8s.io/v1beta1` 来创建 Kind 为 `Sloop` 的资源了，比如

```
kubectl apply -f config/samples/ships_v1beta1_sloop.yaml
```

构建镜像并部署控制器

```
# 替换 IMG 为你自己的
export IMG=feisky/demo-crd:v1
make docker-build
make docker-push
make deploy
```

> kustomize 已经不再支持通配符，因而上述 `make deploy` 可能会碰到 `Load from path ../rbac/*.yaml failed` 错误，解决方法是手动修改 `config/default/kustomization.yaml`：
>
> resources:
>
> - ../rbac/rbac_role.yaml
> - ../rbac/rbac_role_binding.yaml
> - ../manager/manager.yaml
>
> 然后执行 `kustomize build config/default | kubectl apply -f -` 部署，默认部署到 `demo-system` namespace 中。

文档和测试

```
# run unit tests
make test

# generate docs
kubebuilder docs
```

参考文档

- Extend the Kubernetes API with CustomResourceDefinitions
- CustomResourceDefinition API

DaemonSet

DaemonSet 保证在每个 Node 上都运行一个容器副本，常用来部署一些集群的日志、监控或者其他系统管理应用。典型的应用包括：

- 日志收集，比如 fluentd，logstash 等
- 系统监控，比如 Prometheus Node Exporter，collectd，New Relic agent，Ganglia gmond 等
- 系统程序，比如 kube-proxy, kube-dns, glusterd, ceph 等

API 版本对照表

Kubernetes 版本	Deployment 版本
v1.5-v1.6	extensions/v1beta1
v1.7	apps/v1beta1
v1.8	apps/v1beta2
v1.9	apps/v1

使用 Fluentd 收集日志的例子：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: gcr.io/google-containers/fluentd-elasticsearch:1.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
```

```
        hostPath:
          path: /var/lib/docker/containers
```

滚动更新

v1.6 + 支持 DaemonSet 的滚动更新，可以通过 `.spec.updateStrategy.type` 设置更新策略。目前支持两种策略

- OnDelete：默认策略，更新模板后，只有手动删除了旧的 Pod 后才会创建新的 Pod
- RollingUpdate：更新 DaemonSet 模版后，自动删除旧的 Pod 并创建新的 Pod

在使用 RollingUpdate 策略时，还可以设置

- `.spec.updateStrategy.rollingUpdate.maxUnavailable`，默认 1
- `spec.minReadySeconds`，默认 0

回滚

v1.7 + 还支持回滚

```
# 查询历史版本
$ kubectl rollout history daemonset

# 查询某个历史版本的详细信息
$ kubectl rollout history daemonset  --revision=1

# 回滚
$ kubectl rollout undo daemonset  --to-revision=
# 查询回滚状态
$ kubectl rollout status ds/
```

指定 Node 节点

DaemonSet 会忽略 Node 的 unschedulable 状态，有两种方式来指定 Pod 只运行在指定的 Node 节点上：

- nodeSelector：只调度到匹配指定 label 的 Node 上

- nodeAffinity：功能更丰富的 Node 选择器，比如支持集合操作
- podAffinity：调度到满足条件的 Pod 所在的 Node 上

nodeSelector 示例

首先给 Node 打上标签

```
kubectl label nodes node-01 disktype=ssd
```

然后在 daemonset 中指定 nodeSelector 为 `disktype=ssd` ：

```
spec:
  nodeSelector:
    disktype: ssd
```

nodeAffinity 示例

nodeAffinity 目前支持两种：requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution，分别代表必须满足条件和优选条件。比如下面的例子代表调度到包含标签 `kubernetes.io/e2e-az-name` 并且值为 e2e-az1 或 e2e-az2 的 Node 上，并且优选还带有标签 `another-node-label-key=another-node-label-value` 的 Node。

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: gcr.io/google_containers/pause:2.0
```

podAffinity 示例

podAffinity 基于 Pod 的标签来选择 Node，仅调度到满足条件 Pod 所在的 Node 上，支持 podAffinity 和 podAntiAffinity。这个功能比较绕，以下面的例子为例：

- 如果一个 “Node 所在 Zone 中包含至少一个带有 `security=S1` 标签且运行中的 Pod”，那么可以调度到该 Node
- 不调度到 “包含至少一个带有 `security=S2` 标签且运行中 Pod” 的 Node 上

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: gcr.io/google_containers/pause:2.0
```

静态 Pod

除了 DaemonSet，还可以使用静态 Pod 来在每台机器上运行指定的 Pod，这需要 kubelet 在启动的时候指定 manifest 目录：

```
kubelet --pod-manifest-path=/etc/kubernetes/manifests
```

然后将所需要的 Pod 定义文件放到指定的 manifest 目录中。

注意：静态 Pod 不能通过 API Server 来删除，但可以通过删除 manifest 文件来自动删除对应的 Pod。

Deployment

简述

Deployment 为 Pod 和 ReplicaSet 提供了一个声明式定义 (declarative) 方法，用来替代以前的 ReplicationController 来方便的管理应用。

API 版本对照表

Kubernetes 版本	Deployment 版本
v1.5-v1.6	extensions/v1beta1
v1.7	apps/v1beta1
v1.8	apps/v1beta2
v1.9	apps/v1

比如一个简单的 nginx 应用可以定义为

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

扩容：

```
kubectl scale deployment nginx-deployment --replicas 10
```

如果集群支持 horizontal pod autoscaling 的话，还可以为 Deployment 设置自动扩展：

```
kubectl autoscale deployment nginx-deployment --min=10 --max=15 -
```

更新镜像也比较简单:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

回滚：

```
kubectl rollout undo deployment/nginx-deployment
```

Deployment 的 **典型应用场景** 包括：

- 定义 Deployment 来创建 Pod 和 ReplicaSet
- 滚动升级和回滚应用
- 扩容和缩容
- 暂停和继续 Deployment

Deployment 概念解析

Deployment 是什么？

Deployment 为 Pod 和 Replica Set（下一代 Replication Controller）提供声明式更新。

你只需要在 Deployment 中描述你想要的目标状态是什么，Deployment controller 就会帮你将 Pod 和 Replica Set 的实际状态改变到你的目标状态。你可以定义一个全新的 Deployment，也可以创建一个新的替换旧的 Deployment。

一个典型的用例如下：

- 使用 Deployment 来创建 ReplicaSet。ReplicaSet 在后台创建 pod。检查启动状态，看它是成功还是失败。

- 然后，通过更新 Deployment 的 PodTemplateSpec 字段来声明 Pod 的新状态。这会创建一个新的 ReplicaSet，Deployment 会按照控制的速率将 pod 从旧的 ReplicaSet 移动到新的 ReplicaSet 中。
- 如果当前状态不稳定，回滚到之前的 Deployment revision。每次回滚都会更新 Deployment 的 revision。
- 扩容 Deployment 以满足更高的负载。
- 暂停 Deployment 来应用 PodTemplateSpec 的多个修复，然后恢复上线。
- 根据 Deployment 的状态判断上线是否 hang 住了。
- 清除旧的不必要的 ReplicaSet。

创建 Deployment

下面是一个 Deployment 示例，它创建了一个 Replica Set 来启动 3 个 nginx pod。

下载示例文件并执行命令：

```
$ kubectl create -f docs/user-guide/nginx-deployment.yaml --recor
deployment "nginx-deployment" created
```

将 kubectl 的 `-record` 的 flag 设置为 `true` 可以在 annotation 中记录当前命令创建或者升级了该资源。这在未来会很有用，例如，查看在每个 Deployment revision 中执行了哪些命令。

然后立即执行 `get` 将获得如下结果：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A
nginx-deployment   3         0         0            0           1
```

输出结果表明我们希望的 repalica 数是 3（根据 deployment 中的 `.spec.replicas` 配置）当前 replica 数（ `.status.replicas` ）是 0, 最新的 replica 数（ `.status.updatedReplicas` ）是 0，可用的 replica 数（ `.status.availableReplicas` ）是 0。

过几秒后再执行 `get` 命令，将获得如下输出：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A
nginx-deployment   3         3         3            3           1
```

我们可以看到 Deployment 已经创建了 3 个 replica，所有的 replica 都已经是最新的了（包含最新的 pod template），可用的（根据 Deployment 中的 `.spec.minReadySeconds` 声明，处于已就绪状态的 pod 的最少个数）。执行 `kubectl get rs` 和 `kubectl get pods` 会显示 Replica Set（RS）和 Pod 已创建。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-2035384211   3         3         0       18s
```

你可能会注意到 Replica Set 的名字总是 `-` 。

```
$ kubectl get pods --show-labels
NAME                                READY     STATUS    RESTARTS
nginx-deployment-2035384211-7ci7o   1/1       Running   0
nginx-deployment-2035384211-kzszj   1/1       Running   0
nginx-deployment-2035384211-qqcnn   1/1       Running   0
```

刚创建的 Replica Set 将保证总是有 3 个 nginx 的 pod 存在。

注意： 你必须在 Deployment 中的 selector 指定正确 pod template label（在该示例中是 `app = nginx` ），不要跟其他的 controller 搞混了（包括 Deployment、Replica Set、Replication Controller 等）。**Kubernetes 本身不会阻止你这么做** ，如果你真的这么做了，这些 controller 之间会相互打架，并可能导致不正确的行为。

更新 Deployment

注意： Deployment 的 rollout 当且仅当 Deployment 的 pod template（例如 `.spec.template` ）中的 label 更新或者镜像更改时被触发。其他更新，例如扩容 Deployment 不会触发 rollout。

假如我们现在想要让 nginx pod 使用 `nginx:1.9.1` 的镜像来代替原来的 `nginx:1.7.9` 的镜像。

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

我们可以使用 `edit` 命令来编辑 Deployment，修改 `.spec.template.spec.containers[0].image` ，将 `nginx:1.7.9` 改写成 `nginx:1.9.1` 。

```
$ kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
```

查看 rollout 的状态，只要执行：

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

Rollout 成功后， `get` Deployment：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A
nginx-deployment   3         3         3            3           3
```

UP-TO-DATE 的 replica 的数目已经达到了配置中要求的数目。

CURRENT 的 replica 数表示 Deployment 管理的 replica 数量，AVAILABLE 的 replica 数是当前可用的 replica 数量。

我们通过执行 `kubectl get rs` 可以看到 Deployment 更新了 Pod，通过创建一个新的 Replica Set 并扩容了 3 个 replica，同时将原来的 Replica Set 缩容到了 0 个 replica。

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

执行 `get pods` 只会看到当前的新的 pod:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS
nginx-deployment-1564180365-khku8   1/1       Running   0
nginx-deployment-1564180365-nacti   1/1       Running   0
nginx-deployment-1564180365-z9gth   1/1       Running   0
```

下次更新这些 pod 的时候，只需要更新 Deployment 中的 pod 的 template 即可。

Deployment 可以保证在升级时只有一定数量的 Pod 是 down 的。默认的，它会确保至少有比期望的 Pod 数量少一个的 Pod 是 up 状态（最多一个不可用）。

Deployment 同时也可以确保只创建出超过期望数量的一定数量的 Pod。默认的，它会确保最多比期望的 Pod 数量多一个的 Pod 是 up 的（最多 1 个 surge）。

在未来的 Kuberentes 版本中，将从 1-1 变成 25%-25%）。

例如，如果你自己看下上面的 Deployment，你会发现，开始创建一个新的 Pod，然后删除一些旧的 Pod 再创建一个新的。当新的 Pod 创建出来之前不会杀掉旧的 Pod。这样能够确保可用的 Pod 数量至少有 2 个，Pod 的总数最多 4 个。

```
$ kubectl describe deployments
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 12:01:06 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas cre
Events:
  FirstSeen LastSeen    Count   From                    Subobjec
  --------- --------    -----   ----                    --------
  36s       36s         1       {deployment-controller}
  23s       23s         1       {deployment-controller}
  23s       23s         1       {deployment-controller}
  23s       23s         1       {deployment-controller}
  21s       21s         1       {deployment-controller}
  21s       21s         1       {deployment-controller}
```

我们可以看到当我们刚开始创建这个 Deployment 的时候，创建了一个 Replica Set（nginx-deployment-2035384211），并直接扩容到了 3 个 replica。

当我们更新这个 Deployment 的时候，它会创建一个新的 Replica Set（nginx-deployment-1564180365），将它扩容到 1 个 replica，然后缩容原先的 Replica Set 到 2 个 replica，此时满足至少 2 个 Pod 是可用状态，同一时刻最多有 4 个 Pod 处于创建的状态。

接着继续使用相同的 rolling update 策略扩容新的 Replica Set 和缩容旧的 Replica Set。最终，将会在新的 Replica Set 中有 3 个可用的 replica，旧的 Replica Set 的 replica 数目变成 0。

Rollover（多个 rollout 并行）

每当 Deployment controller 观测到有新的 deployment 被创建时，如果没有已存在的 Replica Set 来创建期望个数的 Pod 的话，就会创建出一个新的 Replica Set 来做这件事。已存在的 Replica Set 控制 label 匹配 `.spec.selector` 但是 template 跟 `.spec.template` 不匹配的 Pod 缩容。最终，新的 Replica Set 将会扩容出 `.spec.replicas` 指定数目的 Pod，旧的 Replica Set 会缩容到 0。

如果你更新了一个的已存在并正在进行中的 Deployment，每次更新 Deployment 都会创建一个新的 Replica Set 并扩容它，同时回滚之前扩容的 Replica Set——将它添加到旧的 Replica Set 列表，开始缩容。

例如，假如你创建了一个有 5 个 `niginx:1.7.9` replica 的 Deployment，但是当还只有 3 个 `nginx:1.7.9` 的 replica 创建出来的时候你就开始更新含有 5 个 `nginx:1.9.1` replica 的 Deployment。在这种情况下，Deployment 会立即杀掉已创建的 3 个 `nginx:1.7.9` 的 Pod，并开始创建 `nginx:1.9.1` 的 Pod。它不会等到所有的 5 个 `nginx:1.7.9` 的 Pod 都创建完成后才开始执行滚动更新。

回退 Deployment

有时候你可能想回退一个 Deployment，例如，当 Deployment 不稳定时，比如一直 crash looping。

默认情况下，kubernetes 会在系统中保存所有的 Deployment 的 rollout 历史记录，以便你可以随时回退（你可以修改 `revision history limit` 来更改保存的 revision 数）。

注意： 只要 Deployment 的 rollout 被触发就会创建一个 revision。也就是说当且仅当 Deployment 的 Pod template（如 `.spec.template`）被更改，例如更新 template 中的 label 和容器镜像时，就会创建出一个新的 revision。

其他的更新，比如扩容 Deployment 不会创建 revision——因此我们可以很方便的手动或者自动扩容。这意味着当你回退到历史 revision 时，只有 Deployment 中的 Pod template 部分才会回退。

假设我们在更新 Deployment 的时候犯了一个拼写错误，将镜像的名字写成了 `nginx:1.91` ，而正确的名字应该是 `nginx:1.9.1` ：

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

Rollout 将会卡住。

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

按住 Ctrl-C 停止上面的 rollout 状态监控。

你会看到旧的 replicas（nginx-deployment-1564180365 和 nginx-deployment-2035384211）和新的 replicas （nginx-deployment-3066724191）数目都是 2 个。

```
$ 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
```

看下创建 Pod，你会看到有两个新的 Replica Set 创建的 Pod 处于 ImagePullBackOff 状态，循环拉取镜像。

```
$ kubectl get pods
NAME                                READY     STATUS
nginx-deployment-1564180365-70iae   1/1       Running
nginx-deployment-1564180365-jbqqo   1/1       Running
nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff
nginx-deployment-3066724191-eocby   0/1       ImagePullBackOff
```

注意，Deployment controller 会自动停止坏的 rollout，并停止扩容新的 Replica Set。

```
$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       2 updated | 3 total | 2 available | 2 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     nginx-deployment-1564180365 (2/2 replicas cre
NewReplicaSet:      nginx-deployment-3066724191 (2/2 replicas cre
Events:
  FirstSeen LastSeen    Count   From                    Subobject
  --------- --------    -----   ----                    ---------
  1m        1m          1       {deployment-controller}
  22s       22s         1       {deployment-controller}
  22s       22s         1       {deployment-controller}
  22s       22s         1       {deployment-controller}
  21s       21s         1       {deployment-controller}
  21s       21s         1       {deployment-controller}
  13s       13s         1       {deployment-controller}
  13s       13s         1       {deployment-controller}
  13s       13s         1       {deployment-controller}
```

为了修复这个问题，我们需要回退到稳定的 Deployment revision。

检查 Deployment 升级的历史记录

首先，检查下 Deployment 的 revision：

```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment":
REVISION    CHANGE-CAUSE
1           kubectl create -f docs/user-guide/nginx-deployment.ya
2           kubectl set image deployment/nginx-deployment nginx=n
3           kubectl set image deployment/nginx-deployment nginx=n
```

因为我们创建 Deployment 的时候使用了 `–recored` 参数可以记录命令，我们可以很方便的查看每次 revison 的变化。

查看单个 revision 的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Annotations:  kubernetes.io/change-cause=kubectl set image depl
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
    Environment Variables:
  No volumes.
```

回退到历史版本

现在，我们可以决定回退当前的 rollout 到之前的版本：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
```

也可以使用 `--to-revision` 参数指定某个历史版本：

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=
deployment "nginx-deployment" rolled back
```

与 rollout 相关的命令详细文档见 kubectl rollout。

该 Deployment 现在已经回退到了先前的稳定版本。如你所见，Deployment controller 产生了一个回退到 revison 2 的 `DeploymentRollback` 的 event。

```
$ kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   A
nginx-deployment   3         3         3            3           3

$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas cre
Events:
  FirstSeen LastSeen    Count   From                    Subobject
  --------- --------    -----   ----                    ---------
  30m       30m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  29m       29m         1       {deployment-controller}
  2m        2m          1       {deployment-controller}
  2m        2m          1       {deployment-controller}
  29m       2m          2       {deployment-controller}
```

清理 Policy

你可以通过设置 `.spec.revisonHistoryLimit` 项来指定 deployment 最多保留多少 revison 历史记录。默认的会保留所有的 revision；如果将该项设置为 0，Deployment 就不允许回退了。

Deployment 扩容

你可以使用以下命令扩容 Deployment：

```
$ kubectl scale deployment nginx-deployment --replicas 10
deployment "nginx-deployment" scaled
```

假设你的集群中启用了 horizontal pod autoscaling (HPA)，你可以给 Deployment 设置一个 autoscaler，基于当前 Pod 的 CPU 利用率选择最少和最多的 Pod 数。

```
$ kubectl autoscale deployment nginx-deployment --min=10 --max=15
deployment "nginx-deployment" autoscaled
```

比例扩容

RollingUpdate Deployment 支持同时运行一个应用的多个版本。当你或者 autoscaler 扩容一个正在 rollout 中（进行中或者已经暂停）的 RollingUpdate Deployment 的时候，为了降低风险，Deployment controller 将会平衡已存在的 active 的 ReplicaSets（有 Pod 的 ReplicaSets）和新加入的 replicas。这被称为比例扩容。

例如，你正在运行中含有 10 个 replica 的 Deployment。maxSurge=3，maxUnavailable=2。

```
$ kubectl get deploy
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
nginx-deployment    10        10        10           10
```

你更新了一个镜像，而在集群内部无法解析。

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

镜像更新启动了一个包含 ReplicaSet nginx-deployment-1989198191 的新的 rollout，但是它被阻塞了，因为我们上面提到的 maxUnavailable。

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

然后发起了一个新的 Deployment 扩容请求。autoscaler 将 Deployment 的 replica 数目增加到了 15 个。Deployment controller 需要判断在哪里增加这 5 个新的 replica。如果我们没有使用比例扩容，所有的 5 个 replica 都会加到一个新的 ReplicaSet 中。如果使用比例扩容，新添加的 replica 将传播到所有的 ReplicaSet 中。大的部分加入 replica 数最多的 ReplicaSet 中，小的部分加入到 replica 数少的 ReplciaSet 中。0 个 replica 的 ReplicaSet 不会被扩容。

在我们上面的例子中，3 个 replica 将添加到旧的 ReplicaSet 中，2 个 replica 将添加到新的 ReplicaSet 中。rollout 进程最终会将所有的 replica 移动到新的 ReplicaSet 中，假设新的 replica 成为健康状态。

```
$ kubectl get deploy
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
nginx-deployment    15        18        7            8
$ kubectl get rs
NAME                           DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191    7         7         0         7m
nginx-deployment-618515232     11        11        11        7m
```

暂停和恢复 Deployment

你可以在触发一次或多次更新前暂停一个 Deployment，然后再恢复它。这样你就能多次暂停和恢复 Deployment，在此期间进行一些修复工作，而不会触发不必要的 rollout。

例如使用刚刚创建 Deployment：

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
[mkargaki@dhcp129-211 kubernetes]$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         1m
```

使用以下命令暂停 Deployment：

```
$ kubectl rollout pause deployment/nginx-deployment
deployment "nginx-deployment" paused
```

然后更新 Deplyment 中的镜像：

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

注意没有启动新的 rollout：

```
$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         2m
```

你可以进行任意多次更新，例如更新使用的资源：

```
$ kubectl set resources deployment nginx -c=nginx --limits=cpu=20
deployment "nginx" resource requirements updated
```

Deployment 暂停前的初始状态将继续它的功能，而不会对 Deployment 的更新产生任何影响，只要 Deployment 是暂停的。

最后，恢复这个 Deployment，观察完成更新的 ReplicaSet 已经创建出来了：

```
$ kubectl rollout resume deploy nginx
deployment "nginx" resumed
$ KUBECTL get rs -w
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   2         2         2         2m
nginx-3926361531   2         2         0         6s
nginx-3926361531   2         2         1         18s
nginx-2142116321   1         2         2         2m
nginx-2142116321   1         2         2         2m
nginx-3926361531   3         2         1         18s
nginx-3926361531   3         2         1         18s
nginx-2142116321   1         1         1         2m
nginx-3926361531   3         3         1         18s
nginx-3926361531   3         3         2         19s
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         20s
^C
$ KUBECTL get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         28s
```

注意： 在恢复 Deployment 之前你无法回退一个暂停了的 Deployment。

Deployment 状态

Deployment 在生命周期中有多种状态。在创建一个新的 ReplicaSet 的时候它可以是 progressing 状态， complete 状态，或者 fail to progress 状态。

Progressing Deployment

Kubernetes 将执行过下列任务之一的 Deployment 标记为 *progressing* 状态：

- Deployment 正在创建新的 ReplicaSet 过程中。
- Deployment 正在扩容一个已有的 ReplicaSet。
- Deployment 正在缩容一个已有的 ReplicaSet。
- 有新的可用的 pod 出现。

你可以使用 `kubectl rollout status` 命令监控 Deployment 的进度。

Complete Deployment

Kubernetes 将包括以下特性的 Deployment 标记为 *complete* 状态：

- Deployment 最小可用。最小可用意味着 Deployment 的可用 replica 个数等于或者超过 Deployment 策略中的期望个数。
- 所有与该 Deployment 相关的 replica 都被更新到了你指定版本，也就说更新完成。
- 该 Deployment 中没有旧的 Pod 存在。

你可以用 `kubectl rollout status` 命令查看 Deployment 是否完成。如果 rollout 成功完成，`kubectl rollout status` 将返回一个 0 值的 Exit Code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 of 3 updated replicas are availa
deployment "nginx" successfully rolled out
$ echo $?
0
```

Failed Deployment

你的 Deployment 在尝试部署新的 ReplicaSet 的时候可能卡住，永远也不会完成。这可能是因为以下几个因素引起的：

- 无效的引用
- 不可读的 probe failure
- 镜像拉取错误
- 权限不够
- 范围限制
- 程序运行时配置错误

探测这种情况的一种方式是，在你的 Deployment spec 中指定 `spec.progressDeadlineSeconds`。`spec.progressDeadlineSeconds` 表示 Deployment controller 等待多少秒才能确定（通过 Deployment status）Deployment 进程是卡住的。

下面的 `kubectl` 命令设置 `progressDeadlineSeconds` 使 controller 在 Deployment 在进度卡住 10 分钟后报告：

```
$ kubectl patch deployment/nginx-deployment -p '{"spec":{"progres
"nginx-deployment" patched
```

当超过截止时间后，Deployment controller 会在 Deployment 的 `status.conditions` 中增加一条 DeploymentCondition，它包括如下属性：

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

浏览 Kubernetes API conventions 查看关于 status conditions 的更多信息。

注意: kubernetes 除了报告 `Reason=ProgressDeadlineExceeded` 状态信息外不会对卡住的 Deployment 做任何操作。更高层次的协调器可以利用它并采取相应行动，例如，回滚 Deployment 到之前的版本。

注意： 如果你暂停了一个 Deployment，在暂停的这段时间内 kubernetnes 不会检查你指定的 deadline。你可以在 Deployment 的 rollout 途中安全的暂停它，然后再恢复它，这不会触发超过 deadline 的状态。

你可能在使用 Deployment 的时候遇到一些短暂的错误，这些可能是由于你设置了太短的 timeout，也有可能是因为各种其他错误导致的短暂错误。例如，假设你使用了无效的引用。当你 Describe Deployment 的时候可能会注意到如下信息：

```
$ kubectl describe deployment nginx-deployment
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

执行 `kubectl get deployment nginx-deployment -o yaml`，Deployement 的状态可能看起来像这个样子：

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progres
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods"nginx-deployment-4262182780-"
      object-counts, requested: pods=1, used: pods=3, limited: pc
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

最终，一旦超过 Deployment 进程的 deadline，kuberentes 会更新状态和导致 Progressing 状态的原因：

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

你可以通过缩容 Deployment 的方式解决配额不足的问题，或者增加你的 namespace 的配额。如果你满足了配额条件后，Deployment controller 就会完成你的 Deployment rollout，你将看到 Deployment 的状态更新为成功状态（ `Status=True` 并且 `Reason=NewReplicaSetAvailable` ）。

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

`Type=Available` 、 `Status=True` 意味着你的 Deployment 有最小可用性。 最小可用性是在 Deployment 策略中指定的参数。 `Type=Progressing` 、 `Status=True` 意味着你的 Deployment 或者在部署过程中，或者已经成功部署，达到了期望的最少的可用 replica 数量（查看特定状态的 Reason——在我们的例子中 `Reason=NewReplicaSetAvailable` 意味着 Deployment 已经完成）。

你可以使用 `kubectl rollout status` 命令查看 Deployment 进程是否失败。当 Deployment 过程超过了 deadline， `kubectl rollout status` 将返回非 0 的 exit code。

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

操作失败的 Deployment

所有对完成的 Deployment 的操作都适用于失败的 Deployment。你可以对它扩 / 缩容，回退到历史版本，你甚至可以多次暂停它来应用 Deployment pod template。

清理 Policy

你可以设置 Deployment 中的 `.spec.revisionHistoryLimit` 项来指定保留多少旧的 ReplicaSet。 余下的将在后台被当作垃圾收集。默认的，所有的 revision 历史都会被保留。在未来的版本中，将会更改为 2。

注意： 将该值设置为 0，将导致该 Deployment 的所有历史记录都被清除，也就无法回退了。

用例

Canary Deployment

如果你想要使用 Deployment 对部分用户或服务器发布 release，你可以创建多个 Deployment，每个对一个 release，参照 managing resources 中对 canary 模式的描述。

编写 Deployment Spec

在所有的 Kubernetes 配置中，Deployment 也需要 `apiVersion`，`kind` 和 `metadata` 这些配置项。配置文件的通用使用说明查看 部署应用，配置容器，和使用 kubeclt 管理资源 文档。

Deployment 也需要 `.spec` section.

Pod Template

`.spec.template` 是 `.spec` 中唯一要求的字段。

`.spec.template` 是 pod template. 它跟 Pod 有一模一样的 schema，除了它是嵌套的并且不需要 `apiVersion` 和 `kind` 字段。

另外为了划分 Pod 的范围，Deployment 中的 pod template 必须指定适当的 label（不要跟其他 controller 重复了，参考 selector）和适当的重启策略。

`.spec.template.spec.restartPolicy` 可以设置为 `Always`，如果不指定的话这就是默认配置。

Replicas

`.spec.replicas` 是可以选字段，指定期望的 pod 数量，默认是 1。

Selector

`.spec.selector` 是可选字段，用来指定 label selector ，圈定 Deployment 管理的 pod 范围。

如果被指定， `.spec.selector` 必须匹配 `.spec.template.metadata.labels` ，否则它将被 API 拒绝。如果 `.spec.selector` 没有被指定， `.spec.selector.matchLabels` 默认是 `.spec.template.metadata.labels` 。

在 Pod 的 template 跟 `.spec.template` 不同或者数量超过了 `.spec.replicas` 规定的数量的情况下，Deployment 会杀掉 label 跟 selector 不同的 Pod。

注意： 你不应该再创建其他 label 跟这个 selector 匹配的 pod，或者通过其他 Deployment，或者通过其他 Controller，例如 ReplicaSet 和 ReplicationController。否则该 Deployment 会被把它们当成都是自己创建的。Kubernetes 不会阻止你这么做。

如果你有多个 controller 使用了重复的 selector，controller 们就会互相打架并导致不正确的行为。

策略

`.spec.strategy` 指定新的 Pod 替换旧的 Pod 的策略。 `.spec.strategy.type` 可以是 "Recreate" 或者是 "RollingUpdate"。"RollingUpdate" 是默认值。

Recreate Deployment

`.spec.strategy.type==Recreate` 时，在创建出新的 Pod 之前会先杀掉所有已存在的 Pod。

Rolling Update Deployment

`.spec.strategy.type==RollingUpdate` 时，Deployment 使用 rolling update 的方式更新 Pod 。你可以指定 `maxUnavailable` 和 `maxSurge` 来控制 rolling update 进程。

Max Unavailable

`.spec.strategy.rollingUpdate.maxUnavailable` 是可选配置项，用来指定在升级过程中不可用 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%）。通过计算百分比的绝对值向下取整。如果 `.spec.strategy.rollingUpdate.maxSurge` 为 0 时，这个值不可以为 0。默认值是 1。

例如，该值设置成 30%，启动 rolling update 后旧的 ReplicatSet 将会立即缩容到期望的 Pod 数量的 70%。新的 Pod ready 后，随着新的 ReplicaSet 的扩容，旧的 ReplicaSet 会进一步缩容，确保在升级的所有时刻可以用的 Pod 数量至少是期望 Pod 数量的 70%。

Max Surge

`.spec.strategy.rollingUpdate.maxSurge` 是可选配置项，用来指定可以超过期望的 Pod 数量的最大个数。该值可以是一个绝对值（例如 5）或者是期望的 Pod 数量的百分比（例如 10%）。当 `MaxUnavailable` 为 0 时该值不可以为 0。通过百分比计算的绝对值向上取整。默认值是 1。

例如，该值设置成 30%，启动 rolling update 后新的 ReplicatSet 将会立即扩容，新老 Pod 的总数不能超过期望的 Pod 数量的 130%。旧的 Pod 被杀掉后，新的 ReplicaSet 将继续扩容，旧的 ReplicaSet 会进一步缩容，确保在升级的所有时刻所有的 Pod 数量和不会超过期望 Pod 数量的 130%。

Progress Deadline Seconds

`.spec.progressDeadlineSeconds` 是可选配置项，用来指定在系统报告 Deployment 的 failed progressing ——表现为 resource 的状态中 `type=Progressing`、`Status=False`、`Reason=ProgressDeadlineExceeded` 前可以等待的 Deployment 进行的秒数。Deployment controller 会继续重试该 Deployment。未来，在实现了自动回滚后， deployment controller 在观察到这种状态时就会自动回滚。

如果设置该参数，该值必须大于 `.spec.minReadySeconds`。

Min Ready Seconds

`.spec.minReadySeconds` 是一个可选配置项，用来指定没有任何容器 crash 的 Pod 并被认为是可用状态的最小秒数。默认是 0（Pod 在 ready 后就会被认为是可用状态）。进一步了解什么时候 Pod 会被认为是 ready 状态，参阅 Container Probes。

Rollback To

`.spec.rollbackTo` 是一个可以选配置项，用来配置 Deployment 回退的配置。设置该参数将触发回退操作，每次回退完成后，该值就会被清除。

Revision

`.spec.rollbackTo.revision` 是一个可选配置项，用来指定回退到的 revision。默认是 0，意味着回退到上一个 revision。

Revision History Limit

Deployment revision history 存储在它控制的 ReplicaSets 中。

`.spec.revisionHistoryLimit` 是一个可选配置项，用来指定可以保留的旧的 ReplicaSet 数量。该理想值取决于新 Deployment 的频率和稳定性。如果该值没有设置的话，默认所有旧的 Replicaset 或会被保留，将资源存储在 etcd 中，使用 `kubectl get rs` 查看输出。每个 Deployment 的该配置都保存在 ReplicaSet 中，然而，一旦你删除的旧的 RepelicaSet，你的 Deployment 就无法再回退到那个 revison 了。

如果你将该值设置为 0，所有具有 0 个 replica 的 ReplicaSet 都会被删除。在这种情况下，新的 Deployment rollout 无法撤销，因为 revision history 都被清理掉了。

Paused

`.spec.paused` 是可选配置项，boolean 值。用来指定暂停和恢复 Deployment。Paused 和非 paused 的 Deployment 之间的唯一区别就是，所有对 paused deployment 中的 PodTemplateSpec 的修改都不会触发新的 rollout。Deployment 被创建之后默认是非 paused。

Alternative to Deployments

kubectl rolling update

Kubectl rolling update 虽然使用类似的方式更新 Pod 和 ReplicationController。但是我们推荐使用 Deployment，因为它是声明式的，客户端侧，具有附加特性，例如即使滚动升级结束后也可以回滚到任何历史版本。

Ingress

在本篇文章中你将会看到一些在其他地方被交叉使用的术语，为了防止产生歧义，我们首先来澄清下。

- 节点：Kubernetes 集群中的服务器；
- 集群：Kubernetes 管理的一组服务器集合；
- 边界路由器：为局域网和 Internet 路由数据包的路由器，执行防火墙保护局域网络；
- 集群网络：遵循 Kubernetes网络模型 实现集群内的通信的具体实现，比如 flannel 和 OVS。
- 服务：Kubernetes 的服务 (Service) 是使用标签选择器标识的一组 pod Service。 除非另有说明，否则服务的虚拟 IP 仅可在集群内部访问。

什么是 Ingress？

通常情况下，service 和 pod 的 IP 仅可在集群内部访问。集群外部的请求需要通过负载均衡转发到 service 在 Node 上暴露的 NodePort 上，然后再由 kube-proxy 通过边缘路由器 (edge router) 将其转发给相关的 Pod 或者丢弃。如下图所示

```
 internet
      |
------------
[Services]
```

而 Ingress 就是为进入集群的请求提供路由规则的集合，如下图所示

```
    internet
        |
   [Ingress]
   --|-----|--
   [Services]
```

Ingress 可以给 service 提供集群外部访问的 URL、负载均衡、SSL 终止、HTTP 路由等。为了配置这些 Ingress 规则，集群管理员需要部署一个 Ingress controller，它监听 Ingress 和 service 的变化，并根据规则配置负载均衡并提供访问入口。

Ingress 格式

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

每个 Ingress 都需要配置 `rules`，目前 Kubernetes 仅支持 http 规则。上面的示例表示请求 `/testpath` 时转发到服务 `test` 的 80 端口。

API 版本对照表

Kubernetes 版本	Extension 版本
v1.5-v1.9	extensions/v1beta1

Ingress 类型

根据 Ingress Spec 配置的不同，Ingress 可以分为以下几种类型：

单服务 Ingress

单服务 Ingress 即该 Ingress 仅指定一个没有任何规则的后端服务。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

> 注：单个服务还可以通过设置 `Service.Type=NodePort` 或者 `Service.Type=LoadBalancer` 来对外暴露。

多服务的 Ingress

路由到多服务的 Ingress 即根据请求路径的不同转发到不同的后端服务上，比如

```
foo.bar.com -> 178.91.123.132 -> / foo    s1:80
                                 / bar    s2:80
```

可以通过下面的 Ingress 来定义：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

使用 `kubectl create -f` 创建完 ingress 后：

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -
          foo.bar.com
          /foo          s1:80
          /bar          s2:80
```

虚拟主机 Ingress

虚拟主机 Ingress 即根据名字的不同转发到不同的后端服务上，而他们共用同一个的 IP 地址，如下所示

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

下面是一个基于 Host header 路由请求的 Ingress：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

> 注：没有定义规则的后端服务称为默认后端服务，可以用来方便的处理 404 页面。

TLS Ingress

TLS Ingress 通过 Secret 获取 TLS 私钥和证书 (名为 `tls.crt` 和 `tls.key`)，来执行 TLS 终止。如果 Ingress 中的 TLS 配置部分指定了不同的主机，则它们将根据通过 SNI TLS 扩展指定的主机名（假如 Ingress controller 支持 SNI）在多个相同端口上进行复用。

定义一个包含 `tls.crt` 和 `tls.key` 的 secret：

```
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret
  namespace: default
type: Opaque
```

Ingress 中引用 secret：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
    - secretName: testsecret
  backend:
    serviceName: s1
    servicePort: 80
```

注意，不同 Ingress controller 支持的 TLS 功能不尽相同。 请参阅有关 nginx，GCE 或任何其他 Ingress controller 的文档，以了解 TLS 的支持情况。

更新 Ingress

可以通过 `kubectl edit ing name` 的方法来更新 ingress：

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
$ kubectl edit ing test
```

这会弹出一个包含已有 IngressSpec yaml 文件的编辑器，修改并保存就会将其更新到 kubernetes API server，进而触发 Ingress Controller 重新配置负载均衡：

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
        path: /foo
  - host: bar.baz.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
        path: /foo
..
```

更新后：

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
          bar.baz.com
          /foo          s2:80
```

当然，也可以通过 `kubectl replace -f new-ingress.yaml` 命令来更新，其中 new-ingress.yaml 是修改过的 Ingress yaml。

Ingress Controller

Ingress 正常工作需要集群中运行 Ingress Controller。Ingress Controller 与其他作为 kube-controller-manager 中的在集群创建时自动启动的 controller 成员不同，需要用户选择最适合自己集群的 Ingress Controller，或者自己实现一个。

Ingress Controller 以 Kubernetes Pod 的方式部署，以 daemon 方式运行，保持 watch Apiserver 的 /ingress 接口以更新 Ingress 资源，以满足 Ingress 的请求。比如可以使用 Nginx Ingress Controller：

```
helm install stable/nginx-ingress --name nginx-ingress --set rbac
```

其他 Ingress Controller 还有：

- traefik ingress 提供了一个 Traefik Ingress Controller 的实践案例
- kubernetes/ingress-nginx 提供了一个详细的 Nginx Ingress Controller 示例
- kubernetes/ingress-gce 提供了一个用于 GCE 的 Ingress Controller 示例

参考文档

- Kubernetes Ingress Resource
- Kubernetes Ingress Controller
- 使用 NGINX Plus 负载均衡 Kubernetes 服务
- 使用 NGINX 和 NGINX Plus 的 Ingress Controller 进行 Kubernetes 的负载均衡
- Kubernetes：Ingress Controller with Træfık and Let's Encrypt
- Kubernetes：Træfık and Let's Encrypt at scale
- Kubernetes Ingress Controller-Træfık
- Kubernetes 1.2 and simplifying advanced networking with Ingress

Job

Job 负责批量处理短暂的一次性任务 (short lived one-off tasks)，即仅执行一次的任务，它保证批处理任务的一个或多个 Pod 成功结束。

API 版本对照表

Kubernetes 版本	Batch API 版本	默认开启
v1.5+	batch/v1	是

Job 类型

Kubernetes 支持以下几种 Job：

- 非并行 Job：通常创建一个 Pod 直至其成功结束
- 固定结束次数的 Job：设置 `.spec.completions`，创建多个 Pod，直到 `.spec.completions` 个 Pod 成功结束
- 带有工作队列的并行 Job：设置 `.spec.Parallelism` 但不设置 `.spec.completions`，当所有 Pod 结束并且至少一个成功时，Job 就认为是成功

根据 `.spec.completions` 和 `.spec.Parallelism` 的设置，可以将 Job 划分为以下几种 pattern：

Job 类型	使用示例	行为	completions	Parallelism
一次性 Job	数据库迁移	创建一个 Pod 直至其成功结束	1	1
固定结束次数的 Job	处理工作队列的 Pod	依次创建一个 Pod 运行直至 completions 个成功结束	2+	1
固定结束次数的并行 Job	多个 Pod 同时处理工作队列	依次创建多个 Pod 运行直至 completions 个成功结束	2+	2+
并行 Job	多个 Pod 同时处理工作队列	创建一个或多个 Pod 直至有一个成功结束	1	2+

Job Controller

Job Controller 负责根据 Job Spec 创建 Pod，并持续监控 Pod 的状态，直至其成功结束。如果失败，则根据 restartPolicy（只支持 OnFailure 和 Never，不支持 Always）决定是否创建新的 Pod 再次重试任务。

Job Spec 格式

- spec.template 格式同 Pod
- RestartPolicy 仅支持 Never 或 OnFailure
- 单个 Pod 时，默认 Pod 成功运行后 Job 即结束
- `.spec.completions` 标志 Job 结束需要成功运行的 Pod 个数，默认为 1
- `.spec.parallelism` 标志并行运行的 Pod 的个数，默认为 1
- `spec.activeDeadlineSeconds` 标志失败 Pod 的重试最大时间，超过这个时间不会继续重试

一个简单的例子：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(200
      restartPolicy: Never
```

```
# 创建 Job
$ kubectl create -f ./job.yaml
job "pi" created
# 查看 Job 的状态
$ kubectl describe job pi
Name:        pi
Namespace:    default
Selector:    controller-uid=cd37a621-5b02-11e7-b56e-76933ddd7f55
Labels:        controller-uid=cd37a621-5b02-11e7-b56e-76933ddd7f5
        job-name=pi
Annotations:
Parallelism:    1
Completions:    1
Start Time:    Tue, 27 Jun 2017 14:35:24 +0800
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:    controller-uid=cd37a621-5b02-11e7-b56e-76933ddd7f55
        job-name=pi
  Containers:
   pi:
    Image:    perl
    Port:
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:
    Mounts:
  Volumes:
Events:
  FirstSeen    LastSeen    Count    From        SubObjectPath
  ---------    --------    -----    ----        -------------
  2m        2m        1    job-controller            Normal

# 使用'job-name=pi'标签查询属于该 Job 的 Pod
# 注意不要忘记'--show-all'选项显示已经成功（或失败）的 Pod
$ kubectl get pod --show-all -l job-name=pi
NAME       READY     STATUS      RESTARTS   AGE
pi-nltxv   0/1       Completed   0          3m
```

```
# 使用 jsonpath 获取 pod ID 并查看 Pod 的日志
$ pods=$(kubectl get pods --selector=job-name=pi --output=jsonpat
$ kubectl logs $pods
3.14159265358979323846264338327950 2...
```

固定结束次数的 Job 示例

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox
spec:
  completions: 3
  template:
    metadata:
      name: busybox
    spec:
      containers:
      - name: busybox
        image: busybox
        command: ["echo", "hello"]
      restartPolicy: Never
```

Bare Pods

所谓 Bare Pods 是指直接用 PodSpec 来创建的 Pod（即不在 ReplicaSets 或者 ReplicationCtroller 的管理之下的 Pods）。这些 Pod 在 Node 重启后不会自动重启，但 Job 则会创建新的 Pod 继续任务。所以，推荐使用 Job 来替代 Bare Pods，即便是应用只需要一个 Pod。

参考文档

- Jobs - Run to Completion

LocalVolume

> 注意：仅在 v1.7 + 中支持，并从 v1.10 开始升级为 beta 版本。

本地数据卷（Local Volume）代表一个本地存储设备，比如磁盘、分区或者目录等。主要的应用场景包括分布式存储和数据库等需要高性能和高可靠性的环境里。本地数据卷同时支持块设备和文件系统，通过 `spec.local.path` 指定；但对于文件系统来说，kubernetes 并不会限制该目录可以使用的存储空间大小。

本地数据卷只能以静态创建的 PV 使用。相对于 HostPath，本地数据卷可以直接以持久化的方式使用（它总是通过 NodeAffinity 调度在某个指定的节点上）。

另外，社区还提供了一个 local-volume-provisioner，用于自动创建和清理本地数据卷。

示例

StorageClass

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

创建一个调度到 hostname 为 `example-node` 的本地数据卷：

```
# For kubernetes v1.10
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-local-pv
spec:
  capacity:
    storage: 100Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

```
# For kubernetes v1.7-1.9
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-local-pv
  annotations:
    "volume.alpha.kubernetes.io/node-affinity": '{
      "requiredDuringSchedulingIgnoredDuringExecution": {
        "nodeSelectorTerms": [
          { "matchExpressions": [
            { "key": "kubernetes.io/hostname",
              "operator": "In",
              "values": ["example-node"]
            }
          ]}
         ]}
        }',
spec:
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
```

创建 PVC：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: example-local-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: local-storage
```

创建 Pod，引用 PVC：

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: example-local-claim
```

限制

- 暂不支持一个 Pod 绑定多个本地数据卷的 PVC（计划 v1.9 支持）
- 有可能导致调度冲突，比如 CPU 或者内存资源不足（计划 v1.9 增强）
- 外部 Provisoner 在启动后无法正确检测挂载点的空间大小（需要 Mount Propagation，计划 v1.9 支持）

最佳实践

- 推荐为每个存储卷分配独立的磁盘，以便隔离 IO 请求
- 推荐为每个存储卷分配独立的分区，以便隔离存储空间
- 避免重新创建同名的 Node，否则会导致新 Node 无法识别已绑定旧 Node 的 PV
- 推荐使用 UUID 而不是文件路径，以避免文件路径误配的问题
- 对于不带文件系统的块存储，推荐使用唯一 ID（如 `/dev/disk/by-id/`），以避免块设备路径误配的问题

参考文档

- Local Persistent Storage User Guide

Namespace

Namespace 是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。常见的 pod, service, replication controller 和 deployment 等都是属于某一个 namespace 的（默认是 default），而 node, persistent volume，namespace 等资源则不属于任何 namespace。

Namespace 常用来隔离不同的用户，比如 Kubernetes 自带的服务一般运行在 `kube-system` namespace 中。

Namespace 操作

> `kubectl` 可以通过 `--namespace` 或者 `-n` 选项指定 namespace。如果不指定，默认为 default。查看操作下, 也可以通过设置 --all-namespace=true 来查看所有 namespace 下的资源。

查询

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    11d
kube-system   Active    11d
```

注意：namespace 包含两种状态 "Active" 和 "Terminating"。在 namespace 删除过程中，namespace 状态被设置成 "Terminating"。

创建

```
(1) 命令行直接创建
$ kubectl create namespace new-namespace

(2) 通过文件创建
$ cat my-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: new-namespace

$ kubectl create -f ./my-namespace.yaml
```

注意：命名空间名称满足正则表达式 `[a-z0-9]([-a-z0-9]*[a-z0-9])?`，最大长度为 63 位

删除

```
$ kubectl delete namespaces new-namespace
```

注意：

1. 删除一个 namespace 会自动删除所有属于该 namespace 的资源。
2. `default` 和 `kube-system` 命名空间不可删除。
3. PersistentVolume 是不属于任何 namespace 的，但 PersistentVolumeClaim 是属于某个特定 namespace 的。
4. Event 是否属于 namespace 取决于产生 event 的对象。
5. v1.7 版本增加了 `kube-public` 命名空间，该命名空间用来存放公共的信息，一般以 ConfigMap 的形式存放。

```
$ kubectl get configmap  -n=kube-public
NAME           DATA      AGE
cluster-info   2         29d
```

参考文档

- Kubernetes Namespace
- Share a Cluster with Namespaces

NetworkPolicy

随着微服务的流行，越来越多的云服务平台需要大量模块之间的网络调用。Kubernetes 在 1.3 引入了 Network Policy，Network Policy 提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。

在使用 Network Policy 时，需要注意

- v1.6 以及以前的版本需要在 kube-apiserver 中开启 `extensions/v1beta1/networkpolicies`
- v1.7 版本 Network Policy 已经 GA，API 版本为 `networking.k8s.io/v1`
- v1.8 版本新增 **Egress** 和 **IPBlock** 的支持
- 网络插件要支持 Network Policy，如 Calico、Romana、Weave Net 和 trireme 等，参考 这里

API 版本对照表

Kubernetes 版本	Networking API 版本
v1.5-v1.6	extensions/v1beta1
v1.7+	networking.k8s.io/v1

网络策略

Namespace 隔离

默认情况下，所有 Pod 之间是全通的。每个 Namespace 可以配置独立的网络策略，来隔离 Pod 之间的流量。

v1.7 + 版本通过创建匹配所有 Pod 的 Network Policy 来作为默认的网络策略，比如默认拒绝所有 Pod 之间 Ingress 通信

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

默认拒绝所有 Pod 之间 Egress 通信的策略为

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

甚至是默认拒绝所有 Pod 之间 Ingress 和 Egress 通信的策略为

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

而默认允许所有 Pod 之间 Ingress 通信的策略为

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  ingress:
  - {}
```

默认允许所有 Pod 之间 Egress 通信的策略为

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  egress:
  - {}
```

而 v1.6 版本则通过 Annotation 来隔离 namespace 的所有 Pod 之间的流量，包括从外部到该 namespace 中所有 Pod 的流量以及 namespace 内部 Pod 相互之间的流量：

```
kubectl annotate ns  "net.beta.kubernetes.io/network-policy={\"ir
```

Pod 隔离

通过使用标签选择器（包括 namespaceSelector 和 podSelector）来控制 Pod 之间的流量。比如下面的 Network Policy

- 允许 default namespace 中带有 `role=frontend` 标签的 Pod 访问 default namespace 中带有 `role=db` 标签 Pod 的 6379 端口
- 允许带有 `project=myprojects` 标签的 namespace 中所有 Pod 访问 default namespace 中带有 `role=db` 标签 Pod 的 6379 端口

```
# v1.6 以及更老的版本应该使用 extensions/v1beta1
# apiVersion: extensions/v1beta1
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: tcp
      port: 6379
```

另外一个同时开启 Ingress 和 Egress 通信的策略为

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

它用来隔离 default namespace 中带有 `role=db` 标签的 Pod：

- 允许 default namespace 中带有 `role=frontend` 标签的 Pod 访问 default namespace 中带有 `role=db` 标签 Pod 的 6379 端口

- 允许带有 `project=myprojects` 标签的 namespace 中所有 Pod 访问 default namespace 中带有 `role=db` 标签 Pod 的 6379 端口
- 允许 default namespace 中带有 `role=db` 标签的 Pod 访问 `10.0.0.0/24` 网段的 TCP 5987 端口

简单示例

以 calico 为例看一下 Network Policy 的具体用法。

首先配置 kubelet 使用 CNI 网络插件

```
kubelet --network-plugin=cni --cni-conf-dir=/etc/cni/net.d --cni-
```

安装 calio 网络插件

```
# 注意修改 CIDR，需要跟 k8s pod-network-cidr 一致，默认为 192.168.0.0/1
kubectl apply -f https://docs.projectcalico.org/v3.0/getting-star
```

首先部署一个 nginx 服务

```
$ kubectl run nginx --image=nginx --replicas=2
deployment "nginx" created
$ kubectl expose deployment nginx --port=80
service "nginx" exposed
```

此时，通过其他 Pod 是可以访问 nginx 服务的

```
$ kubectl get svc,pod
NAME                        CLUSTER-IP    EXTERNAL-IP   PORT(S)
svc/kubernetes              10.100.0.1            443/TCP    46m
svc/nginx                   10.100.0.16           80/TCP     33s

NAME                        READY         STATUS        RESTARTS
po/nginx-701339712-e0qfq    1/1           Running       0
po/nginx-701339712-o00ef    1/1           Running       0

$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, st

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

开启 default namespace 的 DefaultDeny Network Policy 后，其他 Pod（包括 namespace 外部）不能访问 nginx 了：

```
$ cat default-deny.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress

$ kubectl create -f default-deny.yaml

$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, st

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
wget: download timed out
/ #
```

最后再创建一个运行带有 `access=true` 的 Pod 访问的网络策略

```
$ cat nginx-policy.yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      run: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"

$ kubectl create -f nginx-policy.yaml
networkpolicy "access-nginx" created

# 不带 access=true 标签的 Pod 还是无法访问 nginx 服务
$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, st

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
wget: download timed out
/ #


# 而带有 access=true 标签的 Pod 可以访问 nginx 服务
$ kubectl run busybox --rm -ti --labels="access=true" --image=bus
Waiting for pod default/busybox-472357175-y0m47 to be running, st

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

最后开启 nginx 服务的外部访问：

```
$ cat nginx-external-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: front-end-access
  namespace: sock-shop
spec:
  podSelector:
    matchLabels:
      run: nginx
  ingress:
    - ports:
        - protocol: TCP
          port: 80

$ kubectl create -f nginx-external-policy.yaml
```

使用场景

禁止访问指定服务

```
kubectl run web --image=nginx --labels app=web,env=prod --expose
```

网络策略

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
      env: prod
```

只允许指定 Pod 访问服务

```
kubectl run apiserver --image=nginx --labels app=bookstore,role=a
```

网络策略

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: bookstore
```

禁止 namespace 中所有 Pod 之间的相互访问

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: default
spec:
  podSelector: {}
```

禁止其他 namespace 访问服务

```
kubectl create namespace secondary
kubectl run web --namespace secondary --image=nginx \
    --labels=app=web --expose --port 80
```

网络策略

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: secondary
  name: web-deny-other-namespaces
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
```

只允许指定 namespace 访问服务

```
kubectl run web --image=nginx \
    --labels=app=web --expose --port 80
```

网络策略

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-prod
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production
```

允许外网访问服务

```
kubectl run web --image=nginx --labels=app=web --port 80
kubectl expose deployment/web --type=LoadBalancer
```

网络策略

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-external
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - ports:
    - port: 80
    from: []
```

参考文档

- Kubernetes network policies

- Declare Network Policy
- Securing Kubernetes Cluster Networking
- Kubernetes Network Policy Recipes

Node

Node 是 Pod 真正运行的主机，可以是物理机，也可以是虚拟机。为了管理 Pod，每个 Node 节点上至少要运行 container runtime（比如 `docker` 或者 `rkt`）、`kubelet` 和 `kube-proxy` 服务。

node

Node 管理

不像其他的资源（如 Pod 和 Namespace），Node 本质上不是 Kubernetes 来创建的，Kubernetes 只是管理 Node 上的资源。虽然可以通过 Manifest 创建一个 Node 对象（如下 yaml 所示），但 Kubernetes 也只是去检查是否真的是有这么一个 Node，如果检查失败，也不会往上调度 Pod。

```
kind: Node
apiVersion: v1
metadata:
  name: 10-240-79-157
  labels:
    name: my-first-k8s-node
```

这个检查是由 Node Controller 来完成的。Node Controller 负责

- 维护 Node 状态
- 与 Cloud Provider 同步 Node
- 给 Node 分配容器 CIDR
- 删除带有 `NoExecute` taint 的 Node 上的 Pods

默认情况下，kubelet 在启动时会向 master 注册自己，并创建 Node 资源。

Node 的状态

每个 Node 都包括以下状态信息：

- 地址：包括 hostname、外网 IP 和内网 IP

- 条件（Condition）：包括 OutOfDisk、Ready、MemoryPressure 和 DiskPressure
- 容量（Capacity）：Node 上的可用资源，包括 CPU、内存和 Pod 总数
- 基本信息（Info）：包括内核版本、容器引擎版本、OS 类型等

Taints 和 tolerations

Taints 和 tolerations 用于保证 Pod 不被调度到不合适的 Node 上，Taint 应用于 Node 上，而 toleration 则应用于 Pod 上（Toleration 是可选的）。

比如，可以使用 taint 命令给 node1 添加 taints：

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value2:NoExecute
```

Taints 和 tolerations 的具体使用方法请参考 调度器章节。

Node 维护模式

标志 Node 不可调度但不影响其上正在运行的 Pod，这种维护 Node 时是非常有用的

```
kubectl cordon $NODENAME
```

参考文档

- Kubernetes Node
- Taints 和 tolerations

PersistentVolume

PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 提供了方便的持久化卷：PV 提供网络存储资源，而 PVC 请求存储资源。这样，设置持久化的工作流包括配置底层文件系统或者云数据卷、创建持久性数据卷、最后创建 PVC 来将 Pod 跟数据卷关联起来。PV 和 PVC 可以将 pod 和数据卷解耦，pod 不需要知道确切的文件系统或者支持它的持久化引擎。

Volume 生命周期

Volume 的生命周期包括 5 个阶段

1. Provisioning，即 PV 的创建，可以直接创建 PV（静态方式），也可以使用 StorageClass 动态创建
2. Binding，将 PV 分配给 PVC
3. Using，Pod 通过 PVC 使用该 Volume，并可以通过准入控制 StorageObjectInUseProtection（1.9 及以前版本为 PVCProtection）阻止删除正在使用的 PVC
4. Releasing，Pod 释放 Volume 并删除 PVC
5. Reclaiming，回收 PV，可以保留 PV 以便下次使用，也可以直接从云存储中删除
6. Deleting，删除 PV 并从云存储中删除后段存储

根据这 5 个阶段，Volume 的状态有以下 4 种

- Available：可用
- Bound：已经分配给 PVC
- Released：PVC 解绑但还未执行回收策略
- Failed：发生错误

API 版本对照表

Kubernetes 版本	PV/PVC 版本	StorageClass 版本
v1.5-v1.6	core/v1	storage.k8s.io/v1beta1
v1.7+	core/v1	storage.k8s.io/v1

PV

PersistentVolume（PV）是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。PV 跟 Volume (卷) 类似，不过会有独立于 Pod 的生命周期。比如一个 NFS 的 PV 可以定义为

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

PV 的访问模式（accessModes）有三种：

- ReadWriteOnce（RWO）：是最基本的方式，可读可写，但只支持被单个节点挂载。
- ReadOnlyMany（ROX）：可以以只读的方式被多个节点挂载。
- ReadWriteMany（RWX）：这种存储可以以读写的方式被多个节点共享。不是每一种存储都支持这三种方式，像共享方式，目前支持的还比较少，比较常用的是 NFS。在 PVC 绑定 PV 时通常根据两个条件来绑定，一个是存储的大小，另一个就是访问模式。

PV 的回收策略（persistentVolumeReclaimPolicy，即 PVC 释放卷的时候 PV 该如何操作）也有三种

- Retain，不清理, 保留 Volume（需要手动清理）
- Recycle，删除数据，即 `rm -rf /thevolume/*`（只有 NFS 和 HostPath 支持）
- Delete，删除存储资源，比如删除 AWS EBS 卷（只有 AWS EBS, GCE PD, Azure Disk 和 Cinder 支持）

StorageClass

上面通过手动的方式创建了一个 NFS Volume，这在管理很多 Volume 的时候不太方便。Kubernetes 还提供了 StorageClass 来动态创建 PV，不仅节省了管理员的时间，还可以封装不同类型的存储供 PVC 选用。

StorageClass 包括四个部分

- provisioner：指定 Volume 插件的类型，包括内置插件（如 `kubernetes.io/glusterfs`）和外部插件（如 external-storage 提供的 `ceph.com/cephfs`）。
- mountOptions：指定挂载选项，当 PV 不支持指定的选项时会直接失败。比如 NFS 支持 `hard` 和 `nfsvers=4.1` 等选项。
- parameters：指定 provisioner 的选项，比如 `kubernetes.io/aws-ebs` 支持 `type`、`zone`、`iopsPerGB` 等参数。
- reclaimPolicy：指定回收策略，同 PV 的回收策略。

在使用 PVC 时，可以通过 `DefaultStorageClass` 准入控制设置默认 StorageClass, 即给未设置 storageClassName 的 PVC 自动添加默认的 StorageClass。而默认的 StorageClass 带有 annotation `storageclass.kubernetes.io/is-default-class=true`。

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE
Glusterfs	✓	Glusterfs
iSCSI	-	-
PhotonPersistentDisk	✓	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

修改默认 StorageClass

取消原来的默认 StorageClass

```
kubectl patch storageclass  -p '{"metadata": {"annotations":{"sto
```

标记新的默认 StorageClass

```
kubectl patch storageclass  -p '{"metadata": {"annotations":{"stc
```

GCE 示例

> 单个 GCE 节点最大支持挂载 16 个 Google Persistent Disk。开启 `AttachVolumeLimit` 特性后，根据节点的类型最大可以挂载 128 个。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zone: us-central1-a
```

Glusterfs 示例

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

OpenStack Cinder 示例

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  type: fast
  availability: nova
```

Ceph RBD 示例

```
apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    name: fast
  provisioner: kubernetes.io/rbd
  parameters:
    monitors: 10.16.153.105:6789
    adminId: kube
    adminSecretName: ceph-secret
    adminSecretNamespace: kube-system
    pool: kube
    userId: kube
    userSecretName: ceph-secret-user
```

Local Volume

Local Volume 允许将 Node 本地的磁盘、分区或者目录作为持久化存储使用。注意，Local Volume 不支持动态创建，使用前需要预先创建好 PV。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  # volumeMode field requires BlockVolume Alpha feature gate to b
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

推荐配置

- 对于需要强 IO 隔离的场景，推荐使用整块磁盘作为 Volume
- 对于需要容量隔离的场景，推荐使用分区作为 Volume
- 避免在集群中重新创建同名的 Node（无法避免时需要先删除通过 Affinity 引用该 Node 的 PV）
- 对于文件系统类型的本地存储，推荐使用 UUID（如 `ls -l /dev/disk/by-uuid`）作为系统挂载点

- 对于无文件系统的块存储，推荐生成一个唯一 ID 作软链接 (如 `/dev/dis/by-id`)。这可以保证 Volume 名字唯一，并不会与其他 Node 上面的同名 Volume 混淆

PVC

PV 是存储资源，而 PersistentVolumeClaim (PVC) 是对 PV 的请求。PVC 跟 Pod 类似：Pod 消费 Node 资源，而 PVC 消费 PV 资源；Pod 能够请求 CPU 和内存资源，而 PVC 请求特定大小和访问模式的数据卷。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

PVC 可以直接挂载到 Pod 中：

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

扩展 PV 空间

> ExpandPersistentVolumes 在 v1.8 开始 Alpha，v1.11 升级为 Beta 版。

v1.8 开始支持扩展 PV 空间，支持在不丢失数据和重启容器的情况下扩展 PV 的大小。注意， **当前的实现仅支持不需要调整文件系统大小（XFS、Ext3、Ext4）的 PV，并且只支持以下几种存储插件** ：

- AzureDisk
- AzureFile
- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd
- Portworx

开启扩展 PV 空间的功能需要配置

- 开启 `ExpandPersistentVolumes` 功能，即配置 `--feature-gates=ExpandPersistentVolumes=true`
- 开启准入控制插件 `PersistentVolumeClaimResize` ，它只允许扩展明确配置 `allowVolumeExpansion=true` 的 StorageClass，比如

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

这样，用户就可以修改 PVC 中请求存储的大小（如通过 `kubectl edit` 命令）请求更大的存储空间。

块存储（Raw Block Volume）

Kubernetes v1.9 新增了 Alpha 版的 Raw Block Volume，可通过设置 `volumeMode: Block`（可选项为 `Filesystem` 和 `Block`）来使用块存储。

> 注意：使用前需要为 kube-apiserver、kube-controller-manager 和 kubelet 开启 `BlockVolume` 特性，即添加命令行选项 `--feature-gates=BlockVolume=true,...`。

支持块存储的 PV 插件包括

- Local Volume
- fc
- iSCSI
- Ceph RBD
- AWS EBS
- GCE PD
- AzureDisk
- Cinder

使用示例

```
# Persistent Volumes using a Raw Block Volume
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
---
# Persistent Volume Claim requesting a Raw Block Volume
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
---
# Pod specification adding Raw Block Device path in container
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
  annotations:
    # apparmor should be unconfied for mounting the device inside
    container.apparmor.security.beta.kubernetes.io/fc-container:
spec:
  containers:
```

```
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: ["tail -f /dev/null"]
      securityContext:
        capabilities:
          # CAP_SYS_ADMIN is required for mount() syscall.
          add: ["SYS_ADMIN"]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

StorageObjectInUseProtection

> 准入控制 StorageObjectInUseProtection 在 v1.11 版本 GA。

当开启准入控制 StorageObjectInUseProtection（`--admission-control=StorageObjectInUseProtection`）时，删除使用中的 PV 和 PVC 后，它们会等待使用者删除后才删除（而不是之前的立即删除）。而在使用者删除之前，它们会一直处于 Terminating 状态。

拓扑感知动态调度

拓扑感知动态存储卷调度（topology-aware dynamic provisioning）是 v1.12 版本的一个 Beta 特性，用来支持在多可用区集群中动态创建和调度持久化存储卷。目前的实现支持以下几种存储：

- AWS EBS
- Azure Disk
- GCE PD (including Regional PD)
- CSI (alpha) - currently only the GCE PD CSI driver has implemented topology support

使用示例

```
# set WaitForFirstConsumer in storage class
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: topology-aware-standard
provisioner: kubernetes.io/gce-pd
volumeBindingMode: WaitForFirstConsumer
parameters:
  type: pd-standard

# Refer storage class
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: failure-domain.beta.kubernetes.io/zone
                operator: In
                values:
                - us-central1-a
                - us-central1-f
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
```

```
              - key: app
                operator: In
                values:
                - nginx
            topologyKey: failure-domain.beta.kubernetes.io/zone
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
        - name: logs
          mountPath: /logs
  volumeClaimTemplates:
   - metadata:
       name: www
     spec:
       accessModes: [ "ReadWriteOnce" ]
       storageClassName: topology-aware-standard
       resources:
         requests:
           storage: 10Gi
   - metadata:
       name: logs
     spec:
       accessModes: [ "ReadWriteOnce" ]
       storageClassName: topology-aware-standard
       resources:
         requests:
           storage: 1Gi
```

然后查看 PV，可以发现它们创建在不同的可用区内

```
$ kubectl get pv -o=jsonpath='{range .items[*]}{.spec.claimRef.na
www-web-0       us-central1-f
logs-web-0      us-central1-f
www-web-1       us-central1-a
logs-web-1      us-central1-a
```

存储快照

存储快照是 v1.12 新增的 Alpha 特性，用来支持给存储卷创建快照。支持的插件包括

- GCE Persistent Disk CSI Driver
- OpenSDS CSI Driver
- Ceph RBD CSI Driver
- Portworx CSI Driver

image-20181014215558480

在使用前需要开启特性开关 VolumeSnapshotDataSource。

使用示例：

```
# create snapshot
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-demo
  namespace: demo-namespace
spec:
  snapshotClassName: csi-snapclass
  source:
    name: mypvc
    kind: PersistentVolumeClaim

# import from snapshot
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshotContent
metadata:
  name: static-snapshot-content
spec:
  csiVolumeSnapshotSource:
    driver: com.example.csi-driver
    snapshotHandle: snapshotcontent-example-id
  volumeSnapshotRef:
    kind: VolumeSnapshot
    name: static-snapshot-demo
    namespace: demo-namespace

# provision volume from snapshot
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-restore
  Namespace: demo-namespace
spec:
  storageClassName: csi-storageclass
  dataSource:
    name: new-snapshot-demo
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
```

```
  resources:
    requests:
      storage: 1Gi
```

参考文档

- Kubernetes Persistent Volumes
- Kubernetes Storage Classes
- Dynamic Volume Provisioning
- Kubernetes CSI Documentation
- Volume Snapshots Documentation

Pod

Pod 是一组紧密关联的容器集合，它们共享 IPC、Network 和 UTS namespace，是 Kubernetes 调度的基本单位。Pod 的设计理念是支持多个容器在一个 Pod 中共享网络和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。

pod

Pod 的特征

- 包含多个共享 IPC、Network 和 UTC namespace 的容器，可直接通过 localhost 通信
- 所有 Pod 内容器都可以访问共享的 Volume，可以访问共享数据
- 无容错性：直接创建的 Pod 一旦被调度后就跟 Node 绑定，即使 Node 挂掉也不会被重新调度（而是被自动删除），因此推荐使用 Deployment、Daemonset 等控制器来容错
- 优雅终止：Pod 删除的时候先给其内的进程发送 SIGTERM，等待一段时间（grace period）后才强制停止依然还在运行的进程
- 特权容器（通过 SecurityContext 配置）具有改变系统配置的权限（在网络插件中大量应用）

> Kubernetes v1.8+ 还支持容器间共享 PID namespace，需要 docker >= 1.13.1，并配置 kubelet `--docker-disable-shared-pid=false`。

API 版本对照表

Kubernetes 版本	Core API 版本	默认开启
v1.5+	core/v1	是

Pod 定义

通过 yaml 或 json 描述 Pod 和其内容器的运行环境以及期望状态，比如一个最简单的 nginx pod 可以定义为

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

> 在生产环境中，推荐使用 Deployment、StatefulSet、Job 或者 CronJob 等控制器来创建 Pod，而不推荐直接创建 Pod。

Docker 镜像支持

目前，Kubernetes 仅支持使用 Docker 镜像来创建容器，但并非支持 Dockerfile 定义的所有行为。如下表所示

Dockerfile 指令	描述	支持	说明
ENTRYPOINT	启动命令	是	containerSpec.command
CMD	命令的参数列表	是	containerSpec.args
ENV	环境变量	是	containerSpec.env
EXPOSE	对外开放的端口	否	使用 containerSpec.ports.containerPort 替代
VOLUME	数据卷	是	使用 volumes 和 volumeMounts
USER	进程运行用户以及用户组	是	securityContext.runAsUser/ supplementalGroups
WORKDIR	工作目录	是	containerSpec.workingDir
STOPSIGNAL	停止容器时给进程发送的信号	是	SIGKILL
HEALTHCHECK	健康检查	否	使用 livenessProbe 和 readinessProbe 替代
SHELL	运行启动命令的 SHELL	否	使用镜像默认 SHELL 启动命令

Pod 生命周期

Kubernetes 以 `PodStatus.Phase` 抽象 Pod 的状态（但并不直接反映所有容器的状态）。可能的 Phase 包括

- Pending: Pod 已经在 apiserver 中创建，但还没有调度到 Node 上面
- Running: Pod 已经调度到 Node 上面，所有容器都已经创建，并且至少有一个容器还在运行或者正在启动
- Succeeded: Pod 调度到 Node 上面后成功运行结束，并且不会重启
- Failed: Pod 调度到 Node 上面后至少有一个容器运行失败（即退出码不为 0 或者被系统终止）
- Unknonwn: 状态未知，通常是由于 apiserver 无法与 kubelet 通信导致

可以用 kubectl 命令查询 Pod Phase：

```
$ kubectl get pod reviews-v1-5bdc544bbd-5qgxj -o jsonpath="{.stat
Running
```

PodSpec 中的 `restartPolicy` 可以用来设置是否对退出的 Pod 重启，可选项包括 `Always`、`OnFailure`、以及 `Never`。比如

- 单容器的 Pod，容器成功退出时，不同 `restartPolicy` 时的动作为
  - Always: 重启 Container; Pod `phase` 保持 Running.
  - OnFailure: Pod `phase` 变成 Succeeded.
  - Never: Pod `phase` 变成 Succeeded.
- 单容器的 Pod，容器失败退出时，不同 `restartPolicy` 时的动作为
  - Always: 重启 Container; Pod `phase` 保持 Running.
  - OnFailure: 重启 Container; Pod `phase` 保持 Running.
  - Never: Pod `phase` 变成 Failed.
- 2个容器的 Pod，其中一个容器在运行而另一个失败退出时，不同 `restartPolicy` 时的动作为
  - Always: 重启 Container; Pod `phase` 保持 Running.
  - OnFailure: 重启 Container; Pod `phase` 保持 Running.
  - Never: 不重启 Container; Pod `phase` 保持 Running.
- 2个容器的 Pod，其中一个容器停止而另一个失败退出时，不同 `restartPolicy` 时的动作为
  - Always: 重启 Container; Pod `phase` 保持 Running.
  - OnFailure: 重启 Container; Pod `phase` 保持 Running.
  - Never: Pod `phase` 变成 Failed.

- 单容器的 Pod，容器内存不足（OOM），不同 `restartPolicy` 时的动作为
  - Always: 重启 Container; Pod `phase` 保持 Running.
  - OnFailure: 重启 Container; Pod `phase` 保持 Running.
  - Never: 记录失败事件; Pod `phase` 变成 Failed.
- Pod 还在运行，但磁盘不可访问时
  - 终止所有容器
  - Pod `phase` 变成 Failed
  - 如果 Pod 是由某个控制器管理的，则重新创建一个 Pod 并调度到其他 Node 运行
- Pod 还在运行，但由于网络分区故障导致 Node 无法访问
  - Node controller等待 Node 事件超时
  - Node controller 将 Pod `phase` 设置为 Failed.
  - 如果 Pod 是由某个控制器管理的，则重新创建一个 Pod 并调度到其他 Node 运行

使用 Volume

Volume 可以为容器提供持久化存储，比如

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

更多挂载存储卷的方法参考 Volume。

私有镜像

在使用私有镜像时，需要创建一个 docker registry secret，并在容器中引用。

创建 docker registry secret：

```
kubectl create secret docker-registry regsecret --docker-server=
```

比如使用 Azure Container Registry（ACR）：

```
ACR_NAME=dregistry
SERVICE_PRINCIPAL_NAME=acr-service-principal

# Populate the ACR login server and resource id.
ACR_LOGIN_SERVER=$(az acr show --name $ACR_NAME --query loginServ
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --outpu

# Create a contributor role assignment with a scope of the ACR re
SP_PASSWD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NA

# Get the service principle client id.
CLIENT_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --c

# Create secret
kubectl create secret docker-registry acr-auth --docker-server $A
```

容器中引用该 secret：

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
    - name: private-reg-container
      image: dregistry.azurecr.io/acr-auth-example
  imagePullSecrets:
    - name: acr-auth
```

RestartPolicy

支持三种 RestartPolicy

- Always：只要退出就重启
- OnFailure：失败退出（exit code 不等于 0）时重启
- Never：只要退出就不再重启

注意，这里的重启是指在 Pod 所在 Node 上面本地重启，并不会调度到其他 Node 上去。

环境变量

环境变量为容器提供了一些重要的资源，包括容器和 Pod 的基本信息以及集群中服务的信息等：

(1) hostname

`HOSTNAME` 环境变量保存了该 Pod 的 hostname。

（2）容器和 Pod 的基本信息

Pod 的名字、命名空间、IP 以及容器的计算资源限制等可以以 Downward API 的方式获取并存储到环境变量中。

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c"]
      args:
      - env
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: MY_POD_SERVICE_ACCOUNT
          valueFrom:
            fieldRef:
              fieldPath: spec.serviceAccountName
        - name: MY_CPU_REQUEST
```

```
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.memory
  restartPolicy: Never
```

(3) 集群中服务的信息

容器的环境变量中还可以引用容器运行前创建的所有服务的信息，比如默认的 kubernetes 服务对应以下环境变量：

```
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.1
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
```

由于环境变量存在创建顺序的局限性（环境变量中不包含后来创建的服务），推荐使用 DNS 来解析服务。

镜像拉取策略

支持三种 ImagePullPolicy

- Always：不管镜像是否存在都会进行一次拉取
- Never：不管镜像是否存在都不会进行拉取
- IfNotPresent：只有镜像不存在时，才会进行镜像拉取

注意：

- 默认为 `IfNotPresent`，但 `:latest` 标签的镜像默认为 `Always`。
- 拉取镜像时 docker 会进行校验，如果镜像中的 MD5 码没有变，则不会拉取镜像数据。
- 生产环境中应该尽量避免使用 `:latest` 标签，而开发环境中可以借助 `:latest` 标签自动拉取最新的镜像。

访问 DNS 的策略

通过设置 dnsPolicy 参数，设置 Pod 中容器访问 DNS 的策略

- ClusterFirst：优先基于 cluster domain（如 `default.svc.cluster.local`）后缀，通过 kube-dns 查询 (默认策略)
- Default：优先从 Node 中配置的 DNS 查询

使用主机的 IPC 命名空间

通过设置 `spec.hostIPC` 参数为 true，使用主机的 IPC 命名空间，默认为 false。

使用主机的网络命名空间

通过设置 `spec.hostNetwork` 参数为 true，使用主机的网络命名空间，默认为 false。

使用主机的 PID 空间

通过设置 `spec.hostPID` 参数为 true，使用主机的 PID 命名空间，默认为 false。

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostIPC: true
  hostPID: true
  hostNetwork: true
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

设置 Pod 的 hostname

通过 `spec.hostname` 参数实现，如果未设置默认使用 `metadata.name` 参数的值作为 Pod 的 hostname。

设置 Pod 的子域名

通过 `spec.subdomain` 参数设置 Pod 的子域名，默认为空。

比如，指定 hostname 为 busybox-2 和 subdomain 为 default-subdomain，完整域名为 `busybox-2.default-subdomain.default.svc.cluster.local`，也可以简写为 `busybox-2.default-subdomain.default`：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

注意：

- 默认情况下，DNS 为 Pod 生成的 A 记录格式为 `pod-ip-address.my-namespace.pod.cluster.local`，如 `1-2-3-4.default.pod.cluster.local`
- 上面的示例还需要在 default namespace 中创建一个名为 `default-subdomain`（即 subdomain）的 headless service，否则其他 Pod 无法通过完整域名访问到该 Pod（只能自己访问到自己）

```
kind: Service
apiVersion: v1
metadata:
  name: default-subdomain
spec:
  clusterIP: None
  selector:
    name: busybox
  ports:
  - name: foo # Actually, no port is needed.
    port: 1234
    targetPort: 1234
```

注意，必须为 headless service 设置至少一个服务端口（`spec.ports`，即便它看起来并不需要），否则 Pod 与 Pod 之间依然无法通过完整域名来访问。

设置 Pod 的 DNS 选项

从 v1.9 开始，可以在 kubelet 和 kube-apiserver 中设置 `--feature-gates=CustomPodDNS=true` 开启设置每个 Pod DNS 地址的功能。

> 注意该功能在 v1.10 中为 Beta 版，v1.9 中为 Alpha 版。

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

对于旧版本的集群，可以使用 ConfigMap 来自定义 Pod 的 `/etc/resolv.conf`，如

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: resolvconf
  namespace: default
data:
  resolv.conf: |
    search default.svc.cluster.local svc.cluster.local cluster.lo
    nameserver 10.0.0.10

---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: dns-test
  namespace: default
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: dns-test
    spec:
      containers:
        - name: dns-test
          image: alpine
          stdin: true
          tty: true
          command: ["sh"]
          volumeMounts:
            - name: resolv-conf
              mountPath: /etc/resolv.conf
              subPath: resolv.conf
      volumes:
        - name: resolv-conf
          configMap:
            name: resolvconf
            items:
            - key: resolv.conf
              path: resolv.conf
```

资源限制

Kubernetes 通过 cgroups 限制容器的 CPU 和内存等计算资源，包括 requests（请求，**调度器保证调度到资源充足的 Node 上，如果无法满足会调度失败**）和 limits（上限）等：

- `spec.containers[].resources.limits.cpu`：CPU 上限，可以短暂超过，容器也不会被停止
- `spec.containers[].resources.limits.memory`：内存上限，不可以超过；如果超过，容器可能会被终止或调度到其他资源充足的机器上
- `spec.containers[].resources.limits.ephemeral-storage`：临时存储（容器可写层、日志以及 EmptyDir 等）的上限，超过后 Pod 会被驱逐
- `spec.containers[].resources.requests.cpu`：CPU 请求，也是调度 CPU 资源的依据，可以超过
- `spec.containers[].resources.requests.memory`：内存请求，也是调度内存资源的依据，可以超过；但如果超过，容器可能会在 Node 内存不足时清理
- `spec.containers[].resources.requests.ephemeral-storage`：临时存储（容器可写层、日志以及 EmptyDir 等）的请求，调度容器存储的依据

比如 nginx 容器请求 30% 的 CPU 和 56MB 的内存，但限制最多只用 50% 的 CPU 和 128MB 的内存：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        requests:
          cpu: "300m"
          memory: "56Mi"
        limits:
          cpu: "1"
          memory: "128Mi"
```

注意

- CPU 的单位是 CPU 个数，可以用 `millicpu (m)` 表示少于 1 个 CPU 的情况，如 `500m = 500millicpu = 0.5cpu` ，而一个 CPU 相当于
  - AWS 上的一个 vCPU
  - GCP 上的一个 Core
  - Azure 上的一个 vCore
  - 物理机上开启超线程时的一个超线程
- 内存的单位则包括 `E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki` 等。
- 从 v1.10 开始，可以设置 `kubelet ----cpu-manager-policy=static` 为 Guaranteed（即 requests.cpu 与 limits.cpu 相等）Pod 绑定 CPU（通过 cpuset cgroups）。

健康检查

为了确保容器在部署后确实处在正常运行状态，Kubernetes 提供了两种探针（Probe）来探测容器的状态：

- LivenessProbe：探测应用是否处于健康状态，如果不健康则删除并重新创建容器
- ReadinessProbe：探测应用是否启动完成并且处于正常服务状态，如果不正常则不会接收来自 Kubernetes Service 的流量

Kubernetes 支持三种方式来执行探针：

- exec：在容器中执行一个命令，如果 命令退出码 返回 `0` 则表示探测成功，否则表示失败
- tcpSocket：对指定的容器 IP 及端口执行一个 TCP 检查，如果端口是开放的则表示探测成功，否则表示失败
- httpGet：对指定的容器 IP、端口及路径执行一个 HTTP Get 请求，如果返回的 状态码 在 `[200,400)` 之间则表示探测成功，否则表示失败

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
    containers:
    - image: nginx
      imagePullPolicy: Always
      name: http
      livenessProbe:
        httpGet:
          path: /
          port: 80
          httpHeaders:
          - name: X-Custom-Header
            value: Awesome
        initialDelaySeconds: 15
        timeoutSeconds: 1
      readinessProbe:
        exec:
          command:
          - cat
          - /usr/share/nginx/html/index.html
        initialDelaySeconds: 5
        timeoutSeconds: 1
    - name: goproxy
      image: gcr.io/google_containers/goproxy:0.1
      ports:
      - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
```

```
        initialDelaySeconds: 15
        periodSeconds: 20
```

Init Container

Pod 能够具有多个容器，应用运行在容器里面，但是它也可能有一个或多个先于应用容器启动的 Init 容器。Init 容器在所有容器运行之前执行（run-to-completion），常用来初始化配置。

如果为一个 Pod 指定了多个 Init 容器，那些容器会按顺序一次运行一个。 每个 Init 容器必须运行成功，下一个才能够运行。 当所有的 Init 容器运行完成时，Kubernetes 初始化 Pod 并像平常一样运行应用容器。

下面是一个 Init 容器的示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html
  # These containers are run during pod initialization
  initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://kubernetes.io
    volumeMounts:
    - name: workdir
      mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
```

因为 Init 容器具有与应用容器分离的单独镜像，使用 init 容器启动相关代码具有如下优势：

- 它们可以包含并运行实用工具，出于安全考虑，是不建议在应用容器镜像中包含这些实用工具的。
- 它们可以包含使用工具和定制化代码来安装，但是不能出现在应用镜像中。例如，创建镜像没必要 FROM 另一个镜像，只需要在安装过程中使用类似 sed、 awk、 python 或 dig 这样的工具。
- 应用镜像可以分离出创建和部署的角色，而没有必要联合它们构建一个单独的镜像。

- 它们使用 Linux Namespace，所以对应用容器具有不同的文件系统视图。因此，它们能够具有访问 Secret 的权限，而应用容器不能够访问。
- 它们在应用容器启动之前运行完成，然而应用容器并行运行，所以 Init 容器提供了一种简单的方式来阻塞或延迟应用容器的启动，直到满足了一组先决条件。

Init 容器的资源计算，选择一下两者的较大值：

- 所有 Init 容器中的资源使用的最大值
- Pod 中所有容器资源使用的总和

Init 容器的重启策略：

- 如果 Init 容器执行失败，Pod 设置的 restartPolicy 为 Never，则 pod 将处于 fail 状态。否则 Pod 将一直重新执行每一个 Init 容器直到所有的 Init 容器都成功。
- 如果 Pod 异常退出，重新拉取 Pod 后，Init 容器也会被重新执行。所以在 Init 容器中执行的任务，需要保证是幂等的。

容器生命周期钩子

容器生命周期钩子（Container Lifecycle Hooks）监听容器生命周期的特定事件，并在事件发生时执行已注册的回调函数。支持两种钩子：

- postStart： 容器创建后立即执行，注意由于是异步执行，它无法保证一定在 ENTRYPOINT 之前运行。如果失败，容器会被杀死，并根据 RestartPolicy 决定是否重启
- preStop：容器终止前执行，常用于资源清理。如果失败，容器同样也会被杀死

而钩子的回调函数支持两种方式：

- exec：在容器内执行命令，如果命令的退出状态码是 `0` 表示执行成功，否则表示失败
- httpGet：向指定 URL 发起 GET 请求，如果返回的 HTTP 状态码在 `[200, 400)` 之间表示请求成功，否则表示失败

postStart 和 preStop 钩子示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        httpGet:
          path: /
          port: 80
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

使用 Capabilities

默认情况下，容器都是以非特权容器的方式运行。比如，不能在容器中创建虚拟网卡、配置虚拟网络。

Kubernetes 提供了修改 Capabilities 的机制，可以按需要给容器增加或删除。比如下面的配置给容器增加了 `CAP_NET_ADMIN` 并删除了 `CAP_KILL` 。

```
apiVersion: v1
kind: Pod
metadata:
  name: cap-pod
spec:
  containers:
  - name: friendly-container
    image: "alpine:3.4"
    command: ["/bin/sleep", "3600"]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        drop:
        - KILL
```

限制网络带宽

可以通过给 Pod 增加 `kubernetes.io/ingress-bandwidth` 和 `kubernetes.io/egress-bandwidth` 这两个 annotation 来限制 Pod 的网络带宽

```
apiVersion: v1
kind: Pod
metadata:
  name: qos
  annotations:
    kubernetes.io/ingress-bandwidth: 3M
    kubernetes.io/egress-bandwidth: 4M
spec:
  containers:
  - name: iperf3
    image: networkstatic/iperf3
    command:
    - iperf3
    - -s
```

> **仅 kubenet 支持限制带宽**
>
> 目前只有 kubenet 网络插件支持限制网络带宽，其他 CNI 网络插件暂不支持这个功能。

kubenet 的网络带宽限制其实是通过 tc 来实现的

```
# setup qdisc (only once)
tc qdisc add dev cbr0 root handle 1: htb default 30
# download rate
tc class add dev cbr0 parent 1: classid 1:2 htb rate 3Mbit
tc filter add dev cbr0 protocol ip parent 1:0 prio 1 u32 match ip
# upload rate
tc class add dev cbr0 parent 1: classid 1:3 htb rate 4Mbit
tc filter add dev cbr0 protocol ip parent 1:0 prio 1 u32 match ip
```

调度到指定的 Node 上

可以通过 nodeSelector、nodeAffinity、podAffinity 以及 Taints 和 tolerations 等来将 Pod 调度到需要的 Node 上。

也可以通过设置 nodeName 参数，将 Pod 调度到指定 node 节点上。

比如，使用 nodeSelector，首先给 Node 加上标签：

```
kubectl label nodes  disktype=ssd
```

接着，指定该 Pod 只想运行在带有 `disktype=ssd` 标签的 Node 上：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

nodeAffinity、podAffinity 以及 Taints 和 tolerations 等的使用方法请参考 调度器章节。

自定义 hosts

默认情况下，容器的 `/etc/hosts` 是 kubelet 自动生成的，并且仅包含 localhost 和 podName 等。不建议在容器内直接修改 `/etc/hosts` 文件，因为在 Pod 启动或重启时会被覆盖。

默认的 `/etc/hosts` 文件格式如下，其中 `nginx-4217019353-fb2c5` 是 podName：

```
$ kubectl exec nginx-4217019353-fb2c5 -- cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
fe00::0    ip6-mcastprefix
fe00::1    ip6-allnodes
fe00::2    ip6-allrouters
10.244.1.4    nginx-4217019353-fb2c5
```

从 v1.7 开始，可以通过 `pod.Spec.HostAliases` 来增加 hosts 内容，如

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
    args:
    - "/etc/hosts"
```

```
$ kubectl logs hostaliases-pod
# Kubernetes-managed hosts file.
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
fe00::0    ip6-mcastprefix
fe00::1    ip6-allnodes
fe00::2    ip6-allrouters
10.244.1.5    hostaliases-pod
127.0.0.1    foo.local
127.0.0.1    bar.local
10.1.2.3    foo.remote
10.1.2.3    bar.remote
```

HugePages

v1.8 + 支持给容器分配 HugePages，资源格式为 `hugepages-`（如 `hugepages-2Mi`）。使用前要配置

- 开启 `--feature-gates="HugePages=true"`
- 在所有 Node 上面预分配好 HugePage ，以便 Kubelet 统计所在 Node 的 HugePage 容量

使用示例

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - image: fedora:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 100Mi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

注意事项

- HugePage 资源的请求和限制必须相同
- HugePage 以 Pod 级别隔离，未来可能会支持容器级的隔离
- 基于 HugePage 的 EmptyDir 存储卷最多只能使用请求的 HugePage 内存
- 使用 `shmget()` 的 `SHM_HUGETLB` 选项时，应用必须运行在匹配 `proc/sys/vm/hugetlb_shm_group` 的用户组（supplemental group）中

优先级

从 v1.8 开始，可以为 Pod 设置一个优先级，保证高优先级的 Pod 优先调度。

优先级调度功能目前为 Beta 版，在 v1.11 版本中默认开启。对 v1.8-1.10 版本中使用前需要开启：

- `--feature-gates=PodPriority=true`
- `--runtime-config=scheduling.k8s.io/v1alpha1=true --admission-control=Controller-Foo,Controller-Bar,...,Priority`

为 Pod 设置优先级前，先创建一个 PriorityClass，并设置优先级（数值越大优先级越高）：

```
apiVersion: scheduling.k8s.io/v1alpha1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service
```

> Kubernetes 自动创建了 `system-cluster-critical` 和 `system-node-critical` 等两个 PriorityClass，用于 Kubernetes 核心组件。

为 Pod 指定优先级

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

当调度队列有多个 Pod 需要调度时，优先调度高优先级的 Pod。而当高优先级的 Pod 无法调度时，Kubernetes 会尝试先删除低优先级的 Pod 再将其调度到对应 Node 上（Preemption）。

注意：**受限于 Kubernetes 的调度策略，抢占并不总是成功。**

PodDisruptionBudget

PodDisruptionBudget (PDB) 用来保证一组 Pod 同时运行的数量，这些 Pod 需要使用 Deployment、ReplicationController、ReplicaSet 或者 StatefulSet 管理。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

Sysctls

Sysctls 允许容器设置内核参数，分为安全 Sysctls 和非安全 Sysctls：

- 安全 Sysctls：即设置后不影响其他 Pod 的内核选项，只作用在容器 namespace 中，默认开启。包括以下几种
  - `kernel.shm_rmid_forced`
  - `net.ipv4.ip_local_port_range`
  - `net.ipv4.tcp_syncookies`
- 非安全 Sysctls：即设置好有可能影响其他 Pod 和 Node 上其他服务的内核选项，默认禁止。如果使用，需要管理员在配置 kubelet 时开启，如 `kubelet --experimental-allowed-unsafe-sysctls 'kernel.msg*,net.ipv4.route.min_pmtu'`

v1.6-v1.10 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  annotations:
    security.alpha.kubernetes.io/sysctls: kernel.shm_rmid_forced=
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.m
spec:
  ...
```

从 v1.11 开始，Sysctls 升级为 Beta 版本，不再区分安全和非安全 sysctl，统一通过 podSpec.securityContext.sysctls 设置，如

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.ipv4.route.min_pmtu
      value: "552"
    - name: kernel.msgmax
      value: "65536"
  ...
```

Pod 时区

很多容器都是配置了 UTC 时区，与国内集群的 Node 所在时区有可能不一致，可以通过 HostPath 存储插件给容器配置与 Node 一样的时区：

```
apiVersion: v1
kind: Pod
metadata:
  name: sh
  namespace: default
spec:
  containers:
  - image: alpine
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: /etc/localtime
      name: time
      readOnly: true
  volumes:
  - hostPath:
      path: /etc/localtime
      type: ""
    name: time
```

参考文档

- What is Pod?
- Kubernetes Pod Lifecycle
- DNS Pods and Services
- Container capabilities
- Configure Liveness and Readiness Probes
- Init Containers
- Linux Capabilities
- Manage HugePages
- Document supported docker image (Dockerfile) features

PodPreset

PodPreset 用来给指定标签的 Pod 注入额外的信息，如环境变量、存储卷等。这样，Pod 模板就不需要为每个 Pod 都显式设置重复的信息。

API 版本对照表

Kubernetes 版本	API 版本	默认开启
v1.6+	settings.k8s.io/v1alpha1	否

开启 PodPreset

- 开启 API `settings.k8s.io/v1alpha1/podpreset`
- 开启准入控制 `PodPreset`

PodPreset 示例

增加环境变量和存储卷的 PodPreset

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

用户提交 Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

经过准入控制 `PodPreset` 后，Pod 会自动增加环境变量和存储卷

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/allow-database: "resource v
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
```

ConfigMap 示例

ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: etcd-env-config
data:
  number_of_members: "1"
  initial_cluster_state: new
  initial_cluster_token: DUMMY_ETCD_INITIAL_CLUSTER_TOKEN
  discovery_token: DUMMY_ETCD_DISCOVERY_TOKEN
  discovery_url: http://etcd_discovery:2379
  etcdctl_peers: http://etcd:2379
  duplicate_key: FROM_CONFIG_MAP
  REPLACE_ME: "a value"
```

PodPreset

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: 6379
    - name: duplicate_key
      value: FROM_ENV
    - name: expansion
      value: $(REPLACE_ME)
  envFrom:
    - configMapRef:
        name: etcd-env-config
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: /etc/app/config.json
      readOnly: true
      name: secret-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secretName: config-details
```

用户提交的 Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

经过准入控制 `PodPreset` 后，Pod 会自动增加 ConfigMap 环境变量

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/allow-database: "resource v
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/app/config.json
          readOnly: true
          name: secret-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
        - name: duplicate_key
          value: FROM_ENV
        - name: expansion
          value: $(REPLACE_ME)
      envFrom:
        - configMapRef:
          name: etcd-env-config
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secretName: config-details
```

ReplicaSet

ReplicationController（也简称为 rc）用来确保容器应用的副本数始终保持在用户定义的副本数，即如果有容器异常退出，会自动创建新的 Pod 来替代；而异常多出来的容器也会自动回收。ReplicationController 的典型应用场景包括确保健康 Pod 的数量、弹性伸缩、滚动升级以及应用多版本发布跟踪等。

在新版本的 Kubernetes 中建议使用 ReplicaSet（也简称为 rs）来取代 ReplicationController。ReplicaSet 跟 ReplicationController 没有本质的不同，只是名字不一样，并且 ReplicaSet 支持集合式的 selector（ReplicationController 仅支持等式）。

虽然也 ReplicaSet 可以独立使用，但建议使用 Deployment 来自动管理 ReplicaSet，这样就无需担心跟其他机制的不兼容问题（比如 ReplicaSet 不支持 rolling-update 但 Deployment 支持），并且还支持版本记录、回滚、暂停升级等高级特性。Deployment 的详细介绍和使用方法见 这里。

API 版本对照表

Kubernetes 版本	ReplicaSet API 版本	ReplicationController 版本
v1.5-v1.7	extensions/v1beta1	core/v1
v1.8	apps/v1beta2	core/v1
v1.9	apps/v1	core/v1

ReplicationController 示例

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

ReplicaSet 示例

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
    # app: guestbook
    # tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set,
  # but we are specifying the selector here to
  # demonstrate its usage.
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service
          # instead access environment variables to find service
```

```
          # info, comment out the 'value: dns' line above, and ur
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

ResourceQuota

资源配额（Resource Quotas）是用来限制用户资源用量的一种机制。

它的工作原理为

- 资源配额应用在 Namespace 上，并且每个 Namespace 最多只能有一个 `ResourceQuota` 对象
- 开启计算资源配额后，创建容器时必须配置计算资源请求或限制（也可以用 LimitRange 设置默认值）
- 用户超额后禁止创建新的资源

开启资源配额功能

- 首先，在 API Server 启动时配置准入控制 `--admission-control=ResourceQuota`
- 然后，在 namespace 中创建一个 `ResourceQuota` 对象

资源配额的类型

- 计算资源，包括 cpu 和 memory
  - cpu, limits.cpu, requests.cpu
  - memory, limits.memory, requests.memory
- 存储资源，包括存储资源的总量以及指定 storage class 的总量
  - requests.storage：存储资源总量，如 500Gi
  - persistentvolumeclaims：pvc 的个数
  - .storageclass.storage.k8s.io/requests.storage
  - .storageclass.storage.k8s.io/persistentvolumeclaims
  - requests.ephemeral-storage 和 limits.ephemeral-storage （需要 v1.8+）
- 对象数，即可创建的对象的个数
  - pods, replicationcontrollers, configmaps, secrets
  - resourcequotas, persistentvolumeclaims

- services, services.loadbalancers, services.nodeports

计算资源示例

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

对象个数示例

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
```

LimitRange

默认情况下，Kubernetes 中所有容器都没有任何 CPU 和内存限制。
LimitRange 用来给 Namespace 增加一个资源限制，包括最小、最大和默认资源。比如

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimits
spec:
  limits:
  - max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 6Mi
    type: Pod
  - default:
      cpu: 300m
      memory: 200Mi
    defaultRequest:
      cpu: 200m
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 100m
      memory: 3Mi
    type: Container
```

```
$ kubectl create -f https://k8s.io/docs/tasks/configure-pod-conta
limitrange "mylimits" created
$ kubectl describe limits mylimits --namespace=limit-example
Name:   mylimits
Namespace:  limit-example
Type        Resource      Min      Max      Default Request
----        --------      ---      ---      ---------------
Pod         cpu           200m     2        -
Pod         memory        6Mi      1Gi      -
Container   cpu           100m     2        200m
Container   memory        3Mi      1Gi      100Mi
```

配额范围

每个配额在创建时可以指定一系列的范围

范围	说明
Terminating	podSpec.ActiveDeadlineSeconds>=0 的 Pod
NotTerminating	podSpec.activeDeadlineSeconds=nil 的 Pod
BestEffort	所有容器的 requests 和 limits 都没有设置的 Pod（Best-Effort）
NotBestEffort	与 BestEffort 相反

Secret

Secret 解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者 Pod Spec 中。Secret 可以以 Volume 或者环境变量的方式使用。

Secret 类型

Secret 有三种类型：

- Opaque：base64 编码格式的 Secret，用来存储密码、密钥等；但数据也通过 base64 --decode 解码得到原始数据，所有加密性很弱。
- `kubernetes.io/dockerconfigjson`：用来存储私有 docker registry 的认证信息。
- `kubernetes.io/service-account-token`： 用于被 serviceaccount 引用。serviceaccout 创建时 Kubernetes 会默认创建对应的 secret。Pod 如果使用了 serviceaccount，对应的 secret 会自动挂载到 Pod 的 `/run/secrets/kubernetes.io/serviceaccount` 目录中。

备注：serviceaccount 用来使得 Pod 能够访问 Kubernetes API

API 版本对照表

Kubernetes 版本	Core API 版本
v1.5+	core/v1

Opaque Secret

Opaque 类型的数据是一个 map 类型，要求 value 是 base64 编码格式：

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
```

创建 secret： `kubectl create -f secrets.yml` 。

```
# kubectl get secret
NAME                  TYPE                                  DATA
default-token-cty7p   kubernetes.io/service-account-token   3
mysecret              Opaque                                2
```

注意：其中 default-token-cty7p 为创建集群时默认创建的 secret，被 serviceacount/default 引用。

如果是从文件创建 secret，则可以用更简单的 kubectl 命令，比如创建 tls 的 secret：

```
$ kubectl create secret generic helloworld-tls \
  --from-file=key.pem \
  --from-file=cert.pem
```

Opaque Secret 的使用

创建好 secret 之后，有两种方式来使用它：

- 以 Volume 方式
- 以环境变量方式

将 Secret 挂载到 Volume 中

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: db
  name: db
spec:
  volumes:
  - name: secrets
    secret:
      secretName: mysecret
  containers:
  - image: gcr.io/my_project_id/pg:v1
    name: db
    volumeMounts:
    - name: secrets
      mountPath: "/etc/secrets"
      readOnly: true
    ports:
    - name: cp
      containerPort: 5432
      hostPort: 5432
```

查看 Pod 中对应的信息：

```
# ls /etc/secrets
password  username
# cat  /etc/secrets/username
admin
# cat  /etc/secrets/password
1f2d1e2e67df
```

将 Secret 导出到环境变量中

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 2
  strategy:
      type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
        visualize: "true"
    spec:
      containers:
      - name: "wordpress"
        image: "wordpress"
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

将 Secret 挂载指定的 key

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: db
  name: db
spec:
  volumes:
  - name: secrets
    secret:
      secretName: mysecret
      items:
      - key: password
        mode: 511
        path: tst/psd
      - key: username
        mode: 511
        path: tst/usr
  containers:
  - image: nginx
    name: db
    volumeMounts:
    - name: secrets
      mountPath: "/etc/secrets"
      readOnly: true
    ports:
    - name: cp
      containerPort: 80
      hostPort: 5432
```

创建 Pod 成功后，可以在对应的目录看到：

```
# kubectl exec db ls /etc/secrets/tst
psd
usr
```

注意 ：

1、 `kubernetes.io/dockerconfigjson` 和 `kubernetes.io/service-account-token` 类型的 secret 也同样可以被挂载成文件 (目录)。 如果使用 `kubernetes.io/dockerconfigjson` 类型的 secret 会在目录下创建一个. dockercfg 文件

```
root@db:/etc/secrets# ls -al
total 4
drwxrwxrwt  3 root root  100 Aug  5 16:06 .
drwxr-xr-x 42 root root 4096 Aug  5 16:06 ..
drwxr-xr-x  2 root root   60 Aug  5 16:06 ..8988_06_08_00_06_52.4
lrwxrwxrwx  1 root root   31 Aug  5 16:06 ..data -> ..8988_06_08_
lrwxrwxrwx  1 root root   17 Aug  5 16:06 .dockercfg -> ..data/.d
```

如果使用 `kubernetes.io/service-account-token` 类型的 secret 则会创建 ca.crt，namespace，token 三个文件

```
root@db:/etc/secrets# ls
ca.crt    namespace  token
```

2、secrets 使用时被挂载到一个临时目录，Pod 被删除后 secrets 挂载时生成的文件也会被删除。

```
root@db:/etc/secrets# df
Filesystem     1K-blocks    Used Available Use% Mounted on
none           123723748 4983104 112432804   5% /
tmpfs            1957660       0   1957660   0% /dev
tmpfs            1957660       0   1957660   0% /sys/fs/cgroup
/dev/vda1       51474044 2444568  46408092   6% /etc/hosts
tmpfs            1957660      12   1957648   1% /etc/secrets
/dev/vdb       123723748 4983104 112432804   5% /etc/hostname
shm               65536       0     65536   0% /dev/shm
```

但如果在 Pod 运行的时候，在 Pod 部署的节点上还是可以看到：

```
# 查看 Pod 中容器 Secret 的相关信息，其中 4392b02d-79f9-11e7-a70a-52540
"Mounts": [
  {
    "Source": "/var/lib/kubelet/pods/4392b02d-79f9-11e7-a70a-5254
    "Destination": "/etc/secrets",
    "Mode": "ro",
    "RW": false,
    "Propagation": "rprivate"
  }
]
#在 Pod 部署的节点查看
root@VM-0-178-ubuntu:/var/lib/kubelet/pods/4392b02d-79f9-11e7-a70
total 4
drwxrwxrwt 3 root root  140 Aug  6 00:15 .
drwxr-xr-x 3 root root 4096 Aug  6 00:15 ..
drwxr-xr-x 2 root root  100 Aug  6 00:15 ..8988_06_08_00_15_14.25
lrwxrwxrwx 1 root root   31 Aug  6 00:15 ..data -> ..8988_06_08_0
lrwxrwxrwx 1 root root   13 Aug  6 00:15 ca.crt -> ..data/ca.crt
lrwxrwxrwx 1 root root   16 Aug  6 00:15 namespace -> ..data/name
lrwxrwxrwx 1 root root   12 Aug  6 00:15 token -> ..data/token
```

kubernetes.io/dockerconfigjson

可以直接用 kubectl 命令来创建用于 docker registry 认证的 secret：

```
$ kubectl create secret docker-registry myregistrykey --docker-se
secret "myregistrykey" created.
```

查看 secret 的内容：

```
# kubectl get secret myregistrykey  -o yaml
apiVersion: v1
data:
  .dockercfg: eyJjY3IuY2NzLnRlbmNlbnR5dW4uY29tL3RlbmNlbnR5dW4iOns
kind: Secret
metadata:
  creationTimestamp: 2017-08-04T02:06:05Z
  name: myregistrykey
  namespace: default
  resourceVersion: "1374279324"
  selfLink: /api/v1/namespaces/default/secrets/myregistrykey
  uid: 78f6a423-78b9-11e7-a70a-525400bc11f0
type: kubernetes.io/dockercfg
```

通过 base64 对 secret 中的内容解码：

```
# echo "eyJjY3IuY2NzLnRlbmNlbnR5dW4uY29tL3RlbmNlbnR5dW4iOnsidXNlc
{"ccr.ccs.tencentyun.com/XXXXXXX":{"username":"3321337XXX","passw
```

也可以直接读取 `~/.dockercfg` 的内容来创建：

```
$ kubectl create secret docker-registry myregistrykey \
  --from-file="~/.dockercfg"
```

在创建 Pod 的时候，通过 `imagePullSecrets` 来引用刚创建的 `myregistrykey`：

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

kubernetes.io/service-account-token

`kubernetes.io/service-account-token` ： 用于被 serviceaccount 引用。serviceaccout 创建时 Kubernetes 会默认创建对应的 secret。Pod 如果使用了 serviceaccount，对应的 secret 会自动挂载到 Pod 的 `/run/secrets/kubernetes.io/serviceaccount` 目录中。

```
$ kubectl run nginx --image nginx
deployment "nginx" created
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
nginx-3137573019-md1u2   1/1       Running   0          13s
$ kubectl exec nginx-3137573019-md1u2 ls /run/secrets/kubernetes.
ca.crt
namespace
token
```

存储加密

v1.7 + 版本支持将 Secret 数据加密存储到 etcd 中，只需要在 apiserver 启动时配置 `--experimental-encryption-provider-config` 。加密配置格式为

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - identity: {}
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - secretbox:
        keys:
        - name: key1
          secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
```

其中

- resources.resources 是 Kubernetes 的资源名
- resources.providers 是加密方法，支持以下几种
  - identity：不加密
  - aescbc：AES-CBC 加密
  - secretbox：XSalsa20 和 Poly1305 加密
  - aesgcm：AES-GCM 加密

Secret 是在写存储的时候加密，因而可以对已有的 secret 执行 update 操作来保证所有的 secrets 都加密

```
kubectl get secrets –o json | kubectl update –f –
```

如果想取消 secret 加密的话，只需要把 `identity` 放到 providers 的第一个位置即可（aescbc 还要留着以便访问已存储的 secret）：

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
```

Secret 与 ConfigMap 对比

相同点：

- key/value 的形式
- 属于某个特定的 namespace
- 可以导出到环境变量
- 可以通过目录 / 文件形式挂载 (支持挂载所有 key 和部分 key)

不同点：

- Secret 可以被 ServerAccount 关联 (使用)
- Secret 可以存储 register 的鉴权信息，用在 ImagePullSecret 参数中，用于拉取私有仓库的镜像
- Secret 支持 Base64 加密
- Secret 分为 Opaque，kubernetes.io/Service Account，kubernetes.io/dockerconfigjson 三种类型, Configmap 不区分类型
- Secret 文件存储在 tmpfs 文件系统中，Pod 删除后 Secret 文件也会对应的删除。

参考文档

- Secret
- Specifying ImagePullSecrets on a Pod

SecurityContext

Security Context 的目的是限制不可信容器的行为，保护系统和其他容器不受其影响。

Kubernetes 提供了三种配置 Security Context 的方法：

- Container-level Security Context：仅应用到指定的容器
- Pod-level Security Context：应用到 Pod 内所有容器以及 Volume
- Pod Security Policies（PSP）：应用到集群内部所有 Pod 以及 Volume

Container-level Security Context

Container-level Security Context 仅应用到指定的容器上，并且不会影响 Volume。比如设置容器运行在特权模式：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    - name: hello-world-container
      # The container definition
      # ...
      securityContext:
        privileged: true
```

Pod-level Security Context

Pod-level Security Context 应用到 Pod 内所有容器，并且还会影响 Volume（包括 fsGroup 和 selinuxOptions）。

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
  # specification of the pod's containers
  # ...
  securityContext:
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: "s0:c123,c456"
```

Pod Security Policies（PSP）

Pod Security Policies（PSP）是集群级的 Pod 安全策略，自动为集群内的 Pod 和 Volume 设置 Security Context。

使用 PSP 需要 API Server 开启 `extensions/v1beta1/podsecuritypolicy`，并且配置 `PodSecurityPolicy` admission 控制器。

支持的控制项

控制项	说明
privileged	运行特权容器
defaultAddCapabilities	可添加到容器的 Capabilities
requiredDropCapabilities	会从容器中删除的 Capabilities
allowedCapabilities	允许使用的 Capabilities 列表
volumes	控制容器可以使用哪些 volume
hostNetwork	允许使用 host 网络
hostPorts	允许的 host 端口列表
hostPID	使用 host PID namespace
hostIPC	使用 host IPC namespace
seLinux	SELinux Context
runAsUser	user ID
supplementalGroups	允许的补充用户组
fsGroup	volume FSGroup
readOnlyRootFilesystem	只读根文件系统
allowedHostPaths	允许 hostPath 插件使用的路径列表
allowedFlexVolumes	允许使用的 flexVolume 插件列表
allowPrivilegeEscalation	允许容器进程设置 `no_new_privs`
defaultAllowPrivilegeEscalation	默认是否允许特权升级

示例

限制容器的 host 端口范围为 8000-8080：

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - min: 8000
    max: 8080
  volumes:
  - '*'
```

限制只允许使用 lvm 和 cifs 等 flexVolume 插件：

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: allow-flex-volumes
spec:
  fsGroup:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
    - flexVolume
  allowedFlexVolumes:
    - driver: example/lvm
    - driver: example/cifs
```

SELinux

SELinux (Security-Enhanced Linux) 是一种强制访问控制（mandatory access control）的实现。它的作法是以最小权限原则（principle of least privilege）为基础，在 Linux 核心中使用 Linux 安全模块（Linux Security Modules）。SELinux 主要由美国国家安全局开发，并于 2000 年 12 月 22 日发行给开放源代码的开发社区。

可以通过 runcon 来为进程设置安全策略，ls 和 ps 的 - Z 参数可以查看文件或进程的安全策略。

开启与关闭 SELinux

修改 / etc/selinux/config 文件方法：

- 开启：SELINUX=enforcing
- 关闭：SELINUX=disabled

通过命令临时修改：

- 开启：setenforce 1
- 关闭：setenforce 0

查询 SELinux 状态：

```
$ getenforce
```

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
  - image: gcr.io/google_containers/busybox:1.24
    name: test-container
    command:
    - sleep
    - "6000"
    volumeMounts:
    - mountPath: /mounted_volume
      name: test-volume
  restartPolicy: Never
  hostPID: false
  hostIPC: false
  securityContext:
    seLinuxOptions:
      level: "s0:c2,c3"
  volumes:
  - name: test-volume
    emptyDir: {}
```

这会自动给 docker 容器生成如下的 `HostConfig.Binds`：

```
/var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/volume
/var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/volume
/var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/etc-ho
```

对应的 volume 也都会正确设置 SELinux：

```
$ ls -Z /var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c000
drwxr-xr-x. root root unconfined_u:object_r:svirt_sandbox_file_t:
drwxr-xr-x. root root unconfined_u:object_r:svirt_sandbox_file_t:
```

参考文档

- Kubernetes Pod Security Policies

Service

Kubernetes 在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了 Service 资源，并通过 kube-proxy 配合 cloud provider 来适应不同的应用场景。随着 kubernetes 用户的激增，用户场景的不断丰富，又产生了一些新的负载均衡机制。目前，kubernetes 中的负载均衡大致可以分为以下几种机制，每种机制都有其特定的应用场景：

- Service：直接用 Service 提供 cluster 内部的负载均衡，并借助 cloud provider 提供的 LB 提供外部访问
- Ingress Controller：还是用 Service 提供 cluster 内部的负载均衡，但是通过自定义 Ingress Controller 提供外部访问
- Service Load Balancer：把 load balancer 直接跑在容器中，实现 Bare Metal 的 Service Load Balancer
- Custom Load Balancer：自定义负载均衡，并替代 kube-proxy，一般在物理部署 Kubernetes 时使用，方便接入公司已有的外部服务

Service

Service 是对一组提供相同功能的 Pods 的抽象，并为它们提供一个统一的入口。借助 Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service 通过标签来选取服务后端，一般配合 Replication Controller 或者 Deployment 来保证后端容器的正常运行。这些匹配标签的 Pod IP 和端口列表组成 endpoints，由 kube-proxy 负责将服务 IP 负载均衡到这些 endpoints 上。

Service 有四种类型：

- ClusterIP：默认类型，自动分配一个仅 cluster 内部可以访问的虚拟 IP
- NodePort：在 ClusterIP 基础上为 Service 在每台机器上绑定一个端口，这样就可以通过 `:NodePort` 来访问该服务。如果 kube-proxy 设置了 `--nodeport-addresses=10.240.0.0/16`（v1.10 支持），那么仅该 NodePort 仅对设置在范围内的 IP 有效。
- LoadBalancer：在 NodePort 的基础上，借助 cloud provider 创建一个外部的负载均衡器，并将请求转发到 `:NodePort`

- ExternalName：将服务通过 DNS CNAME 记录方式转发到指定的域名（通过 `spec.externlName` 设定）。需要 kube-dns 版本在 1.7 以上。

另外，也可以将已有的服务以 Service 的形式加入到 Kubernetes 集群中来，只需要在创建 Service 的时候不指定 Label selector，而是在 Service 创建好后手动为其添加 endpoint。

Service 定义

Service 的定义也是通过 yaml 或 json，比如下面定义了一个名为 nginx 的服务，将服务的 80 端口转发到 default namespace 中带有标签 `run=nginx` 的 Pod 的 80 端口

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx
  name: nginx
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  sessionAffinity: None
  type: ClusterIP
```

```
# service 自动分配了 Cluster IP 10.0.0.108
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     10.0.0.108           80/TCP    18m
# 自动创建的 endpoint
$ kubectl get endpoints nginx
NAME      ENDPOINTS       AGE
nginx     172.17.0.5:80   18m
# Service 自动关联 endpoint
$ kubectl describe service nginx
Name:            nginx
Namespace:        default
Labels:            run=nginx
Annotations:
Selector:        run=nginx
Type:            ClusterIP
IP:            10.0.0.108
Port:            <unset>    80/TCP
Endpoints:        172.17.0.5:80
Session Affinity:    None
Events:
```

当服务需要多个端口时，每个端口都必须设置一个名字

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
    port: 443
    targetPort: 9377
```

协议

Service、Endpoints 和 Pod 支持三种类型的协议：

- TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。
- UDP（User Datagram Protocol，用户数据报协议）是一种无连接的传输层协议，用于不可靠信息传送服务。
- SCTP（Stream Control Transmission Protocol，流控制传输协议），用于通过IP网传输SCN（Signaling Communication Network，信令通信网）窄带信令消息。

API 版本对照表

Kubernetes 版本	Core API 版本
v1.5+	core/v1

不指定 Selectors 的服务

在创建 Service 的时候，也可以不指定 Selectors，用来将 service 转发到 kubernetes 集群外部的服务（而不是 Pod）。目前支持两种方法

（1）自定义 endpoint，即创建同名的 service 和 endpoint，在 endpoint 中设置外部服务的 IP 和端口

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
---
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

（2）通过 DNS 转发，在 service 定义中指定 externalName。此时 DNS 服务会给 `..svc.cluster.local` 创建一个 CNAME 记录，其值为 `my.database.example.com` 。并且，该服务不会自动分配 Cluster IP，需要通过 service 的 DNS 来访问。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: default
spec:
  type: ExternalName
  externalName: my.database.example.com
```

注意：Endpoints 的 IP 地址不能是 127.0.0.0/8、169.254.0.0/16 和 224.0.0.0/24，也不能是 Kubernetes 中其他服务的 clusterIP。

Headless 服务

Headless 服务即不需要 Cluster IP 的服务，即在创建服务的时候指定 `spec.clusterIP=None` 。包括两种类型

- 不指定 Selectors，但设置 externalName，即上面的（2），通过 CNAME 记录处理
- 指定 Selectors，通过 DNS A 记录设置后端 endpoint 列表

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  clusterIP: None
  ports:
  - name: tcp-80-80-3b6tl
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  sessionAffinity: None
  type: ClusterIP
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: default
spec:
  replicas: 2
  revisionHistoryLimit: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        imagePullPolicy: Always
        name: nginx
```

```
        resources:
          limits:
            memory: 128Mi
          requests:
            cpu: 200m
            memory: 128Mi
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

```
# 查询创建的 nginx 服务
$ kubectl get service --all-namespaces=true
NAMESPACE     NAME         CLUSTER-IP      EXTERNAL-IP      PORT(
default       nginx        None                         80/TCP
kube-system   kube-dns     172.26.255.70                53/UDP,53/T
$ kubectl get pod
NAME                        READY     STATUS    RESTARTS   AGE
nginx-2204978904-6o5dg      1/1       Running   0          14s
nginx-2204978904-qyilx      1/1       Running   0          14s
$ dig @172.26.255.70  nginx.default.svc.cluster.local
;; ANSWER SECTION:
nginx.default.svc.cluster.local. 30 IN    A    172.26.1.5
nginx.default.svc.cluster.local. 30 IN    A    172.26.2.5
```

备注： 其中 dig 命令查询的信息中，部分信息省略

保留源 IP

各种类型的 Service 对源 IP 的处理方法不同：

- ClusterIP Service：使用 iptables 模式，集群内部的源 IP 会保留（不做 SNAT）。如果 client 和 server pod 在同一个 Node 上，那源 IP 就是 client pod 的 IP 地址；如果在不同的 Node 上，源 IP 则取决于网络插件是如何处理的，比如使用 flannel 时，源 IP 是 node flannel IP 地址。
- NodePort Service：默认情况下，源 IP 会做 SNAT，server pod 看到的源 IP 是 Node IP。为了避免这种情况，可以给 service 设置 `spec.ExternalTrafficPolicy=Local`（1.6-1.7 版本设置 Annotation `service.beta.kubernetes.io/external-traffic=OnlyLocal`），让 service 只代理本地 endpoint 的请求（如果没有本地 endpoint 则直接丢包），从而保留源 IP。

- LoadBalancer Service：默认情况下，源 IP 会做 SNAT，server pod 看到的源 IP 是 Node IP。设置 `service.spec.ExternalTrafficPolicy=Local` 后可以自动从云平台负载均衡器中删除没有本地 endpoint 的 Node，从而保留源 IP。

工作原理

kube-proxy 负责将 service 负载均衡到后端 Pod 中，如下图所示

Ingress

Service 虽然解决了服务发现和负载均衡的问题，但它在使用上还是有一些限制，比如

- 只支持 4 层负载均衡，没有 7 层功能 - 对外访问的时候，NodePort 类型需要在外部搭建额外的负载均衡，而 LoadBalancer 要求 kubernetes 必须跑在支持的 cloud provider 上面

Ingress 就是为了解决这些限制而引入的新资源，主要用来将服务暴露到 cluster 外面，并且可以自定义服务的访问策略。比如想要通过负载均衡器实现不同子域名到不同服务的访问：

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

可以这样来定义 Ingress：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

注意 Ingress 本身并不会自动创建负载均衡器，cluster 中需要运行一个 ingress controller 来根据 Ingress 的定义来管理负载均衡器。目前社区提供了 nginx 和 gce 的参考实现。

Traefik 提供了易用的 Ingress Controller，使用方法见 https://docs.traefik.io/user-guide/kubernetes/。

更多 Ingress 和 Ingress Controller 的介绍参见 ingress。

Service Load Balancer

在 Ingress 出现以前，Service Load Balancer 是推荐的解决 Service 局限性的方式。Service Load Balancer 将 haproxy 跑在容器中，并监控 service 和 endpoint 的变化，通过容器 IP 对外提供 4 层和 7 层负载均衡服务。

社区提供的 Service Load Balancer 支持四种负载均衡协议：TCP、HTTP、HTTPS 和 SSL TERMINATION，并支持 ACL 访问控制。

> 注意：Service Load Balancer 已不再推荐使用，推荐使用 Ingress Controller。

Custom Load Balancer

虽然 Kubernetes 提供了丰富的负载均衡机制，但在实际使用的时候，还是会碰到一些复杂的场景是它不能支持的，比如

- 接入已有的负载均衡设备
- 多租户网络情况下，容器网络和主机网络是隔离的，这样 `kube-proxy` 就不能正常工作

这个时候就可以自定义组件，并代替 kube-proxy 来做负载均衡。基本的思路是监控 kubernetes 中 service 和 endpoints 的变化，并根据这些变化来配置负载均衡器。比如 weave flux、nginx plus、kube2haproxy 等。

集群外部访问服务

Service 的 ClusterIP 是 Kubernetes 内部的虚拟 IP 地址，无法直接从外部直接访问。但如果需要从外部访问这些服务该怎么办呢，有多种方法

- 使用 NodePort 服务在每台机器上绑定一个端口，这样就可以通过 `:NodePort` 来访问该服务。
- 使用 LoadBalancer 服务借助 Cloud Provider 创建一个外部的负载均衡器，并将请求转发到 `:NodePort`。该方法仅适用于运行在云平台之中的 Kubernetes 集群。对于物理机部署的集群，可以使用 MetalLB 实现类似的功能。
- 使用 Ingress Controller 在 Service 之上创建 L7 负载均衡并对外开放。
- 使用 ECMP 将 Service ClusterIP 网段路由到每个 Node，这样可以直接通过 ClusterIP 来访问服务，甚至也可以直接在集群外部使用 kube-dns。这一版用在物理机部署的情况下。

参考资料

- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/concepts/services-networking/ingress/
- https://github.com/kubernetes/contrib/tree/master/service-loadbalancer
- https://www.nginx.com/blog/load-balancing-kubernetes-services-nginx-plus/
- https://github.com/weaveworks/flux
- https://github.com/AdoHe/kube2haproxy

- Accessing Kubernetes Services Without Ingress, NodePort, or LoadBalancer

ServiceAccount

Service account 是为了方便 Pod 里面的进程调用 Kubernetes API 或其他外部服务而设计的。它与 User account 不同

- User account 是为人设计的，而 service account 则是为 Pod 中的进程调用 Kubernetes API 而设计；
- User account 是跨 namespace 的，而 service account 则是仅局限它所在的 namespace；
- 每个 namespace 都会自动创建一个 default service account
- Token controller 检测 service account 的创建，并为它们创建 secret
- 开启 ServiceAccount Admission Controller 后
  - 每个 Pod 在创建后都会自动设置 `spec.serviceAccountName` 为 default（除非指定了其他 ServiceAccout）
  - 验证 Pod 引用的 service account 已经存在，否则拒绝创建
  - 如果 Pod 没有指定 ImagePullSecrets，则把 service account 的 ImagePullSecrets 加到 Pod 中
  - 每个 container 启动后都会挂载该 service account 的 token 和 `ca.crt` 到 `/var/run/secrets/kubernetes.io/serviceaccount/`

```
$ kubectl exec nginx-3137573019-md1u2 ls /var/run/secrets/kuberne
ca.crt
namespace
token
```

创建 Service Account

```
$ kubectl create serviceaccount jenkins
serviceaccount "jenkins" created
$ kubectl get serviceaccounts jenkins -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2017-05-27T14:32:25Z
  name: jenkins
  namespace: default
  resourceVersion: "45559"
  selfLink: /api/v1/namespaces/default/serviceaccounts/jenkins
  uid: 4d66eb4c-42e9-11e7-9860-ee7d8982865f
secrets:
- name: jenkins-token-l9v7v
```

自动创建的 secret：

```
kubectl get secret jenkins-token-l9v7v -o yaml
apiVersion: v1
data:
  ca.crt: (APISERVER CA BASE64 ENCODED)
  namespace: ZGVmYXVsdA==
  token: (BEARER TOKEN BASE64 ENCODED)
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: jenkins
    kubernetes.io/service-account.uid: 4d66eb4c-42e9-11e7-9860-ee
  creationTimestamp: 2017-05-27T14:32:25Z
  name: jenkins-token-l9v7v
  namespace: default
  resourceVersion: "45558"
  selfLink: /api/v1/namespaces/default/secrets/jenkins-token-l9v7
  uid: 4d697992-42e9-11e7-9860-ee7d8982865f
type: kubernetes.io/service-account-token
```

添加 ImagePullSecrets

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
```

授权

Service Account 为服务提供了一种方便的认证机制，但它不关心授权的问题。可以配合 RBAC 来为 Service Account 鉴权：

- 配置 `--authorization-mode=RBAC` 和 `--runtime-config=rbac.authorization.k8s.io/v1alpha1`
- 配置 `--authorization-rbac-super-user=admin`
- 定义 Role、ClusterRole、RoleBinding 或 ClusterRoleBinding

比如

```
# This role allows to read pods in the namespace "default"
kind: Role
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  namespace: default
  name: pod-reader
rules:
  - apiGroups: [""] # The API group"" indicates the core API Grou
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
    nonResourceURLs: []
---
# This role binding allows "default" to read pods in the namespac
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: ServiceAccount # May be "User", "Group" or "ServiceAccc
    name: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

StatefulSet

StatefulSet 是为了解决有状态服务的问题（对应 Deployments 和 ReplicaSets 是为无状态服务而设计），其应用场景包括

- 稳定的持久化存储，即 Pod 重新调度后还是能访问到相同的持久化数据，基于 PVC 来实现
- 稳定的网络标志，即 Pod 重新调度后其 PodName 和 HostName 不变，基于 Headless Service（即没有 Cluster IP 的 Service）来实现
- 有序部署，有序扩展，即 Pod 是有顺序的，在部署或者扩展的时候要依据定义的顺序依次依序进行（即从 0 到 N-1，在下一个 Pod 运行之前所有之前的 Pod 必须都是 Running 和 Ready 状态），基于 init containers 来实现
- 有序收缩，有序删除（即从 N-1 到 0）

从上面的应用场景可以发现，StatefulSet 由以下几个部分组成：

- 用于定义网络标志（DNS domain）的 Headless Service
- 用于创建 PersistentVolumes 的 volumeClaimTemplates
- 定义具体应用的 StatefulSet

StatefulSet 中每个 Pod 的 DNS 格式为 `statefulSetName-{0..N-1}.serviceName.namespace.svc.cluster.local`，其中

- `serviceName` 为 Headless Service 的名字
- `0..N-1` 为 Pod 所在的序号，从 0 开始到 N-1
- `statefulSetName` 为 StatefulSet 的名字
- `namespace` 为服务所在的 namespace，Headless Service 和 StatefulSet 必须在相同的 namespace
- `.cluster.local` 为 Cluster Domain

API 版本对照表

Kubernetes 版本	Apps 版本
v1.6-v1.7	apps/v1beta1
v1.8	apps/v1beta2
v1.9	apps/v1

简单示例

以一个简单的 nginx 服务 web.yaml 为例：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
```

```
      name: www
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

```
$ kubectl create -f web.yaml
service "nginx" created
statefulset "web" created

# 查看创建的 headless service 和 statefulset
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     None                 80/TCP    1m
$ kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web       2         2         2m

# 根据 volumeClaimTemplates 自动创建 PVC（在 GCE 中会自动创建 kubernete
$ kubectl get pvc
NAME        STATUS    VOLUME
www-web-0   Bound     pvc-d064a004-d8d4-11e6-b521-42010a800002
www-web-1   Bound     pvc-d06a3946-d8d4-11e6-b521-42010a800002

# 查看创建的 Pod，他们都是有序的
$ kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          5m
web-1     1/1       Running   0          4m

# 使用 nslookup 查看这些 Pod 的 DNS
$ kubectl run -i --tty --image busybox dns-test --restart=Never -
/ # nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.2.10
/ # nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.3.12
/ # nslookup web-0.nginx.default.svc.cluster.local
Server:    10.0.0.10
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx.default.svc.cluster.local
Address 1: 10.244.2.10
```

还可以进行其他的操作

```
# 扩容
$ kubectl scale statefulset web --replicas=5

# 缩容
$ kubectl patch statefulset web -p '{"spec":{"replicas":3}}'

# 镜像更新（目前还不支持直接更新 image，需要 patch 来间接实现）
$ kubectl patch statefulset web --type='json' -p='[{"op":"replace

# 删除 StatefulSet 和 Headless Service
$ kubectl delete statefulset web
$ kubectl delete service nginx

# StatefulSet 删除后 PVC 还会保留着，数据不再使用的话也需要删除
$ kubectl delete pvc www-web-0 www-web-1
```

更新 StatefulSet

v1.7 + 支持 StatefulSet 的自动更新，通过 `spec.updateStrategy` 设置更新策略。目前支持两种策略

- OnDelete：当 `.spec.template` 更新时，并不立即删除旧的 Pod，而是等待用户手动删除这些旧 Pod 后自动创建新 Pod。这是默认的更新策略，兼容 v1.6 版本的行为
- RollingUpdate：当 `.spec.template` 更新时，自动删除旧的 Pod 并创建新 Pod 替换。在更新时，这些 Pod 是按逆序的方式进行，依次删除、创建并等待 Pod 变成 Ready 状态才进行下一个 Pod 的更新。

Partitions

RollingUpdate 还支持 Partitions，通过 `.spec.updateStrategy.rollingUpdate.partition` 来设置。当 partition 设置后，只有序号大于或等于 partition 的 Pod 会在 `.spec.template` 更新的时候滚动更新，而其余的 Pod 则保持不变（即便是删除后也是用以前的版本重新创建）。

```
# 设置 partition 为 3
$ kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"t
statefulset "web" patched

# 更新 StatefulSet
$ kubectl patch statefulset web --type='json' -p='[{"op":"replace
statefulset "web" patched

# 验证更新
$ kubectl delete po web-2
pod "web-2" deleted
$ kubectl get po -lapp=nginx -w
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          4m
web-1     1/1       Running             0          4m
web-2     0/1       ContainerCreating   0          11s
web-2     1/1       Running             0          18s
```

Pod 管理策略

v1.7 + 可以通过 `.spec.podManagementPolicy` 设置 Pod 管理策略，支持两种方式

- OrderedReady：默认的策略，按照 Pod 的次序依次创建每个 Pod 并等待 Ready 之后才创建后面的 Pod
- Parallel：并行创建或删除 Pod（不等待前面的 Pod Ready 就开始创建所有的 Pod）

Parallel 示例

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  podManagementPolicy: "Parallel"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
```

```
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

可以看到，所有 Pod 是并行创建的

```
$ kubectl create -f webp.yaml
service "nginx" created
statefulset "web" created

$ kubectl get po -lapp=nginx -w
NAME      READY     STATUS              RESTARTS  AGE
web-0     0/1       Pending             0         0s
web-0     0/1       Pending             0         0s
web-1     0/1       Pending             0         0s
web-1     0/1       Pending             0         0s
web-0     0/1       ContainerCreating   0         0s
web-1     0/1       ContainerCreating   0         0s
web-0     1/1       Running             0         10s
web-1     1/1       Running             0         10s
```

zookeeper

另外一个更能说明 StatefulSet 强大功能的示例为 zookeeper.yaml。

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-headless
  labels:
    app: zk-headless
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
  clusterIP: None
  selector:
    app: zk
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: zk-config
data:
  ensemble: "zk-0;zk-1;zk-2"
  jvm.heap: "2G"
  tick: "2000"
  init: "10"
  sync: "5"
  client.cnxns: "60"
  snap.retain: "3"
  purge.interval: "1"
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-budget
spec:
  selector:
    matchLabels:
      app: zk
  minAvailable: 2
```

```
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: zk
spec:
  serviceName: zk-headless
  replicas: 3
  template:
    metadata:
      labels:
        app: zk
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
        scheduler.alpha.kubernetes.io/affinity: >
            {
              "podAntiAffinity": {
                "requiredDuringSchedulingRequiredDuringExecution"
                  "labelSelector": {
                    "matchExpressions": [{
                      "key": "app",
                      "operator": "In",
                      "values": ["zk-headless"]
                    }]
                  },
                  "topologyKey": "kubernetes.io/hostname"
                }]
              }
            }
    spec:
      containers:
      - name: k8szk
        imagePullPolicy: Always
        image: gcr.io/google_samples/k8szk:v1
        resources:
          requests:
            memory: "4Gi"
            cpu: "1"
        ports:
        - containerPort: 2181
```

```
          name: client
        - containerPort: 2888
          name: server
        - containerPort: 3888
          name: leader-election
        env:
        - name : ZK_ENSEMBLE
          valueFrom:
            configMapKeyRef:
              name: zk-config
              key: ensemble
        - name : ZK_HEAP_SIZE
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: jvm.heap
        - name : ZK_TICK_TIME
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: tick
        - name : ZK_INIT_LIMIT
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: init
        - name : ZK_SYNC_LIMIT
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: tick
        - name : ZK_MAX_CLIENT_CNXNS
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: client.cnxns
        - name: ZK_SNAP_RETAIN_COUNT
          valueFrom:
            configMapKeyRef:
                name: zk-config
```

```
                key: snap.retain
        - name: ZK_PURGE_INTERVAL
          valueFrom:
            configMapKeyRef:
                name: zk-config
                key: purge.interval
        - name: ZK_CLIENT_PORT
          value: "2181"
        - name: ZK_SERVER_PORT
          value: "2888"
        - name: ZK_ELECTION_PORT
          value: "3888"
        command:
        - sh
        - -c
        - zkGenConfig.sh && zkServer.sh start-foreground
        readinessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
        volumeMounts:
        - name: datadir
          mountPath: /var/lib/zookeeper
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
  volumeClaimTemplates:
  - metadata:
      name: datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
```

```
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 20Gi
```

```
kubectl create -f zookeeper.yaml
```

详细的使用说明见 zookeeper stateful application。

StatefulSet 注意事项

1. 推荐在 Kubernetes v1.9 或以后的版本中使用
2. 所有 Pod 的 Volume 必须使用 PersistentVolume 或者是管理员事先创建好
3. 为了保证数据安全，删除 StatefulSet 时不会删除 Volume
4. StatefulSet 需要一个 Headless Service 来定义 DNS domain，需要在 StatefulSet 之前创建好

ThirdPartyResources

ThirdPartyResources（TPR）是一种无需改变代码就可以扩展 Kubernetes API 的机制，可以用来管理自定义对象。每个 ThirdPartyResource 都包含以下属性

- metadata：跟 kubernetes metadata 一样
- kind：自定义的资源类型，采用 `.` 的格式
- description：资源描述
- versions：版本列表
- 其他：还可以保护任何其他自定义的属性

API 版本对照表

Kubernetes 版本	Extension 版本
v1.5-v1.7	extensions/v1beta1
v1.8+	不再支持

> **ThirdPartyResources 已在 v1.8 删除**
>
> ThirdPartyResources 已在 v1.8 版本中删除。建议从 v1.7 开始，迁移到 CustomResourceDefinition（CRD）。

TPR 示例

下面的例子会创建一个 `/apis/stable.example.com/v1/namespaces//crontabs/...` 的 API

```
$ cat resource.yaml
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: cron-tab.stable.example.com
description: "A specification of a Pod to run on a cron style sch
versions:
- name: v1

$ kubectl create -f resource.yaml
thirdpartyresource "cron-tab.stable.example.com" created
```

API 创建好后，就可以创建具体的 CronTab 对象了

```
$ cat my-cronjob.yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
cronSpec: "* * * * /5"
image: my-awesome-cron-image

$ kubectl create -f my-crontab.yaml
crontab "my-new-cron-object" created

$ kubectl get crontab
NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

ThirdPartyResources 与 RBAC

注意 ThirdPartyResources 不是 namespace-scoped 的资源，在普通用户使用之前需要绑定 ClusterRole 权限。

```
$ cat cron-rbac.yaml
apiVersion: rbac.authorization.k8s.io/v1alpha1
kind: ClusterRole
metadata:
  name: cron-cluster-role
rules:
- apiGroups:
  - extensions
  resources:
  - thirdpartyresources
  verbs:
  - '*'
- apiGroups:
  - stable.example.com
  resources:
  - crontabs
  verbs:
  - "*"

$ kubectl create -f cron-rbac.yaml
$ kubectl create clusterrolebinding user1 --clusterrole=cron-clus
```

迁移到 CustomResourceDefinition

1. 首先将 TPR 资源重定义为 CRD 资源，比如下面这个 ThirdPartyResource 资源

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: cron-tab.stable.example.com
description: "A specification of a Pod to run on a cron style sch
versions:
- name: v1
```

需要重新定义为

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  scope: Namespaced
  group: stable.example.com
  version: v1
  names:
    kind: CronTab
    plural: crontabs
    singular: crontab
```

1. 创建 CustomResourceDefinition 定义后，等待 CRD 的 Established 条件：

```
$ kubectl get crd -o 'custom-columns=NAME:{.metadata.name},ESTABL
NAME                          ESTABLISHED
crontabs.stable.example.com   True
```

1. 然后，停止使用 TPR 的客户端和 TPR Controller，启动新的 CRD Controller。
2. 备份数据

```
$ kubectl get crontabs --all-namespaces -o yaml > crontabs.yaml
$ kubectl get thirdpartyresource cron-tab.stable.example.com -o y
```

1. 删除 TPR 定义，TPR 资源会自动复制为 CRD 资源

```
$ kubectl delete thirdpartyresource cron-tab.stable.example.com
```

1. 验证 CRD 数据是否迁移成功，如果有失败发生，可以从备份的 TPR 数据恢复

```
$ kubectl create –f tpr.yaml
```

1. 重启客户端和相关的控制器或监听程序，它们的数据源会自动切换到 CRD（即访问 TPR 的 API 会自动转换为对 CRD 的访问）

Volume

我们知道默认情况下容器的数据都是非持久化的，在容器消亡以后数据也跟着丢失，所以 Docker 提供了 Volume 机制以便将数据持久化存储。类似的，Kubernetes 提供了更强大的 Volume 机制和丰富的插件，解决了容器数据持久化和容器间共享数据的问题。

与 Docker 不同，Kubernetes Volume 的生命周期与 Pod 绑定

- 容器挂掉后 Kubelet 再次重启容器时，Volume 的数据依然还在
- 而 Pod 删除时，Volume 才会清理。数据是否丢失取决于具体的 Volume 类型，比如 emptyDir 的数据会丢失，而 PV 的数据则不会丢

Volume 类型

目前，Kubernetes 支持以下 Volume 类型：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo
- secret
- persistentVolumeClaim

- downwardAPI
- azureFileVolume
- azureDisk
- vsphereVolume
- Quobyte
- PortworxVolume
- ScaleIO
- FlexVolume
- StorageOS
- local

注意，这些 volume 并非全部都是持久化的，比如 emptyDir、secret、gitRepo 等，这些 volume 会随着 Pod 的消亡而消失。

API 版本对照表

Kubernetes 版本	**Core API 版本**
v1.5+	core/v1

emptyDir

如果 Pod 设置了 emptyDir 类型 Volume， Pod 被分配到 Node 上时候，会创建 emptyDir，只要 Pod 运行在 Node 上，emptyDir 都会存在（容器挂掉不会导致 emptyDir 丢失数据），但是如果 Pod 从 Node 上被删除（Pod 被删除，或者 Pod 发生迁移），emptyDir 也会被删除，并且永久丢失。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath

hostPath 允许挂载 Node 上的文件系统到 Pod 里面去。如果 Pod 需要使用 Node 上的文件，可以使用 hostPath。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      path: /data
```

NFS

NFS 是 Network File System 的缩写，即网络文件系统。Kubernetes 中通过简单地配置就可以挂载 NFS 到 Pod 中，而 NFS 中的数据是可以永久保存的，同时 NFS 支持同时写操作。

```
volumes:
- name: nfs
  nfs:
    # FIXME: use the right hostname
    server: 10.254.234.223
    path: "/"
```

gcePersistentDisk

gcePersistentDisk 可以挂载 GCE 上的永久磁盘到容器，需要 Kubernetes 运行在 GCE 的 VM 中。

```
volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

awsElasticBlockStore

awsElasticBlockStore 可以挂载 AWS 上的 EBS 盘到容器，需要 Kubernetes 运行在 AWS 的 EC2 上。

```
volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: >
      fsType: ext4
```

gitRepo

gitRepo volume 将 git 代码下拉到指定的容器路径中

```
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

使用 subPath

Pod 的多个容器使用同一个 Volume 时，subPath 非常有用

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

FlexVolume

如果内置的这些 Volume 不满足要求，则可以使用 FlexVolume 实现自己的 Volume 插件。注意要把 volume plugin 放到 `/usr/libexec/kubernetes/kubelet-plugins/volume/exec//` ，plugin 要实现 `init/attach/detach/mount/umount` 等命令（可参考 lvm 的 示例）。

```
  - name: test
    flexVolume:
      driver: "kubernetes.io/lvm"
      fsType: "ext4"
      options:
        volumeID: "vol1"
        size: "1000m"
        volumegroup: "kube_vg"
```

Projected Volume

Projected volume 将多个 Volume 源映射到同一个目录中，支持 secret、downwardAPI 和 configMap。

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - downwardAPI:
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
```

本地存储限额

v1.7 + 支持对基于本地存储（如 hostPath, emptyDir, gitRepo 等）的容量进行调度限额，可以通过 `--feature-gates=LocalStorageCapacityIsolation=true` 来开启这个特性。

为了支持这个特性，Kubernetes 将本地存储分为两类

- `storage.kubernetes.io/overlay` ，即 `/var/lib/docker` 的大小
- `storage.kubernetes.io/scratch` ，即 `/var/lib/kubelet` 的大小

Kubernetes 根据 `storage.kubernetes.io/scratch` 的大小来调度本地存储空间，而根据 `storage.kubernetes.io/overlay` 来调度容器的存储。比如

为容器请求 64MB 的可写层存储空间

```
apiVersion: v1
kind: Pod
metadata:
  name: ls1
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: busybox
    command: ["df"]
    resources:
      requests:
        storage.kubernetes.io/overlay: 64Mi
```

为 empty 请求 64MB 的存储空间

```
apiVersion: v1
kind: Pod
metadata:
  name: ls1
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: busybox
    command: ["df"]
    volumeMounts:
    - name: data
      mountPath: /data
  volumes:
  - name: data
    emptyDir:
      sizeLimit: 64Mi
```

Mount 传递

在 Kubernetes 中，Volume Mount 默认是 私有的，但从 v1.8 开始，Kubernetes 支持配置 Mount 传递（mountPropagation）。它支持两种选项

- HostToContainer：这是开启 `MountPropagation=true` 时的默认模式，等效于 `rslave` 模式，即容器可以看到 Host 上面在该 volume 内的任何新 Mount 操作
- Bidirectional：等效于 `rshared` 模式，即 Host 和容器都可以看到对方在该 Volume 内的任何新 Mount 操作。该模式要求容器必须运行在特权模式（即 `securityContext.privileged=true` ）

注意：

- 使用 Mount 传递需要开启 `--feature-gates=MountPropagation=true`
- `rslave` 和 `rshared` 的说明可以参考 内核文档

Volume 快照

v1.8 新增了 pre-alpha 版本的 Volume 快照，但还只是一个雏形，并且其实现不在 Kubernetes 核心代码中，而是存放在 kubernetes-incubator/external-storage 中。

> TODO: 补充 Volume 快照的设计原理和示例。

Windows Volume

Windows 容器暂时只支持 local、emptyDir、hostPath、AzureDisk、AzureFile 以及 flexvolume。注意 Volume 的路径格式需要为 `mountPath: "C:\\etc\\foo"` 或者 `mountPath: "C:/etc/foo"` 。

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  containers:
  - name: hostpath-nano
    image: microsoft/nanoserver:1709
    stdin: true
    tty: true
    volumeMounts:
    - name: blah
      mountPath: "C:\\etc\\foo"
      readOnly: true
  nodeSelector:
    beta.kubernetes.io/os: windows
  volumes:
  - name: blah
    hostPath:
      path: "C:\\AzureData"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
  containers:
  - image: microsoft/nanoserver:1709
    name: empty-dir-nano
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: C:/scratch
      name: scratch-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
  - name: scratch-volume
    emptyDir: {}
  nodeSelector:
    beta.kubernetes.io/os: windows
```

挂载传播

挂载传播（MountPropagation）是 v1.9 引入的新功能，并在 v1.10 中升级为 Beta 版本。挂载传播用来解决同一个 Volume 在不同的容器甚至是 Pod 之间挂载的问题。通过设置 `Container.volumeMounts.mountPropagation），可以为该存储卷设置不同的传播类型。

支持三种选项：

- None：即私有挂载（private）
- HostToContainer：即 Host 内在该目录中的新挂载都可以在容器中看到，等价于 Linux 内核的 rslave。
- Bidirectional：即 Host 内在该目录中的新挂载都可以在容器中看到，同样容器内在该目录中的任何新挂载也都可以在 Host 中看到，等价于 Linux 内核的 rshared。仅特权容器（privileged）可以使用 Bidirectional 类型。

注意：

- 使用前需要开启 MountPropagation 特性
- 如未设置，则 v1.9 和 v1.10 中默认为私有挂载（`None`），而 v1.11 中默认为 `HostToContainer`
- Docker 服务的 systemd 配置文件中需要设置 `MountFlags=shared`

其他的 Volume 参考示例

https://github.com/kubernetes/examples/tree/master/staging/volumes/iscsi

- iSCSI Volume 示例
- cephfs Volume 示例
- Flocker Volume 示例
- GlusterFS Volume 示例
- RBD Volume 示例
- Secret Volume 示例
- downwardAPI Volume 示例
- AzureFile Volume 示例
- AzureDisk Volume 示例
- Quobyte Volume 示例
- Portworx Volume 示例
- ScaleIO Volume 示例
- StorageOS Volume 示例

部署配置

部署指南

本章介绍创建的 Kubernetes 集群部署方法、 kubectl 客户端的安装方法以及推荐的配置。

其中 Kubernetes-The-Hard-Way 介绍了在 GCE 的 Ubuntu 虚拟机中一步步部署一套 Kubernetes 高可用集群的详细步骤，这些步骤也同样适用于 CentOS 等其他系统以及 AWS、Azure 等其他公有云平台。

在国内部署集群时，通常还会碰到镜像无法拉取或者拉取过慢的问题。对这类问题的解决方法就是使用国内的镜像，具体可以参考国内镜像列表。

一般部署完成后，还需要运行一系列的测试来验证部署是成功的。sonobuoy 可以简化这个验证的过程，它通过一系列的测试来验证集群的功能是否正常。其使用方法为

- 通过 Sonobuoy Scanner tool 在线使用（需要集群公网可访问）
- 或者使用命令行工具

```
# Install
$ go get -u -v github.com/heptio/sonobuoy

# Run
$ sonobuoy run
$ sonobuoy status
$ sonobuoy logs
$ sonobuoy retrieve .

# Cleanup
$ sonobuoy delete
```

版本依赖

依赖组件	v1.13	v1.12
Etcd	v3.2.24+或v3.3.0+	v3.2.24+ 或 v3.3.0+ etcd2弃用
Docker	1.11.1, 1.12.1, 1.13.1, 17.03, 17.06, 17.09, 18.06	1.11.1, 1.12.1, 1.13.1, 17.03, 17.06, 17.09, 18.06
Go	1.11.2	1.10.4
CNI	v0.6.0	v0.6.0
CSI	1.0.0	0.3.0
Dashboard	v1.10.0	v1.8.3
Heapster	Remains v1.6.0-beta but retired	v1.6.0-beta
Cluster Autoscaler	v1.13.0	v1.12.0
kube-dns	v1.14.13	v1.14.13
Influxdb	v1.3.3	v1.3.3
Grafana	v4.4.3	v4.4.3
Kibana	v6.3.2	v6.3.2
cAdvisor	v0.32.0	v0.30.1
Fluentd	v1.2.4	v1.2.4
Elasticsearch	v6.3.2	v6.3.2
go-oidc	v2.0.0	v2.0.0
calico	v3.3.1	v2.6.7
crictl	v1.12.0	v1.12.0
CoreDNS	v1.2.6	v1.2.2
event-exporter	v0.2.3	v0.2.3
metrics-server	v0.3.1	v0.3.1
ingress-gce	v1.2.3	v1.2.3

依赖组件	v1.13	v1.12
ingress-nginx	v0.21.0	v0.21.0
ip-masq-agent	v2.1.1	v2.1.1
hcsshim	v0.6.11	v0.6.11

部署方法

- 1. 单机部署
- 2. 集群部署
  - kubeadm
  - kops
  - Kubespray
  - Azure
  - Windows
  - LinuxKit
  - Frakti
  - kubeasz
- 3. Kubernetes-The-Hard-Way
  - 准备部署环境
  - 安装必要工具
  - 创建计算资源
  - 配置创建证书
  - 配置生成配置
  - 配置生成密钥
  - 部署Etcd群集
  - 部署控制节点
  - 部署计算节点
  - 配置Kubectl
  - 配置网络路由
  - 部署DNS扩展
  - 烟雾测试
  - 删除集群
- 4. kubectl客户端
- 5. 附加组件
  - Dashboard
  - Heapster
  - EFK

    - Metrics
    - Cluster AutoScaler
- 6. 推荐配置
- 7. 版本支持

kubectl安装

本章介绍 kubectl 的安装方法。

安装方法

OSX

可以使用 Homebrew 或者 `curl` 下载 kubectl：

```
brew install kubectl
```

或者

```
curl -LO https://storage.googleapis.com/kubernetes-release/releas
```

Linux

```
curl -LO https://storage.googleapis.com/kubernetes-release/releas
```

Windows

```
curl -LO https://storage.googleapis.com/kubernetes-release/releas
```

或者使用 Chocolatey 来安装：

```
choco install kubernetes-cli
```

使用方法

kubectl 的详细使用方法请参考 kubectl 指南。

kubectl 插件

你可以使用 krew 来管理 kubectl 插件。

krew 是一个用来管理 kubectl 插件的工具，类似于 apt 或 yum，支持搜索、安装和管理 kubectl 插件。

安装

```
(
  set -x; cd "$(mktemp -d)" &&
  curl -fsSLO "https://storage.googleapis.com/krew/v0.2.1/krew.{t
  tar zxvf krew.tar.gz &&
  ./krew-"$(uname | tr '[:upper:]' '[:lower:]')_amd64" install \
    --manifest=krew.yaml --archive=krew.tar.gz
)
```

安装完成后，把 krew 的二进制文件加入环境变量 PATH 中：

```
export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
```

最后，再执行 kubectl 命令确认安装成功：

```
$ kubectl plugin list
The following kubectl-compatible plugins are available:

/home//.krew/bin/kubectl-krew
```

使用方法

首次使用前，请执行下面的命令更新插件索引：

```
kubectl krew update
```

使用示例：

```
kubectl krew search               # show all plugins
kubectl krew install ssh-jump  # install a plugin named "ssh-jump
kubectl ssh-jump               # use the plugin
kubectl krew upgrade              # upgrade installed plugins
kubectl krew remove ssh-jump   # uninstall a plugin
```

在安装插件后，会输出插件所依赖的外部工具，这些工具需要你自己手动安装。

```
Installing plugin: ssh-jump
CAVEATS:
\
 |  This plugin needs the following programs:
 |  * ssh(1)
 |  * ssh-agent(1)
 |
 |  Please follow the documentation: https://github.com/yokawasa/
/
Installed plugin: ssh-jump
```

最后，就可以通过 `kubectl` 来使用插件了：

```
kubectl ssh-jump  -u  -i ~/.ssh/id_rsa -p ~/.ssh/id_rsa.pub
```

升级方法

```
kubectl krew upgrade
```

参考文档

- https://github.com/GoogleContainerTools/krew

单机部署

minikube

创建 Kubernetes cluster（单机版）最简单的方法是 minikube。国内网络环境下也可以考虑使用 kubeasz 的 AllInOne 部署。

首先下载 kubectl

```
curl -LO https://storage.googleapis.com/kubernetes-release/releas
chmod +x kubectl
```

安装 minikube（以 MacOS 为例）

```
# install minikube
$ brew cask install minikube
$ curl -LO https://storage.googleapis.com/minikube/releases/lates
$ sudo install -o root -g wheel -m 4755 docker-machine-driver-hyp
```

在 Windows 上面

```
choco install minikube
choco install kubernetes-cli
```

最后启动 minikube

```
# start minikube.
# http proxy is required in China
$ minikube start --docker-env HTTP_PROXY=http://proxy-ip:port --d
```

使用 calico

minikube 支持配置使用 CNI 插件，这样可以方便的使用社区提供的各种网络插件，比如使用 calico 还可以支持 Network Policy。

首先使用下面的命令启动 minikube：

```
minikube start --docker-env HTTP_PROXY=http://proxy-ip:port \
    --docker-env HTTPS_PROXY=http://proxy-ip:port \
    --network-plugin=cni \
    --host-only-cidr 172.17.17.1/24 \
    --extra-config=kubelet.ClusterCIDR=192.168.0.0/16 \
    --extra-config=proxy.ClusterCIDR=192.168.0.0/16 \
    --extra-config=controller-manager.ClusterCIDR=192.168.0.0/16
```

安装 calico 网络插件：

```
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
curl -O -L https://docs.projectcalico.org/v3.1/getting-started/ku
sed -i -e '/nodeSelector/d' calico.yaml
sed -i -e '/node-role.kubernetes.io\/master:""/d' calico.yaml
sed -i -e 's/10\.96\.232/10.0.0/' calico.yaml
kubectl apply -f calico.yaml
```

开发版

minikube/localkube 只提供了正式 release 版本，而如果想要部署 master 或者开发版的话，则可以用 `hack/local-up-cluster.sh` 来启动一个本地集群：

```
cd $GOPATH/src/k8s.io/kubernetes

export KUBERNETES_PROVIDER=local
hack/install-etcd.sh
export PATH=$GOPATH/src/k8s.io/kubernetes/third_party/etcd:$PATH
hack/local-up-cluster.sh
```

打开另外一个终端，配置 kubectl：

```
cd $GOPATH/src/k8s.io/kubernetes
export KUBECONFIG=/var/run/kubernetes/admin.kubeconfig
cluster/kubectl.sh
```

或者，使用 kind，以 Docker 容器的方式运行 Kubernetes 集群：

```
$ go get sigs.k8s.io/kind
# ensure that Kubernetes is cloned in $(go env GOPATH)/src/k8s.io
# build a node image
$ kind build node-image
# create a cluster with kind build node-image
$ kind create cluster --image kindest/node:latest
```

参考文档

- Running Kubernetes Locally via Minikube
- https://github.com/kubernetes-sigs/kind

特性开关

特性开关（Feature Gates）是 Kubernetes 中用来开启实验性功能的配置，可以通过选项 `--feature-gates` 来给不同的组件（如 kube-apiserver、kube-controller-manager、kube-scheduler、kubelet、kube-proxy等）开启功能特性。

Feature	Default	Stage
`Accelerators`	`false`	Alpha
`AdvancedAuditing`	`false`	Alpha
`AdvancedAuditing`	`true`	Beta
`AdvancedAuditing`	`true`	GA
`AffinityInAnnotations`	`false`	Alpha
`AllowExtTrafficLocalEndpoints`	`false`	Beta
`AllowExtTrafficLocalEndpoints`	`true`	GA
`APIListChunking`	`false`	Alpha
`APIListChunking`	`true`	Beta
`APIResponseCompression`	`false`	Alpha
`AppArmor`	`true`	Beta
`AttachVolumeLimit`	`false`	Alpha
`BlockVolume`	`false`	Alpha
`CPUManager`	`false`	Alpha
`CPUManager`	`true`	Beta
`CRIContainerLogRotation`	`false`	Alpha
`CRIContainerLogRotation`	`true`	Beta
`CSIBlockVolume`	`false`	Alpha
`CSIPersistentVolume`	`false`	Alpha
`CSIPersistentVolume`	`true`	Beta
`CustomPodDNS`	`false`	Alpha
`CustomPodDNS`	`true`	Beta
`CustomResourceSubresources`	`false`	Alpha
`CustomResourceValidation`	`false`	Alpha
`CustomResourceValidation`	`true`	Beta

Feature	Default	Stage
`DebugContainers`	`false`	Alpha
`DevicePlugins`	`false`	Alpha
`DevicePlugins`	`true`	Beta
`DynamicKubeletConfig`	`false`	Alpha
`DynamicKubeletConfig`	`true`	Beta
`DynamicProvisioningScheduling`	`false`	Alpha
`DynamicVolumeProvisioning`	`true`	Alpha
`DynamicVolumeProvisioning`	`true`	GA
`EnableEquivalenceClassCache`	`false`	Alpha
`ExpandInUsePersistentVolumes`	`false`	Alpha
`ExpandPersistentVolumes`	`false`	Alpha
`ExpandPersistentVolumes`	`true`	Beta
`ExperimentalCriticalPodAnnotation`	`false`	Alpha
`ExperimentalHostUserNamespaceDefaulting`	`false`	Beta
`GCERegionalPersistentDisk`	`true`	Beta
`HugePages`	`false`	Alpha
`HugePages`	`true`	Beta
`HyperVContainer`	`false`	Alpha
`Initializers`	`false`	Alpha
`KubeletConfigFile`	`false`	Alpha
`KubeletPluginsWatcher`	`false`	Alpha
`KubeletPluginsWatcher`	`true`	Beta
`LocalStorageCapacityIsolation`	`false`	Alpha
`LocalStorageCapacityIsolation`	`true`	Beta
`MountContainers`	`false`	Alpha

Feature	Default	Stage
`MountPropagation`	`false`	Alpha
`MountPropagation`	`true`	Beta
`MountPropagation`	`true`	GA
`PersistentLocalVolumes`	`false`	Alpha
`PersistentLocalVolumes`	`true`	Beta
`PodPriority`	`false`	Alpha
`PodReadinessGates`	`false`	Alpha
`PodReadinessGates`	`true`	Beta
`PodShareProcessNamespace`	`false`	Alpha
`PodShareProcessNamespace`	`true`	Beta
`PVCProtection`	`false`	Alpha
`ReadOnlyAPIDataVolumes`	`true`	Deprecated
`ResourceLimitsPriorityFunction`	`false`	Alpha
`RotateKubeletClientCertificate`	`true`	Beta
`RotateKubeletServerCertificate`	`false`	Alpha
`RunAsGroup`	`false`	Alpha
`RuntimeClass`	`false`	Alpha
`SCTPSupport`	`false`	Alpha
`ServiceNodeExclusion`	`false`	Alpha
`StorageObjectInUseProtection`	`true`	Beta
`StorageObjectInUseProtection`	`true`	GA
`StreamingProxyRedirects`	`true`	Beta
`SupportIPVSProxyMode`	`false`	Alpha
`SupportIPVSProxyMode`	`false`	Beta
`SupportIPVSProxyMode`	`true`	Beta

Feature	Default	Stage
`SupportIPVSProxyMode`	`true`	GA
`SupportPodPidsLimit`	`false`	Alpha
`Sysctls`	`true`	Beta
`TaintBasedEvictions`	`false`	Alpha
`TaintNodesByCondition`	`false`	Alpha
`TaintNodesByCondition`	`true`	Beta
`TokenRequest`	`false`	Alpha
`TokenRequest`	`True`	Beta
`TokenRequestProjection`	`false`	Alpha
`TokenRequestProjection`	`True`	Beta
`TTLAfterFinished`	`false`	Alpha
`VolumeScheduling`	`false`	Alpha
`VolumeScheduling`	`true`	Beta
`VolumeSubpathEnvExpansion`	`false`	Alpha
`ScheduleDaemonSetPods`	`true`	Beta

参考文档

- Kubernetes Feature Gates

最佳配置

本文档旨在汇总和强调用户指南、快速开始文档和示例中的最佳实践。该文档会很很活跃并持续更新中。如果你觉得很有用的最佳实践但是本文档中没有包含，欢迎给我们提 Pull Request。

通用配置建议

- 定义配置文件的时候，指定最新的稳定 API 版本。

- 在部署配置文件到集群之前应该保存在版本控制系统中。这样当需要的时候能够快速回滚，必要的时候也可以快速的创建集群。
- 使用 YAML 格式而不是 JSON 格式的配置文件。在大多数场景下它们都可以互换，但是 YAML 格式比 JSON 更友好。
- 尽量将相关的对象放在同一个配置文件里，这样比分成多个文件更容易管理。参考 guestbook-all-in-one.yaml 文件中的配置。
- 使用 `kubectl` 命令时指定配置文件目录。
- 不要指定不必要的默认配置，这样更容易保持配置文件简单并减少配置错误。
- 将资源对象的描述放在一个 annotation 中可以更好的内省。

裸奔的 Pods vs Replication Controllers 和 Jobs

- 如果有其他方式替代“裸奔的 pod”（如没有绑定到 replication controller 上的 pod），那么就使用其他选择。
- 在 node 节点出现故障时，裸奔的 pod 不会被重新调度。
- Replication Controller 总是会重新创建 pod，除了明确指定了 `restartPolicy: Never` 的场景。Job 对象也适用。

Services

- 通常最好在创建相关的 replication controllers 之前先创建 service。这样可以保证容器在启动时就配置了该服务的环境变量。对于新的应用，推荐通过服务的 DNS 名字来访问（而不是通过环境变量）。
- 除非有必要（如运行一个 node daemon），不要使用配置 `hostPort` 的 Pod（用来指定暴露在主机上的端口号）。当你给 Pod 绑定了一个 `hostPort`，该 Pod 会因为端口冲突很难调度。如果是为了调试目的来通过端口访问的话，你可以使用 kubectl proxy and apiserver proxy 或者 kubectl port-forward。你可使用 Service 来对外暴露服务。如果你确实需要将 pod 的端口暴露到主机上，考虑使用 NodePort service。
- 跟 `hostPort` 一样的原因，避免使用 `hostNetwork`。
- 如果你不需要 kube-proxy 的负载均衡的话，可以考虑使用使用 headless services（ClusterIP 为 None）。

使用 Label

- 使用 labels 来指定应用或 Deployment 的语义属性。这样可以让你能够选择合适于场景的对象组，比如 `app: myapp, tire: frontend, phase: test, deployment: v3` 。
- 一个 service 可以被配置成跨越多个 deployment，只需要在它的 label selector 中简单的省略发布相关的 label。
- 注意 Deployment 对象不需要再管理 replication controller 的版本名。Deployment 中描述了对象的期望状态，如果对 spec 的更改被应用了话，Deployment controller 会以控制的速率来更改实际状态到期望状态。
- 利用 label 做调试。因为 Kubernetes replication controller 和 service 使用 label 来匹配 pods，这允许你通过移除 pod 的相关label的方式将其从一个 controller 或者 service 中移除，而 controller 会创建一个新的 pod 来取代移除的 pod。这是一个很有用的方式，帮你在一个隔离的环境中调试之前的“活着的”pod。

容器镜像

- 默认容器镜像拉取策略是 `IfNotPresent`，当本地已存在该镜像的时候 Kubelet 不会再从镜像仓库拉取。如果你希望总是从镜像仓库中拉取镜像的话，在 yaml 文件中指定镜像拉取策略为 `Always`（ `imagePullPolicy: Always` ）或者指定镜像的 tag 为 `:latest` 。
- 如果你没有将镜像标签指定为 `:latest` ，例如指定为 `myimage:v1` ，当该标签的镜像进行了更新，kubelet 也不会拉取该镜像。你可以在每次镜像更新后都生成一个新的 tag（例如 `myimage:v2` ），在配置文件中明确指定该版本。
- 可以使用镜像的摘要（Digest）来保证容器总是使用同一版本的镜像。
- **注意：** 在生产环境下部署容器应该尽量避免使用 `:latest` 标签，因为这样很难追溯到底运行的是哪个版本以及发生故障时该如何回滚。

使用 kubectl

- 尽量使用 `kubectl create -f` 或 `kubectl apply -f` 。kubeclt 会自动查找该目录下的所有后缀名为 `.yaml` 、 `.yml` 和 `.json` 文件并将它们传递给 `create` 或 `apply` 命令。

- `kubectl get` 或 `kubectl delete` 时使用标签选择器可以批量操作一组对象。

- 使用 `kubectl run` 和 `expose` 命令快速创建只有单个容器的 Deployment 和 Service，如

```
kubectl run hello-world --replicas=2 --labels="run=load-balancer-example
kubectl expose deployment hello-world --type=NodePort --name=example-ser
kubectl get pods --selector="run=load-balancer-example" --output=wide
```

参考文档

- Configuration Best Practices

版本支持

版本支持

Kubernetes 版本的格式为 **x.y.z**，其中 x 是主版本号，y 是次版本号，而 z 则是修订版本。版本的格式遵循 Semantic Versioning ，即

- 主版本号：当你做了不兼容的 API 修改，
- 次版本号：当你做了向下兼容的功能性新增，
- 修订号：当你做了向下兼容的问题修正。 Kubernetes 项目只维护最新的三个次版本，每个版本都会放到不同的发布分支中维护。上游版本发现的严重缺陷以及安全修复等都会移植到这些发布分支中，这些分支由 patch release manager 来维护。 次版本一般是每三个月发布一次，所以每个发布分支一般会维护 9 个月。

不同组件的版本支持情况

在 Kubernetes 中，不同组件的版本并不要求完全一致，但不同版本的组件混合部署时也有一些最基本的限制。

kube-apiserver

在 highly-availabile (HA) clusters 集群中，kube-apiserver 的版本差不能超过一个次版本号。比如最新的 kube-apiserver 版本号为 1.13 时，其他 kube-apiserver 的版本只能是 1.13 或者 1.12。

kubelet

Kubelet 的版本不能高于 kube-apiserver 的版本，并且跟 kube-apiserver 相比，最多可以相差两个次版本号。比如：

- `kube-apiserver` 的版本是 **1.13**
- 相应的 `kubelet` 的版本为 **1.13**， **1.12**， and **1.11** 再比如，一个高可用的集群中：
- `kube-apiserver` 版本号为 **1.13** and **1.12**
- 相应的 `kubelet` 版本为 **1.12**， and **1.11** （ **1.13** 不支持，因为它比 kube-apiserver 的 **1.12** 高）

kube-controller-manager，kube-scheduler，and cloud-controller-manager

`kube-controller-manager`，`kube-scheduler`，和 `cloud-controller-manager` 不能高于 kube-apiserver 的版本。通常它们的版本应该跟 kube-apiserver 一致，不过也支持相差一个次版本号同时运行。比如：

- `kube-apiserver` 版本为 **1.13**
- 相应的 `kube-controller-manager`，`kube-scheduler`，和 `cloud-controller-manager` 版本为 **1.13** and **1.12** 再比如，一个高可用的集群中：
- `kube-apiserver` 版本为 **1.13** and **1.12**
- 相应的 `kube-controller-manager`，`kube-scheduler`，和 `cloud-controller-manager` 版本为 **1.12**（ **1.13** 不支持，因为它比 apiserver 的**1.12** 高）

kubectl

kubectl 可以跟 kube-apiserver 相差一个次版本号，比如：

- `kube-apiserver` 版本为 **1.13**
- 相应的 `kubectl` 版本为 **1.14**，**1.13** 和 **1.12**

版本升级顺序

当从 1.n 版本升级到 1.(n+1) 版本时，必须要遵循以下的升级顺序。

kube-apiserver

前提条件：

- 单节点集群中， kube-apiserver 的版本为 1.n；HA 集群中，kube-apiserver 版本为 1.n 或者 1.(n+1)。
- `kube-controller-manager`，`kube-scheduler` 以及 `cloud-controller-manager` 的版本都是 1.n。
- kubelet 的版本是 1.n 或者 1.(n-1)

- 已注册的注入控制 webhook 可以处理新版本的请求，比如 ValidatingWebhookConfiguration 和 MutatingWebhookConfiguration 已经更新为支持 1.(n+1) 版本中新引入的特性。

接下来就可以把 kube-apiserver 升级到 1.(n+1)了，不过要注意**版本升级时不可跳过次版本号**。

kube-controller-manager, kube-scheduler, and cloud-controller-manager

前提条件：

- kube-apiserver 已经升级到 1.(n+1) 版本。

接下来就可以把 `kube-controller-manager` ， `kube-scheduler` 和 `cloud-controller-manager` 都升级到 **1.(n+1)** 版本了。

kubelet

前提条件：

- kube-apiserver 已经升级到 1.(n+1) 版本。
- 升级过程中需要保证 kubelet 跟 kube-apiserver 最多只相差一个次版本号。

接下来就可以把 kubelet 升级到 1.(n+1)了。

参考文档

- Kubernetes Version and Version Skew Support Policy - Kubernetes

集群部署

Kubernetes 集群架构

etcd 集群

从 `https://discovery.etcd.io/new?size=3` 获取 token 后，把 `etcd.yaml` 放到每台机器的 `/etc/kubernetes/manifests/etcd.yaml`，并替换掉 `${DISCOVERY_TOKEN}`，`${NODE_NAME}` 和 `${NODE_IP}`，即可以由 kubelet 来启动一个 etcd 集群。

对于运行在 kubelet 外部的 etcd，可以参考 etcd clustering guide 来手动配置集群模式。

kube-apiserver

把 `kube-apiserver.yaml` 放到每台 Master 节点的 `/etc/kubernetes/manifests/`，并把相关的配置放到 `/srv/kubernetes/`，即可由 kubelet 自动创建并启动 apiserver：

- basic_auth.csv - basic auth user and password
- ca.crt - Certificate Authority cert
- known_tokens.csv - tokens that entities (e.g. the kubelet) can use to talk to the apiserver
- kubecfg.crt - Client certificate, public key
- kubecfg.key - Client certificate, private key
- server.cert - Server certificate, public key
- server.key - Server certificate, private key

apiserver 启动后，还需要为它们做负载均衡，可以使用云平台的弹性负载均衡服务或者使用 haproxy/lvs/nginx 等为 master 节点配置负载均衡。

另外，还可以借助 Keepalived、OSPF、Pacemaker 等来保证负载均衡节点的高可用。

注意：

- 大规模集群注意增加 `--max-requests-inflight`（默认 400）
- 使用 nginx 时注意增加 `proxy_timeout: 10m`

controller manager 和 scheduler

controller manager 和 scheduler 需要保证任何时刻都只有一个实例运行，需要一个选主的过程，所以在启动时要设置 `--leader-elect=true`，比如

```
kube-scheduler --master=127.0.0.1:8080 --v=2 --leader-elect=true
kube-controller-manager --master=127.0.0.1:8080 --cluster-cidr=10
```

把 `kube-scheduler.yaml` 和 `kube-controller-manager` 放到每台 Master 节点的 `/etc/kubernetes/manifests/`，并把相关的配置放到 `/srv/kubernetes/`，即可由 kubelet 自动创建并启动 kube-scheduler 和 kube-controller-manager。

kube-dns

kube-dns 可以通过 Deployment 的方式来部署，默认 kubeadm 会自动创建。但在大规模集群的时候，需要放宽资源限制，比如

```
dns_replicas: 6
dns_cpu_limit: 100m
dns_memory_limit: 512Mi
dns_cpu_requests 70m
dns_memory_requests: 70Mi
```

另外，也需要给 dnsmasq 增加资源，比如增加缓存大小到 10000，增加并发处理数量 `--dns-forward-max=1000` 等。

数据持久化

除了上面提到的这些配置，持久化存储也是高可用 Kubernetes 集群所必须的。

- 对于公有云上部署的集群，可以考虑使用云平台提供的持久化存储，比如 aws ebs 或者 gce persistent disk
- 对于物理机部署的集群，可以考虑使用 iSCSI、NFS、Gluster 或者 Ceph 等网络存储，也可以使用 RAID

Azure

在 Azure 上可以使用 AKS 或者 acs-engine 来部署 Kubernetes 集群，具体部署方法参考 这里。

GCE

在 GCE 上可以利用 cluster 脚本方便的部署集群：

```
# gce,aws,gke,azure-legacy,vsphere,openstack-heat,rackspace,libvi
export KUBERNETES_PROVIDER=gce
curl -sS https://get.k8s.io | bash
cd kubernetes
cluster/kube-up.sh
```

AWS

在 aws 上建议使用 kops 来部署。

物理机或虚拟机

在 Linux 物理机或虚拟机中，建议使用 kubeadm 或 kubespray 来部署 Kubernetes 集群。

kubeadm

> Kubernetes 一键部署脚本（使用 docker 运行时）

```
# on master
export USE_MIRROR=true #国内用户必须使用MIRROR
git clone https://github.com/feiskyer/ops
cd ops
kubernetes/install-kubernetes.sh
# 记住控制台输出的 TOEKN 和 MASTER 地址，在其他 Node 安装时会用到

# on all nodes
git clone https://github.com/feiskyer/ops
cd ops
# Setup token and CIDR first.
# replace this with yours.
export TOKEN="xxxx"
export MASTER_IP="x.x.x.x"
export CONTAINER_CIDR="10.244.2.0/24"

# Setup and join the new node.
./kubernetes/add-node.sh
```

以下是详细的 kubeadm 部署集群步骤。

初始化系统

所有机器都需要初始化 docker 和 kubelet。

ubuntu

```
# for ubuntu 16.04
apt-get update
apt-get install -y apt-transport-https ca-certificates curl softw
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key
add-apt-repository "deb https://download.docker.com/linux/$(. /et
apt-get update && apt-get install -y docker-ce=$(apt-cache madisc

apt-get update && apt-get install -y apt-transport-https curl
curl https://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg |
    cat </etc/apt/sources.list.d/kubernetes.list
deb https://mirrors.aliyun.com/kubernetes/apt/ kubernetes-xenial
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
```

CentOS

```
yum install -y docker
systemctl enable docker && systemctl start docker

cat < /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernete
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
EOF
setenforce 0
sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/selinux/conf
yum install -y kubelet kubeadm kubectl
systemctl enable kubelet && systemctl start kubelet
```

安装 master

```
# --api-advertise-addresses
# for flannel, setup --pod-network-cidr 10.244.0.0/16
kubeadm init --pod-network-cidr 10.244.0.0/16 --kubernetes-versio

# enable schedule pods on the master
export KUBECONFIG=/etc/kubernetes/admin.conf
kubectl taint nodes --all node-role.kubernetes.io/master:NoSchedu
```

如果需要修改 kubernetes 服务的配置选项，则需要创建一个 kubeadm 配置文件，其格式为

```
apiVersion: kubeadm.k8s.io/v1alpha3
kind: InitConfiguration
bootstrapTokens:
- token: "9a08jv.c0izixklcxtmnze7"
  description: "kubeadm bootstrap token"
  ttl: "24h"
- token: "783bde.3f89s0fje9f38fhf"
  description: "another bootstrap token"
  usages:
  - signing
  groups:
  - system:anonymous
nodeRegistration:
  name: "ec2-10-100-0-1"
  criSocket: "/var/run/dockershim.sock"
  taints:
  - key: "kubeadmNode"
    value: "master"
    effect: "NoSchedule"
  kubeletExtraArgs:
    cgroupDriver: "cgroupfs"
apiEndpoint:
  advertiseAddress: "10.100.0.1"
  bindPort: 6443
---
apiVersion: kubeadm.k8s.io/v1alpha3
kind: ClusterConfiguration
etcd:
  # one of local or external
  local:
    image: "k8s.gcr.io/etcd-amd64:3.2.18"
    dataDir: "/var/lib/etcd"
    extraArgs:
      listen-client-urls: "http://10.100.0.1:2379"
    serverCertSANs:
    -  "ec2-10-100-0-1.compute-1.amazonaws.com"
    peerCertSANs:
    - "10.100.0.1"
  external:
    endpoints:
```

```
    - "10.100.0.1:2379"
    - "10.100.0.2:2379"
    caFile: "/etcd/kubernetes/pki/etcd/etcd-ca.crt"
    certFile: "/etcd/kubernetes/pki/etcd/etcd.crt"
    certKey: "/etcd/kubernetes/pki/etcd/etcd.key"
networking:
  serviceSubnet: "10.96.0.0/12"
  podSubnet: "10.100.0.1/24"
  dnsDomain: "cluster.local"
kubernetesVersion: "v1.12.0"
controlPlaneEndpoint: "10.100.0.1:6443"
apiServerExtraArgs:
  authorization-mode: "Node,RBAC"
controlManagerExtraArgs:
  node-cidr-mask-size: 20
schedulerExtraArgs:
  address: "10.100.0.1"
apiServerExtraVolumes:
- name: "some-volume"
  hostPath: "/etc/some-path"
  mountPath: "/etc/some-pod-path"
  writable: true
  pathType: File
controllerManagerExtraVolumes:
- name: "some-volume"
  hostPath: "/etc/some-path"
  mountPath: "/etc/some-pod-path"
  writable: true
  pathType: File
schedulerExtraVolumes:
- name: "some-volume"
  hostPath: "/etc/some-path"
  mountPath: "/etc/some-pod-path"
  writable: true
  pathType: File
apiServerCertSANs:
- "10.100.1.1"
- "ec2-10-100-0-1.compute-1.amazonaws.com"
certificatesDir: "/etc/kubernetes/pki"
imageRepository: "k8s.gcr.io"
```

```
unifiedControlPlaneImage: "k8s.gcr.io/controlplane:v1.12.0"
auditPolicy:
  # https://kubernetes.io/docs/tasks/debug-application-cluster/au
  path: "/var/log/audit/audit.json"
  logDir: "/var/log/audit"
  logMaxAge: 7 # in days
featureGates:
  selfhosting: false
clusterName: "example-cluster"
```

> 注意：JoinConfiguration 重命名自 v1alpha2 API 中的 NodeConfiguration，而 InitConfiguration 重命名自 v1alpha2 API 中的 MasterConfiguration。

然后，在初始化 master 的时候指定 kubeadm.yaml 的路径：

```
kubeadm init --config ./kubeadm.yaml
```

配置 Network plugin

CNI bridge

```
mkdir -p /etc/cni/net.d
cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.0.0/16",
        "routes": [
            {"dst": "0.0.0.0/0"}
        ]
    }
}
EOF
cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "type": "loopback"
}
EOF
```

flannel

注意：需要 `kubeadm init` 时设置 `--pod-network-cidr=10.244.0.0/16`

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel
```

weave

```
sysctl net.bridge.bridge-nf-call-iptables=1
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$
```

calico

注意：需要 `kubeadm init` 时设置 `--pod-network-cidr=192.168.0.0/16`

```
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
```

添加 Node

```
kubeadm join --token  : --discovery-token-ca-cert-hash sha256:<ha
```

跟 Master 一样，添加 Node 的时候也可以自定义 Kubernetes 服务的选项，格式为

```
apiVersion: kubeadm.k8s.io/v1alpha2
caCertPath: /etc/kubernetes/pki/ca.crt
clusterName: kubernetes
discoveryFile: ""
discoveryTimeout: 5m0s
discoveryToken: abcdef.0123456789abcdef
discoveryTokenAPIServers:
- kube-apiserver:6443
discoveryTokenUnsafeSkipCAVerification: true
kind: NodeConfiguration
nodeRegistration:
  criSocket: /var/run/dockershim.sock
  name: thegopher
tlsBootstrapToken: abcdef.0123456789abcdef
token: abcdef.0123456789abcdef
```

在把 Node 加入集群的时候，指定 NodeConfiguration 配置文件的路径

```
kubeadm join --config ./nodeconfig.yml --token $token ${master_ip
```

Cloud Provider

默认情况下，kubeadm 不包括 Cloud Provider 的配置，在 Azure 或者 AWS 等云平台上运行时，还需要配置 Cloud Provider。如

```
kind: MasterConfiguration
apiVersion: kubeadm.k8s.io/v1alpha2
apiServerExtraArgs:
  cloud-provider: "{cloud}"
  cloud-config: "{cloud-config-path}"
apiServerExtraVolumes:
- name: cloud
  hostPath: "{cloud-config-path}"
  mountPath: "{cloud-config-path}"
controllerManagerExtraArgs:
  cloud-provider: "{cloud}"
  cloud-config: "{cloud-config-path}"
controllerManagerExtraVolumes:
- name: cloud
  hostPath: "{cloud-config-path}"
  mountPath: "{cloud-config-path}"
```

删除安装

```
# drain and delete the node first
kubectl drain  --delete-local-data --force --ignore-daemonsets
kubectl delete node

# then reset kubeadm
kubeadm reset
```

动态升级

kubeadm v1.8 开始支持动态升级，升级步骤为

- 首先上传 kubeadm 配置，如 `kubeadm config upload from-flags [flags]`（使用命令行参数）或 `kubeadm config upload from-file --config [config]`（使用配置文件）
- 在 master 上检查新版本 `kubeadm upgrade plan`， 当有新版本（如 v1.8.0）时，执行 `kubeadm upgrade apply v1.8.0` 升级控制平面
- 手动 升级 CNI 插件（如果有新版本的话）
- 添加自动证书回滚的 RBAC 策略 `kubectl create clusterrolebinding kubeadm:node-autoapprove-certificate-rotation --clusterrole=system:certificates.k8s.io:certificatesigningrequests:selfnodeclient --group=system:nodes`
- 最后升级 kubelet

```
$ kubectl drain $HOST --ignore-daemonsets

# 升级软件包
$ apt-get update
$ apt-get upgrade
# CentOS 上面执行 yum 升级
# $ yum update

$ kubectl uncordon $HOST
```

手动升级

kubeadm v1.7 以及以前的版本不支持动态升级，但可以手动升级。

升级 Master

假设你已经有一个使用 kubeadm 部署的 Kubernetes v1.6 集群，那么升级到 v1.7 的方法为：

1. 升级安装包 `apt-get upgrade && apt-get update`
2. 重启 kubelet `systemctl restart kubelet`

3. 删除 kube-proxy DaemonSet `KUBECONFIG=/etc/kubernetes/admin.conf kubectl delete daemonset kube-proxy -n kube-system`
4. `kubeadm init --skip-preflight-checks --kubernetes-version v1.7.1`
5. 更新 CNI 插件

升级 Node

1. 升级安装包 `apt-get upgrade && apt-get update`
2. 重启 kubelet `systemctl restart kubelet`

安全选项

默认情况下，kubeadm 会开启 Node 客户端证书的自动批准，如果不需要的话可以选择关闭，关闭方法为

```
$ kubectl delete clusterrole kubeadm:node-autoapprove-bootstrap
```

关闭后，增加新的 Node 时，`kubeadm join` 会阻塞等待管理员手动批准，匹配方法为

```
$ kubectl get csr
NAME                                                   AGE
node-csr-c69HXe7aYcqkS1bKmH4faEnHAWxn6i2bHZ2mD04jZyQ   18s

$ kubectl certificate approve node-csr-c69HXe7aYcqkS1bKmH4faEnHAW
certificatesigningrequest "node-csr-c69HXe7aYcqkS1bKmH4faEnHAWxn6

$ kubectl get csr
NAME                                                   AGE
node-csr-c69HXe7aYcqkS1bKmH4faEnHAWxn6i2bHZ2mD04jZyQ   1m
```

参考文档

- kubeadm 参考指南
- Upgrading kubeadm clusters from 1.7 to 1.8
- Upgrading kubeadm clusters from 1.6 to 1.7

kops

kops 是一个生产级 Kubernetes 集群部署工具，可以在 AWS、GCE、VMWare vSphere 等平台上自动部署高可用的 Kubernetes 集群。主要功能包括

- 自动部署高可用的 kubernetes 集群
- 支持从 kube-up 创建的集群升级到 kops 版本
- dry-run 和自动幂等升级等基于状态同步模型
- 支持自动生成 AWS CloudFormation 和 Terraform 配置
- 支持自定义扩展 add-ons
- 命令行自动补全

安装 kops 和 kubectl

```
# on macOS
brew install kubectl kops

# on Linux
wget https://github.com/kubernetes/kops/releases/download/1.7.0/k
chmod +x kops-linux-amd64
mv kops-linux-amd64 /usr/local/bin/kops
```

在 AWS 上面部署

首先需要安装 AWS CLI 并配置 IAM：

```
# install AWS CLI
pip install awscli

# configure iam
aws iam create-group --group-name kops
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/
aws iam create-user --user-name kops
aws iam add-user-to-group --user-name kops --group-name kops
aws iam create-access-key --user-name kops

# configure the aws client to use your new IAM user
aws configure           # Use your new access and secret key here
aws iam list-users      # you should see a list of all your IAM u

# Because "aws configure" doesn't export these vars for kops to u
export AWS_ACCESS_KEY_ID=
export AWS_SECRET_ACCESS_KEY=
```

创建 route53 域名

```
aws route53 create-hosted-zone --name dev.example.com --caller-re
```

创建 s3 存储 bucket

```
aws s3api create-bucket --bucket clusters.dev.example.com --regio
aws s3api put-bucket-versioning --bucket clusters.dev.example.com
```

部署 Kubernetes 集群

```
export KOPS_STATE_STORE=s3://clusters.dev.example.com

kops create cluster --zones=us-east-1c useast1.dev.example.com --
```

当然，也可以部署一个高可用的集群

```
kops create cluster \
    --node-count 3 \
    --zones us-west-2a,us-west-2b,us-west-2c \
    --master-zones us-west-2a,us-west-2b,us-west-2c \
    --node-size t2.medium \
    --master-size t2.medium \
    --topology private \
    --networking kopeio-vxlan \
    hacluster.example.com
```

删除集群

```
kops delete cluster --name ${NAME} --yes
```

在 GCE 上面部署

```
# Create cluster in GCE.
# This is an alpha feature.
export KOPS_STATE_STORE="gs://mybucket-kops"
export ZONES=${MASTER_ZONES:-"us-east1-b,us-east1-c,us-east1-d"}
export KOPS_FEATURE_FLAGS=AlphaAllowGCE

kops create cluster kubernetes-k8s-gce.example.com
  --zones $ZONES \
  --master-zones $ZONES \
  --node-count 3
  --project my-gce-project \
  --image "ubuntu-os-cloud/ubuntu-1604-xenial-v20170202" \
  --yes
```

Kubespray

Kubespray 是 Kubernetes incubator 中的项目，目标是提供 Production Ready Kubernetes 部署方案，该项目基础是通过 Ansible Playbook 来定义系统与 Kubernetes 集群部署的任务，具有以下几个特点：

- 可以部署在 AWS，GCE，Azure，OpenStack 以及裸机上.

- 部署 High Available Kubernetes 集群.
- 可组合性 (Composable), 可自行选择 Network Plugin (flannel, calico, canal, weave) 来部署.
- 支持多种 Linux distributions(CoreOS, Debian Jessie, Ubuntu 16.04, CentOS/RHEL7).

本篇将说明如何通过 Kubespray 部署 Kubernetes 至裸机节点，安装版本如下所示：

- Kubernetes v1.7.3
- Etcd v3.2.4
- Flannel v0.8.0
- Docker v17.04.0-ce

节点资讯

本次安装测试环境的作业系统采用 `Ubuntu 16.04 Server`，其他细节内容如下：

IP Address	Role	CPU	Memory
192.168.121.179	master1 + deploy	2	4G
192.168.121.106	node1	2	4G
192.168.121.197	node2	2	4G
192.168.121.123	node3	2	4G

> 这边 master 为主要控制节点，node 为工作节点。

预先准备资讯

- 所有节点的网路之间可以互相通信。
- `部署节点（这边为 master1）` 对其他节点不需要 SSH 密码即可登入。
- 所有节点都拥有 Sudoer 权限，并且不需要输入密码。
- 所有节点需要安装 `Python`。
- 所有节点需要设定 `/etc/hosts` 解析到所有主机。
- 修改所有节点的 `/etc/resolv.conf`

```
$ echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
```

- 部署节点（这边为 master1） 安装 Ansible >= 2.3.0。

Ubuntu 16.04 安装 Ansible:

```
$ sudo sed -i 's/us.archive.ubuntu.com/tw.archive.ubuntu.com/g' /
$ sudo apt-get install -y software-properties-common
$ sudo apt-add-repository -y ppa:ansible/ansible
$ sudo apt-get update && sudo apt-get install -y ansible git cows
```

安装 Kubespray 与准备部署资讯

首先通过 pypi 安装 kubespray-cli，虽然官方说已经改成 Go 语言版本的工具，但是根本没在更新，所以目前暂时用 pypi 版本：

```
$ sudo pip install -U kubespray
```

安装完成後，新增配置檔 ~/.kubespray.yml，並加入以下內容：

```
$ cat < ~/.kubespray.yml
kubespray_git_repo: "https://github.com/kubernetes-incubator/kube
# Logging options
loglevel: "info"
EOF
```

接着用 kubespray cli 来产生 inventory 文件：

```
$ kubespray prepare --masters master1 --etcds master1 --nodes noc
```

在 inventory.cfg，添加部分內容：

```
$ vim ~/.kubespray/inventory/inventory.cfg

[all]
master1  ansible_host=192.168.121.179   ansible_user=root ip=192.
node1    ansible_host=192.168.121.106 ansible_user=root ip=192.16
node2    ansible_host=192.168.121.197 ansible_user=root ip=192.16
node3    ansible_host=192.168.121.123 ansible_user=root ip=192.16

[kube-master]
master1

[kube-node]
node1
node2
node3

[etcd]
master1

[k8s-cluster:children]
kube-node
kube-master
```

> 也可以自己新建 `inventory` 来描述部署节点。

完成后通过以下指令进行部署 Kubernetes 集群：

```
$ time kubespray deploy --verbose -u root -k .ssh/id_rsa -n flanr
Run kubernetes cluster deployment with the above command ? [Y/n]y
...
master1                    : ok=368  changed=89   unreachable=0
node1                      : ok=305  changed=73   unreachable=0
node2                      : ok=276  changed=62   unreachable=0
node3                      : ok=276  changed=62   unreachable=0

Kubernetes deployed successfuly
```

> 其中 `-n` 为部署的网络插件类型，目前支持 calico、flannel、weave 与 canal。

验证集群

当 Ansible 运行完成后，若没发生错误就可以开始进行操作 Kubernetes，如取得版本资讯：

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1
```

取得当前集群节点状态：

```
$ kubectl get node
NAME      STATUS                     AGE       VERSION
master1   Ready,SchedulingDisabled   11m       v1.7.3+coreos.0
node1     Ready                      11m       v1.7.3+coreos.0
node2     Ready                      11m       v1.7.3+coreos.0
node3     Ready                      11m       v1.7.3+coreos.
```

查看当前集群 Pod 状态：

```
$ kubectl get po -n kube-system
NAME                                    READY     STATUS    RESTART
dnsmasq-975202658-6jj3n                 1/1       Running   0
dnsmasq-975202658-h4rn9                 1/1       Running   0
dnsmasq-autoscaler-2349860636-kfpx0     1/1       Running   0
flannel-master1                         1/1       Running   1
flannel-node1                           1/1       Running   1
flannel-node2                           1/1       Running   1
flannel-node3                           1/1       Running   1
kube-apiserver-master1                  1/1       Running   0
kube-controller-manager-master1         1/1       Running   0
kube-proxy-master1                      1/1       Running   1
kube-proxy-node1                        1/1       Running   1
kube-proxy-node2                        1/1       Running   1
kube-proxy-node3                        1/1       Running   1
kube-scheduler-master1                  1/1       Running   0
kubedns-1519522227-thmrh                3/3       Running   0
kubedns-autoscaler-2999057513-tx14j     1/1       Running   0
nginx-proxy-node1                       1/1       Running   1
nginx-proxy-node2                       1/1       Running   1
nginx-proxy-node3                       1/1       Running   1
```

Azure

Azure 容器服务（AKS）是 Microsoft Azure 最近发布的一个托管的 Kubernetes 服务（预览版），它独立于现有的 Azure Container Service （ACS）。借助 AKS 用户无需具备容器业务流程的专业知识就可以快速、轻松的部署和管理容器化的应用程序。AKS 支持自动升级和自动故障修复，按需自动扩展或缩放资源池，消除了用户管理和维护 Kubernetes 集群的负担。并且集群管理本身是免费的，Azure 只收取容器底层的虚拟机的费用。

ACS 是 Microsoft Azure 在 2015 年推出的容器服务，支持 Kubernetes、DCOS 以及 Dockers Swarm 等多种容器编排工具。并且 ACS 的核心功能是开源的，用户可以通过 https://github.com/Azure/acs-engine 来查看和下载使用。

AKS

基本使用

以下文档假设用户已经安装好了 Azure CLI ，如未安装可以参考 这里 操作。

在创建 AKS 集群之前，首先需要开启容器服务

```
# Enable AKS
az provider register -n Microsoft.ContainerService
```

然后创建一个资源组（Resource Group）用来管理所有相关资源

```
# Create Resource Group
az group create --name group1 --location centralus
```

接下来就可以创建 AKS 集群了

```
# Create aks
az aks create --resource-group group1 --name myK8sCluster --node-
```

稍等一会，集群创建好后安装并配置 kubectl

```
# Install kubectl
az aks install-cli

# Configure kubectl
az aks get-credentials --resource-group=group1 --name=myK8sCluste
```

> 注意使用 azure-cli 2.0.24 版本时， `az aks get-credentials` 命令可能会失败，解决方法是升级到更新版本，或回退到 2.0.23 版本。

访问 Dashboard

```
# Create dashboard
az aks browse --resource-group group1 --name myK8SCluster
```

手动扩展或收缩集群

```
az aks scale --resource-group=group1 --name=myK8SCluster --agent-
```

升级集群

```
# 查询当前集群的版本以及可升级的版本
az aks get-versions --name myK8sCluster --resource-group group1 -

# 升级到 1.11.3 版本
az aks upgrade --name myK8sCluster --resource-group group1 --kube
```

下图动态展示了一个部署 `v1.7.7` 版本集群并升级到 `v1.8.1` 的过程：

使用 Helm

当然也可以使用其他 Kubernetes 社区提供的工具和服务，比如使用 Helm 部署 Nginx Ingress 控制器

```
helm init --client-only
helm install stable/nginx-ingress
```

删除集群

当集群不再需要时，可以删除集群

```
az group delete --name group1 --yes --no-wait
```

acs-engine

虽然未来 AKS 是 Azure 容器服务的下一代主打产品，但用户可能还是希望可以自己管理容器集群以保证足够的灵活性（比如自定义 master 服务等）。这时用户可以使用开源的 acs-engine 来创建和管理自己的集群。acs-engine 其实就是 ACS 的核心部分，提供了一个部署和管理 Kubernetes、Swarm 和 DC/OS 集群的命令行工具。它通过将容器集群描述文件转化为一组 ARM（Azure Resource Manager）模板来建立容器集群。

在 acs-engine 中，每个集群都通过一个 json 文件来描述，比如一个 Kubernetes 集群可以描述为

```
{
  "apiVersion": "vlabs",
  "properties": {
    "orchestratorProfile": {
      "orchestratorType": "Kubernetes",
      "orchestratorRelease": "1.12",
      "kubernetesConfig": {
        "networkPolicy": "",
        "enableRbac": true
      }
    },
    "masterProfile": {
      "count": 1,
      "dnsPrefix": "",
      "vmSize": "Standard_D2_v2"
    },
    "agentPoolProfiles": [
      {
        "name": "agentpool1",
        "count": 3,
        "vmSize": "Standard_D2_v2",
        "availabilityProfile": "AvailabilitySet"
      }
    ],
    "linuxProfile": {
      "adminUsername": "azureuser",
      "ssh": {
        "publicKeys": [
          {
            "keyData": ""
          }
        ]
      }
    },
    "servicePrincipalProfile": {
      "clientId": "",
      "secret": ""
    }
  }
}
```

orchestratorType 指定了部署集群的类型，目前支持三种

- Kubernetes
- Swarm
- DCOS

而创建集群的步骤也很简单

```
# create a new resource group.
az group create --name myResourceGroup  --location "centralus"

# start deploy the kubernetes
acs-engine deploy --resource-group myResourceGroup --subscriptior

# setup kubectl
export KUBECONFIG="$(pwd)/_output//kubeconfig/kubeconfig.centralu
kubectl get node
```

开启 RBAC

RBAC 默认是不可以开启的，可以通过设置 enableRbac 开启

```
     "kubernetesConfig": {
        "enableRbac": true
      }
```

自定义 Kubernetes 版本

acs-engine 基于 hyperkube 来部署 Kubernetes 服务，所以只需要使用自定义的 hyperkube 镜像即可。

```
{
    "kubernetesConfig": {
        "customHyperkubeImage": "docker.io/feisky/hyperkube-amd64
    }
}
```

hyperkube 镜像可以从 Kubernetes 源码编译，编译步骤为

```
# Build Kubernetes
bash build/run.sh make KUBE_FASTBUILD=true ARCH=amd64

# Build docker image for hyperkube
cd cluster/images/hyperkube
make VERSION=v1.12.x-dev
cd ../../..

# push docker image
docker tag gcr.io/google-containers/hyperkube-amd64:v1.12.x-dev f
docker push feisky/hyperkube-amd64:v1.12.x-dev
```

添加 Windows 节点

可以通过设置 osType 来添加 Windows 节点（完整示例见 这里）

```
    "agentPoolProfiles": [
      {
        "name": "windowspool2",
        "count": 2,
        "vmSize": "Standard_D2_v2",
        "availabilityProfile": "AvailabilitySet",
        "osType": "Windows"
      }
    ],
    "windowsProfile": {
      "adminUsername": "azureuser",
      "adminPassword": "replacepassword1234$"
    },
```

使用 GPU

设置 vmSize 为 `Standard_NC*` 或 `Standard_NV*` 会自动配置 GPU，并自动安装所需要的 NVDIA 驱动。

自定义网络插件

acs-engine 默认使用 kubenet 网络插件，并通过用户自定义的路由以及 IP-forwarding 转发 Pod 网络。此时，Pod 网络与 Node 网络在不同的子网中，Pod 不受 VNET 管理。

用户还可以使用 Azure CNI plugin 插件将 Pod 连接到 Azure VNET 中

```
"properties": {
    "orchestratorProfile": {
      "orchestratorType": "Kubernetes",
      "kubernetesConfig": {
        "networkPolicy": "azure"
      }
    }
}
```

也可以使用 calico 网络插件

```
"properties": {
    "orchestratorProfile": {
      "orchestratorType": "Kubernetes",
      "kubernetesConfig": {
        "networkPolicy": "calico"
      }
    }
}
```

Azure Container Registry

在 AKS 预览版发布的同时，Azure 还同时发布了 Azure Container Registry（ACR）服务，用于托管用户的私有镜像。

```
# Create ACR
az acr create --resource-group myResourceGroup --name  --sku Basi

# Login
az acr login --name

# Tag the image.
az acr list --resource-group myResourceGroup --query "[].{acrLogi
docker tag azure-vote-front /azure-vote-front:redis-v1

# push image
docker push /azure-vote-front:redis-v1

# List images.
az acr repository list --name  --output table
```

Virtual Kubelet

Azure 容器实例（ACI）提供了在 Azure 中运行容器的最简捷方式，它不需要用户配置任何虚拟机或其它高级服务。ACI 适用于快速突发式增长和资源调整的业务，但其本身的功能相对比较简单。 Virtual Kubelet 可以将 ACI 作为 Kubernetes 集群的一个无限 Node 使用，这样就无需考虑 Node 数量的问题，ACI 会根据运行容器自动管理集群资源。

可以使用 Helm 来部署 Virtual Kubelet：

```
RELEASE_NAME=virtual-kubelet
CHART_URL=https://github.com/virtual-kubelet/virtual-kubelet/raw/

helm install "$CHART_URL" --name "$RELEASE_NAME" --namespace kube
```

在开启 RBAC 的集群中，还需要给 virtual-kubelet 开启对应的权限。最简单的方法是给 service account `kube-system:default` 设置 admin 权限（不推荐生产环境这么设置，应该设置具体的权限），比如

```
kubectl create clusterrolebinding virtual-kubelet-cluster-admin-b
```

部署成功后，会发现集群中会出现一个新的名为 `aci` 的 Node：

```
$ kubectl get nodes aci
NAME      STATUS    ROLES     AGE       VERSION
aci       Ready     agent     34s       v1.8.3
```

此时，就可以通过 **指定 nodeName 或者容忍 taint** `azure.com/aci=NoSchedule` **调度** 到 ACI 上面。比如

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      requests:
        memory: 100M
        cpu: 1
    ports:
    - containerPort: 80
      name: http
      protocol: TCP
    - containerPort: 443
      name: https
  dnsPolicy: ClusterFirst
  nodeName: aci
```

参考文档

- AKS - Managed Kubernetes on Azure
- Azure Container Service (AKS)
- Azure/acs-engine Github
- acs-engine/examples

Windows

Kubernetes 从 v1.5 开始支持 alpha 版的 Windows 节点，并从 v1.9 开始升级为 beta 版。Windows 容器的主要特性包括

- Windows 容器支持 Pod（isolation=process）
- 基于 Virtual Filtering Platform（VFP）Hyper-v Switch Extension 的内核负载均衡
- 基于 Container Runtime Interface（CRI）管理 Windows 容器
- 支持 kubeadm 命令将 Windows 节点加入到已有集群中
- 推荐使用 Windows Server Version 1803+ 和 Docker Version 17.06+

> 注意：
>
> 1. 控制平面的服务依然运行在 Linux 服务器中，而 Windows 节点上只运行 Kubelet、Kube-proxy、Docker 以及网络插件等服务。
> 2. 推荐使用 Windows Server 1803（修复了 Windows 容器软链接的问题，从而 ServiceAccount 和 ConfigMap 可以正常使用）

下载

可以从 下载已发布的用于 Windows 服务器的二进制文件，如

```
wget https://dl.k8s.io/v1.11.2/kubernetes-node-windows-amd64.tar.
```

或者从 Kubernetes 源码编译

```
go get -u k8s.io/kubernetes
cd $GOPATH/src/k8s.io/kubernetes

# Build the kubelet
KUBE_BUILD_PLATFORMS=windows/amd64 make WHAT=cmd/kubelet

# Build the kube-proxy
KUBE_BUILD_PLATFORMS=windows/amd64 make WHAT=cmd/kube-proxy

# You will find the output binaries under the folder _output/loca
```

网络插件

Windows Server 中支持以下几种网络插件（注意 Windows 节点上的网络插件要与 Linux 节点相同）

1. wincni 等 L3 路由网络插件，路由配置在 TOR 交换机、路由器或者云服务中
2. Host Gateway 网络插件，跟上面类似但将 IP 路由配置到每台主机上面
3. Azure VNET CNI Plugin
4. Open vSwitch (OVS) & Open Virtual Network (OVN) with Overlay
5. Flannel v0.10.0+
6. Calico v3.0.1+
7. win-bridge
8. win-overlay

L3 路由拓扑

wincni 网络插件配置示例

```
{
  "cniVersion": "0.2.0",
  "name": "l2bridge",
  "type": "wincni.exe",
  "master": "Ethernet",
  "ipam": {
    "environment": "azure",
    "subnet": "10.10.187.64/26",
    "routes": [{
      "GW": "10.10.187.66"
    }]
  },
  "dns": {
    "Nameservers": [
      "11.0.0.10"
    ]
  },
  "AdditionalArgs": [{
      "Name": "EndpointPolicy",
      "Value": {
        "Type": "OutBoundNAT",
        "ExceptionList": [
          "11.0.0.0/8",
          "10.10.0.0/16",
          "10.127.132.128/25"
        ]
      }
    },
    {
      "Name": "EndpointPolicy",
      "Value": {
        "Type": "ROUTE",
        "DestinationPrefix": "11.0.0.0/8",
        "NeedEncap": true
      }
    },
    {
      "Name": "EndpointPolicy",
      "Value": {
        "Type": "ROUTE",
```

```
        "DestinationPrefix": "10.127.132.213/32",
        "NeedEncap": true
      }
    }
  ]
}
```

OVS 网络拓扑

部署

kubeadm

如果 Master 是通过 kubeadm 来部署的，那 Windows 节点也可以使用 kubeadm 来部署：

```
kubeadm.exe join --token  : --discovery-token-ca-cert-hash sha256
```

Azure

在 Azure 上面推荐使用 acs-engine 自动部署 Master 和 Windows 节点。

首先创建一个包含 Windows 的 Kubernetes 集群配置文件 `windows.json`

```
{
    "apiVersion": "vlabs",
    "properties": {
        "orchestratorProfile": {
            "orchestratorType": "Kubernetes",
            "orchestratorVersion": "1.11.1",
            "kubernetesConfig": {
                "networkPolicy": "none",
                "enableAggregatedAPIs": true,
                "enableRbac": true
            }
        },
        "masterProfile": {
            "count": 3,
            "dnsPrefix": "kubernetes-windows",
            "vmSize": "Standard_D2_v3"
        },
        "agentPoolProfiles": [
            {
                "name": "windowspool1",
                "count": 3,
                "vmSize": "Standard_D2_v3",
                "availabilityProfile": "AvailabilitySet",
                "osType": "Windows"
            }
        ],
        "windowsProfile": {
            "adminUsername": "",
            "adminPassword": ""
        },
        "linuxProfile": {
            "adminUsername": "azure",
            "ssh": {
                "publicKeys": [
                    {
                        "keyData": ""
                    }
                ]
            }
        },
```

```
        "servicePrincipalProfile": {
            "clientId": "",
            "secret": ""
        }
    }
}
```

然后使用 acs-engine 部署：

```
# create a new resource group.
az group create --name myResourceGroup  --location "centralus"

# start deploy the kubernetes
acs-engine deploy --resource-group myResourceGroup --subscriptior

# setup kubectl
export KUBECONFIG="$(pwd)/_output//kubeconfig/kubeconfig.centralu
kubectl get node
```

手动部署

(1) 在 Windows Server 中 安装 Docker

```
Install-Module -Name DockerMsftProvider -Repository PSGallery -Fo
Install-Package -Name Docker -ProviderName DockerMsftProvider
Restart-Computer -Force
```

(2) 根据前面的下载部分下载 kubelet.exe 和 kube-proxy.exe

(3) 从 Master 节点上面拷贝 Node spec file (kube config)

(4) 配置 CNI 网络插件和基础镜像

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtoc
wget https://github.com/Microsoft/SDN/archive/master.zip -o maste
Expand-Archive master.zip -DestinationPath master
mkdir C:/k/
mv master/SDN-master/Kubernetes/windows/* C:/k/
rm -recurse -force master,master.zip
```

```
docker pull microsoft/windowsservercore:1709
docker tag microsoft/windowsservercore:1709 microsoft/windowsserv
cd C:/k/
docker build -t kubeletwin/pause .
```

(5) 使用 start-kubelet.ps1 启动 kubelet.exe，并使用 start-kubeproxy.ps1 启动 kube-proxy.exe

```
[Environment]::SetEnvironmentVariable("KUBECONFIG", "C:\k\config"
./start-kubelet.ps1 -ClusterCidr 192.168.0.0/16
./start-kubeproxy.ps1
```

(6) 如果使用 Host-Gateway 网络插件，还需要使用 AddRoutes.ps1 添加静态路由

详细的操作步骤可以参考 这里。

运行 Windows 容器

使用 NodeSelector `beta.kubernetes.io/os: windows` 将容器调度到 Windows 节点上，比如

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: iis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: iis
    spec:
      nodeSelector:
        beta.kubernetes.io/os: windows
      containers:
      - name: iis
        image: microsoft/iis
        resources:
          limits:
            memory: "128Mi"
            cpu: 2
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: iis
  name: iis
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: iis
  type: NodePort
```

运行 DaemonSet

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: my-DaemonSet
  labels:
    app: foo
spec:
  template:
    metadata:
      labels:
        app: foo
    spec:
      containers:
      - name: foo
        image: microsoft/windowsservercore:1709
      nodeSelector:
        beta.kubernetes.io/os: windows
```

已知问题

Secrets 和 ConfigMaps 只能以环境变量的方式使用

1709和更早版本有这个问题，升级到 1803 即可解决。

Volume 支持情况

Windows 容器暂时只支持 local、emptyDir、hostPath、AzureDisk、AzureFile 以及 flexvolume。注意 Volume 的路径格式需要为 `mountPath: "C:\\etc\\foo"` 或者 `mountPath: "C:/etc/foo"` 。

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  containers:
  - name: hostpath-nano
    image: microsoft/nanoserver:1709
    stdin: true
    tty: true
    volumeMounts:
    - name: blah
      mountPath: "C:\\etc\\foo"
      readOnly: true
  nodeSelector:
    beta.kubernetes.io/os: windows
  volumes:
  - name: blah
    hostPath:
      path: "C:\\AzureData"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: empty-dir-pod
spec:
  containers:
  - image: microsoft/nanoserver:1709
    name: empty-dir-nano
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: C:/scratch
      name: scratch-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
  - name: scratch-volume
    emptyDir: {}
  nodeSelector:
    beta.kubernetes.io/os: windows
```

镜像版本匹配问题

在 `Windows Server version 1709` 中必须使用带有 `1709` 标签的镜像，如

- `microsoft/aspnet:4.7.1-windowsservercore-1709`
- `microsoft/windowsservercore:1709`
- `microsoft/iis:windowsservercore-1709`

同样，在 `Windows Server version 1803` 中必须使用带有 `1803` 标签的镜像。而在 `Windows Server 2016` 上需要使用带有 `ltsc2016` 标签的镜像，如 `microsoft/windowsservercore:ltsc2016` 。

设置 CPU 和内存

从 v1.10 开始，Kubernetes 支持给 Windows 容器设置 CPU 和内存：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: iis
    spec:
      containers:
      - name: iis
        image: microsoft/iis
        resources:
          limits:
            memory: "128Mi"
            cpu: 2
        ports:
        - containerPort: 80
```

Hyper-V 容器

从 v1.10 开始支持 Hyper-V 隔离的容器（Alpha）。 在使用之前，需要配置 kubelet 开启 `HyperVContainer` 特性开关。然后使用 Annotation `experimental.windows.kubernetes.io/isolation-type=hyperv` 来指定容器使用 Hyper-V 隔离：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: iis
      annotations:
        experimental.windows.kubernetes.io/isolation-type: hyperv
    spec:
      containers:
      - name: iis
        image: microsoft/iis
        ports:
        - containerPort: 80
```

其他已知问题

- 仅 Windows Server 1709 或更新的版本才支持在 Pod 内运行多个容器（仅支持 Process 隔离）
- 暂不支持 StatefulSet
- 暂不支持 Windows Server Container Pods 的自动扩展（Horizontal Pod Autoscaling）
- Windows 容器的 OS 版本需要与 Host OS 版本匹配，否则容器无法启动
- 使用 L3 或者 Host GW 网络时，无法从 Windows Node 中直接访问 Kubernetes Services（使用 OVS/OVN 时没有这个问题）
- 在 VMWare Fusion 的 Window Server 中 kubelet.exe 可能会无法启动（已在 #57124 中修复）
- 暂不支持 Weave 网络插件
- Calico 网络插件仅支持 Policy-Only 模式
- 对于需要使用 `:` 作为环境变量的 .NET 容器，可以将环境变量中的 `:` 替换为 `__`（参考 这里）

附录：Docker EE 安装方法

安装 Docker EE 稳定版本

```
Install-Module -Name DockerMsftProvider -Repository PSGallery -Fo
Install-Package -Name docker -ProviderName DockerMsftProvider
Restart-Computer -Force
```

安装 Docker EE 预览版本

```
Install-Module DockerProvider
Install-Package -Name Docker -ProviderName DockerProvider -Requir
```

升级 Docker EE 版本

```
# Check the installed version
Get-Package -Name Docker -ProviderName DockerMsftProvider

# Find the current version
Find-Package -Name Docker -ProviderName DockerMsftProvider

# Upgrade Docker EE
Install-Package -Name Docker -ProviderName DockerMsftProvider -Up
Start-Service Docker
```

参考文档

- Using Windows Server Containers in Kubernetes

LinuxKit

LinuxKit 是以 Container 来建立最小、不可变的 Linux 系统框架，可以参考 LinuxKit 简单介绍。本着则将利用 LinuxKit 来建立 Kubernetes 的映像档，并部署简单的 Kubernetes 集群。

本着教学会在 `Mac OS X` 系统上进行，部署的环境资讯如下：

- Kubernetes v1.7.2
- Etcd v3
- Weave
- Docker v17.06.0-ce

预先准备资讯

- 主机已安装与启动 `Docker` 工具。
- 主机已安装 `Git` 工具。
- 主机以下载 LinuxKit 项目，并建构了 Moby 与 LinuxKit 工具。

建构 Moby 与 LinuxKit 方法如以下操作：

```
$ git clone https://github.com/linuxkit/linuxkit.git
$ cd linuxkit
$ make
$ ./bin/moby version
moby version 0.0
commit: c2b081ed8a9f690820cc0c0568238e641848f58f

$ ./bin/linuxkit version
linuxkit version 0.0
commit: 0e3ca695d07d1c9870eca71fb7dd9ede31a38380
```

建构 Kubernetes 系统映像档

首先要建立一个打包好 Kubernetes 的 Linux 系统，而官方已经有做好范例，利用以下方式即可建构：

```
$ cd linuxkit/projects/kubernetes/
$ make build-vm-images
...
Create outputs:
  kube-node-kernel kube-node-initrd.img kube-node-cmdline
```

部署 Kubernetes cluster

完成建构映像档后，就可以透过以下指令来启动 Master OS，然后获取节点 IP：

```
$ ./boot.sh

(ns: getty) linuxkit-025000000002:~\# ip addr show dev eth0
2: eth0:  mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 02:50:00:00:00:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.65.3/24 brd 192.168.65.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::abf0:9fa4:d0f4:8da2/64 scope link
       valid_lft forever preferred_lft forever
```

启动后，开启新的 Console 来 SSH 进入 Master，来利用 kubeadm 初始化 Master：

```
$ cd linuxkit/projects/kubernetes/
$ ./ssh_into_kubelet.sh 192.168.65.3
linuxkit-025000000002:/\# kubeadm-init.sh
...
kubeadm join --token 4236d3.29f61af661c49dbf 192.168.65.3:6443
```

一旦 kubeadm 完成后，就会看到 Token，这时请记住 Token 资讯。接着开启新 Console，然后执行以下指令来启动 Node：

```
console1>$ ./boot.sh 1 --token 4236d3.29f61af661c49dbf 192.168.65
```

> P.S. 开启节点格式为 `./boot.sh [ ...]`。

接着分别在开两个 Console 来加入集群：

```
console2> $ ./boot.sh 2 --token 4236d3.29f61af661c49dbf 192.168.6
console3> $ ./boot.sh 3 --token 4236d3.29f61af661c49dbf 192.168.6
```

完成后回到 Master 节点上，执行以下指令来查看节点状况：

```
$ kubectl get no
NAME                    STATUS    AGE       VERSION
linuxkit-025000000002   Ready     16m       v1.7.2
linuxkit-025000000003   Ready     6m        v1.7.2
linuxkit-025000000004   Ready     1m        v1.7.2
linuxkit-025000000005   Ready     1m        v1.7.2
```

简单部署 Nginx 服务

Kubernetes 可以选择使用指令直接建立应用程式与服务，或者撰写 YAML 与 JSON 档案来描述部署应用的配置，以下将建立一个简单的 Nginx 服务：

```
$ kubectl run nginx --image=nginx --replicas=1 --port=80
$ kubectl get pods -o wide
NAME                     READY     STATUS    RESTARTS   AGE
nginx-1423793266-v0hpb   1/1       Running   0          38s
```

完成后要接着建立 svc(Service)，来提供外部网络存取应用：

```
$ kubectl expose deploy nginx --port=80 --type=NodePort
$ kubectl get svc
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   10.96.0.1               443/TCP        19m
nginx        10.108.41.230           80:31773/TCP   5s
```

由于不是使用物理机器部署，因此网络使用 Docker namespace 网络，故需透过 `ubuntu-desktop-lxde-vnc` 来浏览 Nginx 应用：

```
$ docker run -it --rm -p 6080:80 dorowu/ubuntu-desktop-lxde-vnc
```

> 完成后透过浏览器连接 HTML VNC。

最后关闭节点只需要执行以下即可：

```
$ halt
[1503.034689] reboot: Power down
```

LinuxKit

附加组件

部署 Kubernetes 集群后，还需要部署一系列的附加组件（addons），这些组件通常是保证集群功能正常运行必不可少的。

通常使用 addon-manager 来管理集群中的附加组件。它运行在 Kubernetes 集群 Master 节点中，管理着 `$ADDON_PATH`（默认是 `/etc/kubernetes/addons/`）目录中的所有扩展，保证它们始终运行在期望状态。

常见的组件包括：

- addon-manager
- cluster-loadbalancing
- cluster-monitoring
- dashboard
- device-plugins/nvidia-gpu
- dns-horizontal-autoscaler
- dns
- fluentd-elasticsearch
- ip-masq-agent
- istio
- kube-proxy
- metrics-server
- node-problem-detector
- prometheus
- storage-class

更多的扩展组件可以参考 Installing Addons 和 Legacy Addons。

Addon-manager

附加组件管理器（Addon-manager）是运行在 Kubernetes 集群 Master 节点、用来管理附加组件（Addons）的服务。它管理着 `$ADDON_PATH`（默认是 `/etc/kubernetes/addons/`）目录中的所有扩展，保证它们始终运行在期望状态。

Addon-manager 支持两种标签

- 对于带有 `addonmanager.kubernetes.io/mode=Reconcile` 标签的扩展，无法通过 API 来修改，即
  - 如果通过 API 修改了，则会自动回滚到 `/etc/kubernetes/addons/` 中的配置
  - 如果通过 API 删除了，则会通过 `/etc/kubernetes/addons/` 中的配置自动重新创建
  - 如果从 `/etc/kubernetes/addons/` 中删除配置，则 Kubernetes 资源也会删除
  - 也就是说只能通过修改 `/etc/kubernetes/addons/` 中的配置来修改
- 对于带有 `addonmanager.kubernetes.io/mode=EnsureExists` 标签到扩展，仅检查扩展是否存在而不检查配置是否更改，即
  - 可以通过 API 来修改配置，不会自动回滚
  - 如果通过 API 删除了，则会通过 `/etc/kubernetes/addons/` 中的配置自动重新创建
  - 如果从 `/etc/kubernetes/addons/` 中删除配置，则 Kubernetes 资源不会删除

部署方法

将下面的 YAML 存入所有 Master 节点的 `/etc/kubernetes/manifests/kube-addon-manager.yaml` 文件中：

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-addon-manager
  namespace: kube-system
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ''
    seccomp.security.alpha.kubernetes.io/pod: 'docker/default'
  labels:
    component: kube-addon-manager
spec:
  hostNetwork: true
  containers:
  - name: kube-addon-manager
    # When updating version also bump it in:
    # - test/kubemark/resources/manifests/kube-addon-manager.yaml
    image: k8s.gcr.io/kube-addon-manager:v8.7
    command:
    - /bin/bash
    - -c
    - exec /opt/kube-addons.sh 1>>/var/log/kube-addon-manager.log
    resources:
      requests:
        cpu: 3m
        memory: 50Mi
    volumeMounts:
    - mountPath: /etc/kubernetes/
      name: addons
      readOnly: true
    - mountPath: /var/log
      name: varlog
      readOnly: false
    env:
    - name: KUBECTL_EXTRA_PRUNE_WHITELIST
      value: {{kubectl_extra_prune_whitelist}}
  volumes:
  - hostPath:
      path: /etc/kubernetes/
    name: addons
  - hostPath:
```

```
      path: /var/log
    name: varlog
```

源码

Addon-manager 的源码维护在 https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/addon-manager。

Dashboard

Kubernetes Dashboard 的部署非常简单，只需要运行

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/das
```

稍等一会，dashborad 就会创建好

```
$ kubectl -n kube-system get service kubernetes-dashboard
NAME                   CLUSTER-IP       EXTERNAL-IP   PORT(S)
kubernetes-dashboard   10.101.211.212          80:32729/TCP   1m
$ kubectl -n kube-system describe service kubernetes-dashboard
Name:            kubernetes-dashboard
Namespace:        kube-system
Labels:            app=kubernetes-dashboard
Annotations:
Selector:        app=kubernetes-dashboard
Type:            NodePort
IP:            10.101.211.212
Port:            <unset>    80/TCP
NodePort:        <unset>    32729/TCP
Endpoints:        10.244.1.3:9090
Session Affinity:    None
Events:
```

然后就可以通过 `http://nodeIP:32729` 来访问了。

登录认证

导入证书登录

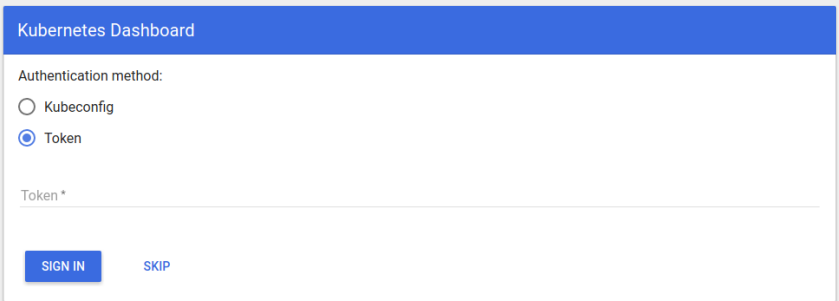
在 v1.7 之前的版本中，Dashboard 并不提供登陆的功能。而通常情况下，Dashboard 服务都是以 https 的方式运行，所以可以在访问它之前将证书导入系统中：

```
openssl pkcs12 -export -in apiserver-kubelet-client.crt -inkey ap
curl -sSL -E ./kube.p12:password -k https://nodeIP:6443/api/v1/pr
```

将 kube.p12 导入系统就可以用浏览器来访问了。注意，如果 nodeIP 不在证书 CN 里面，则需要做个 hosts 映射。

使用 kubeconfig 配置文件登录

从 v1.7.0 版本开始，Dashboard 支持以 kubeconfig 配置文件的方式登录。打开 Dashboard 页面会自动跳转到登录的界面，选择 Kubeconfig 方式，并选择本地的 kubeconfig 配置文件即可。



使用 Token 登录

从 v1.7.0 版本开始，Dashboard 支持以 Token 的方式登录。注意从 Kubernetes 中取得的 Token 需要以 Base64 解码后才可以用来登录。

下面是一个在开启 RBAC 时创建一个只可以访问 demo namespace 的 service account token 示例：

```
# 创建 demo namespace
kubectl create namespace demo

# 创建并限制只可以访问 demo namespace
cat <<-f -
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: demo
  name: default-role
rules:
  - apiGroups:
    - '*'
    resources:
    - '*'
    verbs:
    - '*'
EOF
kubectl create rolebinding default-rolebinding --serviceaccount=d

# 获取 token
secret=$(kubectl -n demo get sa default -o jsonpath='{.secrets[0]
kubectl -n demo get secret $secret -o jsonpath='{.data.token}' |
```

注意，由于使用该 token 仅可以访问 demo namespace，故而需要登录后将访问 URL 中的 default 改成 demo。

其他用户界面

除了 Kubernetes 社区提供的 Dashboard，还可以使用下列用户界面来管理 Kubernetes 集群

- Cabin：Android/iOS App，用于在移动端管理 Kubernetes
- Kubernetic：Kubernetes 桌面客户端
- Kubernator：低级（low-level） Web 界面，用于直接管理 Kubernetes 的资源对象（即 YAML 配置）

kubernator

监控

Kubernetes 社区提供了一些列的工具来监控容器和集群的状态，并借助 Prometheus 提供告警的功能。

- cAdvisor 负责单节点内部的容器和节点资源使用统计，内置在 Kubelet 内部，并通过 Kubelet `/metrics/cadvisor` 对外提供 API
- InfluxDB 是一个开源分布式时序、事件和指标数据库；而 Grafana 则是 InfluxDB 的 Dashboard，提供了强大的图表展示功能。它们常被组合使用展示图表化的监控数据。
- metrics-server 提供了整个集群的资源监控数据，但要注意
  - Metrics API 只可以查询当前的度量数据，并不保存历史数据
  - Metrics API URI 为 `/apis/metrics.k8s.io/`，在 k8s.io/metrics 维护
  - 必须部署 `metrics-server` 才能使用该 API，metrics-server 通过调用 Kubelet Summary API 获取数据
- kube-state-metrics 提供了 Kubernetes 资源对象（如 DaemonSet、Deployments 等）的度量。
- Prometheus 是另外一个监控和时间序列数据库，还提供了告警的功能。
- Node Problem Detector 监测 Node 本身的硬件、内核或者运行时等问题。
- Heapster ~~提供了整个集群的资源监控，并支持持久化数据存储到 InfluxDB 等后端存储中（已弃用）~~

cAdvisor

cAdvisor 是一个来自 Google 的容器监控工具，也是 Kubelet 内置的容器资源收集工具。它会自动收集本机容器 CPU、内存、网络和文件系统的资源占用情况，并对外提供 cAdvisor 原生的 API（默认端口为 `--cadvisor-port=4194`）。

从 v1.7 开始，Kubelet metrics API 不再包含 cadvisor metrics，而是提供了一个独立的 API 接口：

- Kubelet metrics: `http://127.0.0.1:8001/api/v1/proxy/nodes//metrics`

- Cadvisor metrics: `http://127.0.0.1:8001/api/v1/proxy/nodes//metrics/cadvisor`

这样，在 Prometheus 等工具中需要使用新的 Metrics API 来获取这些数据，比如下面的 Prometheus 自动配置了 cadvisor metrics API：

```
helm install stable/prometheus --set rbac.create=true --name prom
```

注意：cadvisor 监听的端口将在 v1.12 中删除，建议所有外部工具使用 Kubelet Metrics API 替代。

InfluxDB 和 Grafana

InfluxDB 是一个开源分布式时序、事件和指标数据库；而 Grafana 则是 InfluxDB 的 Dashboard，提供了强大的图表展示功能。它们常被组合使用展示图表化的监控数据。

Heapster

Kubelet 内置的 cAdvisor 只提供了单机的容器资源占用情况，而 Heapster 则提供了整个集群的资源监控，并支持持久化数据存储到 InfluxDB、Google Cloud Monitoring 或者 其他的存储后端。注意：

- 仅 Kubernetes v1.7.X 或者更老的集群推荐使用 Heapster。
- 从 Kubernetes v1.8 开始，资源使用情况的度量（如容器的 CPU 和内存使用）就已经通过 Metrics API 获取，并且 HPA 也从 metrics-server 查询必要的数据。
- **Heapster 已在 v1.11 中弃用，推荐 v1.8 及以上版本部署 metrics-server 替代 Heapster**

Heapster 首先从 Kubernetes apiserver 查询所有 Node 的信息，然后再从 kubelet 提供的 API 采集节点和容器的资源占用，同时在 `/metrics` API 提供了 Prometheus 格式的数据。Heapster 采集到的数据可以推送到各种持久化的后端存储中，如 InfluxDB、Google Cloud Monitoring、OpenTSDB 等。

部署 Heapster、InfluxDB 和 Grafana

在 Kubernetes 部署成功后，dashboard、DNS 和监控的服务也会默认部署好，比如通过 `cluster/kube-up.sh` 部署的集群默认会开启以下服务：

```
$ kubectl cluster-info
Kubernetes master is running at https://kubernetes-master
Heapster is running at https://kubernetes-master/api/v1/proxy/nam
KubeDNS is running at https://kubernetes-master/api/v1/proxy/name
kubernetes-dashboard is running at https://kubernetes-master/api/
Grafana is running at https://kubernetes-master/api/v1/proxy/name
InfluxDB is running at https://kubernetes-master/api/v1/proxy/nam
```

如果这些服务没有自动部署的话，可以参考 kubernetes/heapster 来部署这些服务：

```
git clone https://github.com/kubernetes/heapster
cd heapster
kubectl create -f deploy/kube-config/influxdb/
kubectl create -f deploy/kube-config/rbac/heapster-rbac.yaml
```

注意在访问这些服务时，需要先在浏览器中导入 apiserver 证书才可以认证。为了简化访问过程，也可以使用 kubectl 代理来访问（不需要导入证书）：

```
# 启动代理
kubectl proxy --address='0.0.0.0' --port=8080 --accept-hosts='^*$
```

然后打开 `http://:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana` 就可以访问 Grafana。

Prometheus

Prometheus 是另外一个监控和时间序列数据库，并且还提供了告警的功能。它提供了强大的查询语言和 HTTP 接口，也支持将数据导出到 Grafana 中展示。

prometheus

使用 Prometheus 监控 Kubernetes 需要配置好数据源，一个简单的示例是 prometheus.yml。

推荐使用 Prometheus Operator 或 Prometheus Chart 来部署和管理 Prometheus，比如

```
# 使用 prometheus operator
helm repo add coreos https://s3-eu-west-1.amazonaws.com/coreos-ch
helm install coreos/prometheus-operator --name prometheus-operato
helm install coreos/kube-prometheus --name kube-prometheus --name
```

使用端口转发的方式访问 Prometheus，如 `kubectl --namespace monitoring port-forward service/kube-prometheus-prometheus :9090`

prometheus-web

如果发现 exporter-kubelets 功能不正常，比如报 `server returned HTTP status 401 Unauthorized` 错误，则需要给 Kubelet 配置 webhook 认证：

```
kubelet --authentication-token-webhook=true --authorization-mode=
```

如果发现 K8SControllerManagerDown 和 K8SSchedulerDown 告警，则说明 kube-controller-manager 和 kube-scheduler 是以 Pod 的形式运行在集群中的，并且 prometheus 部署的监控服务与它们的标签不一致。可通过修改服务标签的方法解决，如

```
kubectl -n kube-system set selector service kube-prometheus-expor
kubectl -n kube-system set selector service kube-prometheus-expor
```

查询 Grafana 的管理员密码

```
kubectl get secret --namespace monitoring kube-prometheus-grafana
kubectl get secret --namespace monitoring kube-prometheus-grafana
```

然后，以端口转发的方式访问 Grafana 界面

```
kubectl port-forward -n monitoring service/kube-prometheus-grafar
```

添加 Prometheus 类型的 Data Source，填入原地址 `http://prometheus-prometheus-server.monitoring` 。

Node Problem Detector

Kubernetes node 有可能会出现各种硬件、内核或者运行时等问题，这些问题有可能导致服务异常。而 Node Problem Detector（NPD）就是用来监测这些异常的服务。NPD 以 DaemonSet 的方式运行在每台 Node 上面，并在异常发生时更新 NodeCondition（比如 KernelDaedlock、DockerHung、BadDisk 等）或者 Node Event（比如 OOM Kill 等）。

可以参考 kubernetes/node-problem-detector 来部署 NPD，或者也可以使用 Helm 来部署：

```
# add repo
helm repo add feisky https://feisky.xyz/kubernetes-charts
helm update

# install packages
helm install feisky/node-problem-detector --namespace kube-system
```

Node 重启守护进程

Kubernetres 集群中的节点通常会开启自动安全更新，这样有助于尽可能避免因系统漏洞带来的损失。但一般来说，涉及到内核的更新需要重启系统才可生效。此时，就需要手动或自动的方法来重启节点。

Kured (KUbernetes REboot Daemon) 就是这样一个守护进程，它会

- 监控 `/var/run/reboot-required` 信号后重启节点
- 通过 DaemonSet Annotation 的方式每次仅重启一台节点
- 重启前驱逐节点，重启后恢复调度
- 根据 Prometheus 告警（`--alert-filter-regexp=^(RebootRequired|AnotherBenignAlert|...$`）取消重启
- Slack 通知

部署方法

```
kubectl apply -f https://github.com/weaveworks/kured/releases/dow
```

其他容器监控系统

除了以上监控工具，还有很多其他的开源或商业系统可用来辅助监控，如

- Sysdig
- Weave scope
- CoScale
- Datadog
- Sematext

sysdig

sysdig 是一个容器排错工具，提供了开源和商业版本。对于常规排错来说，使用开源版本即可。

除了 sysdig，还可以使用其他两个辅助工具

- csysdig：与 sysdig 一起自动安装，提供了一个命令行界面
- sysdig-inspect：为 sysdig 保存的跟踪文件（如 `sudo sysdig -w filename.scap`）提供了一个图形界面（非实时）

安装 sysdig

```
# on Linux
curl -s https://s3.amazonaws.com/download.draios.com/stable/insta

# on MacOS
brew install sysdig
```

使用示例

```
# Refer https://www.sysdig.org/wiki/sysdig-examples/.
# View the top network connections for a single container
sysdig -pc -c topconns

# Show the network data exchanged with the host 192.168.0.1
sysdig -s2000 -A -c echo_fds fd.cip=192.168.0.1

# List all the incoming connections that are not served by apache
sysdig -p"%proc.name %fd.name" "evt.type=accept and proc.name!=ht

# View the CPU/Network/IO usage of the processes running inside t
sysdig -pc -c topprocs_cpu container.id=2e854c4525b8
sysdig -pc -c topprocs_net container.id=2e854c4525b8
sysdig -pc -c topfiles_bytes container.id=2e854c4525b8

# See the files where apache spends the most time doing I/O
sysdig -c topfiles_time proc.name=httpd

# Show all the interactive commands executed inside a given conta
sysdig -pc -c spy_users

# Show every time a file is opened under /etc.
sysdig evt.type=open and fd.name
```

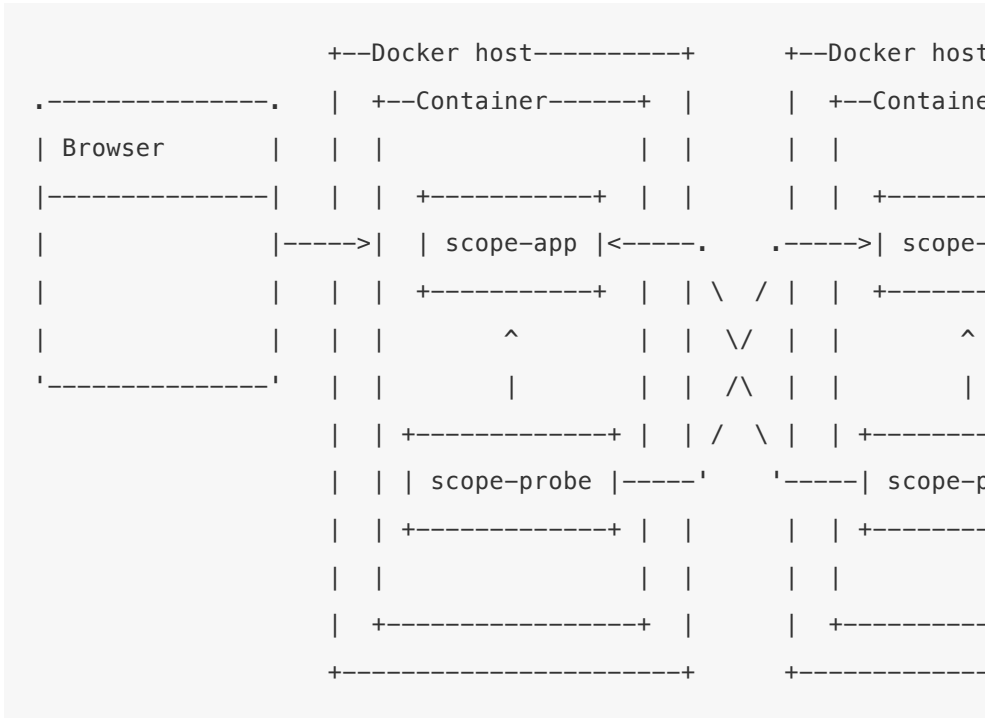
Weave Scope

Weave Scope 是另外一款可视化容器监控和排错工具。与 sysdig 相比，它没有强大的命令行工具，但提供了一个简单易用的交互界面，自动描绘了整个集群的拓扑，并可以通过插件扩展其功能。从其官网的介绍来看，其提供的功能包括

- 交互式拓扑界面
- 图形模式和表格模式
- 过滤功能
- 搜索功能
- 实时度量
- 容器排错
- 插件扩展

Weave Scope 由 App 和 Probe 两部分组成，它们

- Probe 负责收集容器和宿主的信息，并发送给 App
- App 负责处理这些信息，并生成相应的报告，并以交互界面的形式展示



安装 Weave scope

```
kubectl apply -f "https://cloud.weave.works/k8s/scope.yaml?k8s-ve
```

安装完成后，可以通过 weave-scope-app 来访问交互界面

```
kubectl -n weave get service weave-scope-app
```

点击 Pod，还可以查看该 Pod 所有容器的实时状态和度量数据：

参考文档

- Kubernetes Heapster

日志

ELK 可谓是容器日志收集、处理和搜索的黄金搭档：

- Logstash（或者 Fluentd）负责收集日志

- Elasticsearch 存储日志并提供搜索
- Kibana 负责日志查询和展示

注意：Kubernetes 默认使用 fluentd（以 DaemonSet 的方式启动）来收集日志，并将收集的日志发送给 elasticsearch。

小提示

在使用 `cluster/kube-up.sh` 部署集群的时候，可以设置 `KUBE_LOGGING_DESTINATION` 环境变量自动部署 Elasticsearch 和 Kibana，并使用 fluentd 收集日志（配置参考 addons/fluentd-elasticsearch）：

```
KUBE_LOGGING_DESTINATION=elasticsearch
KUBE_ENABLE_NODE_LOGGING=true
cluster/kube-up.sh
```

如果使用 GCE 或者 GKE 的话，还可以 将日志发送给 Google Cloud Logging，并可以集成 Google Cloud Storage 和 BigQuery。

如果需要集成其他的日志方案，还可以自定义 docker 的 log driver，将日志发送到 splunk 或者 awslogs 等。

部署方法

由于 Fluentd daemonset 只会调度到带有标签 `kubectl label nodes --all beta.kubernetes.io/fluentd-ds-ready=true` 的 Node 上，需要给 Node 设置标签

```
kubectl label nodes --all beta.kubernetes.io/fluentd-ds-ready=tru
```

然后下载 manifest 部署：

```
$ git clone https://github.com/kubernetes/kubernetes
$ cd cluster/addons/fluentd-elasticsearch
$ kubectl apply -f .
clusterrole "elasticsearch-logging" configured
clusterrolebinding "elasticsearch-logging" configured
replicationcontroller "elasticsearch-logging-v1" configured
service "elasticsearch-logging" configured
serviceaccount "elasticsearch-logging" configured
clusterrole "fluentd-es" configured
clusterrolebinding "fluentd-es" configured
daemonset "fluentd-es-v1.24" configured
serviceaccount "fluentd-es" configured
deployment "kibana-logging" configured
service "kibana-logging" configured
```

注意：Kibana 容器第一次启动的时候会用较长的时间（Optimizing and caching bundles for kibana and statusPage. This may take a few minutes），可以通过日志观察初始化的情况

```
$ kubectl -n kube-system logs kibana-logging-1237565573-p88lm -f
```

访问 Kibana

可以从 `kubectl cluster-info` 的输出中找到 Kibana 服务的访问地址，注意需要在浏览器中导入 apiserver 证书才可以认证：

```
$ kubectl cluster-info | grep Kibana
Kibana is running at https://10.0.4.3:6443/api/v1/namespaces/kube
```

这里采用另外一种方式，使用 kubectl 代理来访问（不需要导入证书）：

```
# 启动代理
kubectl proxy --address='0.0.0.0' --port=8080 --accept-hosts='^*$
```

然后打开 `http://:8080/api/v1/proxy/namespaces/kube-system/services/kibana-logging/app/kibana#` 。在 Settings -> Indices 页面创建一个 index，选中 Index contains time-based events，使用默认的 `logstash-*` pattern，点击 Create。

Filebeat

除了 Fluentd 和 Logstash，还可以使用 Filebeat 来收集日志：

```
kubectl apply -f https://raw.githubusercontent.com/elastic/beats/
```

注意，默认假设 Elasticsearch 可通过 `elasticsearch:9200` 访问，如果不同的话，需要先修改再部署

```
- name: ELASTICSEARCH_HOST
  value: elasticsearch
- name: ELASTICSEARCH_PORT
  value: "9200"
- name: ELASTICSEARCH_USERNAME
  value: elastic
- name: ELASTICSEARCH_PASSWORD
  value: changeme
```

参考文档

- Logging Agent For Elasticsearch
- Logging Using Elasticsearch and Kibana

Metrics

从 v1.8 开始，资源使用情况的度量（如容器的 CPU 和内存使用）可以通过 Metrics API 获取。注意

- Metrics API 只可以查询当前的度量数据，并不保存历史数据
- Metrics API URI 为 `/apis/metrics.k8s.io/`，在 k8s.io/metrics 维护
- 必须部署 `metrics-server` 才能使用该 API，metrics-server 通过调用 Kubelet Summary API 获取数据

Kubernetes 监控架构

Kubernetes 监控架构由以下两部分组成：

- 核心度量流程（下图黑色部分）：这是 Kubernetes 正常工作所需要的核心度量，从 Kubelet、cAdvisor 等获取度量数据，再由 metrics-server 提供给 Dashboard、HPA 控制器等使用。
- 监控流程（下图蓝色部分）：基于核心度量构建的监控流程，比如 Prometheus 可以从 metrics-server 获取核心度量，从其他数据源（如 Node Exporter 等）获取非核心度量，再基于它们构建监控告警系统。

开启API Aggregation

在部署 metrics-server 之前，需要在 kube-apiserver 中开启 API Aggregation，即增加以下配置

```
--requestheader-client-ca-file=/etc/kubernetes/certs/proxy-ca.crt
--proxy-client-cert-file=/etc/kubernetes/certs/proxy.crt
--proxy-client-key-file=/etc/kubernetes/certs/proxy.key
--requestheader-allowed-names=aggregator
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
```

如果kube-proxy没有在Master上面运行，还需要配置

```
--enable-aggregator-routing=true
```

部署 metrics-server

```
$ git clone https://github.com/kubernetes-incubator/metrics-serve
$ cd metrics-server
$ kubectl create -f deploy/1.8+/
```

稍后就可以看到 metrics-server 运行起来：

```
kubectl -n kube-system get pods -l k8s-app=metrics-server
```

Metrics API

可以通过 `kubectl proxy` 来访问 Metrics API：

- `http://127.0.0.1:8001/apis/metrics.k8s.io/v1beta1/nodes`
- `http://127.0.0.1:8001/apis/metrics.k8s.io/v1beta1/nodes/`
- `http://127.0.0.1:8001/apis/metrics.k8s.io/v1beta1/pods`
- `http://127.0.0.1:8001/apis/metrics.k8s.io/v1beta1/namespace//pods/`

也可以直接通过 kubectl 命令来访问这些 API，比如

- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/pods`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/`
- `kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespace//pods/`

排错

如果发现 metrics-server Pod 无法正常启动，比如处于 CrashLoopBackOff 状态，并且 restartCount 在不停增加，则很有可能是其跟 kube-apiserver 通信有问题。查看该 Pod 的日志，可以发现

```
dial tcp 10.96.0.1:443: i/o timeout
```

解决方法是：

```
echo "ExecStartPost=/sbin/iptables -P FORWARD ACCEPT" >> /etc/sys
systemctl daemon-reload
systemctl restart docker
```

参考文档

- Core metrics pipeline
- metrics-server

GPU

Kubernetes 支持容器请求 GPU 资源（目前仅支持 NVIDIA GPU），在深度学习等场景中有大量应用。

使用方法

Kubernetes v1.8 及更新版本

从 Kubernetes v1.8 开始，GPU 开始以 DevicePlugin 的形式实现。在使用之前需要配置

- kubelet/kube-apiserver/kube-controller-manager：`--feature-gates="DevicePlugins=true"`
- 在所有的 Node 上安装 Nvidia 驱动，包括 NVIDIA Cuda Toolkit 和 cuDNN 等
- Kubelet 配置使用 docker 容器引擎（默认就是 docker），其他容器引擎暂不支持该特性

NVIDIA 插件

NVIDIA 需要 nvidia-docker。

安装 nvidia-docker

```
# Install docker-ce
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo ap
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
sudo apt-get update
sudo apt-get install docker-ce

# Add the package repositories
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | \
  sudo apt-key add -
curl -s -L https://nvidia.github.io/nvidia-docker/ubuntu16.04/amd
  sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update

# Install nvidia-docker2 and reload the Docker daemon configurati
sudo apt-get install -y nvidia-docker2
sudo pkill -SIGHUP dockerd

# Test nvidia-smi with the latest official CUDA image
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

设置 Docker 默认运行时为 nvidia

```
# cat /etc/docker/daemon.json
{
    "default-runtime": "nvidia",
    "runtimes": {
        "nvidia": {
            "path": "/usr/bin/nvidia-container-runtime",
            "runtimeArgs": []
        }
    }
}
```

部署 NVDIA 设备插件

```
# For Kubernetes v1.8
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-de

# For Kubernetes v1.9
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-de
```

GCE/GKE GPU 插件

该插件不需要 nvidia-docker，并且也支持 CRI 容器运行时。

```
# Install NVIDIA drivers on Container-Optimized OS:
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPl

# Install NVIDIA drivers on Ubuntu (experimental):
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPl

# Install the device plugin:
kubectl create -f https://raw.githubusercontent.com/kubernetes/ku
```

请求 `nvidia.com/gpu` 资源示例

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/tes
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1 # requesting 1 GPU
```

Kubernetes v1.6 和 v1.7

> `alpha.kubernetes.io/nvidia-gpu` 已在 v1.10 中删除，新版本请使用 `nvidia.com/gpu`。

在 Kubernetes v1.6 和 v1.7 中使用 GPU 需要预先配置

- 在所有的 Node 上安装 Nvidia 驱动，包括 NVIDIA Cuda Toolkit 和 cuDNN 等
- 在 apiserver 和 kubelet 上开启 `--feature-gates="Accelerators=true"`
- Kubelet 配置使用 docker 容器引擎（默认就是 docker），其他容器引擎暂不支持该特性

使用资源名 `alpha.kubernetes.io/nvidia-gpu` 指定请求 GPU 的个数，如

```
apiVersion: v1
kind: Pod
metadata:
  name: tensorflow
spec:
  restartPolicy: Never
  containers:
  - image: gcr.io/tensorflow/tensorflow:latest-gpu
    name: gpu-container-1
    command: ["python"]
    env:
    - name: LD_LIBRARY_PATH
      value: /usr/lib/nvidia
    args:
    - -u
    - -c
    - from tensorflow.python.client import device_lib; print devi
    resources:
      limits:
        alpha.kubernetes.io/nvidia-gpu: 1 # requests one GPU
    volumeMounts:
    - mountPath: /usr/local/nvidia/bin
      name: bin
    - mountPath: /usr/lib/nvidia
      name: lib
    - mountPath: /usr/lib/x86_64-linux-gnu/libcuda.so
      name: libcuda-so
    - mountPath: /usr/lib/x86_64-linux-gnu/libcuda.so.1
      name: libcuda-so-1
    - mountPath: /usr/lib/x86_64-linux-gnu/libcuda.so.375.66
      name: libcuda-so-375-66
  volumes:
    - name: bin
      hostPath:
        path: /usr/lib/nvidia-375/bin
    - name: lib
      hostPath:
        path: /usr/lib/nvidia-375
    - name: libcuda-so
      hostPath:
```

```
        path: /usr/lib/x86_64-linux-gnu/libcuda.so
    - name: libcuda-so-1
      hostPath:
        path: /usr/lib/x86_64-linux-gnu/libcuda.so.1
    - name: libcuda-so-375-66
      hostPath:
        path: /usr/lib/x86_64-linux-gnu/libcuda.so.375.66
```

```
$ kubectl create -f pod.yaml
pod "tensorflow" created

$ kubectl logs tensorflow
...
[name: "/cpu:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 9675741273569321173
, name: "/gpu:0"
device_type: "GPU"
memory_limit: 11332668621
locality {
  bus_id: 1
}
incarnation: 7807115828340118187
physical_device_desc: "device: 0, name: Tesla K80, pci bus id: 00
]
```

注意

- GPU 资源必须在 `resources.limits` 中请求，`resources.requests` 中无效
- 容器可以请求 1 个或多个 GPU，不能只请求一部分
- 多个容器之间不能共享 GPU
- 默认假设所有 Node 安装了相同型号的 GPU

多种型号的 GPU

如果集群 Node 中安装了多种型号的 GPU，则可以使用 Node Affinity 来调度 Pod 到指定 GPU 型号的 Node 上。

首先，在集群初始化时，需要给 Node 打上 GPU 型号的标签

```
# Label your nodes with the accelerator type they have.
kubectl label nodes  accelerator=nvidia-tesla-k80
kubectl label nodes  accelerator=nvidia-tesla-p100
```

然后，在创建 Pod 时设置 Node Affinity：

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/tes
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100 # or nvidia-tesla-k80 etc.
```

使用 CUDA 库

NVIDIA Cuda Toolkit 和 cuDNN 等需要预先安装在所有 Node 上。为了访问 `/usr/lib/nvidia-375`，需要将 CUDA 库以 hostPath volume 的形式传给容器：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: nvidia-smi
  labels:
    name: nvidia-smi
spec:
  template:
    metadata:
      labels:
        name: nvidia-smi
    spec:
      containers:
      - name: nvidia-smi
        image: nvidia/cuda
        command: ["nvidia-smi"]
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            alpha.kubernetes.io/nvidia-gpu: 1
        volumeMounts:
        - mountPath: /usr/local/nvidia/bin
          name: bin
        - mountPath: /usr/lib/nvidia
          name: lib
      volumes:
        - name: bin
          hostPath:
            path: /usr/lib/nvidia-375/bin
        - name: lib
          hostPath:
            path: /usr/lib/nvidia-375
      restartPolicy: Never
```

```
$ kubectl create -f job.yaml
job "nvidia-smi" created

$ kubectl get job
NAME         DESIRED   SUCCESSFUL   AGE
nvidia-smi   1         1            14m

$ kubectl get pod -a
NAME               READY     STATUS      RESTARTS   AGE
nvidia-smi-kwd2m   0/1       Completed   0          14m

$ kubectl logs nvidia-smi-kwd2m
Fri Jun 16 19:49:53 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.66                 Driver Version: 375.66                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util
|===============================+======================+======================
|   0  Tesla K80           Off  | 0000:00:04.0     Off |
| N/A   74C    P0    80W / 149W |      0MiB / 11439MiB |    100%
+-------------------------------+----------------------+----------------------

+-----------------------------------------------------------------------------+
| Processes:
|  GPU       PID  Type  Process name
|=============================================================================
|  No running processes found
+-----------------------------------------------------------------------------
```

附录：CUDA 安装方法

安装 CUDA：

```
# Check for CUDA and try to install.
if ! dpkg-query -W cuda; then
  # The 16.04 installer works with 16.10.
  curl -O http://developer.download.nvidia.com/compute/cuda/repos
  dpkg -i ./cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
  apt-get update
  apt-get install cuda -y
fi
```

安装 cuDNN：

首先到网站 https://developer.nvidia.com/cudnn 注册，并下载 cuDNN v5.1，然后运行命令安装

```
tar zxvf cudnn-8.0-linux-x64-v5.1.tgz
ln -s /usr/local/cuda-8.0 /usr/local/cuda
sudo cp -P cuda/include/cudnn.h /usr/local/cuda/include
sudo cp -P cuda/lib64/libcudnn* /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/li
```

安装完成后，可以运行 nvidia-smi 查看 GPU 设备的状态

```
$ nvidia-smi
Fri Jun 16 19:33:35 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.66                 Driver Version: 375.66                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 0000:00:04.0     Off |                    0 |
| N/A   74C    P0    80W / 149W |      0MiB / 11439MiB |    100%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

参考文档

- NVIDIA/k8s-device-plugin
- Schedule GPUs on Kubernetes
- GoogleCloudPlatform/container-engine-accelerators

ClusterAutoscaler

Cluster AutoScaler 是一个自动扩展和收缩 Kubernetes 集群 Node 的扩展。当集群容量不足时，它会自动去 Cloud Provider （支持 GCE、GKE、Azure、AKS、AWS 等）创建新的 Node，而在 Node 长时间（超过 10 分钟）资源利用率很低时（低于 50%）自动将其删除以节省开支。

Cluster AutoScaler 独立于 Kubernetes 主代码库，维护在 https://github.com/kubernetes/autoscaler。

部署

Cluster AutoScaler v1.0+ 可以基于 Docker 镜像 `gcr.io/google_containers/cluster-autoscaler:v1.3.0` 来部署，详细的部署步骤可以参考

- GCE：https://kubernetes.io/docs/concepts/cluster-administration/cluster-management/
- GKE：https://cloud.google.com/container-engine/docs/cluster-autoscaler
- AWS：https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md
- Azure：https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler/cloudprovider/azure

注意，在开启 RBAC 的集群中创建 cluster-autoscaler ClusterRole。

工作原理

Cluster AutoScaler 定期（默认间隔 10s）检测是否有充足的资源来调度新创建的 Pod，当资源不足时会调用 Cloud Provider 创建新的 Node。

为了自动创建和初始化 Node，Cluster Autoscaler 要求 Node 必须属于某个 Node Group，比如

- GCE/GKE 中的 Managed instance groups（MIG）
- AWS 中的 Autoscaling Groups
- Azure 中的 Scale Sets 和 Availability Sets

当集群中有多个 Node Group 时，可以通过 `--expander=` 选项配置选择 Node Group 的策略，支持如下四种方式

- random：随机选择
- most-pods：选择容量最大（可以创建最多 Pod）的 Node Group
- least-waste：以最小浪费原则选择，即选择有最少可用资源的 Node Group
- price：选择最便宜的 Node Group（仅支持 GCE 和 GKE）

目前，Cluster Autoscaler 可以保证

- 小集群（小于 100 个 Node）可以在不超过 30 秒内完成扩展（平均 5 秒）
- 大集群（100-1000 个 Node）可以在不超过 60 秒内完成扩展（平均 15 秒）

Cluster AutoScaler 也会定期（默认间隔 10s）自动监测 Node 的资源使用情况，当一个 Node 长时间（超过 10 分钟其期间没有执行任何扩展操作）资源利用率都很低时（低于 50%）自动将其所在虚拟机从云服务商中删除（注意删除时会有 1 分钟的 graceful termination 时间）。此时，原来的 Pod 会自动调度到其他 Node 上面（通过 Deployment、StatefulSet 等控制器）。

注意，Cluster Autoscaler 仅根据 Pod 的调度情况和 Node 的整体资源使用清空来增删 Node，跟 Pod 或 Node 的资源度量（metrics）没有直接关系。

用户在启动 Cluster AutoScaler 时可以配置 Node 数量的范围（包括最大 Node 数和最小 Node 数）。

在使用 Cluster AutoScaler 时需要注意：

- 由于在删除 Node 时会发生 Pod 重新调度的情况，所以应用必须可以容忍重新调度和短时的中断（比如使用多副本的 Deployment）
- 当 Node 上面的 Pods 满足下面的条件之一 时，Node 不会删除
  - Pod 配置了 PodDisruptionBudget (PDB)
  - kube-system Pod 默认不在 Node 上运行或者未配置 PDB
  - Pod 不是通过 deployment, replica set, job, stateful set 等控制器创建的
  - Pod 使用了本地存储
  - 其他原因导致的 Pod 无法重新调度，如资源不足，其他 Node 无法满足 NodeSelector 或 Affinity 等

最佳实践

- Cluster AutoScaler 可以和 Horizontal Pod Autoscaler (HPA) 配合使用
- 不要手动修改 Node 配置，保证集群内的所有 Node 有相同的配置并属于同一个 Node 组
- 运行 Pod 时指定资源请求
- 必要时使用 PodDisruptionBudgets 阻止 Pod 被误删除

- 确保云服务商的配额充足
- Cluster AutoScaler **与云服务商提供的 Node 自动扩展功能以及基于 CPU 利用率的 Node 自动扩展机制冲突，不要同时启用**

参考文档

- Kubernetes Autoscaler
- Kubernetes Cluster AutoScaler Support

ip-masq-agent

ip-masq-agent 是一个用来管理 IP 伪装的扩展，即管理节点中 IP 网段的 SNAT 规则。

ip-masq-agent 配置 iptables 规则，以便将流量发送到集群节点之外的目标时处理 IP 伪装。默认情况下，RFC 1918 定一个的三个私有 IP 范围是非伪装网段，即 10.0.0.0/8、172.16.0.0/12 和 192.168.0.0/16。另外，链接本地地址（169.254.0.0/16）也被视为非伪装网段。

image-20181014212528267

部署方法

首先，标记要运行 ip-masq-agent 的 Node

```
kubectl label nodes my-node beta.kubernetes.io/masq-agent-ds-reac
```

然后部署 ip-masq-agent：

```
kubectl create -f https://raw.githubusercontent.com/kubernetes-ir
```

部署好，查看 iptables 规则，可以发现

```
iptables -t nat -L IP-MASQ-AGENT
RETURN     all  --  anywhere             169.254.0.0/16       /*
RETURN     all  --  anywhere             10.0.0.0/8           /*
RETURN     all  --  anywhere             172.16.0.0/12        /*
RETURN     all  --  anywhere             192.168.0.0/16       /*
MASQUERADE  all  --  anywhere              anywhere             /*
```

使用方法

自定义 SNAT 网段的方法：

```
cat >config <
```

这样，查看 iptables 规则可以发现

```
$ iptables -t nat -L IP-MASQ-AGENT
Chain IP-MASQ-AGENT (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             169.254.0.0/16       /* ip-masq-age
RETURN     all  --  anywhere             10.0.0.0/8           /* ip-masq-age
MASQUERADE  all  --  anywhere              anywhere             /* ip-masq-ag
```

Kubernetes-The-Hard-Way

翻译注：本部分翻译自 Kubernetes The Hard Way，译者 @kweisamx 和 @feiskyer。该教程指引用户在 Google Cloud Platform 上面一步步搭建一个高可用的 Kubernetes 集群。

如果你正在使用 Microsoft Azure，那么请参考 kubernetes-the-hard-way-on-azure 在 Azure 上面搭建 Kubernetes 集群。

如有翻译不好的地方或文字上的错误，欢迎提出 Issue 或是 PR。

本教程将带领你一步步配置和部署一套高可用的 Kubernetes 集群。它不适用于想要一键自动化部署 Kubernetes 集群的人。如果你想要一键自动化部署，请参考 Google Container Engine 或 Getting Started Guides。

Kubernetes The Hard Way 的主要目的是学习，也就是说它会花很多时间来保障读者可以真正理解搭建 Kubernetes 的每个步骤。

> 使用该教程部署的集群不应该直接视为生产环境可用，并且也可能无法获得 Kubernetes 社区的许多支持，但这都不影响你想真正了解 Kubernetes 的决心！

目标读者

该教程的目标是给那些计划要将 Kubernetes 应用到生产环境的人，并想了解每个有关 Kubernetes 的环节以及他们如何运作的。

集群版本

Kubernetes The Hard Way 将引导你建立高可用的 Kubernetes 集群，包括每个组件之间的加密以及 RBAC 认证

- Kubernetes 1.12.0
- Containerd Container Runtime 1.2.0-rc0
- CNI Container Networking 0.6.0
- gVisor 50c283b9f56bb7200938d9e207355f05f79f0d17
- etcd 3.3.9
- CoreDNS v1.2.2

实验步骤

这份教程假设你已经创建并配置好了 Google Cloud Platform 账户。该教程只是将 GCP 作为最基础的架构，教程的内容也同样适用于其他的平台。

- 准备部署环境
- 安装必要工具
- 创建计算资源
- 配置创建证书

- 配置生成配置
- 配置生成密钥
- 部署Etcd群集
- 部署控制节点
- 部署计算节点
- 配置Kubectl
- 配置网络路由
- 部署DNS扩展
- 烟雾测试
- 删除集群

准备部署环境

Google Cloud Platform

该指南使用 Google Cloud Platform 作为 kubernetes 集群的环境平台。注册 即可获得 300 美元的试用金。

估计 完成教学的花费金额：每小时 0.22 美元（每天 5.39 美元）.

> 注意：该教程所需要的计算资源会超出 GCP 免费额度。

Google Cloud Platform SDK

安装 Google Cloud SDK

按照 Google Cloud SDK 文档 的步骤去安装并设置 gcloud` 命令。验证 Google Cloud SDK 版本为 218.0.0 或更高：

```
gcloud version
```

设置默认 Region 和 Zone

本指南假设你的默认 Region 和 Zone 已经设置好了。如果你第一次使用 `gcloud` 指令工具，`init` 是一个最简单的设定方式：

```
gcloud init
```

或者，执行下面的命令手动设定 default compute region：

```
gcloud config set compute/region us-west1
```

手动设定 compute zone

```
gcloud config set compute/zone us-west1-c
```

> 使用 `gcloud compute zones list` 指令来查询其他的 region 和 zone。

使用 tmux 并行执行命令

tmux 可以用来在多个虚拟机中并行执行命令。该教程中的某些步骤需要在多台虚拟机中操作，此时可以考虑使用 tmux 来加速执行过程。

> tmux 是可选的，不是该教程的必要工具。

> 开启 tmux 同步的方法：按下 `ctrb+b` 和 `shift`，接着输入 `set synchronize-panes on`。关闭同步可以输入 `set synchronize-panes off`。

下一步：安装命令行工具

安装必要工具

本次实验你将会安装一些实用的命令行工具，用来完成这份指南，这包括 cfssl、cfssljson 以及 kubectl。

安装 CFSSL

从 cfssl 网站 下载 `cfssl` 和 `cfssljson` 并安装：

OS X

```
curl -o cfssl https://pkg.cfssl.org/R1.2/cfssl_darwin-amd64
curl -o cfssljson https://pkg.cfssl.org/R1.2/cfssljson_darwin-amd
chmod +x cfssl cfssljson
sudo mv cfssl cfssljson /usr/local/bin/
```

或者使用 Homebrew 来安装

```
brew install cfssl
```

Linux

```
wget -q --show-progress --https-only --timestamping \
  https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 \
  https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
chmod +x cfssl_linux-amd64 cfssljson_linux-amd64
sudo mv cfssl_linux-amd64 /usr/local/bin/cfssl
sudo mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
```

验证

验证 `cfssl` 的版本为 1.2.0 或是更高

```
cfssl version
```

> 输出为

```
Version: 1.2.0
Revision: dev
Runtime: go1.6
```

> 注意：`cfssljson` 命令行工具没有提供查询版本的方法。

安装 kubectl

`kubectl` 命令行工具用来与 Kubernetes API Server 交互，可以在 Kubernetes 官方网站下载并安装 `kubectl`。

OS X

```
curl -o kubectl https://storage.googleapis.com/kubernetes-release
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Linux

```
wget https://storage.googleapis.com/kubernetes-release/release/v1
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

验证

验证 `kubectl` 的安装版本为 `1.12.0` 或是更高

```
kubectl version --client
```

> 输出为

```
Client Version: version.Info{Major:"1", Minor:"12", GitVersion:"v
```

下一步： 准备计算资源

创建计算资源

Kubernetes 需要一些机器去搭建管理 Kubernetes 的控制平台，也需要一些工作节点（work node）来运行容器。在这个实验中你将会创建一些虚拟机，并利用 GCE Compute Zone 来运行安全且高可用的 Kubernetes 集群。

> 请确定默认 Compute Zone 和 Region 已按照 事前准备 的设定步骤完成。

网络

Kubernetes 网络模型 假设使用扁平网路能让每个容器与节点都可以相互通信。 在这里我们先忽略用于控制容器网络隔离的 Network policies（Network Policies 不在本指南的范围内）。

虚拟私有网络（VPC）

本节将会创建一个专用的 Virtual Private Cloud（VPC）网络来搭建我们的 Kubernetes 集群。

首先创建一个名为 kubernetes-the-hard-way 的 VPC 网络：

```
gcloud compute networks create kubernetes-the-hard-way --mode cus
```

为了给 Kubernetes 集群的每个节点分配私有 IP 地址，需要创建一个含有足够大 IP 地址池的子网。 在 `kubernetes-the-hard-way` VPC 网络中创建 `kubernetes` 子网：

```
gcloud compute networks subnets create kubernetes \
  --network kubernetes-the-hard-way \
  --range 10.240.0.0/24
```

> `10.240.0.0/24` IP 地址范围，可以分配 254 个计算节点。

防火墙规则

创建一个防火墙规则允许内部网路通过所有协议进行通信：

```
gcloud compute firewall-rules create kubernetes-the-hard-way-allo
  --allow tcp,udp,icmp \
  --network kubernetes-the-hard-way \
  --source-ranges 10.240.0.0/24,10.200.0.0/16
```

创建一个防火墙规则允许外部 SSH、ICMP 以及 HTTPS 等通信：

```
gcloud compute firewall-rules create kubernetes-the-hard-way-allo
  --allow tcp:22,tcp:6443,icmp \
  --network kubernetes-the-hard-way \
  --source-ranges 0.0.0.0/0
```

> 外部负载均衡器 被用来暴露 Kubernetes API Servers 给远端客户端。

列出在 `kubernetes-the-hard-way` VPC 网络中的防火墙规则：

```
gcloud compute firewall-rules list --filter="network:kubernetes-t
```

> 输出为

```
NAME                                    NETWORK
kubernetes-the-hard-way-allow-external  kubernetes-the-hard-way
kubernetes-the-hard-way-allow-internal  kubernetes-the-hard-way
```

Kubernetes 公网 IP 地址

分配固定的 IP 地址，被用来连接外部的负载平衡器至 Kubernetes API Servers：

```
gcloud compute addresses create kubernetes-the-hard-way \
  --region $(gcloud config get-value compute/region)
```

验证 `kubernetes-the-hard-way` 固定 IP 地址已经在默认的 Compute Region 中创建出来：

```
gcloud compute addresses list --filter="name=('kubernetes-the-har
```

> 输出为

```
NAME                     REGION    ADDRESS        STATUS
kubernetes-the-hard-way  us-west1  XX.XXX.XXX.XX  RESERVED
```

计算实例

本节将会创建基于 Ubuntu Server 18.04 的计算实例，原因是它对 containerd 容器引擎有很好的支持。每个虚拟机将会分配一个私有 IP 地址用以简化 Kubernetes 的设置。

Kubernetes 控制节点

建立三个计算节点用以配置 Kubernetes 控制平面：

```
for i in 0 1 2; do
  gcloud compute instances create controller-${i} \
    --async \
    --boot-disk-size 200GB \
    --can-ip-forward \
    --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud \
    --machine-type n1-standard-1 \
    --private-network-ip 10.240.0.1${i} \
    --scopes compute-rw,storage-ro,service-management,service-cor
    --subnet kubernetes \
    --tags kubernetes-the-hard-way,controller
done
```

Kubernetes 工作节点

每台 worker 节点都需要从 Kubernetes 集群 CIDR 范围中分配一个 Pod 子网。 Pod 子网分配将会在之后的容器网路章节做练习。在 worker 节点内部可以通过 `pod-cidr` 元数据来获得 Pod 子网的分配结果。

> Kubernetes 集群 CIDR 的范围可以通过 Controller Manager 的 `--cluster-cidr` 参数来设定。在本次教学中我们会设置为 `10.200.0.0/16`，它支持 254 个子网。

创建三个计算节点用来作为 Kubernetes Worker 节点：

```
for i in 0 1 2; do
  gcloud compute instances create worker-${i} \
    --async \
    --boot-disk-size 200GB \
    --can-ip-forward \
    --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud \
    --machine-type n1-standard-1 \
    --metadata pod-cidr=10.200.${i}.0/24 \
    --private-network-ip 10.240.0.2${i} \
    --scopes compute-rw,storage-ro,service-management,service-cor
    --subnet kubernetes \
    --tags kubernetes-the-hard-way,worker
done
```

验证

列出所有在默认 Compute Zone 的计算节点：

```
gcloud compute instances list
```

输出为：

```
NAME          ZONE        MACHINE_TYPE   PREEMPTIBLE  INTERNAL_IP
controller-0  us-west1-c  n1-standard-1               10.240.0.10
controller-1  us-west1-c  n1-standard-1               10.240.0.11
controller-2  us-west1-c  n1-standard-1               10.240.0.12
worker-0      us-west1-c  n1-standard-1               10.240.0.20
worker-1      us-west1-c  n1-standard-1               10.240.0.21
worker-2      us-west1-c  n1-standard-1               10.240.0.22
```

配置 SSH

本教程使用 SSH 来配置控制节点和工作节点。当通过 `gcloud compute ssh` 第一次连接计算实例时，会自动生成 SSH 证书，并保存在项目或者实例的元数据中。

验证 `controller-0` 的 SSH 访问

```
gcloud compute ssh controller-0
```

因为这是第一次访问，此时会生成 SSH 证书。按照提示操作

```
WARNING: The public SSH key file for gcloud does not exist.
WARNING: The private SSH key file for gcloud does not exist.
WARNING: You do not have an SSH key for gcloud.
WARNING: SSH keygen will be executed to generate a key.
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

此时，SSH 证书回保存在你的项目中：

```
Your identification has been saved in /home/$USER/.ssh/google_com
Your public key has been saved in /home/$USER/.ssh/google_compute
The key fingerprint is:
SHA256:nz1i8jHmgQuGt+WscqP5SeIaSy5wyIJeL71MuV+QruE $USER@$HOSTNAM
The key's randomart image is:
+---[RSA 2048]----+
|                 |
|                 |
|                 |
|        .        |
|o.     oS        |
|=... .o .o o     |
|+.+ =+=.+.X o    |
|.+ ==O*B.B = .   |
| .+.=EB++ o      |
+----[SHA256]-----+
Updating project ssh metadata...-Updated [https://www.googleapis.
Updating project ssh metadata...done.
Waiting for SSH key to propagate.
```

SSH 证书更新后，你就可以登录到 `controller-0` 实例中了：

```
Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.15.0-1006-gcp x86_64)

...

Last login: Sun May 13 14:34:27 2018 from XX.XXX.XXX.XX
```

下一步：配置 CA 和创建 TLS 证书

配置创建证书

我们将使用 CloudFlare's PKI 工具 cfssl 来配置 PKI Infrastructure，然后使用它去创建 Certificate Authority（CA），并为 etcd、kube-apiserver、kubelet 以及 kube-proxy 创建 TLS 证书。

Certificate Authority

本节创建用于生成其他 TLS 证书的 Certificate Authority。

新建 CA 配置文件

```
cat > ca-config.json <"signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": ["signing", "key encipherment", "server auth",
        "expiry": "8760h"
      }
    }
  }
}
EOF
```

新建 CA 凭证签发请求文件：

```
cat > ca-csr.json <"CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "CA",
      "ST": "Oregon"
    }
  ]
}
EOF
```

生成 CA 凭证和私钥：

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca
```

结果将生成以下两个文件：

```
ca-key.pem
ca.pem
```

client 与 server 凭证

本节将创建用于 Kubernetes 组件的 client 与 server 凭证，以及一个用于 Kubernetes admin 用户的 client 凭证。

Admin 客户端凭证

创建 `admin` client 凭证签发请求文件：

```
cat > admin-csr.json <"CN": "admin",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:masters",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF
```

创建 `admin` client 凭证和私钥：

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  admin-csr.json | cfssljson -bare admin
```

结果将生成以下两个文件

```
admin-key.pem
admin.pem
```

Kubelet 客户端凭证

Kubernetes 使用 special-purpose authorization mode（被称作 Node Authorizer）授权来自 Kubelet 的 API 请求。为了通过 Node Authorizer 的授权，Kubelet 必须使用一个署名为 `system:node:` 的凭证来证明它属于 `system:nodes` 用户组。本节将会给每台 worker 节点创建符合 Node Authorizer 要求的凭证。

给每台 worker 节点创建凭证和私钥：

```
for instance in worker-0 worker-1 worker-2; do
cat > ${instance}-csr.json <"CN": "system:node:${instance}",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:nodes",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF

EXTERNAL_IP=$(gcloud compute instances describe ${instance} \
  --format 'value(networkInterfaces[0].accessConfigs[0].natIP)')

INTERNAL_IP=$(gcloud compute instances describe ${instance} \
  --format 'value(networkInterfaces[0].networkIP)')

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=${instance},${EXTERNAL_IP},${INTERNAL_IP} \
  -profile=kubernetes \
  ${instance}-csr.json | cfssljson -bare ${instance}
done
```

结果将产生以下几个文件：

```
worker-0-key.pem
worker-0.pem
worker-1-key.pem
worker-1.pem
worker-2-key.pem
worker-2.pem
```

Kube-controller-manager 客户端凭证

```
cat > kube-controller-manager-csr.json <"CN": "system:kube-contro
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:kube-controller-manager",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-controller-manager-csr.json | cfssljson -bare kube-control
```

结果将产生以下几个文件：

```
kube-controller-manager-key.pem
kube-controller-manager.pem
```

Kube-proxy 客户端凭证

```
cat > kube-proxy-csr.json <"CN": "system:kube-proxy",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:node-proxier",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-proxy-csr.json | cfssljson -bare kube-proxy
```

结果将产生以下两个文件：

```
kube-proxy-key.pem
kube-proxy.pem
```

kube-scheduler 证书

```
cat > kube-scheduler-csr.json <"CN": "system:kube-scheduler",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:kube-scheduler",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-scheduler-csr.json | cfssljson -bare kube-scheduler
```

结果将产生以下两个文件：

```
kube-scheduler-key.pem
kube-scheduler.pem
```

Kubernetes API Server 证书

为了保证客户端与 Kubernetes API 的认证，Kubernetes API Server 凭证 中必需包含 `kubernetes-the-hard-way` 的静态 IP 地址。

首先查询 `kubernetes-the-hard-way` 的静态 IP 地址：

```
KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute addresses describe kub
  --region $(gcloud config get-value compute/region) \
  --format 'value(address)')
```

创建 Kubernetes API Server 凭证签发请求文件：

```
cat > kubernetes-csr.json <"CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF
```

创建 Kubernetes API Server 凭证与私钥：

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=10.32.0.1,10.240.0.10,10.240.0.11,10.240.0.12,${KUBER
  -profile=kubernetes \
  kubernetes-csr.json | cfssljson -bare kubernetes
```

结果产生以下两个文件：

```
kubernetes-key.pem
kubernetes.pem
```

Service Account 证书

```
at > service-account-csr.json <"CN": "service-accounts",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Kubernetes The Hard Way",
      "ST": "Oregon"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  service-account-csr.json | cfssljson -bare service-account
```

结果将生成以下两个文件

```
service-account-key.pem
service-account.pem
```

分发客户端和服务器证书

将客户端凭证以及私钥复制到每个工作节点上：

```
for instance in worker-0 worker-1 worker-2; do
  gcloud compute scp ca.pem ${instance}-key.pem ${instance}.pem $
done
```

将服务器凭证以及私钥复制到每个控制节点上：

```
for instance in controller-0 controller-1 controller-2; do
  gcloud compute scp ca.pem ca-key.pem kubernetes-key.pem kuberne
    service-account-key.pem service-account.pem ${instance}:~/
done
```

> `kube-proxy`、`kube-controller-manager`、`kube-scheduler` 和 `kubelet` 客户端凭证将会在下一节中用来创建客户端签发请求文件。

下一步：配置和生成 Kubernetes 配置文件。

配置生成配置

本部分内容将会创建 kubeconfig 配置文件，它们是 Kubernetes 客户端与 API Server 认证与鉴权的保证。

客户端认证配置

本节将会创建用于 `kube-proxy`、`kube-controller-manager`、`kube-scheduler` 和 `kubelet` 的 kubeconfig 文件。

Kubernetes 公有 IP 地址

每一个 kubeconfig 文件都需要一个 Kuberntes API Server 的 IP 地址。为了保证高可用性，我们将该 IP 分配给 API Server 之前的外部负载均衡器。

查询 `kubernetes-the-hard-way` 的静态 IP 地址：

```
KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute addresses describe kub
  --region $(gcloud config get-value compute/region) \
  --format 'value(address)')
```

kubelet 配置文件

为了确保 Node Authorizer 授权，Kubelet 配置文件中的客户端证书必需匹配 Node 名字。

为每个 worker 节点创建 kubeconfig 配置：

```
for instance in worker-0 worker-1 worker-2; do
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://${KUBERNETES_PUBLIC_ADDRESS}:6443 \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-credentials system:node:${instance} \
    --client-certificate=${instance}.pem \
    --client-key=${instance}-key.pem \
    --embed-certs=true \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:node:${instance} \
    --kubeconfig=${instance}.kubeconfig

  kubectl config use-context default --kubeconfig=${instance}.kub
done
```

结果将会生成以下 3 个文件：

```
worker-0.kubeconfig
worker-1.kubeconfig
worker-2.kubeconfig
```

kube-proxy 配置文件

为 kube-proxy 服务生成 kubeconfig 配置文件：

```
{
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://${KUBERNETES_PUBLIC_ADDRESS}:6443 \
    --kubeconfig=kube-proxy.kubeconfig

  kubectl config set-credentials system:kube-proxy \
    --client-certificate=kube-proxy.pem \
    --client-key=kube-proxy-key.pem \
    --embed-certs=true \
    --kubeconfig=kube-proxy.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:kube-proxy \
    --kubeconfig=kube-proxy.kubeconfig

  kubectl config use-context default --kubeconfig=kube-proxy.kube
}
```

kube-controller-manager 配置文件

```
{
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://127.0.0.1:6443 \
    --kubeconfig=kube-controller-manager.kubeconfig

  kubectl config set-credentials system:kube-controller-manager \
    --client-certificate=kube-controller-manager.pem \
    --client-key=kube-controller-manager-key.pem \
    --embed-certs=true \
    --kubeconfig=kube-controller-manager.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:kube-controller-manager \
    --kubeconfig=kube-controller-manager.kubeconfig

  kubectl config use-context default --kubeconfig=kube-controller
}
```

kube-scheduler 配置文件

```
{
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://127.0.0.1:6443 \
    --kubeconfig=kube-scheduler.kubeconfig

  kubectl config set-credentials system:kube-scheduler \
    --client-certificate=kube-scheduler.pem \
    --client-key=kube-scheduler-key.pem \
    --embed-certs=true \
    --kubeconfig=kube-scheduler.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:kube-scheduler \
    --kubeconfig=kube-scheduler.kubeconfig

  kubectl config use-context default --kubeconfig=kube-scheduler.
}
```

Admin 配置文件

```
{
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://127.0.0.1:6443 \
    --kubeconfig=admin.kubeconfig

  kubectl config set-credentials admin \
    --client-certificate=admin.pem \
    --client-key=admin-key.pem \
    --embed-certs=true \
    --kubeconfig=admin.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=admin \
    --kubeconfig=admin.kubeconfig

  kubectl config use-context default --kubeconfig=admin.kubeconfi
}
```

分发配置文件

将 `kubelet` 与 `kube-proxy` kubeconfig 配置文件复制到每个 worker 节点上：

```
for instance in worker-0 worker-1 worker-2; do
  gcloud compute scp ${instance}.kubeconfig kube-proxy.kubeconfig
done
```

将 `admin`、`kube-controller-manager` 与 `kube-scheduler` kubeconfig 配置文件复制到每个 controller 节点上：

```
for instance in controller-0 controller-1 controller-2; do
  gcloud compute scp admin.kubeconfig kube-controller-manager.kub
done
```

下一步：配置和生成密钥。

配置生成密钥

Kubernetes 存储了集群状态、应用配置和密钥等很多不同的数据。而 Kubernetes 也支持集群数据的加密存储。

本部分将会创建加密密钥以及一个用于加密 Kubernetes Secrets 的 加密配置文件。

加密密钥

建立加密密钥：

```
ENCRYPTION_KEY=$(head -c 32 /dev/urandom | base64)
```

加密配置文件

生成名为 `encryption-config.yaml` 的加密配置文件：

```
cat > encryption-config.yaml <${ENCRYPTION_KEY}
      - identity: {}
EOF
```

将 `encryption-config.yaml` 复制到每个控制节点上：

```
for instance in controller-0 controller-1 controller-2; do
  gcloud compute scp encryption-config.yaml ${instance}:~/
done
```

下一步：部署 etcd 群集。

部署Etcd群集

Kubernetes 组件都是无状态的，所有的群集状态都储存在 etcd 集群中。

本部分内容将部署一套三节点的 etcd 群集，并配置高可用以及远程加密访问。

事前准备

本部分的命令需要在每个控制节点上都运行以便，包括 `controller-0` 、 `controller-1` 和 `controller-2` 。可以使用 `gcloud` 命令登录每个控制节点，比如

```
gcloud compute ssh controller-0
```

可以使用 `tmux` 同时登录到三点控制节点上，加快部署步骤。

部署 etcd 集群成员

下载并安装 etcd 二进制文件

从 coreos/etcd GitHub 中下载 etcd 发布文件：

```
wget -q --show-progress --https-only --timestamping \
  "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v
```

解压缩并安装 `etcd` 服务与 `etcdctl` 命令行工具：

```
  tar -xvf etcd-v3.3.9-linux-amd64.tar.gz
  sudo mv etcd-v3.3.9-linux-amd64/etcd* /usr/local/bin/
```

配置 etcd Server

```
  sudo mkdir -p /etc/etcd /var/lib/etcd
  sudo cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```

使用虚拟机的内网 IP 地址来作为 etcd 集群的服务地址。查询当前节点的内网 IP 地址：

```
INTERNAL_IP=$(curl -s -H "Metadata-Flavor: Google" \
  http://metadata.google.internal/computeMetadata/v1/instance/net
```

每个 etcd 成员必须有一个整集群中唯一的名字，使用 hostname 作为 etcd name：

```
ETCD_NAME=$(hostname -s)
```

生成 `etcd.service` 的 systemd 配置文件

```
cat <local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster controller-0=https://10.240.0.10:2380,control
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

启动 etcd Server

```
sudo systemctl daemon-reload
sudo systemctl enable etcd
sudo systemctl start etcd
```

> 不要忘记在所有控制节点上都运行上述命令，包括 `controller-0` 、 `controller-1` 和 `controller-2` 等。

验证

列出 etcd 的群集成员：

```
sudo ETCDCTL_API=3 etcdctl member list \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/etcd/ca.pem \
  --cert=/etc/etcd/kubernetes.pem \
  --key=/etc/etcd/kubernetes-key.pem
```

> 输出

```
3a57933972cb5131, started, controller-2, https://10.240.0.12:2380
f98dc20bce6225a0, started, controller-0, https://10.240.0.10:2380
ffed16798470cab5, started, controller-1, https://10.240.0.11:2380
```

下一步：部署 Kubernetes 控制节点。

部署控制节点

本部分将会在三台控制节点上部署 Kubernetes 控制服务，并配置高可用的集群架构。并且还会创建一个用于外部访问的负载均衡器。每个控制节点上需要部署的服务包括：Kubernetes API Server、Scheduler 以及 Controller Manager 等。

事前准备

以下命令需要在每台控制节点上面都运行一遍，包括 `controller-0` 、 `controller-1` 和 `controller-2` 。可以使用 `gcloud` 命令登录每个控制节点。例如：

```
gcloud compute ssh controller-0
```

可以使用 `tmux` 同时登录到三点控制节点上，加快部署步骤。

部署 Kubernetes 控制平面

创建 Kubernetes 配置目录

```
sudo mkdir -p /etc/kubernetes/config
```

下载并安装 Kubernetes Controller 二进制文件

```
wget -q --show-progress --https-only --timestamping \
  "https://storage.googleapis.com/kubernetes-release/release/v1.1
  "https://storage.googleapis.com/kubernetes-release/release/v1.1
  "https://storage.googleapis.com/kubernetes-release/release/v1.1
  "https://storage.googleapis.com/kubernetes-release/release/v1.1

chmod +x kube-apiserver kube-controller-manager kube-scheduler ku
sudo mv kube-apiserver kube-controller-manager kube-scheduler kub
```

配置 Kubernetes API Server

```
  sudo mkdir -p /var/lib/kubernetes/

  sudo mv ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem \
    service-account-key.pem service-account.pem \
    encryption-config.yaml /var/lib/kubernetes/
```

使用节点的内网 IP 地址作为 API server 与集群内部成员的广播地址。
首先查询当前节点的内网 IP 地址：

```
INTERNAL_IP=$(curl -s -H "Metadata-Flavor: Google" \
  http://metadata.google.internal/computeMetadata/v1/instance/net
```

生成 `kube-apiserver.service` systemd 配置文件：

```
cat <local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --audit-log-maxage=30 \\
  --audit-log-maxbackup=3 \\
  --audit-log-maxsize=100 \\
  --audit-log-path=/var/log/audit.log \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-admission-plugins=Initializers,NamespaceLifecycle,Node
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --etcd-servers=https://10.240.0.10:2379,https://10.240.0.11:237
  --event-ttl=1h \\
  --experimental-encryption-provider-config=/var/lib/kubernetes/e
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
  --kubelet-https=true \\
  --runtime-config=api/all \\
  --service-account-key-file=/var/lib/kubernetes/service-account.
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

配置 Kubernetes Controller Manager

生成 `kube-controller-manager.service` systemd 配置文件：

```
sudo mv kube-controller-manager.kubeconfig /var/lib/kubernetes/

cat <local/bin/kube-controller-manager \\
  --address=0.0.0.0 \\
  --cluster-cidr=10.200.0.0/16 \\
  --cluster-name=kubernetes \\
  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \\
  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \\
  --kubeconfig=/var/lib/kubernetes/kube-controller-manager.kubecc
  --leader-elect=true \\
  --root-ca-file=/var/lib/kubernetes/ca.pem \\
  --service-account-private-key-file=/var/lib/kubernetes/service-
  --service-cluster-ip-range=10.32.0.0/24 \\
  --use-service-account-credentials=true \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

配置 Kubernetes Scheduler

生成 `kube-scheduler.service` systemd 配置文件：

```
sudo mv kube-scheduler.kubeconfig /var/lib/kubernetes/

cat <"/var/lib/kubernetes/kube-scheduler.kubeconfig"
leaderElection:
  leaderElect: true
EOF

cat <local/bin/kube-scheduler \\
  --config=/etc/kubernetes/config/kube-scheduler.yaml \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

启动控制器服务

```
  sudo systemctl daemon-reload
  sudo systemctl enable kube-apiserver kube-controller-manager ku
  sudo systemctl start kube-apiserver kube-controller-manager kub
```

> 请等待 10 秒以便 Kubernetes API Server 初始化。

开启 HTTP 健康检查

Google Network Load Balancer 将用在在三个 API Server 之前作负载均衡，并可以终止 TLS 并验证客户端证书。但是该负载均衡仅支持 HTTP 健康检查，因而这里部署 nginx 来代理 API Server 的 `/healthz` 连接。

> `/healthz` API 默认不需要认证。

```
sudo apt-get install -y nginx

cat > kubernetes.default.svc.cluster.local <<s /etc/nginx/sites-a

sudo systemctl restart nginx
sudo systemctl enable nginx
```

验证

```
kubectl get componentstatuses --kubeconfig admin.kubeconfig
```

将输出结果

```
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-2               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
etcd-1               Healthy   {"health": "true"}
```

验证 Nginx HTTP 健康检查

```
curl -H "Host: kubernetes.default.svc.cluster.local" -i http://12
```

将输出

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Mon, 14 May 2018 13:45:39 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 2
Connection: keep-alive

ok
```

> 记得在每台控制节点上面都运行一遍，包括 `controller-0` 、 `controller-1` 和 `controller-2` 。

Kubelet RBAC 授权

本节将会配置 API Server 访问 Kubelet API 的 RBAC 授权。访问 Kubelet API 是获取 metrics、日志以及执行容器命令所必需的。

> 这里设置 Kubeket `--authorization-mode` 为 `Webhook` 模式。Webhook 模式使用 `SubjectAccessReview` API 来决定授权。

```
gcloud compute ssh controller-0
```

创建 `system:kube-apiserver-to-kubelet` `ClusterRole` 以允许请求 Kubelet API 和执行许用来管理 Pods 的任务：

```
cat <-f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-apiserver-to-kubelet
rules:
  - apiGroups:
      - ""
    resources:
      - nodes/proxy
      - nodes/stats
      - nodes/log
      - nodes/spec
      - nodes/metrics
    verbs:
      - "*"
EOF
```

Kubernetes API Server 使用客户端凭证授权 Kubelet 为 `kubernetes` 用户，此凭证用 `--kubelet-client-certificate` flag 来定义。

绑定 `system:kube-apiserver-to-kubelet` ClusterRole 到 `kubernetes` 用户：

```
cat <-f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: system:kube-apiserver
  namespace: ""
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-apiserver-to-kubelet
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: kubernetes
EOF
```

Kubernetes 前端负载均衡器

本节将会建立一个位于 Kubernetes API Servers 前端的外部负载均衡器。 `kubernetes-the-hard-way` 静态 IP 地址将会配置在这个负载均衡器上。

> 本指南创建的虚拟机内部并没有操作负载均衡器的权限，需要到创建这些虚拟机的那台机器上去做下面的操作。

创建外部负载均衡器网络资源：

```
  KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute addresses describe k
    --region $(gcloud config get-value compute/region) \
    --format 'value(address)')

  gcloud compute http-health-checks create kubernetes \
    --description "Kubernetes Health Check" \
    --host "kubernetes.default.svc.cluster.local" \
    --request-path "/healthz"

  gcloud compute firewall-rules create kubernetes-the-hard-way-al
    --network kubernetes-the-hard-way \
    --source-ranges 209.85.152.0/22,209.85.204.0/22,35.191.0.0/16
    --allow tcp

  gcloud compute target-pools create kubernetes-target-pool \
    --http-health-check kubernetes

  gcloud compute target-pools add-instances kubernetes-target-poo
   --instances controller-0,controller-1,controller-2

  gcloud compute forwarding-rules create kubernetes-forwarding-ru
    --address ${KUBERNETES_PUBLIC_ADDRESS} \
    --ports 6443 \
    --region $(gcloud config get-value compute/region) \
    --target-pool kubernetes-target-pool
```

验证

查询 `kubernetes-the-hard-way` 静态 IP 地址：

```
KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute addresses describe kub
  --region $(gcloud config get-value compute/region) \
  --format 'value(address)')
```

发送一个查询 Kubernetes 版本信息的 HTTP 请求

```
curl --cacert ca.pem https://${KUBERNETES_PUBLIC_ADDRESS}:6443/ve
```

结果为

```
{
  "major": "1",
  "minor": "12",
  "gitVersion": "v1.12.0",
  "gitCommit": "0ed33881dc4355495f623c6f22e7dd0b7632b7c0",
  "gitTreeState": "clean",
  "buildDate": "2018-09-27T16:55:41Z",
  "goVersion": "go1.10.4",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

下一步：部署 Kubernetes Worker 节点。

部署计算节点

本部分将会部署三个 Kubernetes Worker 节点。每个节点上将会安装以下服务：runc，gVisor，container networking plugins，containerd，kubelet，和 kube-proxy。

事前准备

以下命令需要在所有 worker 节点上面都运行一遍，包括 `worker-0`，`worker-1` 和 `worker-2`。可以使用 `gcloud` 命令登录到 worker 节点上，比如

```
gcloud compute ssh worker-0
```

可以使用 tmux 同时登录到三个 Worker 节点上，加快部署步骤。

部署 Kubernetes Worker 节点

安装 OS 依赖组件：

```
sudo apt-get update
sudo apt-get -y install socat conntrack ipset
```

> socat 命令用于支持 `kubectl port-forward` 命令。

下载并安装 worker 二进制文件

```
wget -q --show-progress --https-only --timestamping \
  https://github.com/kubernetes-sigs/cri-tools/releases/download/
  https://storage.googleapis.com/kubernetes-the-hard-way/runsc-50
  https://github.com/opencontainers/runc/releases/download/v1.0.0
  https://github.com/containernetworking/plugins/releases/downloa
  https://github.com/containerd/containerd/releases/download/v1.2
  https://storage.googleapis.com/kubernetes-release/release/v1.12
  https://storage.googleapis.com/kubernetes-release/release/v1.12
  https://storage.googleapis.com/kubernetes-release/release/v1.12
```

创建安装目录：

```
sudo mkdir -p \
  /etc/cni/net.d \
  /opt/cni/bin \
  /var/lib/kubelet \
  /var/lib/kube-proxy \
  /var/lib/kubernetes \
  /var/run/kubernetes
```

安装 worker 二进制文件

```
  sudo mv runsc-50c283b9f56bb7200938d9e207355f05f79f0d17 runsc
  sudo mv runc.amd64 runc
  chmod +x kubectl kube-proxy kubelet runc runsc
  sudo mv kubectl kube-proxy kubelet runc runsc /usr/local/bin/
  sudo tar -xvf crictl-v1.12.0-linux-amd64.tar.gz -C /usr/local/b
  sudo tar -xvf cni-plugins-amd64-v0.6.0.tgz -C /opt/cni/bin/
  sudo tar -xvf containerd-1.2.0-rc.0.linux-amd64.tar.gz -C /
```

配置 CNI 网路

查询当前计算节点的 Pod CIDR 范围：

```
POD_CIDR=$(curl -s -H "Metadata-Flavor: Google" \
  http://metadata.google.internal/computeMetadata/v1/instance/att
```

生成 `bridge` 网络插件配置文件

```
cat <"cniVersion": "0.3.1",
    "name": "bridge",
    "type": "bridge",
    "bridge": "cnio0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "ranges": [
          [{"subnet": "${POD_CIDR}"}]
        ],
        "routes": [{"dst": "0.0.0.0/0"}]
    }
}
EOF
```

生成 `loopback` 网络插件配置文件

```
cat <"cniVersion": "0.3.1",
    "type": "loopback"
}
EOF
```

配置 containerd

```
sudo mkdir -p /etc/containerd/

# Untrusted workloads will be run using the gVisor (runsc) runtim
cat << EOF | sudo tee /etc/containerd/config.toml
[plugins]
  [plugins.cri.containerd]
    snapshotter = "overlayfs"
    [plugins.cri.containerd.default_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runc"
      runtime_root = ""
    [plugins.cri.containerd.untrusted_workload_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runsc"
      runtime_root = "/run/containerd/runsc"
    [plugins.cri.containerd.gvisor]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runsc"
      runtime_root = "/run/containerd/runsc"
EOF

# Create the containerd.service systemd unit file
cat <
```

配置 Kubelet

```
  sudo mv ${HOSTNAME}-key.pem ${HOSTNAME}.pem /var/lib/kubelet/
  sudo mv ${HOSTNAME}.kubeconfig /var/lib/kubelet/kubeconfig
  sudo mv ca.pem /var/lib/kubernetes/
```

生成 `kubelet.service` systemd 配置文件：

```
# The resolvConf configuration is used to avoid loops
# when using CoreDNS for service discovery on systems running systemd-resolv
cat <false
  webhook:
    enabled: true
  x509:
    clientCAFile: "/var/lib/kubernetes/ca.pem"
authorization:
  mode: Webhook
clusterDomain: "cluster.local"
clusterDNS:
  - "10.32.0.10"
podCIDR: "${POD_CIDR}"
resolvConf: "/run/systemd/resolve/resolv.conf"
runtimeRequestTimeout: "15m"
tlsCertFile: "/var/lib/kubelet/${HOSTNAME}.pem"
tlsPrivateKeyFile: "/var/lib/kubelet/${HOSTNAME}-key.pem"
EOF

cat <local/bin/kubelet \\
  --config=/var/lib/kubelet/kubelet-config.yaml \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

配置 Kube-Proxy

```
sudo mv kube-proxy.kubeconfig /var/lib/kube-proxy/kubeconfig
```

生成 `kube-proxy.service` systemd 配置文件：

```
cat <"/var/lib/kube-proxy/kubeconfig"
mode: "iptables"
clusterCIDR: "10.200.0.0/16"
EOF

cat <local/bin/kube-proxy \\
  --config=/var/lib/kube-proxy/kube-proxy-config.yaml
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

启动 worker 服务

```
  sudo systemctl daemon-reload
  sudo systemctl enable containerd kubelet kube-proxy
  sudo systemctl start containerd kubelet kube-proxy
```

> 记得在所有 worker 节点上面都运行一遍，包括 `worker-0`，`worker-1` 和 `worker-2`。

验证

登入任意一台控制节点查询 Nodes 列表

```
gcloud compute ssh controller-0 \
  --command "kubectl get nodes --kubeconfig admin.kubeconfig"
```

输出为

```
NAME       STATUS   ROLES    AGE   VERSION
worker-0   Ready       35s   v1.12.0
worker-1   Ready       36s   v1.12.0
worker-2   Ready       36s   v1.12.0
```

下一步：配置 Kubectl。

配置Kubectl

本部分将生成一个用于 admin 用户的 kubeconfig 文件。

> 注意：在生成 admin 客户端证书的目录来运行本部分的指令。

admin kubeconfig

每一个 kubeconfig 都需要一个 Kuberntes API Server 地址。为了保证高可用，这里将使用 API Servers 前端外部负载均衡器的 IP 地址。

查询 `kubernetes-the-hard-way` 的静态 IP 地址：

```
KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute addresses describe kub
    --region $(gcloud config get-value compute/region) \
    --format 'value(address)')
```

为 `admin` 用户生成 kubeconfig 文件：

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${KUBERNETES_PUBLIC_ADDRESS}:6443

kubectl config set-credentials admin \
  --client-certificate=admin.pem \
  --client-key=admin-key.pem

kubectl config set-context kubernetes-the-hard-way \
  --cluster=kubernetes-the-hard-way \
  --user=admin

kubectl config use-context kubernetes-the-hard-way
```

验证

检查远端 Kubernetes 群集的健康状况：

```
kubectl get componentstatuses
```

输出为

```
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-2               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
etcd-1               Healthy   {"health": "true"}
```

列出远端 kubernetes cluster 的节点：

```
kubectl get nodes
```

输出为

```
NAME       STATUS   ROLES    AGE    VERSION
worker-0   Ready       117s   v1.12.0
worker-1   Ready       118s   v1.12.0
worker-2   Ready       118s   v1.12.0
```

下一步：配置 Pod 网络路由。

配置网络路由

每个 Pod 都会从所在 Node 的 Pod CIDR 中分配一个 IP 地址。由于网络 路由 还没有配置，跨节点的 Pod 之间还无法通信。

本部分将为每个 worker 节点创建一条路由，将匹配 Pod CIDR 的网络请求路由到 Node 的内网 IP 地址上。

> 也可以选择 其他方法 来实现 Kubernetes 网络模型。

路由表

本节将为创建 `kubernetes-the-hard-way` VPC 路由收集必要的信息。

列出每个 worker 节点的内部 IP 地址和 Pod CIDR 范围：

```
for instance in worker-0 worker-1 worker-2; do
  gcloud compute instances describe ${instance} \
    --format 'value[separator=" "](networkInterfaces[0].networkIP
done
```

输出为

```
10.240.0.20 10.200.0.0/24
10.240.0.21 10.200.1.0/24
10.240.0.22 10.200.2.0/24
```

路由

为每个 worker 节点创建网络路由：

```
for i in 0 1 2; do
  gcloud compute routes create kubernetes-route-10-200-${i}-0-24
    --network kubernetes-the-hard-way \
    --next-hop-address 10.240.0.2${i} \
    --destination-range 10.200.${i}.0/24
done
```

列出 `kubernetes-the-hard-way` VPC 网络的路由表：

```
gcloud compute routes list --filter "network: kubernetes-the-hard
```

输出为

```
NAME                            NETWORK                  DEST_RAN
default-route-081879136902de56  kubernetes-the-hard-way  10.240.0
default-route-55199a5aa126d7aa  kubernetes-the-hard-way  0.0.0.0/
kubernetes-route-10-200-0-0-24  kubernetes-the-hard-way  10.200.0
kubernetes-route-10-200-1-0-24  kubernetes-the-hard-way  10.200.1
kubernetes-route-10-200-2-0-24  kubernetes-the-hard-way  10.200.2
```

下一步：部署 DNS 扩展。

部署DNS扩展

本部分将部署 DNS 扩展，用于为集群内的应用提供服务发现。

DNS 扩展

部属 `kube-dns` 群集扩展：

```
kubectl apply -f https://storage.googleapis.com/kubernetes-the-ha
```

输出为

```
serviceaccount/coredns created
clusterrole.rbac.authorization.k8s.io/system:coredns created
clusterrolebinding.rbac.authorization.k8s.io/system:coredns creat
configmap/coredns created
deployment.extensions/coredns created
service/kube-dns created
```

列出 `kube-dns` 部署的 Pod 列表：

```
kubectl get pods -l k8s-app=kube-dns -n kube-system
```

输出为

```
NAME                       READY   STATUS    RESTARTS   AGE
coredns-699f8ddd77-94qv9   1/1     Running   0          20s
coredns-699f8ddd77-gtcgb   1/1     Running   0          20s
```

验证

建立一个 `busybox` 部署：

```
kubectl run busybox --image=busybox --command -- sleep 3600
```

列出 `busybox` 部署的 Pod：

```
kubectl get pods -l run=busybox
```

输出为

```
NAME                       READY     STATUS    RESTARTS   AGE
busybox-2125412808-mt2vb   1/1       Running   0          15s
```

查询 `busybox` Pod 的全名：

```
POD_NAME=$(kubectl get pods -l run=busybox -o jsonpath="{.items[0
```

在 `busybox` Pod 中查询 DNS：

```
kubectl exec -ti $POD_NAME -- nslookup kubernetes
```

输出为

```
Server:    10.32.0.10
Address 1: 10.32.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes
Address 1: 10.32.0.1 kubernetes.default.svc.cluster.local
```

下一步：烟雾测试。

烟雾测试

本部分将会运行一系列的测试来验证 Kubernetes 集群的功能正常。

数据加密

本节将会验证 encrypt secret data at rest 的功能。

创建一个 Secret：

```
kubectl create secret generic kubernetes-the-hard-way \
  --from-literal="mykey=mydata"
```

查询存在 etcd 里 16 进位编码的 `kubernetes-the-hard-way` secret：

```
gcloud compute ssh controller-0 \
  --command "sudo ETCDCTL_API=3 etcdctl get \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/etcd/ca.pem \
  --cert=/etc/etcd/kubernetes.pem \
  --key=/etc/etcd/kubernetes-key.pem\
  /registry/secrets/default/kubernetes-the-hard-way | hexdump -C"
```

输出为

```
00000000  2f 72 65 67 69 73 74 72  79 2f 73 65 63 72 65 74  |/reg
00000010  73 2f 64 65 66 61 75 6c  74 2f 6b 75 62 65 72 6e  |s/de
00000020  65 74 65 73 2d 74 68 65  2d 68 61 72 64 2d 77 61  |etes
00000030  79 0a 6b 38 73 3a 65 6e  63 3a 61 65 73 63 62 63  |y.k8
00000040  3a 76 31 3a 6b 65 79 31  3a ea 7c 76 32 43 62 6f  |:v1:
00000050  44 02 02 8c b7 ca fe 95  a5 33 f6 a1 18 6c 3d 53  |D...
00000060  e7 9c 51 ee 32 f6 e4 17  ea bb 11 d5 2f e2 40 00  |..Q.
00000070  ae cf d9 e7 ba 7f 68 18  d3 c1 10 10 93 43 35 bd  |....
00000080  24 dd 66 b4 f8 f9 82 77  4a d5 78 03 19 41 1e bc  |$.f.
00000090  94 3f 17 41 ad cc 8c ba  9f 8f 8e 56 97 7e 96 fb  |.?.A
000000a0  8f 2e 6a a5 bf 08 1f 0b  c3 4b 2b 93 d1 ec f8 70  |..j.
000000b0  c1 e4 1d 1a d2 0d f8 74  3a a1 4f 3c e0 c9 6d 3f  |....
000000c0  de a3 f5 fd 76 aa 5e bc  27 d9 3c 6b 8f 54 97 45  |....
```

Etcd 的密钥以 `k8s:enc:aescbc:v1:key1` 为前缀，表示使用密钥为 `key1` 的 `aescbc` 加密数据。

部署

本节将会验证建立与管理 Deployments 的功能。

创建一个 Deployment 用来搭建 nginx Web 服务：

```
kubectl run nginx --image=nginx
```

列出 `nginx` deployment 的 pods:

```
kubectl get pods -l run=nginx
```

输出为

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-4217019353-b5gzn   1/1       Running   0          15s
```

端口转发

本节将会验证使用 port forwarding 从远端进入容器的功能。

查询 `nginx` pod 的全名：

```
POD_NAME=$(kubectl get pods -l run=nginx -o jsonpath="{.items[0].
```

将本地机器的 8080 端口转发到 nginx pod 的 80 端口：

```
kubectl port-forward $POD_NAME 8080:80
```

输出为

```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

开一个新的终端来做 HTTP 请求测试：

```
curl --head http://127.0.0.1:8080
```

输出为

```
HTTP/1.1 200 OK
Server: nginx/1.13.5
Date: Mon, 02 Oct 2017 01:04:20 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 08 Aug 2017 15:25:00 GMT
Connection: keep-alive
ETag: "5989d7cc-264"
Accept-Ranges: bytes
```

回到前面的终端并按下 `Ctrl + C` 停止 port forwarding 命令：

```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
^C
```

容器日志

本节会验证 获取容器日志 的功能。

输出 nginx Pod 的容器日志：

```
kubectl logs $POD_NAME
```

输出为

```
127.0.0.1 - - [02/Oct/2017:01:04:20 +0000] "HEAD / HTTP/1.1" 200
```

执行容器命令

本节将验证 在容器里执行命令 的功能。

使用 `nginx -v` 命令在 `nginx` Pod 中输出 nginx 的版本：

```
kubectl exec -ti $POD_NAME -- nginx -v
```

输出为

```
nginx version: nginx/1.13.7
```

服务（Service）

本节将验证 Kubernetes Service。

将 `nginx` 部署导出为 NodePort 类型的 Service：

```
kubectl expose deployment nginx --port 80 --type NodePort
```

> LoadBalancer 类型的 Service 不能使用是因为没有设置 cloud provider 集成。 设定 cloud provider 不在本教程范围之内。

查询 `nginx` 服务分配的 Node Port：

```
NODE_PORT=$(kubectl get svc nginx \
  --output=jsonpath='{range .spec.ports[0]}{.nodePort}')
```

建立防火墙规则允许外网访问该 Node 端口：

```
gcloud compute firewall-rules create kubernetes-the-hard-way-allo
  --allow=tcp:${NODE_PORT} \
  --network kubernetes-the-hard-way
```

查询 worker 节点的外网 IP 地址：

```
EXTERNAL_IP=$(gcloud compute instances describe worker-0 \
  --format 'value(networkInterfaces[0].accessConfigs[0].natIP)')
```

对得到的外网 IP 地址 + nginx 服务的 Node Port 做 HTTP 请求测试：

```
curl -I http://${EXTERNAL_IP}:${NODE_PORT}
```

输出为

```
HTTP/1.1 200 OK
Server: nginx/1.13.7
Date: Mon, 18 Dec 2017 14:52:09 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 21 Nov 2017 14:28:04 GMT
Connection: keep-alive
ETag: "5a1437f4-264"
Accept-Ranges: bytes
```

非可信应用

非可信应用可以运行在 gVisor 容器引擎之中。

```
cat <<f -
apiVersion: v1
kind: Pod
metadata:
  name: untrusted
  annotations:
    io.kubernetes.cri.untrusted-workload: "true"
spec:
  containers:
    - name: webserver
      image: gcr.io/hightowerlabs/helloworld:2.0.0
EOF
```

验证

```
kubectl get pods -o wide
```

```
NAME                       READY     STATUS    RESTARTS   AGE
busybox-68654f944b-djjjb   1/1       Running   0          5m
nginx-65899c769f-xkfcn     1/1       Running   0          4m
untrusted                  1/1       Running   0          10s
```

查看 untrusted Pod 运行信息

```
# SSH to the node
INSTANCE_NAME=$(kubectl get pod untrusted --output=jsonpath='{.sp
gcloud compute ssh ${INSTANCE_NAME}

# List the containers running under gVisor
sudo runsc --root  /run/containerd/runsc/k8s.io list
```

输出

```
I0514 14:03:56.108368   14988 x:0] ***************************
I0514 14:03:56.108548   14988 x:0] Args: [runsc --root /run/conta
I0514 14:03:56.108730   14988 x:0] Git Revision: 08879266fef3a67f
I0514 14:03:56.108787   14988 x:0] PID: 14988
I0514 14:03:56.108838   14988 x:0] UID: 0, GID: 0
I0514 14:03:56.108877   14988 x:0] Configuration:
I0514 14:03:56.108912   14988 x:0]              RootDir: /run/cor
I0514 14:03:56.109000   14988 x:0]              Platform: ptrace
I0514 14:03:56.109080   14988 x:0]              FileAccess: proxy
I0514 14:03:56.109159   14988 x:0]              Network: sandbox,
I0514 14:03:56.109238   14988 x:0]              Strace: false, ma
I0514 14:03:56.109315   14988 x:0] ***************************
ID
3528c6b270c76858e15e10ede61bd1100b77519e7c9972d51b370d6a3c60adbb
7ff747c919c2dcf31e64d7673340885138317c91c7c51ec6302527df680ba981
I0514 14:03:56.111287   14988 x:0] Exiting with status: 0
```

查询容器中的进程

```
POD_ID=$(sudo crictl -r unix:///var/run/containerd/containerd.soc
  pods --name untrusted -q)

CONTAINER_ID=$(sudo crictl -r unix:///var/run/containerd/containe
  ps -p ${POD_ID} -q)

sudo runsc --root /run/containerd/runsc/k8s.io ps ${CONTAINER_ID}
```

输出

```
I0514 14:05:16.499237   15096 x:0] ***************************
I0514 14:05:16.499542   15096 x:0] Args: [runsc --root /run/conta
I0514 14:05:16.499597   15096 x:0] Git Revision: 08879266fef3a67f
I0514 14:05:16.499644   15096 x:0] PID: 15096
I0514 14:05:16.499695   15096 x:0] UID: 0, GID: 0
I0514 14:05:16.499734   15096 x:0] Configuration:
I0514 14:05:16.499769   15096 x:0]              RootDir: /run/cor
I0514 14:05:16.499880   15096 x:0]              Platform: ptrace
I0514 14:05:16.499962   15096 x:0]              FileAccess: proxy
I0514 14:05:16.500042   15096 x:0]              Network: sandbox,
I0514 14:05:16.500120   15096 x:0]              Strace: false, ma
I0514 14:05:16.500197   15096 x:0] ***************************
UID       PID       PPID      C         STIME     TIME      CMD
0         1         0         0         14:02     40ms      app
I0514 14:05:16.501354   15096 x:0] Exiting with status: 0
```

下一步：删除集群。

删除集群

本部分将删除该教程所创建的全部计算资源。

计算节点

删除所有的控制节点和 `worker` 节点：

```
gcloud -q compute instances delete \
  controller-0 controller-1 controller-2 \
  worker-0 worker-1 worker-2
```

网路

删除外部负载均衡器以及网络资源：

```
gcloud -q compute forwarding-rules delete kubernetes-forwarding-r
    --region $(gcloud config get-value compute/region)
gcloud -q compute target-pools delete kubernetes-target-pool
gcloud -q compute http-health-checks delete kubernetes
gcloud -q compute addresses delete kubernetes-the-hard-way
```

删除 `kubernetes-the-hard-way` 防火墙规则：

```
gcloud -q compute firewall-rules delete \
  kubernetes-the-hard-way-allow-nginx-service \
  kubernetes-the-hard-way-allow-internal \
  kubernetes-the-hard-way-allow-external \
  kubernetes-the-hard-way-allow-health-check
```

删除 Pod 网络路由：

```
gcloud -q compute routes delete \
    kubernetes-route-10-200-0-0-24 \
    kubernetes-route-10-200-1-0-24 \
    kubernetes-route-10-200-2-0-24
```

删除 `kubernetes` 子网：

```
gcloud -q compute networks subnets delete kubernetes
```

删除 `kubernetes-the-hard-way` 网络 VPC：

```
gcloud -q compute networks delete kubernetes-the-hard-way
```

插件扩展

API扩展

Kubernetes 的架构非常灵活，提供了从 API、认证授权、准入控制、网络、存储、运行时以及云平台等一系列的扩展机制，方便用户无侵入的扩展集群的功能。

从 API 的角度来说，可以通过 Aggregation 和 CustomResourceDefinition（CRD） 等扩展 Kubernetes API。

- API Aggregation 允许在不修改 Kubernetes 核心代码的同时将第三方服务注册到 Kubernetes API 中，这样就可以通过 Kubernetes API 来访问外部服务。
- CustomResourceDefinition 则可以在集群中新增资源对象，并可以与已有资源对象（如 Pod、Deployment 等）相同的方式来管理它们。

CRD 相比 Aggregation 更易用，两者对比如下

CRDs	**Aggregated API**
无需编程即可使用 CRD 管理资源	需要使用 Go 来构建 Aggregated APIserver
不需要运行额外服务，但一般需要一个 CRD 控制器同步和管理这些资源	需要独立的第三方服务
任何缺陷都会在 Kubernetes 核心中修复	可能需要定期从 Kubernetes 社区同步缺陷修复方法并重新构建 Aggregated APIserver.
无需额外管理版本	需要第三方服务来管理版本

更多的特性对比

Feature	Description	CRDs
Validation	Help users prevent errors and allow you to evolve your API independently of your clients. These features are most useful when there are many clients who can't all update at the same time.	Yes. Most validation can be specified in the CRD using OpenAPI v3.0 validation. Any other validations supported by addition of a Validating Webhook.
Defaulting	See above	Yes, via a Mutating Webhook; Planned, via CRD OpenAPI schema.
Multi-versioning	Allows serving the same object through two API versions. Can help ease API changes like renaming fields. Less important if you control your client versions.	No, but planned

Feature	Description	CRDs
Custom Storage	If you need storage with a different performance mode (for example, time-series database instead of key-value store) or isolation for security (for example, encryption secrets or different	No
Custom Business Logic	Perform arbitrary checks or actions when creating, reading, updating or deleting an object	Yes, using Webhooks.
Scale Subresource	Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource	Yes
Status Subresource	Finer-grained access control: user writes spec section, controller writes status section.Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource)	Yes
Other Subresources	Add operations other than CRUD, such as “logs” or “exec”.	No

Feature	Description	CRDs
strategic-merge-patch	The new endpoints support PATCH with `Content-Type: application/strategic-merge-patch+json`. Useful for updating objects that may be modified both locally, and by the server. For more information, see "Update API Objects in Place Using kubectl patch"	No, but similar functionality planned
Protocol Buffers	The new resource supports clients that want to use Protocol Buffers	No
OpenAPI Schema	Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (in other words, don't put an `int` in a `string` field?)	No, but planned

使用方法

详细的使用方法请参考

- Aggregation
- CustomResourceDefinition

Aggregation

API Aggregation 允许在不修改 Kubernetes 核心代码的同时扩展 Kubernetes API，即将第三方服务注册到 Kubernetes API 中，这样就可以通过 Kubernetes API 来访问外部服务。

> 备注：另外一种扩展 Kubernetes API 的方法是使用 CustomResourceDefinition (CRD)。

何时使用 Aggregation

满足以下条件时使用 API Aggregation	满足以下条件时使用独立 API
Your API is Declarative.	Your API does not fit the Declarative model.
You want your new types to be readable and writable using `kubectl`.	`kubectl` support is not required
You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types.	Kubernetes UI support is not required.
You are developing a new API.	You already have a program that serves your API and works well.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the API Overview.)	You need to have specific REST paths to be compatible with an already defined REST API.
Your resources are naturally scoped to a cluster or to namespaces of a cluster.	Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths.
You want to reuse Kubernetes API support features.	You don't need those features.

开启 API Aggregation

kube-apiserver 增加以下配置

```
--requestheader-client-ca-file=
--requestheader-allowed-names=aggregator
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
--proxy-client-cert-file=
--proxy-client-key-file=
```

如果 `kube-proxy` 没有在 Master 上面运行，还需要配置

```
--enable-aggregator-routing=true
```

创建扩展 API

1. 确保开启 APIService API（默认开启，可用 `kubectl get apiservice` 命令验证）
2. 创建 RBAC 规则
3. 创建一个 namespace，用来运行扩展的 API 服务
4. 创建 CA 和证书，用于 https
5. 创建一个存储证书的 secret
6. 创建一个部署扩展 API 服务的 deployment，并使用上一步的 secret 配置证书，开启 https 服务
7. 创建一个 ClusterRole 和 ClusterRoleBinding
8. 创建一个非 namespace 的 apiservice，注意设置 `spec.caBundle`
9. 运行 `kubectl get` ，正常应该返回 `No resources found.`

可以使用 apiserver-builder 工具自动化上面的步骤。

```
# 初始化项目
$ cd GOPATH/src/github.com/my-org/my-project
$ apiserver-boot init repo --domain
$ apiserver-boot init glide

# 创建资源
$ apiserver-boot create group version resource --group  --versior

# 编译
$ apiserver-boot build executables
$ apiserver-boot build docs

# 本地运行
$ apiserver-boot run local

# 集群运行
$ apiserver-boot run in-cluster --name nameofservicetorun --names
$ kubectl create -f sample/<type>.yaml
```

示例

见 sample-apiserver 和 apiserver-builder/example。

访问控制

Kubernetes 对 API 访问提供了三种安全访问控制措施：认证、授权和 Admission Control。认证解决用户是谁的问题，授权解决用户能做什么的问题，Admission Control 则是资源管理方面的作用。通过合理的权限管理，能够保证系统的安全可靠。

Kubernetes 集群的所有操作基本上都是通过 kube-apiserver 这个组件进行的，它提供 HTTP RESTful 形式的 API 供集群内外客户端调用。需要注意的是：认证授权过程只存在 HTTPS 形式的 API 中。也就是说，如果客户端使用 HTTP 连接到 kube-apiserver，那么是不会进行认证授权的。所以说，可以这么设置，在集群内部组件间通信使用 HTTP，集群外部就使用 HTTPS，这样既增加了安全性，也不至于太复杂。

下图是 API 访问要经过的三个步骤，前面两个是认证和授权，第三个是 Admission Control。

认证

开启 TLS 时，所有的请求都需要首先认证。Kubernetes 支持多种认证机制，并支持同时开启多个认证插件（只要有一个认证通过即可）。如果认证成功，则用户的 `username` 会传入授权模块做进一步授权验证；而对于认证失败的请求则返回 HTTP 401。

> **Kubernetes 不直接管理用户**
>
> 虽然 Kubernetes 认证和授权用到了 username，但 Kubernetes 并不直接管理用户，不能创建 `user` 对象， 也不存储 username。但是 Kubernetes 提供了 Service Account，用来与 API 交互。

目前，Kubernetes 支持以下认证插件：

- X509 证书
- 静态 Token 文件
- 引导 Token
- 静态密码文件
- Service Account
- OpenID
- Webhook
- 认证代理
- OpenStack Keystone 密码

详细使用方法请参考这里

授权

授权主要是用于对集群资源的访问控制，通过检查请求包含的相关属性值，与相对应的访问策略相比较，API 请求必须满足某些策略才能被处理。跟认证类似，Kubernetes 也支持多种授权机制，并支持同时开启多个授权插件（只要有一个验证通过即可）。如果授权成功，则用户的请求会发送到准入控制模块做进一步的请求验证；对于授权失败的请求则返回 HTTP 403。

Kubernetes 授权仅处理以下的请求属性：

- user，group，extra
- API、请求方法（如 get、post、update、patch 和 delete）和请求路径（如 `/api`）
- 请求资源和子资源
- Namespace
- API Group

目前，Kubernetes 支持以下授权插件：

- ABAC
- RBAC
- Webhook
- Node

> **AlwaysDeny 和 AlwaysAllow**
>
> Kubernetes 还支持 AlwaysDeny 和 AlwaysAllow 模式，其中 AlwaysDeny 仅用来测试，而 AlwaysAllow 则 允许所有请求（会覆盖其他模式）。

ABAC 授权

使用 ABAC 授权需要 API Server 配置 `--authorization-policy-file=SOME_FILENAME`，文件格式为每行一个 json 对象，比如

```
{
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy",
    "spec": {
        "group": "system:authenticated",
        "nonResourcePath": "*",
        "readonly": true
    }
}
{
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy",
    "spec": {
        "group": "system:unauthenticated",
        "nonResourcePath": "*",
        "readonly": true
    }
}
{
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy",
    "spec": {
        "user": "admin",
        "namespace": "*",
        "resource": "*",
        "apiGroup": "*"
    }
}
```

RBAC 授权

见 RBAC 授权。

WebHook 授权

使用 WebHook 授权需要 API Server 配置 `--authorization-webhook-config-file=SOME_FILENAME` 和 `--runtime-config=authorization.k8s.io/v1beta1=true`，配置文件格式同 kubeconfig，如

```
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authz-service
    cluster:
      # CA for verifying the remote service.
      certificate-authority: /path/to/ca.pem
      # URL of remote service to query. Must use 'https'.
      server: https://authz.example.com/authorize

# users refers to the API Server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      # cert for the webhook plugin to use
      client-certificate: /path/to/cert.pem
       # key matching the cert
      client-key: /path/to/key.pem

# kubeconfig files require a context. Provide one for the API Ser
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authz-service
    user: name-of-api-server
  name: webhook
```

API Server 请求 Webhook server 的格式为

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "kittensandponies",
      "verb": "get",
      "group": "unicorn.example.org",
      "resource": "pods"
    },
    "user": "jane",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```

而 Webhook server 需要返回授权的结果，允许 (allowed=true) 或拒绝(allowed=false)：

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}
```

Node 授权

v1.7 + 支持 Node 授权，配合 `NodeRestriction` 准入控制来限制 kubelet 仅可访问 node、endpoint、pod、service 以及 secret、configmap、PV 和 PVC 等相关的资源，配置方法为

`--authorization-mode=Node,RBAC --admission-control=...,NodeRestriction,...`

注意，kubelet 认证需要使用 `system:nodes` 组，并使用用户名 `system:node:` 。

参考文档

- Authenticating
- Authorization
- Bootstrap Tokens
- Managing Service Accounts
- ABAC Mode
- Webhook Mode
- Node Authorization

认证

开启 TLS 时，所有的请求都需要首先认证。Kubernetes 支持多种认证机制，并支持同时开启多个认证插件（只要有一个认证通过即可）。如果认证成功，则用户的 `username` 会传入授权模块做进一步授权验证；而对于认证失败的请求则返回 HTTP 401。

> **Kubernetes 不直接管理用户**
>
> 虽然 Kubernetes 认证和授权用到了 username，但 Kubernetes 并不直接管理用户，不能创建 `user` 对象， 也不存储 username。但是 Kubernetes 提供了 Service Account，用来与 API 交互。

目前，Kubernetes 支持以下认证插件：

- X509 证书
- 静态 Token 文件
- 引导 Token
- 静态密码文件
- Service Account
- OpenID
- Webhook
- 认证代理
- OpenStack Keystone 密码

X509 证书

使用 X509 客户端证书只需要 API Server 启动时配置 `--client-ca-file=SOMEFILE` 。在证书认证时，其 CN 域用作用户名，而组织机构域则用作 group 名。

创建一个客户端证书的方法为：

```
openssl req -new -key jbeda.pem -out jbeda-csr.pem -subj "/CN=jbe
```

静态 Token 文件

使用静态 Token 文件认证只需要 API Server 启动时配置 `--token-auth-file=SOMEFILE` 。该文件为 csv 格式，每行至少包括三列 `token,username,user id` ，后面是可选的 group 名，比如

```
token,user,uid,"group1,group2,group3"
```

客户端在使用 token 认证时，需要在请求头中加入 Bearer Authorization 头，比如

```
Authorization: Bearer 31ada4fd-adec-460c-809a-9e56ceb75269
```

引导 Token

引导 Token 是动态生成的，存储在 kube-system namespace 的 Secret 中，用来部署新的 Kubernetes 集群。

使用引导 Token 需要 API Server 启动时配置 `--experimental-bootstrap-token-auth` ，并且 Controller Manager 开启 TokenCleaner `--controllers=*,tokencleaner,bootstrapsigner` 。

在使用 kubeadm 部署 Kubernetes 时，kubeadm 会自动创建默认 token，可通过 `kubeadm token list` 命令查询。

静态密码文件

需要 API Server 启动时配置 `--basic-auth-file=SOMEFILE`，文件格式为 csv，每行至少三列 `password, user, uid`，后面是可选的 group 名，如

```
password,user,uid,"group1,group2,group3"
```

客户端在使用密码认证时，需要在请求头重加入 Basic Authorization 头，如

```
Authorization: Basic BASE64ENCODED(USER:PASSWORD)
```

Service Account

ServiceAccount 是 Kubernetes 自动生成的，并会自动挂载到容器的 `/var/run/secrets/kubernetes.io/serviceaccount` 目录中。

在认证时，ServiceAccount 的用户名格式为 `system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)`，并从属于两个 group：`system:serviceaccounts` 和 `system:serviceaccounts:(NAMESPACE)`。

OpenID

OpenID 提供了 OAuth2 的认证机制，是很多云服务商（如 GCE、Azure 等）的首选认证方法。

使用 OpenID 认证，API Server 需要配置

- `--oidc-issuer-url`，如 `https://accounts.google.com`
- `--oidc-client-id`，如 `kubernetes`
- `--oidc-username-claim`，如 `sub`
- `--oidc-groups-claim`，如 `groups`
- `--oidc-ca-file`，如 `/etc/kubernetes/ssl/kc-ca.pem`

Webhook

API Server 需要配置

```
# 配置如何访问 webhook server
--authentication-token-webhook-config-file
# 默认 2 分钟
--authentication-token-webhook-cache-ttl
```

配置文件格式为

```
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authn-service
    cluster:
      # CA for verifying the remote service.
      certificate-authority: /path/to/ca.pem
      # URL of remote service to query. Must use 'https'.
      server: https://authn.example.com/authenticate

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      # cert for the webhook plugin to use
      client-certificate: /path/to/cert.pem
       # key matching the cert
      client-key: /path/to/key.pem

# kubeconfig files require a context. Provide one for the API ser
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authn-service
    user: name-of-api-sever
  name: webhook
```

Kubernetes 发给 webhook server 的请求格式为

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "(BEARERTOKEN)"
  }
}
```

示例：kubernetes-github-authn 实现了一个基于 WebHook 的 github 认证。

认证代理

API Server 需要配置

```
--requestheader-username-headers=X-Remote-User
--requestheader-group-headers=X-Remote-Group
--requestheader-extra-headers-prefix=X-Remote-Extra-
# 为了防止头部欺骗，证书是必选项
--requestheader-client-ca-file
# 设置允许的 CN 列表。可选。
--requestheader-allowed-names
```

OpenStack Keystone 密码

需要 API Server 在启动时指定 `--experimental-keystone-url=`，而 https 时还需要设置 `--experimental-keystone-ca-file=SOMEFILE`。

> **不支持 Keystone v3**
>
> 目前只支持 keystone v2.0，不支持 v3（无法传入 domain）。

匿名请求

如果使用 AlwaysAllow 以外的认证模式，则匿名请求默认开启，但可用 `--anonymous-auth=false` 禁止匿名请求。

匿名请求的用户名格式为 `system:anonymous`，而 group 则为 `system:unauthenticated`。

Credential Plugin

从 v1.11 开始支持 Credential Plugin（Beta），通过调用外部插件来获取用户的访问凭证。这是一种客户端认证插件，用来支持不在 Kubernetes 中内置的认证协议，如 LDAP、OAuth2、SAML 等。它通常与 Webhook 配合使用。

Credential Plugin 可以在 kubectl 的配置文件中设置，比如

```
apiVersion: v1
kind: Config
users:
- name: my-user
  user:
    exec:
      # Command to execute. Required.
      command: "example-client-go-exec-plugin"

      # API version to use when decoding the ExecCredentials resc
      #
      # The API version returned by the plugin MUST match the ver
      #
      # To integrate with tools that support multiple versions (s
      # set an environment variable or pass an argument to the to
      apiVersion: "client.authentication.k8s.io/v1beta1"

      # Environment variables to set when executing the plugin. C
      env:
      - name: "FOO"
        value: "bar"

      # Arguments to pass when executing the plugin. Optional.
      args:
      - "arg1"
      - "arg2"
clusters:
- name: my-cluster
  cluster:
    server: "https://172.17.4.100:6443"
    certificate-authority: "/etc/kubernetes/ca.pem"
contexts:
- name: my-cluster
  context:
    cluster: my-cluster
    user: my-user
current-context: my-cluster
```

具体的插件开发及使用方法请参考 kubernetes/client-go。

RBAC授权

Kubernetes 从 1.6 开始支持基于角色的访问控制机制（Role-Based Access，RBAC），集群管理员可以对用户或服务账号的角色进行更精确的资源访问控制。在 RBAC 中，权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。这就极大地简化了权限的管理。在一个组织中，角色是为了完成各种工作而创造，用户则依据它的责任和资格来被指派相应的角色，用户可以很容易地从一个角色被指派到另一个角色。

前言

Kubernetes 1.6 中的一个亮点时 RBAC 访问控制机制升级到了 beta 版本（版本为 `rbac.authorization.k8s.io/v1beta1` ）。RBAC，基于角色的访问控制机制，是用来管理 kubernetes 集群中资源访问权限的机制。使用 RBAC 可以很方便的更新访问授权策略而不用重启集群。

从 Kubernetes 1.8 开始，RBAC 进入稳定版，其 API 为 `rbac.authorization.k8s.io/v1` 。

在使用 RBAC 时，只需要在启动 kube-apiserver 时配置 `--authorization-mode=RBAC` 即可。

RBAC vs ABAC

目前 kubernetes 中已经有一系列 l 鉴权机制。鉴权的作用是，决定一个用户是否有权使用 Kubernetes API 做某些事情。它除了会影响 kubectl 等组件之外，还会对一些运行在集群内部并对集群进行操作的软件产生作用，例如使用了 Kubernetes 插件的 Jenkins，或者是利用 Kubernetes API 进行软件部署的 Helm。ABAC 和 RBAC 都能够对访问策略进行配置。

ABAC（Attribute Based Access Control）本来是不错的概念，但是在 Kubernetes 中的实现比较难于管理和理解，而且需要对 Master 所在节点的 SSH 和文件系统权限，而且要使得对授权的变更成功生效，还需要重新启动 API Server。

而 RBAC 的授权策略可以利用 kubectl 或者 Kubernetes API 直接进行配置。**RBAC 可以授权给用户，让用户有权进行授权管理，这样就可以无需接触节点，直接进行授权管理。**RBAC 在 Kubernetes 中被映射为 API 资源和操作。

因为 Kubernetes 社区的投入和偏好，相对于 ABAC 而言，RBAC 是更好的选择。

基础概念

需要理解 RBAC 一些基础的概念和思路，RBAC 是让用户能够访问 Kubernetes API 资源 的授权方式。

RBAC 架构图 1

在 RBAC 中定义了两个对象，用于描述在用户和资源之间的连接权限。

Role 与 ClusterRole

Role（角色）是一系列权限的集合，例如一个角色可以包含读取 Pod 的权限和列出 Pod 的权限。Role 只能用来给某个特定 namespace 中的资源作鉴权，对多 namespace 和集群级的资源或者是非资源类的 API（如 `/healthz`）使用 ClusterRole。

```
# Role 示例
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] #"" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```
# ClusterRole 示例
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RoleBinding 和 ClusterRoleBinding

RoleBinding 把角色（Role 或 ClusterRole）的权限映射到用户或者用户组，从而让这些用户继承角色在 namespace 中的权限。
ClusterRoleBinding 让用户继承 ClusterRole 在整个集群中的权限。

注意 ServiceAccount 的用户名格式为 `system:serviceaccount:`，并且都属于 `system:serviceaccounts:` 用户组。

```
# RoleBinding 示例（引用 Role）
# This role binding allows "jane" to read pods in the "default" r
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

RBAC 架构图 2

```
# RoleBinding 示例（引用 ClusterRole）
# This role binding allows "dave" to read secrets in the "develop
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-secrets
  namespace: development # This only grants permissions within th
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

ClusterRole 聚合

从 v1.9 开始，在 ClusterRole 中可以通过 `aggregationRule` 来与其他 ClusterRole 聚合使用（该特性在 v1.11 GA）。

比如

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # Rules are automatically filled in by the controller m
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# These rules will be added to the "monitoring" role.
rules:
- apiGroups: [""]
  resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

默认 ClusterRole

RBAC 现在被 Kubernetes 深度集成，并使用它给系统组件进行授权。System Roles 一般具有前缀 `system:`，很容易识别：

```
$ kubectl get clusterroles --namespace=kube-system
NAME                                             AGE
admin                                            10d
cluster-admin                                    10d
edit                                             10d
system:auth-delegator                            10d
system:basic-user                                10d
system:controller:attachdetach-controller        10d
system:controller:certificate-controller         10d
system:controller:cronjob-controller             10d
system:controller:daemon-set-controller          10d
system:controller:deployment-controller          10d
system:controller:disruption-controller          10d
system:controller:endpoint-controller            10d
system:controller:generic-garbage-collector      10d
system:controller:horizontal-pod-autoscaler      10d
system:controller:job-controller                 10d
system:controller:namespace-controller           10d
system:controller:node-controller                10d
system:controller:persistent-volume-binder       10d
system:controller:pod-garbage-collector          10d
system:controller:replicaset-controller          10d
system:controller:replication-controller         10d
system:controller:resourcequota-controller       10d
system:controller:route-controller               10d
system:controller:service-account-controller     10d
system:controller:service-controller             10d
system:controller:statefulset-controller         10d
system:controller:ttl-controller                 10d
system:discovery                                 10d
system:heapster                                  10d
system:kube-aggregator                           10d
system:kube-controller-manager                   10d
system:kube-dns                                  10d
system:kube-scheduler                            10d
system:node                                      10d
system:node-bootstrapper                         10d
system:node-problem-detector                     10d
system:node-proxier                              10d
```

```
system:persistent-volume-provisioner            10d
view                                            10d
```

其他的内置角色可以参考 default-roles-and-role-bindings。

RBAC 系统角色已经完成足够的覆盖，让集群可以完全在 RBAC 的管理下运行。

从 ABAC 迁移到 RBAC

在 ABAC 到 RBAC 进行迁移的过程中，有些在 ABAC 集群中缺省开放的权限，在 RBAC 中会被视为不必要的授权，会对其进行 降级。这种情况会影响到使用 Service Account 的应用。ABAC 配置中，从 Pod 中发出的请求会使用 Pod Token，API Server 会为其授予较高权限。例如下面的命令在 ABAC 集群中会返回 JSON 结果，而在 RBAC 的情况下则会返回错误。

```
$ kubectl run nginx --image=nginx:latest
$ kubectl exec -it $(kubectl get pods -o jsonpath='{.items[0].met
$ apt-get update && apt-get install -y curl
$ curl -ik \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/
  https://kubernetes/api/v1/namespaces/default/pods
```

所有在 Kubernetes 集群中运行的应用，一旦和 API Server 进行通信，都会有可能受到迁移的影响。

要平滑的从 ABAC 升级到 RBAC，在创建 1.6 集群的时候，可以同时启用 ABAC 和 RBAC。当他们同时启用的时候，对一个资源的权限请求，在任何一方获得放行都会获得批准。然而在这种配置下的权限太过粗放，很可能无法在单纯的 RBAC 环境下工作。

目前 RBAC 已经进入稳定版，ABAC 可能会被弃用。在可见的未来 ABAC 依然会保留在 kubernetes 中，不过开发的重心已经转移到了 RBAC。

Permissive RBAC

所谓 Permissive RBAC 是指授权给所有的 Service Accounts 管理员权限。注意，这是一个不推荐的配置。

```
kubectl create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \
  --user=admin \
  --user=kubelet \
  --group=system:serviceaccounts
```

参考文档

- RBAC documentation
- Google Cloud Next talks 1
- Google Cloud Next talks 2
- 在 Kubernetes Pod 中使用 Service Account 访问 API Server
- 部分翻译自 RBAC Support in Kubernetes（转载自kubernetes中文社区，译者催总，Jimmy Song 做了稍许修改）

准入控制

准入控制（Admission Control）在授权后对请求做进一步的验证或添加默认参数。不同于授权和认证只关心请求的用户和操作，准入控制还处理请求的内容，并且仅对创建、更新、删除或连接（如代理）等有效，而对读操作无效。

准入控制支持同时开启多个插件，它们依次调用，只有全部插件都通过的请求才可以放过进入系统。

Kubernetes 目前提供了以下几种准入控制插件

- AlwaysAdmit：接受所有请求。
- AlwaysPullImages：总是拉取最新镜像。在多租户场景下非常有用。
- DenyEscalatingExec：禁止特权容器的 exec 和 attach 操作。
- ImagePolicyWebhook：通过 webhook 决定 image 策略，需要同时配置 `--admission-control-config-file`，配置文件格式见 这里。
- ServiceAccount：自动创建默认 ServiceAccount，并确保 Pod 引用的 ServiceAccount 已经存在
- SecurityContextDeny：拒绝包含非法 SecurityContext 配置的容器
- ResourceQuota：限制 Pod 的请求不会超过配额，需要在 namespace 中创建一个 ResourceQuota 对象

- LimitRanger：为 Pod 设置默认资源请求和限制，需要在 namespace 中创建一个 LimitRange 对象
- InitialResources：根据镜像的历史使用记录，为容器设置默认资源请求和限制
- NamespaceLifecycle：确保处于 termination 状态的 namespace 不再接收新的对象创建请求，并拒绝请求不存在的 namespace
- DefaultStorageClass：为 PVC 设置默认 StorageClass（见 这里）
- DefaultTolerationSeconds：设置 Pod 的默认 forgiveness toleration 为 5 分钟
- PodSecurityPolicy：使用 Pod Security Policies 时必须开启
- NodeRestriction：限制 kubelet 仅可访问 node、endpoint、pod、service 以及 secret、configmap、PV 和 PVC 等相关的资源（仅适用于 v1.7+）
- EventRateLimit：限制事件请求数量（仅适用于 v1.9）
- ExtendedResourceToleration：为使用扩展资源（如 GPU 和 FPGA 等）的 Pod 自动添加 tolerations
- StorageProtection：自动给新创建的 PVC 增加 `kubernetes.io/pvc-protection` finalizer（v1.9 及以前版本为 `PVCProtection`，v.11 GA）
- PersistentVolumeClaimResize：允许设置 `allowVolumeExpansion=true` 的 StorageClass 调整 PVC 大小（v1.11 Beta）
- PodNodeSelector：限制一个 Namespace 中可以使用的 Node 选择标签
- ValidatingAdmissionWebhook：使用 Webhook 验证请求，这些 Webhook 并行调用，并且任何一个调用拒绝都会导致请求失败
- MutatingAdmissionWebhook：使用 Webhook 修改请求，这些 Webhook 依次顺序调用

Kubernetes v1.7 + 还支持 Initializers 和 GenericAdmissionWebhook，可以用来方便地扩展准入控制。

Initializers

Initializers 可以用来给资源执行策略或者配置默认选项，包括 Initializers 控制器和用户定义的 Initializer 任务，控制器负责执行用户提交的任务，并完成后将任务从 `metadata.initializers` 列表中删除。

Initializers 的开启方法为

- kube-apiserver 配置 `--admission-control=...,Initializers`
- kube-apiserver 开启 `admissionregistration.k8s.io/v1alpha1` API，即配置 `--runtime-config=admissionregistration.k8s.io/v1alpha1`
- 部署 Initializers 控制器

另外，可以使用 `initializerconfigurations` 来自定义哪些资源开启 Initializer 功能

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: InitializerConfiguration
metadata:
  name: example-config
initializers:
  # the name needs to be fully qualified, i.e., containing at lea
  - name: podimage.example.com
    rules:
      # apiGroups, apiVersion, resources all support wildcard "*"
      # "*" cannot be mixed with non-wildcard.
      - apiGroups:
          - ""
        apiVersions:
          - v1
        resources:
          - pods
```

Initializers 可以用来

- 修改资源的配置，比如自动给 Pod 添加一个 sidecar 容器或者存储卷
- 如果不需要修改对象的话，建议使用性能更好的 GenericAdmissionWebhook。

如何开发 Initializers

- 参考 Kubernetes Initializer Tutorial 开发 Initializer
- Initializer 必须有一个全局唯一的名字，比如 `initializer.vaultproject.io`
- Initializer 有可能收到信息不全的资源（比如还未调度的 Pod 没有 nodeName 和 status），在实现时需要考虑这种情况
- 对于 Initializer 自身的部署，可以使用 Deployment，但需要手动设置 initializers 列表为空，以避免无法启动的问题，如

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  initializers:
    pending: []
```

GenericAdmissionWebhook

GenericAdmissionWebhook 提供了一种 Webhook 方式的准入控制机制，它不会改变请求对象，但可以用来验证用户的请求。

GenericAdmissionWebhook 的开启方法

- kube-apiserver 配置 `--admission-control=...,GenericAdmissionWebhook`
- kube-apiserver 开启 `admissionregistration.k8s.io/v1alpha1` API，即配置 `--runtime-config=admissionregistration.k8s.io/v1alpha1`
- 实现并部署 webhook 准入控制器，参考 这里 的示例

注意，webhook 准入控制器必须使用 TLS，并需要通过 `externaladmissionhookconfigurations.clientConfig.caBundle` 向 kube-apiserver 注册：

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: ExternalAdmissionHookConfiguration
metadata:
  name: example-config
externalAdmissionHooks:
- name: pod-image.k8s.io
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
  # fail upon a communication error with the webhook admission cc
  # Other options: Ignore
  failurePolicy: Fail
  clientConfig:
    caBundle: >
    service:
      name: >
      namespace: >
```

推荐配置

对于 Kubernetes >= 1.9.0, 推荐配置以下插件

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount
```

对于 Kubernetes >= 1.6.0, 推荐 kube-apiserver 开启以下插件

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount
```

对于 Kubernetes >= 1.4.0, 推荐配置以下插件

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount
```

参考文档

- Using Admission Controllers
- How Kubernetes Initializers work

Scheduler扩展

如果默认的调度器不满足要求，还可以部署自定义的调度器。并且，在整个集群中还可以同时运行多个调度器实例，通过 `podSpec.schedulerName` 来选择使用哪一个调度器（默认使用内置的调度器）。

开发自定义调度器

自定义调度器主要的功能是查询未调度的 Pod，按照自定义的调度策略选择新的 Node，并将其更新到 Pod 的 Node Binding 上。

比如，一个最简单的调度器可以用 shell 来编写（假设 Kubernetes 监听在 `localhost:8001`）：

```
#!/bin/bash
SERVER='localhost:8001'
while true;
do
    for PODNAME in $(kubectl --server $SERVER get pods -o json |
;
    do
        NODES=($(kubectl --server $SERVER get nodes -o json | jq
        NUMNODES=${#NODES[@]}
        CHOSEN=${NODES[$[ $RANDOM % $NUMNODES]]}
        curl --header "Content-Type:application/json" --request P
: "Node", "name": "'$CHOSEN'"}}' http://$SERVER/api/v1/namespaces
        echo "Assigned $PODNAME to $CHOSEN"
    done
    sleep 1
done
```

使用自定义调度器

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  # 选择使用自定义调度器 my-scheduler
  schedulerName: my-scheduler
  containers:
  - name: nginx
    image: nginx:1.10
```

网络插件

网络模型

- IP-per-Pod，每个 Pod 都拥有一个独立 IP 地址，Pod 内所有容器共享一个网络命名空间
- 集群内所有 Pod 都在一个直接连通的扁平网络中，可通过 IP 直接访问
  - 所有容器之间无需 NAT 就可以直接互相访问
  - 所有 Node 和所有容器之间无需 NAT 就可以直接互相访问
  - 容器自己看到的 IP 跟其他容器看到的一样
- Service cluster IP 尽可在集群内部访问，外部请求需要通过 NodePort、LoadBalance 或者 Ingress 来访问

官方插件

目前，Kubernetes 支持以下两种插件：

- kubenet：这是一个基于 CNI bridge 的网络插件（在 bridge 插件的基础上扩展了 port mapping 和 traffic shaping ），是目前推荐的默认插件

- CNI：CNI 网络插件，需要用户将网络配置放到 `/etc/cni/net.d` 目录中，并将 CNI 插件的二进制文件放入 `/opt/cni/bin`
- ~~exec：通过第三方的可执行文件来为容器配置网络，已在 v1.6 中移除，见~~ kubernetes#39254

kubenet

kubenet 是一个基于 CNI bridge 的网络插件，它为每个容器建立一对 veth pair 并连接到 cbr0 网桥上。kubenet 在 bridge 插件的基础上拓展了很多功能，包括

- 使用 host-local IPAM 插件为容器分配 IP 地址， 并定期释放已分配但未使用的 IP 地址
- 设置 sysctl `net.bridge.bridge-nf-call-iptables = 1`
- 为 Pod IP 创建 SNAT 规则
  - `-A POSTROUTING ! -d 10.0.0.0/8 -m comment --comment "kubenet: SNAT for outbound traffic from cluster" -m addrtype ! --dst-type LOCAL -j MASQUERADE`
- 开启网桥的 hairpin 和 promisc 模式，允许 Pod 访问它自己所在的 Service IP（即通过 NAT 后再访问 Pod 自己）

```
-A OUTPUT -j KUBE-DEDUP
-A KUBE-DEDUP -p IPv4 -s a:58:a:f4:2:1 -o veth+ --ip-src 10.2
-A KUBE-DEDUP -p IPv4 -s a:58:a:f4:2:1 -o veth+ --ip-src 10.2
```

- HostPort 管理以及设置端口映射
- Traffic shaping，支持通过 `kubernetes.io/ingress-bandwidth` 和 `kubernetes.io/egress-bandwidth` 等 Annotation 设置 Pod 网络带宽限制

未来 kubenet 插件会迁移到标准的 CNI 插件（如 ptp），具体计划见这里。

CNI plugin

安装 CNI：

```
cat < /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.
EOF

yum install -y kubernetes-cni
```

配置 CNI brige 插件：

```
    mkdir -p /etc/cni/net.d
cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.0.0/16",
        "routes": [
            {"dst": "0.0.0.0/0"}
        ]
    }
}
EOF
cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "type": "loopback"
}
EOF
```

更多 CNI 网络插件的说明请参考 CNI 网络插件。

Flannel

Flannel 是一个为 Kubernetes 提供 overlay network 的网络插件，它基于 Linux TUN/TAP，使用 UDP 封装 IP 包来创建 overlay 网络，并借助 etcd 维护网络的分配情况。

```
kubectl create -f https://github.com/coreos/flannel/raw/master/Do
kubectl create -f https://github.com/coreos/flannel/raw/master/Do
```

Weave Net

Weave Net 是一个多主机容器网络方案，支持去中心化的控制平面，各个 host 上的 wRouter 间通过建立 Full Mesh 的 TCP 链接，并通过 Gossip 来同步控制信息。这种方式省去了集中式的 K/V Store，能够在一定程度上减低部署的复杂性，Weave 将其称为 “data centric”，而非 RAFT 或者 Paxos 的 “algorithm centric”。

数据平面上，Weave 通过 UDP 封装实现 L2 Overlay，封装支持两种模式，一种是运行在 user space 的 sleeve mode，另一种是运行在 kernal space 的 fastpath mode。Sleeve mode 通过 pcap 设备在 Linux bridge 上截获数据包并由 wRouter 完成 UDP 封装，支持对 L2 traffic 进行加密，还支持 Partial Connection，但是性能损失明显。Fastpath mode 即通过 OVS 的 odp 封装 VxLAN 并完成转发，wRouter 不直接参与转发，而是通过下发 odp 流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

```
kubectl apply -f https://git.io/weave-kube
```

Calico

Calico 是一个基于 BGP 的纯三层的数据中心网络方案（不需要 Overlay），并且与 OpenStack、Kubernetes、AWS、GCE 等 IaaS 和容器平台都有良好的集成。

Calico 在每一个计算节点利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发，而每个 vRouter 通过 BGP 协议负责把自己上运行的 workload 的路由信息像整个 Calico 网络内传播——小规模部署可以直接互联，大规模下可通过指定的 BGP route reflector 来完成。 这样保证最终所有的 workload 之间的数据流量都是通过 IP 路由的方式完成互联的。Calico 节点组网可以直接利用数据中心的网络结构（无论是 L2 或者 L3），不需要额外的 NAT，隧道或者 Overlay Network。

此外，Calico 基于 iptables 还提供了丰富而灵活的网络 Policy，保证通过各个节点上的 ACLs 来提供 Workload 的多租户隔离、安全组以及其他可达性限制等功能。

```
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-start
```

OVS

https://kubernetes.io/docs/admin/ovs-networking/ 提供了一种简单的基于 OVS 的网络配置方法：

- 每台机器创建一个 Linux 网桥 kbr0，并配置 docker 使用该网桥（而不是默认的 docker0），其子网为 10.244.x.0/24
- 每台机器创建一个 OVS 网桥 obr0，通过 veth pair 连接 kbr0 并通过 GRE 将所有机器互联
- 开启 STP
- 路由 10.244.0.0/16 到 OVS 隧道

OVN

OVN (Open Virtual Network) 是 OVS 提供的原生虚拟化网络方案，旨在解决传统 SDN 架构（比如 Neutron DVR）的性能问题。

OVN 为 Kubernetes 提供了两种网络方案：

- Overaly：通过 ovs overlay 连接容器
- Underlay：将 VM 内的容器连到 VM 所在的相同网络（开发中）

其中，容器网络的配置是通过 OVN 的 CNI 插件来实现。

Contiv

Contiv 是思科开源的容器网络方案，主要提供基于 Policy 的网络管理，并与主流容器编排系统集成。Contiv 最主要的优势是直接提供了多租户网络，并支持 L2(VLAN)，L3(BGP)，Overlay (VXLAN) 以及思科自家的 ACI。

Romana

Romana 是 Panic Networks 在 2016 年提出的开源项目，旨在借鉴 route aggregation 的思路来解决 Overlay 方案给网络带来的开销。

OpenContrail

OpenContrail 是 Juniper 推出的开源网络虚拟化平台，其商业版本为 Contrail。其主要由控制器和 vRouter 组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter 提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

其中，vRouter 支持三种模式

- Kernel vRouter：类似于 ovs 内核模块
- DPDK vRouter：类似于 ovs-dpdk
- Netronome Agilio Solution (商业产品)：支持 DPDK，SR-IOV and Express Virtio (XVIO)

Juniper/contrail-kubernetes 提供了 Kubernetes 的集成，包括两部分：

- kubelet network plugin 基于 kubernetes v1.6 已经删除的 exec network plugin
- kube-network-manager 监听 kubernetes API，并根据 label 信息来配置网络策略

Midonet

Midonet 是 Midokura 公司开源的 OpenStack 网络虚拟化方案。

- 从组件来看，Midonet 以 Zookeeper+Cassandra 构建分布式数据库存储 VPC 资源的状态——Network State DB Cluster，并将 controller 分布在转发设备（包括 vswitch 和 L3 Gateway）本地——Midolman（L3 Gateway 上还有 quagga bgpd），设备的转发则保留了 ovs kernel 作为 fast datapath。可以看到，Midonet 和 DragonFlow、OVN 一样，在架构的设计上都是沿着 OVS-Neutron-Agent 的思路，将 controller 分布到设备本地，并在 neutron plugin 和设备 agent 间嵌入自己的资源数据库作为 super controller。
- 从接口来看，NSDB 与 Neutron 间是 REST API，Midolman 与 NSDB 间是 RPC，这俩没什么好说的。Controller 的南向方面，Midolman 并没有用 OpenFlow 和 OVSDB，它干掉了 user space 中的 vswitchd 和 ovsdb-server，直接通过 linux netlink 机制操作 kernel space 中的 ovs datapath。

Host network

最简单的网络模型就是让容器共享 Host 的 network namespace，使用宿主机的网络协议栈。这样，不需要额外的配置，容器就可以共享宿主的各种网络资源。

优点

- 简单，不需要任何额外配置
- 高效，没有 NAT 等额外的开销

缺点

- 没有任何的网络隔离
- 容器和 Host 的端口号容易冲突
- 容器内任何网络配置都会影响整个宿主机

> 注意：HostNetwork 是在 Pod 配置文件中设置的，kubelet 在启动时还是需要配置使用 CNI 或者 kubenet 插件（默认 kubenet）。

其他

ipvs

Kubernetes v1.8 已经支持 ipvs 负载均衡模式（alpha 版）。

Canal

Canal 是 Flannel 和 Calico 联合发布的一个统一网络插件，提供 CNI 网络插件，并支持 network policy。

kuryr-kubernetes

kuryr-kubernetes 是 OpenStack 推出的集成 Neutron 网络插件，主要包括 Controller 和 CNI 插件两部分，并且也提供基于 Neutron LBaaS 的 Service 集成。

Cilium

Cilium 是一个基于 eBPF 和 XDP 的高性能容器网络方案，提供了 CNI 和 CNM 插件。

项目主页为 https://github.com/cilium/cilium。

kope

kope 是一个旨在简化 Kubernetes 网络配置的项目，支持三种模式：

- Layer2：自动为每个 Node 配置路由
- Vxlan：为主机配置 vxlan 连接，并建立主机和 Pod 的连接（通过 vxlan interface 和 ARP entry）
- ipsec：加密链接

项目主页为 https://github.com/kopeio/kope-routing。

Kube-router

Kube-router 是一个基于 BGP 的网络插件，并提供了可选的 ipvs 服务发现（替代 kube-proxy）以及网络策略功能。

部署 Kube-router：

```
kubectl apply -f https://raw.githubusercontent.com/cloudnativelab
```

部署 Kube-router 并替换 kube-proxy（这个功能其实不需要了，kube-proxy 已经内置了 ipvs 模式的支持）：

```
kubectl apply -f https://raw.githubusercontent.com/cloudnativelab
# Remove kube-proxy
kubectl -n kube-system delete ds kube-proxy
docker run --privileged --net=host gcr.io/google_containers/kube-
```

CNI

Container Network Interface（CNI）最早是由CoreOS发起的容器网络规范，是Kubernetes网络插件的基础。其基本思想为：Container Runtime在创建容器时，先创建好network namespace，然后调用CNI插件为这个netns配置网络，其后再启动容器内的进程。现已加入CNCF，成为CNCF主推的网络模型。

CNI插件包括两部分：

- CNI Plugin负责给容器配置网络，它包括两个基本的接口
  - 配置网络：AddNetwork(net *NetworkConfig*, *rt* RuntimeConf) (types.Result, error)
  - 清理网络：DelNetwork(net *NetworkConfig*, *rt* RuntimeConf) error
- IPAM Plugin负责给容器分配IP地址，主要实现包括host-local和dhcp。

Kubernetes Pod 中的其他容器都是Pod所属pause容器的网络，创建过程为：

1. kubelet 先创建pause容器生成network namespace
2. 调用网络CNI driver
3. CNI driver 根据配置调用具体的cni 插件

4. cni 插件给pause 容器配置网络
5. pod 中其他的容器都使用 pause 容器的网络

所有CNI插件均支持通过环境变量和标准输入传入参数：

```
$ echo '{"cniVersion": "0.3.1","name": "mynet","type": "macvlan",

$ echo '{"cniVersion": "0.3.1","type":"IGNORED", "name": "a","ipa
```

常见的CNI网络插件有

CNI Plugin Chains

CNI还支持Plugin Chains，即指定一个插件列表，由Runtime依次执行每个插件。这对支持端口映射（portmapping）、虚拟机等非常有帮助。配置方法可以参考后面的端口映射示例。

Bridge

Bridge是最简单的CNI网络插件，它首先在Host创建一个网桥，然后再通过veth pair连接该网桥到container netns。

注意：**Bridge模式下，多主机网络通信需要额外配置主机路由，或使用overlay网络。**可以借助Flannel或者Quagga动态路由等来自动配置。比如overlay情况下的网络结构为

配置示例

```
{
    "cniVersion": "0.3.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "mynet0",
    "isDefaultGateway": true,
    "forceAddress": false,
    "ipMasq": true,
    "hairpinMode": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.10.0.0/16"
    }
}
```

```
# export CNI_PATH=/opt/cni/bin
# ip netns add ns
# /opt/cni/bin/cnitool add mynet /var/run/netns/ns
{
    "interfaces": [
        {
            "name": "mynet0",
            "mac": "0a:58:0a:0a:00:01"
        },
        {
            "name": "vethc763e31a",
            "mac": "66:ad:63:b4:c6:de"
        },
        {
            "name": "eth0",
            "mac": "0a:58:0a:0a:00:04",
            "sandbox": "/var/run/netns/ns"
        }
    ],
    "ips": [
        {
            "version": "4",
            "interface": 2,
            "address": "10.10.0.4/16",
            "gateway": "10.10.0.1"
        }
    ],
    "routes": [
        {
            "dst": "0.0.0.0/0",
            "gw": "10.10.0.1"
        }
    ],
    "dns": {}
}
# ip netns exec ns ip addr
1: lo:  mtu 65536 qdisc noop state DOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
9: eth0@if8:  mtu 1500 qdisc noqueue state UP group default
    link/ether 0a:58:0a:0a:00:04 brd ff:ff:ff:ff:ff:ff link-netns
```

```
    inet 10.10.0.4/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::8c78:6dff:fe19:f6bf/64 scope link tentative dadfa
       valid_lft forever preferred_lft forever
# ip netns exec ns ip route
default via 10.10.0.1 dev eth0
10.10.0.0/16 dev eth0  proto kernel  scope link  src 10.10.0.4
```

IPAM

DHCP

DHCP插件是最主要的IPAM插件之一，用来通过DHCP方式给容器分配IP地址，在macvlan插件中也会用到DHCP插件。

在使用DHCP插件之前，需要先启动dhcp daemon：

```
/opt/cni/bin/dhcp daemon &
```

然后配置网络使用dhcp作为IPAM插件

```
{
    ...
    "ipam": {
        "type": "dhcp",
    }
}
```

host-local

host-local是最常用的CNI IPAM插件，用来给container分配IP地址。

IPv4：

```
{
    "ipam": {
        "type": "host-local",
        "subnet": "10.10.0.0/16",
        "rangeStart": "10.10.1.20",
        "rangeEnd": "10.10.3.50",
        "gateway": "10.10.0.254",
        "routes": [
            { "dst": "0.0.0.0/0" },
            { "dst": "192.168.0.0/16", "gw": "10.10.5.1" }
        ],
        "dataDir": "/var/my-orchestrator/container-ipam-state"
    }
}
```

IPv6:

```
{
  "ipam": {
        "type": "host-local",
        "subnet": "3ffe:ffff:0:01ff::/64",
        "rangeStart": "3ffe:ffff:0:01ff::0010",
        "rangeEnd": "3ffe:ffff:0:01ff::0020",
        "routes": [
            { "dst": "3ffe:ffff:0:01ff::1/64" }
        ],
        "resolvConf": "/etc/resolv.conf"
    }
}
```

ptp

ptp插件通过veth pair给容器和host创建点对点连接：veth pair一端在container netns内，另一端在host上。可以通过配置host端的IP和路由来让ptp连接的容器之前通信。

```
{
    "name": "mynet",
    "type": "ptp",
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.1.0/24"
    },
    "dns": {
        "nameservers": [ "10.1.1.1", "8.8.8.8" ]
    }
}
```

IPVLAN

IPVLAN 和 MACVLAN 类似，都是从一个主机接口虚拟出多个虚拟网络接口。一个重要的区别就是所有的虚拟接口都有相同的 mac 地址，而拥有不同的 ip 地址。因为所有的虚拟接口要共享 mac 地址，所以有些需要注意的地方：

- DHCP 协议分配 ip 的时候一般会用 mac 地址作为机器的标识。这个情况下，客户端动态获取 ip 的时候需要配置唯一的 ClientID 字段，并且 DHCP server 也要正确配置使用该字段作为机器标识，而不是使用 mac 地址

IPVLAN支持两种模式：

- L2 模式：此时跟macvlan bridge 模式工作原理很相似，父接口作为交换机来转发子接口的数据。同一个网络的子接口可以通过父接口来转发数据，而如果想发送到其他网络，报文则会通过父接口的路由转发出去。
- L3 模式：此时ipvlan 有点像路由器的功能，它在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相 ping 通对方，因为 ipvlan 会在中间做报文的转发工作。注意 L3 模式下的虚拟接口 不会接收到多播或者广播的报文（这个模式下，所有的网络都会发送给父接口，所有的 ARP 过程或者其他多播报文都是在底层的父接口完成的）。另外外部网络默认情况下是不知道 ipvlan 虚拟出来的网络的，如果不在外部路由器上配置好对应的路由规则，ipvlan 的网络是不能被外部直接访问的。

创建ipvlan的简单方法为

```
ip link add link   type ipvlan mode { l2 | L3 }
```

cni配置格式为

```
{
    "name": "mynet",
    "type": "ipvlan",
    "master": "eth0",
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.2.0/24"
    }
}
```

需要注意的是

- ipvlan插件下，容器不能跟Host网络通信
- 主机接口（也就是master interface）不能同时作为ipvlan和macvlan的master接口

MACVLAN

MACVLAN可以从一个主机接口虚拟出多个macvtap，且每个macvtap设备都拥有不同的mac地址（对应不同的linux字符设备）。MACVLAN支持四种模式

- bridge模式：数据可以在同一master设备的子设备之间转发
- vepa模式：VEPA 模式是对 802.1Qbg 标准中的 VEPA 机制的软件实现，MACVTAP 设备简单的将数据转发到master设备中，完成数据汇聚功能，通常需要外部交换机支持 Hairpin 模式才能正常工作
- private模式：Private 模式和 VEPA 模式类似，区别是子 MACVTAP 之间相互隔离
- passthrough模式：内核的 MACVLAN 数据处理逻辑被跳过，硬件决定数据如何处理，从而释放了 Host CPU 资源

创建macvlan的简单方法为

```
ip link add link  name macvtap0 type macvtap
```

cni配置格式为

```
{
    "name": "mynet",
    "type": "macvlan",
    "master": "eth0",
    "ipam": {
        "type": "dhcp"
    }
}
```

需要注意的是

- macvlan需要大量 mac 地址，每个虚拟接口都有自己的 mac 地址
- 无法和 802.11(wireless) 网络一起工作
- 主机接口（也就是master interface）不能同时作为ipvlan和macvlan的master接口

Flannel

Flannel通过给每台宿主机分配一个子网的方式为容器提供虚拟网络，它基于Linux TUN/TAP，使用UDP封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。

Weave Net

Weave Net是一个多主机容器网络方案，支持去中心化的控制平面，各个host上的wRouter间通过建立Full Mesh的TCP链接，并通过Gossip来同步控制信息。这种方式省去了集中式的K/V Store，能够在一定程度上减低部署的复杂性，Weave将其称为“data centric”，而非RAFT或者Paxos的“algorithm centric”。

数据平面上，Weave通过UDP封装实现L2 Overlay，封装支持两种模式，一种是运行在user space的sleeve mode，另一种是运行在kernal space的 fastpath mode。Sleeve mode通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。Fastpath mode即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

Contiv

Contiv是思科开源的容器网络方案，主要提供基于Policy的网络管理，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN)，L3(BGP)，Overlay (VXLAN)以及思科自家的ACI。

Calico

Calico 是一个基于BGP的纯三层的数据中心网络方案（不需要Overlay），并且与OpenStack、Kubernetes、AWS、GCE等IaaS和容器平台都有良好的集成。

Calico在每一个计算节点利用Linux Kernel实现了一个高效的vRouter来负责数据转发，而每个vRouter通过BGP协议负责把自己上运行的workload的路由信息像整个Calico网络内传播——小规模部署可以直接互联，大规模下可通过指定的BGP route reflector来完成。 这样保证最终所有的workload之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2或者L3），不需要额外的NAT，隧道或者Overlay Network。

此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

OVN

OVN (Open Virtual Network) 是OVS提供的原生虚拟化网络方案，旨在解决传统SDN架构（比如Neutron DVR）的性能问题。

OVN为Kubernetes提供了两种网络方案：

- Overaly：通过ovs overlay连接容器
- Underlay：将VM内的容器连到VM所在的相同网络（开发中）

其中，容器网络的配置是通过OVN的CNI插件来实现。

SR-IOV

Intel维护了一个SR-IOV的CNI插件，fork自hustcat/sriov-cni，并扩展了DPDK的支持。

项目主页见https://github.com/Intel-Corp/sriov-cni。

Romana

Romana是Panic Networks在2016年提出的开源项目，旨在借鉴 route aggregation的思路来解决Overlay方案给网络带来的开销。

OpenContrail

OpenContrail是Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。其主要由控制器和vRouter组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

其中，vRouter支持三种模式

- Kernel vRouter：类似于ovs内核模块
- DPDK vRouter：类似于ovs-dpdk
- Netronome Agilio Solution（商业产品）：支持DPDK，SR-IOV and Express Virtio（XVIO）

michaelhenkel/opencontrail-cni-plugin提供了一个OpenContrail的CNI插件。

Network Configuration Lists

CNI SPEC 支持指定网络配置列表，包含多个网络插件，由 Runtime 依次执行。注意

- ADD 操作，按顺序依次调用每个插件；而 DEL 操作调用顺序相反
- ADD 操作，除最后一个插件，前面每个插件需要增加 `prevResult` 传递给其后的插件

- 第一个插件必须要包含 `ipam` 插件

端口映射示例

下面的例子展示了 `bridge+portmap` 插件的用法。

首先，配置 `CNI` 网络使用 `bridge+portmap` 插件：

```
# cat /root/mynet.conflist
{
  "name": "mynet",
  "cniVersion": "0.3.0",
  "plugins": [
    {
      "type": "bridge",
      "bridge": "mynet",
      "ipMasq": true,
      "isGateway": true,
      "ipam": {
      "type": "host-local",
      "subnet": "10.244.10.0/24",
      "routes": [
          {"dst": "0.0.0.0/0"}
      ]
      }
    },
    {
       "type": "portmap",
       "capabilities": {"portMappings": true}
    }
  ]
}
```

然后通过 `CAP_ARGS` 设置端口映射参数：

```
# export CAP_ARGS='{
    "portMappings": [
        {
            "hostPort":      9090,
            "containerPort": 80,
            "protocol":      "tcp",
            "hostIP":        "127.0.0.1"
        }
    ]
}'
```

测试添加网络接口：

```
# ip netns add test
# CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cnitool add mynet /va
{
    "interfaces": [
        {
            "name": "mynet",
            "mac": "0a:58:0a:f4:0a:01"
        },
        {
            "name": "veth2cfb1d64",
            "mac": "4a:dc:1f:b7:56:b1"
        },
        {
            "name": "eth0",
            "mac": "0a:58:0a:f4:0a:07",
            "sandbox": "/var/run/netns/test"
        }
    ],
    "ips": [
        {
            "version": "4",
            "interface": 2,
            "address": "10.244.10.7/24",
            "gateway": "10.244.10.1"
        }
    ],
    "routes": [
        {
            "dst": "0.0.0.0/0"
        }
    ],
    "dns": {}
}
```

可以从 `iptables` 规则中看到添加的规则：

```
# iptables-save | grep 10.244.10.7
-A CNI-DN-be1eedf7a76853f303ebd -d 127.0.0.1/32 -p tcp -m tcp --d
-A CNI-SN-be1eedf7a76853f303ebd -s 127.0.0.1/32 -d 10.244.10.7/32
```

最后，清理网络接口：

```
# CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cnitool del mynet /va
```

其他

Canal

Canal是Flannel和Calico联合发布的一个统一网络插件，提供CNI网络插件，并支持network policy。

kuryr-kubernetes

kuryr-kubernetes是OpenStack推出的集成Neutron网络插件，主要包括Controller和CNI插件两部分，并且也提供基于Neutron LBaaS的Service集成。

Cilium

Cilium是一个基于eBPF和XDP的高性能容器网络方案，提供了CNI和CNM插件。

项目主页为https://github.com/cilium/cilium。

CNI-Genie

CNI-Genie是华为PaaS团队推出的同时支持多种网络插件（支持calico，canal，romana，weave等）的CNI插件。

项目主页为https://github.com/Huawei-PaaS/CNI-Genie。

Flannel

Flannel通过给每台宿主机分配一个子网的方式为容器提供虚拟网络，它基于Linux TUN/TAP，使用UDP封装IP包来创建overlay网络，并借助etcd维护网络的分配情况。

Flannel原理

控制平面上host本地的flanneld负责从远端的ETCD集群同步本地和其它host上的subnet信息，并为POD分配IP地址。数据平面flannel通过Backend（比如UDP封装）来实现L3 Overlay，既可以选择一般的TUN设备又可以选择VxLAN设备。

```
{
    "Network": "10.0.0.0/8",
    "SubnetLen": 20,
    "SubnetMin": "10.10.0.0",
    "SubnetMax": "10.99.0.0",
    "Backend": {
        "Type": "udp",
        "Port": 7890
    }
}
```

除了UDP，Flannel还支持很多其他的Backend：

- udp：使用用户态udp封装，默认使用8285端口。由于是在用户态封装和解包，性能上有较大的损失
- vxlan：vxlan封装，需要配置VNI，Port（默认8472）和GBP
- host-gw：直接路由的方式，将容器网络的路由信息直接更新到主机的路由表中，仅适用于二层直接可达的网络
- aws-vpc：使用 Amazon VPC route table 创建路由，适用于AWS上运行的容器
- gce：使用Google Compute Engine Network创建路由，所有instance需要开启IP forwarding，适用于GCE上运行的容器
- ali-vpc：使用阿里云VPC route table 创建路由，适用于阿里云上运行的容器

Docker集成

```
source /run/flannel/subnet.env
docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} &
```

CNI集成

CNI flannel插件会将flannel网络配置转换为bridge插件配置，并调用bridge插件给容器netns配置网络。比如下面的flannel配置

```
{
    "name": "mynet",
    "type": "flannel",
    "delegate": {
        "bridge": "mynet0",
        "mtu": 1400
    }
}
```

会被cni flannel插件转换为

```
{
    "name": "mynet",
    "type": "bridge",
    "mtu": 1472,
    "ipMasq": false,
    "isGateway": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.1.17.0/24"
    }
}
```

Kubernetes集成

使用flannel前需要配置 `kube-controller-manager --allocate-node-cidrs=true --cluster-cidr=10.244.0.0/16` 。

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel
```

这会启动flanneld容器，并配置CNI网络插件：

```
$ ps -ef | grep flannel | grep -v grep
root      3625  3610  0 13:57 ?        00:00:00 /opt/bin/flanneld
root      9640  9619  0 13:51 ?        00:00:00 /bin/sh -c set -e

$ cat /etc/cni/net.d/10-flannel.conf
{
  "name": "cbr0",
  "type": "flannel",
  "delegate": {
    "isDefaultGateway": true
  }
}
```

flanneld自动连接kubernetes API，根据 `node.Spec.PodCIDR` 配置本地的flannel网络子网，并为容器创建vxlan和相关的子网路由。

```
$ cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.0.1/24
FLANNEL_MTU=1410
FLANNEL_IPMASQ=true

$ ip -d link show flannel.1
12: flannel.1:  mtu 1410 qdisc noqueue state UNKNOWN mode DEFAULT
    link/ether 8e:5a:0d:07:0f:0d brd ff:ff:ff:ff:ff:ff promiscuit
    vxlan id 1 local 10.146.0.2 dev ens4 srcport 0 0 dstport 8472
```

优点

- 配置安装简单，使用方便
- 与云平台集成较好，VPC的方式没有额外的性能损失

缺点

- VXLAN模式对zero-downtime restarts支持不好

> When running with a backend other than udp, the kernel is providing the data path with flanneld acting as the control plane. As such, flanneld can be restarted (even to do an upgrade) without disturbing existing flows. However in the case of vxlan backend, this needs to be done within a few seconds as ARP entries can start to timeout requiring the flannel daemon to refresh them. Also, to avoid interruptions during restart, the configuration must not be changed (e.g. VNI, --iface values).

参考文档

- https://github.com/coreos/flannel
- https://coreos.com/flannel/docs/latest/

Calico

Calico 是一个纯三层的数据中心网络方案（不需要 Overlay），并且与 OpenStack、Kubernetes、AWS、GCE 等 IaaS 和容器平台都有良好的集成。

Calico 在每一个计算节点利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发，而每个 vRouter 通过 BGP 协议负责把自己上运行的 workload 的路由信息像整个 Calico 网络内传播——小规模部署可以直接互联，大规模下可通过指定的 BGP route reflector 来完成。 这样保证最终所有的 workload 之间的数据流量都是通过 IP 路由的方式完成互联的。Calico 节点组网可以直接利用数据中心的网络结构（无论是 L2 或者 L3），不需要额外的 NAT，隧道或者 Overlay Network。

此外，Calico 基于 iptables 还提供了丰富而灵活的网络 Policy，保证通过各个节点上的 ACLs 来提供 Workload 的多租户隔离、安全组以及其他可达性限制等功能。

Calico 架构

Calico 主要由 Felix、etcd、BGP client 以及 BGP Route Reflector 组成

1. Felix，Calico Agent，跑在每台需要运行 Workload 的节点上，主要负责配置路由及 ACLs 等信息来确保 Endpoint 的连通状态；

2. etcd，分布式键值存储，主要负责网络元数据一致性，确保 Calico 网络状态的准确性；
3. BGP Client（BIRD），主要负责把 Felix 写入 Kernel 的路由信息分发到当前 Calico 网络，确保 Workload 间的通信的有效性；
4. BGP Route Reflector（BIRD），大规模部署时使用，摒弃所有节点互联的 mesh 模式，通过一个或者多个 BGP Route Reflector 来完成集中式的路由分发。
5. calico/calico-ipam，主要用作 Kubernetes 的 CNI 插件

IP-in-IP

Calico 控制平面的设计要求物理网络得是 L2 Fabric，这样 vRouter 间都是直接可达的，路由不需要把物理设备当做下一跳。为了支持 L3 Fabric，Calico 推出了 IPinIP 的选项。

Calico CNI

见 https://github.com/projectcalico/cni-plugin。

Calico CNM

Calico 通过 Pool 和 Profile 的方式实现了 docker CNM 网络：

1. Pool，定义可用于 Docker Network 的 IP 资源范围，比如：10.0.0.0/8 或者 192.168.0.0/16；
2. Profile，定义 Docker Network Policy 的集合，由 tags 和 rules 组成；每个 Profile 默认拥有一个和 Profile 名字相同的 Tag，每个 Profile 可以有多个 Tag，以 List 形式保存。

具体实现见 https://github.com/projectcalico/libnetwork-plugin，而使用方法可以参考 https://docs.projectcalico.org/master/getting-started/docker/。

Calico Kubernetes

对于使用 kubeadm 创建的 Kubernetes 集群，使用以下配置安装 calico 时需要配置

- `--pod-network-cidr=192.168.0.0/16`
- `--service-cidr=10.96.0.0/12` （不能与 Calico 网络重叠）

然后运行

```
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-star
```

更详细的自定义配置方法见 https://docs.projectcalico.org/v3.0/getting-started/kubernetes。

这会在 Pod 中启动 Calico-etcd，在所有 Node 上启动 bird6、felix 以及 confd，并配置 CNI 网络为 calico 插件：

```
# Calico 相关进程
$ ps -ef | grep calico | grep -v grep
root      9012  8995  0 14:51 ?        00:00:00 /bin/sh -c /usr/l
root      9038  9012  0 14:51 ?        00:00:01 /usr/local/bin/et
root      9326  9325  0 14:51 ?        00:00:00 bird6 -R -s /var/
root      9327  9322  0 14:51 ?        00:00:00 confd -confdir=/e
root      9328  9324  0 14:51 ?        00:00:00 bird -R -s /var/r
root      9329  9323  1 14:51 ?        00:00:04 calico-felix
```

```
# CNI 网络插件配置
$ cat /etc/cni/net.d/10-calico.conf
{
    "name": "k8s-pod-network",
    "cniVersion": "0.1.0",
    "type": "calico",
    "etcd_endpoints": "http://10.96.232.136:6666",
    "log_level": "info",
    "ipam": {
        "type": "calico-ipam"
    },
    "policy": {
        "type": "k8s",
         "k8s_api_root": "https://10.96.0.1:443",
         "k8s_auth_token": ""
    },
    "kubernetes": {
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
    }
}

$ cat /etc/cni/net.d/calico-kubeconfig
# Kubeconfig file for Calico CNI plugin.
apiVersion: v1
kind: Config
clusters:
- name: local
  cluster:
    insecure-skip-tls-verify: true
users:
- name: calico
contexts:
- name: calico-context
  context:
    cluster: local
    user: calico
current-context: calico-context
```

Calico 的不足

- 既然是三层实现，当然不支持 VRF
- 不支持多租户网络的隔离功能，在多租户场景下会有网络安全问题
- Calico 控制平面的设计要求物理网络得是 L2 Fabric，这样 vRouter 间都是直接可达的

参考文档

- https://xuxinkun.github.io/2016/07/22/cni-cnm/
- https://www.projectcalico.org/
- http://blog.dataman-inc.com/shurenyun-docker-133/

Weave

Weave Net是一个多主机容器网络方案，支持去中心化的控制平面，各个host上的wRouter间通过建立Full Mesh的TCP链接，并通过Gossip来同步控制信息。这种方式省去了集中式的K/V Store，能够在一定程度上减低部署的复杂性，Weave将其称为“data centric”，而非RAFT或者Paxos的“algorithm centric”。

数据平面上，Weave通过UDP封装实现L2 Overlay，封装支持两种模式：

- 运行在user space的sleeve mode：通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。
- 运行在kernal space的 fastpath mode：即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp 流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

Sleeve Mode:

Fastpath Mode:

关于Service的发布，weave做的也比较完整。首先，wRouter集成了DNS功能，能够动态地进行服务发现和负载均衡，另外，与libnetwork 的overlay driver类似，weave要求每个POD有两个网卡，一个就连在lb/ovs上处理L2 流量，另一个则连在docker0上处理Service流量，docker0后面仍然是iptables作NAT。

Weave已经集成了主流的容器系统

- Docker: https://www.weave.works/docs/net/latest/plugin/
- Kubernetes: https://www.weave.works/docs/net/latest/kube-addon/
  - `kubectl apply -f https://git.io/weave-kube`
- CNI: https://www.weave.works/docs/net/latest/cni-plugin/
- Prometheus: https://www.weave.works/docs/net/latest/metrics/

Weave Kubernetes

```
kubectl apply -n kube-system -f "https://cloud.weave.works/k8s/ne
```

这会在所有Node上启动Weave插件以及Network policy controller：

```
$ ps -ef | grep weave | grep -v grep
root     25147 25131  0 16:22 ?        00:00:00 /bin/sh /home/wea
root     25204 25147  0 16:22 ?        00:00:00 /home/weave/weave
root     25669 25654  0 16:22 ?        00:00:00 /usr/bin/weave-np
```

这样，容器网络为

- 所有容器都连接到weave网桥
- weave网桥通过veth pair连到内核的openvswitch模块
- 跨主机容器通过openvswitch vxlan通信
- policy controller通过配置iptables规则为容器设置网络策略

Weave Scope

Weave Scope是一个容器监控和故障排查工具，可以方便的生成整个集群的拓扑并智能分组（Automatic Topologies and Intelligent Grouping）。

Weave Scope主要由scope-probe和scope-app组成

```
+--Docker host----------+
|  +--Container------+  |    .---------------.
|  |                 |  |    | Browser       |
|  |  +------------+  |  |    |---------------|
|  |  | scope-app |<---------|               |
|  |  +------------+  |  |    |               |
|  |        ^         |  |    |               |
|  |        |         |  |    '---------------'
|  | +-------------+ |  |
|  | | scope-probe | |  |
|  | +-------------+ |  |
|  |                 |  |
|  +-----------------+  |
+-----------------------+
```

优点

- 去中心化
- 故障自动恢复
- 加密通信
- Multicast networking

缺点

- UDP模式性能损失较大

参考文档

- https://github.com/weaveworks/weave
- https://www.weave.works/products/weave-net/
- https://github.com/weaveworks/scope
- https://www.weave.works/guides/monitor-docker-containers/
- http://www.sdnlab.com/17141.html

Cilium

Cilium 是一个基于 eBPF 和 XDP 的高性能容器网络方案，代码开源在 https://github.com/cilium/cilium。其主要功能特性包括

- 安全上，支持 L3/L4/L7 安全策略，这些策略按照使用方法又可以分为
  - 基于身份的安全策略（security identity）
  - 基于 CIDR 的安全策略
  - 基于标签的安全策略
- 网络上，支持三层平面网络（flat layer 3 network），如
  - 覆盖网络（Overlay），包括 VXLAN 和 Geneve 等
  - Linux 路由网络，包括原生的 Linux 路由和云服务商的高级网络路由等
- 提供基于 BPF 的负载均衡
- 提供便利的监控和排错能力

eBPF 和 XDP

eBPF（extended Berkeley Packet Filter）起源于BPF，它提供了内核的数据包过滤机制。BPF的基本思想是对用户提供两种SOCKET选项：`SO_ATTACH_FILTER`和`SO_ATTACH_BPF`，允许用户在sokcet上添加自定义的filter，只有满足该filter指定条件的数据包才会上发到用户空间。`SO_ATTACH_FILTER`插入的是cBPF代码，`SO_ATTACH_BPF`插入的是eBPF代码。eBPF是对cBPF的增强，目前用户端的tcpdump等程序还是用的cBPF版本，其加载到内核中后会被内核自动的转变为eBPF。Linux 3.15 开始引入 eBPF。其扩充了 BPF 的功能，丰富了指令集。它在内核提供了一个虚拟机，用户态将过滤规则以虚拟机指令的形式传递到内核，由内核根据这些指令来过滤网络数据包。

XDP（eXpress Data Path）为Linux内核提供了高性能、可编程的网络数据路径。由于网络包在还未进入网络协议栈之前就处理，它给Linux网络带来了巨大的性能提升。XDP 看起来跟 DPDK 比较像，但它比 DPDK 有更多的优点，如

- 无需第三方代码库和许可
- 同时支持轮询式和中断式网络
- 无需分配大页
- 无需专用的CPU
- 无需定义新的安全网络模型

当然，XDP的性能提升是有代价的，它牺牲了通用型和公平性：（1）不提供缓存队列（qdisc），TX设备太慢时直接丢包，因而不要在RX比TX快的设备上使用XDP；（2）XDP程序是专用的，不具备网络协议栈的通用性。

部署

版本要求

- Linux Kernel >= 4.8 （推荐 4.9.17 LTS）
- KV 存储（etcd >= 3.1.0 或 consul >= 0.6.4）

Kubernetes Cluster

```
# mount BPF filesystem on all nodes
$ mount bpffs /sys/fs/bpf -t bpf

$ wget https://raw.githubusercontent.com/cilium/cilium/doc-1.0/ex
$ vim cilium.yaml
[adjust the etcd address]

$ kubectl create -f ./cilium.yaml
```

minikube

```
minikube start --network-plugin=cni --bootstrapper=localkube --me
kubectl create clusterrolebinding kube-system-default-binding-clu
kubectl create -f https://raw.githubusercontent.com/cilium/cilium
kubectl create -f https://raw.githubusercontent.com/cilium/cilium
```

Istio

```
# cluster clusterrolebindings
kubectl create clusterrolebinding kube-system-default-binding-clu
# etcd
kubectl create -f https://raw.githubusercontent.com/cilium/cilium

# cilium
curl -s https://raw.githubusercontent.com/cilium/cilium/HEAD/exam
  sed -e 's/sidecar-http-proxy: "false"/sidecar-http-proxy: "true
  kubectl create -f -

# Istio
curl -L https://git.io/getLatestIstio | sh -
ISTIO_VERSION=$(curl -L -s https://api.github.com/repos/istio/ist
cd istio-${ISTIO_VERSION}
cp bin/istioctl /usr/local/bin

# Patch with cilium pilot
sed -e 's,docker\.io/istio/pilot:,docker.io/cilium/istio_pilot:,'
      < install/kubernetes/istio.yaml | \
      kubectl create -f -

# Configure Istio's sidecar injection to use Cilium's Docker imag
kubectl create -f https://raw.githubusercontent.com/cilium/cilium
```

安全策略

TCP 策略：

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L3-L4 policy to restrict deathstar access to empire
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

CIDR 策略

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "cidr-rule"
spec:
  endpointSelector:
    matchLabels:
      app: myService
  egress:
  - toCIDR:
    - 20.1.1.1/32
  - toCIDRSet:
    - cidr: 10.0.0.0/8
      except:
      - 10.96.0.0/12
```

L7 HTTP 策略：

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L7 policy to restrict access to specific HTTP call"
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        http:
        - method: "POST"
          path: "/v1/request-landing"
```

监控

microscope 汇集了所有 Nodes 的监控数据（从 `cilium monitor` 获取）。使用方法为：

```
$ kubectl apply -f
https://github.com/cilium/microscope/blob/master/docs/microscope.
$ kubectl exec -n kube-system microscope -- microscope -h
```

参考资料

- Cilium documentation

OVN

ovn-kubernetes 提供了一个ovs OVN 网络插件，支持 underlay 和 overlay 两种模式。

- underlay：容器运行在虚拟机中，而ovs则运行在虚拟机所在的物理机上，OVN将容器网络和虚拟机网络连接在一起
- overlay：OVN通过logical overlay network连接所有节点的容器，此时ovs可以直接运行在物理机或虚拟机上

Overlay模式

配置master

```
# start ovn
/usr/share/openvswitch/scripts/ovn-ctl start_northd
/usr/share/openvswitch/scripts/ovn-ctl start_controller

# start ovnkube
nohup sudo ovnkube -k8s-kubeconfig kubeconfig.yaml -net-controlle
 -loglevel=4 \
 -k8s-apiserver="http://$CENTRAL_IP:8080" \
 -logfile="/var/log/openvswitch/ovnkube.log" \
 -init-master=$NODE_NAME -cluster-subnet="$CLUSTER_IP_SUBNET" \
 -service-cluster-ip-range=$SERVICE_IP_SUBNET \
 -nodeport \
 -nb-address="tcp://$CENTRAL_IP:6631" \
 -sb-address="tcp://$CENTRAL_IP:6632" 2>&1 &
```

配置Node

```
nohup sudo ovnkube -k8s-kubeconfig kubeconfig.yaml -loglevel=4 \
    -logfile="/var/log/openvswitch/ovnkube.log" \
    -k8s-apiserver="http://$CENTRAL_IP:8080" \
    -init-node="$NODE_NAME"  \
    -nodeport \
    -nb-address="tcp://$CENTRAL_IP:6631" \
    -sb-address="tcp://$CENTRAL_IP:6632" -k8s-token="$TOKEN" \
    -init-gateways \
    -service-cluster-ip-range=$SERVICE_IP_SUBNET \
    -cluster-subnet=$CLUSTER_IP_SUBNET 2>&1 &
```

CNI插件原理

ADD操作

- 从`ovn annotation`获取ip/mac/gateway
- 在容器netns中配置接口和路由
- 添加ovs端口

```
ovs-vsctl add-port br-int veth_outside \
  --set interface veth_outside \
    external_ids:attached_mac=mac_address \
    external_ids:iface-id=namespace_pod \
    external_ids:ip_address=ip_address
```

DEL操作

```
ovs-vsctl del-port br-int port
```

Underlay模式

暂未实现。

OVN 安装方法

所有节点配置安装源并安装公共依赖

```
sudo apt-get install apt-transport-https
echo "deb https://packages.wand.net.nz $(lsb_release -sc) main" |
sudo curl https://packages.wand.net.nz/keyring.gpg -o /etc/apt/tr
sudo apt-get update

sudo apt-get build-dep dkms
sudo apt-get install python-six openssl python-pip -y
sudo -H pip install --upgrade pip

sudo apt-get install openvswitch-datapath-dkms -y
sudo apt-get install openvswitch-switch openvswitch-common -y
sudo -H pip install ovs
```

Master 节点安装 ovn-central

```
sudo apt-get install ovn-central ovn-common ovn-host -y
```

Node 节点安装 ovn-host

```
sudo apt-get install ovn-host ovn-common -y
```

参考文档

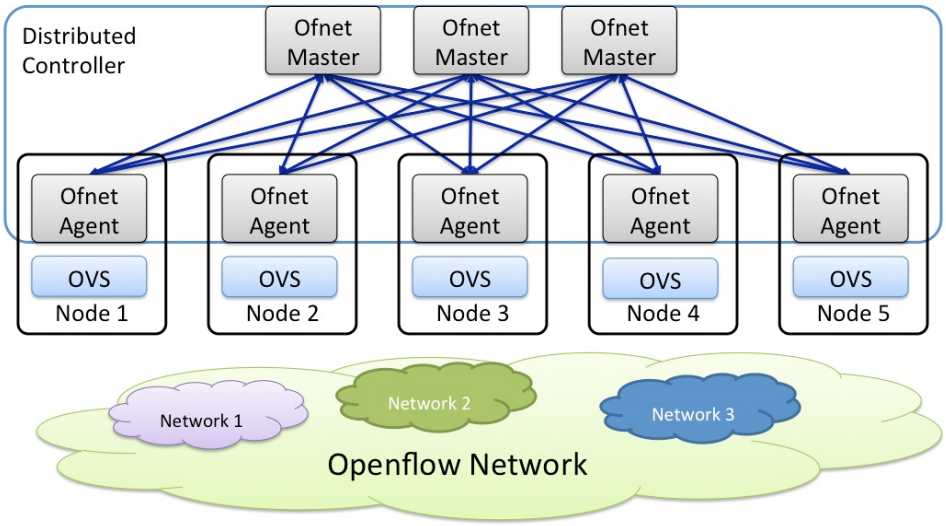
- https://github.com/openvswitch/ovn-kubernetes

Contiv

Contiv是思科开源的容器网络方案，是一个用于跨虚拟机、裸机、公有云或私有云的异构容器部署的开源容器网络架构，并与主流容器编排系统集成。Contiv最主要的优势是直接提供了多租户网络，并支持L2(VLAN)，L3(BGP)，Overlay (VXLAN)以及思科自家的ACI。

主要特征

- 原生的Tenant支持，一个Tenant就是一个virtual routing and forwarding (VRF)
- 两种网络模式
  - L2 VLAN Bridged
  - Routed network, e.g. vxlan, BGP, ACI
- Network Policy，如Bandwidth，Isolation等



Kubernetes集成

Ansible部署见https://github.com/kubernetes/contrib/tree/master/ansible/roles/contiv。

```
export VERSION=1.0.0-beta.3
curl -L -O https://github.com/contiv/install/releases/download/$V
tar xf contiv-$VERSION.tgz
cd ~/contiv/contiv-$VERSION/install/k8s
netctl --netmaster http://$netmaster:9999 global set --fwd-mode r

cd ~/contiv/contiv-$VERSION
install/k8s/install.sh -n 10.87.49.77 -v b -w routing

# check contiv pods
export NETMASTER=http://10.87.49.77:9999
netctl global info

# create a network
# netctl network create --encap=vlan --pkt-tag=3280 --subnet=10.1
netctl net create -t default --subnet=20.1.1.0/24 default-net

#  create BGP connections to each of the nodes
netctl bgp create devstack-77 --router-ip="30.30.30.77/24" --as="
netctl bgp create devstack-78 --router-ip="30.30.30.78/24" --as="
netctl bgp create devstack-71 --router-ip="30.30.30.79/24" --as="

# then create pod with label "io.contiv.network"
```

参考文档

- http://contiv.github.io/
- https://github.com/contiv/netplugin
- http://blogs.cisco.com/cloud/introducing-contiv-1-0
- Kubernetes and Contiv on Bare-Metal with L3/BGP

SR-IOV

SR-IOV 技术是一种基于硬件的虚拟化解决方案，可提高性能和可伸缩性

> SR-IOV 标准允许在虚拟机之间高效共享 PCIe（Peripheral Component Interconnect Express，快速外设组件互连）设备，并且它是在硬件中实现的，可以获得能够与本机性能媲美的 I/O 性能。SR-IOV 规范定义了新的标准，根据该标准，创建的新设备可允许将虚拟机直接连接到 I/O 设备（SR-IOV 规范由 PCI-SIG 在 http://www.pcisig.com 上进行定义和维护）。单个 I/O 资源可由许多虚拟机共享。共享的设备将提供专用的资源，并且还使用共享的通用资源。这样，每个虚拟机都可访问唯一的资源。因此，启用了 SR-IOV 并且具有适当的硬件和 OS 支持的 PCIe 设备（例如以太网端口）可以显示为多个单独的物理设备，每个都具有自己的 PCIe 配置空间。

SR-IOV主要用于虚拟化中，当然也可以用于容器。

SR-IOV配置

```
modprobe ixgbevf
lspci –Dvmm|grep –B 1 –A 4 Ethernet
echo 2 > /sys/bus/pci/devices/0000:82:00.0/sriov_numvfs
# check ifconfig –a. You should see a number of new interfaces cr
```

docker sriov plugin

Intel给docker写了一个SR-IOV network plugin，源码位于https://github.com/clearcontainers/sriov，同时支持runc和clearcontainer。

CNI插件

Intel维护了一个SR-IOV的CNI插件，fork自hustcat/sriov-cni，并扩展了DPDK的支持。

项目主页见https://github.com/Intel-Corp/sriov-cni。

优点

- 性能好

- 不占用计算资源

缺点

- VF数量有限
- 硬件绑定，不支持容器迁移

参考文档

- http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/
- https://github.com/clearcontainers/sriov
- https://software.intel.com/en-us/articles/single-root-inputoutput-virtualization-sr-iov-with-linux-containers
- http://jason.digitalinertia.net/exposing-docker-containers-with-sr-iov/

Romana

Romana是Panic Networks在2016年提出的开源项目，旨在解决Overlay方案给网络带来的开销。

Kubernetes部署

对使用kubeadm部署的Kubernetes集群：

```
kubectl apply -f https://raw.githubusercontent.com/romana/romana/
```

对使用kops部署的Kubernetes集群：

```
kubectl apply -f https://raw.githubusercontent.com/romana/romana/
```

使用kops时要注意

- 设置网络插件使用CNI `--networking cni`
- 对于aws还提供 `romana-aws` 和 `romana-vpcrouter` 自动配置Node和Zone之间的路由

工作原理

- layer 3 networking，消除overlay带来的开销
- 基于iptables ACL的网络隔离
- 基于hierarchy CIDR管理Host/Tenant/Segment ID

优点

- 纯三层网络，性能好

缺点

- 基于IP管理租户，有规模上的限制
- 物理设备变更或地址规划变更麻烦

参考文档

- http://romana.io/
- Romana basics
- Romana Github
- Romana 2.0

OpenContrail

OpenContrail是Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。

架构

OpenContrail主要由控制器和vRouter组成：

- 控制器提供虚拟网络的配置、控制和分析功能
- vRouter提供分布式路由，负责虚拟路由器、虚拟网络的建立以及数据转发

vRouter支持三种模式

- Kernel vRouter：类似于ovs内核模块

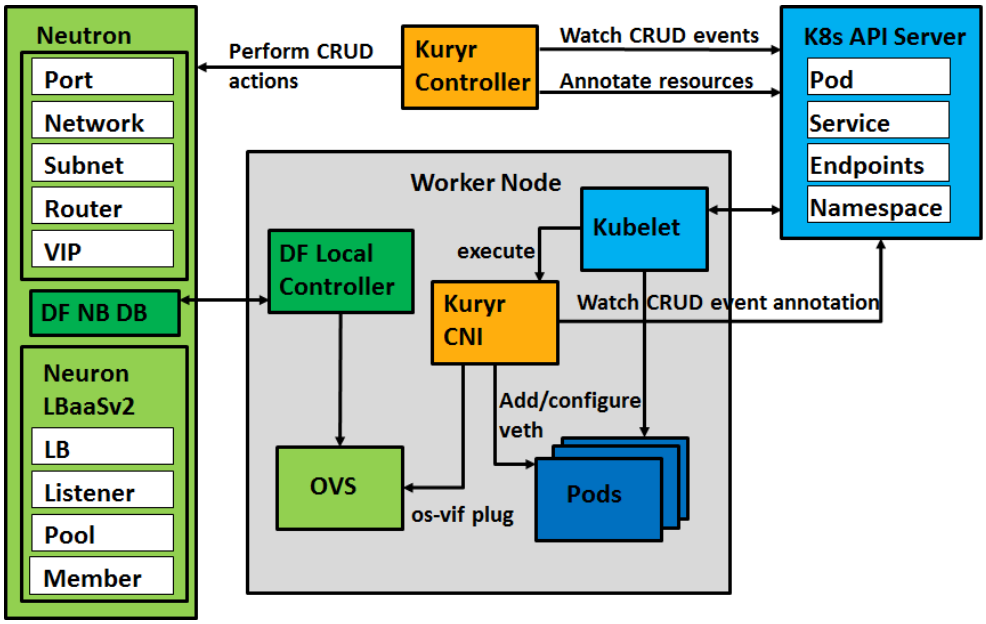
- DPDK vRouter：类似于ovs-dpdk
- Netronome Agilio Solution（商业产品）：支持DPDK, SR-IOV and Express Virtio (XVIO)

参考文档

- http://www.opencontrail.org/opencontrail-architecture-documentation/
- http://www.opencontrail.org/network-virtualization-architecture-deep-dive/

Kuryr

Kuryr 是 OpenStack Neutron 的子项目，其主要目标是透过该项目来集成 OpenStack 与 Kubernetes 的网络。该项目在 Kubernetes 中实作了原生 Neutron-based 的网络，因此使用 Kuryr-Kubernetes 可以让 OpenStack VM 与 Kubernetes Pods 能够选择在同一个子网络上运作，并且能够使用 Neutron L3 与 Security Group 来对网络进行路由，以及阻挡特定来源 Port，并且也提供基于 Neutron LBaaS 的 Service 集成。



Kuryr-Kubernetes 有以两个主要部分组成：

1. **Kuryr Controller**: Controller 主要目的是监控 Kubernetes API 的来获取 Kubernetes 资源的变化，然后依据 Kubernetes 资源的需求来运行子资源的分配和资源管理。
2. **Kuryr CNI**：主要是依据 Kuryr Controller 分配的资源来绑定网络至 Pods 上。

devstack 部署

最简单的方式是使用 devstack 部署一个单机环境：

```
$ git clone https://git.openstack.org/openstack-dev/devstack
$ ./devstack/tools/create-stack-user.sh
$ sudo su stack

$ git clone https://git.openstack.org/openstack-dev/devstack
$ git clone https://git.openstack.org/openstack/kuryr-kubernetes
$ cp kuryr-kubernetes/devstack/local.conf.sample devstack/local.c

# start install
$ ./devstack/stack.sh
```

部署完成后，验证安装成功

```
$ source /devstack/openrc admin admin
$ openstack service list
+----------------------------------+------------------+----------
| ID                               | Name             | Type
+----------------------------------+------------------+----------
| 091e3e2813cc4904b74b60c41e8a98b3 | kuryr-kubernetes | kuryr-kub
| 2b6076dd5fc04bf180e935f78c12d431 | neutron          | network
| b598216086944714aed2c233123fc22d | keystone         | identity
+----------------------------------+------------------+----------

$ kubectl get nodes
NAME        STATUS    AGE       VERSION
localhost   Ready     2m        v1.6.2
```

多机部署

本篇我們將說明如何利用 `DevStack` 與 `Kubespray` 建立一個簡單的測試環境。

环境资源与事前准备

准备两台实体机器，这边测试的作业系统为 `CentOS 7.x`，该环境将在平面的网络下进行。

IP Address 1	Role
172.24.0.34	controller, k8s-master
172.24.0.80	compute1, k8s-node1
172.24.0.81	compute2, k8s-node2

更新每台节点的 CentOS 7.x 包:

```
$ sudo yum --enablerepo=cr update -y
```

然后关闭 firewalld 以及 SELinux 来避免实现发生问题：

```
$ sudo setenforce 0
$ sudo systemctl disable firewalld && sudo systemctl stop firewal
```

OpenStack Controller 安装

首先进入 `172.24.0.34 (controller)` ，并且运行以下命令。

然后运行以下命令来建立 DevStack 专用用户：

```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/
```

接着切换至该用户环境来创建 OpenStack：

```
$ sudo su - stack
```

下载 DevStack：

```
$ git clone https://git.openstack.org/openstack-dev/devstack
$ cd devstack
```

新增 `local.conf` 文档，来描述部署资讯：

```
[[local|localrc]]
HOST_IP=172.24.0.34
GIT_BASE=https://github.com

ADMIN_PASSWORD=passwd
DATABASE_PASSWORD=passwd
RABBIT_PASSWORD=passwd
SERVICE_PASSWORD=passwd
SERVICE_TOKEN=passwd
MULTI_HOST=1
```

> 修改 HOST_IP 为自己的 IP 。

完成后，运行以下命令开始部署：

```
$ ./stack.sh
```

Openstack Compute 安装

进入到 `172.24.0.80（compute）` 與 `172.24.0.81（node2）`，并且运行以下命令。

然后运行以下命令来建立 DevStack 专用用户：

```
$ sudo useradd -s /bin/bash -d /opt/stack -m stack
$ echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/
```

接着切换至该用户环境来创建 OpenStack：

```
$ sudo su - stack
```

下载 DevStack：

```
$ git clone https://git.openstack.org/openstack-dev/devstack
$ cd devstack
```

新增 `local.conf` 文档，来描述部署资讯：

```
[[local|localrc]]
HOST_IP=172.24.0.80
GIT_BASE=https://github.com
MULTI_HOST=1
LOGFILE=/opt/stack/logs/stack.sh.log
ADMIN_PASSWORD=passwd
DATABASE_PASSWORD=passwd
RABBIT_PASSWORD=passwd
SERVICE_PASSWORD=passwd
DATABASE_TYPE=mysql

SERVICE_HOST=172.24.0.34
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
ENABLED_SERVICES=n-cpu,q-agt,n-api-meta,c-vol,placement-client
NOVA_VNC_ENABLED=True
NOVNCPROXY_URL="http://$SERVICE_HOST:6080/vnc_auto.html"
VNCSERVER_LISTEN=$HOST_IP
VNCSERVER_PROXYCLIENT_ADDRESS=$VNCSERVER_LISTEN
```

> 修改 HOST_IP 为自己的主机位置。 修改 SERVICE_HOST 为 Master 的 IP。

完成后，运行以下命令开始部署：

```
$ ./stack.sh
```

创建 Kubernetes 集群环境

首先确认所有节点之间不需要 SSH 密码即可登入，接着进入到 `172.24.0.34 (k8s-master)` 并且运行以下命令。

接着安装所需要的软件包：

```
$ sudo yum -y install software-properties-common ansible git gcc
$ sudo pip install -U kubespray
```

完成后，创建 kubespray 配置档：

```
$ cat <  ~/.kubespray.yml
kubespray_git_repo: "https://github.com/kubernetes-incubator/kube
# Logging options
loglevel: "info"
EOF
```

利用 `kubespray-cli` 快速产生环境的 `inventory` 文档，并修改部分内容：

```
$ sudo -i
$ kubespray prepare --masters master --etcds master --nodes node1
```

编辑 `/root/.kubespray/inventory/inventory.cfg` 文档，修改以下内容：

```
[all]
master  ansible_host=172.24.0.34 ansible_user=root ip=172.24.0.34
node1    ansible_host=172.24.0.80 ansible_user=root ip=172.24.0.8
node2    ansible_host=172.24.0.81 ansible_user=root ip=172.24.0.8

[kube-master]
master

[kube-node]
master
node1
node2

[etcd]
master

[k8s-cluster:children]
kube-node
kube-master
```

完成后，即可利用 `kubespray-cli` 来进行部署：

```
$ kubespray deploy --verbose -u root -k .ssh/id_rsa -n calico
```

经过一段时间后就会部署完成，这时候检查节点是否正常：

```
$ kubectl get no
NAME      STATUS         AGE       VERSION
master    Ready,master   2m        v1.7.4
node1     Ready          2m        v1.7.4
node2     Ready          2m        v1.7.4
```

接着为了方便让 Kuryr Controller 简单取得 K8s API Server，这边修改 `/etc/kubernetes/manifests/kube-apiserver.yml` 文档，加入以下内容：

```
- "--insecure-bind-address=0.0.0.0"
- "--insecure-port=8080"
```

> 将 insecure 绑定到 0.0.0.0 之上，以及开启 8080 Port。

安装 Openstack Kuryr Controller

进入到 `172.24.0.34 (controller)`，并且运行以下命令。

首先在节点安装所需要的软件包：

```
$ sudo yum -y install  gcc libffi-devel python-devel openssl-deve
```

下载 kuryr-kubernetes 并进行安装：

```
$ git clone http://git.openstack.org/openstack/kuryr-kubernetes
$ pip install -e kuryr-kubernetes
```

创建 `kuryr.conf` 至 `/etc/kuryr` 目录

```
$ cd kuryr-kubernetes
$ ./tools/generate_config_file_samples.sh
$ sudo mkdir -p /etc/kuryr/
$ sudo cp etc/kuryr.conf.sample /etc/kuryr/kuryr.conf
```

使用 OpenStack Dashboard 建立项目，在浏览器输入 `http://172.24.0.34`，并运行以下步骤。

1. 创建 k8s project。
2. 创建 kuryr-kubernetes service，并修改 k8s project member 加入到 service project。
3. 在该 Project 中新增 Security Groups，参考 kuryr-kubernetes manually。
4. 在该 Project 中新增 pod_subnet 子网络。
5. 在该 Project 中新增 service_subnet 子网络。

完成后，修改 `/etc/kuryr/kuryr.conf` 文档，加入以下内容：

```
[DEFAULT]
use_stderr = true
bindir = /usr/local/libexec/kuryr

[kubernetes]
api_root = http://172.24.0.34:8080

[neutron]
auth_url = http://172.24.0.34/identity
username = admin
user_domain_name = Default
password = admin
project_name = service
project_domain_name = Default
auth_type = password

[neutron_defaults]
ovs_bridge = br-int
pod_security_groups = {id_of_secuirity_group_for_pods}
pod_subnet = {id_of_subnet_for_pods}
project = {id_of_project}
service_subnet = {id_of_subnet_for_k8s_services}
```

完成后运行 kuryr-k8s-controller：

```
$ kuryr-k8s-controller --config-file /etc/kuryr/kuryr.conf&
```

安装 Kuryr-CNI

进入到 `172.24.0.80 (node1)` 與 `172.24.0.81 (node2)` 并运行以下命令。

首先在节点安装所需要的软件包

```
$ sudo yum -y install  gcc libffi-devel python-devel openssl-deve
```

安装 Kuryr-CNI 来提供给 kubelet 使用：

```
$ git clone http://git.openstack.org/openstack/kuryr-kubernetes
$ sudo pip install -e kuryr-kubernetes
```

创建 `kuryr.conf` 至 `/etc/kuryr` 目录：

```
$ cd kuryr-kubernetes
$ ./tools/generate_config_file_samples.sh
$ sudo mkdir -p /etc/kuryr/
$ sudo cp etc/kuryr.conf.sample /etc/kuryr/kuryr.conf
```

修改 `/etc/kuryr/kuryr.conf` 文档，加入以下内容：

```
[DEFAULT]
use_stderr = true
bindir = /usr/local/libexec/kuryr
[kubernetes]
api_root = http://172.24.0.34:8080
```

创建 CNI bin 与 Conf 目录：

```
$ sudo mkdir -p /opt/cni/bin
$ sudo ln -s $(which kuryr-cni) /opt/cni/bin/
$ sudo mkdir -p /etc/cni/net.d/
```

新增 `/etc/cni/net.d/10-kuryr.conf` CNI 配置档：

```
{
    "cniVersion": "0.3.0",
    "name": "kuryr",
    "type": "kuryr-cni",
    "kuryr_conf": "/etc/kuryr/kuryr.conf",
    "debug": true
}
```

完成后，更新 oslo 与 vif python 库：

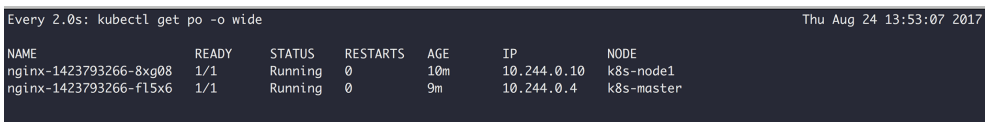
```
$ sudo pip install 'oslo.privsep>=1.20.0' 'os-vif>=1.5.0'
```

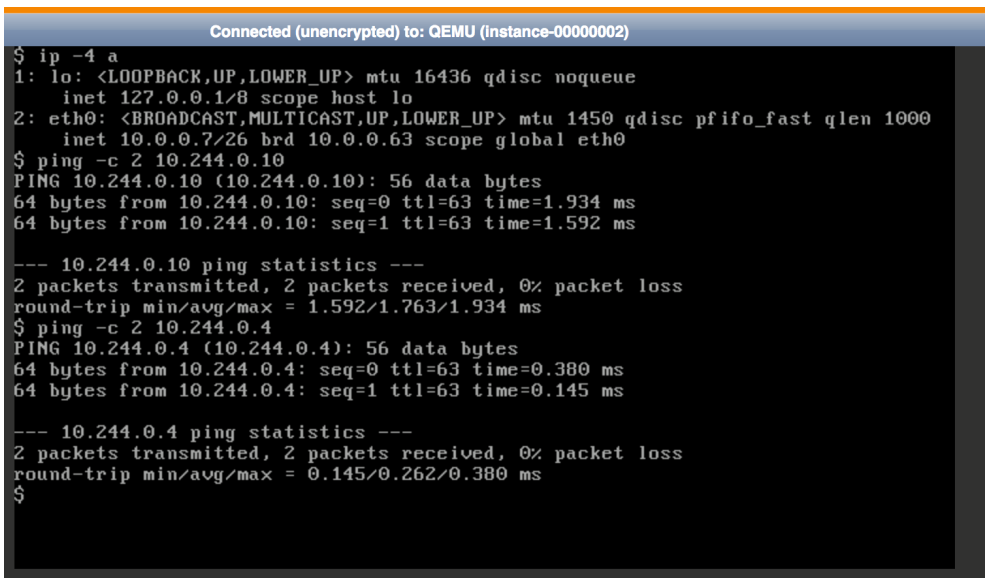
最后重新启动服务：

```
$ sudo systemctl daemon-reload && systemctl restart kubelet.servi
```

测试结果

创建一个 Pod 与 OpenStack VM 来进行通信：





参考文档

- Kuryr kubernetes documentation

运行时插件CRI

容器运行时插件（Container Runtime Interface，简称 CRI）是 Kubernetes v1.5 引入的容器运行时接口，它将 Kubelet 与容器运行时解耦，将原来完全面向 Pod 级别的内部接口拆分成面向 Sandbox 和 Container 的 gRPC 接口，并将镜像管理和容器管理分离到不同的服务。

CRI 最早从从 1.4 版就开始设计讨论和开发，在 v1.5 中发布第一个测试版。在 v1.6 时已经有了很多外部容器运行时，如 frakti 和 cri-o 等。v1.7 中又新增了 cri-containerd 支持用 Containerd 来管理容器。

CRI 接口

CRI 基于 gRPC 定义了 RuntimeService 和 ImageService 等两个 gRPC 服务，分别用于容器运行时和镜像的管理。其定义在

- v1.10-v1.12: pkg/kubelet/apis/cri/runtime/v1alpha2
- v1.7-v1.9: pkg/kubelet/apis/cri/v1alpha1/runtime
- v1.6: pkg/kubelet/api/v1alpha1/runtime

Kubelet 作为 CRI 的客户端，而容器运行时则需要实现 CRI 的服务端（即 gRPC server，通常称为 CRI shim）。容器运行时在启动 gRPC server 时需要监听在本地的 Unix Socket （Windows 使用 tcp 格式）。

开发 CRI 容器运行时

开发新的容器运行时只需要实现 CRI 的 gRPC Server，包括 RuntimeService 和 ImageService。该 gRPC Server 需要监听在本地的 unix socket（Linux 支持 unix socket 格式，Windows 支持 tcp 格式）。

一个简单的示例为

```
import (
    // Import essential packages
    "google.golang.org/grpc"
    runtime "k8s.io/kubernetes/pkg/kubelet/apis/cri/runtime/v1alp
)

// Serivice implements runtime.ImageService and runtime.RuntimeSe
type Service struct {
    ...
}

func main() {
    service := &Service{}
    s := grpc.NewServer(grpc.MaxRecvMsgSize(maxMsgSize),
        grpc.MaxSendMsgSize(maxMsgSize))
    runtime.RegisterRuntimeServiceServer(s, service)
    runtime.RegisterImageServiceServer(s, service)
    lis, err := net.Listen("unix", "/var/run/runtime.sock")
    if err != nil {
        logrus.Fatalf("Failed to create listener: %v", err)
    }
    go s.Serve(lis)

    // Other codes
}
```

对于 Streaming API（Exec、PortForward 和 Attach），CRI 要求容器运行时返回一个 streaming server 的 URL 以便 Kubelet 重定向 API Server 发送过来的请求。在 v1.10 及更早版本中，容器运行时必需返回一个 API Server 可直接访问的 URL（通常跟 Kubelet 使用相同的监听地址）；而从 v1.11 开始，Kubelet 新增了 `--redirect-container-streaming`（默认为 false），默认不再转发而是代理 Streaming 请求，这样运行时可以返回一个 localhost 的 URL（当然也不再需要配置 TLS）。

详细的实现方法可以参考 dockershim 或者 cri-o。

Kubelet 配置

在启动 kubelet 时传入容器运行时监听的 Unix Socket 文件路径，比如

```
kubelet --container-runtime=remote --container-runtime-endpoint=u
```

容器运行时

目前基于 CRI 容器引擎已经比较丰富了，包括

- Docker：核心代码依然保留在 kubelet 内部（pkg/kubelet/dockershim），是最稳定和特性支持最好的运行时
- OCI 容器运行时：
  - 社区有两个实现
    - Containerd，支持 kubernetes v1.7+
    - CRI-O，支持 Kubernetes v1.6+
  - 支持的 OCI 容器引擎包括
    - runc：OCI 标准容器引擎
    - gVisor：谷歌开源的基于用户空间内核的沙箱容器引擎
    - Clear Containers：Intel 开源的基于虚拟化的容器引擎
    - Kata Containers：基于虚拟化的容器引擎，由 Clear Containers 和 runV 合并而来
- PouchContainer：阿里巴巴开源的胖容器引擎
- Frakti：支持 Kubernetes v1.6+，提供基于 hypervisor 和 docker 的混合运行时，适用于运行非可信应用，如多租户和 NFV 等场景
- Rktlet：支持 rkt 容器引擎（rknetes 代码已在 v1.10 中弃用）
- Virtlet：Mirantis 开源的虚拟机容器引擎，直接管理 libvirt 虚拟机，镜像须是 qcow2 格式
- Infranetes：直接管理 IaaS 平台虚拟机，如 GCE、AWS 等

Containerd

以 Containerd 为例，在 1.0 及以前版本将 dockershim 和 docker daemon 替换为 cri-containerd + containerd，而在 1.1 版本直接将 cri-containerd 内置在 Containerd 中，简化为一个 CRI 插件。

Containerd 内置的 CRI 插件实现了 Kubelet CRI 接口中的 Image Service 和 Runtime Service，通过内部接口管理容器和镜像，并通过 CNI 插件给 Pod 配置网络。

RuntimeClass

RuntimeClass 是 v1.12 引入的新 API 对象，用来支持多容器运行时，比如

- Kata Containers/gVisor + runc
- Windows Process isolation + Hyper-V isolation containers

RuntimeClass 表示一个运行时对象，在使用前需要开启特性开关 `RuntimeClass`，并创建 RuntimeClass CRD：

```
kubectl apply -f https://github.com/kubernetes/kubernetes/tree/ma
```

然后就可以定义 RuntimeClass 对象

```
apiVersion: node.k8s.io/v1alpha1  # RuntimeClass is defined in th
kind: RuntimeClass
metadata:
  name: myclass  # The name the RuntimeClass will be referenced b
  # RuntimeClass is a non-namespaced resource
spec:
  runtimeHandler: myconfiguration  # The name of the correspondir
```

而在 Pod 中定义使用哪个 RuntimeClass：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: myclass
  # ...
```

参考文档

- Runtime Class Documentation

- Sandbox Isolation Level Decision

CRI-tools

通常，容器引擎会提供一个命令行工具来帮助用户调试容器应用并简化故障排错。比如使用 Docker 作为容器运行时的时候，可以使用 `docker` 命令来查看容器和镜像的状态，并验证容器的配置是否正确。但在使用其他容器引擎时，推荐使用 `crictl` 来替代 `docker` 工具。

`crictl` 是 cri-tools 的一部分，它提供了类似于 docker 的命令行工具，不需要通过 Kubelet 就可以通过 CRI 跟容器运行时通信。它是专门为 Kubernetes 设计的，提供了Pod、容器和镜像等资源的管理命令，可以帮助用户和开发者调试容器应用或者排查异常问题。`crictl` 可以用于所有实现了 CRI 接口的容器运行时。

注意，`crictl` 并非 `kubectl` 的替代品，它只通过 CRI 接口与容器运行时通信，可以用来调试和排错，但并不用于运行容器。虽然 `crictl` 也提供运行 Pod 和容器的子命令，但这些命令仅推荐用于调试。需要注意的是，如果是在 Kubernetes Node 上面创建了新的 Pod，那么它们会被 Kubelet 停止并删除。

除了 `crictl`，cri-tools 还提供了用于验证容器运行时是否实现 CRI 需要功能的验证测试工具 `critest`。`critest` 通过运行一系列的测试验证容器运行时在实现 CRI 时是否与 Kubelet 的需求一致，推荐所有的容器运行时在发布前都要通过其测试。一般情况下，`critest` 可以作为容器运行时集成测试的一部分，用以保证代码更新不会破坏 CRI 功能。

cri-tools 已在 v1.11 版 GA，详细使用方法请参考 kubernetes-sigs/cri-tools 和 Debugging Kubernetes nodes with crictl。

crictl 示例

查询 Pod

```
$ crictl pods --name nginx-65899c769f-wv2gp
POD ID              CREATED             STATE               NAME
4dccb216c4adb       2 minutes ago       Ready               nginx
```

Pod 列表

```
$ crictl pods
POD ID              CREATED              STATE               NAME
926f1b5a1d33a       About a minute ago   Ready               sh-8
4dccb216c4adb       About a minute ago   Ready               ngir
a86316e96fa89       17 hours ago         Ready               kube
919630b8f81f1       17 hours ago         Ready               nvid
```

镜像列表

```
$ crictl images
IMAGE                                     TAG                 IMA
busybox                                   latest              8c8
k8s-gcrio.azureedge.net/hyperkube-amd64   v1.10.3             e17
k8s-gcrio.azureedge.net/pause-amd64       3.1                 da8
nginx                                     latest              cd5
```

容器列表

```
$ crictl ps -a
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493b
9c5951df22c78       busybox@sha256:141c253bc4c3fd0a201d32dc1f493b
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha25
```

容器内执行命令

```
$ crictl exec -i -t 1f73f2d81bf98 ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

容器日志

```
crictl logs 87d3992f84f74
10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200
10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200
```

参考文档

- Debugging Kubernetes nodes with crictl
- https://github.com/kubernetes-sigs/cri-tools

Frakti

简介

Frakti是一个基于Kubelet CRI的运行时，它提供了hypervisor级别的隔离性，特别适用于运行不可信应用以及多租户场景下。Frakti实现了一个混合运行时：

- 特权容器以Docker container的方式运行
- 而普通容器则以hyper container的方法运行在VM内

Allinone安装方法

Frakti提供了一个简便的安装脚本，可以一键在Ubuntu或CentOS上启动一个本机的Kubernetes+frakti集群。

```
curl -sSL https://github.com/kubernetes/frakti/raw/master/cluster
```

集群部署

首先需要在所有机器上安装hyperd，docker，frakti，CNI 和 kubelet。

安装hyperd

Ubuntu 16.04+:

```
apt-get update && apt-get install -y qemu libvirt-bin
curl -sSL https://hypercontainer.io/install | bash
```

CentOS 7:

```
curl -sSL https://hypercontainer.io/install | bash
```

配置hyperd:

```
echo -e "Kernel=/var/lib/hyper/kernel\n\
Initrd=/var/lib/hyper/hyper-initrd.img\n\
Hypervisor=qemu\n\
StorageDriver=overlay\n\
gRPCHost=127.0.0.1:22318" > /etc/hyper/config
systemctl enable hyperd
systemctl restart hyperd
```

安装docker

Ubuntu 16.04+:

```
apt-get update
apt-get install -y docker.io
```

CentOS 7:

```
yum install -y docker
```

启动docker:

```
systemctl enable docker
systemctl start docker
```

安装frakti

```
curl -sSL https://github.com/kubernetes/frakti/releases/download/
chmod +x /usr/bin/frakti
cgroup_driver=$(docker info | awk '/Cgroup Driver/{print $3}')
cat < /lib/systemd/system/frakti.service
[Unit]
Description=Hypervisor-based container runtime for Kubernetes
Documentation=https://github.com/kubernetes/frakti
After=network.target

[Service]
ExecStart=/usr/bin/frakti --v=3 \
          --log-dir=/var/log/frakti \
          --logtostderr=false \
          --cgroup-driver=${cgroup_driver} \
          --listen=/var/run/frakti.sock \
          --streaming-server-addr=%H \
          --hyper-endpoint=127.0.0.1:22318
MountFlags=shared
TasksMax=8192
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
TimeoutStartSec=0
Restart=on-abnormal

[Install]
WantedBy=multi-user.target
EOF
```

安装CNI

Ubuntu 16.04+:

```
apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | a
cat < /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubernetes-cni
```

CentOS 7:

```
cat < /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.
EOF
setenforce 0
yum install -y kubernetes-cni
```

配置CNI网络，注意

- frakti目前仅支持bridge插件
- 所有机器上Pod的子网不能相同，比如master上可以用 `10.244.1.0/24` ，而第一个Node上可以用 `10.244.2.0/24`

```
mkdir -p /etc/cni/net.d
cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.1.0/24",
        "routes": [
            { "dst": "0.0.0.0/0"  }
        ]
    }
}
EOF
cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
    "cniVersion": "0.3.0",
    "type": "loopback"
}
EOF
```

安装Kubelet

Ubuntu 16.04+:

```
apt-get install -y kubelet kubeadm kubectl
```

CentOS 7:

```
yum install -y kubelet kubeadm kubectl
```

配置Kubelet使用frakti runtime:

```
sed -i '2 i\Environment="KUBELET_EXTRA_ARGS=--container-runtime=r
systemctl daemon-reload
```

配置Master

```
kubeadm init kubeadm init --pod-network-cidr 10.244.0.0/16 --kube

# Optional: enable schedule pods on the master
export KUBECONFIG=/etc/kubernetes/admin.conf
kubectl taint nodes --all node-role.kubernetes.io/master:NoSchedu
```

配置Node

```
# get token on master node
token=$(kubeadm token list | grep authentication,signing | awk '{

# join master on worker nodes
kubeadm join --token $token ${master_ip}
```

配置CNI网络路由

在集群模式下，需要为容器网络配置直接路由，假设有一台master和两台Node：

```
NODE   IP_ADDRESS   CONTAINER_CIDR
master 10.140.0.1  10.244.1.0/24
node-1 10.140.0.2  10.244.2.0/24
node-2 10.140.0.3  10.244.3.0/24
```

CNI的网络路由可以这么配置：

```
# on master
ip route add 10.244.2.0/24 via 10.140.0.2
ip route add 10.244.3.0/24 via 10.140.0.3

# on node-1
ip route add 10.244.1.0/24 via 10.140.0.1
ip route add 10.244.3.0/24 via 10.140.0.3

# on node-2
ip route add 10.244.1.0/24 via 10.140.0.1
ip route add 10.244.2.0/24 via 10.140.0.2
```

参考文档

- Frakti部署指南

存储插件

Kubernetes 已经提供丰富的 Volume 和 Persistent Volume 插件，可以根据需要使用这些插件给容器提供持久化存储。

如果内置的这些 Volume 还不满足要求，则可以使用 FlexVolume 或者容器存储接口 CSI 实现自己的 Volume 插件。

容器存储接口CSI

Container Storage Interface (CSI) 是从 v1.9 引入的容器存储接口，用于扩展 Kubernetes 的存储生态。实际上，CSI 是整个容器生态的标准存储接口，同样适用于 Mesos、Cloud Foundry 等其他的容器集群调度系统。

版本信息

Kubernetes	CSI Spec	Status
v1.9	v0.1	Alpha
v1.10	v0.2	Beta
v1.11-v1.12	v0.3	Beta

Sidecar 容器版本

Container Name	CSI spec	Latest Release Tag
csi-provisioner	v0.3	v0.3.1
csi-attacher	v0.3	v0.3.0
driver-registrar	v0.3	v0.3.0

原理

类似于 CRI，CSI 也是基于 gRPC 实现。详细的 CSI SPEC 可以参考 这里，它要求插件开发者要实现三个 gRPC 服务：

- **Identity Service**：用于 Kubernetes 与 CSI 插件协调版本信息
- **Controller Service**：用于创建、删除以及管理 Volume 存储卷
- **Node Service**：用于将 Volume 存储卷挂载到指定的目录中以便 Kubelet 创建容器时使用（需要监听在 `/var/lib/kubelet/plugins/[SanitizedCSIDriverName]/csi.sock` ）

由于 CSI 监听在 unix socket 文件上， kube-controller-manager 并不能直接调用 CSI 插件。为了协调 Volume 生命周期的管理，并方便开发者实现 CSI 插件，Kubernetes 提供了几个 sidecar 容器并推荐使用下述方法来部署 CSI 插件：

该部署方法包括：

- StatefuelSet：副本数为 1 保证只有一个实例运行，它包含三个容器
  - 用户实现的 CSI 插件
  - External Attacher：Kubernetes 提供的 sidecar 容器，它监听 *VolumeAttachment* 和 *PersistentVolume* 对象的变化情况，并调用 CSI 插件的 ControllerPublishVolume 和 ControllerUnpublishVolume 等 API 将 Volume 挂载或卸载到指定的 Node 上
  - External Provisioner：Kubernetes 提供的 sidecar 容器，它监听 *PersistentVolumeClaim* 对象的变化情况，并调用 CSI 插件的 *ControllerPublish* 和 *ControllerUnpublish* 等 API 管理 Volume
- Daemonset：将 CSI 插件运行在每个 Node 上，以便 Kubelet 可以调用。它包含 2 个容器
  - 用户实现的 CSI 插件

    - Driver Registrar：注册 CSI 插件到 kubelet 中，并初始化 *NodeId*（即给 Node 对象增加一个 Annotation `csi.volume.kubernetes.io/nodeid`）

配置

- API Server 配置：

```
--allow-privileged=true
--feature-gates=CSIPersistentVolume=true,MountPropagation=true
--runtime-config=storage.k8s.io/v1alpha1=true
```

- Controller-manager 配置：

```
--feature-gates=CSIPersistentVolume=true
```

- Kubelet 配置：

```
--allow-privileged=true
--feature-gates=CSIPersistentVolume=true,MountPropagation=true
```

示例

Kubernetes 提供了几个 CSI 示例，包括 NFS、ISCSI、HostPath、Cinder 以及 FlexAdapter 等。在实现 CSI 插件时，这些示例可以用作参考。

Name	Status	More Information
Cinder	v0.2.0	A Container Storage Interface (CSI) Storage Plug-in for Cinder
DigitalOcean Block Storage	v0.0.1 (alpha)	A Container Storage Interface (CSI) Driver for DigitalOcean Block Storage
AWS Elastic Block Storage	v0.0.1(alpha)	A Container Storage Interface (CSI) Driver for AWS Elastic Block Storage (EBS)
GCE Persistent Disk	Alpha	A Container Storage Interface (CSI) Storage Plugin for Google Compute Engine Persistent Disk
OpenSDS	Beta	For more information, please visit releases and https://github.com/opensds/nbp/tree/master/csi
Portworx	0.2.0	CSI implementation is available here which can be used as an example also.
RBD	v0.2.0	A Container Storage Interface (CSI) Storage RBD Plug-in for Ceph

Name	Status	More Information
CephFS	v0.2.0	A Container Storage Interface (CSI) Storage Plug-in for CephFS
ScaleIO	v0.1.0	A Container Storage Interface (CSI) Storage Plugin for DellEMC ScaleIO
vSphere	v0.1.0	A Container Storage Interface (CSI) Storage Plug-in for VMware vSphere
NetApp	v0.2.0 (alpha)	A Container Storage Interface (CSI) Storage Plug-in for NetApp's Trident container storage orchestrator
Ember CSI	v0.2.0 (alpha)	Multi-vendor CSI plugin supporting over 80 storage drivers to provide block and mount storage to Container Orchestration systems.
Nutanix	beta	A Container Storage Interface (CSI) Storage Driver for Nutanix
Quobyte	v0.2.0	A Container Storage Interface (CSI) Plugin for Quobyte

下面以 NFS 为例来看一下 CSI 插件的使用方法。

首先需要部署 NFS 插件：

```
git clone https://github.com/kubernetes-csi/drivers
cd drivers/pkg/nfs
kubectl create -f deploy/kubernetes
```

然后创建一个使用 NFS 存储卷的容器

```
kubectl create -f examples/kubernetes/nginx.yaml
```

该例中已直接创建 PV 的方式使用 NFS

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data-nfsplugin
  labels:
    name: data-nfsplugin
  annotations:
    csi.volume.kubernetes.io/volume-attributes: '{"server":"10.10
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 100Gi
  csi:
    driver: csi-nfsplugin
    volumeHandle: data-id
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-nfsplugin
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  selector:
    matchExpressions:
    - key: name
      operator: In
      values: ["data-nfsplugin"]
```

也可以用在 StorageClass 中

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-sc-nfsplugin
provisioner: csi-nfsplugin
parameters:

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: request-for-storage
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: csi-sc-nfsplugin
```

参考文档

- Kubernetes CSI Documentation
- CSI Volume Plugins in Kubernetes Design Doc

FlexVolume

FlexVolume 是 Kubernetes v1.8+ 支持的一种存储插件扩展方式。类似于 CNI 插件，它需要外部插件将二进制文件放到预先配置的路径中（如 `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/` ），并需要在系统中安装好所有需要的依赖。

FlexVolume 接口

实现一个 FlexVolume 包括两个步骤

- 实现 FlexVolume 插件接口，包括 `init/attach/detach/waitforattach/isattached/mountdevice/unmountdevice/mount/umount` 等命令（可参考 lvm 示例 和 NFS 示例）
- 将插件放到 `/usr/libexec/kubernetes/kubelet-plugins/volume/exec//` 目录中

FlexVolume 的接口包括

- init：kubelet/kube-controller-manager 初始化存储插件时调用，插件需要返回是否需要要 `attach` 和 `detach` 操作
- attach：将存储卷挂载到 Node 上
- detach：将存储卷从 Node 上卸载
- waitforattach： 等待 attach 操作成功（超时时间为 10 分钟）
- isattached：检查存储卷是否已经挂载
- mountdevice：将设备挂载到指定目录中以便后续 bind mount 使用
- unmountdevice：将设备取消挂载
- mount：将存储卷挂载到指定目录中
- umount：将存储卷取消挂载

而存储驱动在实现这些接口时需要以 JSON 格式返回数据，数据格式为

```
{
  "status": "",
  "message": "",
  "device": "",
  "volumeName": "",
  "attached": "",
    "capabilities":
    {
        "attach": ""
    }
}
```

使用 FlexVolume

在使用 flexVolume 时，需要指定卷的 driver，格式为 `/`，如下面的例子使用了 `kubernetes.io/lvm`

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: test
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: test
    flexVolume:
      driver: "kubernetes.io/lvm"
      fsType: "ext4"
      options:
        volumeID: "vol1"
        size: "1000m"
        volumegroup: "kube_vg"
```

注意：

- 在 v1.7 版本，部署新的 FlevVolume 插件后需要重启 kubelet 和 kube-controller-manager；而从 v1.8 开始不需要重启它们了。
- FlexVolume 不支持 Dynamic Volume Provisioning，以 PVC 方式使用前需要管理员预先创建好对应的 PV。

glusterfs

我们复用 kubernetes 的三台主机做 GlusterFS 存储。

安装 GlusterFS

我们直接在物理机上使用 yum 安装，如果你选择在 kubernetes 上安装，请参考 https://github.com/gluster/gluster-kubernetes/blob/master/docs/setup-guide.md。

```
# 先安装 gluster 源
$ yum install centos-release-gluster -y

# 安装 glusterfs 组件
$ yum install -y glusterfs glusterfs-server glusterfs-fuse gluste

## 创建 glusterfs 目录
$ mkdir /opt/glusterd

## 修改 glusterd 目录
$ sed -i 's/var\/lib/opt/g' /etc/glusterfs/glusterd.vol

# 启动 glusterfs
$ systemctl start glusterd.service

# 设置开机启动
$ systemctl enable glusterd.service

#查看状态
$ systemctl status glusterd.service
```

配置 GlusterFS

```
# 配置 hosts

$ vi /etc/hosts
172.20.0.113   sz-pg-oam-docker-test-001.tendcloud.com
172.20.0.114   sz-pg-oam-docker-test-002.tendcloud.com
172.20.0.115   sz-pg-oam-docker-test-003.tendcloud.com
```

```
# 开放端口
$ iptables -I INPUT -p tcp --dport 24007 -j ACCEPT

# 创建存储目录
$ mkdir /opt/gfs_data
```

```
# 添加节点到 集群
# 执行操作的本机不需要 probe 本机
[root@sz-pg-oam-docker-test-001 ~]#
gluster peer probe sz-pg-oam-docker-test-002.tendcloud.com
gluster peer probe sz-pg-oam-docker-test-003.tendcloud.com

# 查看集群状态
$ gluster peer status
Number of Peers: 2

Hostname: sz-pg-oam-docker-test-002.tendcloud.com
Uuid: f25546cc-2011-457d-ba24-342554b51317
State: Peer in Cluster (Connected)

Hostname: sz-pg-oam-docker-test-003.tendcloud.com
Uuid: 42b6cad1-aa01-46d0-bbba-f7ec6821d66d
State: Peer in Cluster (Connected)
```

配置 volume

GlusterFS 中的 volume 的模式有很多种，包括以下几种：

- **分布卷(默认模式)**：即 DHT，也叫 分布卷：将文件以 hash 算法随机分布到 一台服务器节点中存储。
- **复制模式**：即 AFR，创建 volume 时带 replica x 数量：将文件复制到 replica x 个节点中。
- **条带模式**：即 Striped，创建 volume 时带 stripe x 数量： 将文件切割成数据块，分别存储到 stripe x 个节点中 (类似 raid 0)。
- **分布式条带模式**：最少需要 4 台服务器才能创建。 创建 volume 时 stripe 2 server = 4 个节点： 是 DHT 与 Striped 的组合型。
- **分布式复制模式**：最少需要 4 台服务器才能创建。 创建 volume 时 replica 2 server = 4 个节点：是 DHT 与 AFR 的组合型。
- **条带复制卷模式**：最少需要 4 台服务器才能创建。 创建 volume 时 stripe 2 replica 2 server = 4 个节点： 是 Striped 与 AFR 的组合型。
- **三种模式混合**： 至少需要 8 台 服务器才能创建。 stripe 2 replica 2 ，每 4 个节点 组成一个 组。

这几种模式的示例图参考 GlusterFS Documentation。

因为我们只有三台主机，在此我们使用默认的**分布卷模式。请勿在生产环境上使用该模式，容易导致数据丢失。**

```
# 创建分布卷
$ gluster volume create k8s-volume transport tcp sz-pg-oam-docker

# 查看 volume 状态
$ gluster volume info
Volume Name: k8s-volume
Type: Distribute
Volume ID: 9a3b0710-4565-4eb7-abae-1d5c8ed625ac
Status: Created
Snapshot Count: 0
Number of Bricks: 3
Transport-type: tcp
Bricks:
Brick1: sz-pg-oam-docker-test-001.tendcloud.com:/opt/gfs_data
Brick2: sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data
Brick3: sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data
Options Reconfigured:
transport.address-family: inet
nfs.disable: on

# 启动 分布卷
$ gluster volume start k8s-volume
```

Glusterfs 调优

```
# 开启 指定 volume 的配额
$ gluster volume quota k8s-volume enable

# 限制 指定 volume 的配额
$ gluster volume quota k8s-volume limit-usage / 1TB

# 设置 cache 大小, 默认 32MB
$ gluster volume set k8s-volume performance.cache-size 4GB

# 设置 io 线程, 太大会导致进程崩溃
$ gluster volume set k8s-volume performance.io-thread-count 16

# 设置 网络检测时间, 默认 42s
$ gluster volume set k8s-volume network.ping-timeout 10

# 设置 写缓冲区的大小, 默认 1M
$ gluster volume set k8s-volume performance.write-behind-window-s
```

Kubernetes 中使用 GlusterFS

官方的文档见https://github.com/kubernetes/examples/tree/master/staging/volumes/glusterfs.

以下用到的所有 yaml 和 json 配置文件可以在 glusterfs 中找到。注意替换其中私有镜像地址为你自己的镜像地址。

kubernetes 安装客户端

```
# 在所有 k8s node 中安装 glusterfs 客户端
$ yum install -y glusterfs glusterfs-fuse

# 配置 hosts
$ vi /etc/hosts
172.20.0.113   sz-pg-oam-docker-test-001.tendcloud.com
172.20.0.114   sz-pg-oam-docker-test-002.tendcloud.com
172.20.0.115   sz-pg-oam-docker-test-003.tendcloud.com
```

因为我们 glusterfs 跟 kubernetes 集群复用主机，因为此这一步可以省去。

配置 endpoints

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernetes

# 修改 endpoints.json ，配置 glusters 集群节点 ip
# 每一个 addresses 为一个 ip 组

    {
      "addresses": [
        {
          "ip": "172.22.0.113"
        }
      ],
      "ports": [
        {
          "port": 1990
        }
      ]
    },

# 导入 glusterfs-endpoints.json

$ kubectl apply -f glusterfs-endpoints.json

# 查看 endpoints 信息
$ kubectl get ep
```

配置 service

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernetes

# service.json 里面查找的是 enpointes 的名称与端口，端口默认配置为 1，我改成

# 导入 glusterfs-service.json
$ kubectl apply -f glusterfs-service.json

# 查看 service 信息
$ kubectl get svc
```

创建测试 pod

```
$ curl -O https://github.com/kubernetes/examples/raw/master/stagi

# 编辑 glusterfs-pod.json
# 修改 volumes  下的 path 为上面创建的 volume 名称

"path": "k8s-volume"

# 导入 glusterfs-pod.json
$ kubectl apply -f glusterfs-pod.json

# 查看 pods 状态
$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   A
glusterfs                         1/1       Running   0          1

# 查看 pods 所在 node
$ kubectl describe pods/glusterfs

# 登陆 node 物理机，使用 df 可查看挂载目录
$ df -h
172.20.0.113:k8s-volume  1.0T     0  1.0T   0% /var/lib/kubelet/p
```

配置 PersistentVolume

PersistentVolume（PV）和 PersistentVolumeClaim（PVC）是 kubernetes 提供的两种 API 资源，用于抽象存储细节。管理员关注于如何通过 pv 提供存储功能而无需关注用户如何使用，同样的用户只需要挂载 PVC 到容器中而不需要关注存储卷采用何种技术实现。PVC 和 PV 的关系跟 pod 和 node 关系类似，前者消耗后者的资源。PVC 可以向 PV 申请指定大小的存储资源并设置访问模式。

PV 属性

- storage 容量
- 读写属性：分别为
  - ReadWriteOnce：单个节点读写；
  - ReadOnlyMany：多节点只读 ；
  - ReadWriteMany：多节点读写

```
$ cat glusterfs-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: gluster-dev-volume
spec:
  capacity:
    storage: 8Gi
  accessModes:
    - ReadWriteMany
  glusterfs:
    endpoints: "glusterfs-cluster"
    path: "k8s-volume"
    readOnly: false

# 导入 PV
$ kubectl apply -f glusterfs-pv.yaml

# 查看 pv
$ kubectl get pv
NAME                CAPACITY   ACCESSMODES   RECLAIMPOLICY   STA
gluster-dev-volume   8Gi        RWX           Retain          Ava
```

PVC 属性

- 访问属性与 PV 相同
- 容量：向 PV 申请的容量 <= PV 总容量

配置 PVC

```
$ cat glusterfs-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: glusterfs-nginx
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 8Gi

# 导入 pvc
$ kubectl apply -f glusterfs-pvc.yaml

# 查看 pvc

$ kubectl get pv
NAME              STATUS    VOLUME               CAPACITY   ACCES
glusterfs-nginx   Bound     gluster-dev-volume   8Gi        RWX
```

创建 nginx deployment 挂载 volume

```
$ vi nginx-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-dm
spec:
  replicas: 2
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          volumeMounts:
            - name: gluster-dev-volume
              mountPath: "/usr/share/nginx/html"
      volumes:
      - name: gluster-dev-volume
        persistentVolumeClaim:
          claimName: glusterfs-nginx

# 导入 deployment
$ kubectl apply -f nginx-deployment.yaml

# 查看 deployment
$ kubectl get pods |grep nginx-dm
nginx-dm-3698525684-g0mvt       1/1       Running   0          6s
nginx-dm-3698525684-hbzq1       1/1       Running   0          6s

# 查看 挂载
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- df -h|grep k8s-vo
172.20.0.113:k8s-volume         1.0T     0  1.0T   0% /usr/share/

# 创建文件 测试
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- touch /usr/share/
```

```
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- ls -lt /usr/share
-rw-r--r-- 1 root root 0 May  4 11:36 /usr/share/nginx/html/index

# 验证 glusterfs
# 因为我们使用分布卷，所以可以看到某个节点中有文件
[root@sz-pg-oam-docker-test-001 ~] ls /opt/gfs_data/
[root@sz-pg-oam-docker-test-002 ~] ls /opt/gfs_data/
index.html
[root@sz-pg-oam-docker-test-003 ~] ls /opt/gfs_data/
```

参考

- CentOS 7 安装 GlusterFS
- https://github.com/gluster/gluster-kubernetes

网络策略

Network Policy 提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。Network Policy 需要网络插件来监测这些策略和 Pod 的变更，并为 Pod 配置流量控制。

如何开发 Network Policy 扩展

实现一个支持 Network Policy 的网络扩展需要至少包含两个组件

- CNI 网络插件：负责给 Pod 配置网络接口
- Policy controller：监听 Network Policy 的变化，并将 Policy 应用到相应的网络接口

支持 Network Policy 的网络插件

- Calico
- Romana
- Weave Net
- Trireme

- OpenContrail

Network Policy 使用方法

具体 Network Policy 的使用方法可以参考 这里。

IngressController

Ingress 为 Kubernetes 集群中的服务提供了外部入口以及路由，而 Ingress Controller 监测 Ingress 和 Service 资源的变更并根据规则配置负载均衡、路由规则和 DNS 等并提供访问入口。

如何开发 Ingress Controller 扩展

NGINX Ingress Controller 和 GLBC 提供了两个 Ingress Controller 的完整示例，可以在此基础上方便的开发新的 Ingress Controller。

常见 Ingress Controller

- Nginx Ingress

```
helm install stable/nginx-ingress --name nginx-ingress --set rbac
```

- HAProxy Ingress controller
- Linkerd
- traefik
- AWS Application Load Balancer Ingress Controller
- kube-ingress-aws-controller
- Voyager: HAProxy Ingress Controller

Ingress 使用方法

具体 Ingress 的使用方法可以参考 这里。

Ingress+Letsencrypt

申请域名

在使用 Let's Encrypt 之前需要申请一个域名，比如可以到 GoDaddy、Name 等网站购买。具体步骤这里不再细说，可以参考网络教程操作。

部署 Nginx Ingress Controller

直接使用 Helm 部署即可：

```
helm install stable/nginx-ingress --name nginx-ingress --set rbac
```

部署成功后，查询 Ingress 服务的公网 IP 地址（下文中假设该 IP 是 `6.6.6.6`）：

```
$ kubectl -n kube-system get service nginx-ingress-controller
NAME                       TYPE           CLUSTER-IP     EXTERNAL
nginx-ingress-controller   LoadBalancer   10.0.216.124   6.6.6.6
```

然后到域名注册服务商网站中，创建 A 记录，将需要的域名解析到 `6.6.6.6`。

开启 Let's Encrypt

```
# Install cert-manager
helm install --namespace=kube-system --name cert-manager stable/c

# create cluster issuer
kubectl apply -f https://raw.githubusercontent.com/feiskyer/kuber
```

创建 Ingress

首先，创建一个 Secret，用于登录认证：

```
$ htpasswd -c auth foo
$ kubectl -n kube-system create secret generic basic-auth --from-
```

HTTP Ingress 示例

为 nginx 服务（端口 80）创建 TLS Ingress，并且自动将 `http://echo-tls.example.com` 重定向到 `https://echo-tls.example.com`：

```
cat <<f-
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web
  namespace: default
  annotations:
    kubernetes.io/tls-acme: "true"
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/ssl-redirect: "true"
    certmanager.k8s.io/cluster-issuer: letsencrypt
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - echo-tls.example.com
    secretName: web-tls
  rules:
  - host: echo-tls.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
EOF
```

TLS Ingress

为 Kubernetes Dashboard 服务（端口443）创建 TLS Ingress，并且禁止该域名的 HTTP 访问：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    kubernetes.io/ingress.allow-http: "false"
    nginx.ingress.kubernetes.io/auth-realm: Authentication Requir
    nginx.ingress.kubernetes.io/auth-secret: basic-auth
    nginx.ingress.kubernetes.io/auth-type: basic
    nginx.ingress.kubernetes.io/secure-backends: "true"
    certmanager.k8s.io/cluster-issuer: letsencrypt
  name: dashboard
  namespace: kube-system
spec:
  tls:
  - hosts:
    - dashboard.example.com
    secretName: dashboard-ingress-tls
  rules:
  - host: dashboard.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kubernetes-dashboard
          servicePort: 443
```

参考文档

- Nginx Ingress Controller Documentation

minikubeIngress

虽然 minikube 支持 LoadBalancer 类型的服务，但它并不会创建外部的负载均衡器，而是为这些服务开放一个 NodePort。这在使用 Ingress 时需要注意。

本节展示如何在 minikube 上开启 Ingress Controller 并创建和管理 Ingress 资源。

启动 Ingress Controller

minikube 已经内置了 ingress addon，只需要开启一下即可

```
$ minikube addons enable ingress
```

稍等一会，nginx-ingress-controller 和 default-http-backend 就会起来

```
$ kubectl get pods -n kube-system
NAME                              READY     STATUS    RESTARTS   A
default-http-backend-5374j        1/1       Running   0          1
kube-addon-manager-minikube       1/1       Running   0          2
kube-dns-268032401-rhrx6          3/3       Running   0          1
kubernetes-dashboard-xh74p        1/1       Running   0          2
nginx-ingress-controller-78mk6    1/1       Running   0          1
```

创建 Ingress

首先启用一个 echo server 服务

```
$ kubectl run echoserver --image=gcr.io/google_containers/echoser
$ kubectl expose deployment echoserver --type=NodePort
$ minikube service echoserver --url
http://192.168.64.36:31957
```

然后创建一个 Ingress，将 `http://mini-echo.io` 和 `http://mini-web.io/echo` 转发到刚才创建的 echoserver 服务上

```
$ cat <<f -
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: echo
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  backend:
    serviceName: default-http-backend
    servicePort: 80
  rules:
  - host: mini-echo.io
    http:
      paths:
      - path: /
        backend:
          serviceName: echoserver
          servicePort: 8080
  - host: mini-web.io
    http:
      paths:
      - path: /echo
        backend:
          serviceName: echoserver
          servicePort: 8080
EOF
```

为了访问 `mini-echo.io` 和 `mini-web.io` 这两个域名，手动在 hosts 中增加一个映射

```
$ echo "$(minikube ip) mini-echo.io mini-web.io" | sudo tee -a /e
```

然后，就可以通过 `http://mini-echo.io` 和 `http://mini-web.io/echo` 来访问服务了。

使用 xip.io

前面的方法需要每次在使用不同域名时手动配置 hosts，借助 `xip.io` 可以省掉这个步骤。

跟前面类似，先启动一个 nginx 服务

```
$ kubectl run nginx --image=nginx --port=80
$ kubectl expose deployment nginx --type=NodePort
```

然后创建 Ingress，与前面不同的是 host 使用 `nginx.$(minikube ip).xip.io` ：

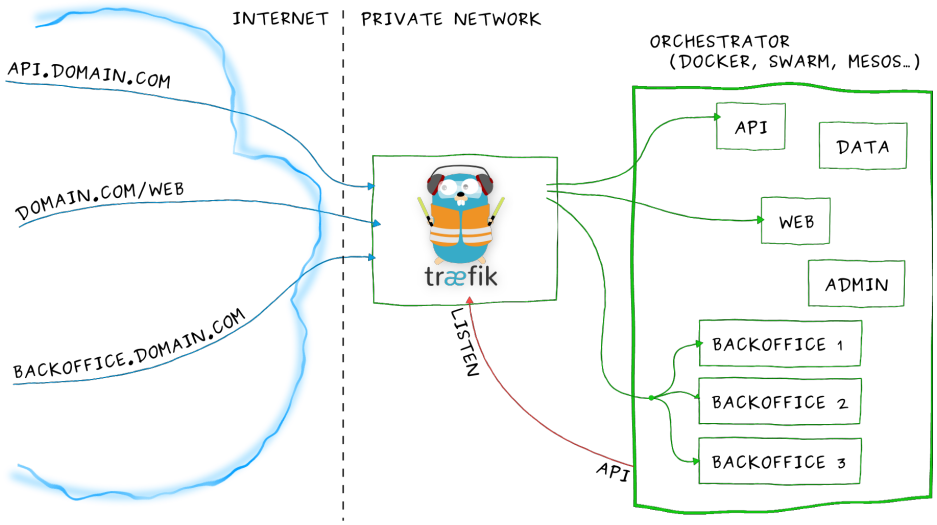
```
$ cat <<f -
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: my-nginx-ingress
spec:
 rules:
  - host: nginx.$(minikube ip).xip.io
    http:
     paths:
      - path: /
        backend:
         serviceName: nginx
         servicePort: 80
EOF
```

然后就可以直接访问该域名了

```
$ curl nginx.$(minikube ip).xip.io
```

TraefikIngress

Traefik 是一款开源的反向代理与负载均衡工具，它监听后端的变化并自动更新服务配置。Traefik 最大的优点是能够与常见的微服务系统直接整合，可以实现自动化动态配置。目前支持 Docker、Swarm,Marathon、Mesos、Kubernetes、Consul、Etcd、Zookeeper、BoltDB 和 Rest API 等后端模型。



主要功能包括

- Golang编写，部署容易
- 快（nginx的85%）
- 支持众多的后端（Docker，Swarm，Kubernetes，Marathon，Mesos，Consul，Etcd等）
- 内置Web UI、Metrics和Let’s Encrypt支持，管理方便
- 自动动态配置
- 集群模式高可用
- 支持 Proxy Protocol

Ingress简介

简单的说，ingress就是从kubernetes集群外访问集群的入口，将用户的URL请求转发到不同的service上。Ingress相当于nginx、apache等负载均衡反向代理服务器，其中还包括规则定义，即URL的路由信息，路由信息的刷新由 Ingress controller 来提供。

Ingress Controller 实质上可以理解为是个监视器，Ingress Controller 通过不断地跟 kubernetes API 打交道，实时的感知后端 service、pod 等变化，比如新增和减少 pod，service 增加与减少等；当得到这些变化信息后，Ingress Controller 再结合下文的 Ingress 生成配置，然后更新反向代理负载均衡器，并刷新其配置，达到服务发现的作用。

Helm 部署 Traefik

```
# Setup domain, user and password first.
$ export USER=user
$ export DOMAIN=ingress.feisky.xyz
$ htpasswd -c auth $USER
New password:
Re-type new password:
Adding password for user user
$ PASSWORD=$(cat auth| awk -F: '{print $2}')

# Deploy with helm.
helm install stable/traefik --name --namespace kube-system --set
```

稍等一会，traefik Pod 就会运行起来：

```
$ kubectl -n kube-system get pod -l app=traefik
NAME                       READY     STATUS    RESTARTS   AGE
traefik-65d8dc4489-k97cg   1/1       Running   0          5m

$ kubectl -n kube-system get ingress
NAME                HOSTS                    ADDRESS   PORTS     A
traefik-dashboard   ui.ingress.feisky.xyz              80        2

$ kubectl -n kube-system get svc traefik
NAME      TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
traefik   LoadBalancer   10.0.206.26   172.20.0.115    80:31662/T
```

通过配置 DNS 解析（CNAME 记录域名到 Ingress Controller 服务的外网IP）、修改 `/etc/hosts` 添加域名映射（见下述测试部分）或者使用 `xip.io` （参考 minikube ingress 使用方法）等方法，就可以通过配置的域名直接访问所需服务了。比如上述的 Dashboard 服务可以通过域名 `ui.ingress.feisky.xyz` 来访问：

kubernetes-dashboard

上图中，左侧黄色部分部分列出的是所有的rule，右侧绿色部分是所有的backend。

Ingress 示例

下面来看一个更复杂的示例。创**建名为 `traefik-ingress` 的 ingress**，文件名traefik.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: traefik.nginx.io
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
  - host: traefik.frontend.io
    http:
      paths:
      - path: /
        backend:
          serviceName: frontend
          servicePort: 80
```

其中，

- `backend` 中要配置 `default namespace` 中启动的 `service` 名字
- `path` 就是URL地址后的路径，如 `traefik.frontend.io/path`
- `host` 最好使用 `service-name.filed1.filed2.domain-name` 这种类似主机名称的命名方式，方便区分服务

根据你自己环境中部署的 `service` 名称和端口自行修改，有新 `service` 增加时，修改该文件后可以使用 `kubectl replace -f traefik.yaml` 来更新。

测试

在集群的任意一个节点上执行。假如现在我要访问nginx的"/"路径。

```
$ curl -H Host:traefik.nginx.io http://172.20.0.115/
```

```
# Welcome to nginx!

If you see this page, the nginx web server is successfully instal
working. Further configuration is required.

For online documentation and support please refer to
"http://nginx.org/">nginx.org.

Commercial support is available at
"http://nginx.com/">nginx.com.

*Thank you for using nginx.*
```

如果你需要在 kubernetes 集群以外访问就需要设置 DNS，或者修改本机的hosts文件在其中加入：

```
172.20.0.115 traefik.nginx.io
172.20.0.115 traefik.frontend.io
```

所有访问这些地址的流量都会发送给 `172.20.0.115` 这台主机，就是我们启动traefik的主机。Traefik会解析http请求header里的Host参数将流量转发给Ingress配置里的相应service。

traefik-nginx

traefik-guestbook

参考文档

- Traefik-kubernetes 初试
- Traefik简介
- Guestbook example

Keepalived-VIP

Kubernetes 使用 keepalived 来产生虚拟 IP address

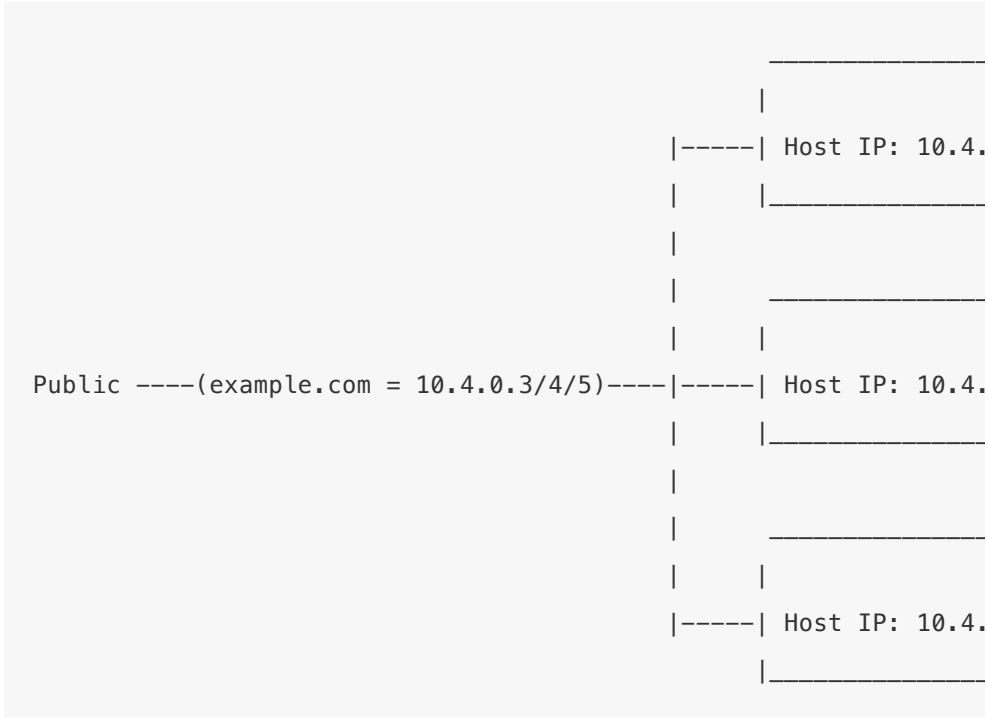
我们将探讨如何利用 IPVS - The Linux Virtual Server Project" 来 kubernetes 配置 VIP

前言

kubernetes v1.6 版提供了三种方式去暴露 Service：

1. **L4 的 LoadBalacncer** ： 只能在 cloud providers 上被使用 像是 GCE 或 AWS
2. **NodePort** ： NodePort 允许在每个节点上开启一个 port 口，借由这个 port 口会再将请求导向到随机的 pod 上
3. **L7 Ingress** ：Ingress 为一个 LoadBalancer(例: nginx, HAProxy, traefik, vulcand) 会将 HTTP/HTTPS 的各个请求导向到相对应的 service endpoint

有了这些方式，为何我们还需要 *keepalived* ？



我们假设 Ingress 运行在 3 个 kubernetes 节点上，并对外暴露 `10.4.0.x` 的 IP 去做 loadbalance

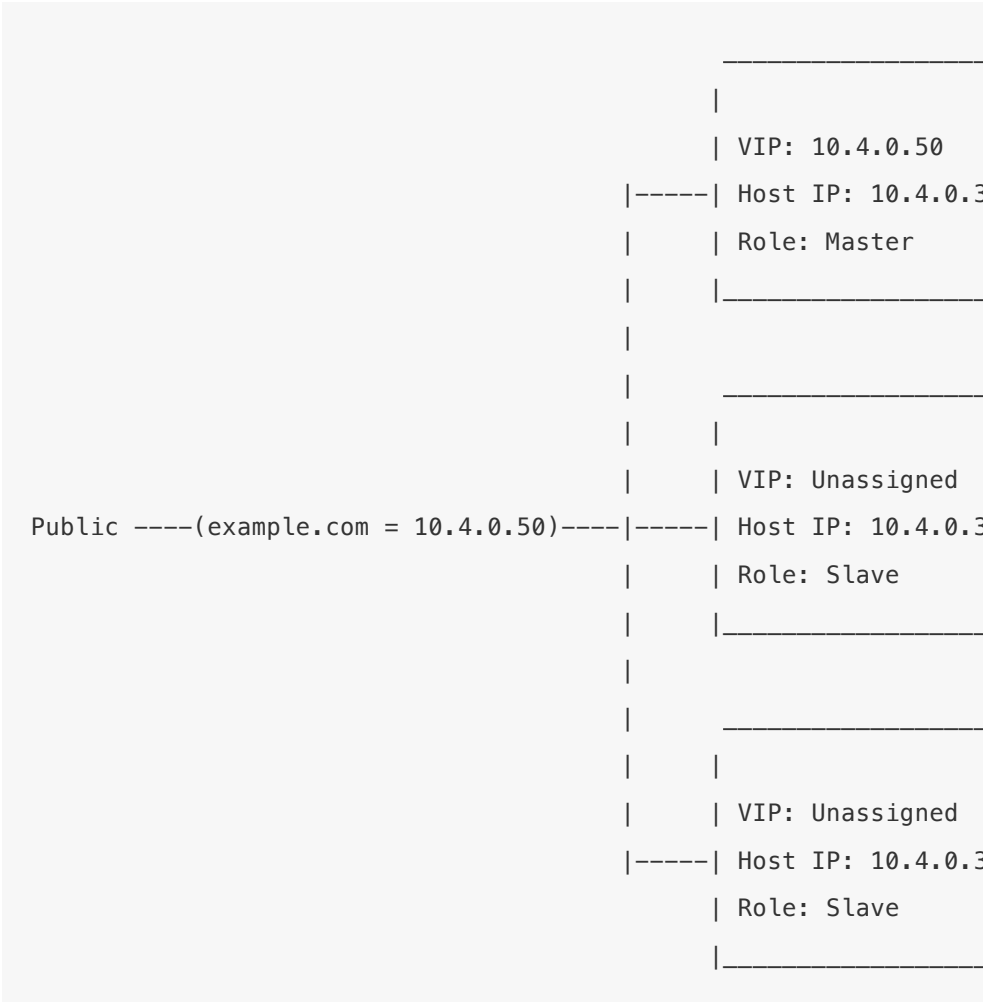
DNS Round Robin (RR) 将对应到 `example.com` 的请求轮循给这 3 个节点，如果 `10.4.0.3` 掛了，仍有三分之一的流量会导向 `10.4.0.3`，这样就会有一段 downtime，直到 DNS 发现 `10.4.0.3` 掛了并修正导向

严格来说，这并没有真正的做到 High Availability (HA)

这边 IPVS 可以帮助我们解决这件事，这个想法是虚拟 IP(VIP) 对应到每个 service 上，并将 VIP 暴露到 kubernetes 群集之外

与 service-loadbalancer 或 ingress-nginx 的区别

我们看到以下的图



我们可以看到只有一个 node 被选为 Master(透过 VRRP 选择的)，而我们的 VIP 是 `10.4.0.50`，如果 `10.4.0.3` 掛掉了，那会从剩余的节点中选一个成为 Master 并接手 VIP，这样我们就可以确保落实真正的 HA

环境需求

只需要确认要运行 keepalived-vip 的 kubernetes 群集 DaemonSets 功能是正常的就行了

RBAC

由于 kubernetes 在 1.6 后引进了 RBAC 的概念，所以我们要先去设定 rule，至於有关 RBAC 的详情请至 说明。

vip-rbac.yaml

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: kube-keepalived-vip
rules:
- apiGroups: [""]
  resources:
  - pods
  - nodes
  - endpoints
  - services
  - configmaps
  verbs: ["get", "list", "watch"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-keepalived-vip
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: kube-keepalived-vip
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kube-keepalived-vip
subjects:
- kind: ServiceAccount
  name: kube-keepalived-vip
  namespace: default
```

clusterrolebinding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1alpha1
kind: ClusterRoleBinding
metadata:
  name: kube-keepalived-vip
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kube-keepalived-vip
subjects:
  - kind: ServiceAccount
    name: kube-keepalived-vip
    namespace: default
```

```
$ kubectl create -f vip-rbac.yaml
$ kubectl create -f clusterrolebinding.yaml
```

示例

先建立一个简单的 service

nginx-deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30302
    targetPort: 80
    protocol: TCP
    name: http
  selector:
    app: nginx
```

主要功能就是 pod 去监听听 80 port，再开启 service NodePort 监听 30320

```
$ kubecrl create -f nginx-deployment.yaml
```

接下来我们要做的是 config map

```
$ echo "apiVersion: v1
kind: ConfigMap
metadata:
  name: vip-configmap
data:
  10.87.2.50: default/nginx" | kubectl create -f -
```

注意，这边的 `10.87.2.50` 必须换成你自己同网段下无使用的 IP e.g. 10.87.2.X 后面 `nginx` 为 service 的 name，这边可以自行更换

接着确认一下

```
$kubectl get configmap
NAME            DATA      AGE
vip-configmap   1         23h
```

再来就是设置 keepalived-vip

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: kube-keepalived-vip
spec:
  template:
    metadata:
      labels:
        name: kube-keepalived-vip
    spec:
      hostNetwork: true
      containers:
        - image: gcr.io/google_containers/kube-keepalived-vip:0.9
          name: kube-keepalived-vip
          imagePullPolicy: Always
          securityContext:
            privileged: true
          volumeMounts:
            - mountPath: /lib/modules
              name: modules
              readOnly: true
            - mountPath: /dev
              name: dev
          # use downward API
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          # to use unicast
          args:
          - --services-configmap=default/vip-configmap
          # unicast uses the ip of the nodes instead of multicast
          # this is useful if running in cloud providers (like AW
          #- --use-unicast=true
```

```
      volumes:
        - name: modules
          hostPath:
            path: /lib/modules
        - name: dev
          hostPath:
            path: /dev
```

建立 daemonset

```
$ kubectl get daemonset kube-keepalived-vip
NAME                  DESIRED   CURRENT   READY     UP-TO-DATE
kube-keepalived-vip   5         5         5         5
```

检查一下配置状态

```
kubectl get pod -o wide |grep keepalive
kube-keepalived-vip-c4sxw         1/1       Running            0
kube-keepalived-vip-c9p7n         1/1       Running            0
kube-keepalived-vip-psdp9         1/1       Running            0
kube-keepalived-vip-xfmxg         1/1       Running            0
kube-keepalived-vip-zjts7         1/1       Running            3
```

可以随机挑一个 pod，去看里面的配置

```
 $ kubectl exec kube-keepalived-vip-c4sxw cat /etc/keepalived/kee


global_defs {
  vrrp_version 3
  vrrp_iptables KUBE-KEEPALIVED-VIP
}

vrrp_instance vips {
  state BACKUP
  interface eno1
  virtual_router_id 50
  priority 103
  nopreempt
  advert_int 1

  track_interface {
    eno1
  }



  virtual_ipaddress {
    10.87.2.50
  }
}


# Service: default/nginx
virtual_server 10.87.2.50 80 { // 此为 service 开的口
  delay_loop 5
  lvs_sched wlc
  lvs_method NAT
  persistence_timeout 1800
  protocol TCP


  real_server 10.2.49.30 8080 { // 这里说明 pod 的真实状况
    weight 1
    TCP_CHECK {
```

```
      connect_port 80
      connect_timeout 3
    }
  }

}
```

最后我们去测试这功能

```
$ curl  10.87.2.50
```

Welcome to nginx!

If you see this page, the nginx web server is successfully instal
working. Further configuration is required.

For online documentation and support please refer to
"http://nginx.org/">nginx.org.

Commercial support is available at
"http://nginx.com/">nginx.com.

Thank you for using nginx.

10.87.2.50:80(我们假设的 VIP，实际上其实没有 node 是用这 IP)即可帮我们导向这个 service

以上的程式代码都在 github 上可以找到。

参考文档

- kweisamx/kubernetes-keepalived-vip
- kubernetes/keepalived-vip

CloudProvider扩展

当 Kubernetes 集群运行在云平台内部时，Cloud Provider 使得 Kubernetes 可以直接利用云平台实现持久化卷、负载均衡、网络路由、DNS 解析以及横向扩展等功能。

常见 Cloud Provider

Kubenretes 内置的 Cloud Provider 包括

- GCE
- AWS
- Azure
- Mesos
- OpenStack
- CloudStack
- Ovirt
- Photon
- Rackspace
- Vsphere

当前 Cloud Provider 工作原理

- apiserver, kubelet, controller-manager 都配置 cloud provider 选项
- Kubelet
  - 通过 Cloud Provider 接口查询 nodename
  - 向 API Server 注册 Node 时查询 InstanceID、ProviderID、ExternalID 和 Zone 等信息
  - 定期查询 Node 是否新增了 IP 地址
  - 设置无法调度的条件（condition），直到云服务商的路由配置完成

- kube-apiserver
  - 向所有 Node 分发 SSH 密钥以便建立 SSH 隧道
  - PersistentVolumeLabel 负责 PV 标签
  - PersistentVolumeClainResize 动态扩展 PV 的大小
- kube-controller-manager
  - Node 控制器检查 Node 所在 VM 的状态。当 VM 删除后自动从 API Server 中删除该 Node。
  - Volume 控制器向云提供商创建和删除持久化存储卷，并按需要挂载或卸载到指定的 VM 上。
  - Route 控制器给所有已注册的 Nodes 配置云路由。
  - Service 控制器给 LoadBalancer 类型的服务创建负载均衡器并更新服务的外网 IP。

独立 Cloud Provider 工作 原理 以及跟踪进度

- Kubelet 必须配置 `--cloud-provider=external`，并且 `kube-apiserver` 和 `kube-controller-manager` 必须不配置 cloud provider。
- `kube-apiserver` 的准入控制选项不能包含 PersistentVolumeLabel。
- `cloud-controller-manager` 独立运行，并开启 `InitializerConifguration`。
- Kubelet 可以通过 `provider-id` 选项配置 `ExternalID`，启动后会自动给 Node 添加 taint `node.cloudprovider.kubernetes.io/uninitialized=NoSchedule`。
- `cloud-controller-manager` 在收到 Node 注册的事件后再次初始化 Node 配置，添加 zone、类型等信息，并删除上一步 Kubelet 自动创建的 taint。
- 主要逻辑（也就是合并了 kube-apiserver 和 kube-controller-manager 跟云相关的逻辑）
  - Node 控制器检查 Node 所在 VM 的状态。当 VM 删除后自动从 API Server 中删除该 Node。
  - Volume 控制器向云提供商创建和删除持久化存储卷，并按需要挂载或卸载到指定的 VM 上。
  - Route 控制器给所有已注册的 Nodes 配置云路由。
  - Service 控制器给 LoadBalancer 类型的服务创建负载均衡器并更新服务的外网 IP。

    - PersistentVolumeLable 准入控制负责 PV 标签
    - PersistentVolumeClainResize 准入控制动态扩展 PV 大小

如何开发 Cloud Provider 扩展

Kubernetes 的 Cloud Provider 目前正在重构中

- v1.6 添加了独立的 `cloud-controller-manager` 服务，云提供商可以构建自己的 `cloud-controller-manager` 而无须修改 Kubernetes 核心代码
- v1.7-v1.10 进一步重构 `cloud-controller-manager`，解耦了 Controller Manager 与 Cloud Controller 的代码逻辑
- v1.11 External Cloud Provider 升级为 Beta 版

构建一个新的云提供商的 Cloud Provider 步骤为

- 编写实现 cloudprovider.Interface 的 cloudprovider 代码
- 将该 cloudprovider 链接到 `cloud-controller-manager`
    - 在 `cloud-controller-manager` 中导入新的 cloudprovider：`import "pkg/new-cloud-provider"`
    - 初始化时传入新 cloudprovider 的名字，如 `cloudprovider.InitCloudProvider("rancher", s.CloudConfigFile)`
- 配置 kube-controller-manager `--cloud-provider=external`
- 启动 `cloud-controller-manager`

具体实现方法可以参考 rancher-cloud-controller-manager 和 cloud-controller-manager。

Device插件

Kubernetes v1.8 开始增加了 Alpha 版的 Device 插件，用来支持 GPU、FPGA、高性能 NIC、InfiniBand 等各种设备。这样，设备厂商只需要根据 Device Plugin 的接口实现一个特定设备的插件，而不需要修改 Kubernetes 核心代码。

> 在 v1.10 中该特性升级为 Beta 版本。

Device 插件原理

使用 Device 插件之前，首先要开启 DevicePlugins 功能，即配置 `--feature-gates=DevicePlugins=true`（默认是关闭的）。

Device 插件实际上是一个 gPRC 接口，需要实现 `ListAndWatch()` 和 `Allocate()` 等方法，并监听 gRPC Server 的 Unix Socket 在 `/var/lib/kubelet/device-plugins/` 目录中，如 `/var/lib/kubelet/device-plugins/nvidiaGPU.sock`。在实现 Device 插件时需要注意

- 插件启动时，需要通过 `/var/lib/kubelet/device-plugins/kubelet.sock` 向 Kubelet 注册，同时提供插件的 Unix Socket 名称、API 的版本号和插件名称（格式为 `vendor-domain/resource`，如 `nvidia.com/gpu`）。Kubelet 会将这些设备暴露到 Node 状态中，方便后续调度器使用
- 插件启动后向 Kubelet 发送插件列表、按需分配设备并持续监控设备的实时状态
- 插件启动后要持续监控 Kubelet 的状态，并在 Kubelet 重启后重新注册自己。比如，Kubelet 刚启动后会清空 `/var/lib/kubelet/device-plugins/` 目录，所以插件作者可以监控自己监听的 unix socket 是否被删除了，并根据此事件重新注册自己

Device 插件一般推荐使用 DaemonSet 的方式部署，并将 `/var/lib/kubelet/device-plugins` 以 Volume 的形式挂载到容器中。当然，也可以手动运行的方式来部署，但这样就没有失败自动恢复的功能了。

NVIDIA GPU 插件

NVIDIA 提供了一个基于 Device Plugins 接口的 GPU 设备插件 NVIDIA/k8s-device-plugin。

编译

```
git clone https://github.com/NVIDIA/k8s-device-plugin
cd k8s-device-plugin
docker build -t nvidia-device-plugin:1.0.0 .
```

部署

```
kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-dev
```

创建 Pod 时请求 GPU 资源

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  restartPolicy: OnFailure
  containers:
  - image: nvidia/cuda
    name: pod1-ctr
    command: ["sleep"]
    args: ["100000"]

    resources:
      limits:
        nvidia.com/gpu: 1
```

注意：**使用**该**插件**时**需要配置** nvidia-docker 2.0**，并配置** `nvidia` 为**默**认运**行**时 **(即配置 docker daemon 的**选项 `--default-runtime=nvidia` **)**。nvidia-docker 2.0 的安装方法为（以 Ubuntu Xenial 为例，其他系统的安装方法可以参考 这里）：

```
# Configure repository
curl -L https://nvidia.github.io/nvidia-docker/gpgkey | \
sudo apt-key add -
sudo tee /etc/apt/sources.list.d/nvidia-docker.list <<< \
"deb https://nvidia.github.io/libnvidia-container/ubuntu16.04/amd
deb https://nvidia.github.io/nvidia-container-runtime/ubuntu16.04
deb https://nvidia.github.io/nvidia-docker/ubuntu16.04/amd64 /"
sudo apt-get update

# Install nvidia-docker 2.0
sudo apt-get install nvidia-docker2
sudo pkill -SIGHUP dockerd

# Check installation
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

GCP GPU 插件

GCP 也提供了一个 GPU 设备的插件，仅适用于 Google Container Engine，可以访问 GoogleCloudPlatform/container-engine-accelerators 查看。

参考文档

- Device Manager Proposal
- Device Plugins
- NVIDIA device plugin for Kubernetes
- NVIDIA/nvidia-docker 2.0

服务治理

服务治理

本章介绍 Kubernetes 服务治理，包括容器应用管理、Service Mesh 以及 Operator 等。

目前最常用的是手动管理 Manifests，比如 kubernetes github 代码库就提供了很多的 manifest 示例

- https://github.com/kubernetes/kubernetes/tree/master/cluster/addons
- https://github.com/kubernetes/examples
- https://github.com/kubernetes/contrib
- https://github.com/kubernetes/ingress-nginx

手动管理的一个问题就是繁琐，特别是应用复杂并且 Manifest 比较多的时候，还需要考虑他们之间部署关系。Kubernetes 开源社区正在推动更易用的管理方法，如

- 一般准则
- 滚动升级
- Helm
- Operator
- Service Mesh
- Linkerd
- Conduit
- Istio
  - 安装
  - 流量管理
  - 安全管理
  - 策略管理
  - Metrics
  - 排错
  - 社区
- Devops
  - Draft
  - Jenkins X
  - Spinnaker
  - Kompose
  - Skaffold
  - Argo
  - Flux GitOps

一般准则

- 分离构建和运行环境
- 使用 `dumb-int` 等避免僵尸进程
- 不推荐直接使用Pod，而是推荐使用Deployment/DaemonSet等

- 不推荐在容器中使用后台进程，而是推荐将进程前台运行，并使用探针保证服务确实在运行中
- 推荐容器中应用日志打到stdout和stderr，方便日志插件的处理
- 由于容器采用了COW，大量数据写入有可能会有性能问题，推荐将数据写入到Volume中
- 不推荐生产环境镜像使用 `latest` 标签，但开发环境推荐使用并设置 `imagePullPolicy` 为 `Always`
- 推荐使用Readiness探针检测服务是否真正运行起来了
- 使用 `activeDeadlineSeconds` 避免快速失败的Job无限重启
- 引入Sidecar处理代理、请求速率控制和连接控制等问题

分离构建和运行环境

注意分离构建和运行环境，直接通过Dockerfile构建的镜像不仅体积大，包含了很多运行时不必要的包，并且还容易引入安全隐患，如包含了应用的源代码。

可以使用Docker多阶段构建来简化这个步骤。

```
FROM golang:1.7.3 as builder
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go    .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o ap

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /go/src/github.com/alexellis/href-counter/app
CMD ["./app"]
```

僵尸进程和孤儿进程

- 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

- 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。

在容器中，很容易掉进的一个陷阱就是init进程没有正确处理SIGTERM等退出信号。这种情景很容易构造出来，比如

```
# 首先运行一个容器
$ docker run busybox sleep 10000

# 打开另外一个terminal
$ ps uax | grep sleep
sasha    14171  0.0  0.0 139736 17744 pts/18   Sl+  13:25   0:00
root     14221  0.1  0.0   1188     4 ?        Ss   13:25   0:00

# 接着kill掉第一个进程
$ kill 14171
# 现在会发现sleep进程并没有退出
$ ps uax | grep sleep
root     14221  0.0  0.0   1188     4 ?        Ss   13:25   0:00
```

解决方法就是保证容器的init进程可以正确处理SIGTERM等退出信号，比如使用dumb-init

```
$ docker run quay.io/gravitational/debian-tall /usr/bin/dumb-init
```

参考文档

- Kubernetes Production Patterns

滚动升级

当有镜像发布新版本，新版本服务上线时如何实现服务的滚动和平滑升级？

如果你使用 **ReplicationController** 创建的 pod 可以使用 `kubectl rollingupdate` 命令滚动升级，如果使用的是 **Deployment** 创建的 Pod 可以直接修改 yaml 文件后执行 `kubectl apply` 即可。

Deployment 已经内置了 RollingUpdate strategy，因此不用再调用 `kubectl rollingupdate` 命令，升级的过程是先创建新版的 pod 将流量导入到新 pod 上后销毁原来的旧的 pod。

Rolling Update 适用于 `Deployment`、`Replication Controller`，官方推荐使用 Deployment 而不再使用 Replication Controller。

使用 ReplicationController 时的滚动升级请参考官网说明：https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/

ReplicationController 与 Deployment 的关系

ReplicationController 和 Deployment 的 RollingUpdate 命令有些不同，但是实现的机制是一样的，关于这两个 kind 的关系我引用了 ReplicationController 与 Deployment 的区别 中的部分内容如下，详细区别请查看原文。

ReplicationController

Replication Controller 为 Kubernetes 的一个核心内容，应用托管到 Kubernetes 之后，需要保证应用能够持续的运行，Replication Controller 就是这个保证的 key，主要的功能如下：

- 确保 pod 数量：它会确保 Kubernetes 中有指定数量的 Pod 在运行。如果少于指定数量的 pod，Replication Controller 会创建新的，反之则会删除掉多余的以保证 Pod 数量不变。
- 确保 pod 健康：当 pod 不健康，运行出错或者无法提供服务时，Replication Controller 也会杀死不健康的 pod，重新创建新的。
- 弹性伸缩 ：在业务高峰或者低峰期的时候，可以通过 Replication Controller 动态的调整 pod 的数量来提高资源的利用率。同时，配置相应的监控功能（Hroizontal Pod Autoscaler），会定时自动从监控平台获取 Replication Controller 关联 pod 的整体资源使用情况，做到自动伸缩。
- 滚动升级：滚动升级为一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始化升级的时候就可以及时发现和解决问题，避免问题不断扩大。

Deployment

Deployment 同样为 Kubernetes 的一个核心内容，主要职责同样是为了保证 pod 的数量和健康，90% 的功能与 Replication Controller 完全一样，可以看做新一代的 Replication Controller。但是，它又具备了 Replication Controller 之外的新特性：

- Replication Controller 全部功能：Deployment 继承了上面描述的 Replication Controller 全部功能。
- 事件和状态查看：可以查看 Deployment 的升级详细进度和状态。
- 回滚：当升级 pod 镜像或者相关参数的时候发现问题，可以使用回滚操作回滚到上一个稳定的版本或者指定的版本。
- 版本记录：每一次对 Deployment 的操作，都能保存下来，给予后续可能的回滚使用。
- 暂停和启动：对于每一次升级，都能够随时暂停和启动。
- 多种升级方案：Recreate：删除所有已存在的 pod，重新创建新的；RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用 pod 数量，最小升级间隔时间等等。

创建测试镜像

我们来创建一个特别简单的 web 服务，当你访问网页时，将输出一句版本信息。通过区分这句版本信息输出我们就可以断定升级是否完成。

所有配置和代码见 manifests/test/rolling-update-test 目录。

Web 服务的代码 main.go

```
package main

import (
  "fmt"
  "log"
  "net/http"
)

func sayhello(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "This is version 1.") // 这个写入到 w 的是输出到客户端
}

func main() {
  http.HandleFunc("/", sayhello) // 设置访问的路由
  log.Println("This is version 1.")
  err := http.ListenAndServe(":9090", nil) // 设置监听的端口
  if err != nil {
    log.Fatal("ListenAndServe:", err)
  }
}
```

创建 Dockerfile

```
FROM alpine:3.5
ADD hellov2 /
ENTRYPOINT ["/hellov2"]
```

注意修改添加的文件的名称。

创建 Makefile

修改镜像仓库的地址为你自己的私有镜像仓库地址。

修改 `Makefile` 中的 `TAG` 为新的版本号。

```
all: build push clean
.PHONY: build push clean

TAG = v1

# Build for linux amd64
build:
  GOOS=linux GOARCH=amd64 go build -o hello${TAG} main.go
  docker build -t sz-pg-oam-docker-hub-001.tendcloud.com/library/

# Push to tenxcloud
push:
  docker push sz-pg-oam-docker-hub-001.tendcloud.com/library/hell

# Clean
clean:
  rm -f hello${TAG}
```

编译

```
make all
```

分别修改 `main.go` 中的输出语句、`Dockerfile` 中的文件名称和 `Makefile` 中的 `TAG`，创建两个版本的镜像。

测试

我们使用 `Deployment` 部署服务来测试。

配置文件 `rolling-update-test.yaml`：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    name: rolling-update-test
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: rolling-update-test
    spec:
      containers:
      - name: rolling-update-test
        image: sz-pg-oam-docker-hub-001.tendcloud.com/library/hel
        ports:
        - containerPort: 9090
---
apiVersion: v1
kind: Service
metadata:
  name: rolling-update-test
  labels:
    app: rolling-update-test
spec:
  ports:
  - port: 9090
    protocol: TCP
    name: http
  selector:
    app: rolling-update-test
```

部署 service

```
kubectl create -f rolling-update-test.yaml
```

修改 traefik ingress 配置

在 `ingress.yaml` 文件中增加新 service 的配置。

```
  - host: rolling-update-test.traefik.io
    http:
      paths:
      - path: /
        backend:
          serviceName: rolling-update-test
          servicePort: 9090
```

修改本地的 host 配置，增加一条配置：

```
172.20.0.119 rolling-update-test.traefik.io
```

注意：172.20.0.119 是我们之前使用 keepalived 创建的 VIP。

打开浏览器访问 `http://rolling-update-test.traefik.io` 将会看到以下输出：

```
This is version 1.
```

滚动升级

只需要将 `rolling-update-test.yaml` 文件中的 `image` 改成新版本的镜像名，然后执行：

```
kubectl apply -f rolling-update-test.yaml
```

也可以参考 Kubernetes Deployment Concept 中的方法，直接设置新的镜像。

```
kubectl set image deployment/rolling-update-test rolling-update-t
```

或者使用 `kubectl edit deployment/rolling-update-test` 修改镜像名称后保存。

使用以下命令查看升级进度：

```
kubectl rollout status deployment/rolling-update-test
```

升级完成后在浏览器中刷新 `http://rolling-update-test.traefik.io` 将会看到以下输出：

```
This is version 2.
```

说明滚动升级成功。

使用 ReplicationController 创建的 Pod 如何 RollingUpdate

以上讲解使用 **Deployment** 创建的 Pod 的 RollingUpdate 方式，那么如果使用传统的 **ReplicationController** 创建的 Pod 如何 Update 呢？

举个例子：

```
$ kubectl -n spark-cluster rolling-update zeppelin-controller --i
Created zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b
Scaling up zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b f
Scaling zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b up t
Scaling zeppelin-controller down to 0
Update succeeded. Deleting old controller: zeppelin-controller
Renaming zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b to
replicationcontroller "zeppelin-controller" rolling updated
```

只需要指定新的镜像即可，当然你可以配置 RollingUpdate 的策略。

参考

- Rolling update 机制解析
- Running a Stateless Application Using a Deployment
- Simple Rolling Update
- 使用 kubernetes 的 deployment 进行 RollingUpdate

Helm

Helm 是一个类似于 yum/apt/homebrew 的 Kubernetes 应用管理工具。Helm 使用 Chart 来管理 Kubernetes manifest 文件。

Helm 基本使用

安装 `helm` 客户端

```
brew install kubernetes-helm
```

初始化 Helm 并安装 `Tiller` 服务（需要事先配置好 kubectl）

```
helm init
```

更新 charts 列表

```
helm repo update
```

部署服务，比如 mysql

```
➜  ~ helm install stable/mysql
NAME:   quieting-warthog
LAST DEPLOYED: Tue Feb 21 16:13:02 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                    TYPE    DATA  AGE
quieting-warthog-mysql  Opaque  2     1s

==> v1/PersistentVolumeClaim
NAME                    STATUS   VOLUME  CAPACITY  ACCESSMODES  A
quieting-warthog-mysql  Pending  1s

==> v1/Service
NAME                    CLUSTER-IP    EXTERNAL-IP  PORT(S)   AGE
quieting-warthog-mysql  10.3.253.105         3306/TCP  1s

==> extensions/v1beta1/Deployment
NAME                    DESIRED  CURRENT  UP-TO-DATE  AVAILABLE
quieting-warthog-mysql  1        1        1           0



NOTES:
MySQL can be accessed via port 3306 on the following DNS name frc
quieting-warthog-mysql.default.svc.cluster.local

To get your root password run:

    kubectl get secret --namespace default quieting-warthog-mysql

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Ne

2. Install the mysql client:
```

```
    $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:
    $ mysql -h quieting-warthog-mysql -p
```

更多命令的使用方法可以参考下面的 "Helm 命令参考" 部分。

Helm 工作原理

基本概念

Helm 的三个基本概念

- Chart：Helm 应用（package），包括该应用的所有 Kubernetes manifest 模版，类似于 YUM RPM 或 Apt dpkg 文件
- Repository：Helm package 存储仓库
- Release：chart 的部署实例，每个 chart 可以部署一个或多个 release

Helm 工作原理

Helm 包括两个部分，`helm` 客户端和 `tiller` 服务端。

> the client is responsible for managing charts, and the server is responsible for managing releases.

helm 客户端

helm 客户端是一个命令行工具，负责管理 charts、repository 和 release。它通过 gPRC API（使用 `kubectl port-forward` 将 tiller 的端口映射到本地，然后再通过映射后的端口跟 tiller 通信）向 tiller 发送请求，并由 tiller 来管理对应的 Kubernetes 资源。

`helm` 命令的使用方法可以参考下面的 "Helm 命令参考" 部分。

tiller 服务端

tiller 接收来自 helm 客户端的请求，并把相关资源的操作发送到 Kubernetes，负责管理（安装、查询、升级或删除等）和跟踪 Kubernetes 资源。为了方便管理，tiller 把 release 的相关信息保存在 kubernetes 的 ConfigMap 中。

tiller 对外暴露 gRPC API，供 helm 客户端调用。

Helm Charts

Helm 使用 Chart 来管理 Kubernetes manifest 文件。每个 chart 都至少包括

- 应用的基本信息 `Chart.yaml`
- 一个或多个 Kubernetes manifest 文件模版（放置于 templates / 目录中），可以包括 Pod、Deployment、Service 等各种 Kubernetes 资源

Chart.yaml 示例

```
name: The name of the chart (required)
version: A SemVer 2 version (required)
description: A single-sentence description of this project (optic
keywords:
  - A list of keywords about this project (optional)
home: The URL of this project's home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
maintainers: # (optional)
  - name: The maintainer's name (required for each maintainer)
    email: The maintainer's email (optional for each maintainer)
engine: gotpl # The name of the template engine (optional, defaul
icon: A URL to an SVG or PNG image to be used as an icon (optiona
```

依赖管理

Helm 支持两种方式管理依赖的方式：

- 直接把依赖的 package 放在 `charts/` 目录中
- 使用 `requirements.yaml` 并用 `helm dep up foochart` 来自动下载依赖的 packages

```
dependencies:
  - name: apache
    version: 1.2.3
    repository: http://example.com/charts
  - name: mysql
    version: 3.2.1
    repository: http://another.example.com/charts
```

Chart 模版

Chart 模板基于 Go template 和 Sprig，比如

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: deis-database
  namespace: deis
  labels:
    heritage: deis
spec:
  replicas: 1
  selector:
    app: deis-database
  template:
    metadata:
      labels:
        app: deis-database
    spec:
      serviceAccount: deis-database
      containers:
        - name: deis-database
          image: {{.Values.imageRegistry}}/postgres:{{.Values.doc
          imagePullPolicy: {{.Values.pullPolicy}}
          ports:
            - containerPort: 5432
          env:
            - name: DATABASE_STORAGE
              value: {{default "minio" .Values.storage}}
```

模版参数的默认值必须放到 `values.yaml` 文件中，其格式为

```
imageRegistry: "quay.io/deis"
dockerTag: "latest"
pullPolicy: "alwaysPull"
storage: "s3"

# 依赖的 mysql chart 的默认参数
mysql:
  max_connections: 100
  password: "secret"
```

Helm 插件

插件提供了扩展 Helm 核心功能的方法，它在客户端执行，并放在 `$(helm home)/plugins` 目录中。

一个典型的 helm 插件格式为

```
$(helm home)/plugins/
  |- keybase/
      |
      |- plugin.yaml
      |- keybase.sh
```

而 plugin.yaml 格式为

```
name: "keybase"
version: "0.1.0"
usage: "Integreate Keybase.io tools with Helm"
description: |-
  This plugin provides Keybase services to Helm.
ignoreFlags: false
useTunnel: false
command: "$HELM_PLUGIN_DIR/keybase.sh"
```

这样，就可以用 `helm keybase` 命令来使用这个插件。

Helm 命令参考

查询 charts

```
helm search
helm search mysql
```

查询 package 详细信息

```
helm inspect stable/mariadb
```

部署 package

```
helm install stable/mysql
```

部署之前可以自定义 package 的选项：

```
# 查询支持的选项
helm inspect values stable/mysql

# 自定义 password
echo "mysqlRootPassword: passwd" > config.yaml
helm install -f config.yaml stable/mysql
```

另外，还可以通过打包文件（.tgz）或者本地 package 路径（如 path/foo）来部署应用。

查询服务 (Release) 列表

```
➜  ~ helm ls
NAME                REVISION    UPDATED                     STATU
quieting-warthog    1           Tue Feb 21 20:13:02 2017    DEPLO
```

查询服务 (Release) 状态

```
➜  ~ helm status quieting-warthog
LAST DEPLOYED: Tue Feb 21 16:13:02 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME                     TYPE    DATA  AGE
quieting-warthog-mysql  Opaque  2     9m

==> v1/PersistentVolumeClaim
NAME                     STATUS  VOLUME
quieting-warthog-mysql  Bound   pvc-90af9bf9-f80d-11e6-930a-42010

==> v1/Service
NAME                     CLUSTER-IP    EXTERNAL-IP  PORT(S)   AGE
quieting-warthog-mysql  10.3.253.105         3306/TCP  9m

==> extensions/v1beta1/Deployment
NAME                     DESIRED  CURRENT  UP-TO-DATE  AVAILABLE
quieting-warthog-mysql  1        1        1           1


NOTES:
MySQL can be accessed via port 3306 on the following DNS name fro
quieting-warthog-mysql.default.svc.cluster.local

To get your root password run:

    kubectl get secret --namespace default quieting-warthog-mysql

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Ne

2. Install the mysql client:

    $ apt-get update && apt-get install mysql-client -y
```

```
3. Connect using the mysql cli, then provide your password:
    $ mysql -h quieting-warthog-mysql -p
```

升级和回滚 Release

```
# 升级
cat "mariadbUser: user1" >panda.yaml
helm upgrade -f panda.yaml happy-panda stable/mariadb

# 回滚
helm rollback happy-panda 1
```

删除 Release

```
helm delete quieting-warthog
```

repo 管理

```
# 添加 incubator repo
helm repo add incubator https://kubernetes-charts-incubator.stora

# 查询 repo 列表
helm repo list

# 生成 repo 索引（用于搭建 helm repository）
helm repo index
```

chart 管理

```
# 创建一个新的 chart
helm create deis-workflow

# validate chart
helm lint

# 打包 chart 到 tgz
helm package deis-workflow
```

Helm UI

Kubeapps 提供了一个开源的 Helm UI 界面，方便以图形界面的形式管理 Helm 应用。

```
curl -s https://api.github.com/repos/kubeapps/kubeapps/releases/l
sudo mv kubeapps-$(uname -s| tr '[:upper:]' '[:lower:]')-amd64 /u
sudo chmod +x /usr/local/bin/kubeapps

kubeapps up
kubeapps dashboard
```

更多使用方法请参考 Kubeapps 官方网站。

Helm Repository

官方 repository:

- https://hub.helm.sh/
- https://github.com/kubernetes/charts

第三方 repository:

- https://github.com/coreos/prometheus-operator/tree/master/helm
- https://github.com/deis/charts

- https://github.com/bitnami/charts
- https://github.com/att-comdev/openstack-helm
- https://github.com/sapcc/openstack-helm
- https://github.com/helm/charts
- https://github.com/jackzampolin/tick-charts

常用 Helm 插件

1. helm-tiller - Additional commands to work with Tiller
2. Technosophos's Helm Plugins - Plugins for GitHub, Keybase, and GPG
3. helm-template - Debug/render templates client-side
4. Helm Value Store - Plugin for working with Helm deployment values
5. Drone.io Helm Plugin - Run Helm inside of the Drone CI/CD system

Operator

Operator 是 CoreOS 推出的旨在简化复杂有状态应用管理的框架，它是一个感知应用状态的控制器，通过扩展 Kubernetes API 来自动创建、管理和配置应用实例。

Operator 原理

Operator 基于 Third Party Resources 扩展了新的应用资源，并通过控制器来保证应用处于预期状态。比如 etcd operator 通过下面的三个步骤模拟了管理 etcd 集群的行为：

1. 通过 Kubernetes API 观察集群的当前状态；
2. 分析当前状态与期望状态的差别；
3. 调用 etcd 集群管理 API 或 Kubernetes API 消除这些差别。

etcd

如何创建 Operator

Operator 是一个感知应用状态的控制器，所以实现一个 Operator 最关键的就是把管理应用状态的所有操作封装到配置资源和控制器中。通常来说 Operator 需要包括以下功能：

- Operator 自身以 deployment 的方式部署
- Operator 自动创建一个 Third Party Resources 资源类型，用户可以用该类型创建应用实例
- Operator 应该利用 Kubernetes 内置的 Serivce/ReplicaSet 等管理应用
- Operator 应该向后兼容，并且在 Operator 自身退出或删除时不影响应用的状态
- Operator 应该支持应用版本更新
- Operator 应该测试 Pod 失效、配置错误、网络错误等异常情况

如何使用 Operator

为了方便描述，以 Etcd Operator 为例，具体的链接可以参考 -Etcd Operator。

在 Kubernetes 部署 Operator： 通过在 Kubernetes 集群中创建一个 deploymet 实例，来部署对应的 Operator。具体的 Yaml 示例如下：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin
  namespace: default

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  name: admin
subjects:
  - kind: ServiceAccount
    name: admin
    namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: etcd-operator
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: etcd-operator
    spec:
      serviceAccountName: admin
      containers:
      - name: etcd-operator
        image: quay.io/coreos/etcd-operator:v0.4.2
        env:
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

```
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
```

```
# kubectl create -f deployment.yaml
serviceaccount "admin" created
clusterrolebinding "admin" created
deployment "etcd-operator" created

# kubectl  get pod
NAME                              READY     STATUS    RESTARTS   AG
etcd-operator-334633986-3nzk1   1/1       Running   0          31
```

查看 operator 是否部署成功：

```
# kubectl get thirdpartyresources
NAME                       DESCRIPTION             VERSION(S)
cluster.etcd.coreos.com   Managed etcd clusters   v1beta1
```

对应的有状态服务 yaml 文件示例如下：

```
apiVersion: "etcd.coreos.com/v1beta1"
kind: "Cluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.1.8"
```

部署对应的有状态服务：

```
# kubectl create -f example-etcd-cluster.yaml
Cluster "example-etcd-cluster" created

# kubectl get  cluster
NAME                                          KIND
example-etcd-cluster   Cluster.v1beta1.etcd.coreos.com

# kubectl get  service
NAME                           CLUSTER-IP      EXTERNAL-IP   PORT(
example-etcd-cluster           None                          2379/TCP,23
example-etcd-cluster-client    10.105.90.190                 2379/TCP

# kubectl get pod
NAME                             READY     STATUS    RESTARTS   AG
example-etcd-cluster-0002        1/1       Running   0          5h
example-etcd-cluster-0003        1/1       Running   0          4h
example-etcd-cluster-0004        1/1       Running   0          4h
```

其他示例

- Prometheus Operator
- Rook Operator
- Tectonic Operators
- https://github.com/sapcc/kubernetes-operators
- https://github.com/kbst/memcached
- https://github.com/Yolean/kubernetes-kafka
- https://github.com/krallistic/kafka-operator
- https://github.com/huawei-cloudfederation/redis-operator
- https://github.com/upmc-enterprises/elasticsearch-operator
- https://github.com/pires/nats-operator
- https://github.com/rosskukulinski/rethinkdb-operator
- https://istio.io/

与其他工具的关系

- StatefulSets：StatefulSets 为有状态服务提供了 DNS、持久化存储等，而 Operator 可以自动处理服务失效、备份、重配置等复杂的场景。
- Puppet：Puppet 是一个静态配置工具，而 Operator 则可以实时、动态地保证应用处于预期状态
- Helm：Helm 是一个打包工具，可以将多个应用打包到一起部署，而 Operator 则可以认为是 Helm 的补充，用来动态保证这些应用的正常运行

参考资料

- Kubernetes Operators

ServiceMesh

Service Mesh（服务网格）是一个用于保证服务间安全、快速、可靠通信的网络代理组件，是随着微服务和云原生应用兴起而诞生的基础设施层。它通常以轻量级网络代理的方式同应用部署在一起（比如sidecar方式，如下图所示）。Serivce Mesh可以看作是一个位于TCP/IP之上的网络模型，抽象了服务间可靠通信的机制。但与TCP不同，它是面向应用的，为应用提供了统一的可视化和控制。

为了保证服务间通信的可靠性，Service Mesh需要支持熔断机制、延迟感知的负载均衡、服务发现、重试等一些列的特性。比如Linkerd处理一个请求的流程包括

- 查找动态路由确定请求的服务
- 查找该服务的实例
- Linkerd跟响应延迟等因素选择最优的实例
- 将请求转发给最优实例，记录延迟和响应情况
- 如果请求失败或实例实效，则转发给其他实例重试（需要是幂等请求）
- 如果请求超时，则直接失败，避免给后端增加更多的负载
- 记录请求的度量和分布式跟踪情况

为什么Service Mesh是必要的

- 将服务治理与实际服务解耦，避免微服务化过程中对应用的侵入

- 加速传统应用转型微服务或云原生应用

Service Mesh并非一个全新的功能，而是将已存在于众多应用之中的相关功能分离出来，放到统一的组件来管理。特别是在微服务应用中，服务数量庞大，并且可能是基于不同的框架和语言构建，分离出来的Service Mesh组件更容易管理和协调它们。

常见的 Service Mesh 框架包括

- Istio
- Conduit
- Linkerd

Linkerd

Linkerd 是一个面向云原生应用的 Service Mesh 组件，也是 CNCF 项目之一。它为服务间通信提供了一个统一的管理和控制平面，并且解耦了应用程序代码和通信机制，从而无需更改应用程序就可以可视化控制服务间的通信。linkerd 实例是无状态的，可以以每个应用一个实例（sidecar）或者每台 Node 一个实例的方式部署。

Linkerd 的主要特性包括

- 服务发现
- 动态请求路由
- HTTP 代理集成，支持 HTTP、TLS、gRPC、HTTP/2 等
- 感知时延的负载均衡，支持多种负载均衡算法，如 Power of Two Choices (P2C) Least Loaded、Power of Two Choices (P2C) peak ewma、Aperture: least loaded、Heap: least loaded、Round robin 等
- 熔断机制，自动移除不健康的后端实例，包括 fail fast（只要连接失败就移除实例）和 failure accrual（超过 5 个请求处理失败时才将其标记为失效，并保留一定的恢复时间 ）两种
- 分布式跟踪和度量

Linkerd 原理

Linkerd 路由将请求处理分解为多个步骤

- (1) IDENTIFICATION：为实际请求设置逻辑名字（即请求的目的服务），如默认将 HTTP 请求 `GET http://example/hello` 赋值名字 `/svc/example`

- （2）BINDING：dtabs 负责将逻辑名与客户端名字绑定起来，客户端名字总是以 `/#` 或 `/$` 开头，比如

```
# 假设 dtab 为
/env => /#/io.l5d.serversets/discovery
/svc => /env/prod

# 那么服务名 / svc/users 将会绑定为
/svc/users
/env/prod/users
/#/io.l5d.serversets/discovery/prod/users
```

- （3）RESOLUTION：namer 负责解析客户端名，并得到真实的服务地址（IP + 端口）
- （4）LOAD BALANCING：根据负载均衡算法选择如何发送请求

Linkerd 部署

Linkerd 以 DaemonSet 的方式部署在每个 Node 节点上：

```
# Deploy linkerd.
# For CNI, deploy linkerd-cni.yml instead.
# kubectl apply -f https://github.com/linkerd/linkerd-examples/ra
kubectl create ns linkerd
kubectl apply -f https://raw.githubusercontent.com/linkerd/linker

$ kubectl -n linkerd get pod
NAME        READY     STATUS    RESTARTS   AGE
l5d-6v67t   2/2       Running   0          2m
l5d-rn6v4   2/2       Running   0          2m
$ kubectl -n linkerd get svc
NAME      TYPE           CLUSTER-IP   EXTERNAL-IP     POR    AGE
l5d       LoadBalancer   10.0.71.9           4140:32728/TCP,4141:
```

默认情况下，Linkerd 的 Dashboard 监听在每个容器实例的 9990 端口（注意未在 l5d 服务中对外暴露），可以通过服务的相应端口来访问。

```
kubectl -n linkerd port-forward $(kubectl -n linkerd get pod -l a
echo "open http://localhost:9990 in browser"
```

Grafana 和 Prometheus

```
$ kubectl -n linkerd apply -f https://github.com/linkerd/linkerd-
$ kubectl -n linkerd get svc linkerd-viz
NAME          TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
linkerd-viz   LoadBalancer   10.0.235.21        80:30895/TCP,9191
```

TLS

```
kubectl -n linkerd apply -f https://github.com/linkerd/linkerd-ex
kubectl -n linkerd delete ds/l5d configmap/l5d-config
kubectl -n linkerd apply -f https://github.com/linkerd/linkerd-ex
```

Zipkin

```
# Deploy zipkin.
kubectl -n linkerd apply -f https://github.com/linkerd/linkerd-ex

# Deploy linkerd for zipkin.
kubectl -n linkerd apply -f https://github.com/linkerd/linkerd-ex

# Get zipkin endpoint.
ZIPKIN_LB=$(kubectl get svc zipkin -o jsonpath="{.status.loadBala
echo "open http://$ZIPKIN_LB in browser"
```

NAMERD

```
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/link
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/link

$ go get -u github.com/linkerd/namerctl
$ go install github.com/linkerd/namerctl
$ NAMERD_INGRESS_LB=$(kubectl get svc namerd -o jsonpath="{.statu
$ export NAMERCTL_BASE_URL=http://$NAMERD_INGRESS_LB:4180
$ $ namerctl dtab get internal
# version MjgzNjk5NzI=
/srv         => /#/io.l5d.k8s/default/http ;
/host        => /srv ;
/tmp         => /srv ;
/svc         => /host ;
/host/world  => /srv/world-v1 ;
```

Ingress Controller

Linkerd 也可以作为 Kubernetes Ingress Controller 使用，注意下面的步骤将 Linkerd 部署到了 l5d-system namespace。

```
$ kubectl create ns l5d-system
$ kubectl apply -f https://raw.githubusercontent.com/linkerd/link

# If load balancer is supported in kubernetes cluster
$ L5D_SVC_IP=$(kubectl get svc l5d -n l5d-system -o jsonpath="{.s
$ echo open http://$L5D_SVC_IP:9990

# Or else
$ HOST_IP=$(kubectl get po -l app=l5d -n l5d-system -o jsonpath="
$ L5D_SVC_IP=$HOST_IP:$(kubectl get svc l5d -n l5d-system -o 'jso
$ echo open http://$HOST_IP:$(kubectl get svc l5d -n l5d-system -
```

然后通过 `kubernetes.io/ingress.class: "linkerd"` annotation 使用 linkerd ingress 控制器：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world
  annotations:
    kubernetes.io/ingress.class: "linkerd"
spec:
  backend:
    serviceName: world-v1
    servicePort: http
  rules:
  - host: world.v2
    http:
      paths:
      - backend:
          serviceName: world-v2
          servicePort: http
```

更多使用方法见这里。

应用示例

可以通过 HTTP 代理和 linkerd-inject 等两种方式来使用 Linkerd。

HTTP 代理

应用程序在使用 Linkerd 时需要为应用设置 HTTP 代理，其中

- HTTP 使用 `$(NODE_NAME):4140`
- HTTP/2 使用 `$(NODE_NAME):4240`
- gRPC 使用 `$(NODE_NAME):4340`

在 Kubernetes 中，可以使用 Downward API 来获取 `NODE_NAME`，比如

```
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello
spec:
  replicas: 3
  selector:
    app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      dnsPolicy: ClusterFirst
      containers:
      - name: service
        image: buoyantio/helloworld:0.1.6
        env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: http_proxy
          value: $(NODE_NAME):4140
        args:
        - "-addr=:7777"
        - "-text=Hello"
        - "-target=world"
        ports:
        - name: service
          containerPort: 7777
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: hello
spec:
  selector:
    app: hello
  clusterIP: None
  ports:
  - name: http
    port: 7777
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: world-v1
spec:
  replicas: 3
  selector:
    app: world-v1
  template:
    metadata:
      labels:
        app: world-v1
    spec:
      dnsPolicy: ClusterFirst
      containers:
      - name: service
        image: buoyantio/helloworld:0.1.6
        env:
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: TARGET_WORLD
          value: world
        args:
        - "-addr=:7778"
        ports:
        - name: service
          containerPort: 7778
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: world-v1
spec:
  selector:
    app: world-v1
  clusterIP: None
  ports:
  - name: http
    port: 7778
```

linkerd-inject

```
# install linkerd-inject
$ go get github.com/linkerd/linkerd-inject

# inject init container and deploy this config
$ kubectl apply -f <(linkerd-inject -f .yml -linkerdPort 4140)
```

参考文档

- WHAT'S A SERVICE MESH? AND WHY DO I NEED ONE?
- Linkerd 官方文档
- A SERVICE MESH FOR KUBERNETES
- Linkerd examples
- Service Mesh Pattern
- https://conduit.io

Linkerd2

Linkerd2 （曾命名为 Conduit） 是 Buoyant 公司推出的下一代轻量级服务网格框架，开源在 https://github.com/linkerd/linkerd2。与 linkerd 不同的是，它专用于 Kubernetes 集群中，并且比 linkerd

更轻量级（基于 Rust 和 Go，没有了 JVM 等大内存的开销），可以以 sidecar 的方式把代理服务跟实际服务的 Pod 运行在一起（这点跟 Istio 类似）。Linkerd2 的主要特性包括：

- 轻量级，速度快，每个代理容器仅占用 10mb RSS，并且额外延迟只有亚毫妙级
- 安全，基于 Rust，默认开启 TLS
- 端到端可视化
- 增强 Kubernetes 的可靠性、可视性以及安全性

部署

```
$ linkerd install | kubectl apply -f -
namespace/linkerd configured
serviceaccount/linkerd-controller configured
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-controller
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-cont
serviceaccount/linkerd-prometheus configured
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-prometheus
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-prom
service/api configured
service/proxy-api configured
deployment.extensions/controller configured
service/web configured
deployment.extensions/web configured
service/prometheus configured
deployment.extensions/prometheus configured
configmap/prometheus-config configured
service/grafana configured
deployment.extensions/grafana configured
configmap/grafana-config configured

$ kubectl -n linkerd get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
api          ClusterIP   10.0.173.27            8085/TCP
grafana      ClusterIP   10.0.49.44             3000/TCP
prometheus   ClusterIP   10.0.205.82            9090/TCP
proxy-api    ClusterIP   10.0.170.201           8086/TCP
web          ClusterIP   10.0.88.136            8084/TCP,9994/TCP

$ kubectl -n linkerd get pod
NAME                          READY     STATUS    RESTARTS   AGE
controller-67489d768d-75wjz   5/5       Running   0          163m
grafana-5df745d8b8-pv6tf      2/2       Running   0          163m
prometheus-d96f9bf89-2s6jg    2/2       Running   0          163m
web-5cd59f97b6-wf8nk          2/2       Running   0          57s
```

Dashboard

```
$ linkerd dashboard
Linkerd dashboard available at:
http://127.0.0.1:37737/api/v1/namespaces/linkerd/services/web:htt
Grafana dashboard available at:
http://127.0.0.1:37737/api/v1/namespaces/linkerd/services/grafana
Opening Linkerd dashboard in the default browser
```

示例应用

```
curl https://run.linkerd.io/emojivoto.yml \
  | linkerd inject - \
  | kubectl apply -f -
```

查看服务的网络流量统计情况：

```
linkerd -n emojivoto stat deployment
NAME       MESHED   SUCCESS      RPS   LATENCY_P50   LATENCY_P95
emoji         1/1   100.00%   8.1rps           1ms           1ms
vote-bot      1/1         -        -             -             -
voting        1/1    87.88%   1.1rps           1ms           1ms
web           1/1    93.65%   2.1rps           1ms           9ms
```

跟踪服务的网络流量

```
$ linkerd -n emojivoto tap deploy voting
req id=0:809 src=10.244.6.239:57202 dst=10.244.1.237:8080 :methoc
rsp id=0:809 src=10.244.6.239:57202 dst=10.244.1.237:8080 :status
end id=0:809 src=10.244.6.239:57202 dst=10.244.1.237:8080 grpc-st
req id=0:810 src=10.244.6.239:57202 dst=10.244.1.237:8080 :methoc
rsp id=0:810 src=10.244.6.239:57202 dst=10.244.1.237:8080 :status
end id=0:810 src=10.244.6.239:57202 dst=10.244.1.237:8080 grpc-st
```

参考文档

- A SERVICE MESH FOR KUBERNETES
- Service Mesh Pattern
- https://linkerd.io/2/overview/

Istio

Istio 是 Google、IBM 和 Lyft 联合开源的服务网格（Service Mesh）框架，旨在解决大量微服务的发现、连接、管理、监控以及安全等问题。Istio 对应用是透明的，不需要改动任何服务代码就可以实现透明的服务治理。

Istio 的主要特性包括：

- HTTP、gRPC、WebSocket 和 TCP 网络流量的自动负载均衡
- 细粒度的网络流量行为控制， 包括丰富的路由规则、重试、故障转移和故障注入等
- 可选策略层和配置 API 支持访问控制、速率限制以及配额管理
- 自动度量、日志记录和跟踪所有进出的流量
- 强大的身份认证和授权机制实现服务间的安全通信

Istio 原理

Istio 从逻辑上可以分为数据平面和控制平面：

- **数据平面**主要由一系列的智能代理（默认为 Envoy）组成，管理微服务之间的网络通信
- **控制平面**负责管理和配置代理来路由流量，并配置 Mixer 以进行策略部署和遥测数据收集

Istio 架构可以如下图所示

它主要由以下组件构成

- Envoy：Lyft 开源的高性能代理，用于调解服务网格中所有服务的入站和出站流量。它支持动态服务发现、负载均衡、TLS 终止、HTTP/2 和 gPRC 代理、熔断、健康检查、故障注入和性能测量等丰富的功能。Envoy 以 sidecar 的方式部署在相关的服务的 Pod 中，从而无需重新构建或重写代码。

- Mixer：负责访问控制、执行策略并从 Envoy 代理中收集遥测数据。Mixer 支持灵活的插件模型，方便扩展（支持 GCP、AWS、Prometheus、Heapster 等多种后端）。
- Pilot：动态管理 Envoy 实例的生命周期，提供服务发现、智能路由和弹性流量管理（如超时、重试）等功能。它将流量管理策略转化为 Envoy 数据平面配置，并传播到 sidecar 中。
- Pilot 为 Envoy sidecar 提供服务发现功能，为智能路由（例如 A/B 测试、金丝雀部署等）和弹性（超时、重试、熔断器等）提供流量管理功能。它将控制流量行为的高级路由规则转换为特定于 Envoy 的配置，并在运行时将它们传播到 sidecar。Pilot 将服务发现机制抽象为符合 Envoy 数据平面 API 的标准格式，以便支持在多种环境下运行并保持流量管理的相同操作接口。
- Citadel 通过内置身份和凭证管理提供服务间和最终用户的身份认证。支持基于角色的访问控制、基于服务标识的策略执行等。

在数据平面上，除了 Envoy，还可以选择使用 nginxmesh、linkerd 等作为网络代理。比如，使用 nginxmesh 时，Istio 的控制平面（Pilot、Mixer、Auth）保持不变，但用 Nginx Sidecar 取代 Envoy：

安装

Istio 的安装部署步骤见 这里。

注入 Sidecar 容器前对 Pod 的要求

为 Pod 注入 Sidecar 容器后才能成为服务网格的一部分。Istio 要求 Pod 必须满足以下条件：

- Pod 要关联服务并且必须属于单一的服务，不支持属于多个服务的 Pod
- 端口必须要命名，格式为 `<协议>[-<后缀>]`，其中协议包括 `http`、`http2`、`grpc`、`mongo` 以及 `redis`。否则会被视为 TCP 流量
- 推荐所有 Deployment 中增加 `app` 标签，用来在分布式跟踪中添加上下文信息

示例应用

> 以下步骤假设命令行终端在 安装部署 时下载的 `istio-${ISTIO_VERSION}` 目录中。

手动注入 sidecar 容器

在部署应用时，可以通过 `istioctl kube-inject` 给 Pod 手动插入 Envoy sidecar 容器，即

```
$  kubectl apply -f <(istioctl kube-inject --debug -f samples/boo
service "details" configured
deployment.extensions "details-v1" configured
service "ratings" configured
deployment.extensions "ratings-v1" configured
service "reviews" configured
deployment.extensions "reviews-v1" configured
deployment.extensions "reviews-v2" configured
deployment.extensions "reviews-v3" configured
service "productpage" configured
deployment.extensions "productpage-v1" configured
ingress.extensions "gateway" configured

$ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.y
```

原始应用如下图所示

`istioctl kube-inject` 在原始应用的每个 Pod 中插入了一个 Envoy 容器

服务启动后，可以通过 Gateway 地址 `http:///productpage` 来访问 BookInfo 应用：

```
$ kubectl get svc istio-ingressgateway -n istio-system
kubectl get svc istio-ingressgateway -n istio-system
NAME                   TYPE           CLUSTER-IP    EXTERNAL-IP
istio-ingressgateway   LoadBalancer   10.0.203.82   x.x.x.x
```

默认情况下，三个版本的 reviews 服务以负载均衡的方式轮询。

自动注入 sidecar 容器

首先确认 `admissionregistration` API 已经开启：

```
$ kubectl api-versions | grep admissionregistration
admissionregistration.k8s.io/v1beta1
```

然后确认 istio-sidecar-injector 正常运行

```
# Conform istio-sidecar-injector is working
$ kubectl -n istio-system get deploy istio-sidecar-injector
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILAE
istio-sidecar-injector   1         1         1            1
```

为需要自动注入 sidecar 的 namespace 加上标签 `istio-injection=enabled`：

```
# default namespace 没有 istio-injection 标签
$ kubectl get namespace -L istio-injection
NAME           STATUS        AGE       ISTIO-INJECTION
default        Active        1h
istio-system   Active        1h
kube-public    Active        1h
kube-system    Active        1h

# 打上 istio-injection=enabled 标签
$ kubectl label namespace default istio-injection=enabled
```

这样，在 default namespace 中创建 Pod 后自动添加 istio sidecar 容器。

参考文档

- https://istio.io/
- Istio - A modern service mesh
- https://lyft.github.io/envoy/
- https://github.com/nginmesh/nginmesh
- WHAT'S A SERVICE MESH? AND WHY DO I NEED ONE?
- A SERVICE MESH FOR KUBERNETES
- Service Mesh Pattern
- Request Routing and Policy Management with the Istio Service Mesh

安装

在安装 Istio 之前要确保 Kubernetes 集群（仅支持 v1.9.0 及以后版本）已部署并配置好本地的 kubectl 客户端。比如，使用 minikube：

```
minikube start --memory=4096 --kubernetes-version=v1.11.1 --vm-dr
```

下载 Istio

```
curl -L https://git.io/getLatestIstio | sh -
sudo apt-get install -y jq
ISTIO_VERSION=$(curl -L -s https://api.github.com/repos/istio/ist
cd istio-${ISTIO_VERSION}
cp bin/istioctl /usr/local/bin
```

部署 Istio 服务

初始化 Helm Tiller：

```
kubectl create -f install/kubernetes/helm/helm-service-account.ya
helm init --service-account tiller
```

然后使用 Helm 部署：

```
kubectl apply -f install/kubernetes/helm/istio/templates/crds.yam
helm install install/kubernetes/helm/istio --name istio --namespa
  --set ingress.enabled=true \
  --set gateways.enabled=true \
  --set galley.enabled=true \
  --set sidecarInjectorWebhook.enabled=true \
  --set mixer.enabled=true \
  --set prometheus.enabled=true \
  --set grafana.enabled=true \
  --set servicegraph.enabled=true \
  --set tracing.enabled=true \
  --set kiali.enabled=false
```

部署完成后，可以检查 `isotio-system namespace` 中的服务是否正常运行：

```
$  kubectl -n istio-system get pod
NAME                                        READY     STATUS    R
grafana-5fb774bcc9-2rkng                    1/1       Running   0
istio-citadel-5b956fdf54-5nb25              1/1       Running   0
istio-egressgateway-6cff45b4db-gt8tr        1/1       Running   0
istio-galley-699888c459-sgz7z               1/1       Running   0
istio-ingress-fc79cc885-dvjqh               1/1       Running   0
istio-ingressgateway-fc648887c-q5s5h        1/1       Running   0
istio-pilot-6cd95f9cc4-fjdb5                2/2       Running   0
istio-policy-75f75cc6fd-4mlhn               2/2       Running   0
istio-sidecar-injector-6d59d46ff4-m79tl     1/1       Running   0
istio-statsd-prom-bridge-7f44bb5ddb-phkh6   1/1       Running   0
istio-telemetry-544b8d7dcf-mk5kw            2/2       Running   0
istio-tracing-ff94688bb-7hmfb               1/1       Running   0
prometheus-84bd4b9796-hcjwc                 1/1       Running   0
servicegraph-6c6dbbf599-q4rxd               1/1       Running   0

$ kubectl -n istio-system get service
NAME                       TYPE           CLUSTER-IP     EXTERNAL
grafana                    ClusterIP      10.0.150.84
istio-citadel              ClusterIP      10.0.9.108
istio-egressgateway        ClusterIP      10.0.168.237
istio-galley               ClusterIP      10.0.160.216
istio-ingress              LoadBalancer   10.0.55.174    x.x.x.x
istio-ingressgateway       LoadBalancer   10.0.203.82    x.x.x.x
istio-pilot                ClusterIP      10.0.195.162
istio-policy               ClusterIP      10.0.14.130
istio-sidecar-injector     ClusterIP      10.0.160.50
istio-statsd-prom-bridge   ClusterIP      10.0.133.84
istio-telemetry            ClusterIP      10.0.247.30
jaeger-agent               ClusterIP      None
jaeger-collector           ClusterIP      10.0.29.72
jaeger-query               ClusterIP      10.0.19.250
prometheus                 ClusterIP      10.0.19.53
servicegraph               ClusterIP      10.0.251.76
tracing                    ClusterIP      10.0.62.176
zipkin                     ClusterIP      10.0.158.231
```

网格扩展

Istio 还支持管理非 Kubernetes 应用。此时需要在应用所在的 VM 或者物理中部署 Istio，具体步骤请参考 https://istio.io/docs/setup/kubernetes/mesh-expansion.html。注意，在部署前需要满足以下条件

- 待接入服务器必须能够通过 IP 接入网格中的服务端点。通常这需要 VPN 或者 VPC 的支持，或者容器网络为服务端点提供直接路由（非 NAT 或者防火墙屏蔽）。该服务器无需访问 Kubernetes 指派的集群 IP 地址。
- Istio 控制平面服务（Pilot、Mixer、Citadel）以及 Kubernetes 的 DNS 服务器必须能够从虚拟机进行访问，通常会使用内部负载均衡器（也可以使用 NodePort）来满足这一要求，在虚拟机上运行 Istio 组件，或者使用自定义网络配置。

部署好后，就可以向 Istio 注册应用，如

```
# istioctl register servicename machine-ip portname:port
$ istioctl -n onprem register mysql 1.2.3.4 3306
$ istioctl -n onprem register svc1 1.2.3.4 http:7000
```

Prometheus、Grafana 和 Zipkin

等所有 Pod 启动后，可以通过 NodePort、负载均衡服务的外网 IP 或者 `kubectl proxy` 来访问这些服务。比如通过 `kubectl proxy` 方式，先启动 kubectl proxy

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

通过 `http://localhost:8001/api/v1/namespaces/istio-system/services/grafana:3000/proxy/` 访问 Grafana 服务

通过 `http://localhost:8001/api/v1/namespaces/istio-system/services/servicegraph:8088/proxy/` 访问 ServiceGraph 服务，展示服务之间调用关系图

- `/force/forcegraph.html` As explored above, this is an interactive D3.js visualization.

- `/dotviz` is a static Graphviz visualization.
- `/dotgraph` provides a DOT) serialization.
- `/d3graph` provides a JSON serialization for D3 visualization.
- `/graph` provides a generic JSON serialization.

通过 `http://localhost:8001/api/v1/namespaces/istio-system/services/zipkin:9411/proxy/` 访问 Zipkin 跟踪页面

通过 `http://localhost:8001/api/v1/namespaces/istio-system/services/prometheus:9090/proxy/` 访问 Prometheus 页面

流量管理

Istio 提供了强大的流量管理功能，如智能路由、服务发现与负载均衡、故障恢复、故障注入等。

istio-traffic-management

流量管理的功能由 Pilot 配合 Envoy 负责，并接管进入和离开容器的所有流量：

- 流量管理的核心组件是 Pilot，负责管理和配置服务网格中的所有 Envoy 实例
- 而 Envoy 实例则负责维护负载均衡以及健康检查信息，从而允许其在目标实例之间智能分配流量，同时遵循其指定的路由规则

pilot

request-flow

API 版本

Istio 0.7.X 及以前版本仅支持 `config.istio.io/v1alpha2`，0.8.0 将其升级为 `networking.istio.io/v1alpha3`，并且重命名了流量管理的几个资源对象：

- RouteRule -> `VirtualService`：定义服务网格内对服务的请求如何进行路由控制，支持根据 host、sourceLabels 、http headers 等不同的路由方式，也支持百分比、超时、重试、错误注入等功能。
- DestinationPolicy -> `DestinationRule`：定义 `VirtualService` 之后的路由策略，包括断路器、负载均衡以及 TLS 等。

- EgressRule -> ServiceEntry：定义了服务网格之外的服务，支持两种类型：网格内部和网格外部。网格内的条目和其他的内部服务类似，用于显式的将服务加入网格。可以用来把服务作为服务网格扩展的一部分加入不受管理的基础设置（例如加入到基于 Kubernetes 的服务网格中的虚拟机）中。网格外的条目用于表达网格外的服务。对这种条目来说，双向 TLS 认证是禁止的，策略实现需要在客户端执行，而不像内部服务请求中的服务端执行。
- Ingress -> Gateway：定义边缘网络流量的负载均衡。

服务发现和负载均衡

为了接管流量，Istio 假设所有容器在启动时自动将自己注册到 Istio 中（通过自动或手动给 Pod 注入 Envoy sidecar 容器）。Envoy 收到外部请求后，会对请求作负载均衡，并支持轮询、随机和加权最少请求等负载均衡算法。除此之外，Envoy 还会以熔断机制定期检查服务后端容器的健康状态，自动移除不健康的容器和加回恢复正常的容器。容器内也可以返回 HTTP 503 显示将自己从负载均衡中移除。

流量接管

Istio 假定进入和离开服务网络的所有流量都会通过 Envoy 代理进行传输。Envoy sidecar 使用 iptables 把进入 Pod 和从 Pod 发出的流量转发到 Envoy 进程监听的端口（即 15001 端口）上：

```
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [1:60]
:OUTPUT ACCEPT [482:44962]
:POSTROUTING ACCEPT [482:44962]
:ISTIO_INBOUND - [0:0]
:ISTIO_IN_REDIRECT - [0:0]
:ISTIO_OUTPUT - [0:0]
:ISTIO_REDIRECT - [0:0]
-A PREROUTING -p tcp -j ISTIO_INBOUND
-A OUTPUT -p tcp -j ISTIO_OUTPUT
-A ISTIO_INBOUND -p tcp -m tcp --dport 9080 -j ISTIO_IN_REDIRECT
-A ISTIO_IN_REDIRECT -p tcp -j REDIRECT --to-ports 15001
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -j ISTIO_REDIRECT
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -d 127.0.0.1/32 -j RETURN
-A ISTIO_OUTPUT -j ISTIO_REDIRECT
-A ISTIO_REDIRECT -p tcp -j REDIRECT --to-ports 15001
```

故障恢复

Istio 提供了一系列开箱即用的故障恢复功能，如

- 超时处理
- 重试处理，如限制最大重试时间以及可变重试间隔
- 健康检查，如自动移除不健康的容器
- 请求限制，如并发请求数和并发连接数
- 熔断

这些功能均可以使用 VirtualService 动态配置。比如以下为用户 jason 的请求返回 500 （而其他用户均可正常访问）：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - match:
    - headers:
        cookie:
          regex: "^(.*?;)?(user=jason)(;.*)?$"
    fault:
      abort:
        percent: 100
        httpStatus: 500
    route:
    - destination:
        host: ratings
        subset: v1
  - route:
    - destination:
        host: ratings
        subset: v1
```

熔断示例：

```
cat <<f -
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      http:
        consecutiveErrors: 1
        interval: 1s
        baseEjectionTime: 3m
        maxEjectionPercent: 100
EOF
```

故障注入

Istio 支持为应用注入故障，以模拟实际生产中碰到的各种问题，包括

- 注入延迟（模拟网络延迟和服务过载）
- 注入失败（模拟应用失效）

这些故障均可以使用 VirtualService 动态配置。如以下配置 2 秒的延迟：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percent: 100
        fixedDelay: 2s
    route:
    - destination:
        host: ratings
        subset: v1
```

金丝雀部署

`service-versions`

首先部署 `bookinfo`，并配置默认路由为 `v1` 版本：

```
# 以下命令假设 bookinfo 示例程序已部署，如未部署，可以执行下面的命令
$ kubectl apply -f <(istioctl kube-inject -f samples/bookinfo/pla
# 此时，三个版本的 reviews 服务以负载均衡的方式轮询。

# 创建默认路由，全部请求转发到 v1
$ istioctl create -f samples/bookinfo/routing/route-rule-all-v1.y

$ kubectl get virtualservice reviews -o yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

示例一：将 10% 请求发送到 v2 版本而其余 90% 发送到 v1 版本

```
cat <<f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 75
    - destination:
        host: reviews
        subset: v2
      weight: 25
EOF
```

示例二：将 jason 用户的请求全部发到 v2 版本

```
cat <-f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - match:
    - sourceLabels:
        app: reviews
        version: v2
      headers:
        end-user:
          exact: jason
EOF
```

示例三：全部切换到 v2 版本

```
cat <-f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
EOF
```

示例四：限制并发访问

```
cat <<f -
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
EOF
```

为了查看访问次数限制的效果，可以使用 `wrk` 给应用加一些压力：

```
export BOOKINFO_URL=$(kubectl get po -n istio-system -l istio=ing
wrk -t1 -c1 -d20s http://$BOOKINFO_URL/productpage
```

Gateway

Istio 在部署时会自动创建一个 Istio Gateway，用来控制 Ingress 访问。

```
# prepare
kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbi
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /tmp/

# get ingress external IP (suppose load balancer service)
kubectl get svc istio-ingressgateway -n istio-system
export INGRESS_HOST=$(kubectl -n istio-system get service istio-i
export INGRESS_PORT=$(kubectl -n istio-system get service istio-i
export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service

# create gateway
cat <-f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementat
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "httpbin.example.com"
EOF

# configure routes for the gateway
cat <-f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
  - "httpbin.example.com"
  gateways:
  - httpbin-gateway
  http:
```

```
  - match:
    - uri:
        prefix: /status
    - uri:
        prefix: /delay
    route:
    - destination:
        port:
          number: 8000
        host: httpbin
EOF

# validate 200
curl --resolve httpbin.example.com:$INGRESS_PORT:$INGRESS_HOST -H

# invalidate 404
curl --resolve httpbin.example.com:$INGRESS_PORT:$INGRESS_HOST -H
```

使用 TLS：

```
kubectl create -n istio-system secret tls istio-ingressgateway-ce

cat <<-f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "httpbin.example.com"
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
      privateKey: /etc/istio/ingressgateway-certs/tls.key
    hosts:
    - "httpbin.example.com"
EOF


# validate 200
curl --resolve httpbin.example.com:$SECURE_INGRESS_PORT:$INGRESS_
```

Egress 流量

默认情况下，Istio 接管了容器的内外网流量，从容器内部无法访问 Kubernetes 集群外的服务。可以通过 ServiceEntry 为需要的容器开放 Egress 访问，如

```
$ cat <apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-ext
spec:
  hosts:
  - httpbin.org
  ports:
  - number: 80
    name: http
    protocol: HTTP
EOF

$ cat <apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin-ext
spec:
  hosts:
    - httpbin.org
  http:
  - timeout: 3s
    route:
      - destination:
          host: httpbin.org
        weight: 100
EOF
```

需要注意的是 ServiceEntry 仅支持 HTTP、TCP 和 HTTPS，对于其他协议需要通过 `--includeIPRanges` 的方式设置 IP 地址范围，如

```
helm template @install/kubernetes/helm/istio@ --name istio --name
```

流量镜像

```
cat <-f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - httpbin
  http:
  - route:
    - destination:
        host: httpbin
        subset: v1
      weight: 100
    mirror:
      host: httpbin
      subset: v2
EOF
```

参考文档

- Istio traffic management overview

安全管理

Istio 提供了 RBAC 访问控制、双向 TLS 认证以及密钥管理等安全管理功能。

RBAC

Istio Role-Based Access Control (RBAC) 提供了 namespace、service 以及 method 级别的访问控制。其特性包括

- 简单易用：提供基于角色的语意

- 支持认证：提供服务 - 服务和用户 - 服务的认证
- 灵活：提供角色和角色绑定的自定义属性

image-20180423202459184

开启 RBAC

通过 `RbacConfig` 来启用 RBAC，其中 mode 支持如下选项：

- **OFF**：停用 RBAC。
- **ON**：为网格中的所有服务启用 RBAC。
- **ON_WITH_INCLUSION**：只对 `inclusion` 字段中包含的命名空间和服务启用 RBAC。
- **ON_WITH_EXCLUSION**：对网格内的所有服务启用 RBAC，除 `exclusion` 字段中包含的命名空间和服务之外。

下面的例子为 `default` 命名空间开启 RBAC：

```
apiVersion: "config.istio.io/v1alpha2"
kind: RbacConfig
metadata:
  name: default
  namespace: istio-system
spec:
  mode: ON_WITH_INCLUSION
  inclusion:
    namespaces: ["default"]
```

访问控制

Istio RBAC 提供了 ServiceRole 和 ServiceRoleBinding 两种资源对象，并以 CustomResourceDefinition（CRD）的方式管理。

- ServiceRole 定义了一个可访问特定资源（namespace 之内）的服务角色，并支持以前缀通配符和后缀通配符的形式匹配一组服务
- ServiceRoleBinding 定义了赋予指定角色的绑定，即可以指定的角色和动作访问服务

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: products-viewer
  namespace: default
spec:
  rules:
  - services: ["products.default.svc.cluster.local"]
    methods: ["GET", "HEAD"]

---
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: test-binding-products
  namespace: default
spec:
  subjects:
  - user: "service-account-a"
  - user: "istio-ingress-service-account"
    properties:
    - request.auth.claims[email]: "a@foo.com"
    roleRef:
    kind: ServiceRole
    name: "products-viewer"
```

双向 TLS

双向 TLS 为服务间通信提供了 TLS 认证，并提供管理系统自动管理密钥和证书的生成、分发、替换以及撤销。

实现原理

Istio Auth 由三部分组成：

- 身份（Identity）：Istio 使用 Kubernetes service account 来识别服务的身份，格式为 `spiffe://<*domain*>/ns/<*namespace*>/sa/<*serviceaccount*>`

- 通信安全：端到端 TLS 通信通过服务器端和客户端的 Envoy 容器完成
- 证书管理：Istio CA（Certificate Authority）负责为每个 service account 生成 SPIFEE 密钥和证书、分发到 Pod（通过 Secret Volume Mount 的形式）、定期轮转（Rotate）以及必要时撤销。对于 Kuberentes 之外的服务，CA 配合 Istio node agent 共同完成整个过程。

这样，一个容器使用证书的流程为

- 首先，Istio CA 监听 Kubernetes API，并为 service account 生成 SPIFFE 密钥及证书，再以 secret 形式存储到 Kubernetes 中
- 然后，Pod 创建时，Kubernetes API Server 将 secret 挂载到容器中
- 最后，Pilot 生成一个访问控制的配置，定义哪些 service account 可以访问服务，并分发给 Envoy
- 而当容器间通信时，Pod 双方的 Envoy 就会基于访问控制配置来作认证

最佳实践

- 为不同团队创建不同 namespace 分别管理
- 将 Istio CA 运行在单独的 namespace 中，并且仅授予管理员权限

参考文档

- Istio Security 文档
- Istio Role-Based Access Control (RBAC)
- Istio 双向 TLS 文档

策略管理

Mixer 为应用程序和基础架构后端之间提供了一个通用的策略控制层，负责先决条件检查（如认证授权）、配额管理并从 Envoy 代理中收集遥测数据等。

Mixer 是高度模块化和可扩展的组件。他的一个关键功能就是把不同后端的策略和遥测收集系统的细节抽象出来，使得 Istio 的其余部分对这些后端不知情。Mixer 处理不同基础设施后端的灵活性是通过使用通用插件模型实现的。每个插件都被称为 **Adapter**，Mixer通过它们与不同的基础设施后端连

接，这些后端可提供核心功能，例如日志、监控、配额、ACL 检查等。通过配置能够决定在运行时使用的确切的适配器套件，并且可以轻松扩展到新的或定制的基础设施后端。

实现原理

本质上，Mixer 是一个 属性 处理机，进入 Mixer 的请求带有一系列的属性，Mixer 按照不同的处理阶段处理：

- 通过全局 Adapters 为请求引入新的属性
- 通过解析（Resolution）识别要用于处理请求的配置资源
- 处理属性，生成 Adapter 参数
- 分发请求到各个 Adapters 后端处理

流量限制示例

```
apiVersion: "config.istio.io/v1alpha2"
kind: memquota
metadata:
  name: handler
  namespace: istio-system
spec:
  quotas:
  - name: requestcount.quota.istio-system
    maxAmount: 5000
    validDuration: 1s
    # The first matching override is applied.
    # A requestcount instance is checked against override dimensi
    overrides:
    # The following override applies to 'ratings' when
    # the source is 'reviews'.
    - dimensions:
        destination: ratings
        source: reviews
      maxAmount: 1
      validDuration: 1s
    # The following override applies to 'ratings' regardless
    # of the source.
    - dimensions:
        destination: ratings
      maxAmount: 100
      validDuration: 1s

---
apiVersion: "config.istio.io/v1alpha2"
kind: quota
metadata:
  name: requestcount
  namespace: istio-system
spec:
  dimensions:
    source: source.labels["app"] | source.service | "unknown"
    sourceVersion: source.labels["version"] | "unknown"
    destination: destination.labels["app"] | destination.service
    destinationVersion: destination.labels["version"] | "unknown"
```

```
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: quota
  namespace: istio-system
spec:
  actions:
  - handler: handler.memquota
    instances:
    - requestcount.quota
---
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpec
metadata:
  name: request-count
  namespace: istio-system
spec:
  rules:
  - quotas:
    - charge: 1
      quota: requestcount
---
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpecBinding
metadata:
  name: request-count
  namespace: istio-system
spec:
  quotaSpecs:
  - name: request-count
    namespace: istio-system
  services:
  - name: ratings
  - name: reviews
  - name: details
  - name: productpage
```

参考文档

- Istio Mixer

度量管理

新增指标

Istio 支持 自定义指标、日志 以及 TCP 指标。可以通过指标配置来新增这些度量，每个配置包括三方面的内容：

1. 从 Istio 属性中生成度量实例，如logentry 、metrics 等。
2. 创建处理器（适配 Mixer），用来处理生成的度量实例，如 prometheus。
3. 根据一系列的股则，把度量实例传递给处理器，即创建 rule。

```yaml

### 指标 instance 的配置

apiVersion: "config.istio.io/v1alpha2" kind: metric metadata: name: doublerequestcount namespace: istio-system spec: value: "2" # 每个请求计数两次 dimensions: source: source.service | "unknown" destination: destination.service | "unknown" message: '"twice the fun!"' monitored_resource_type: '"UNSPECIFIED"'

---

## prometheus handler 的配置

apiVersion: "config.istio.io/v1alpha2" kind: prometheus metadata: name: doublehandler namespace: istio-system spec: metrics:

- name: double_request_count # Prometheus 指标名称 instance_name: doublerequestcount.metric.istio-system
```

```
# Mixer Instance 名称（全限定名称） kind: COUNTER
label_names:
  ◦ source
  ◦ destination
  ◦ message
```

将指标 Instance 发送给 prometheus handler 的 rule 对象

apiVersion: "config.istio.io/v1alpha2" kind: rule metadata: name: doubleprom namespace: istio-system spec: actions:

- handler: doublehandler.prometheus instances:
  ◦ doublerequestcount.metric ```

Prometheus

在命令行中执行以下命令：

```
$ kubectl -n istio-system port-forward service/prometheus 9090:90
```

在 Web 浏览器中访问 `http://localhost:9090` 即可以访问 Prometheus UI，查询度量指标。

Jaeger 分布式跟踪

在命令行中执行以下命令：

```
$ kubectl -n istio-system port-forward service/jaeger-query 16686
```

在 Web 浏览器中访问 `http://localhost:16686` 即可以访问 Jaeger UI。

Grafana 可视化

在命令行中执行以下命令：

```
$ kubectl -n istio-system port-forward service/grafana 3000:3000
```

在 Web 浏览器中访问 `http://localhost:3000` 即可以访问 Grafana 界面。

服务图

在命令行中执行以下命令：

```
$ kubectl -n istio-system port-forward $(kubectl -n istio-system
```

在 Web 浏览器中访问 `http://localhost:8088/force/forcegraph.html` 即可以访问生成的服务图。

排错

请见 https://istio.io/help/。

社区社区

- Github
- 设计文档
- 工作组（Working Groups）
- 邮件列表
  - istio-users@
  - istio-dev@
  - istio-announce@
- Twitter
- Rocket.Chat
- FAQ
- 术语表

Devops

Kubernetes 生态中的 Devops 工具实践。

源码部署（Source to Deployment）

- Draft： 提供了一个用于简化容器构建和部署（基于 Helm）的工具，使用方法见这里
- Skaffold：同 Draft 类似，但不支持 Helm，使用方法见这里
- Metaparticle：提供了一套用于开发云原生应用的标准库，使用方法见 https://metaparticle.io

CI/CD

- Jenkins X
- Spinnaker
- Argo
- Flux GitOps

其他

- Kompose

Draft

Draft 是微软 Deis 团队开源（见 https://github.com/azure/draft）的容器应用开发辅助工具，它可以帮助开发人员简化容器应用程序的开发流程。

Draft 主要由三个命令组成

- `draft init`：初始化 docker registry 账号，并在 Kubernetes 集群中部署 draftd（负责镜像构建、将镜像推送到 docker registry 以及部署应用等）
- `draft create`：draft 根据 packs 检测应用的开发语言，并自动生成 Dockerfile 和 Kubernetes Helm Charts

- `draft up`：根据 Dockfile 构建镜像，并使用 Helm 将应用部署到 Kubernetes 集群（支持本地或远端集群）。同时，还会在本地启动一个 draft client，监控代码变化，并将更新过的代码推送给 draftd。

Draft 安装

由于 Draft 需要构建镜像并部署应用到 Kubernetes 集群，因而在安装 Draft 之前需要

- 部署一个 Kubernetes 集群，部署方法可以参考 kubernetes 部署方法
- 安装并初始化 helm（需要 v2.4.x 版本，并且不要忘记运行 `helm init`），具体步骤可以参考 helm 使用方法
- 注册 docker registry 账号，比如 Docker Hub 或Quay.io
- 配置 Ingress Controller 并在 DNS 中设置通配符域 `*` 的 A 记录（如 `*.draft.example.com`）到 Ingress IP 地址。最简单的 Ingress Controller 创建方式是使用 helm：

```
# 部署 nginx ingress controller
$ helm install stable/nginx-ingress --namespace=kube-system --nam
# 等待 ingress controller 配置完成，并记下外网 IP
$ kubectl --namespace kube-system get services -w nginx-ingress-r
```

> **minikube Ingress Controller**
>
> minikube 中配置和使用 Ingress Controller 的方法可以参考这里。

初始化好 Kubernetes 集群和 Helm 后，可以在 这里 下载 draft 二进制文件，并配置 draft

```
# 注意修改用户名、密码和邮件
$ token=$(echo '{"username":"feisky","password":"secret","email":
# 注意修改 registry.org 和 basedomain
$ draft init --set registry.url=docker.io,registry.org=feisky,reg
```

Draft 入门

draft 源码中提供了很多应用的 示例，我们来看一下怎么用 draft 来简化 python 应用的开发流程。

```
$ git clone https://github.com/Azure/draft.git
$ cd draft/examples/python
$ ls
app.py           requirements.txt

$ cat requirements.txt
flask
$ cat app.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return "Hello, World!\n"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Draft create 生成 Dockerfile 和 chart

```
$ draft create
--> Python app detected
--> Ready to sail
$ ls
Dockerfile       app.py           chart            draft.toml
$ cat Dockerfile
FROM python:onbuild
EXPOSE 8080
ENTRYPOINT ["python"]
CMD ["app.py"]
$ cat draft.toml
[environments]
  [environments.development]
    name = "virulent-sheep"
    namespace = "default"
    watch = true
    watch_delay = 2
```

Draft Up 构建镜像并部署应用

```
$ draft up
--> Building Dockerfile
Step 1 : FROM python:onbuild
onbuild: Pulling from library/python
10a267c67f42: Pulling fs layer
....
Digest: sha256:5178d22192c2b8b4e1140a3bae9021ee0e808d754b43100147
Status: Downloaded newer image for python:onbuild
# Executing 3 build triggers...
Step 1 : COPY requirements.txt /usr/src/app/
Step 1 : RUN pip install --no-cache-dir -r requirements.txt
....
Successfully built f742caba47ed
--> Pushing docker.io/feisky/virulent-sheep:de7e97d0d889b4cdb81ae
....
de7e97d0d889b4cdb81ae4b972097d759c59e06e: digest: sha256:7ee10c1a
--> Deploying to Kubernetes
    Release "virulent-sheep" does not exist. Installing it now.
--> Status: DEPLOYED
--> Notes:

  http://virulent-sheep.app.feisky.xyzto access your application

Watching local files for changes...
```

打开一个新的 shell，就可以通过子域名来访问应用了

```
$ curl virulent-sheep.app.feisky.xyz
Hello, World!
```

JenkinsX

Jenkins X 是一个基于 Jenkins 和 Kubernetes 的 CI/CD 平台，旨在解决微服务架构下云原生应用的持续集成和持续交付问题。它使用 Jenkins、Helm、Draft、GitOps 以及 Github 等工具链构造了一个从集群安装、环境管理、持续集成、持续部署一直到应用发布等支持整个流程的平台。

安装部署

安装 jx 命令行工具

```
# MacOS
brew tap jenkins-x/jx
brew install jx

# Linux
curl -L https://github.com/jenkins-x/jx/releases/download/v1.1.10
sudo mv jx /usr/local/bin
```

部署 Kubernetes 集群

如果 Kubernetes 集群已经部署好了，那么该步可以忽略。

`jx` 命令提供了在公有云中直接部署 Kubernetes 的功能，比如

```
create cluster aks      # Create a new kubernetes cluster on AKS:
create cluster aws      # Create a new kubernetes cluster on AWS
create cluster gke      # Create a new kubernetes cluster on GKE:
create cluster minikube # Create a new kubernetes cluster with mi
```

部署 Jenkins X 服务

注意在安装 Jenkins X 服务之前，Kubernetes 集群需要开启 RBAC 并开启 insecure docker registries（`dockerd --insecure-registry=10.0.0.0/16`）。

运行下面的命令按照提示操作，该过程会配置

- Ingress Controller（如果没有安装的话）
- Ingress 公网 IP 的 DNS（默认使用 `ip.xip.io`）
- Github API token（用于创建 github repo 和 webhook）
- Jenkins-X 服务
- 创建 staging 和 production 等示例项目，包括 github repo 以及 Jenkins 配置等

```
jx install --provider=kubernetes
```

安装完成后，会输出 Jenkins 的访问入口以及管理员的用户名和密码，用于登录 Jenkins。

创建应用

Jenkins X 支持快速创建新的应用

```
# 创建 Spring Boot 应用
jx create spring -d web -d actuator

# 创建快速启动项目
jx create quickstart  -l go
```

也支持导入已有的应用，只是需要注意导入前要保证

- 使用 Github 等 git 系统管理源码并设置好 Jenkins webhook
- 添加 Dockerfile、Jenkinsfile 以及运行应用所需要的 Helm Chart

```
# 从本地导入
$ cd my-cool-app
$ jx import

# 从 Github 导入
jx import --github --org myname

# 从 URL 导入
jx import --url https://github.com/jenkins-x/spring-boot-web-exam
```

发布应用

```
# 发布新版本到生产环境中
jx promote myapp --version 1.2.3 --env production
```

常用命令

```
# Get pipelines
jx get pipelines

# Get pipeline activities
jx get activities

# Get build logs
jx get build logs -f myapp

# Open Jenkins in brower
jx console

# Get applications
jx get applications

# Get environments
jx get environments
```

Spinnaker

Spinnaker 是 Google 与 Netflix 发布的企业级持续交付平台，具有多云部署、自动发布、权限控制以及应用最佳实践等诸多优点。

部署

```
helm install --name spinnaker stable/spinnaker
```

Kompose

Kompose是一个将docker-compose配置转换成Kubernetes manifests的工具，官方网站为http://kompose.io/。

Kompose安装

```
# Linux
$ curl -L https://github.com/kubernetes-incubator/kompose/release

# macOS
$ curl -L https://github.com/kubernetes-incubator/kompose/release

# Windows
$ curl -L https://github.com/kubernetes-incubator/kompose/release

# 放到PATH中
$ chmod +x kompose
$ sudo mv ./kompose /usr/local/bin/kompose
```

Kompose使用

docker-compose.yaml

```
version: "2"

services:

  redis-master:
    image: gcr.io/google_containers/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v1
    ports:
      - "6379"
    environment:
      - GET_HOSTS_FROM=dns

  frontend:
    image: gcr.io/google-samples/gb-frontend:v4
    ports:
      - "80:80"
    environment:
      - GET_HOSTS_FROM=dns
    labels:
      kompose.service.type: LoadBalancer
```

kompose up

```
$ kompose up
We are going to create Kubernetes Deployments, Services and Persi
If you need different kind of resources, use the 'kompose convert

INFO Successfully created Service: redis
INFO Successfully created Service: web
INFO Successfully created Deployment: redis
INFO Successfully created Deployment: web

Your application has been deployed to Kubernetes. You can run 'ku
```

kompose convert

```
$ kompose convert
INFO file "frontend-service.yaml" created
INFO file "redis-master-service.yaml" created
INFO file "redis-slave-service.yaml" created
INFO file "frontend-deployment.yaml" created
INFO file "redis-master-deployment.yaml" created
INFO file "redis-slave-deployment.yaml" created
```

Skaffold

Skaffold 是谷歌开源的简化本地 Kubernetes 应用开发的工具。它将构建镜像、推送镜像以及部署 Kubernetes 服务等流程自动化，可以方便地对 Kubernetes 应用进行持续开发。其功能特点包括

- 没有服务器组件
- 自动检测代码更改并自动构建、推送和部署服务
- 自动管理镜像标签
- 支持已有工作流
- 保存文件即部署

安装

```
# Linux
curl -Lo skaffold https://storage.googleapis.com/skaffold/release

# MacOS
curl -Lo skaffold https://storage.googleapis.com/skaffold/release
```

使用

在使用 skaffold 之前需要确保

- Kubernetes 集群已部署并配置好本地 kubectl 命令行

- 本地 Docker 处于运行状态并登录 DockerHub 或其他的 Docker Registry
- skaffold 命令行已下载并放到系统 PATH 路径中

skaffold 代码库提供了一些列的示例，我们来看一个最简单的。

下载示例应用：

```
$ git clone https://github.com/GoogleCloudPlatform/skaffold
$ cd skaffold/examples/getting-started
```

修改 `k8s-pod.yaml` 和 `skaffold.yaml` 文件中的镜像，将 `gcr.io/k8s-skaffold` 替换为已登录的 Docker Registry。然后运行 skaffold

```
$ skaffold dev
Starting build...
Found [minikube] context, using local docker daemon.
Sending build context to Docker daemon  6.144kB
Step 1/5 : FROM golang:1.9.4-alpine3.7
 ---> fb6e10bf973b
Step 2/5 : WORKDIR /go/src/github.com/GoogleCloudPlatform/skaffol
 ---> Using cache
 ---> e9d19a54595b
Step 3/5 : CMD ./app
 ---> Using cache
 ---> 154b6512c4d9
Step 4/5 : COPY main.go .
 ---> Using cache
 ---> e097086e73a7
Step 5/5 : RUN go build -o app main.go
 ---> Using cache
 ---> 9c4622e8f0e7
Successfully built 9c4622e8f0e7
Successfully tagged 930080f0965230e824a79b9e7eccffbd:latest
Successfully tagged gcr.io/k8s-skaffold/skaffold-example:9c4622e8
Build complete in 657.426821ms
Starting deploy...
Deploying k8s-pod.yaml...
Deploy complete in 173.770268ms
[getting-started] Hello world!
```

此时，打开另外一个终端，修改 `main.go` 的内容后 skaffold 会自动执行

- 构建一个新的镜像（带有不同的 sha256 TAG）
- 修改 `k8s-pod.yaml` 文件中的镜像为新的 TAG
- 重新部署 `k8s-pod.yaml`

Argo

Argo 是一个基于 Kubernetes 的工作流引擎，同时也支持 CI、CD 等丰富的功能。Argo 开源在 https://github.com/argoproj。

安装 Argo

使用 argo install

```
# Downlaod Argo.
curl -sSL -o argo https://github.com/argoproj/argo/releases/downl
chmod +x argo
sudo mv argo /usr/local/bin/argo

# Deploy to kubernetes
kubectl create namespace argo
argo install -n argo
```

```
ACCESS_KEY=AKIAIOSFODNN7EXAMPLE
ACCESS_SECRET_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

helm install --namespace argo --name argo-artifacts --set accessK
```

创建名为 `argo-bucket` 的 Bucket（可以通过 `kubectl port-forward service/argo-artifacts-minio :9000` 访问 Minio UI 来操作）：

```
# download mc client
sudo wget https://dl.minio.io/client/mc/release/linux-amd64/mc -O
sudo chmod +x /usr/local/bin/mc

# create argo-bucket
EXTERNAL_IP=$(kubectl -n argo get service argo-artifacts-minio -o
mc config host add argo-artifacts-minio-local http://$EXTERNAL_IP
mc mb argo-artifacts-minio-local/argo-bucket
```

然后修改 Argo 工作流控制器使用 Minio：

```
$ kubectl -n argo create secret generic argo-artifacts-minio --fr
$ kubectl edit configmap workflow-controller-configmap -n argo
...
    executorImage: argoproj/argoexec:v2.0.0
    artifactRepository:
      s3:
        bucket: argo-bucket
        endpoint: argo-artifacts-minio.argo:9000
        insecure: true
        # accessKeySecret and secretKeySecret are secret selector
        # It references the k8s secret named 'argo-artifacts-mini
        # which was created during the minio helm install. The ke
        # 'accesskey' and 'secretkey', inside that secret are whe
        # actual minio credentials are stored.
        accessKeySecret:
          name: argo-artifacts-minio
          key: accesskey
        secretKeySecret:
          name: argo-artifacts-minio
          key: secretkey
```

使用 Helm

> 注意：当前 Helm Charts 使用的 Minio 版本较老，部署有可能会失败。

```
# Downlaod Argo.
curl -sSL -o /usr/local/bin/argo https://github.com/argoproj/argo
chmod +x /usr/local/bin/argo

# Deploy to kubernetes
helm repo add argo https://argoproj.github.io/argo-helm/
kubectl create clusterrolebinding default-admin --clusterrole=clu
helm install argo/argo-ci --name argo-ci --namespace=kube-system
```

访问 Argo UI

```
$ kubectl -n argo port-forward service/argo-ui :80
Forwarding from 127.0.0.1:52592 -> 8001
Forwarding from [::1]:52592 -> 8001

# 使用浏览器打开 127.0.0.1:52592
```

工作流

首先，给默认的 `ServiceAccount` 授予集群管理权限

```
# Authz yourself if you are not admin.
kubectl create clusterrolebinding default-admin --clusterrole=clu
```

示例1： 最简单的工作流

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-
spec:
  entrypoint: whalesay
  templates:
  - name: whalesay
    container:
      image: docker/whalesay:latest
      command: [cowsay]
      args: ["hello world"]
```

```
argo -n argo submit https://raw.githubusercontent.com/argoproj/ar
```

示例2：包含多个容器的工作流

```
# This example demonstrates the ability to pass artifacts
# from one step to the next.
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: artifact-passing-
spec:
  entrypoint: artifact-example
  templates:
  - name: artifact-example
    steps:
    - - name: generate-artifact
        template: whalesay
    - - name: consume-artifact
        template: print-message
        arguments:
          artifacts:
          - name: message
            from: "{{steps.generate-artifact.outputs.artifacts.he

  - name: whalesay
    container:
      image: docker/whalesay:latest
      command: [sh, -c]
      args: ["cowsay hello world | tee /tmp/hello_world.txt"]
    outputs:
      artifacts:
      - name: hello-art
        path: /tmp/hello_world.txt

  - name: print-message
    inputs:
      artifacts:
      - name: message
        path: /tmp/message
    container:
      image: alpine:latest
      command: [sh, -c]
      args: ["cat /tmp/message"]
```

```
argo -n argo submit https://raw.githubusercontent.com/argoproj/ar
```

工作流创建完成后，可以查询它们的状态和日志，并在不需要时删除：

```
$ argo list
NAME                       STATUS    AGE   DURATION
artifact-passing-65p6g     Running   6s    4s
hello-world-cdnpq          Running   8s    6s

$ argo -n argo logs hello-world-4dhg8
 _____________
< hello world >
 -------------
    \
     \
      \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/

$ argo -n argo delete hello-world-4dhg8
Workflow 'hello-world-4dhg8' deleted
```

更多工作流 YAML 的格式见官方文档和工作流示例。

FluxGitOps

TODO: https://github.com/weaveworks/flux.

实践案例

实践案例

Kubernetes 实践及常用技巧，包括

- 监控
- 日志
- 高可用
- 调试
- 端口映射
- 端口转发
- GPU
- 安全
- 审计

高可用

Kubernetes 从 1.5 开始，通过 `kops` 或者 `kube-up.sh` 部署的集群会自动部署一个高可用的系统，包括

- Etcd 集群模式
- kube-apiserver 负载均衡
- kube-controller-manager、kube-scheduler 和 cluster-autoscaler 自动选主（有且仅有一个运行实例）

如下图所示

注意：以下步骤假设每台机器上 Kubelet 和 Docker 已配置并处于正常运行状态。

Etcd 集群

安装 cfssl

```
# On all etcd nodes
curl -o /usr/local/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_lir
curl -o /usr/local/bin/cfssljson https://pkg.cfssl.org/R1.2/cfssl
chmod +x /usr/local/bin/cfssl*
```

生成 CA certs:

```
# SSH etcd0
mkdir -p /etc/kubernetes/pki/etcd
cd /etc/kubernetes/pki/etcd
cat >ca-config.json <"signing": {
        "default": {
            "expiry": "43800h"
        },
        "profiles": {
            "server": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "server auth",
                    "client auth"
                ]
            },
            "client": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "client auth"
                ]
            },
            "peer": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "server auth",
                    "client auth"
                ]
            }
        }
    }
}
EOF
cat >ca-csr.json <"CN": "etcd",
    "key": {
```

```
        "algo": "rsa",
        "size": 2048
    }
}
EOF
cfssl gencert -initca ca-csr.json | cfssljson -bare ca -

# generate client certs
cat >client.json <"CN": "client",
    "key": {
        "algo": "ecdsa",
        "size": 256
    }
}
EOF
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.jso
```

生成 etcd server/peer certs

```
# Copy files to other etcd nodes
mkdir -p /etc/kubernetes/pki/etcd
cd /etc/kubernetes/pki/etcd
scp root@:/etc/kubernetes/pki/etcd/ca.pem .
scp root@:/etc/kubernetes/pki/etcd/ca-key.pem .
scp root@:/etc/kubernetes/pki/etcd/client.pem .
scp root@:/etc/kubernetes/pki/etcd/client-key.pem .
scp root@:/etc/kubernetes/pki/etcd/ca-config.json .

# Run on all etcd nodes
cfssl print-defaults csr > config.json
sed -i '0,/CN/{s/example\.net/'"$PEER_NAME"'/}' config.json
sed -i 's/www\.example\.net/'"$PRIVATE_IP"'/' config.json
sed -i 's/example\.net/'"$PUBLIC_IP"'/' config.json
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.jso
cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.jso
```

最后运行 etcd，将如下的 yaml 配置写入每台 etcd 节点的 `/etc/kubernetes/manifests/etcd.yaml` 文件中，注意替换

- ` ` 为 etcd 节点名称 （比如 `etcd0`，`etcd1` 和 `etcd2`）

- , and 为 etcd 节点的内网 IP 地址

```
cat >/etc/kubernetes/manifests/etcd.yaml <
namespace: kube-system
spec:
containers:
- command:
    - etcd --name ${PEER_NAME} \
    - --data-dir /var/lib/etcd \
    - --listen-client-urls https://${PRIVATE_IP}:2379 \
    - --advertise-client-urls https://${PRIVATE_IP}:2379 \
    - --listen-peer-urls https://${PRIVATE_IP}:2380 \
    - --initial-advertise-peer-urls https://${PRIVATE_IP}:2380 \
    - --cert-file=/certs/server.pem \
    - --key-file=/certs/server-key.pem \
    - --client-cert-auth \
    - --trusted-ca-file=/certs/ca.pem \
    - --peer-cert-file=/certs/peer.pem \
    - --peer-key-file=/certs/peer-key.pem \
    - --peer-client-cert-auth \
    - --peer-trusted-ca-file=/certs/ca.pem \
    - --initial-cluster etcd0=https://:2380,etcd1=https://:2380,e
    - --initial-cluster-token my-etcd-token \
    - --initial-cluster-state new
    image: gcr.io/google_containers/etcd-amd64:3.1.0
    livenessProbe:
    httpGet:
        path: /health
        port: 2379
        scheme: HTTP
    initialDelaySeconds: 15
    timeoutSeconds: 15
    name: etcd
    env:
    - name: PUBLIC_IP
    valueFrom:
        fieldRef:
        fieldPath: status.hostIP
    - name: PRIVATE_IP
    valueFrom:
        fieldRef:
        fieldPath: status.podIP
```

```
    - name: PEER_NAME
    valueFrom:
        fieldRef:
        fieldPath: metadata.name
    volumeMounts:
    - mountPath: /var/lib/etcd
    name: etcd
    - mountPath: /certs
    name: certs
hostNetwork: true
volumes:
- hostPath:
    path: /var/lib/etcd
    type: DirectoryOrCreate
    name: etcd
- hostPath:
    path: /etc/kubernetes/pki/etcd
    name: certs
EOF
```

> 注意：以上方法需要每个 etcd 节点都运行 kubelet。如果不想使用 kubelet，还可以通过 systemd 的方式来启动 etcd：

```
export ETCD_VERSION=v3.1.10
curl -sSL https://github.com/coreos/etcd/releases/download/$
rm -rf etcd-$ETCD_VERSION-linux-amd64*

touch /etc/etcd.env
echo "PEER_NAME=$PEER_NAME" >> /etc/etcd.env
echo "PRIVATE_IP=$PRIVATE_IP" >> /etc/etcd.env

cat >/etc/systemd/system/etcd.service <local/bin/etcd --name
    --data-dir /var/lib/etcd \
    --listen-client-urls https://${PRIVATE_IP}:2379 \
    --advertise-client-urls https://${PRIVATE_IP}:2379 \
    --listen-peer-urls https://${PRIVATE_IP}:2380 \
    --initial-advertise-peer-urls https://${PRIVATE_IP}:2380
    --cert-file=/etc/kubernetes/pki/etcd/server.pem \
    --key-file=/etc/kubernetes/pki/etcd/server-key.pem \
    --client-cert-auth \
    --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.pem \
    --peer-cert-file=/etc/kubernetes/pki/etcd/peer.pem \
    --peer-key-file=/etc/kubernetes/pki/etcd/peer-key.pem \
    --peer-client-cert-auth \
    --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.pem \
    --initial-cluster etcd0=https://:2380,etcd1=https://:238
    --initial-cluster-token my-etcd-token \
    --initial-cluster-state new

[Install]
WantedBy=multi-user.target
EOF

systemctl daemon-reload
systemctl start etcd
```

kube-apiserver

把 kube-apiserver.yaml 放到每台 Master 节点的 `/etc/kubernetes/manifests/`，并把相关的配置放到 `/srv/kubernetes/`，即可由 kubelet 自动创建并启动 apiserver：

- basic_auth.csv - basic auth user and password
- ca.crt - Certificate Authority cert
- known_tokens.csv - tokens that entities (e.g. the kubelet) can use to talk to the apiserver
- kubecfg.crt - Client certificate, public key
- kubecfg.key - Client certificate, private key
- server.cert - Server certificate, public key
- server.key - Server certificate, private key

> 注意：确保 kube-apiserver 配置 --etcd-quorum-read=true（v1.9 之后默认为 true）。

kubeadm

如果使用 kubeadm 来部署集群的话，可以按照如下步骤配置：

```
# on master0
# deploy master0
cat >config.yaml <"LOAD_BALANCER_DNS"
api:
    controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
etcd:
  local:
    extraArgs:
      listen-client-urls: "https://127.0.0.1:2379,https://CP0_IP:
      advertise-client-urls: "https://CP0_IP:2379"
      listen-peer-urls: "https://CP0_IP:2380"
      initial-advertise-peer-urls: "https://CP0_IP:2380"
      initial-cluster: "CP0_HOSTNAME=https://CP0_IP:2380"
    serverCertSANs:
      - CP0_HOSTNAME
      - CP0_IP
    peerCertSANs:
      - CP0_HOSTNAME
      - CP0_IP
networking:
    # This CIDR is a Calico default. Substitute or remove for you
    podSubnet: "192.168.0.0/16"
EOF
kubeadm init --config=config.yaml

# copy TLS certs to other master nodes
CONTROL_PLANE_IPS="10.0.0.7 10.0.0.8"
for host in ${CONTROL_PLANE_IPS}; do
    scp /etc/kubernetes/pki/ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.pub "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/etcd/ca.crt "${USER}"@$host:etcd-ca.c
    scp /etc/kubernetes/pki/etcd/ca.key "${USER}"@$host:etcd-ca.k
    scp /etc/kubernetes/admin.conf "${USER}"@$host:
done
```

```
# on other master nodes
cat > kubeadm-config.yaml <"LOAD_BALANCER_DNS"
api:
    controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
etcd:
  local:
    extraArgs:
      listen-client-urls: "https://127.0.0.1:2379,https://CP1_IP:
      advertise-client-urls: "https://CP1_IP:2379"
      listen-peer-urls: "https://CP1_IP:2380"
      initial-advertise-peer-urls: "https://CP1_IP:2380"
      initial-cluster: "CP0_HOSTNAME=https://CP0_IP:2380,CP1_HOST
      initial-cluster-state: existing
    serverCertSANs:
      - CP1_HOSTNAME
      - CP1_IP
    peerCertSANs:
      - CP1_HOSTNAME
      - CP1_IP
networking:
    # This CIDR is a calico default. Substitute or remove for you
    podSubnet: "192.168.0.0/16"
EOF
# move files
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
mv /home/${USER}/admin.conf /etc/kubernetes/admin.conf
# Run the kubeadm phase commands to bootstrap the kubelet:
kubeadm alpha phase certs all --config kubeadm-config.yaml
kubeadm alpha phase kubelet config write-to-disk --config kubeadm
kubeadm alpha phase kubelet write-env-file --config kubeadm-confi
kubeadm alpha phase kubeconfig kubelet --config kubeadm-config.ya
systemctl start kubelet
```

```
# Add the node to etcd cluster
CP0_IP=10.0.0.7
CP0_HOSTNAME=cp0
CP1_IP=10.0.0.8
CP1_HOSTNAME=cp1
KUBECONFIG=/etc/kubernetes/admin.conf kubectl exec -n kube-system
kubeadm alpha phase etcd local --config kubeadm-config.yaml
# Deploy the master components
kubeadm alpha phase kubeconfig all --config kubeadm-config.yaml
kubeadm alpha phase controlplane all --config kubeadm-config.yaml
kubeadm alpha phase mark-master --config kubeadm-config.yaml
```

kube-apiserver 启动后，还需要为它们做负载均衡，可以使用云平台的弹性负载均衡服务或者使用 haproxy/lvs 等为 master 节点配置负载均衡。

kube-controller-manager 和 kube-scheduler

kube-controller manager 和 kube-scheduler 需要保证任何时刻都只有一个实例运行，需要一个选主的过程，所以在启动时要设置 `--leader-elect=true`，比如

```
kube-scheduler --master=127.0.0.1:8080 --v=2 --leader-elect=true
kube-controller-manager --master=127.0.0.1:8080 --cluster-cidr=10
```

把 kube-scheduler.yaml 和 kube-controller-manager.yaml 放到每台 master 节点的 `/etc/kubernetes/manifests/` 即可。

kube-dns

kube-dns 可以通过 Deployment 的方式来部署，默认 kubeadm 会自动创建。但在大规模集群的时候，需要放宽资源限制，比如

```
dns_replicas: 6
dns_cpu_limit: 100m
dns_memory_limit: 512Mi
dns_cpu_requests 70m
dns_memory_requests: 70Mi
```

另外，也需要给 dnsmasq 增加资源，比如增加缓存大小到 10000，增加并发处理数量 `--dns-forward-max=1000` 等。

kube-proxy

默认 kube-proxy 使用 iptables 来为 Service 作负载均衡，这在大规模时会产生很大的 Latency，可以考虑使用 IPVS 的替代方式（注意 IPVS 在 v1.9 中还是 beta 状态）。

另外，需要注意配置 kube-proxy 使用 kube-apiserver 负载均衡的 IP 地址：

```
kubectl get configmap -n kube-system kube-proxy -o yaml > kube-pr
sed -i 's#server:.*#server: https://:6443#g' kube-proxy-cm.yaml
kubectl apply -f kube-proxy-cm.yaml --force
# restart all kube-proxy pods to ensure that they load the new cc
kubectl delete pod -n kube-system -l k8s-app=kube-proxy
```

kubelet

kubelet 需要配置 kube-apiserver 负载均衡的 IP 地址

```
sudo sed -i 's#server:.*#server: https://:6443#g' /etc/kubernetes
sudo systemctl restart kubelet
```

数据持久化

除了上面提到的这些配置，持久化存储也是高可用 Kubernetes 集群所必须的。

- 对于公有云上部署的集群，可以考虑使用云平台提供的持久化存储，比如 aws ebs 或者 gce persistent disk
- 对于物理机部署的集群，可以考虑使用 iSCSI、NFS、Gluster 或者 Ceph 等网络存储，也可以使用 RAID

参考文档

- Creating Highly Available Clusters with kubeadm
- http://kubecloud.io/setup-ha-k8s-kops/
- https://github.com/coreos/etcd/blob/master/Documentation/op-guide/clustering.md
- Kubernetes Master Tier For 1000 Nodes Scale
- Scaling Kubernetes to Support 50000 Services

调试

对于普通的服务器进程，我们可以很方便的使用宿主机上的各种工具来调试；但容器经常是仅包含必要的应用程序，一般不包含常用的调试工具，那如何在线调试容器中的进程呢？最简单的方法是再起一个新的包含了调试工具的容器。

来看一个最简单的 web 容器如何调试。

webserver 容器

用 Go 编写一个最简单的 webserver：

```
// go-examples/basic/webserver
package main

import "net/http"
import "fmt"
import "log"

func index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello World")
}

func main() {
    http.HandleFunc("/", index)
    err := http.ListenAndServe(":80", nil)
    if err != nil {

        log.Println(err)
    }
}
```

以 linux 平台方式编译

```
GOOS=linux go build -o webserver
```

然后用下面的 Docker build 一个 docker 镜像：

```
FROM scratch

COPY ./webserver /
CMD ["/webserver"]
```

```
# docker build -t feisky/hello-world .
Sending build context to Docker daemon 5.655 MB
Step 1/3 : FROM scratch
 --->
Step 2/3 : COPY ./webserver /
 ---> 184eb7c074b5
Removing intermediate container abf107844295
Step 3/3 : CMD /webserver
 ---> Running in fe9fa4841e70
 ---> dca5ec00b3e7
Removing intermediate container fe9fa4841e70
Successfully built dca5ec00b3e7
```

最后启动 webserver 容器

```
docker run -itd --name webserver -p 80:80 feisky/hello-world
```

访问映射后的 80 端口，webserver 容器正常返回 "Hello World"

```
# curl http://$(hostname):80
Hello World
```

新建一个容器调试 webserver

用一个包含调试工具或者方便安装调试工具的镜像（如 alpine）创建一个新的 container，为了便于获取 webserver 进程的状态，新的容器共享 webserver 容器的 pid namespace 和 net namespace，并增加必要的 capability：

```
docker run -it --rm --pid=container:webserver --net=container:web
/ # ps -ef
PID   USER     TIME   COMMAND
    1 root       0:00 /webserver
   13 root       0:00 sh
   18 root       0:00 ps -ef
```

这样，新的容器可以直接 attach 到 webserver 进程上来在线调试，比如 strace 到 webserver 进程

```
# 继续在刚创建的新容器 sh 中执行
/ # apk update && apk add strace
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/main/x86_64/APKIN
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/community/x86_64/
v3.5.1-34-g1d3b13bd53 [http://dl-cdn.alpinelinux.org/alpine/v3.5/
v3.5.1-29-ga981b1f149 [http://dl-cdn.alpinelinux.org/alpine/v3.5/
OK: 7958 distinct packages available
(1/1) Installing strace (4.14-r0)
Executing busybox-1.25.1-r0.trigger
OK: 5 MiB in 12 packages
/ # strace -p 1
strace: Process 1 attached
epoll_wait(4,
^Cstrace: Process 1 detached
```

也可以获取 webserver 容器的网络状态

```
# 继续在刚创建的新容器 sh 中执行
/ # apk add lsof
(1/1) Installing lsof (4.89-r0)
Executing busybox-1.25.1-r0.trigger
OK: 5 MiB in 13 packages
/ # lsof -i TCP
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
webserver   1 root    3u  IPv6  14233      0t0  TCP *:http (LISTE
```

当然，也可以访问 webserver 容器的文件系统

```
/ # ls -l /proc/1/root/
total 5524
drwxr-xr-x    5 root     root           360 Feb 14 13:16 dev
drwxr-xr-x    2 root     root          4096 Feb 14 13:16 etc
dr-xr-xr-x  128 root     root             0 Feb 14 13:16 proc
dr-xr-xr-x   13 root     root             0 Feb 14 13:16 sys
-rwxr-xr-x    1 root     root       5651357 Feb 14 13:15 webserve
```

Kubernetes 社区也在提议增加一个 `kubectl debug` 命令，用类似的方式在 Pod 中启动一个新容器来调试运行中的进程，可以参见 https://github.com/kubernetes/community/pull/649。

端口映射

在创建 Pod 时，可以指定容器的 hostPort 和 containerPort 来创建端口映射，这样可以通过 Pod 所在 Node 的 IP:hostPort 来访问服务。比如

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

注意事项

使用了 hostPort 的容器只能调度到端口不冲突的 Node 上，除非有必要（比如运行一些系统级的 daemon 服务），不建议使用端口映射功能。如果需要对外暴露服务，建议使用 NodePort Service。

端口转发

端口转发是 kubectl 的一个子命令，通过 `kubectl port-forward` 可以将本地端口转发到指定的 Pod。

Pod 端口转发

可以将本地端口转发到指定 Pod 的端口。

```
# Listen on ports 5000 and 6000 locally, forwarding data to/from
kubectl port-forward mypod 5000 6000

# Listen on port 8888 locally, forwarding to 5000 in the pod
kubectl port-forward mypod 8888:5000

# Listen on a random port locally, forwarding to 5000 in the pod
kubectl port-forward mypod :5000

# Listen on a random port locally, forwarding to 5000 in the pod
kubectl port-forward mypod 0:5000
```

服务端口转发

也可以将本地端口转发到服务、复制控制器或者部署的端口。

```
# Forward to deployment
kubectl port-forward deployment/redis-master 6379:6379

# Forward to replicaSet
kubectl port-forward rs/redis-master 6379:6379

# Forward to service
kubectl port-forward svc/redis-master 6379:6379
```

HugePage

HugePage 是 v1.9 中引入的新特性（v1.9 Alpha，v1.10 Beta），允许在容器中直接使用 Node 上的 HugePage。

配置

- `--feature-gates=HugePages=true`
- Node 节点上预先分配好 HugePage，如

```
mount -t hugetlbfs \
    -o uid=,gid=,mode=,pagesize=,size=,\
    min_size=,nr_inodes= none /mnt/huge
```

使用

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - image: fedora:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 100Mi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

注意

- HugePage 请求和限制必须相等
- HugePage 提供 Pod 级别的隔离，暂不支持容器级别的隔离
- 基于 HugePage 的 EmptyDir 存储卷仅可使用请求的 HugePage 内存
- 可以通过 ResourceQuota 限制 HugePage 的用量
- 容器应用内使用 `shmget(SHM_HUGETLB)` 获取 HugePage 时，必需配置与 `proc/sys/vm/hugetlb_shm_group` 中一致的用户组（ `securityContext.SupplementalGroups` ）

安全

Kubernetes 提供了多种机制来限制容器的行为，减少容器攻击面，保证系统安全性。

- Security Context：限制容器的行为，包括 Capabilities、ReadOnlyRootFilesystem、Privileged、RunAsNonRoot、RunAsUser 以及 SELinuxOptions 等
- Pod Security Policy：集群级的 Pod 安全策略，自动为集群内的 Pod 和 Volume 设置 Security Context
- Sysctls：允许容器设置内核参数，分为安全 Sysctls 和非安全 Sysctls
- AppArmor：限制应用的访问权限
- Seccomp：Secure computing mode 的缩写，限制容器应用可执行的系统调用

Security Context 和 Pod Security Policy

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    # Seccomp v1.11 使用 'runtime/default'，而 v1.10 及更早版本使用 'd
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'ru
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'ru
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'ru
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'ru
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalati
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin a
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor rather tha
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
```

```
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

完整参考见这里。

Sysctls

Sysctls 允许容器设置内核参数，分为安全 Sysctls 和非安全 Sysctls

- 安全 Sysctls：即设置后不影响其他 Pod 的内核选项，只作用在容器 namespace 中，默认开启。包括以下几种
  - `kernel.shm_rmid_forced`
  - `net.ipv4.ip_local_port_range`
  - `net.ipv4.tcp_syncookies`
- 非安全 Sysctls：即设置好有可能影响其他 Pod 和 Node 上其他服务的内核选项，默认禁止。如果使用，需要管理员在配置 kubelet 时开启，如 `kubelet --experimental-allowed-unsafe-sysctls 'kernel.msg*,net.ipv4.route.min_pmtu'`

Sysctls 在 v1.11 升级为 Beta 版，可以通过 PSP spec 直接设置，如

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - kernel.msg*
  forbiddenSysctls:
  - kernel.shm_rmid_forced
```

而 v1.10 及更早版本则为 Alpha 阶段，需要通过 Pod annotation 设置，如：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  annotations:
    security.alpha.kubernetes.io/sysctls: kernel.shm_rmid_forced=
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.m
spec:
  ...
```

AppArmor

AppArmor(Application Armor) 是 Linux 内核的一个安全模块，允许系统管理员将每个程序与一个安全配置文件关联，从而限制程序的功能。通过它你可以指定程序可以读、写或运行哪些文件，是否可以打开网络端口等。作为对传统 Unix 的自主访问控制模块的补充，AppArmor 提供了强制访问控制机制。

在使用 AppArmor 之前需要注意

- Kubernetes 版本 >=v1.4
- apiserver 和 kubelet 已开启 AppArmor 特性，`--feature-gates=AppArmor=true`
- 已开启 apparmor 内核模块，通过 `cat /sys/module/apparmor/parameters/enabled` 查看
- 仅支持 docker container runtime
- AppArmor profile 已经加载到内核，通过 `cat /sys/kernel/security/apparmor/profiles` 查看

AppArmor 还在 alpha 阶段，需要通过 Pod annotation `container.apparmor.security.beta.kubernetes.io/` 来设置。可选的值包括

- `runtime/default`：使用 Container Runtime 的默认配置
- `localhost/`：使用已加载到内核的 AppArmor profile

```
$ sudo apparmor_parser -q <#include

profile k8s-apparmor-example-deny-write flags=(attach_disconnecte
  #include

  file,

  # Deny all file writes.
  deny /** w,
}
EOF'

$ kubectl create -f /dev/stdin <Hello AppArmor!'&& sleep 1h"]
EOF
pod "hello-apparmor" created

$ kubectl exec hello-apparmor cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)

$ kubectl exec hello-apparmor touch /tmp/test
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with nc
```

Seccomp

Seccomp 是 Secure computing mode 的缩写，它是 Linux 内核提供的一个操作，用于限制一个进程可以执行的系统调用. Seccomp 需要有一个配置文件来指明容器进程允许和禁止执行的系统调用。

在 Kubernetes 中，需要将 seccomp 配置文件放到 `/var/lib/kubelet/seccomp` 目录中（可以通过 kubelet 选项 `--seccomp-profile-root` 修改）。比如禁止 chmod 的格式为

```
$ cat /var/lib/kubelet/seccomp/chmod.json
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "name": "chmod",
            "action": "SCMP_ACT_ERRNO"
        }
    ]
}
```

Seccomp 还在 alpha 阶段，需要通过 Pod annotation 设置，包括

- `security.alpha.kubernetes.io/seccomp/pod`：应用到该 Pod 的所有容器
- `security.alpha.kubernetes.io/seccomp/container/`：应用到指定容器

而 value 有三个选项

- `runtime/default`：使用 Container Runtime 的默认配置
- `unconfined`：允许所有系统调用
- `localhost/`：使用 Node 本地安装的 seccomp，需要放到 `/var/lib/kubelet/seccomp` 目录中

比如使用刚才创建的 seccomp 配置：

```
apiVersion: v1
kind: Pod
metadata:
  name: trustworthy-pod
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: localhost/chmod
spec:
  containers:
    - name: trustworthy-container
      image: sotrustworthy:latest
```

kube-bench

kube-bench 提供了一个简单的工具来检查 Kubernetes 的配置（包括 master 和 node）是否符合最佳的安全实践（基于 CIS Kubernetes Benchmark）。

推荐所有生产环境的 Kubernetes 集群定期运行 kube-bench，保证集群配置符合最佳的安全实践。

安装 `kube-bench`：

```
$ docker run --rm -v `pwd`:/host aquasec/kube-bench:latest instal
$ ./kube-bench
```

当然，kube-bench 也可以直接在容器内运行，比如通常对 Master 和 Node 的检查命令分别为：

```
# Run master check
$ kubectl run --rm -i -t kube-bench-master --image=aquasec/kube-b

# Run node check
kubectl run --rm -i -t kube-bench-node --image=aquasec/kube-bench
```

镜像安全

Clair 是 CoreOS 开源的容器安全工具，用来静态分析镜像中潜在的安全问题。推荐将 Clair 集成到 Devops 流程中，自动对所有镜像进行安全扫描。

安装 Clair 的方法为：

```
git clone https://github.com/coreos/clair
cd clair/contrib/helm
helm dependency update clair
helm install clair
```

Clair 项目本身只提供了 API，在实际使用中还需要一个客户端（或集成 Clair的服务）配合使用。比如，使用 reg 的方法为

```
# Install
$ go get github.com/genuinetools/reg

# Vulnerability Reports
$ reg vulns --clair https://clair.j3ss.co r.j3ss.co/chrome

# Generating Static Website for a Registry
$ $ reg server --clair https://clair.j3ss.co
```

其他安全工具

开源产品：

- falco：容器运行时安全行为监控工具。
- docker-bench-security：Docker 环境安全检查工具。
- kube-hunter：Kubernetes 集群渗透测试工具。

商业产品

- Twistlock
- Aqua Container Security Platform
- Sysdig Secure

参考文档

- Securing a Kubernetes cluster
- kube-bench

审计

Kubernetes 审计（Audit）提供了安全相关的时序操作记录，支持日志和 webhook 两种格式，并可以通过审计策略自定义事件类型。

审计日志

通过配置 kube-apiserver 的下列参数开启审计日志

- audit-log-path：审计日志路径

- audit-log-maxage：旧日志最长保留天数
- audit-log-maxbackup：旧日志文件最多保留个数
- audit-log-maxsize：日志文件最大大小（单位 MB），超过后自动做轮转（默认为 100MB）

每条审计记录包括两行

- 请求行包括：唯一 ID 和请求的元数据（如源 IP、用户名、请求资源等）
- 响应行包括：唯一 ID（与请求 ID 一致）和响应的元数据（如 HTTP 状态码）

比如，admin 用户查询默认 namespace 的 Pod 列表的审计日志格式为

```
2017-03-21T03:57:09.106841886-04:00 AUDIT: id="c939d2a7-1c37-4ef1
2017-03-21T03:57:09.108403639-04:00 AUDIT: id="c939d2a7-1c37-4ef1
```

审计策略

v1.7 + 支持实验性的高级审计特性，可以自定义审计策略（选择记录哪些事件）和审计存储后端（日志和 webhook）等。开启方法为

```
kube-apiserver ... --feature-gates=AdvancedAuditing=true
```

注意开启 AdvancedAuditing 后，日志的格式有一些修改，如新增了 stage 字段（包括 RequestReceived，ResponseStarted ，ResponseComplete，Panic 等）。

审计策略

审计策略选择记录哪些事件，设置方法为

```
kube-apiserver ... --audit-policy-file=/etc/kubernetes/audit-poli
```

其中，设计策略的配置格式为

```
rules:
  # Don't log watch requests by the"system:kube-proxy" on endpoir
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs:
    - "/api*" # Wildcard matching.
    - "/version"

  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
    - group: "" # core API group
      resources: ["configmaps"]
    # This rule only applies to resources in the "kube-system" na
    # The empty string "" can be used to select non-namespaced re
    namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at t
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]

  # Log all other resources in core and extensions at the Request
  - level: Request
    resources:
    - group: "" # core API group
    - group: "extensions" # Version of group should NOT be includ

  # A catch-all rule to log all other requests at the Metadata le
  - level: Metadata
```

在生产环境中，推荐参考 GCE 审计策略 配置。

审计存储后端

审计存储后端支持两种方式

- 日志，配置 `--audit-log-path` 开启，格式为

```
2017-06-15T21:50:50.259470834Z AUDIT: id="591e9fde-6a98-46f6-b7bc
2017-06-15T21:50:50.259470834Z AUDIT: id="591e9fde-6a98-46f6-b7bc
```

- webhook，配置 `--audit-webhook-config-file=/etc/kubernetes/audit-webhook-kubeconfig --audit-webhook-mode=batch` 开启，其中 audit-webhook-mode 支持 batch 和 blocking 两种格式，而 webhook 配置文件格式为

```
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-audit-service
    cluster:
      certificate-authority: /path/to/ca.pem  # CA for verifying
      server: https://audit.example.com/audit # URL of remote ser

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhoc
      client-key: /path/to/key.pem          # key matching the ce

# kubeconfig files require a context. Provide one for the API ser
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-audit-service
    user: name-of-api-sever
  name: webhook
```

所有的事件以 JSON 格式 POST 给 webhook server，如

```
{
  "kind": "EventList",
  "apiVersion": "audit.k8s.io/v1alpha1",
  "items": [
    {
      "metadata": {
        "creationTimestamp": null
      },
      "level": "Metadata",
      "timestamp": "2017-06-15T23:07:40Z",
      "auditID": "4faf711a-9094-400f-a876-d9188ceda548",
      "stage": "ResponseComplete",
      "requestURI": "/apis/rbac.authorization.k8s.io/v1beta1/name
      "verb": "get",
      "user": {
        "username": "system:apiserver",
        "uid": "97a62906-e4d7-4048-8eda-4f0fb6ff8f1e",
        "groups": [
          "system:masters"
        ]
      },
      "sourceIPs": [
        "127.0.0.1"
      ],
      "objectRef": {
        "resource": "rolebindings",
        "namespace": "kube-public",
        "name": "system:controller:bootstrap-signer",
        "apiVersion": "rbac.authorization.k8s.io/v1beta1"
      },
      "responseStatus": {
        "metadata": {},
        "code": 200
      }
    }
  ]
}
```

大规模集群

Kubernetes v1.6-v1.11 单集群最大支持 5000 个节点，也就是说 Kubernetes 最新稳定版的单个集群支持

- 不超过 5000 个节点
- 不超过 150000 个 Pod
- 不超过 300000 个容器
- 每台 Node 上不超过 100 个 Pod

公有云配额

对于公有云上的 Kubernetes 集群，规模大了之后很容器碰到配额问题，需要提前在云平台上增大配额。这些需要增大的配额包括

- 虚拟机个数
- vCPU 个数
- 内网 IP 地址个数
- 公网 IP 地址个数
- 安全组条数
- 路由表条数
- 持久化存储大小

Etcd 存储

除了常规的 Etcd 高可用集群配置、使用 SSD 存储等，还需要为 Events 配置单独的 Etcd 集群。即部署两套独立的 Etcd 集群，并配置 kube-apiserver

```
--etcd-servers="http://etcd1:2379,http://etcd2:2379,http://etcd3:
```

另外，Etcd 默认存储限制为 2GB，可以通过 `--quota-backend-bytes` 选项增大。

Master 节点大小

可以参考 AWS 配置 Master 节点的大小：

- 1-5 nodes: m3.medium
- 6-10 nodes: m3.large
- 11-100 nodes: m3.xlarge
- 101-250 nodes: m3.2xlarge
- 251-500 nodes: c4.4xlarge
- more than 500 nodes: c4.8xlarge

为扩展分配更多资源

Kubernetes 集群内的扩展也需要分配更多的资源，包括为这些 Pod 分配更大的 CPU 和内存以及增大容器副本数量等。当 Node 本身的容量太小时，还需要增大 Node 本身的 CPU 和内存（特别是在公有云平台上）。

以下扩展服务需要增大 CPU 和内存：

- DNS (kube-dns or CoreDNS)
- InfluxDB and Grafana
- Kibana
- FluentD with ElasticSearch Plugin
- FluentD with GCP Plugin

以下扩展服务需要增大副本数：

- elasticsearch
- DNS (kube-dns or CoreDNS)

另外，为了保证多个副本分散调度到不同的 Node 上，需要为容器配置 AntiAffinity。比如，对 kube-dns，可以增加如下的配置：

```
affinity:
 podAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - weight: 100
     labelSelector:
       matchExpressions:
       - key: k8s-app
         operator: In
         values:
         - kube-dns
     topologyKey: kubernetes.io/hostname
```

Kube-apiserver 配置

- 设置 `--max-requests-inflight=3000`
- 设置 `--max-mutating-requests-inflight=1000`

Kube-scheduler 配置

- 设置 `--kube-api-qps=100`

Kube-controller-manager 配置

- 设置 `--kube-api-qps=100`
- 设置 `--kube-api-burst=100`

Kubelet 配置

- 设置 `--image-pull-progress-deadline=30m`
- 设置 `--serialize-image-pulls=false`（需要 Docker 使用 overlay2 ）
- Kubelet 单节点允许运行的最大 Pod 数：`--max-pods=110`（默认是 110，可以根据实际需要设置）

Docker 配置

- 设置 `max-concurrent-downloads=10`
- 使用 SSD 存储 `graph=/ssd-storage-path`
- 预加载 pause 镜像，比如 `docker image save -o /opt/preloaded_docker_images.tar` 和 `docker image load -i /opt/preloaded_docker_images.tar`

节点配置

增大内核选项配置 `/etc/sysctl.conf`：

```
fs.file-max=1000000

net.ipv4.ip_forward=1
net.netfilter.nf_conntrack_max=10485760
net.netfilter.nf_conntrack_tcp_timeout_established=300
net.netfilter.nf_conntrack_buckets=655360
net.core.netdev_max_backlog=10000

net.ipv4.neigh.default.gc_thresh1=1024
net.ipv4.neigh.default.gc_thresh2=4096
net.ipv4.neigh.default.gc_thresh3=8192

net.netfilter.nf_conntrack_max=10485760
net.netfilter.nf_conntrack_tcp_timeout_established=300
net.netfilter.nf_conntrack_buckets=655360
net.core.netdev_max_backlog=10000

fs.inotify.max_user_instances=524288
fs.inotify.max_user_watches=524288
```

应用配置

在运行 Pod 的时候也需要注意遵循一些最佳实践，比如

- 为容器设置资源请求和限制
  - `spec.containers[].resources.limits.cpu`
  - `spec.containers[].resources.limits.memory`
  - `spec.containers[].resources.requests.cpu`
  - `spec.containers[].resources.requests.memory`
  - `spec.containers[].resources.limits.ephemeral-storage`
  - `spec.containers[].resources.requests.ephemeral-storage`
- 对关键应用使用 PodDisruptionBudget、nodeAffinity、podAffinity 和 podAntiAffinity 等保护
- 尽量使用控制器来管理容器（如 Deployment、StatefulSet、DaemonSet、Job 等）
- 更多内容参考这里

必要的扩展

监控、告警以及可视化（如 Prometheus 和 Grafana）至关重要，推荐部署并开启。

参考文档

- Building Large Clusters
- Scaling Kubernetes to 2,500 Nodes
- Scaling Kubernetes for 25M users

大数据与机器学习

Kubernetes 在大数据与机器学习中的实践案例。

Spark

Kubernetes 从 v1.8 开始支持 原生的 Apache Spark 应用（需要 Spark 支持 Kubernetes，比如 v2.3），可以通过 `spark-submit` 命令直接提交 Kubernetes 任务。比如计算圆周率

```
bin/spark-submit \
  --deploy-mode cluster \
  --class org.apache.spark.examples.SparkPi \
  --master k8s://https://: \
  --kubernetes-namespace default \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=kubespark/spark-dri
  --conf spark.kubernetes.executor.docker.image=kubespark/spark-e
  local:///opt/spark/examples/jars/spark-examples_2.11-2.2.0-k8s-
```

或者使用 Python 版本

```
bin/spark-submit \
  --deploy-mode cluster \
  --master k8s://https://: \
  --kubernetes-namespace  \
  --conf spark.executor.instances=5 \
  --conf spark.app.name=spark-pi \
  --conf spark.kubernetes.driver.docker.image=kubespark/spark-dri
  --conf spark.kubernetes.executor.docker.image=kubespark/spark-e
  --jars local:///opt/spark/examples/jars/spark-examples_2.11-2.2
  --py-files local:///opt/spark/examples/src/main/python/sort.py
  local:///opt/spark/examples/src/main/python/pi.py 10
```

Spark on Kubernetes 部署

Kubernetes 示例 github 上提供了一个详细的 spark 部署方法，由于步骤复杂，这里简化一些部分让大家安装的时候不用去多设定一些东西。

部署条件

- 一个 kubernetes 群集，可参考 集群部署
- kube-dns 正常运作

创建一个命名空间

namespace-spark-cluster.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: "spark-cluster"
  labels:
    name: "spark-cluster"
```

```
$ kubectl create -f examples/staging/spark/namespace-spark-cluste
```

这边原文提到需要将 kubectl 的执行环境转到 spark-cluster，这边为了方便我们不这样做，而是将之后的佈署命名空间都加入 spark-cluster

部署 Master Service

建立一个 replication controller，来运行 Spark Master 服务

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: spark-master-controller
  namespace: spark-cluster
spec:
  replicas: 1
  selector:
    component: spark-master
  template:
    metadata:
      labels:
        component: spark-master
    spec:
      containers:
        - name: spark-master
          image: gcr.io/google_containers/spark:1.5.2_v1
          command: ["/start-master"]
          ports:
            - containerPort: 7077
            - containerPort: 8080
          resources:
            requests:
              cpu: 100m
```

```
$ kubectl create -f spark-master-controller.yaml
```

创建 master 服务

spark-master-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: spark-master
  namespace: spark-cluster
spec:
  ports:
    - port: 7077
      targetPort: 7077
      name: spark
    - port: 8080
      targetPort: 8080
      name: http
  selector:
    component: spark-master
```

```
$ kubectl create -f spark-master-service.yaml
```

检查 Master 是否正常运行

```
$ kubectl get pod -n spark-cluster
spark-master-controller-qtwm8     1/1       Running   0
```

```
$ kubectl logs spark-master-controller-qtwm8 -n spark-cluster
17/08/07 02:34:54 INFO Master: Registered signal handlers for [TE
17/08/07 02:34:54 INFO SecurityManager: Changing view acls to: ro
17/08/07 02:34:54 INFO SecurityManager: Changing modify acls to:
17/08/07 02:34:54 INFO SecurityManager: SecurityManager: authenti
17/08/07 02:34:55 INFO Slf4jLogger: Slf4jLogger started
17/08/07 02:34:55 INFO Remoting: Starting remoting
17/08/07 02:34:55 INFO Remoting: Remoting started; listening on a
17/08/07 02:34:55 INFO Utils: Successfully started service 'spark
17/08/07 02:34:55 INFO Master: Starting Spark master at spark://s
17/08/07 02:34:55 INFO Master: Running Spark version 1.5.2
17/08/07 02:34:56 INFO Utils: Successfully started service 'Maste
17/08/07 02:34:56 INFO MasterWebUI: Started MasterWebUI at http:/
17/08/07 02:34:56 INFO Utils: Successfully started service on por
17/08/07 02:34:56 INFO StandaloneRestServer: Started REST server
17/08/07 02:34:56 INFO Master: I have been elected leader! New st
```

若 master 已经被建立与运行，我们可以透过 Spark 开发的 webUI 来察看我们 spark 的群集状况，我们将佈署 specialized proxy

spark-ui-proxy-controller.yaml

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: spark-ui-proxy-controller
  namespace: spark-cluster
spec:
  replicas: 1
  selector:
    component: spark-ui-proxy
  template:
    metadata:
      labels:
        component: spark-ui-proxy
    spec:
      containers:
        - name: spark-ui-proxy
          image: elsonrodriguez/spark-ui-proxy:1.0
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: 100m
          args:
            - spark-master:8080
          livenessProbe:
              httpGet:
                path: /
                port: 80
              initialDelaySeconds: 120
              timeoutSeconds: 5
```

```
$ kubectl create -f spark-ui-proxy-controller.yaml
```

提供一个 service 做存取，这边原文是使用 LoadBalancer type，这边我们改成 NodePort，如果你的 kubernetes 运行环境是在 cloud provider，也可以参考原文作法

spark-ui-proxy-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: spark-ui-proxy
  namespace: spark-cluster
spec:
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
  selector:
    component: spark-ui-proxy
  type: NodePort
```

```
$ kubectl create -f spark-ui-proxy-service.yaml
```

部署完后你可以利用 `kubecrl proxy` 来察看你的 Spark 群集状态

```
$ kubectl proxy --port=8001
```

可以透过 `http://localhost:8001/api/v1/proxy/namespaces/spark-cluster/services/spark-master:8080/` 察看，若 kubectl 中断就无法这样观察了，但我们再先前有设定 nodeport 所以也可以透过任意台 node 的端口 30080 去察看（例如 `http://10.201.2.34:30080`）。

部署 Spark workers

要先确定 Matser 是再运行的状态

spark-worker-controller.yaml

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: spark-worker-controller
  namespace: spark-cluster
spec:
  replicas: 2
  selector:
    component: spark-worker
  template:
    metadata:
      labels:
        component: spark-worker
    spec:
      containers:
        - name: spark-worker
          image: gcr.io/google_containers/spark:1.5.2_v1
          command: ["/start-worker"]
          ports:
            - containerPort: 8081
          resources:
            requests:
              cpu: 100m
```

```
$ kubectl create -f spark-worker-controller.yaml
replicationcontroller "spark-worker-controller" created
```

透过指令察看运行状况

```
$ kubectl get pod -n spark-cluster
spark-master-controller-qtwm8     1/1       Running   0
spark-worker-controller-4rxrs     1/1       Running   0
spark-worker-controller-z6f21     1/1       Running   0
spark-ui-proxy-controller-d4br2   1/1       Running   4
```

也可以透过上面建立的 WebUI 服务去察看

基本上到这边 Spark 的群集已经建立完成了

创建 Zeppelin UI

我们可以利用 Zeppelin UI 经由 web notebook 直接去执行我们的任务，详情可以看 Zeppelin UI 与 Spark architecture

zeppelin-controller.yaml

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: zeppelin-controller
  namespace: spark-cluster
spec:
  replicas: 1
  selector:
    component: zeppelin
  template:
    metadata:
      labels:
        component: zeppelin
    spec:
      containers:
        - name: zeppelin
          image: gcr.io/google_containers/zeppelin:v0.5.6_v1
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: 100m
```

```
$ kubectl create -f zeppelin-controller.yaml
replicationcontroller "zeppelin-controller" created
```

然后一样佈署 Service

zeppelin-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: zeppelin
  namespace: spark-cluster
spec:
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30081
  selector:
    component: zeppelin
  type: NodePort
```

```
$ kubectl create -f zeppelin-service.yaml
```

可以看到我们把 NodePort 设再 30081，一样可以透过任意台 node 的 30081 port 访问 zeppelin UI。

通过命令行访问 pyspark（记得把 pod 名字换成你自己的）：

```
$ kubectl exec -it zeppelin-controller-8f14f -n spark-cluster pys
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more informa
17/08/14 01:59:22 WARN Utils: Service 'SparkUI' could not bind or
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/

Using Python version 2.7.9 (default, Mar  1 2015 12:57:24)
SparkContext available as sc, HiveContext available as sqlContext
>>>
```

接着就能使用 Spark 的服务了，如有错误欢迎更正。

zeppelin 常见问题

- zeppelin 的镜像非常大，所以再 pull 时会花上一些时间，而 size 大小的问题现在也正在解决中，详情可参考 issue #17231
- 在 GKE 的平台上，`kubectl post-forward` 可能有些不稳定，如果你看现 zeppelin 的状态为 `Disconnected`，`port-forward` 可能已经失败你需要去重新启动它，详情可参考 #12179

参考文档

- Apache Spark on Kubernetes
- https://github.com/kweisamx/spark-on-kubernetes
- Spark examples

Tensorflow

Kubeflow 是 Google 发布的用于在 Kubernetes 集群中部署和管理 tensorflow 任务的框架。主要功能包括

- 用于管理 Jupyter 的 JupyterHub 服务
- 用于管理训练任务的 Tensorflow Training Controller
- 用于模型服务的 TF Serving 容器

部署

部署之前需要确保

- 一套部署好的 Kubernetes 集群或者 Minikube，并配置好 kubectl 命令行工具
- 安装 ksonnet 0.8.0 以上版本

对于开启 RBAC 的 Kubernetes 集群，首先要创建管理员角色绑定：

```
kubectl create clusterrolebinding tf-admin --clusterrole=cluster-
```

然后运行以下命令部署

```
ks init my-kubeflow
cd my-kubeflow
ks registry add kubeflow github.com/google/kubeflow/tree/master/k
ks pkg install kubeflow/core
ks pkg install kubeflow/tf-serving
ks pkg install kubeflow/tf-job
ks generate core kubeflow-core --name=kubeflow-core
ks apply default -c kubeflow-core
```

如果有多个 Kubernetes 集群，也可以切换到其他其集群中部署，如

```
kubectl config use-context gke
ks env add gke
ks apply gke -c kubeflow-core
```

稍等一会，就可以看到 `tf-hub-lb` 服务的公网IP，也就是 JupyterHub 的访问地址

```
kubectl get svc tf-hub-lb
```

对于不支持 LoadBalancer Service 的集群，还可以通过端口转发（`http://127.0.0.1:8100`）的方式来访问：

```
kubectl port-forward tf-hub-0 8100:8000
```

JupyterHub 默认可以用任意用户名和密码登录。登陆后，可以使用自定义镜像来启动 Notebook Server，比如使用

- `gcr.io/kubeflow/tensorflow-notebook-cpu`
- `gcr.io/kubeflow/tensorflow-notebook-gpu`

训练示例

使用 CPU：

```
ks generate tf-cnn cnn --name=cnn
ks apply gke -c cnn
```

使用 GPU：

```
ks param set cnn num_gpus 1
ks param set  cnn num_workers 1
ks apply default -c cnn
```

模型部署

```
MODEL_COMPONENT=serveInception
MODEL_NAME=inception
MODEL_PATH=gs://cloud-ml-dev_jlewi/tmp/inception
ks generate tf-serving ${MODEL_COMPONENT} --name=${MODEL_NAME} --

ks apply gke -c ${MODEL_COMPONENT}
```

参考文档

- Introducing Kubeflow - A Composable, Portable, Scalable ML Stack Built for Kubernetes
- https://github.com/google/kubeflow

Serverless

Serverless，即无服务器架构，将大家从服务器中解放了出来，只需要关注业务逻辑本身。用户只需要关注数据和业务逻辑，无需维护服务器，也不需要关心系统的容量和扩容。Serverless 本质上是一种更简单易用的 PaaS，包含两种含义：

仅依赖云端服务来管理业务逻辑和状态的应用或服务，一般称为 BaaS (Backend as a Service) 事件驱动且短时执行应用或服务，其主要逻辑由开发者完成，但由第三方管理（比如 AWS Lambda），一般称为 FaaS (Function as a Service)。目前大火的 Serverless 一般是指 FaaS。

引入 serverless 可以给应用开发者带来明显的好处

- 用户无需配置和管理服务器
- 用户服务不需要基于特定框架或软件库

- 部署简单，只需要将代码上传到 serverless 平台即可
- 完全自动化的横向扩展
- 事件触发，比如 http 请求触发、文件更新触发、时间触发、消息触发等
- 低成本，比如 AWS Lambda 按执行时间和触发次数收费，在代码未运行时无需付费

当然，serverless 也并非银弹，也有其特有的局限性

- 无状态，服务的任何进程内或主机状态均无法被后续调用所使用，需要借助外部数据库或网络存储管理状态
- 每次函数调用的时间一般都有限制，比如 AWS Lambda 限制每个函数最长只能运行 5 分钟
- 启动延迟，特别是应用不活跃或者突发流量的情况下延迟尤为明显
- 平台依赖，比如服务发现、监控、调试、API Gateway 等都依赖于 serverless 平台提供的功能

开源框架

- OpenFaas: https://github.com/openfaas/faas
- Fission: https://github.com/fission/fission
- Kubeless: https://github.com/kubeless/kubeless
- OpenWhisk: https://github.com/apache/incubator-openwhisk
- Fn: https://fnproject.io/

商业产品

- AWS Lambda: http://docs.aws.amazon.com/lambda/latest/dg/welcome.html
- AWS Fargate: https://aws.amazon.com/cn/fargate/
- Azure Container Instance (ACI): https://azure.microsoft.com/zh-cn/services/container-instances/
- Azure Functions: https://azure.microsoft.com/zh-cn/services/functions/
- Google Cloud Functions: https://cloud.google.com/functions/
- Hyper: https://hyper.sh/

- Huawei CCI: https://www.huaweicloud.com/product/cci.html
- Aliyun Serverless Kubernetes: https://help.aliyun.com/document_detail/71480.html

很多商业产品也可以与 Kubernetes 进行无缝集成，即通过 Virtual Kubelet 将商业 Serverless 产品（如 ACI 和 Fargate等）作为 Kubernetes 集群的一个无限 Node 使用，这样就无需考虑 Node 数量的问题。

参考文档

- Awesome Serverless
- AWS Lambda
- Serverless Architectures
- TNS Guide to Serverless Technologies
- Serverless blogs and posts

排错指南

排错概览

Kubernetes 集群以及应用排错的一般方法，主要包括

- 集群状态异常排错
- Pod运行异常排错
- 网络异常排错
- 持久化存储异常排错
  - AzureDisk 排错
  - AzureFile 排错
- Windows容器排错
- 云平台异常排错
  - Azure 排错
- 常用排错工具

在排错过程中，`kubectl` 是最重要的工具，通常也是定位错误的起点。这里也列出一些常用的命令，在后续的各种排错过程中都会经常用到。

查看 Pod 状态以及运行节点

```
kubectl get pods –o wide
kubectl –n kube–system get pods –o wide
```

查看 Pod 事件

```
kubectl describe pod
```

查看 Node 状态

```
kubectl get nodes
kubectl describe node
```

kube-apiserver 日志

```
PODNAME=$(kubectl –n kube–system get pod –l component=kube–apiser
kubectl –n kube–system logs $PODNAME ––tail 100
```

以上命令操作假设控制平面以 Kubernetes 静态 Pod 的形式来运行。如果 kube-apiserver 是用 systemd 管理的，则需要登录到 master 节点上，然后使用 journalctl -u kube-apiserver 查看其日志。

kube-controller-manager 日志

```
PODNAME=$(kubectl –n kube–system get pod –l component=kube–contro
kubectl –n kube–system logs $PODNAME ––tail 100
```

以上命令操作假设控制平面以 Kubernetes 静态 Pod 的形式来运行。如果 kube-controller-manager 是用 systemd 管理的，则需要登录到 master 节点上，然后使用 journalctl -u kube-controller-manager 查看其日志。

kube-scheduler 日志

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-schedu
kubectl -n kube-system logs $PODNAME --tail 100
```

以上命令操作假设控制平面以 Kubernetes 静态 Pod 的形式来运行。如果 kube-scheduler 是用 systemd 管理的，则需要登录到 master 节点上，然后使用 journalctl -u kube-scheduler 查看其日志。

kube-dns 日志

kube-dns 通常以 Addon 的方式部署，每个 Pod 包含三个容器，最关键的是 kubedns 容器的日志：

```
PODNAME=$(kubectl -n kube-system get pod -l k8s-app=kube-dns -o j
kubectl -n kube-system logs $PODNAME -c kubedns
```

Kubelet 日志

Kubelet 通常以 systemd 管理。查看 Kubelet 日志需要首先 SSH 登录到 Node 上。

```
journalctl -l -u kubelet
```

Kube-proxy 日志

Kube-proxy 通常以 DaemonSet 的方式部署，可以直接用 kubectl 查询其日志

```
$ kubectl -n kube-system get pod -l component=kube-proxy
NAME               READY     STATUS    RESTARTS   AGE
kube-proxy-42zpn   1/1       Running   0          1d
kube-proxy-7gd4p   1/1       Running   0          3d
kube-proxy-87dbs   1/1       Running   0          4d
$ kubectl -n kube-system logs kube-proxy-42zpn
```

集群排错

本章介绍集群状态异常的排错方法，包括 Kubernetes 主要组件以及必备扩展（如 kube-dns）等，而有关网络的异常排错请参考网络异常排错方法。

概述

排查集群状态异常问题通常从 Node 和 Kubernetes 服务 的状态出发，定位出具体的异常服务，再进而寻找解决方法。集群状态异常可能的原因比较多，常见的有

- 虚拟机或物理机宕机
- 网络分区
- Kubernetes 服务未正常启动
- 数据丢失或持久化存储不可用（一般在公有云或私有云平台中）
- 操作失误（如配置错误）

按照不同的组件来说，具体的原因可能包括

- kube-apiserver 无法启动会导致
  - 集群不可访问
  - 已有的 Pod 和服务正常运行（依赖于 Kubernetes API 的除外）
- etcd 集群异常会导致
  - kube-apiserver 无法正常读写集群状态，进而导致 Kubernetes API 访问出错
  - kubelet 无法周期性更新状态
- kube-controller-manager/kube-scheduler 异常会导致
  - 复制控制器、节点控制器、云服务控制器等无法工作，从而导致 Deployment、Service 等无法工作，也无法注册新的 Node 到集群中来
  - 新创建的 Pod 无法调度（总是 Pending 状态）
- Node 本身宕机或者 Kubelet 无法启动会导致
  - Node 上面的 Pod 无法正常运行
  - 已在运行的 Pod 无法正常终止
- 网络分区会导致 Kubelet 等与控制平面通信异常以及 Pod 之间通信异常

为了维持集群的健康状态，推荐在部署集群时就考虑以下

- 在云平台上开启 VM 的自动重启功能

- 为 Etcd 配置多节点高可用集群，使用持久化存储（如 AWS EBS 等），定期备份数据
- 为控制平面配置高可用，比如多 kube-apiserver 负载均衡以及多节点运行 kube-controller-manager、kube-scheduler 以及 kube-dns 等
- 尽量使用复制控制器和 Service，而不是直接管理 Pod
- 跨地域的多 Kubernetes 集群

查看 Node 状态

一般来说，可以首先查看 Node 的状态，确认 Node 本身是不是 Ready 状态

```
kubectl get nodes
kubectl describe node
```

如果是 NotReady 状态，则可以执行 `kubectl describe node` 命令来查看当前 Node 的事件。这些事件通常都会有助于排查 Node 发生的问题。

SSH 登录 Node

在排查 Kubernetes 问题时，通常需要 SSH 登录到具体的 Node 上面查看 kubelet、docker、iptables 等的状态和日志。在使用云平台时，可以给相应的 VM 绑定一个公网 IP；而在物理机部署时，可以通过路由器上的端口映射来访问。但更简单的方法是使用 SSH Pod （不要忘记替换成你自己的 nodeName）：

```
# cat ssh.yaml
apiVersion: v1
kind: Service
metadata:
  name: ssh
spec:
  selector:
    app: ssh
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 22
    targetPort: 22
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ssh
  labels:
    app: ssh
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ssh
  template:
    metadata:
      labels:
        app: ssh
    spec:
      containers:
      - name: alpine
        image: alpine
        ports:
        - containerPort: 22
        stdin: true
        tty: true
      hostNetwork: true
      nodeName: >
```

```
$ kubectl create -f ssh.yaml
$ kubectl get svc ssh
NAME      TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)
ssh       LoadBalancer   10.0.99.149   52.52.52.52   22:32008/TCP
```

接着，就可以通过 ssh 服务的外网 IP 来登录 Node，如 `ssh user@52.52.52.52` 。

在使用完后， 不要忘记删除 SSH 服务 `kubectl delete -f ssh.yaml` 。

查看日志

一般来说，Kubernetes 的主要组件有两种部署方法

- 直接使用 systemd 等启动控制节点的各个服务
- 使用 Static Pod 来管理和启动控制节点的各个服务

使用 systemd 等管理控制节点服务时，查看日志必须要首先 SSH 登录到机器上，然后查看具体的日志文件。如

```
journalctl -l -u kube-apiserver
journalctl -l -u kube-controller-manager
journalctl -l -u kube-scheduler
journalctl -l -u kubelet
journalctl -l -u kube-proxy
```

或者直接查看日志文件

- /var/log/kube-apiserver.log
- /var/log/kube-scheduler.log
- /var/log/kube-controller-manager.log
- /var/log/kubelet.log
- /var/log/kube-proxy.log

而对于使用 Static Pod 部署集群控制平面服务的场景，可以参考下面这些查看日志的方法。

kube-apiserver 日志

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-apiser
kubectl -n kube-system logs $PODNAME --tail 100
```

kube-controller-manager 日志

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-contro
kubectl -n kube-system logs $PODNAME --tail 100
```

kube-scheduler 日志

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-schedu
kubectl -n kube-system logs $PODNAME --tail 100
```

kube-dns 日志

```
PODNAME=$(kubectl -n kube-system get pod -l k8s-app=kube-dns -o j
kubectl -n kube-system logs $PODNAME -c kubedns
```

Kubelet 日志

查看 Kubelet 日志需要首先 SSH 登录到 Node 上。

```
journalctl -l -u kubelet
```

Kube-proxy 日志

Kube-proxy 通常以 DaemonSet 的方式部署

```
$ kubectl -n kube-system get pod -l component=kube-proxy
NAME               READY     STATUS    RESTARTS   AGE
kube-proxy-42zpn   1/1       Running   0          1d
kube-proxy-7gd4p   1/1       Running   0          3d
kube-proxy-87dbs   1/1       Running   0          4d
$ kubectl -n kube-system logs kube-proxy-42zpn
```

Kube-dns/Dashboard CrashLoopBackOff

由于 Dashboard 依赖于 kube-dns，所以这个问题一般是由于 kube-dns 无法正常启动导致的。查看 kube-dns 的日志

```
$ kubectl logs --namespace=kube-system $(kubectl get pods --names
$ kubectl logs --namespace=kube-system $(kubectl get pods --names
$ kubectl logs --namespace=kube-system $(kubectl get pods --names
```

可以发现如下的错误日志

```
Waiting for services and endpoints to be initialized from apiserv
skydns: failure to forward request "read udp 10.240.0.18:47848->1
Timeout waiting for initialization
```

这说明 kube-dns pod 无法转发 DNS 请求到上游 DNS 服务器。解决方法为

- 如果使用的 Docker 版本大于 1.12，则在每个 Node 上面运行 `iptables -P FORWARD ACCEPT` 开启 Docker 容器的 IP 转发
- 等待一段时间，如果还未恢复，则检查 Node 网络是否正确配置，比如是否可以正常请求上游DNS服务器、是否开启了 IP 转发（包括 Node 内部和公有云上虚拟网卡等）、是否有安全组禁止了 DNS 请求等

如果错误日志中不是转发 DNS 请求超时，而是访问 kube-apiserver 超时，比如

```
E0122 06:56:04.774977       1 reflector.go:199] k8s.io/dns/vendor
I0122 06:56:04.775358       1 dns.go:174] Waiting for services ar
E0122 06:56:04.775574       1 reflector.go:199] k8s.io/dns/vendor
I0122 06:56:05.275295       1 dns.go:174] Waiting for services ar
I0122 06:56:05.775182       1 dns.go:174] Waiting for services ar
I0122 06:56:06.275288       1 dns.go:174] Waiting for services ar
```

这说明 Pod 网络（一般是多主机之间）访问异常，包括 Pod->Node、Node->Pod 以及 Node-Node 等之间的往来通信异常。可能的原因比较多，具体的排错方法可以参考网络异常排错指南。

Node NotReady

Node 处于 NotReady 状态，社区 issue #45419。

NotReady 的原因比较多，在排查时最重要的就是执行 `kubectl describe node` 并查看 Kubelet 日志中的错误信息。常见问题的修复方法为：

- CNI 网络插件未部署：部署 CNI 插件。
- Docker 僵死（API 不响应）：重启 Docker。
- 磁盘空间不足：清理磁盘空间，比如镜像、临时文件等。

Kubelet: failed to initialize top level QOS containers

重启 kubelet 时报错 `Failed to start ContainerManager failed to initialise top level QOS containers` （参考 #43856），临时解决方法是：

1. 在 docker.service 配置中增加 `--exec-opt native.cgroupdriver=systemd` 选项。
2. 重启主机

该问题已于2017年4月27日修复（v1.7.0+， #44940）。更新集群到新版本即可解决这个问题。

Kubelet 一直报 FailedNodeAllocatableEnforcement 事件

当 NodeAllocatable 特性未开启时（即 kubelet 设置了 `--cgroups-per-qos=false` ），查看 node 的事件会发现每分钟都会有 `Failed to update Node Allocatable Limits` 的警告信息：

```
$ kubectl describe node node1
Events:
  Type     Reason                             Age
  ----     ------                             ----
  Warning  FailedNodeAllocatableEnforcement  2m (x1001 over 16h)
```

如果 NodeAllocatable 特性确实不需要，那么该警告事件可以忽略。但根据 Kubernetes 文档 Reserve Compute Resources for System Daemons，最好开启该特性：

> Kubernetes nodes can be scheduled to `Capacity`. Pods can consume all the available capacity on a node by default. This is an issue because nodes typically run quite a few system daemons that power the OS and Kubernetes itself. Unless resources are set aside for these system daemons, pods and system daemons compete for resources and lead to resource starvation issues on the node.
>
> The `kubelet` exposes a feature named `Node Allocatable` that helps to reserve compute resources for system daemons. Kubernetes recommends cluster administrators to configure `Node Allocatable` based on their workload density on each node.

```
      Node Capacity
---------------------------
|     kube-reserved       |
|-------------------------|
|     system-reserved     |
|-------------------------|
|    eviction-threshold   |
|-------------------------|
|                         |
|      allocatable        |
|   (available for pods)  |
|                         |
|                         |
---------------------------
```

开启方法为：

```
kubelet --cgroups-per-qos=true --enforce-node-allocatable=pods ..
```

Kube-proxy: error looking for path of conntrack

kube-proxy 报错，并且 service 的 DNS 解析异常

```
kube-proxy[2241]: E0502 15:55:13.889842    2241 conntrack.go:42]
```

解决方式是安装 `conntrack-tools` 包后重启 `kube-proxy` 即可。

Dashboard 中无资源使用图表

正常情况下，Dashboard 首页应该会显示资源使用情况的图表，如

如果没有这些图表，则需要首先检查 Heapster 是否正在运行（因为 Dashboard 需要访问 Heapster 来查询资源使用情况）：

```
kubectl -n kube-system get pods -l k8s-app=heapster
NAME                        READY     STATUS    RESTARTS   AGE
heapster-86b59f68f6-h4vt6   2/2       Running   0          5d
```

如果查询结果为空，说明 Heapster 还未部署，可以参考 https://github.com/kubernetes/heapster 来部署。

但如果 Heapster 处于正常状态，那么需要查看 dashboard 的日志，确认是否还有其他问题

```
$ kubectl -n kube-system get pods -l k8s-app=kubernetes-dashboard
NAME                                  READY     STATUS    RESTAR
kubernetes-dashboard-665b4f7df-dsjpn   1/1       Running   0

$ kubectl -n kube-system logs kubernetes-dashboard-665b4f7df-dsjp
```

HPA 不自动扩展 Pod

查看 HPA 的事件，发现

```
$ kubectl describe hpa php-apache
Name:                                                  php-apache
Namespace:                                             default
Labels:
Annotations:
CreationTimestamp:                                     Wed, 27 De
Reference:                                             Deployment
Metrics:                                               ( current
  resource cpu on pods  (as a percentage of request):   / 50%
Min replicas:                                          1
Max replicas:                                          10
Conditions:
  Type           Status  Reason                   Message
  ----           ------  ------                   -------
  AbleToScale    True    SucceededGetScale        the HPA control
  ScalingActive  False   FailedGetResourceMetric  the HPA was una
Events:
  Type     Reason                   Age                  From
  ----     ------                   ----                 ----
  Warning  FailedGetResourceMetric  3m (x2231 over 18h)  horizont
```

这说明 `metrics-server` 未部署，可以参考 这里 部署。

Node 存储空间不足

Node 存储空间不足一般是容器镜像未及时清理导致的，比如短时间内运行了很多使用较大镜像的容器等。Kubelet 会自动清理未使用的镜像，但如果想要立即清理，可以使用 `spotify/docker-gc`：

```
sudo docker run --rm -v /var/run/docker.sock:/var/run/docker.sock
```

/sys/fs/cgroup 空间不足

很多发行版默认的 `fs.inotify.max_user_watches` 太小，只有 8192，可以通过增大该配置解决。比如

```
$ sudo sysctl fs.inotify.max_user_watches=524288
```

大量 ConfigMap/Secret 导致 Kubernetes缓慢

这是从 Kubernetes 1.12 开始才有的问题，Kubernetes issue: #74412。

> This worked well on version 1.11 of Kubernetes. After upgrading to 1.12 or 1.13, I've noticed that doing this will cause the cluster to significantly slow down; up to the point where nodes are being marked as NotReady and no new work is being scheduled.
>
> For example, consider a scenario in which I schedule 400 jobs, each with its own ConfigMap, which print "Hello World" on a single-node cluster would.
>
> - On v1.11, it takes about 10 minutes for the cluster to process all jobs. New jobs can be scheduled.
> - On v1.12 and v1.13, it takes about 60 minutes for the cluster to process all jobs. After this, no new jobs can be scheduled.
>
> This is related to max concurrent http2 streams and the change of configmap manager of kubelet. By default, max concurrent http2 stream of http2 server in kube-apiserver is 250, and every configmap will consume one stream to watch in kubelet at least from version 1.13.x. Kubelet will stuck to communicate to kube-apiserver and then become NotReady if too many pods with configmap scheduled to it. A work around is to change the config http2-max-streams-per-connection of kube-apiserver to a bigger value.

临时解决方法：为 Kubelet 设置 `configMapAndSecretChangeDetectionStrategy: Cache` （参考 这里 ）。

修复方法：升级 Go 版本到 1.12 后重新构建 Kubernetes（社区正在进行中）。修复后，Kubelet 可以 watch 的 configmap 可以从之前的 236 提高到至少 10000。

Kubelet 内存泄漏

这是从 1.12 版本开始有的问题（只在使用 hyperkube 启动 kubelet 时才有问题），社区 issue 为 #73587。

```
(pprof) root@ip-172-31-10-50:~# go tool pprof  http://localhost:1
Fetching profile from http://localhost:10248/debug/pprof/heap
Saved profile in /root/pprof/pprof.hyperkube.localhost:10248.allo
Entering interactive mode (type "help" for commands)
(pprof) top
2406.93MB of 2451.55MB total (98.18%)
Dropped 2863 nodes (cum <= 12.26MB)
Showing top 10 nodes out of 34 (cum >= 2411.39MB)
      flat  flat%   sum%        cum   cum%
 2082.07MB 84.93% 84.93%  2082.07MB 84.93%  k8s.io/kubernetes/ver
  311.65MB 12.71% 97.64%  2398.72MB 97.84%  k8s.io/kubernetes/ver
   10.71MB  0.44% 98.08%  2414.43MB 98.49%  k8s.io/kubernetes/ver
    2.50MB   0.1% 98.18%  2084.57MB 85.03%  k8s.io/kubernetes/ver
         0     0% 98.18%  2412.06MB 98.39%  k8s.io/kubernetes/cmd
         0     0% 98.18%  2412.06MB 98.39%  k8s.io/kubernetes/pkg
         0     0% 98.18%  2412.06MB 98.39%  k8s.io/kubernetes/pkg
```

```
curl -s localhost:10255/metrics | sed 's/{.*//' | sort | uniq -c
  25749 reflector_watch_duration_seconds
  25749 reflector_list_duration_seconds
  25749 reflector_items_per_watch
  25749 reflector_items_per_list
   8583 reflector_watches_total
   8583 reflector_watch_duration_seconds_sum
   8583 reflector_watch_duration_seconds_count
   8583 reflector_short_watches_total
   8583 reflector_lists_total
   8583 reflector_list_duration_seconds_sum
   8583 reflector_list_duration_seconds_count
   8583 reflector_last_resource_version
   8583 reflector_items_per_watch_sum
   8583 reflector_items_per_watch_count
   8583 reflector_items_per_list_sum
   8583 reflector_items_per_list_count
    165 storage_operation_duration_seconds_bucket
     51 kubelet_runtime_operations_latency_microseconds
     44 rest_client_request_latency_seconds_bucket
     33 kubelet_docker_operations_latency_microseconds
     17 kubelet_runtime_operations_latency_microseconds_sum
     17 kubelet_runtime_operations_latency_microseconds_count
     17 kubelet_runtime_operations
```

修复方法：禁止 Reflector metrics。

参考文档

- Troubleshoot Clusters
- SSH into Azure Container Service (AKS) cluster nodes
- Kubernetes dashboard FAQ

Pod排错

本章介绍 Pod 运行异常的排错方法。

一般来说，无论 Pod 处于什么异常状态，都可以执行以下命令来查看 Pod 的状态

- `kubectl get pod -o yaml` 查看 Pod 的配置是否正确
- `kubectl describe pod` 查看 Pod 的事件
- `kubectl logs [-c ]` 查看容器日志

这些事件和日志通常都会有助于排查 Pod 发生的问题。

Pod 一直处于 Pending 状态

Pending 说明 Pod 还没有调度到某个 Node 上面。可以通过 `kubectl describe pod` 命令查看到当前 Pod 的事件，进而判断为什么没有调度。如

```
$ kubectl describe pod mypod
...
Events:
  Type     Reason            Age                From
  ----     ------            ----               ----
  Warning  FailedScheduling  12s (x6 over 27s)  default-scheduler
```

可能的原因包括

- 资源不足，集群内所有的 Node 都不满足该 Pod 请求的 CPU、内存、GPU 或者临时存储空间等资源。解决方法是删除集群内不用的 Pod 或者增加新的 Node。
- HostPort 端口已被占用，通常推荐使用 Service 对外开放服务端口

Pod 一直处于 Waiting 或 ContainerCreating 状态

首先还是通过 `kubectl describe pod` 命令查看到当前 Pod 的事件

```
$ kubectl -n kube-system describe pod nginx-pod
Events:
  Type     Reason                  Age                From
  ----     ------                  ----               ----
  Normal   Scheduled               1m                 default-sched
  Normal   SuccessfulMountVolume   1m                 kubelet, gpu1
  Normal   SuccessfulMountVolume   1m                 kubelet, gpu1
  Warning  FailedSync              2s (x4 over 46s)   kubelet, gpu1
  Normal   SandboxChanged          1s (x4 over 46s)   kubelet, gpu1
```

可以发现，该 Pod 的 Sandbox 容器无法正常启动，具体原因需要查看 Kubelet 日志：

```
$ journalctl -u kubelet
...
Mar 14 04:22:04 node1 kubelet[29801]: E0314 04:22:04.649912   298
Mar 14 04:22:04 node1 kubelet[29801]: E0314 04:22:04.649941   298
Mar 14 04:22:04 node1 kubelet[29801]: W0314 04:22:04.891337   298
Mar 14 04:22:05 node1 kubelet[29801]: E0314 04:22:05.965801   298
```

发现是 cni0 网桥配置了一个不同网段的 IP 地址导致，删除该网桥（网络插件会自动重新创建）即可修复

```
$ ip link set cni0 down
$ brctl delbr cni0
```

除了以上错误，其他可能的原因还有

- 镜像拉取失败，比如
  - 配置了错误的镜像
  - Kubelet 无法访问镜像（国内环境访问 `gcr.io` 需要特殊处理）
  - 私有镜像的密钥配置错误
  - 镜像太大，拉取超时（可以适当调整 kubelet 的 `--image-pull-progress-deadline` 和 `--runtime-request-timeout` 选项）
- CNI 网络错误，一般需要检查 CNI 网络插件的配置，比如
  - 无法配置 Pod 网络
  - 无法分配 IP 地址
- 容器无法启动，需要检查是否打包了正确的镜像或者是否配置了正确的容器参数

Pod 处于 ImagePullBackOff 状态

这通常是镜像名称配置错误或者私有镜像的密钥配置错误导致。这种情况可以使用 `docker pull` 来验证镜像是否可以正常拉取。

```
$ kubectl describe pod mypod
...
Events:
  Type     Reason                  Age                From
  ----     ------                  ----               ----
  Normal   Scheduled               36s                default-sche
  Normal   SuccessfulMountVolume   35s                kubelet, k8s
  Normal   Pulling                 17s (x2 over 33s)  kubelet, k8s
  Warning  Failed                  14s (x2 over 29s)  kubelet, k8s
  Warning  Failed                  14s (x2 over 29s)  kubelet, k8s
  Normal   SandboxChanged          4s (x7 over 28s)   kubelet, k8s
  Normal   BackOff                 4s (x5 over 25s)   kubelet, k8s
  Warning  Failed                  1s (x6 over 25s)   kubelet, k8s
```

如果是私有镜像，需要首先创建一个 `docker-registry` 类型的 `Secret`

```
kubectl create secret docker-registry my-secret --docker-server=D
```

然后在容器中引用这个 `Secret`

```
spec:
  containers:
  - name: private-reg-container
    image: >
  imagePullSecrets:
  - name: my-secret
```

Pod 一直处于 CrashLoopBackOff 状态

`CrashLoopBackOff` 状态说明容器曾经启动了，但又异常退出了。此时 `Pod` 的 `RestartCounts` 通常是大于 `0` 的，可以先查看一下容器的日志

```
kubectl describe pod
kubectl logs
kubectl logs --previous
```

这里可以发现一些容器退出的原因，比如

- 容器进程退出
- 健康检查失败退出
- OOMKilled

```
$ kubectl describe pod mypod
...
Containers:
  sh:
    Container ID:  docker://3f7a2ee0e7e0e16c22090a25f9b6e42b5c06e
    Image:         alpine
    Image ID:      docker-pullable://alpine@sha256:7b848083f93822
    Port:
    Host Port:
    State:          Terminated
      Reason:       OOMKilled
      Exit Code:    2
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    2
    Ready:          False
    Restart Count:  3
    Limits:
      cpu:     1
      memory:  1G
    Requests:
      cpu:        100m
      memory:     500M
...
```

如果此时如果还未发现线索，还可以到容器内执行命令来进一步查看退出原因

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

如果还是没有线索，那就需要 SSH 登录该 Pod 所在的 Node 上，查看 Kubelet 或者 Docker 的日志进一步排查了

```
# Query Node
kubectl get pod  -o wide

# SSH to Node
ssh @
```

Pod 处于 Error 状态

通常处于 Error 状态说明 Pod 启动过程中发生了错误。常见的原因包括

- 依赖的 ConfigMap、Secret 或者 PV 等不存在
- 请求的资源超过了管理员设置的限制，比如超过了 LimitRange 等
- 违反集群的安全策略，比如违反了 PodSecurityPolicy 等
- 容器无权操作集群内的资源，比如开启 RBAC 后，需要为 ServiceAccount 配置角色绑定

Pod 处于 Terminating 或 Unknown 状态

从 v1.5 开始，Kubernetes 不会因为 Node 失联而删除其上正在运行的 Pod，而是将其标记为 Terminating 或 Unknown 状态。想要删除这些状态的 Pod 有三种方法：

- 从集群中删除该 Node。使用公有云时，kube-controller-manager 会在 VM 删除后自动删除对应的 Node。而在物理机部署的集群中，需要管理员手动删除 Node（如 `kubectl delete node` 。
- Node 恢复正常。Kubelet 会重新跟 kube-apiserver 通信确认这些 Pod 的期待状态，进而再决定删除或者继续运行这些 Pod。
- 用户强制删除。用户可以执行 `kubectl delete pods --grace-period=0 --force` 强制删除 Pod。除非明确知道 Pod 的确处于停止状态（比如 Node 所在 VM 或物理机已经关机），否则不建议使用该方法。特别是 StatefulSet 管理的 Pod，强制删除容易导致脑裂或者数据丢失等问题。

如果 Kubelet 是以 Docker 容器的形式运行的，此时 kubelet 日志中可能会发现如下的错误：

```
{"log":"I0926 19:59:07.162477   54420 kubelet.go:1894] SyncLoop (
{"log":"I0926 19:59:39.977126   54420 reconciler.go:186] operatic
{"log":"E0926 19:59:39.977461   54420 nestedpendingoperations.go:
```

如果是这种情况，则需要给 kubelet 容器设置 `--containerized` 参数并传入以下的存储卷

```
# 以使用 calico 网络插件为例
      -v /:/rootfs:ro,shared \
      -v /sys:/sys:ro \
      -v /dev:/dev:rw \
      -v /var/log:/var/log:rw \
      -v /run/calico/:/run/calico/:rw \
      -v /run/docker/:/run/docker/:rw \
      -v /run/docker.sock:/run/docker.sock:rw \
      -v /usr/lib/os-release:/etc/os-release \
      -v /usr/share/ca-certificates/:/etc/ssl/certs \
      -v /var/lib/docker/:/var/lib/docker:rw,shared \
      -v /var/lib/kubelet/:/var/lib/kubelet:rw,shared \
      -v /etc/kubernetes/ssl/:/etc/kubernetes/ssl/ \
      -v /etc/kubernetes/config/:/etc/kubernetes/config/ \
      -v /etc/cni/net.d/:/etc/cni/net.d/ \
      -v /opt/cni/bin/:/opt/cni/bin/ \
```

处于 `Terminating` 状态的 Pod 在 Kubelet 恢复正常运行后一般会自动删除。但有时也会出现无法删除的情况，并且通过 `kubectl delete pods --grace-period=0 --force` 也无法强制删除。此时一般是由于 `finalizers` 导致的，通过 `kubectl edit` 将 `finalizers` 删除即可解决。

```
"finalizers": [
  "foregroundDeletion"
]
```

Pod 行为异常

这里所说的行为异常是指 Pod 没有按预期的行为执行，比如没有运行 podSpec 里面设置的命令行参数。这一般是 podSpec yaml 文件内容有误，可以尝试使用 `--validate` 参数重建容器，比如

```
kubectl delete pod mypod
kubectl create --validate -f mypod.yaml
```

也可以查看创建后的 podSpec 是否是对的，比如

```
kubectl get pod mypod -o yaml
```

修改静态 Pod 的 Manifest 后未自动重建

Kubelet 使用 inotify 机制检测 `/etc/kubernetes/manifests` 目录（可通过 Kubelet 的 `--pod-manifest-path` 选项指定）中静态 Pod 的变化，并在文件发生变化后重新创建相应的 Pod。但有时也会发生修改静态 Pod 的 Manifest 后未自动创建新 Pod 的情景，此时一个简单的修复方法是重启 Kubelet。

参考文档

- Troubleshoot Applications

网络排错

本章主要介绍各种常见的网络问题以及排错方法，包括 Pod 访问异常、Service 访问异常以及网络安全策略异常等。

说到 Kubernetes 的网络，其实无非就是以下三种情况之一

- Pod 访问容器外部网络
- 从容器外部访问 Pod 网络
- Pod 之间相互访问

当然，以上每种情况还都分别包括本地访问和跨主机访问两种场景，并且一般情况下都是通过 Service 间接访问 Pod。

排查网络问题基本上也是从这几种情况出发，定位出具体的网络异常点，再进而寻找解决方法。网络异常可能的原因比较多，常见的有

- CNI 网络插件配置错误，导致多主机网络不通，比如
  - IP 网段与现有网络冲突
  - 插件使用了底层网络不支持的协议
  - 忘记开启 IP 转发等
    - `sysctl net.ipv4.ip_forward`
    - `sysctl net.bridge.bridge-nf-call-iptables`
- Pod 网络路由丢失，比如
  - kubenet 要求网络中有 podCIDR 到主机 IP 地址的路由，这些路由如果没有正确配置会导致 Pod 网络通信等问题
  - 在公有云平台上，kube-controller-manager 会自动为所有 Node 配置路由，但如果配置不当（如认证授权失败、超出配额等），也有可能导致无法配置路由
- 主机内或者云平台的安全组、防火墙或者安全策略等阻止了 Pod 网络，比如
  - 非 Kubernetes 管理的 iptables 规则禁止了 Pod 网络
  - 公有云平台的安全组禁止了 Pod 网络（注意 Pod 网络有可能与 Node 网络不在同一个网段）
  - 交换机或者路由器的 ACL 禁止了 Pod 网络

Flannel Pods 一直处于 Init:CrashLoopBackOff 状态

Flannel 网络插件非常容易部署，只要一条命令即可

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel
```

然而，部署完成后，Flannel Pod 有可能会碰到初始化失败的错误

```
$ kubectl -n kube-system get pod
NAME                              READY     STATUS
kube-flannel-ds-ckfdc             0/1       Init:CrashLoopBackOff
kube-flannel-ds-jpp96             0/1       Init:CrashLoopBackOff
```

查看日志会发现

```
$ kubectl -n kube-system logs kube-flannel-ds-jpp96 -c install-cr
cp: can't create '/etc/cni/net.d/10-flannel.conflist': Permissior
```

这一般是由于 SELinux 开启导致的，关闭 SELinux 既可解决。有两种方法：

- 修改 `/etc/selinux/config` 文件方法：`SELINUX=disabled`
- 通过命令临时修改（重启会丢失）：`setenforce 0`

Pod 无法分配 IP

Pod 一直处于 ContainerCreating 状态，查看事件发现网络插件无法为其分配 IP：

```
  Normal   SandboxChanged          5m (x74 over 8m)    kubelet, k
  Warning  FailedCreatePodSandBox  21s (x204 over 8m)  kubelet, k
```

查看网络插件的 IP 分配情况，进一步发现 IP 地址确实已经全部分配完，但真正处于 Running 状态的 Pod 数却很少：

```
# 详细路径取决于具体的网络插件，当使用 host-local IPAM 插件时，路径位于 /var/
$ cd /var/lib/cni/networks/kubenet
$ ls -al|wc -l
258

$ docker ps | grep POD | wc -l
7
```

这有两种可能的原因

- 网络插件本身的问题，Pod 停止后其 IP 未释放
- Pod 重新创建的速度比 Kubelet 调用 CNI 插件回收网络（垃圾回收时删除已停止 Pod 前会先调用 CNI 清理网络）的速度快

对第一个问题，最好联系插件开发者询问修复方法或者临时性的解决方法。当然，如果对网络插件的工作原理很熟悉的话，也可以考虑手动释放未使用的 IP 地址，比如：

- 停止 Kubelet
- 找到 IPAM 插件保存已分配 IP 地址的文件，比如 `/var/lib/cni/networks/cbr0`（flannel）或者 `/var/run/azure-vnet-ipam.json`（Azure CNI）等
- 查询容器已用的 IP 地址，比如 `kubectl get pod -o wide --all-namespaces | grep`
- 对比两个列表，从 IPAM 文件中删除未使用的 IP 地址，并手动删除相关的虚拟网卡和网络命名空间（如果有的话）
- 重启启动 Kubelet

```
# Take kubenet for example to delete the unused IPs
$ for hash in $(tail -n +1 * | grep '^[A-Za-z0-9]*$' | cut -c 1-8
```

而第二个问题则可以给 Kubelet 配置更快的垃圾回收，如

```
--minimum-container-ttl-duration=15s
--maximum-dead-containers-per-container=1
--maximum-dead-containers=100
```

Pod 无法解析 DNS

如果 Node 上安装的 Docker 版本大于 1.12，那么 Docker 会把默认的 iptables FORWARD 策略改为 DROP。这会引发 Pod 网络访问的问题。解决方法则在每个 Node 上面运行 `iptables -P FORWARD ACCEPT`，比如

```
echo "ExecStartPost=/sbin/iptables -P FORWARD ACCEPT" >> /etc/sys
systemctl daemon-reload
systemctl restart docker
```

如果使用了 flannel/weave 网络插件，更新为最新版本也可以解决这个问题。

DNS 无法解析也有可能是 kube-dns 服务异常导致的，可以通过下面的命令来检查 kube-dns 是否处于正常运行状态

```
$ kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
NAME                    READY     STATUS    RESTARTS   AGE
...
kube-dns-v19-ezo1y      3/3       Running   0          1h
...
```

如果 kube-dns 处于 CrashLoopBackOff 状态，那么可以参考 Kube-dns/Dashboard CrashLoopBackOff 排错 来查看具体排错方法。

如果 kube-dns Pod 处于正常 Running 状态，则需要进一步检查是否正确配置了 kube-dns 服务：

```
$ kubectl get svc kube-dns --namespace=kube-system
NAME          CLUSTER-IP     EXTERNAL-IP   PORT(S)             AG
kube-dns      10.0.0.10              53/UDP,53/TCP        1h

$ kubectl get ep kube-dns --namespace=kube-system
NAME       ENDPOINTS                       AGE
kube-dns   10.180.3.17:53,10.180.3.17:53    1h
```

如果 kube-dns service 不存在，或者 endpoints 列表为空，则说明 kube-dns service 配置错误，可以重新创建 kube-dns service，比如

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 10.0.0.10
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

DNS解析缓慢

由于内核的一个 BUG，连接跟踪模块会发生竞争，导致 DNS 解析缓慢。

临时解决方法：为容器配置 `options single-request-reopen`

```
        lifecycle:
          postStart:
            exec:
              command:
              - /bin/sh
              - -c
              - "/bin/echo 'options single-request-reopen' >> /et
```

修复方法：升级内核并保证包含以下两个补丁

1. "netfilter: nf_conntrack: resolve clash for matching conntracks" fixes the 1st race (accepted).

2. "netfilter: nf_nat: return the same reply tuple for matching CTs" fixes the 2nd race (waiting for a review).

其他可能的原因和修复方法还有：

- Kube-dns 和 CoreDNS 同时存在时也会有问题，只保留一个即可。
- kube-dns 或者 CoreDNS 的资源限制太小时会导致 DNS 解析缓慢，这时候需要增大资源限制。

更多 DNS 配置的方法可以参考 Customizing DNS Service。

Service 无法访问

访问 Service ClusterIP 失败时，可以首先确认是否有对应的 Endpoints

```
kubectl get endpoints
```

如果该列表为空，则有可能是该 Service 的 LabelSelector 配置错误，可以用下面的方法确认一下

```
# 查询 Service 的 LabelSelector
kubectl get svc  -o jsonpath='{.spec.selector}'

# 查询匹配 LabelSelector 的 Pod
kubectl get pods -l key1=value1,key2=value2
```

如果 Endpoints 正常，可以进一步检查

- Pod 的 containerPort 与 Service 的 containerPort 是否对应
- 直接访问 `podIP:containerPort` 是否正常

再进一步，即使上述配置都正确无误，还有其他的原因会导致 Service 无法访问，比如

- Pod 内的容器有可能未正常运行或者没有监听在指定的 containerPort 上
- CNI 网络或主机路由异常也会导致类似的问题

- kube-proxy 服务有可能未启动或者未正确配置相应的 iptables 规则，比如正常情况下名为 `hostnames` 的服务会配置以下 iptables 规则

```
$ iptables-save | grep hostnames
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --commer
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --commer
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --commer
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "de
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostna
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostna
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostna
```

Pod 无法通过 Service 访问自己

这通常是 hairpin 配置错误导致的，可以通过 Kubelet 的 `--hairpin-mode` 选项配置，可选参数包括 "promiscuous-bridge"、"hairpin-veth" 和 "none"（默认为"promiscuous-bridge"）。

对于 hairpin-veth 模式，可以通过以下命令来确认是否生效

```
$ for intf in /sys/devices/virtual/net/cbr0/brif/*; do cat $intf/
1
1
1
1
```

而对于 promiscuous-bridge 模式，可以通过以下命令来确认是否生效

```
$ ifconfig cbr0 |grep PROMISC
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```

无法访问 Kubernetes API

很多扩展服务需要访问 Kubernetes API 查询需要的数据（比如 kube-dns、Operator 等）。通常在 Kubernetes API 无法访问时，可以首先通过下面的命令验证 Kubernetes API 是正常的：

```
$ kubectl run curl  --image=appropriate/curl -i -t  --restart=Nev
If you don't see a command prompt, try pressing enter.
/ #
/ # KUBE_TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccour
/ # curl -sSk -H "Authorization: Bearer $KUBE_TOKEN" https://$KUE
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "2285"
  },
  "items": [
   ...
  ]
 }
```

如果出现超时错误，则需要进一步确认名为 `kubernetes` 的服务以及 endpoints 列表是正常的：

```
$ kubectl get service kubernetes
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1            443/TCP   25m
$ kubectl get endpoints kubernetes
NAME         ENDPOINTS          AGE
kubernetes   172.17.0.62:6443   25m
```

然后可以直接访问 endpoints 查看 kube-apiserver 是否可以正常访问。无法访问时通常说明 kube-apiserver 未正常启动，或者有防火墙规则阻止了访问。

但如果出现了 `403 – Forbidden` 错误，则说明 Kubernetes 集群开启了访问授权控制（如 RBAC），此时就需要给 Pod 所用的 ServiceAccount 创建角色和角色绑定授权访问所需要的资源。比如 CoreDNS 就需要创建以下 ServiceAccount 以及角色绑定：

```
# 1. service account
apiVersion: v1
kind: ServiceAccount
metadata:
  name: coredns
  namespace: kube-system
  labels:
      kubernetes.io/cluster-service: "true"
      addonmanager.kubernetes.io/mode: Reconcile
---
# 2. cluster role
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
    addonmanager.kubernetes.io/mode: Reconcile
  name: system:coredns
rules:
- apiGroups:
  - ""
  resources:
  - endpoints
  - services
  - pods
  - namespaces
  verbs:
  - list
  - watch
---
# 3. cluster role binding
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
    addonmanager.kubernetes.io/mode: EnsureExists
  name: system:coredns
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:coredns
subjects:
- kind: ServiceAccount
  name: coredns
  namespace: kube-system
---
# 4. use created service account
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: coredns
  namespace: kube-system
  labels:
    k8s-app: coredns
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
    kubernetes.io/name: "CoreDNS"
spec:
  replicas: 2
  selector:
    matchLabels:
      k8s-app: coredns
  template:
    metadata:
      labels:
        k8s-app: coredns
    spec:
      serviceAccountName: coredns
      ...
```

内核导致的问题

除了以上问题，还有可能碰到因内核问题导致的服务无法访问或者服务访问超时的错误，比如

- 未设置 `--random-fully` 导致无法为 SNAT 分配端口，进而会导致服务访问超时。注意， Kubernetes 暂时没有为 SNAT 设置 `--random-fully` 选项，如果碰到这个问题可以参考这里 配置。

参考文档

- Troubleshoot Applications
- Debug Services

PV排错

本章介绍持久化存储异常（PV、PVC、StorageClass等）的排错方法。

一般来说，无论 PV 处于什么异常状态，都可以执行 `kubectl describe pv/pvc` 命令来查看当前 PV 的事件。这些事件通常都会有助于排查 PV 或 PVC 发生的问题。

```
kubectl get pv
kubectl get pvc
kubectl get sc

kubectl describe pv
kubectl describe pvc
kubectl describe sc
```

AzureDisk

AzureDisk 为 Azure 上面运行的虚拟机提供了弹性块存储服务，它以 VHD 的形式挂载到虚拟机中，并可以在 Kubernetes 容器中使用。AzureDisk 有点是性能高，特别是 Premium Storage 提供了非常好的性能；其缺点是不支持共享，只可以用在单个 Pod 内。

根据配置的不同，Kubernetes 支持的 AzureDisk 可以分为以下几类

- Managed Disks：由 Azure 自动管理磁盘和存储账户
- Blob Disks：
  - Dedicated（默认）：为每个 AzureDisk 创建单独的存储账户，当删除 PVC 的时候删除该存储账户
  - Shared：AzureDisk 共享 ResourceGroup 内的同一个存储账户，这时删除 PVC 不会删除该存储账户

> 注意：
>
> - AzureDisk 的类型必须跟 VM OS Disk 类型一致，即要么都是 Manged Disks，要么都是 Blob Disks。当两者不一致时，AzureDisk PV 会报无法挂载的错误。
> - 由于 Managed Disks 需要创建和管理存储账户，其创建过程会比 Blob Disks 慢（3 分钟 vs 1-2 分钟）。
> - 但节点最大支持同时挂载 16 个 AzureDisk。

使用 acs-engine 部署的 Kubernetes 集群，会自动创建两个 StorageClass，默认为managed-standard（即HDD）：

```
kubectl get storageclass
NAME                 PROVISIONER                AGE
default (default)    kubernetes.io/azure-disk   45d
managed-premium      kubernetes.io/azure-disk   53d
managed-standard     kubernetes.io/azure-disk   53d
```

AzureDisk 挂载失败

在 AzureDisk 从一个 Pod 迁移到另一 Node 上面的 Pod 时或者同一台 Node 上面使用了多块 AzureDisk 时有可能会碰到这个问题。这是由于 kube-controller-manager 未对 AttachDisk 和 DetachDisk 操作加锁从而引发了竞争问题（kubernetes#60101 acs-engine#2002 ACS#12）。

通过 kube-controller-manager 的日志，可以查看具体的错误原因。常见的错误日志为

```
Cannot attach data disk 'cdb-dynamic-pvc-92972088-11b9-11e8-888f-
```

临时性解决方法为

（1）更新所有受影响的虚拟机状态

```
$vm = Get-AzureRMVM -ResourceGroupName $rg -Name $vmname
Update-AzureRmVM -ResourceGroupName $rg -VM $vm -verbose -debug
```

（2）重启虚拟机

- `kubectl cordon NODE`
- 如果 Node 上运行有 StatefulSet，需要手动删除相应的 Pod
- `kubectl drain NODE`
- `Get-AzureRMVM -ResourceGroupName $rg -Name $vmname | Restart-AzureVM`
- `kubectl uncordon NODE`

该问题的修复 #60183 已包含在 v1.10 中。

挂载新的 AzureDisk 后，该 Node 中其他 Pod 已挂载的 AzureDisk 不可用

在 Kubernetes v1.7 中，AzureDisk 默认的缓存策略修改为 `ReadWrite`，这会导致在同一个 Node 中挂载超过 5 块 AzureDisk 时，已有 AzureDisk 的盘符会随机改变（kubernetes#60344 kubernetes#57444 AKS#201 acs-engine#1918）。比如，当挂载第六块 AzureDisk 后，原来 lun0 磁盘的挂载盘符有可能从 `sdc` 变成 `sdk`：

```
$ tree /dev/disk/azure
...
â""â"€â"€ scsi1
    â"œâ"€â"€ lun0 -> ../../../sdk
    â"œâ"€â"€ lun1 -> ../../../sdj
    â"œâ"€â"€ lun2 -> ../../../sde
    â"œâ"€â"€ lun3 -> ../../../sdf
    â"œâ"€â"€ lun4 -> ../../../sdg
    â"œâ"€â"€ lun5 -> ../../../sdh
    â""â"€â"€ lun6 -> ../../../sdi
```

这样，原来使用 lun0 磁盘的 Pod 就无法访问 AzureDisk 了

```
[root@admin-0 /]# ls /datadisk
ls: reading directory .: Input/output error
```

临时性解决方法是设置 AzureDisk StorageClass 的 `cachingmode: None` ，如

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: managed-standard
provisioner: kubernetes.io/azure-disk
parameters:
  skuname: Standard_LRS
  kind: Managed
  cachingmode: None
```

该问题的修复 #60346 将包含在 v1.10 中。

AzureDisk 挂载慢

AzureDisk PVC 的挂载过程一般需要 1 分钟的时间，这些时间主要消耗在 Azure ARM API 的调用上（查询 VM 以及挂载 Disk）。#57432 为 Azure VM 增加了一个缓存，消除了 VM 的查询时间，将整个挂载过程缩短到大约 30 秒。该修复包含在v1.9.2+ 和 v1.10 中。

另外，如果 Node 使用了 `Standard_B1s` 类型的虚拟机，那么 AzureDisk 的第一次挂载一般会超时，等再次重复时才会挂载成功。这是因为在 `Standard_B1s` 虚拟机中格式化 AzureDisk 就需要很长时间（如超过 70 秒）。

```
$ kubectl describe pod
...
Events:
  FirstSeen     LastSeen        Count   From
  ---------     --------        -----   ----
  8m            8m              1       default-scheduler
  7m            7m              1       kubelet, aks-nodepool1-15
  5m            5m              1       kubelet, aks-nodepool1-15
d9e-0a58ac1f0a2e)": timeout expired waiting for volumes to attach
  5m            5m              1       kubelet, aks-nodepool1-15
  4m            4m              1       kubelet, aks-nodepool1-15
a2e"
  4m            4m              1       kubelet, aks-nodepool1-15
  3m            3m              1       kubelet, aks-nodepool1-15
  3m            3m              1       kubelet, aks-nodepool1-15
  2m            2m              1       kubelet, aks-nodepool1-15
```

Azure German Cloud 无法使用 AzureDisk

Azure German Cloud 仅在 v1.7.9+、v1.8.3+ 以及更新版本中支持（#50673），升级 Kubernetes 版本即可解决。

MountVolume.WaitForAttach failed

```
MountVolume.WaitForAttach failed for volume "pvc-f1562ecb-3e5f-11
```

该问题 仅在 Kubernetes v1.10.0 和 v1.10.1 中存在，将在 v1.10.2 中修复。

参考文档

- Known kubernetes issues on Azure
- Introduction of AzureDisk
- AzureDisk volume examples

- High-performance Premium Storage and managed disks for VMs

AzureFile

AzureFile 提供了基于 SMB 协议（也称 CIFS）托管文件共享服务。它支持 Windows 和 Linux 容器，并支持跨主机的共享，可用于多个 Pod 之间的共享存储。AzureFile 的缺点是性能较差（AKS#223），并且不提供 Premium 存储。

推荐基于 StorageClass 来使用 AzureFile，即

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
parameters:
  skuName: Standard_LRS
```

访问权限

AzureFile 使用 mount.cifs 将其远端存储挂载到 Node 上，而 `fileMode` 和 `dirMode` 控制了挂载后文件和目录的访问权限。不同的 Kubernetes 版本，`fileMode` 和 `dirMode` 的默认选项是不同的

Kubernetes 版本	fileMode和dirMode
v1.6.x, v1.7.x	0777
v1.8.0-v1.8.5	0700
v1.8.6 or above	0755
v1.9.0	0700
v1.9.1 or above	0755

按照默认的权限会导致非跟用户无法在目录中创建新的文件，解决方法为

- v1.8.0-v1.8.5：设置容器以 root 用户运行，如设置 `spec.securityContext.runAsUser: 0`
- v1.8.6 以及更新版本：在 AzureFile StorageClass 通过 mountOptions 设置默认权限，比如设置为 `0777` 的方法为

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
parameters:
  skuName: Standard_LRS
```

Windows Node 重启后无法访问 AzureFile

Windows Node 重启后，挂载 AzureFile 的 Pod 可以看到如下错误（#60624）：

```
Warning  Failed                 1m (x7 over 1m)  kubelet, 77890k8
  Normal   SandboxChanged         1m (x8 over 1m)  kubelet, 77890
```

临时性解决方法为删除并重新创建使用了 AzureFile 的 Pod。当 Pod 使用控制器（如 Deployment、StatefulSet等）时，删除 Pod 后控制器会自动创建一个新的 Pod。

该问题的修复 #60625 包含在 v1.10 中。

AzureFile ProvisioningFailed

Azure 文件共享的名字最大只允许 63 个字节，因而在集群名字较长的集群（Kubernetes v1.7.10 或者更老的集群）里面有可能会碰到 AzureFile 名字长度超限的情况，导致 AzureFile ProvisioningFailed：

```
persistentvolume-controller    Warning    ProvisioningFailed Fail
```

碰到该问题时可以通过升级集群解决，其修复 #48326 已经包含在 v1.7.11、v1.8 以及更新版本中。

在开启 RBAC 的集群中，由于 AzureFile 需要访问 Secret，而 kube-controller-manager 中并未为 AzureFile 自动授权，从而也会导致 ProvisioningFailed：

```
Events:
  Type     Reason              Age   From
  ----     ------              ----  ----
  Warning  ProvisioningFailed  8s    persistentvolume-controller
m:persistent-volume-binder" cannot create secrets in the namespac
  Warning  ProvisioningFailed  8s    persistentvolume-controller
```

解决方法是为 ServiceAccount `persistent-volume-binder` 授予 Secret 的访问权限：

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:azure-cloud-provider
rules:
- apiGroups: ['']
  resources: ['secrets']
  verbs:     ['get','create']
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: system:azure-cloud-provider
roleRef:
  kind: ClusterRole
  apiGroup: rbac.authorization.k8s.io
  name: system:azure-cloud-provider
subjects:
- kind: ServiceAccount
  name: persistent-volume-binder
  namespace: kube-system
```

Azure German Cloud 无法使用 AzureFile

Azure German Cloud 仅在 v1.7.11+、v1.8+ 以及更新版本中支持（#48460），升级 Kubernetes 版本即可解决。

参考文档

- Known kubernetes issues on Azure
- Introduction of Azure File Storage
- AzureFile volume examples
- Persistent volumes with Azure files
- Azure Files scalability and performance targets

Windows排错

本章介绍 Windows 容器异常的排错方法。

RDP 登录到 Node

通常在排查 Windows 容器异常问题时需要通过 RDP 登录到 Windows Node上面查看 kubelet、docker、HNS 等的状态和日志。在使用云平台时，可以给相应的 VM 绑定一个公网 IP；而在物理机部署时，可以通过路由器上的端口映射来访问。

除此之外，还有一种更简单的方法，即通过 Kubernetes Service 对外暴露 Node 的 3389 端口（注意替换为你自己的 node-ip）：

```
# rdp.yaml
apiVersion: v1
kind: Service
metadata:
  name: rdp
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 3389
    targetPort: 3389
---
kind: Endpoints
apiVersion: v1
metadata:
  name: rdp
subsets:
  - addresses:
      - ip: >
    ports:
      - port: 3389
```

```
$ kubectl create -f rdp.yaml
$ kubectl get svc rdp
NAME      TYPE           CLUSTER-IP    EXTERNAL-IP      PORT(S)
rdp       LoadBalancer   10.0.99.149   52.52.52.52   3389:32008/T
```

接着，就可以通过 rdp 服务的外网 IP 来登录 Node，如 `mstsc.exe -v 52.52.52.52` 。

在使用完后， 不要忘记删除 RDP 服务 `kubectl delete -f rdp.yaml` 。

Windows Pod 一直处于 ContainerCreating 状态

一般有两种可能的原因

- Pause 镜像配置错误
- 容器镜像版本与 Windows 系统不兼容

在 Windows Server 1709 上面需要使用 1709 标签的镜像，比如

```
* `microsoft/aspnet:4.7.2-windowsservercore-1709`
* `microsoft/windowsservercore:1709`
* `microsoft/iis:windowsservercore-1709`
```

在 Windows Server 1803 上面需要使用 1803 标签的镜像，比如

```
* `microsoft/aspnet:4.7.2-windowsservercore-1803`
* `microsoft/iis:windowsservercore-1803`
* `microsoft/windowsservercore:1803`
```

Windows Pod 内无法解析 DNS

这是一个已知问题，有以下三种临时解决方法：

（1）Windows 重启后，清空 HNS Policy 并重启 KubeProxy 服务：

```
Start-BitsTransfer -Source https://raw.githubusercontent.com/Micr
Import-Module .\hns.psm1

Stop-Service kubeproxy
Stop-Service kubelet
Get-HnsNetwork | ? Name -eq l2Bridge | Remove-HnsNetwork
Get-HnsPolicyList | Remove-HnsPolicyList
Start-Service kubelet
Start-Service kubeproxy
```

（2）是为 Pod 直接配置 kube-dns Pod 的地址：

```
$adapter=Get-NetAdapter
Set-DnsClientServerAddress -InterfaceIndex $adapter.ifIndex -Serv
Set-DnsClient -InterfaceIndex $adapter.ifIndex -ConnectionSpecifi
```

（3）更简单的为每个 Windows Node 多运行一个 Pod，即保证每台 Node 上面至少有两个 Pod 在运行。此时，DNS 解析也是正常的。

如果 Windows Node 运行在 Azure 上面，并且部署 Kubernetes 时使用了自定义 VNET，那么需要为该 VNET 添加路由表：

```
#!/bin/bash
# KubernetesSubnet is the name of the vnet subnet
# KubernetesCustomVNET is the name of the custom VNET itself
rt=$(az network route-table list -g acs-custom-vnet -o json | jq
az network vnet subnet update -n KubernetesSubnet \
-g acs-custom-vnet \
--vnet-name KubernetesCustomVNET \
--route-table $rt
```

如果 VNET 在不同的 ResourceGroup 里面，那么

```
rt=$(az network route-table list -g RESOURCE_GROUP_NAME_KUBE -o j
az network vnet subnet update \
-g RESOURCE_GROUP_NAME_VNET \
--route-table $rt \
--ids "/subscriptions/SUBSCRIPTION_ID/resourceGroups/RESOURCE_GRO
```

Remote endpoint creation failed: HNS failed with error: The switch-port was not found

这个错误发生在 kube-proxy 为服务配置负载均衡的时候，需要安装 KB4089848：

```
Start-BitsTransfer http://download.windowsupdate.com/d/msdownload
wusa.exe windows10.0-kb4089848-x64_db7c5aad31c520c6983a937c3d5317
Restart-Computer
```

重启后确认更新安装成功：

```
PS C:\k> Get-HotFix

Source        Description      HotFixID      InstalledBy
------        -----------      --------      -----------
27171k8s9000  Update           KB4087256     NT AUTHORITY\SYSTEM
27171k8s9000  Update           KB4089848     NT AUTHORITY\SYSTEM
```

安装更新后，如果 DNS 解析还是有问题，可以按照上一节中的方法（1） 重启 kubelet 和 kube-proxy。

Windows Pod 内无法访问 ServiceAccount Secret

这是老版本 Windows 的已知问题，升级 Windows 到 1803 即可解决，升级步骤见这里。

Windows Pod 内无法访问 Kubernetes API

如果使用了 Hyper-V 隔离容器，需要开启 MAC spoofing 。

Windows Node 内无法访问 Service ClusterIP

这是个当前 Windows 网络协议栈的已知问题，只有在 Pod 内才可以访问 Service ClusterIP。

Kubelet 无法启动

使用 Docker 18.03 版本和 Kubelet v1.12.x 时，Kubelet 无法正常启动报错：

```
Error response from daemon: client version 1.38 is too new. Maxim
```

解决方法是为 Windows 上面的 Docker 设置 API 版本的环境变量：

```
[System.Environment]::SetEnvironmentVariable('DOCKER_API_VERSION'
```

参考文档

- Kubernetes On Windows - Troubleshooting Kubernetes

云平台排错

本章主要介绍在公有云中运行 Kubernetes 时可能会碰到的问题以及解决方法。

在公有云平台上运行 Kubernetes，一般可以使用云平台提供的托管 Kubernetes 服务（比如 Google 的 GKE、微软 Azure 的 AKS 或者 AWS 的 Amazon EKS 等）。当然，为了更自由的灵活性，也可以直接在这些公有云平台的虚拟机中部署 Kubernetes。无论哪种方法，一般都需要给 Kubernetes 配置 Cloud Provider 选项，以方便直接利用云平台提供的高级网络、持久化存储以及安全控制等功能。

而在云平台中运行 Kubernetes 的常见问题有

- 认证授权问题：比如 Kubernetes Cloud Provider 中配置的认证方式无权操作虚拟机所在的网络或持久化存储。这一般从 kube-controller-manager 的日志中很容易发现。
- 网络路由配置失败：正常情况下，Cloud Provider 会为每个 Node 配置一条 PodCIDR 至 NodeIP 的路由规则，如果这些规则有问题就会导致多主机 Pod 相互访问的问题。
- 公网 IP 分配失败：比如 LoadBalancer 类型的 Service 无法分配公网 IP 或者指定的公网 IP 无法使用。这一版也是配置错误导致的。
- 安全组配置失败：比如无法为 Service 创建安全组（如超出配额等）或与已有的安全组冲突等。
- 持久化存储分配或者挂载问题：比如分配 PV 失败（如超出配额、配置错误等）或挂载到虚拟机失败（比如 PV 正被其他异常 Pod 引用而导致无法从旧的虚拟机中卸载）。
- 网络插件使用不当：比如网络插件使用了云平台不支持的网络协议等。

Node 未注册到集群中

通常，在 Kubelet 启动时会自动将自己注册到 kubernetes API 中，然后通过 `kubectl get nodes` 就可以查询到该节点。 如果新的 Node 没有自动注册到 Kubernetes 集群中，那说明这个注册过程有错误发生，需要检查 kubelet 和 kube-controller-manager 的日志，进而再根据日志查找具体的错误原因。

Kubelet 日志

查看 Kubelet 日志需要首先 SSH 登录到 Node 上，然后运行 `journalctl` 命令查看 kubelet 的日志：

```
journalctl –l –u kubelet
```

kube-controller-manager 日志

kube-controller-manager 会自动在云平台中给 Node 创建路由，如果路由创建创建失败也有可能导致 Node 注册失败。

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-contro
kubectl -n kube-system logs $PODNAME --tail 100
```

Azure

Azure 负载均衡

使用 Azure Cloud Provider 后，Kubernetes 会为 LoadBalancer 类型的 Service 创建 Azure 负载均衡器以及相关的 公网 IP、BackendPool 和 Network Security Group (NSG)。注意目前 Azure Cloud Provider 仅支持 `Basic` SKU 的负载均衡，并将在 v1.11 中支持 Standard SKU。`Basic` 与 `Standard` SKU 负载均衡相比有一定的局限：

Load Balancer	Basic	Standard
Back-end pool size	up to 100	up to 1,000
Back-end pool boundary	Availability Set	virtual network, region
Back-end pool design	VMs in Availability Set, virtual machine scale set in Availability Set	Any VM instance in the virtual network
HA Ports	Not supported	Available
Diagnostics	Limited, public only	Available
VIP Availability	Not supported	Available
Fast IP Mobility	Not supported	Available
Availability Zones scenarios	Zonal only	Zonal, Zone-redundant, Cross-zone load-balancing
Outbound SNAT algorithm	On-demand	Preallocated
Outbound SNAT front-end selection	Not configurable, multiple candidates	Optional configuration to reduce candidates
Network Security Group	Optional on NIC/subnet	Required

同样，对应的 Public IP 也是 Basic SKU，与 Standard SKU 相比也有一定的局限：

Public IP	Basic	Standard
Availability Zones scenarios	Zonal only	Zone-redundant (default), zonal (optional)
Fast IP Mobility	Not supported	Available
VIP Availability	Not supported	Available
Counters	Not supported	Available
Network Security Group	Optional on NIC	Required

在创建 Service 时，可以通过 `metadata.annotation` 来自定义 Azure 负载均衡的行为，可选的选项包括

Annotation	功能
service.beta.kubernetes.io/azure-load-balancer-internal	如果设置，则创建内网负载均衡
service.beta.kubernetes.io/azure-load-balancer-internal-subnet	设置内网负载均衡 IP 使用的子网
service.beta.kubernetes.io/azure-load-balancer-mode	设置如何为负载均衡选择所属的 AvailabilitySet（之所以有该选项是因为在 Azure 的每个 AvailabilitySet 中只能创建最多一个外网负载均衡和一个内网负载均衡）。可选项为：（1）不设置或者设置为空，使用 `/etc/kubernetes/azure.json` 中设置的 `primaryAvailabilitySet`；（2）设置为 `auto`，选择负载均衡规则最少的 AvailabilitySet；（3）设置为 `as1,as2`，指定 AvailabilitySet 列表
service.beta.kubernetes.io/azure-dns-label-name	设置后为公网 IP 创建 外网 DNS
service.beta.kubernetes.io/azure-shared-securityrule	如果设置，则为多个 Service 共享相同的 NSG 规则。注意该选项需要 Augmented Security Rules
service.beta.kubernetes.io/azure-load-balancer-resource-group	当为 Service 指定公网 IP 并且该公网 IP 与 Kubernetes 集群不在同一个 Resource Group 时，需要使用该 Annotation 指定公网 IP 所在的 Resource Group

在 Kubernetes 中，负载均衡的创建逻辑都在 kube-controller-manager 中，因而排查负载均衡相关的问题时，除了查看 Service 自身的状态，如

```
kubectl describe service
```

还需要查看 kube-controller-manager 是否有异常发生：

```
PODNAME=$(kubectl -n kube-system get pod -l component=kube-contro
kubectl -n kube-system logs $PODNAME --tail 100
```

LoadBalancer Service 一直处于 pending 状态

查看 Service `kubectl describe service` 没有错误信息，但 EXTERNAL-IP 一直是 `<pending>`，说明 Azure Cloud Provider 在创建 LB/NSG/PublicIP 过程中出错。一般按照前面的步骤查看 kube-controller-manager 可以查到具体失败的原因，可能的因素包括

- clientId、clientSecret、tenandId 或 subscriptionId 配置错误导致 Azure API 认证失败：更新所有节点的 `/etc/kubernetes/azure.json`，修复错误的配置即可恢复服务
- 配置的客户端无权管理 LB/NSG/PublicIP/VM：可以为使用的 clientId 增加授权或创建新的 `az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions//resourceGroups/"`
- Kuberentes v1.8.X 中还有可能出现 `Security rule must specify SourceAddressPrefixes, SourceAddressPrefix, or SourceApplicationSecurityGroups` 的错误，这是由于 Azure Go SDK 的问题导致的，可以通过升级集群到 v1.9.X/v1.10.X 或者将 SourceAddressPrefixes 替换为多条 SourceAddressPrefix 规则来解决

负载均衡公网 IP 无法访问

Azure Cloud Provider 会为负载均衡器创建探测器，只有探测正常的服务才可以响应用户的请求。负载均衡公网 IP 无法访问一般是探测失败导致的，可能原因有：

- 后端 VM 本身不正常（可以重启 VM 恢复）
- 后端容器未监听在设置的端口上（可通过配置正确的端口解决）
- 防火墙或网络安全组阻止了要访问的端口（可通过增加安全规则解决）
- 当使用内网负载均衡时，从同一个 ILB 的后端 VM 上访问 ILB VIP 时也会失败，这是 Azure 的预期行为（此时可以访问 service 的 clusterIP）
- 后端容器不响应（部分或者全部）外部请求时也会导致负载均衡 IP 无法访问。注意这里包含**部分容器不响应的场景**，这是由于 Azure 探测器与 Kubernetes 服务发现机制共同导致的结果：
  - （1）Azure 探测器定期去访问 service 的端口（即 NodeIP:NodePort）
  - （2）Kubernetes 将其负载均衡到后端容器中
  - （3）当负载均衡到异常容器时，访问失败会导致探测失败，进而 Azure 可能会将 VM 移出负载均衡
  - 该问题的解决方法是使用健康探针，保证异常容器自动从服务的后端（endpoints）中删除。

内网负载均衡 BackendPool 为空

Kubernetes 1.9.0-1.9.3 中会有这个问题（kubernetes#59746 kubernetes#60060 acs-engine#2151），这是由于一个查找负载均衡所属 AvaibilitySet 的缺陷导致的。

该问题的修复（kubernetes#59747 kubernetes#59083）将包含到 v1.9.4 和 v1.10 中。

外网负载均衡均衡 BackendPool 为空

在使用不支持 Cloud Provider 的工具（如 kubeadm）部署的集群中，如果未给 Kubelet 配置 `--cloud-provider=azure --cloud-config=/etc/kubernetes/cloud-config`，那么 Kubelet 会以 hostname 将其注

册到集群中。此时，查看该 Node 的信息（kubectl get node -o yaml），可以发现其 externalID 与 hostname 相同。此时，kube-controller-manager 也无法将其加入到负载均衡的后端中。

一个简单的确认方式是查看 Node 的 externalID 和 name 是否不同：

```
$ kubectl get node -o jsonpath='{.items[*].metadata.name}'
k8s-agentpool1-27347916-0
$ kubectl get node -o jsonpath='{.items[*].spec.externalID}'
/subscriptions//resourceGroups//providers/Microsoft.Compute/virtu
```

该问题的解决方法是先删除 Node `kubectl delete node`，为 Kubelet 配置 `--cloud-provider=azure --cloud-config=/etc/kubernetes/cloud-config`，最后再重启 Kubelet。

Service 删除后 Azure 公网 IP 未自动删除

Kubernetes 1.9.0-1.9.3 中会有这个问题（kubernetes#59255）：当创建超过 10 个 LoadBalancer Service 后有可能会碰到由于超过 FrontendIPConfiguations Quota（默认为 10）导致负载均衡无法创建的错误。此时虽然负载均衡无法创建，但公网 IP 已经创建成功了，由于 Cloud Provider 的缺陷导致删除 Service 后公网 IP 却未删除。

该问题的修复（kubernetes#59340）将包含到 v1.9.4 和 v1.10 中。

另外，超过 FrontendIPConfiguations Quota 的问题可以参考 Azure subscription and service limits, quotas, and constraints 增加 Quota 来解决。

MSI 无法使用

配置 `"useManagedIdentityExtension": true` 后，可以使用 Managed Service Identity (MSI) 来管理 Azure API 的认证授权。但由于 Cloud Provider 的缺陷（kubernetes #60691 未定义 `useManagedIdentityExtension` yaml 标签导致无法解析该选项。

该问题的修复（kubernetes#60775）将包含在 v1.10 中。

Azure ARM API 调用请求过多

有时 kube-controller-manager 或者 kubelet 会因请求调用过多而导致 Azure ARM API 失败的情况，比如

```
"OperationNotAllowed",\r\n    "message": "The server rejected the
```

特别是在 Kubernetes 集群创建或者批量增加 Nodes 的时候。从 v1.9.2 和 v1.10 开始， Azure cloud provider 为一些列的 Azure 资源（如 VM、VMSS、安全组和路由表等）增加了缓存，大大缓解了这个问题。

一般来说，如果该问题重复出现可以考虑

- 使用 Azure instance metadata，即为所有 Node 的 `/etc/kubernetes/azure.json` 设置 `"useInstanceMetadata": true` 并重启 kubelet
- 为 kube-controller-manager 增大 `--route-reconciliation-period`（默认为 10s），比如在 `/etc/kubernetes/manifests/kube-controller-manager.yaml` 中设置 `--route-reconciliation-period=1m` 后 kubelet 会自动重新创建 kube-controller-manager Pod。

AKS kubectl logs connection timed out

`kubectl logs` 命令报 `getsockopt: connection timed out` 的错误（AKS#232）：

```
$ kubectl --v=8 logs x
I0308 10:32:21.539580   26486 round_trippers.go:417] curl -k -v -
I0308 10:34:32.790295   26486 round_trippers.go:436] GET https://
I0308 10:34:32.790356   26486 round_trippers.go:442] Response Hea
I0308 10:34:32.790376   26486 round_trippers.go:445]     Content-
I0308 10:34:32.790390   26486 round_trippers.go:445]     Content-
I0308 10:34:32.790414   26486 round_trippers.go:445]     Date: Th
I0308 10:34:32.790504   26486 request.go:836] Response Body: {"ki
I0308 10:34:32.790999   26486 helpers.go:207] server response obj
  "metadata": {},
  "status": "Failure",
  "message": "Get https://aks-nodepool1-53392281-1:10250/containe
  "code": 500
}]
F0308 10:34:32.791043   26486 helpers.go:120] Error from server:
```

在 AKS 中，kubectl logs，exec，and attach 等命令需要 Master 与 Nodes 节点之间建立隧道连接。在 `kube-system` namespace 中可以看到 `tunnelfront` 和 `kube-svc-redirect` Pod：

```
$ kubectl -n kube-system get po -l component=tunnel
NAME                            READY     STATUS    RESTARTS   AGE
tunnelfront-7644cd56b7-l5jmc   1/1       Running   0          2d

$ kubectl -n kube-system get po -l component=kube-svc-redirect
NAME                        READY     STATUS    RESTARTS   AGE
kube-svc-redirect-pq6kf   1/1       Running   0          2d
kube-svc-redirect-x6sq5   1/1       Running   0          2d
kube-svc-redirect-zjl7x   1/1       Running   1          2d
```

如果它们不是处于 `Running` 状态或者 Exec/Logs/PortForward 等命令报 `net/http: TLS handshake timeout` 错误，删除 `tunnelfront` Pod，稍等一会就会自动创建新的出来，如：

```
$ kubectl -n kube-system delete po -l component=tunnel
pod "tunnelfront-7644cd56b7-l5jmc" deleted
```

使用 Virtual Kubelet 后 LoadBalancer Service 无法分配公网 IP

使用 Virtual Kubelet 后，LoadBalancer Service 可能会一直处于 pending 状态，无法分配 IP 地址。查看该服务的事件（如 `kubectl describe svc`）会发现错误 `CreatingLoadBalancerFailed 4m (x15 over 45m) service-controller Error creating load balancer (will retry): failed to ensure load balancer for service default/nginx: ensure(default/nginx): lb(kubernetes) - failed to ensure host in pool: "instance not found"` 。这是由于 Virtual Kubelet 创建的虚拟 Node 并不存在于 Azure 云平台中，因而无法将其加入到 Azure Load Balancer 的后端中。

解决方法是开启 ServiceNodeExclusion 特性，即设置 `kube-controller-manager --feature-gates=ServiceNodeExclusion=true` 。开启后，所有带有 `alpha.service-controller.kubernetes.io/exclude-balancer` 标签的 Node 都不会加入到云平台负载均衡的后端中。

注意该特性仅适用于 Kubernetes 1.9 及以上版本。

Node 的 GPU 数总是 0

当在 AKS 集群中运行 GPU 负载时，发现它们无法调度，这可能是由于 Node 容量中的 `nvidia.com/gpu` 总是0。

解决方法是重新部署 nvidia-gpu 设备插件扩展：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    kubernetes.io/cluster-service: "true"
  name: nvidia-device-plugin
  namespace: kube-system
spec:
  template:
    metadata:
      # Mark this pod as a critical add-on; when enabled, the cri
      # reserves resources for critical add-on pods so that they
      # a failure.  This annotation works in tandem with the tole
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        name: nvidia-device-plugin-ds
    spec:
      tolerations:
      # Allow this pod to be rescheduled while the node is in "cr
      # This, along with the annotation above marks this pod as a
      - key: CriticalAddonsOnly
        operator: Exists
      containers:
      - image: nvidia/k8s-device-plugin:1.10
        name: nvidia-device-plugin-ctr
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop: ["ALL"]
        volumeMounts:
          - name: device-plugin
            mountPath: /var/lib/kubelet/device-plugins
      volumes:
        - name: device-plugin
          hostPath:
            path: /var/lib/kubelet/device-plugins
      nodeSelector:
        beta.kubernetes.io/os: linux
        accelerator: nvidia
```

参考文档

- Azure subscription and service limits, quotas, and constraints
- Virtual Kubelet - Missing Load Balancer IP addresses for services
- Troubleshoot Azure Load Balancer
- Troubleshooting CustomScriptExtension (CSE) and acs-engine

排错工具

本章主要介绍在 Kubernetes 排错中常用的工具。

必备工具

- `kubectl`：用于查看 Kubernetes 集群以及容器的状态，如 `kubectl describe pod`
- `journalctl`：用于查看 Kubernetes 组件日志，如 `journalctl -u kubelet -l`
- `iptables` 和 `ebtables`：用于排查 Service 是否工作，如 `iptables -t nat -nL` 查看 kube-proxy 配置的 iptables 规则是否正常
- `tcpdump`：用于排查容器网络问题，如 `tcpdump -nn host 10.240.0.8`
- `perf`：Linux 内核自带的性能分析工具，常用来排查性能问题，如 Container Isolation Gone Wrong 问题的排查

sysdig

sysdig 是一个容器排错工具，提供了开源和商业版本。对于常规排错来说，使用开源版本即可。

除了 sysdig，还可以使用其他两个辅助工具

- csysdig：与 sysdig 一起自动安装，提供了一个命令行界面
- sysdig-inspect：为 sysdig 保存的跟踪文件（如 `sudo sysdig -w filename.scap`）提供了一个图形界面（非实时）

安装

```
# on Ubuntu
curl -s https://s3.amazonaws.com/download.draios.com/DRAIOS-GPG-K
curl -s -o /etc/apt/sources.list.d/draios.list http://download.dr
apt-get update
apt-get -y install linux-headers-$(uname -r)
apt-get -y install sysdig

# on REHL
rpm --import https://s3.amazonaws.com/download.draios.com/DRAIOS-
curl -s -o /etc/yum.repos.d/draios.repo http://download.draios.co
rpm -i http://mirror.us.leaseweb.net/epel/6/i386/epel-release-6-8
yum -y install kernel-devel-$(uname -r)
yum -y install sysdig

# on MacOS
brew install sysdig
```

示例

```
# Refer https://www.sysdig.org/wiki/sysdig-examples/.
# View the top network connections
sudo sysdig -pc -c topconns
# View the top network connections inside the wordpress1 containe
sudo sysdig -pc -c topconns container.name=wordpress1

# Show the network data exchanged with the host 192.168.0.1
sudo sysdig fd.ip=192.168.0.1
sudo sysdig -s2000 -A -c echo_fds fd.cip=192.168.0.1

# List all the incoming connections that are not served by apache
sudo sysdig -p"%proc.name %fd.name" "evt.type=accept and proc.nam

# View the CPU/Network/IO usage of the processes running inside t
sudo sysdig -pc -c topprocs_cpu container.id=2e854c4525b8
sudo sysdig -pc -c topprocs_net container.id=2e854c4525b8
sudo sysdig -pc -c topfiles_bytes container.id=2e854c4525b8

# See the files where apache spends the most time doing I/O
sudo sysdig -c topfiles_time proc.name=httpd

# Show all the interactive commands executed inside a given conta
sudo sysdig -pc -c spy_users

# Show every time a file is opened under /etc.
sudo sysdig evt.type=open and fd.name

# View the list of processes with container context
sudo csysdig -pc
```

更多示例和使用方法可以参考 Sysdig User Guide。

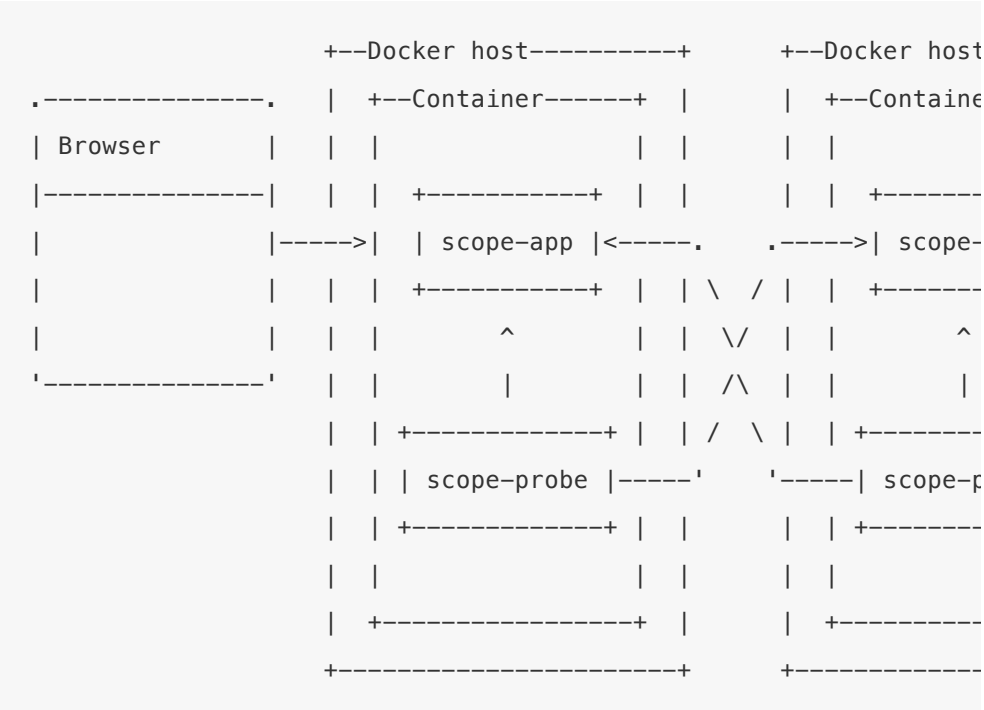
Weave Scope

Weave Scope 是另外一款可视化容器监控和排错工具。与 sysdig 相比，它没有强大的命令行工具，但提供了一个简单易用的交互界面，自动描绘了整个集群的拓扑，并可以通过插件扩展其功能。从其官网的介绍来看，其提供的功能包括

- 交互式拓扑界面
- 图形模式和表格模式
- 过滤功能
- 搜索功能
- 实时度量
- 容器排错
- 插件扩展

Weave Scope 由 App 和 Probe 两部分组成，它们

- Probe 负责收集容器和宿主的信息，并发送给 App
- App 负责处理这些信息，并生成相应的报告，并以交互界面的形式展示



安装

```
kubectl apply -f "https://cloud.weave.works/k8s/scope.yaml?k8s-ve
```

查看界面

安装完成后，可以通过 `weave-scope-app` 来访问交互界面

```
kubectl -n weave get service weave-scope-app
kubectl -n weave port-forward service/weave-scope-app :80
```

点击 Pod，还可以查看该 Pod 所有容器的实时状态和度量数据：

已知问题

在 Ubuntu 内核 4.4.0 上面开启 `--probe.ebpf.connections` 时（默认开启），Node 有可能会因为内核问题而不停重启：

```
[ 263.736006] CPU: 0 PID: 6309 Comm: scope Not tainted 4.4.0-119-
[ 263.736006] Hardware name: Microsoft Corporation Virtual Machir
[ 263.736006] task: ffff88011cef5400 ti: ffff88000a0e4000 task.ti
[ 263.736006] RIP: 0010:[] [] bpf_map_lookup_elem+0x6/0x20
[ 263.736006] RSP: 0018:ffff88000a0e7a70 EFLAGS: 00010082
[ 263.736006] RAX: ffffffff8117cd70 RBX: ffffc90000762068 RCX: 00
[ 263.736006] RDX: 0000000000000000 RSI: ffff88000a0e7cd8 RDI: 00
[ 263.736006] RBP: ffff88000a0e7cf8 R08: 0000000005080021 R09: 00
[ 263.736006] R10: 0000000000000020 R11: ffff880159e1c700 R12: 00
[ 263.736006] R13: ffff88011cfaf400 R14: ffff88000a0e7e38 R15: ff
[ 263.736006] FS: 00007f5b0cd79700(0000) GS:ffff88015b600000(0000
[ 263.736006] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 263.736006] CR2: 000000001cdee3a8 CR3: 000000011ce04000 CR4: 00
[ 263.736006] Stack:
[ 263.736006] ffff88000a0e7cf8 ffffffff81177411 0000000000000000
[ 263.736006] 000000001cdee380 ffff88000a0e7cd8 0000000000000000
[ 263.736006] 0000000005080021 ffff88000a0e7e38 0000000000000000
[ 263.736006] Call Trace:
[ 263.736006] [] ? __bpf_prog_run+0x7a1/0x1360
[ 263.736006] [] ? update_curr+0x79/0x170
[ 263.736006] [] ? update_cfs_shares+0xbc/0x100
[ 263.736006] [] ? update_curr+0x79/0x170
[ 263.736006] [] ? dput+0xb8/0x230
[ 263.736006] [] ? follow_managed+0x265/0x300
[ 263.736006] [] ? kmem_cache_alloc_trace+0x1d4/0x1f0
[ 263.736006] [] ? seq_open+0x5a/0xa0
[ 263.736006] [] ? probes_open+0x33/0x100
[ 263.736006] [] ? dput+0x34/0x230
[ 263.736006] [] ? mntput+0x24/0x40
[ 263.736006] [] trace_call_bpf+0x37/0x50
[ 263.736006] [] kretprobe_perf_func+0x3d/0x250
[ 263.736006] [] ? pre_handler_kretprobe+0x135/0x1b0
[ 263.736006] [] kretprobe_dispatcher+0x3d/0x60
[ 263.736006] [] ? do_sys_open+0x1b2/0x2a0
[ 263.736006] [] ? kretprobe_trampoline_holder+0x9/0x9
[ 263.736006] [] trampoline_handler+0x133/0x210
[ 263.736006] [] ? do_sys_open+0x1b2/0x2a0
[ 263.736006] [] kretprobe_trampoline+0x25/0x57
[ 263.736006] [] ? kretprobe_trampoline_holder+0x9/0x9
```

```
[ 263.736006] [] SyS_openat+0x14/0x20
[ 263.736006] [] entry_SYSCALL_64_fastpath+0x1c/0xbb
```

解决方法有两种

- 禁止 eBPF 探测，如 `--probe.ebpf.connections=false`
- 升级内核，如升级到 4.13.0

参考文档

- Overview of kubectl
- Monitoring Kuberietes with sysdig

社区贡献

开发指南

配置开发环境

以 Ubuntu 为例，配置一个 Kubernetes 的开发环境

```
apt-get install -y gcc make socat git build-essential

# 安装 Docker
sh -c 'echo"deb https://apt.dockerproject.org/repo ubuntu-$(lsb_r
curl -fsSL https://apt.dockerproject.org/gpg | sudo apt-key add -
apt-key fingerprint 58118E89F3A912897C070ADBF76221572C52609D
apt-get update
apt-get -y install "docker-engine=1.13.1-0~ubuntu-$(lsb_release -

# 安装 etcd
ETCD_VER=v3.2.18
DOWNLOAD_URL="https://github.com/coreos/etcd/releases/download"
curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
sudo /bin/cp -f etcd-${ETCD_VER}-linux-amd64/{etcd,etcdctl} /usr/
rm -rf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz etcd-${ETCD_VER}-

# 安装 Go
curl -sL https://storage.googleapis.com/golang/go1.10.2.linux-amd
export GOPATH=/gopath
export PATH=$PATH:$GOPATH/bin:/usr/local/bin:/usr/local/go/bin/

# 下载 Kubernetes 代码
mkdir -p $GOPATH/src/k8s.io
git clone https://github.com/kubernetes/kubernetes $GOPATH/src/k8
cd $GOPATH/src/k8s.io/kubernetes

# 启动一个本地集群
export KUBERNETES_PROVIDER=local
hack/local-up-cluster.sh
```

打开另外一个终端，配置 `kubectl` 之后就可以开始使用了：

```
cd $GOPATH/src/k8s.io/kubernetes
export KUBECONFIG=/var/run/kubernetes/admin.kubeconfig
cluster/kubectl.sh
```

单元测试

单元测试是 Kubernetes 开发中不可缺少的，一般在代码修改的同时还要更新或添加对应的单元测试。这些单元测试大都支持在不同的系统上直接运行，比如 OSX、Linux 等。

比如，加入修改了 `pkg/kubelet/kuberuntime` 的代码后，

```
# 可以加上 Go package 的全路径来测试
go test -v k8s.io/kubernetes/pkg/kubelet/kuberuntime
# 也可以用相对目录
go test -v ./pkg/kubelet/kuberuntime
```

端到端测试

端到端（e2e）测试需要启动一个 Kubernetes 集群，仅支持在 Linux 系统上运行。

本地运行方法示例：

```
make WHAT='test/e2e/e2e.test'
make ginkgo

export KUBERNETES_PROVIDER=local
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Port\sfor
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Feature:S
```

> 注：Kubernetes 的每个 PR 都会自动运行一系列的 e2e 测试。

Node e2e 测试

Node e2e 测试需要启动 Kubelet，目前仅支持在 Linux 系统上运行。

```
export KUBERNETES_PROVIDER=local
make test-e2e-node FOCUS="InitContainer"
```

> 注：Kubernetes 的每个 PR 都会自动运行 node e2e 测试。

有用的 git 命令

很多时候，我们需要把 Pull Request 拉取到本地来测试，比如拉取 Pull Request #365 的方法为

```
git fetch upstream pull/365/merge:branch-fix-1
git checkout branch-fix-1
```

当然，也可以配置 `.git/config` 并运行 `git fetch` 拉取所有的 Pull Requests（注意 Kubernetes 的 Pull Requests 非常多，这个过程可能会很慢）：

```
fetch = +refs/pull/*:refs/remotes/origin/pull/*
```

其他参考

- 编译 release 版：`make quick-release`
- 机器人命令：命令列表 和 使用文档。
- Kubernetes TestGrid，包含所有的测试历史
- Kuberentes Submit Queue Status，包含所有的 Pull Request 状态以及合并队列
- Node Performance Dashboard，包含 Node 组性能测试报告
- Kubernetes Performance Dashboard，包含 Density 和 Load 测试报告
- Kubernetes PR Dashboard，包含主要关注的 Pull Request 列表（需要以 Github 登录）
- Jenkins Logs 和 Prow Status，包含所有 Pull Request 的 Jenkins 测试日志

单元测试和集成测试

- Current Test Status
- Aggregated Failures
- Test Grid

单元测试

单元测试仅依赖于源代码，是测试代码逻辑是否符合预期的最简单方法。

运行所有的单元测试

```
make test
```

仅测试指定的 package

```
# 单个 package
make test WHAT=./pkg/api
# 多个 packages
make test WHAT=./pkg/{api,kubelet}
```

或者，也可以直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/kubelet
```

仅测试指定 package 的某个测试 case

```
# Runs TestValidatePod in pkg/api/validation with the verbose fla
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_A

# Runs tests that match the regex ValidatePod|ValidateConfigMap i
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_A
```

或者直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/api/validation -run ^TestValidat
```

并行测试

并行测试是 root out flakes 的一种有效方法：

```
# Have 2 workers run all tests 5 times each (10 total iterations)
make test PARALLEL=2 ITERATION=5
```

生成测试报告

```
make test KUBE_COVER=y
```

Benchmark 测试

```
go test ./pkg/apiserver -benchmem -run=XXX -bench=BenchmarkWatch
```

集成测试

Kubernetes 集成测试需要安装 etcd（只要按照即可，不需要启动），比如

```
hack/install-etcd.sh  # Installs in ./third_party/etcd
echo export PATH="\$PATH:$(pwd)/third_party/etcd" >> ~/.profile
```

集成测试会在需要的时候自动启动 etcd 和 kubernetes 服务，并运行 test/integration 里面的测试。

运行所有集成测试

```
make test-integration  # Run all integration tests.
```

指定集成测试用例

```
# Run integration test TestPodUpdateActiveDeadlineSeconds with th
make test-integration KUBE_GOFLAGS="-v" KUBE_TEST_ARGS="-run ^Tes
```

End to end (e2e) 测试

End to end (e2e) 测试模拟用户行为操作 Kubernetes，用来保证 Kubernetes 服务或集群的行为完全符合设计预期。

在开启 e2e 测试之前，需要先编译测试文件，并设置 KUBERNETES_PROVIDER（默认为 gce）：

```
make WHAT='test/e2e/e2e.test'
make ginkgo
export KUBERNETES_PROVIDER=local
```

启动 cluster，测试，最后停止 cluster

```
# build Kubernetes, up a cluster, run tests, and tear everything
go run hack/e2e.go -- -v --build --up --test --down
```

仅测试指定的用例

```
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Kubectl\s
```

跳过测试用例

```
go run hack/e2e.go -- -v --test --test_args="--ginkgo.skip=Pods.*
```

并行测试

```
# Run tests in parallel, skip any that must be run serially
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--gi

# Run tests in parallel, skip any that must be run serially and k
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--gi
```

清理测试资源

```
go run hack/e2e.go -- -v --down
```

有用的 `-ctl`

```
# -ctl can be used to quickly call kubectl against your e2e clust
# cleaning up after a failed test or viewing logs. Use -v to avoi
# kubectl output.
go run hack/e2e.go -- -v -ctl='get events'
go run hack/e2e.go -- -v -ctl='delete pod foobar'
```

Fedaration e2e 测试

```
export FEDERATION=true
export E2E_ZONES="us-central1-a us-central1-b us-central1-f"
# or export FEDERATION_PUSH_REPO_BASE="quay.io/colin_hom"
export FEDERATION_PUSH_REPO_BASE="gcr.io/${GCE_PROJECT_NAME}"

# build container images
KUBE_RELEASE_RUN_TESTS=n KUBE_FASTBUILD=true go run hack/e2e.go -

# push the federation container images
build/push-federation-images.sh

# Deploy federation control plane
go run hack/e2e.go -- -v --up

# Finally, run the tests
go run hack/e2e.go -- -v --test --test_args="--ginkgo.focus=\[Fea

# Don't forget to teardown everything down
go run hack/e2e.go -- -v --down
```

可以用 `cluster/log-dump.sh` 方便的下载相关日志，帮助排查测试中碰到的问题。

Node e2e 测试

Node e2e 仅测试 Kubelet 的相关功能，可以在本地或者集群中测试

```
export KUBERNETES_PROVIDER=local
make test-e2e-node FOCUS="InitContainer"
make test_e2e_node TEST_ARGS="--experimental-cgroups-per-qos=true
```

补充说明

借助 kubectl 的模版可以方便获取想要的数据，比如查询某个 container 的镜像的方法为

```
kubectl get pods nginx-4263166205-ggst4 -o template '--template={
```

参考文档

- Kubernetes testing
- End-to-End Testing
- Node e2e test
- How to write e2e test
- Coding Conventions

社区贡献

Kubernetes 支持以许多种方式来贡献社区，包括汇报代码缺陷、提交问题修复和功能实现、添加或修复文档、协助用户解决问题等等。

社区结构

Kubernetes 社区由三部分组成

- Steering committe
- Special Interest Groups (SIG)
- Working Groups (WG)

提交 Pull Request 到主分支

当需要修改 Kubernetes 代码时，可以给 Kubernetes 主分支提 Pull Request。这其实是一个标准的 Github 工作流：

一些加快 PR 合并的方法：

- 使用小的提交，将不同功能的代码分拆到不同的提交甚至是不同的 Pull Request 中
- 必要的逻辑添加注释说明变更的理由
- 遵循代码约定，如 Coding Conventions、API Conventions 和 kubectl Conventions
- 确保修改部分可以本地跑过单元测试和功能测试
- 使用 Bot 命令 设置正确的标签或重试失败的测试

提交 Pull Request 到发布分支

发布分支的问题一般是首先在主分支里面修复（发送 Pull Request 到主分支并通过代码审核之后合并），然后通过 cherry-pick 的方式发送 Pull Request 到老的分支（如 `release-1.7` 等）。

对于主分支的 PR，待 Reviewer 添加 `cherrypick-candidate` 标签后就可以开始 cherry-pick 到老的分支了。但首先需要安装一个 Github 发布的 hub 工具，如

```
# on macOS
brew install hub

# on others
go get github.com/github/hub
```

然后执行下面的脚本自动 cherry-pick 并发送 PR 到需要的分支，其中 `upstream/release-1.7` 是要发布的分支，而 `51870` 则是发送到主分支的 PR 号：

```
hack/cherry_pick_pull.sh upstream/release-1.7 51870
```

然后安装输出中的提示操作即可。如果合并过程中发生错误，需要另开一个终端手动合并冲突，并执行 `git add . && git am --continue`，最后再回去继续，直到 PR 发送成功。

注意：提交到发布分支的每个 PR 除了需要正常的代码审核之外，还需要对应版本的 release manager 批准。当前所有版本的 release manager 可以在 这里 找到。

参考文档

如果在社区贡献中碰到问题，可以参考以下指南

- **Kubernetes Contributor Community**
- **Kubernetes Contributor Guide**
- **Kubernetes Developer Guide**
- Special Interest Groups
- Feature Tracking and Backlog
- Community Expectations
- Kubernetes release managers

附录

生态圈

云原生全景

图片来源：https://landscape.cncf.io/。

云原生地图

图片来源：https://github.com/cncf/landscape。

Serverless

图片来源：https://s.cncf.io。

学习资源

官方文档

- Kubernetes官方网站
- Kubernetes文档

- Kubernetes tutorials

在线课程

- edX: Introduction to Kubernetes
- Udacity: Scalable Microservices with Kubernetes
- edX: Fundamentals of Containers, Kubernetes, and Red Hat OpenShift

在线指导

- Kubernetes the hard way
- AWS Workshop for Kubernetes
- Learn Kubernetes using Interactive Browser-Based Scenarios
- Kubernetes Bootcamp
- Kubernetes Certified Administration (CKA) online resources
- Kubernetes By Example
- Kubernetes Learning Resources
- Kubernetes Best Practices

电子书籍

- Designing Distributed Systems
- Kubernetes Handbook (Kubernetes 指南)

国内镜像

Docker Hub 镜像

- Docker 中国镜像：https://registry.docker-cn.com
- 开源社镜像：https://dockerhub.akscn.io

示例

```
docker pull registry.docker-cn.com/library/nginx
```

GCR（Google Container Registry）镜像

- 开源社镜像：https://gcr.akscn.io/google_containers

示例

```
docker pull gcr.akscn.io/google_containers/hyperkube:v1.12.1
docker pull gcr.akscn.io/google_containers/pause-amd64:3.1
```

Kubernetes RPM/DEB镜像

- 开源社镜像

示例：

```
# Ubuntu
cat </etc/apt/sources.list.d/kubernetes.list
deb http://mirror.azure.cn/kubernetes/packages/apt/ kubernetes-xe
EOF
```

Helm Charts 镜像

- Helm: http://mirror.azure.cn/kubernetes/helm/
- Stable Charts: http://mirror.azure.cn/kubernetes/charts/
- Incubator Charts: http://mirror.azure.cn/kubernetes/charts-incubator/

示例

```
helm repo add stable http://mirror.azure.cn/kubernetes/charts/
helm repo add incubator http://mirror.azure.cn/kubernetes/charts-
```

操作系统镜像

- 开源社开源镜像
- 网易开源镜像

以 Ubuntu 18.04 (Bionic) 为例，修改 /etc/apt/sources.list 文件的内容为

```
deb http://azure.archive.ubuntu.com/ubuntu/ bionic main restricte
deb http://azure.archive.ubuntu.com/ubuntu/ bionic-security main
deb http://azure.archive.ubuntu.com/ubuntu/ bionic-updates main r
deb http://azure.archive.ubuntu.com/ubuntu/ bionic-proposed main
deb http://azure.archive.ubuntu.com/ubuntu/ bionic-backports mair
deb-src http://azure.archive.ubuntu.com/ubuntu/ bionic main restr
deb-src http://azure.archive.ubuntu.com/ubuntu/ bionic-security m
deb-src http://azure.archive.ubuntu.com/ubuntu/ bionic-updates ma
deb-src http://azure.archive.ubuntu.com/ubuntu/ bionic-proposed m
deb-src http://azure.archive.ubuntu.com/ubuntu/ bionic-backports
```

如何贡献

1. 在 Github 上 Fork 项目到自己的仓库。
2. 将 fork 后的项目拉到本地：`git clone https://github.com//kubernetes-handbook` 。
3. 新建一个分支，并添加或编辑内容：`git checkout -b new-branch` 。
4. 提交并推送到 github：`git commit -am "comments"; git push` 。
5. 在 Github 上提交 Pull Request。

参考文档

- Kubernetes官方网站
- Kubernetes Documentation
- Discuss Kubernetes
- Kubernetes Contributor Community
  - Kubernetes Contributor Guide
  - Kubernetes Developer Guide
  - Special Interest Groups
  - Kubernetes Features and KEPs

    - Kubernetes release managers
    - Kubernetes submit queue
    - Kubernetes Performance Dashboard
    - Node Performance Dashboard
- CNCF项目贡献统计
- Kubernetes项目贡献统计
- Kubernetes github metrics
- Github public data
- Kubernetes the hard way
- Kubernetes Bootcamp
- Design patterns for container-based distributed systems
- Designing Distributed Systems
- Awesome Kubernetes
- Awesome Docker