

K u b e r n e t e s

H a n d b o o k

kubernetes

指南

作者：倪朋飞



扫一扫，了解更多

Geekbang
极客邦科技

| InfoQ



kubernetes
by Google

卷首语

张磊

Hyper 项目成员，Kubernetes 开发者

这可能是一本最值得推荐的 Kubernetes 电子书。

2017 年 6 月 2 日，在湾区圣何塞召开了一次不太起眼的闭门会议，与会人数并不多，主题叫“Kubernetes Leadership Summit”。

不过，这次聚会随后公开的数据，却很快成为了技术媒体争相报道的热点：根据 Linux 基金会的统计和评分，Kubernetes 在过去的一年里同 Linux、Homebrew 等明星项目一起，跻身为这个星球上最受欢迎的开源项目之一，同领域排名接近的还有前 Docker 项目。

作为 Kubernetes 社区的一位开发者，这样的结果并不让人意外。但是如果考虑到国内外容器圈子在过去的一年里的各种跌宕起伏，Kubernetes 项目能取得这样的成绩也委实不容易。

比较有意思的是，相比于国外容器生态的如火如荼，国内的容器圈子在很多方面还是稍欠火候。随后的几周，我们和 Li Xiang 等几位开发者一同讨论了这些问题，也归纳出一些结论。总的来说，国内的容器技术逐渐繁荣其实一直伴随着资本和创业公司的兴起，这其中 Google 公司由于特殊原因而长期缺位，其实给国内容器技术的认知造成了不小的偏差。

所谓“兼听则明，偏信则暗”。

Google 和 Kubernetes 在国内的技术主体缺位，加上语言障碍和信息壁垒，在很长一段时期里都给国内容器技术生态带来了“一条腿走路”的窘况：我们总是过度关注于容器运行时甚至 Kernel 特性，忽视或者错误估计容器编排管理的位



置和复杂程度。

现如今，国内容器云创业热潮已经告一段落，管理层变动也好，融资并购也罢，都已经成为了过去时。这也难得地为我们带来了一个回头审视技术的良好时机。

所以当倪 飞 (feiskyer) 出于个人兴趣编纂他的《Kubernetes 指南》的时候，我们和 InfoQ 都强烈希望他把这个成果分享给所有关心容器圈子的技术人。这个目的无比单纯：现在国内的容器技术社区，需要一本能及时更新的、深入浅出的、没有任何商业目的和厂商背书的纯粹的技术著作。更何况，他的这本手册，早已在 Github 上收获了几百个 star。更更何况，feiskyer 何许人也？Kubernetes 核心特性 CRI 的主要作者之一，国内屈指可数的核心模块维护者和 Approver，容器网络专家，多年的一线大厂工程经验和 hyper.sh 的核心开发者。要知道，我们在给 OpenStack 基金会捐献 Stackube 项目（原 Hypernetes）时，对方专家是这么评价的：**“Yes, I know feiskyer, so I will never worry about the code stuff.”**

所谓实至名归。

卓越的技术背景，中立的写作目的，高效的工作能力，我们没有理由不把这本《Kubernetes 指南》列为将来推荐给圈内好友的上上之选。甚至，我个人认为这会是一本最值得推荐的 Kubernetes 电子书。

毕竟，像 feiskyer 这么牛的人愿意抽出时间来做这样一件公益性的事情，已经令我们感到庆幸万分了。

版权声明

Kubernetes 指南

@ 倪朋飞

本书版权为本书作者倪朋飞所有，授权 InfoQ 中文站独家发布电子版，未经出版者预先的书面许可，不得以任何方式复制或抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区洛娃大厦 C 座 1607

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn

目录

前言	1.1
如何贡献	1.2
1. Kubernetes 简介	1.3
1.1 基本概念	1.3.1
1.2 Kubernetes 101	1.3.2
1.3 Kubernetes 201	1.3.3
1.4 Kubernetes 集群	1.3.4

核心原理

2. 核心原理	2.1
2.1 架构原理	2.1.1
设计理念	2.1.1.1
2.2 主要概念	2.1.2
Pod	2.1.2.1
Namespace	2.1.2.2
Node	2.1.2.3
Service	2.1.2.4
Volume	2.1.2.5
Persistent Volume	2.1.2.6
Local Volume	2.1.2.7
Deployment	2.1.2.8
Secret	2.1.2.9
StatefulSet	2.1.2.10
DaemonSet	2.1.2.11
ServiceAccount	2.1.2.12
ReplicationController和ReplicaSet	2.1.2.13
Job	2.1.2.14
CronJob	2.1.2.15

SecurityContext和PSP	2.1.2.16
Resource Quota	2.1.2.17
Horizontal Pod Autoscaling	2.1.2.18
Network Policy	2.1.2.19
Ingress	2.1.2.20
ConfigMap	2.1.2.21
PodPreset	2.1.2.22
ThirdPartyResources	2.1.2.23
CustomResourceDefinition	2.1.2.24
3. 核心组件	2.2
3.1 etcd	2.2.1
3.2 API Server	2.2.2
工作原理	2.2.2.1
访问API	2.2.2.2
扩展API	2.2.2.3
3.3 Scheduler	2.2.3
工作原理	2.2.3.1
3.4 Controller Manager	2.2.4
工作原理	2.2.4.1
3.5 kubelet	2.2.5
工作原理	2.2.5.1
Container Runtime	2.2.5.2
3.6 kube-proxy	2.2.6
工作原理	2.2.6.1
3.7 Kube DNS	2.2.7
工作原理	2.2.7.1
3.8 Federation	2.2.8
3.9 kubeadm	2.2.9
3.10 hyperkube	2.2.10
3.11 kubectrl	2.2.11
4. 部署配置	2.3

4.1 单机部署	2.3.1
4.2 集群部署	2.3.2
kubeadm	2.3.2.1
Kubespray	2.3.2.2
Frakti+Hyper	2.3.2.3
CentOS手动部署	2.3.2.4
4.3 kubectl客户端	2.3.3
4.4 附加组件	2.3.4
Dashboard	2.3.4.1
Heapster	2.3.4.2
EFK	2.3.4.3
4.5 推荐配置	2.3.5

插件指南

5. 插件扩展	3.1
5.1 访问控制	3.1.1
RBAC授权	3.1.1.1
准入控制	3.1.1.2
5.2 网络	3.1.2
网络模型和插件	3.1.2.1
CNI	3.1.2.2
CNI介绍	3.1.2.2.1
Flannel	3.1.2.2.2
Weave	3.1.2.2.3
Contiv	3.1.2.2.4
Calico	3.1.2.2.5
SR-IOV	3.1.2.2.6
Romana	3.1.2.2.7
OpenContrail	3.1.2.2.8
CNI Plugin Chains	3.1.2.2.9
5.3 Volume插件	3.1.3

glusterfs	3.1.3.1
5.4 Container Runtime Interface	3.1.4
5.5 Network Policy	3.1.5
5.6 Ingress Controller	3.1.6
Traefik	3.1.6.1
Traefik Ingress	3.1.6.1.1
负载测试	3.1.6.1.2
网络测试	3.1.6.1.3
边缘节点配置	3.1.6.1.4
minikube Ingress	3.1.6.2
5.7 Cloud Provider	3.1.7
5.8 Scheduler扩展	3.1.8

实践案例

6. 应用管理	4.1
一般准则	4.1.1
滚动升级	4.1.2
Helm	4.1.3
Helm参考	4.1.3.1
Draft	4.1.4
Operator	4.1.5
Deis workflow	4.1.6
Kompose	4.1.7
Istio	4.1.8
7. 实践案例	4.2
监控	4.2.1
日志	4.2.2
高可用	4.2.3
调试	4.2.4
端口映射	4.2.5
端口转发	4.2.6

GPU	4.2.7
容器安全	4.2.8
审计	4.2.9

开发与社区贡献

8. 开发指南	5.1
开发环境搭建	5.1.1
单元测试和集成测试	5.1.2
社区贡献	5.1.3

附录

9. 附录	6.1
生态圈	6.1.1
Play-With-Kubernetes	6.1.2
FAQ	6.1.3
参考文档	6.1.4

Kubernetes指南

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。

注：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。

在线阅读

可以通过[GitBook](#)或者[Github](#)来在线阅读。

也可以下载[ePub](#)或者[PDF](#)版本。

项目源码

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

贡献者

欢迎参与维护项目，贡献方法参考[这里](#)。感谢所有的贡献者，贡献者列表见[这里](#)。

如何贡献

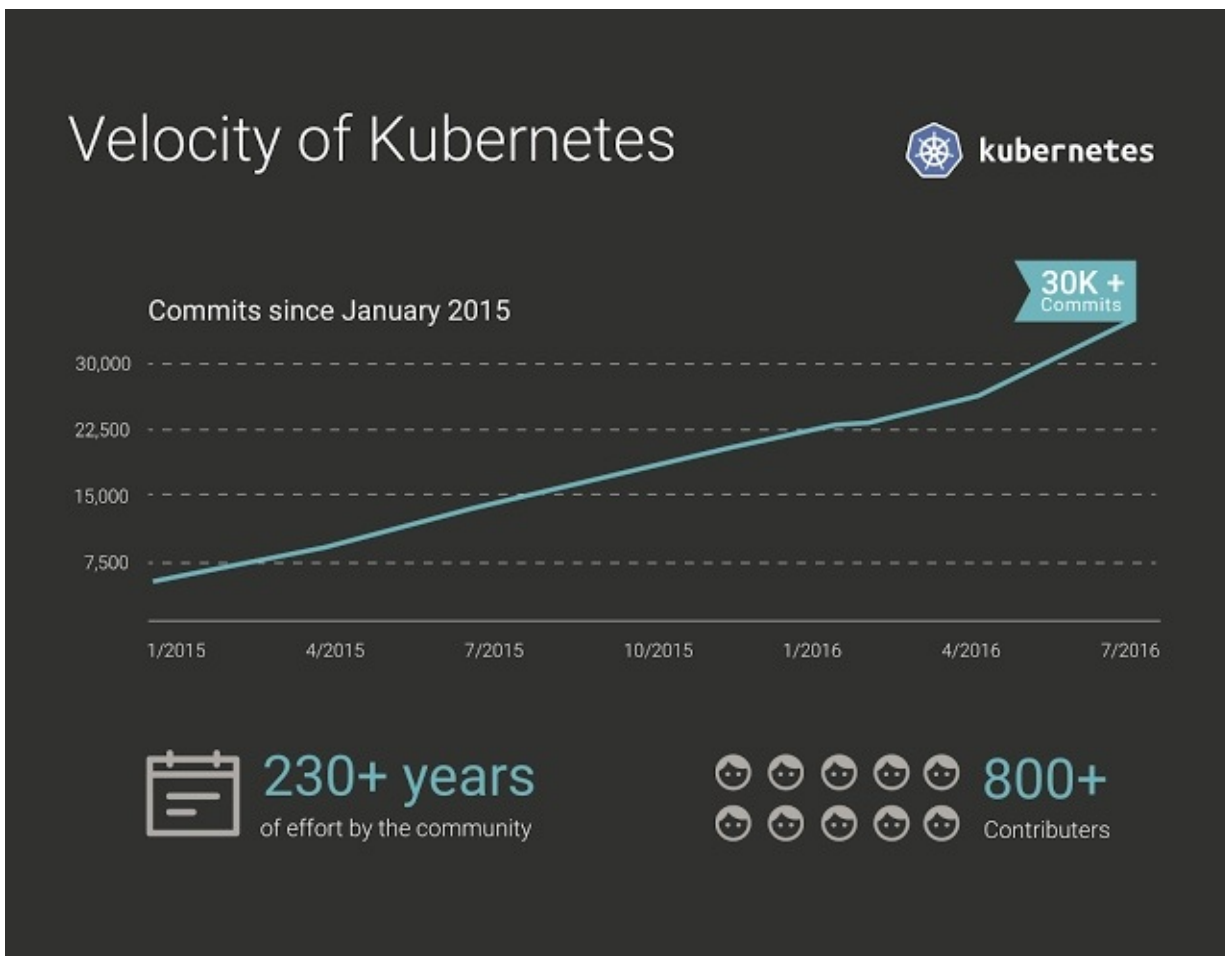
1. 在Github上[Fork](#)到自己的仓库
2. 将fork后的项目拉到本地: `git clone https://github.com/<user-name>/kubernetes-handbook`
3. 新建一个分支，并添加或编辑内容: `git checkout -b new-branch`
4. 提交并推送到github: `git commit -am "comments"; git push`
5. 在Github上提交Pull Request。

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

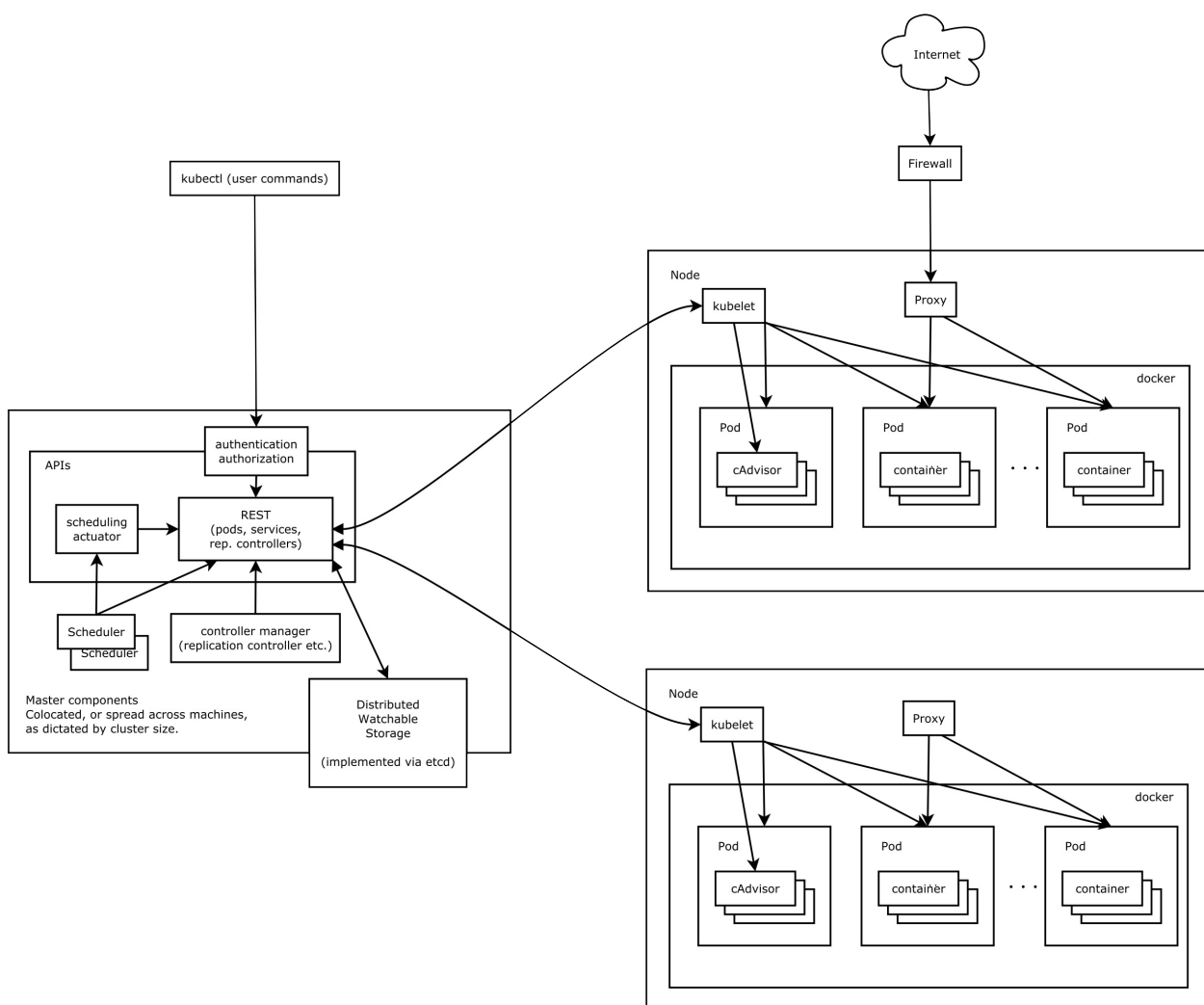
Kubernetes 发展非常迅速，已经成为容器编排领域的领导者。



Kubernetes 架构

Kubernetes主要由以下几个核心组件组成：

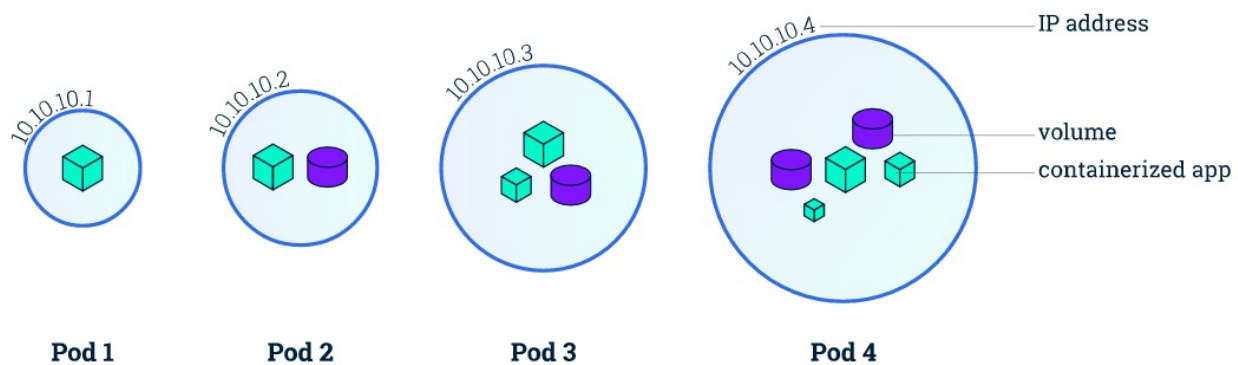
- etcd保存了整个集群的状态；
- apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- kubelet负责维护容器的生命周期，同时也负责Volume（CVI）和网络（CNI）的管理；
- Container runtime负责镜像管理以及Pod和容器的真正运行（CRI）；
- kube-proxy负责为Service提供cluster内部的服务发现和负载均衡



Kubernetes 基本概念

Pod

Pod是一组紧密关联的容器集合，它们共享IPC、Network和UTC 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。

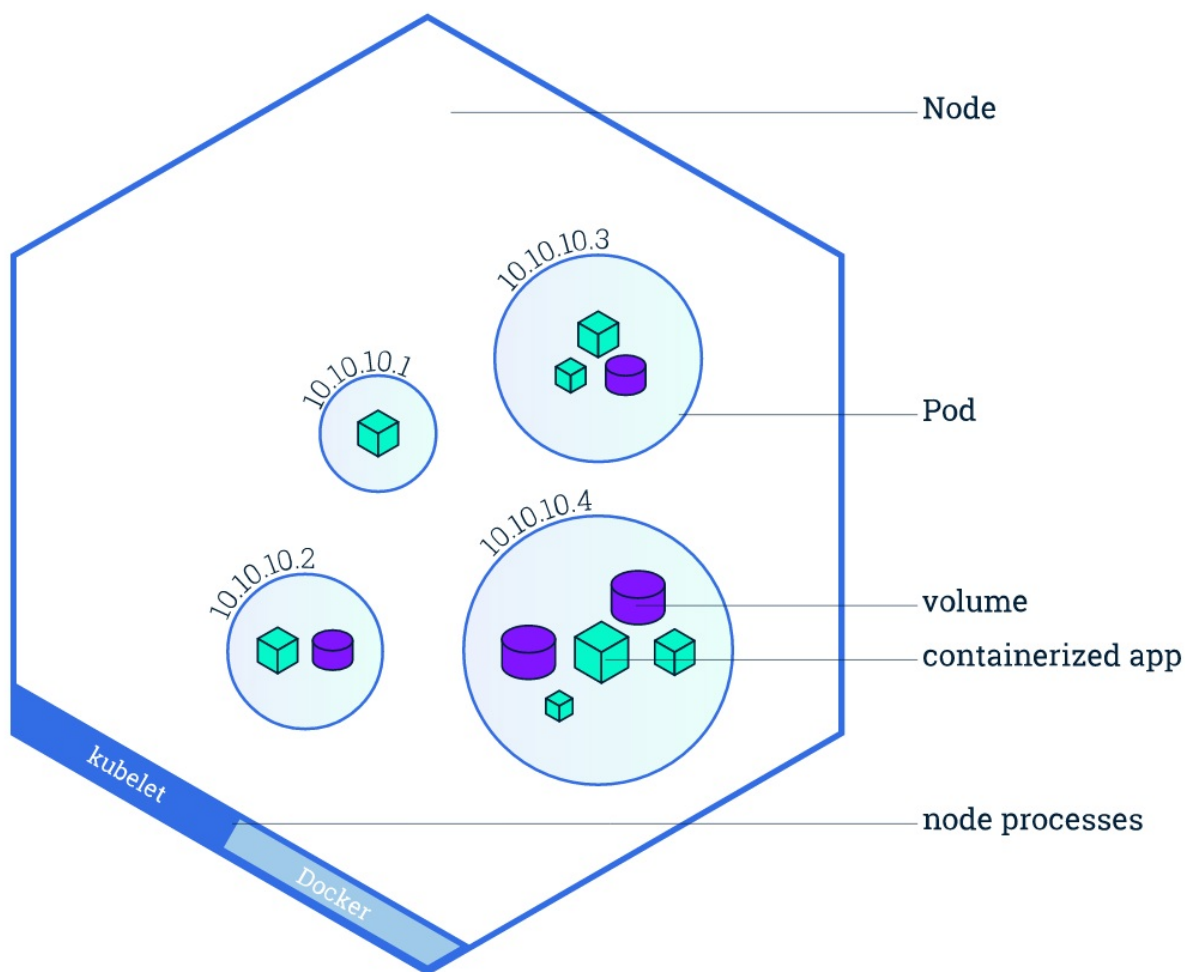


在Kubernetes中，所有对象都使用manifest（yaml或json）来定义，比如一个简单的nginx服务可以定义为nginx.yaml，它包含一个镜像为nginx的容器：

```
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。



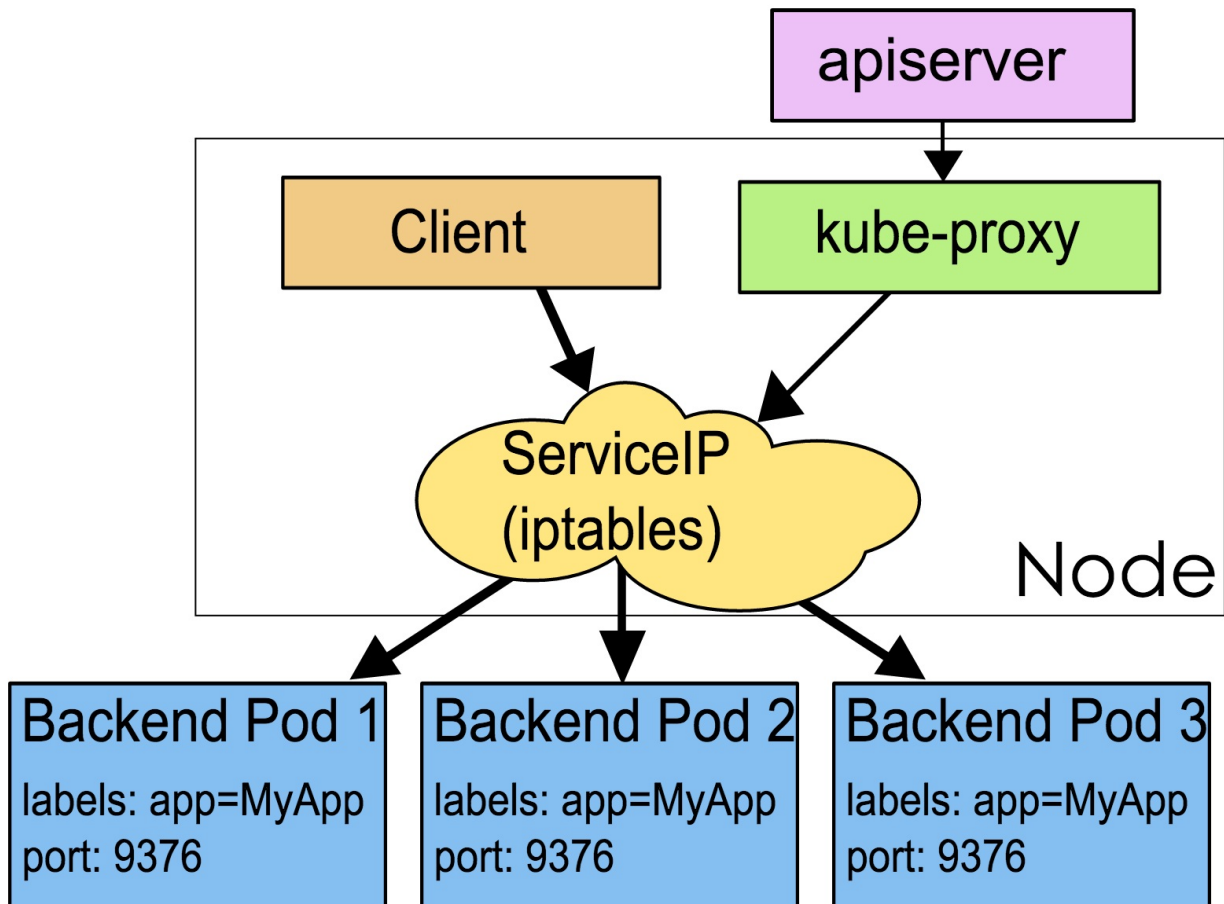
Namespace

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。

Kubernetes 101

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 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	COMM
root	1	0.0	0.5	31736	5108	?	Ss	00:19	0:00	nginx: master process nginx -g daemon off;
nginx	5	0.0	0.2	32124	2844	?	S	00:19	0:00	nginx: worker process
root	18	0.0	0.2	17500	2112	?	Rs	00:25	0:00	ps aux

```
$ 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://4ef989b57d0a7638ad9c5bbc22e16
d5ea5b459281c77074fc982eba50973107f
    Image:         nginx
    Image ID:      docker://sha256:4efb2fcdb1ab05fb03c9435234
343c1cc65289eeb016be86193e88d3a5d84f6b
    Port:          80/TCP
    State:         Running
      Started:     Tue, 06 Sep 2016 08:19:30 +0800
    Ready:         True
    Restart Count: 0
    Environment Variables:  <none>
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    S
ubobjectPath    Type    Reason    Message
  -----
-----
  8m            8m            1    {default-scheduler }
Normal    Scheduled    Successfully assigne
d nginx-app-4028413181-cnt1i to boot2docker
  8m            8m            1    {kubelet boot2docker}    s
pec.containers{nginx-app}    Normal    Pulling    pulli
ng image "nginx"
  7m            7m            1    {kubelet boot2docker}    s
pec.containers{nginx-app}    Normal    Pulled     Succe

```

```

ssfully pulled image "nginx"
 7m          7m          1          {kubelet boot2docker}      s
pec.containers{nginx-app}          Normal          Created          Creat
ed container with docker id 4ef989b57d0a
 7m          7m          1          {kubelet boot2docker}      s
pec.containers{nginx-app}          Normal          Started          Start
ed container with docker id 4ef989b57d0a

$ kubectl logs nginx-app-4028413181-cnt1i
127.0.0.1 - - [06/Sep/2016:00:27:13 +0000] "GET / HTTP/1.0 " 200 612
"_" "_" "_"
127.0.0.1 - - [06/Sep/2016:00:27:15 +0000] "GET / HTTP/1.0 " 200 612
"_" "_" "_"

```

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Deployment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：

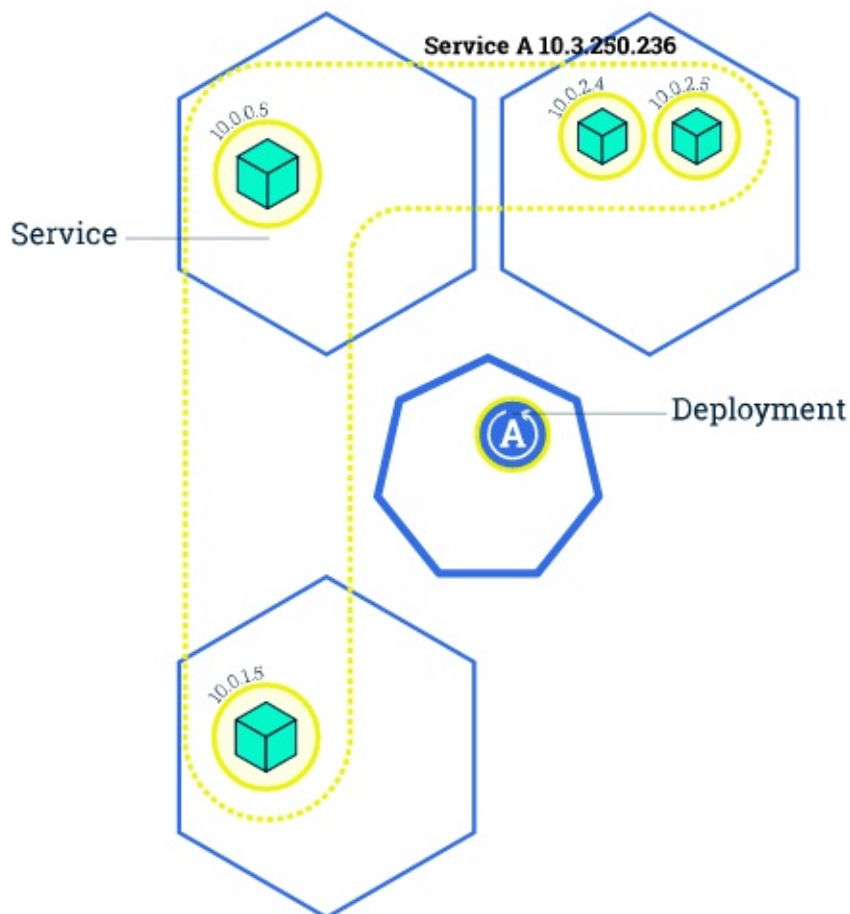
```
$ kubectl expose deployment nginx-app --type=NodePort --port=80 --target-port=80
service "nginx-app" exposed
$ kubectl describe service nginx-app
Name:          nginx-app
Namespace:     default
Labels:        run=nginx-app
Selector:      run=nginx-app
Type:          NodePort
IP:            10.0.0.66
Port:          <unset>      80/TCP
NodePort:      <unset>      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。

Kubernetes 201

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：



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
```

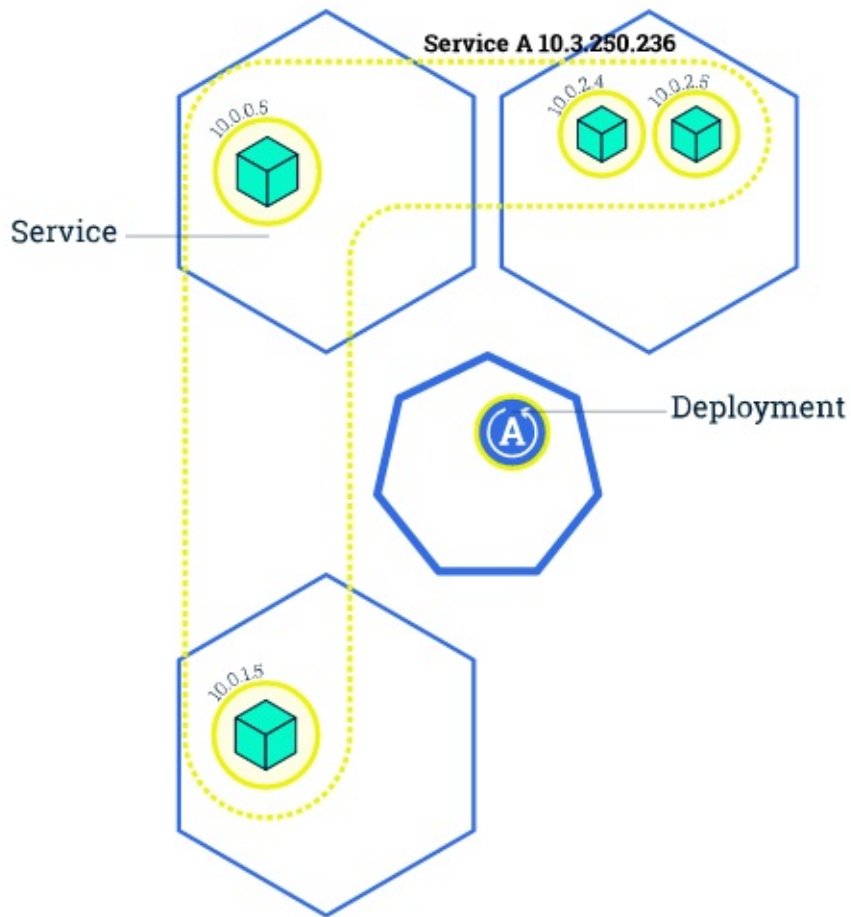
```
$ kubectl get deploy
```

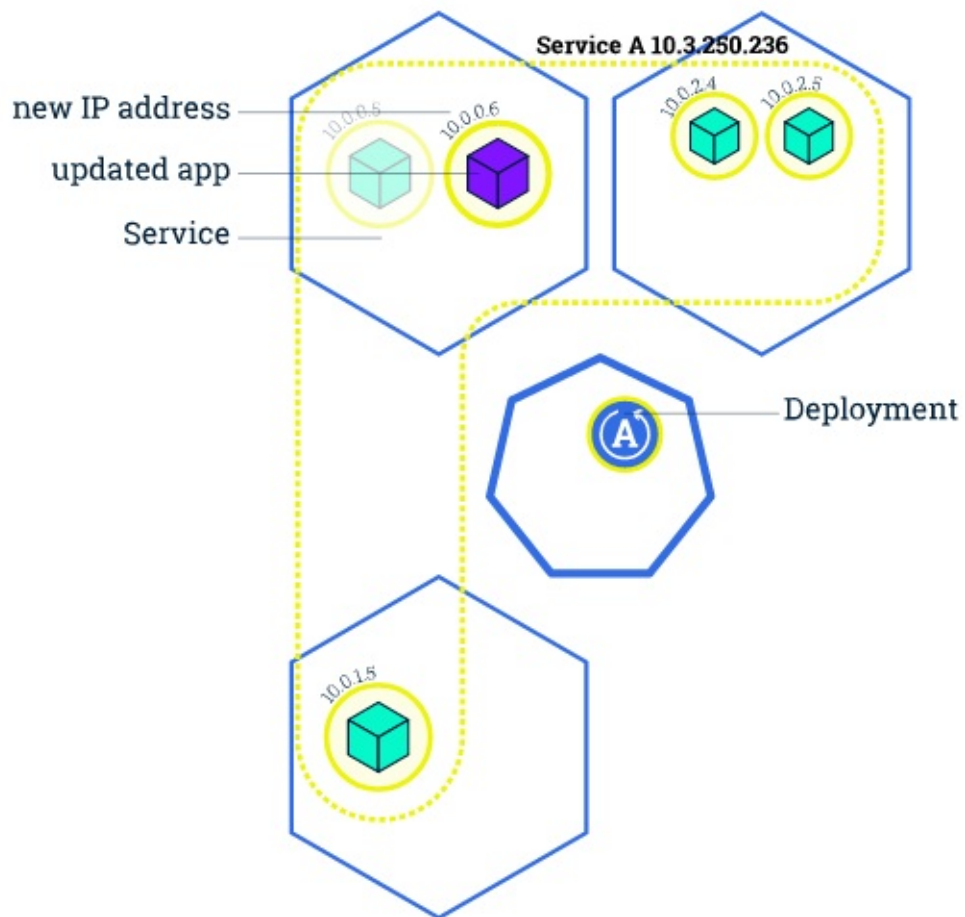
NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-app	3	3	3	3	10m

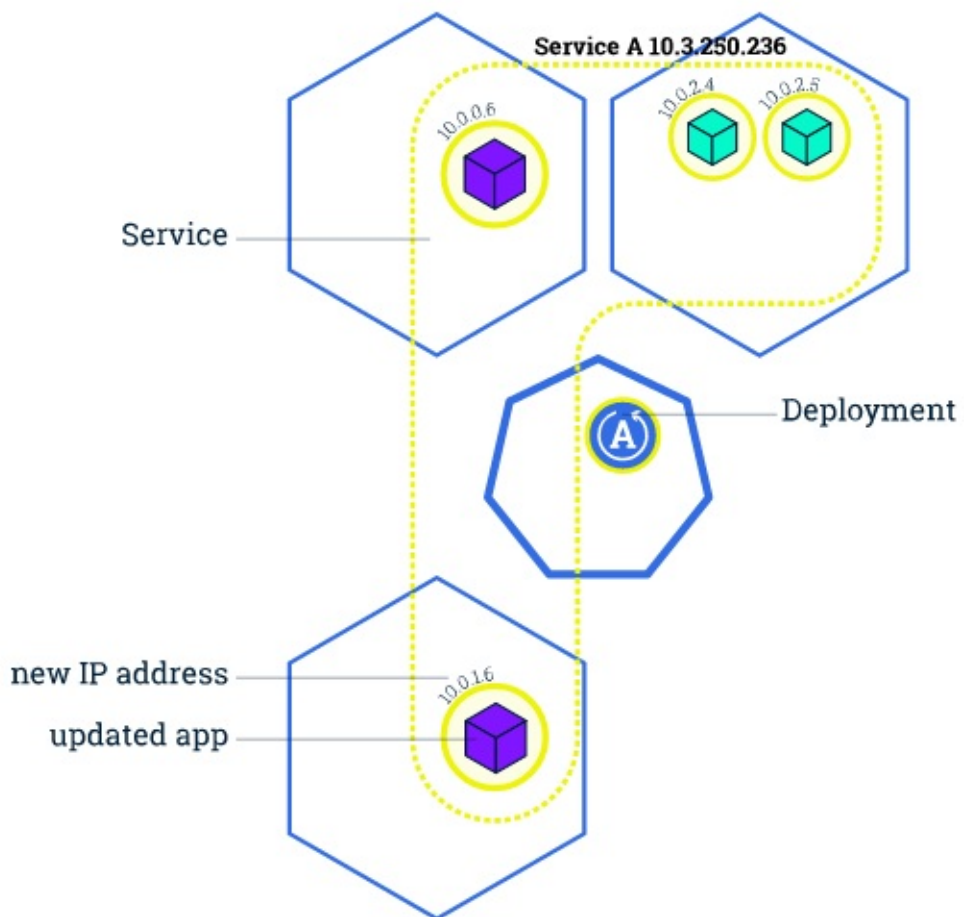
滚动升级

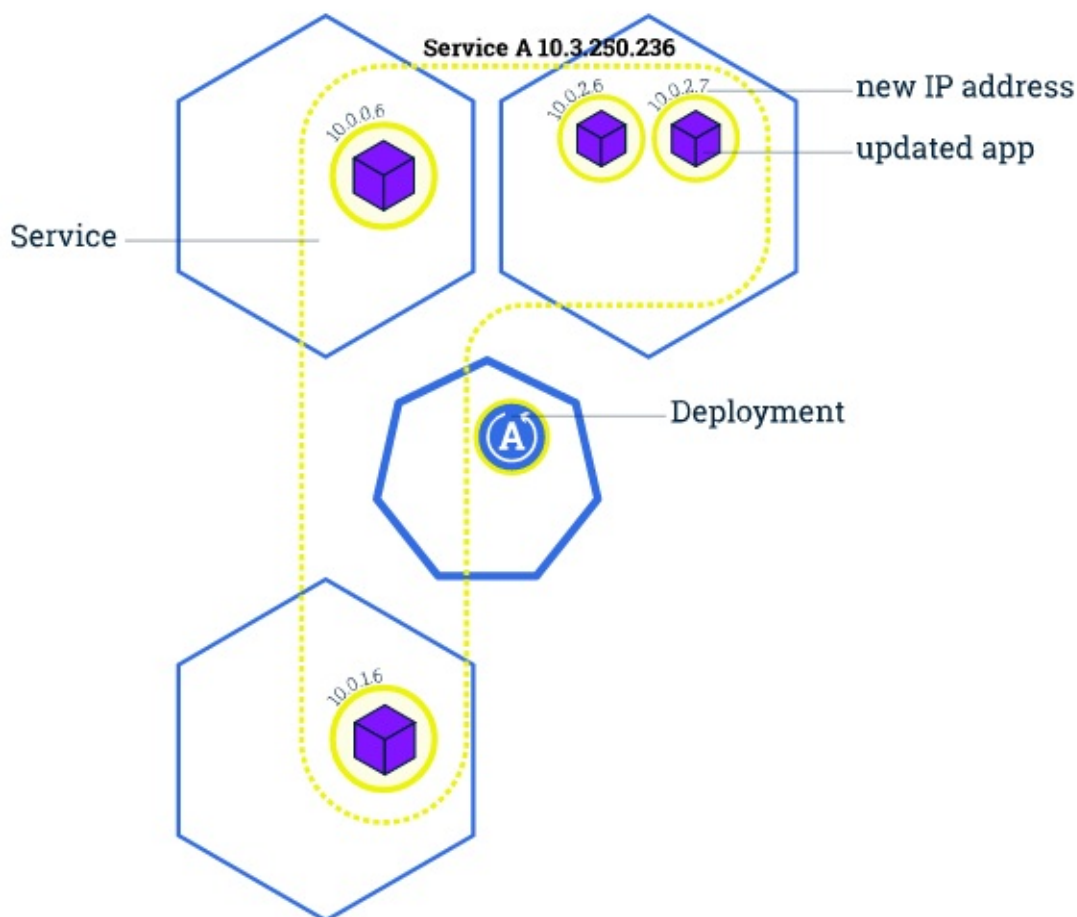
滚动升级（Rolling Update）通过逐个容器替代升级的方式来实现无中断的服务升级：

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









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，rolling-update只针对ReplicationController，不能用在策略不是RollingUpdate的Deployment上（Deployment可以在spec中设置更新策略为RollingUpdate，默认就是RollingUpdate）：

```
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 updated...
Waiting for rollout to finish: 2 of 3 updated replicas are available.
..
Waiting for rollout to finish: 2 of 3 updated replicas are available.
..
Waiting for rollout to finish: 2 of 3 updated replicas are available.
..
Waiting for rollout to finish: 2 of 3 updated replicas are available.
..
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           <none>
2           <none>

$ kubernetes 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=cpu=500m,memory=128Mi
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

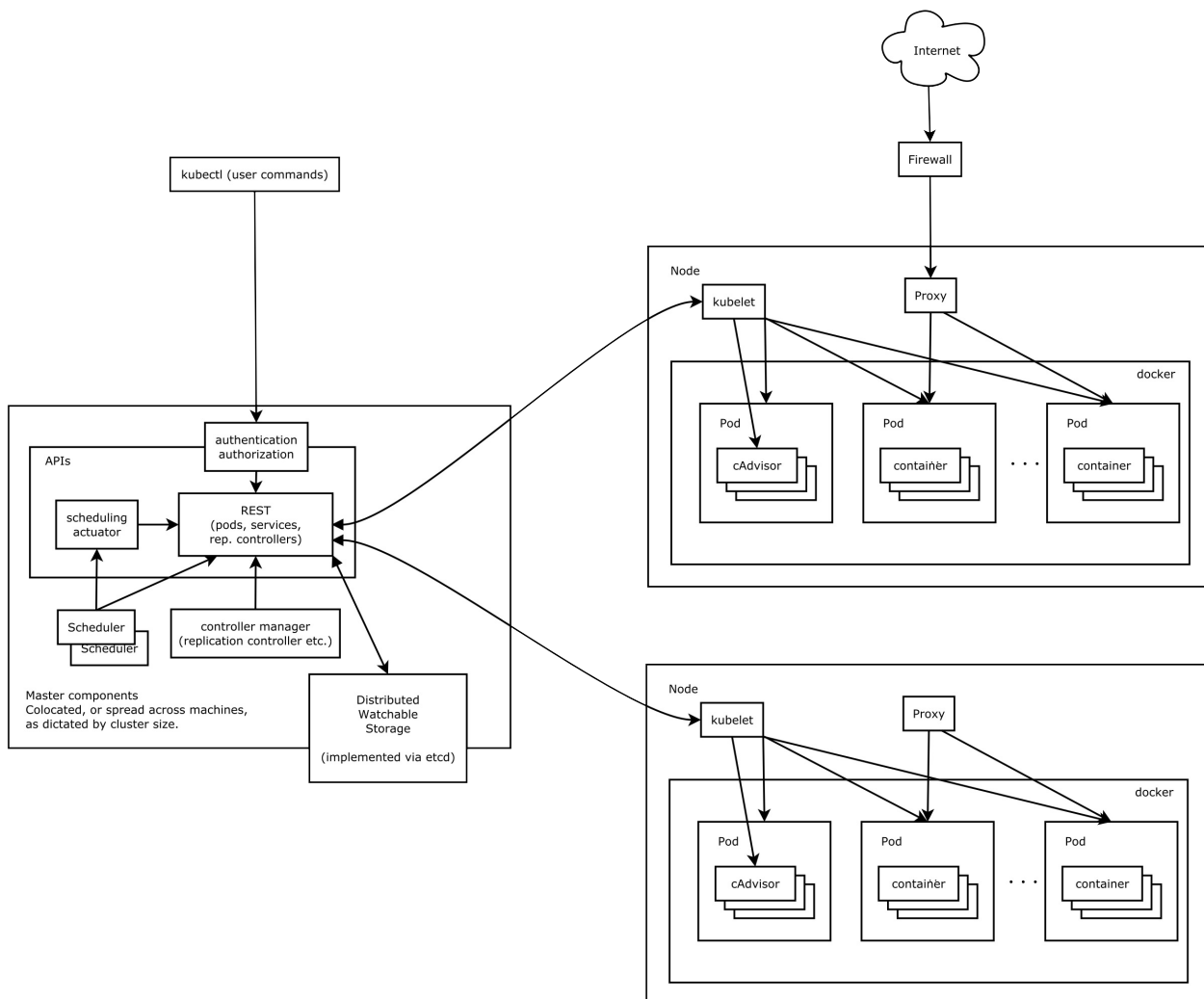
Kubernetes作为一个面向应用的集群管理工具，需要确保容器在部署后确实处在正常的运行状态。Kubernetes提供了两种探针（Probe，支持exec、tcp和http方式）来探测容器的状态：

- **LivenessProbe**：探测应用是否处于健康状态，如果不健康则删除重建改容器
- **ReadinessProbe**：探测应用是否启动完成并且处于正常服务状态，如果不正常则更新容器的状态

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: /ping
          port: 80
          initialDelaySeconds: 5
          timeoutSeconds: 1
```

Kubernetes 集群



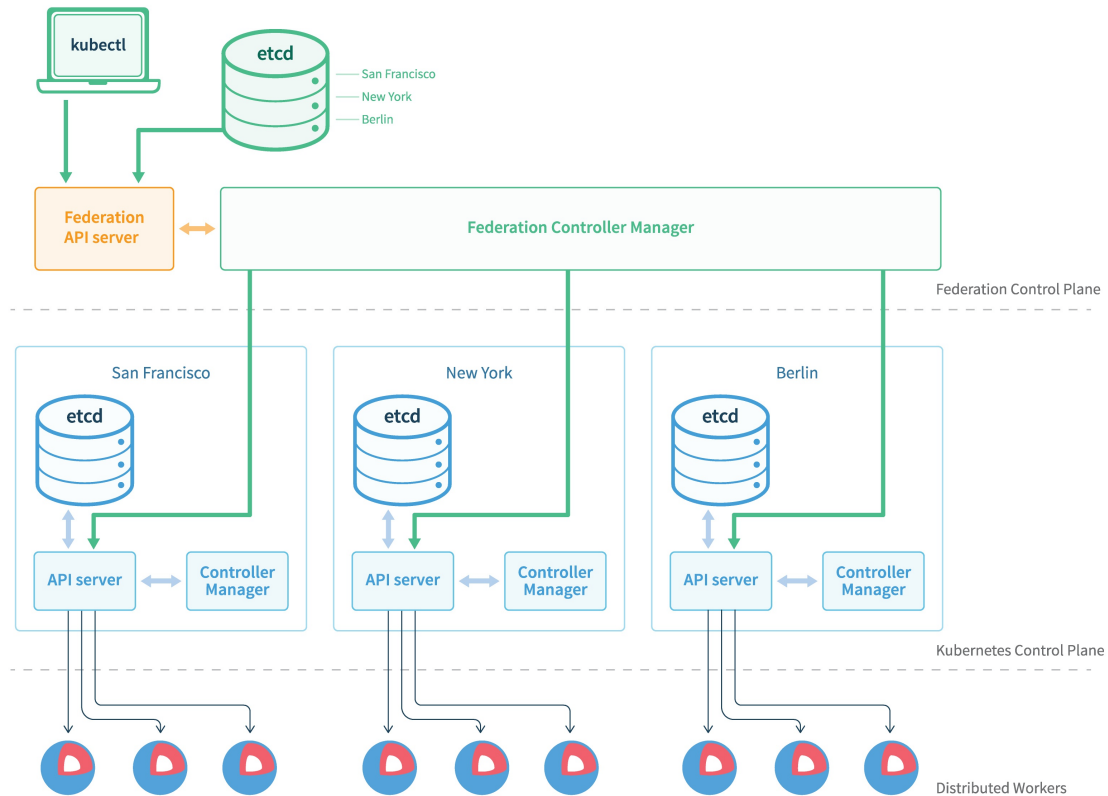
一个Kubernetes集群由分布式存储etcd、控制节点controller以及服务节点Node组成。

- 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/v1/
proxy/namespaces/kube-system/services/kubernetes-dashboard

To further debug and diagnose cluster problems, use 'kubectl cluster-
info dump'.
```

play-with-k8s

[Play with Kubernetes](http://play-with-k8s.com)提供了一个免费的Kubernets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，每个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，点解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Kuberentes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://git.io/kube-dashboard
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

介绍Kubernetes架构以及核心概念。

Kubernetes架构

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。

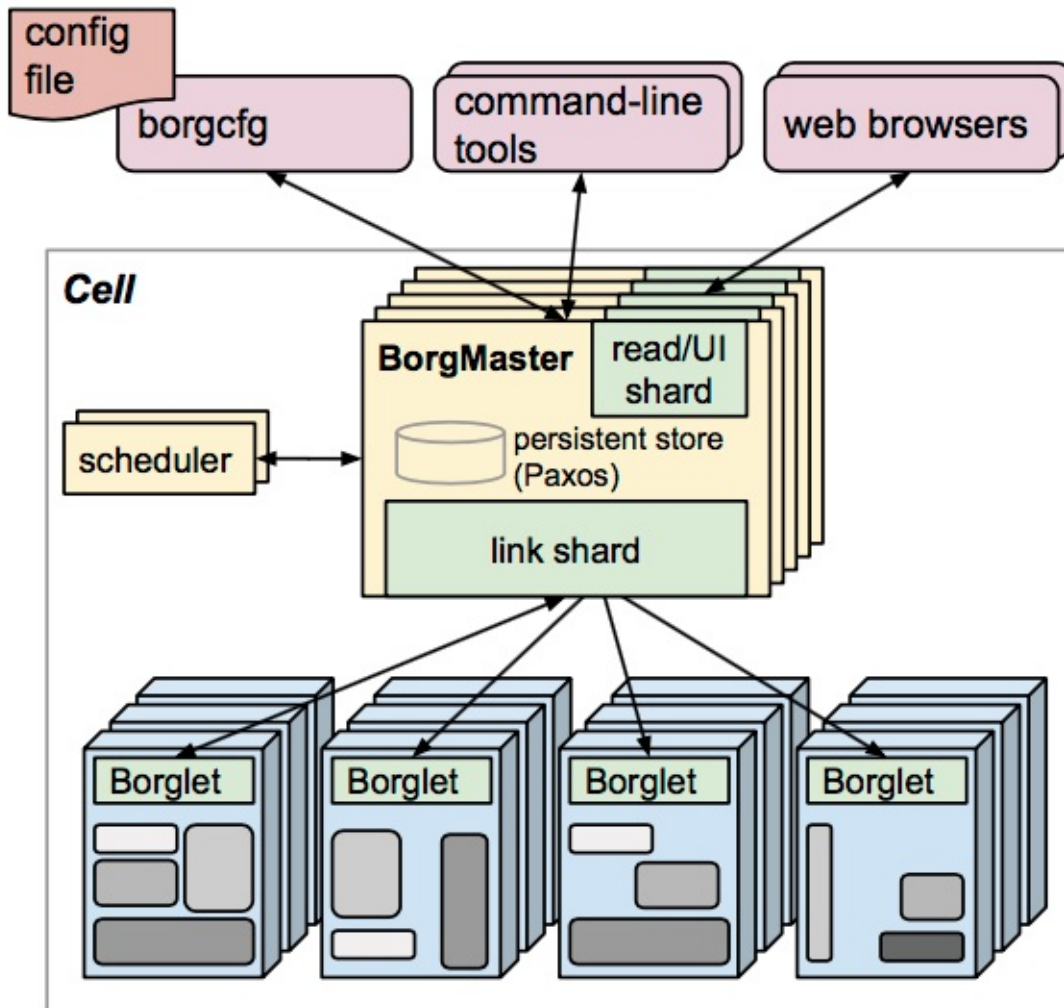
Kubernetes 还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

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。

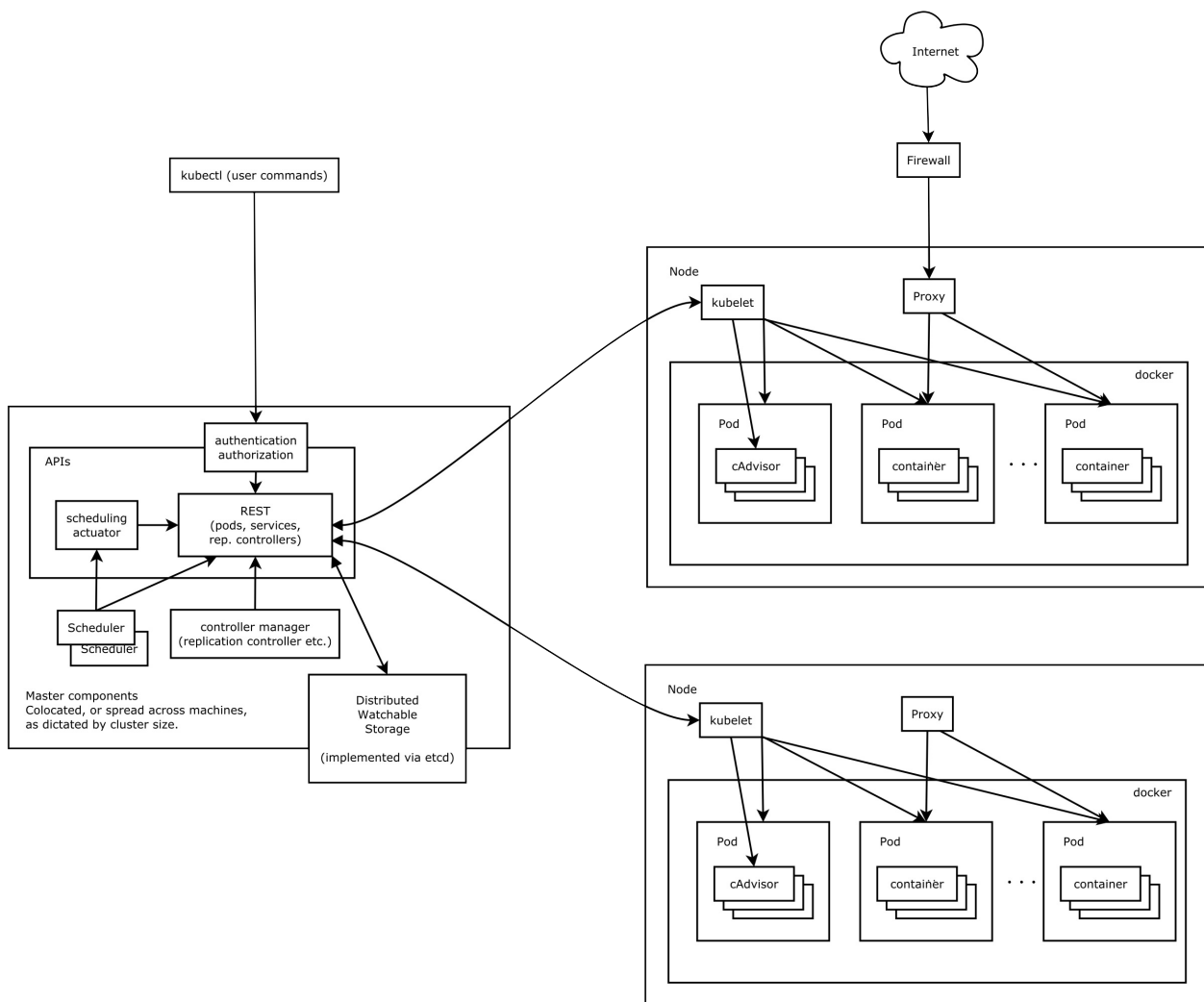
Borg主要由BorgMaster、Borglet、borgcfg和Scheduler组成，如下图所示



- 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；
- Scheduler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



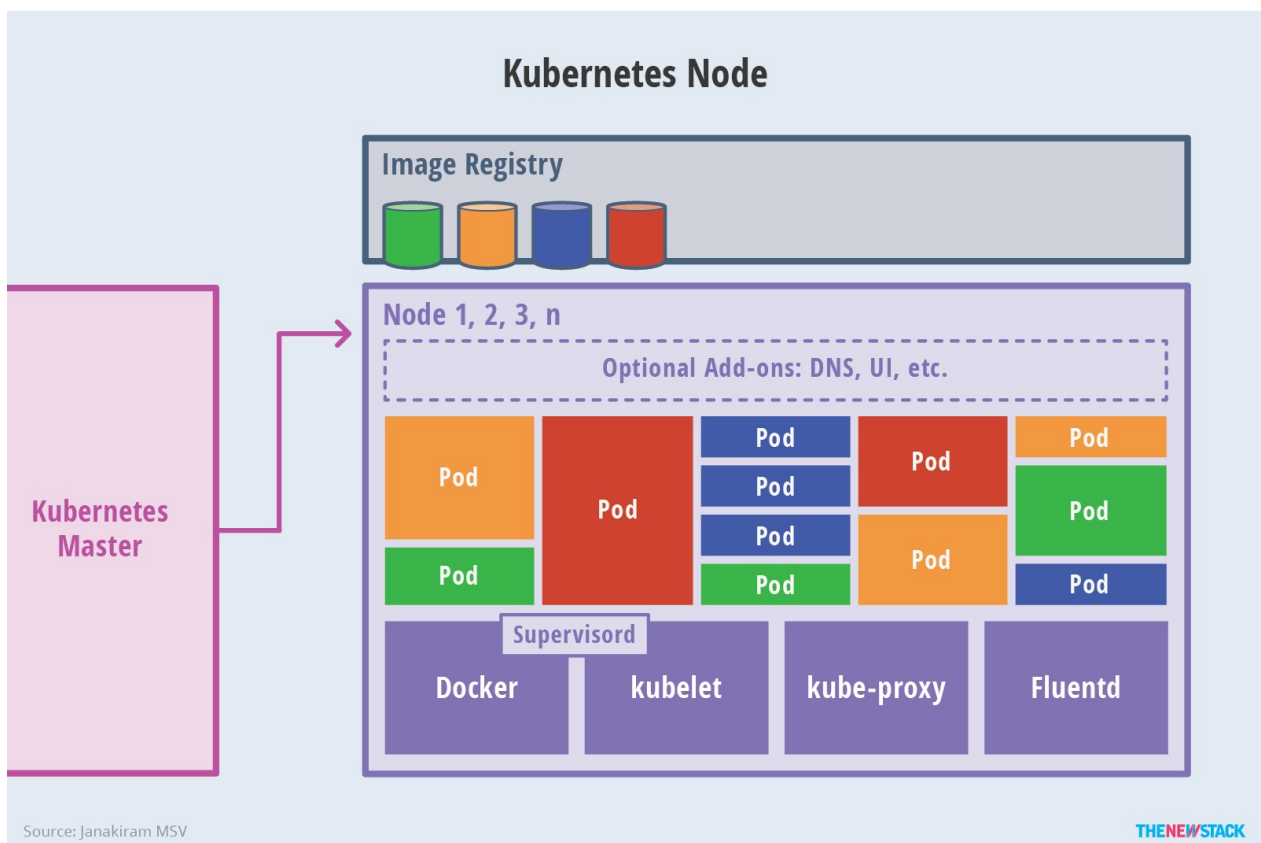
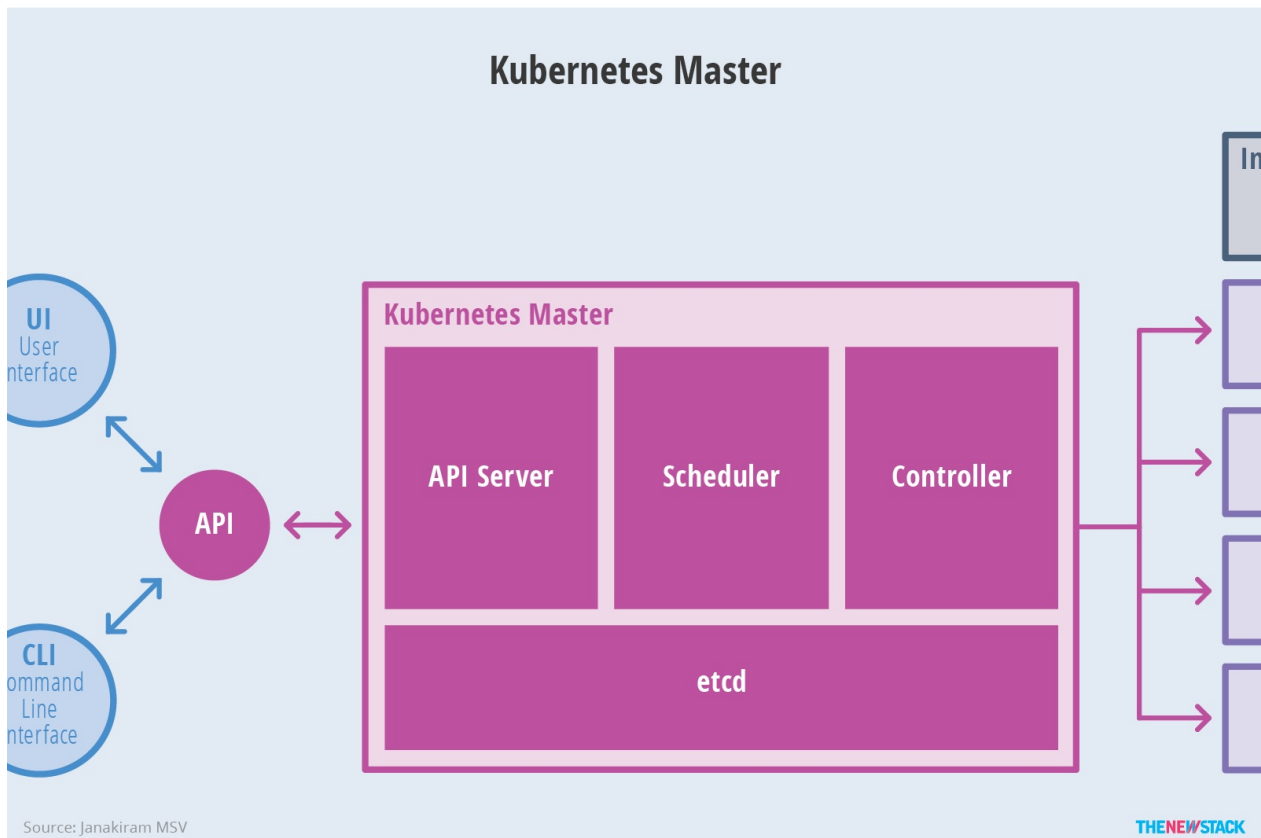
Kubernetes主要由以下几个核心组件组成：

- 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；

除了核心组件，还有一些推荐的Add-ons：

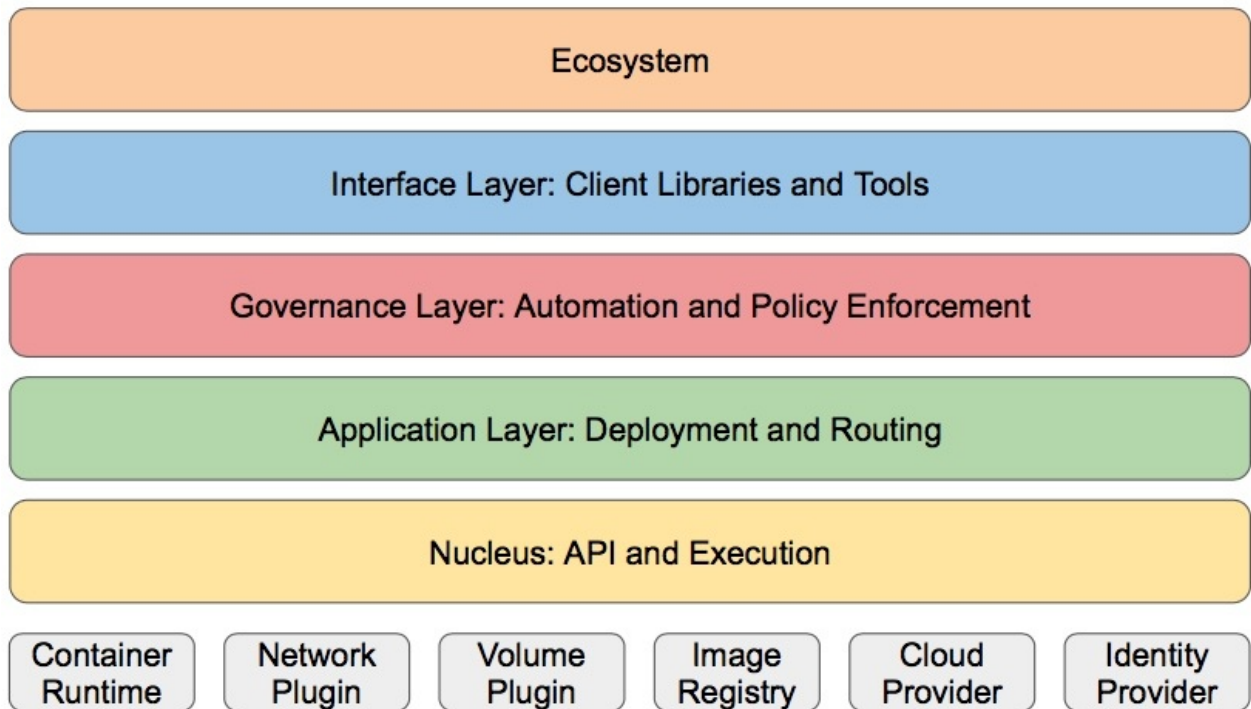
- kube-dns负责为整个集群提供DNS服务
- Ingress Controller为服务提供外网入口
- Heapster提供资源监控
- Dashboard提供GUI

- 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

关于分层架构，可以关注下Kubernetes社区正在推进的Kubernetes architectural roadmap (<https://docs.google.com/document/d/1XkjVm4b0eiVkj-Xt1LgoGiqWsBfNozJ51dyI-ljzt1o> ，需要加入kubernetes-dev google groups才可以查看)。

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 architectural roadmap](#)

Kubernetes的设计理念

Kubernetes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，正如本文前面所说，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，理解掌握的API，就好比抓住了K8s系统的牛鼻子。K8s系统API的设计有以下几条原则：

1. 所有API应该是声明式的。正如前文所说，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，隐藏实现的细节的同时，也就保留了系统未来持续优化的可能性。此外，声明式的API，同时隐含了所有的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分布式对象。
2. API对象是彼此互补而且可组合的。这里面实际是鼓励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。事实上，K8s这种分布式系统管理平台，也是一种业务系统，只不过它的业务就是调度和管理容器服务。
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K8s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PetSet和ReplicaSet，本来就是两种Pod集合，那么K8s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，以为不必担心

引入高级功能影响原有的基本功能。

Kubernetes的核心技术概念和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；那么复制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候型（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PetSet，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，只适用于长期伺候型的业务类型，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，未来对所有长期伺候型的业务的管理，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做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

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持K8s集群运行的服务。

有状态服务集（PetSet）

K8s在1.3版本里发布了Alpha版的PetSet功能。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PetSet是用来控制有状态服务，PetSet中的每个Pod的名字都是事先确定的，不能更改。PetSet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PetSet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PetSet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。PetSet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PetSet，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，PetSet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。PetSet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；也支持比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，有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，减少暴漏机会。

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与云原生应用](#)

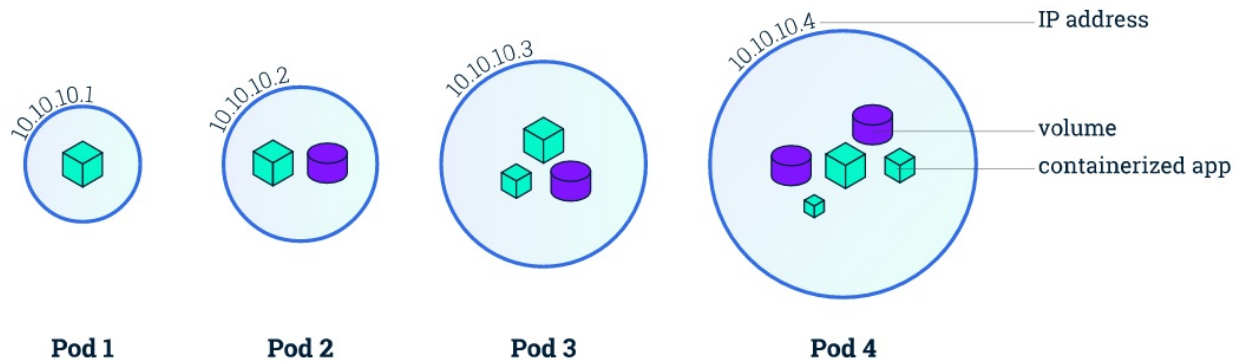
主要概念

Kubernetes主要概念和对象介绍。

- Pod, Service, Namespace和Node
- Service
- Volume和Persistent Volume
- Deployment
- Secret
- StatefulSet
- DaemonSet
- ServiceAccount
- ReplicationController和ReplicaSet
- Job
- CronJob
- SecurityContext
- Resource Quota
- Pod Security Policy
- Horizontal Pod Autoscaling
- Network Policy
- Ingress
- ThirdPartyResources

Pod

Pod是一组紧密关联的容器集合，它们共享IPC、Network和UTC 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内容容器都可以访问共享的Volume，可以访问共享数据
- Pod一旦调度后就跟Node绑定，即使Node挂掉也不会重新调度，推荐使用Deployments、Daemonsets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）

Pod定义

通过yaml或json描述Pod和其内Container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
```

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=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

容器中引用该secret：

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
    - name: private-reg-container
      image: <your-private-image>
  imagePullSecrets:
    - name: regsecret
```

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包括了容器运行前创建的所有服务的信息，比如默认的kubernetes服务对应了环境变量

```
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。

ImagePullPolicy

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 后缀，通过kube-dns查询 -Default：优先从 kubelet中配置的DNS查询

注意：

- 默认配置的dnsPolicy是ClusterFirst

使用主机的IPC命名空间

通过设置hostIPC参数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hostNetwork参数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hostPID参数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中的hostname

通过hostname参数实现，如果未设置默认使用PodName作为Pod的hostname

设置Pod的子域名

通过subdomain参数设置Pod的子域名，默认为空

- 指定hostname为busybox-2和subdomain为default-subdomain，完整域名为 `busybox-2.default-subdomain.default.svc.cluster.local` :


```
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
```

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）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，容器可能会被停止或调度到其他资源充足的机器上
- `spec.containers[].resources.requests.cpu`：CPU请求，可以超过
- `spec.containers[].resources.requests.memory`：内存请求，可以超过；但如果超过，容器可能会在Node内存不足时清理

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: "500m"
          memory: "128Mi"
```

注意，CPU的单位是milicpu，500mcpu=0.5cpu；而内存的单位则包括E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki等。

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，支持exec、tcp和httpGet方式）来探测容器的状态：

- **LivenessProbe**：探测应用是否处于健康状态，如果不健康则删除重建改容器
- **ReadinessProbe**：探测应用是否启动完成并且处于正常服务状态，如果不正常则更新容器的状态

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
    name: nginx
spec:
  containers:
    - image: nginx
      imagePullPolicy: Always
      name: http
      livenessProbe:
        httpGet:
          path: /
          port: 80
          initialDelaySeconds: 15
          timeoutSeconds: 1
      readinessProbe:
        httpGet:
          path: /ping
          port: 80
          initialDelaySeconds: 5
          timeoutSeconds: 1
```

Init Container

Init Container在所有容器运行之前执行（run-to-completion），常用来初始化配置。

```
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: {}
```

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**：容器启动后执行，注意由于是异步执行，它无法保证一定在ENTRYPOINT之后运行。如果失败，容器会被杀死，并根据RestartPolicy决定是否重启
- **preStop**：容器停止前执行，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
- `httpGet`：向指定URL发起GET请求

`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:
          exec:
            command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]
        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: hello-world
spec:
  containers:
  - name: friendly-container
    image: "alpine:3.4"
    command: ["/bin/echo", "hello", "world"]
    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
```

[warning] 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 dst
  10.1.0.3/32 flowid 1:2
# 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 src
  10.1.0.3/32 flowid 1:3
```

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调度到制定node节点上。

比如，使用nodeSelector，首先给Node加上标签：

```
kubectl label nodes <your-node-name> 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
```

参考文档

- [What is Pod?](#)

- [Kubernetes Pod Lifecycle](#)
- [Container capabilities](#)
- [Linux Capabilities](#)

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。

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-namespaces=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：

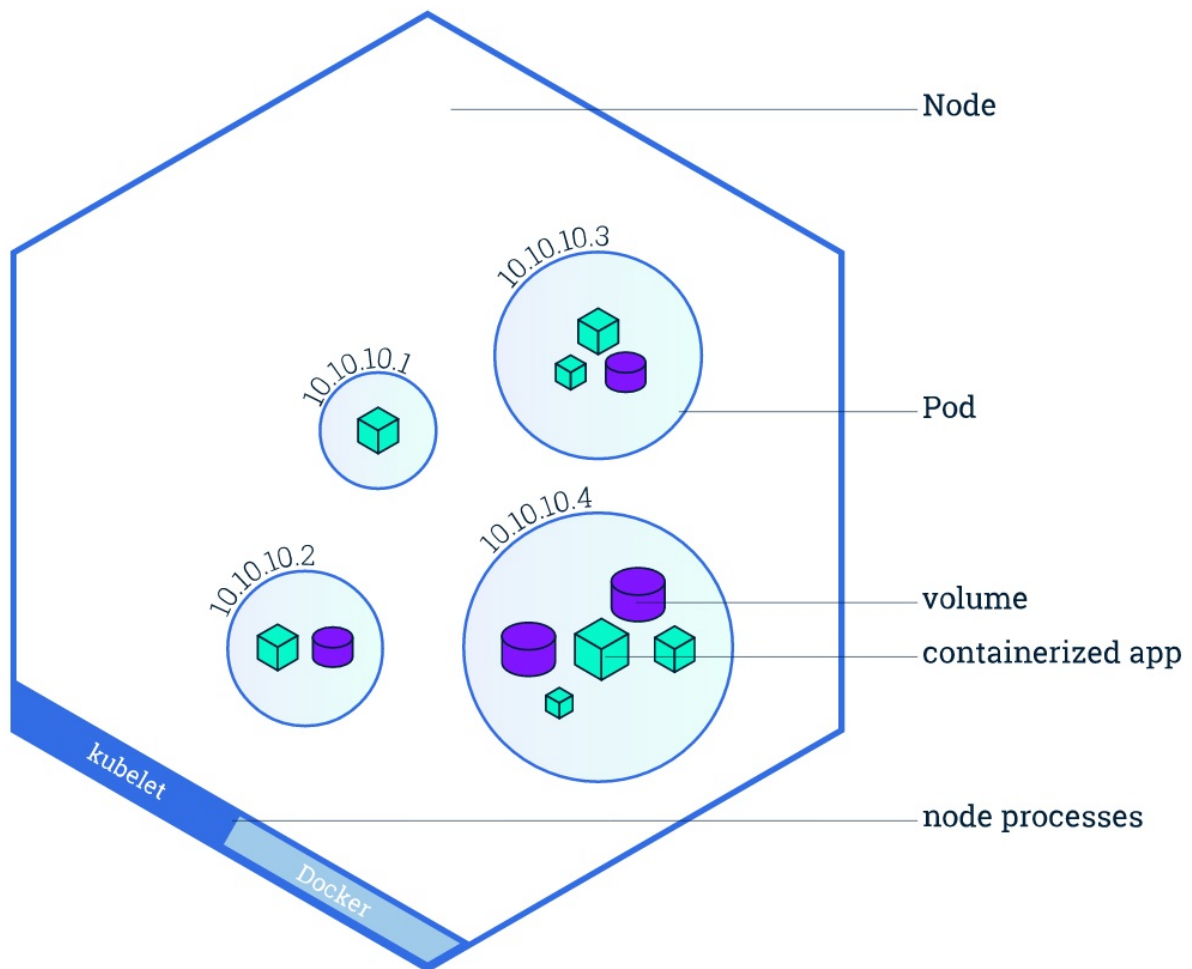
1. 删除一个namespace会自动删除所有属于该namespace的资源。
2. `default` 和 `kube-system` 命名空间不可删除。
3. `PersistentVolumes`是不属于任何namespace的，但`PersistentVolumeClaim`是属于某个特定namespace的。
4. `Events`是否属于namespace取决于产生events的对象。

参考文档

- [Kubernetes Namespace](#)
- [Share a Cluster with Namespaces](#)

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json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

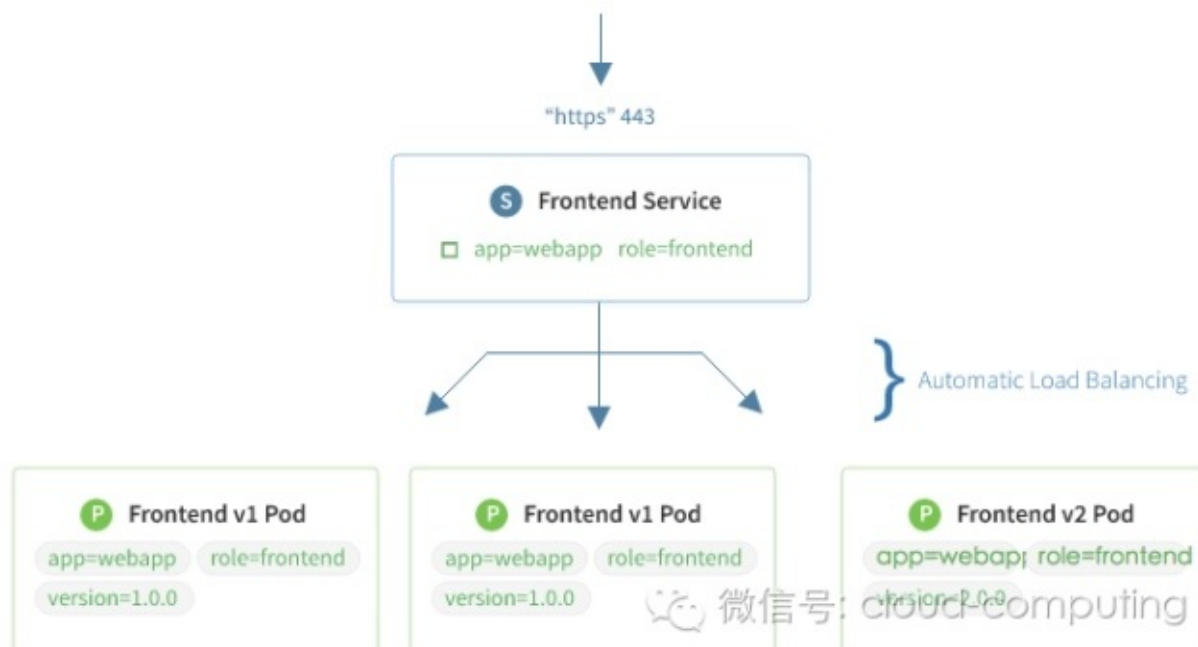
- [Kubernetes Node](#)
- [Taints和tolerations](#)

服务发现与负载均衡

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LB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 `<NodeIP>:NodePort` 来访问该服务
- 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 `<NodeIP>:NodePort`
- ExternalName：将服务通过DNS CNAME记录方式转发到指定的域名（通过 `spec.externalName` 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    <none>         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:    <none>
Selector:      run=nginx
Type:          ClusterIP
IP:            10.0.0.108
Port:          <unset>      80/TCP
Endpoints:     172.17.0.5:80
Session Affinity:  None
Events:        <none>
```

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 `<service-name>.<namespace>.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（这种服务也称为Headless Service）。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: default
spec:
  type: ExternalName
  externalName: my.database.example.com
```

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(S)
default	nginx	None	<none>	80/TCP
kube-system	kube-dns	172.26.255.70	<none>	53/UDP, 53/TCP

```
$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE	I
nginx-2204978904-6o5dg	1/1	Running	0	14s	1
nginx-2204978904-qyilx	1/1	Running	0	14s	1

```
$ 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**：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加上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**：源IP会做SNAT，server pod看到的源IP是Node IP。在GKE/GCE中，添加annotation `service.beta.kubernetes.io/external-traffic=OnlyLocal` 后可以自动从负载均衡器中删除没有本地endpoint的Node。

Ingress Controller

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。

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

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>

Kubernetes 存储卷

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。

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: <volume-id>
    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/<vendor~driver>/<driver>`，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
```

其他的**Volume**参考示例

- [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](#)
- [AzureFileVolume 示例](#)
- [AzureDiskVolume 示例](#)
- [Quobyte Volume 示例](#)
- [PortworxVolume Volume 示例](#)
- [ScaleIO Volume 示例](#)
- [StorageOS Volume 示例](#)

Persistent Volume

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claim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
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

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

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Pod挂载。
- **ReadOnlyMany（ROX）**：可以以只读的方式被多个Pod挂载。
- **ReadWriteMany（RWX）**：这种存储可以以读写的方式被多个Pod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`）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。

在使用PVC时，可以通过 `DefaultStorageClass` 准入控制给未指定 `storageClassName`的PVC自动添加默认的StorageClass。默认的StorageClass带有annotation `storageclass.kubernetes.io/is-default-class=true`。

修改默认StorageClass

取消原来的默认StorageClass

```
kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

标记新的默认StorageClass

```
kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

GCE 示例

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zone: us-central1-a
```

Ceph RBD 示例

```
apiVersion: storage.k8s.io/v1beta1
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
```

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
```

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的源，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
```

本地数据卷

注意：仅在v1.7+中支持，目前为alpha版。

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

创建一个调度到hostname为 `example-node` 的本地数据卷：

```
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": ["my-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
```

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。典型的应用场景包括：

- 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

比如一个简单的nginx应用可以定义为

```
apiVersion: extensions/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
```

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 --cpu-percent=80
```

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概念详细解析

本文翻译自kubernetes官方文档：

档：<https://github.com/kubernetes/kubernetes.github.io/blob/master/docs/concepts/workloads/controllers/deployment.md>

根据2017年5月10日的Commit 8481c02 翻译。

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 --record
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   AGE
nginx-deployment    3         0         0            0           1s
```

输出结果表明我们希望的replica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   AGE
nginx-deployment    3         3         3            3           18s
```

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 `<Deployment的名字>-<pod template的hash值>`。

```
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-2035384211-7ci7o	1/1	Running	0	18s
Labels: app=nginx, pod-template-hash=2035384211				
nginx-deployment-2035384211-kzszj	1/1	Running	0	18s
Labels: app=nginx, pod-template-hash=2035384211				
nginx-deployment-2035384211-qgcnn	1/1	Running	0	18s
Labels: app=nginx, pod-template-hash=2035384211				

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了，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 updated...
deployment "nginx-deployment" successfully rolled out
```

Rollout成功后，`get Deployment`：

```
$ kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	36s

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	AGE
nginx-deployment-1564180365-khku8	1/1	Running	0	14s
nginx-deployment-1564180365-nacti	1/1	Running	0	14s
nginx-deployment-1564180365-z9gth	1/1	Running	0	14s

下次更新这些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Kubernetes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: <none>
NewReplicaSet:   nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen Count From SubobjectPath
  Type Reason Message
  -----
  36s 36s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled up replica set nginx-deplo
  yment-2035384211 to 3
  23s 23s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled up replica set nginx-deplo
  yment-1564180365 to 1
  23s 23s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled down replica set nginx-dep
  loyment-2035384211 to 2
  23s 23s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled up replica set nginx-deplo
  yment-1564180365 to 2
  21s 21s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled down replica set nginx-dep
  loyment-2035384211 to 0
  21s 21s 1 {deployment-controller }
  Normal ScalingReplicaSet Scaled up replica set nginx-deplo
  yment-1564180365 to 3
```

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 `nginx: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会在系统中保存前两次的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是，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 updated...
```

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	REST
ARTS AGE			
nginx-deployment-1564180365-70iae	1/1	Running	0
25s			
nginx-deployment-1564180365-jbqqo	1/1	Running	0
25s			
nginx-deployment-3066724191-08mng	0/1	ImagePullBackOff	0
6s			
nginx-deployment-3066724191-eocby	0/1	ImagePullBackOff	0
6s			

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 created)
 NewReplicaSet: nginx-deployment-3066724191 (2/2 replicas created)
 Events:

FirstSeen	LastSeen	Count	From	SubobjectPath
Type	Reason		Message	
-----	-----	-----	----	-----
-----	-----	-----	-----	
1m	1m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-2035384211 to 3	
22s	22s	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-1564180365 to 1	
22s	22s	1	{deployment-controller }	

```

Normal          ScalingReplicaSet   Scaled down replica set nginx-depl
oyment-2035384211 to 2
22s            22s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled up replica set nginx-deploy
ment-1564180365 to 2
21s            21s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled down replica set nginx-depl
oyment-2035384211 to 0
21s            21s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled up replica set nginx-deploy
ment-1564180365 to 3
13s            13s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled up replica set nginx-deploy
ment-3066724191 to 1
13s            13s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled down replica set nginx-depl
oyment-1564180365 to 2
13s            13s            1            {deployment-controller }
Normal          ScalingReplicaSet   Scaled up replica set nginx-deploy
ment-3066724191 to 2

```

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.yaml -
-record
2           kubectl set image deployment/nginx-deployment nginx=nginx
:1.9.1
3           kubectl set image deployment/nginx-deployment nginx=nginx
:1.9.1

```

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

查看单个revision的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels:      app=nginx
              pod-template-hash=1159050644
  Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
  Containers:
    nginx:
      Image:      nginx:1.9.1
      Port:      80/TCP
      QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
      Environment Variables:      <none>
  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=2
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revision 2的 `DeploymentRollback` 的event。

```
$ kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	30m

```
$ 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: <none>
NewReplicaSet:    nginx-deployment-1564180365 (3/3 replicas created)
Events:
```

FirstSeen	LastSeen	Count	From	SubobjectPath
Type	Reason		Message	
30m	30m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-2035384211 to 3	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-1564180365 to 1	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled down replica set nginx-deployment-2035384211 to 2	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-1564180365 to 2	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled down replica set nginx-deployment-2035384211 to 0	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-3066724191 to 2	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled up replica set nginx-deployment-3066724191 to 1	
29m	29m	1	{deployment-controller }	
Normal	ScalingReplicaSet		Scaled down replica set nginx-deployment-1564180365 to 2	

```

2m          2m          1          {deployment-controller }
Normal      ScalingReplicaSet    Scaled down replica set nginx-depl
oyment-3066724191 to 0
2m          2m          1          {deployment-controller }
Normal      DeploymentRollback   Rolled back deployment "nginx-depl
oyment" to revision 2
29m         2m          2          {deployment-controller }
Normal      ScalingReplicaSet    Scaled up replica set nginx-deploy
ment-1564180365 to 3

```

清理Policy

你可以通过设置 `.spec.revisionHistoryLimit` 项来指定deployment最多保留多少revision历史记录。默认会保留所有的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](#)，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 --c
pu-percent=80
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	AGE
nginx-deployment	10	10	10	10	50s

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ReplicaSet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	AGE
nginx-deployment	15	18	7	8	7m

```
$ 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Deployment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         <none>

$ 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=200m,memory=512Mi
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个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 available.
..
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。这可能是由于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":{"progressDeadlineSeconds":600}}'
"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kubernetes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`，Deployment 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 progressing
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-" is
    forbidden: exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2

```

最终，一旦超过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

你可以通过扩容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 updated...
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/扩容，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，将导致所有的Deployment历史记录都会被清除，该Deployment就无法再回退了。

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[kubectl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不会阻止你这么去做。

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ReplicaSet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ReplicaSet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，意味着回退到历史中最老的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revision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` 是可以可选配置项，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。

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：

- Service Account：用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的 `/run/secrets/kubernetes.io/serviceaccount` 目录中；
- Opaque：base64编码格式的Secret，用来存储密码、密钥等；
- `kubernetes.io/dockerconfigjson`：用来存储私有docker registry的认证信息。

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`。

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
```

Secret引用

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
```

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
```

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-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
secret "myregistrykey" created.
```

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
```

Service Account

Service Account用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到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.io/s
erviceaccount
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: c2VjcmV0IGlzIHNNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
```

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，

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/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  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
        annotations:
          volume.alpha.kubernetes.io/storage-class: anything
      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	<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kubernetes.io/gce-pd类型的volume）

```
$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPA
www-web-0	Bound	pvc-d064a004-d8d4-11e6-b521-42010a800002	1Gi
	RWO	16s	
www-web-1	Bound	pvc-d06a3946-d8d4-11e6-b521-42010a800002	1Gi
	RWO	16s	

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 --rm /bin/sh
```

```
/ # 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",
"path": "/spec/template/spec/containers/0/image", "value":"gcr.io/google_containers/nginx-slim:0.7"}]'
```

```
# 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":{"type":"RollingUpdate","rollingUpdate":{"partition":3}}}}'
statefulset "web" patched

# 更新StatefulSet
$ kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/image", "value":"gcr.io/google_containers/nginx-slim:0.7"}]'
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:
        - "zk0k.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. 还在beta状态，需要kubernetes v1.5版本以上才支持
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

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

使用Fluentd收集日志的例子：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  template:
    metadata:
      labels:
        app: logging
        id: fluentd
        name: fluentd
    spec:
      containers:
        - name: fluentd-es
          image: gcr.io/google_containers/fluentd-elasticsearch:1.3
          env:
            - name: FLUENTD_ARGS
              value: -qq
          volumeMounts:
            - name: containers
              mountPath: /var/lib/docker/containers
            - name: varlog
              mountPath: /varlog
      volumes:
        - hostPath:
            path: /var/lib/docker/containers
            name: containers
        - hostPath:
            path: /var/log
            name: varlog
```

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 <daemonset-name>

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

# 回滚
$ kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>

# 查询回滚状态
$ kubectl rollout status ds/<daemonset-name>
```

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。

Service Account

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.serviceAccount` 为default（除非指定了其他ServiceAccount）
 - 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 /run/secrets/kubernetes.io/serviceaccount
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-ee7d89
82865f
  creationTimestamp: 2017-05-27T14:32:25Z
  name: jenkins-token-l9v7v
  namespace: default
  resourceVersion: "45558"
  selfLink: /api/v1/namespaces/default/secrets/jenkins-token-l9v7v
  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 Group.
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
    nonResourceURLs: []
---
# This role binding allows "default" to read pods in the namespace "default"
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: ServiceAccount # May be "User", "Group" or "ServiceAccount"
    name: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

ReplicationController和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](#)。

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, th
```

```
en to
    # instead access environment variables to find service host
    # info, comment out the 'value: dns' line above, and uncomm
ent the
    # line below.
    # value: env
ports:
- containerPort: 80
```

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。

Kubernetes支持以下几种Job：

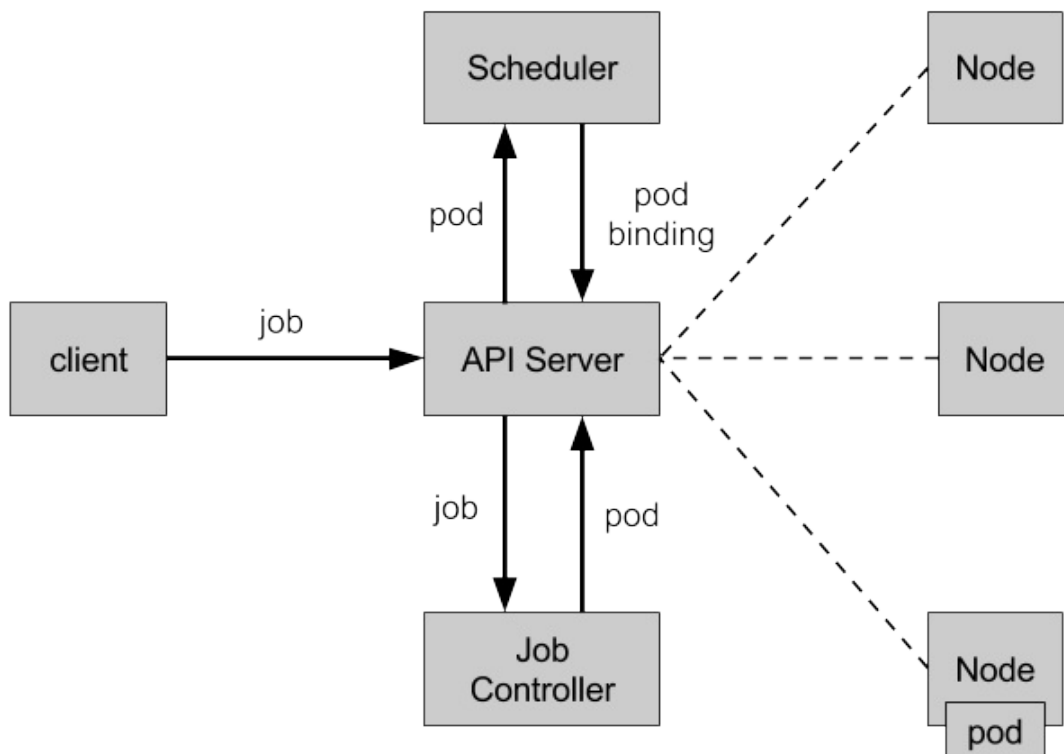
- 非并行Job：通常创建一个Pod直至其成功结束
- 固定结束次数的Job：设置 `.spec.completions`，创建多个Pod，直到 `.spec.completions` 个Pod成功结束
- 带有工作队列的并行Job：设置 `.spec.Parallelism` 但不设置 `.spec.completions`，当所有Pod结束并且至少一个成功时，Job就认为是成功

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(2000)"]
      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-76933ddd7f55
                job-name=pi
Annotations:    <none>
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:      <none>
    Mounts:           <none>
    Volumes:          <none>
Events:
  FirstSeen    LastSeen    Count   From            SubObjectPath  Type
  Reason                Message
  ----
  2m           2m          1    job-controller  Normal        Su
ccessfulCreate    Created pod: pi-nltxv

# 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=jsonpath={.
items..metadata.name})
$ kubectl logs $pods
3.141592653589793238462643383279502...

```

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ReplicationController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](#)

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。在Kubernetes 1.5，使用CronJob需要开启 `batch/v2alpha1` API，即 `--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/v2alpha1
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 --image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
```

```
$ kubectl get cronjob
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST-SCHEDULE
hello	*/1 * * * *	False	0	<none>

```
$ kubectl get jobs
```

NAME	DESIRED	SUCCESSFUL	AGE
hello-1202039034	1	1	49s

```
$ pods=$(kubectl get pods --selector=job-name=hello-1202039034 --output=jsonpath={.items..metadata.name} -a)
```

```
$ 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 delete job`来删除

```
$ kubectl delete cronjob hello
```

```
cronjob "hello" deleted
```

Security Context

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
volumes	控制容器可以使用哪些volume
hostNetwork	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

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:
    - '*'
```

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/volumes/kubernetes.io~empty-dir/test-volume:/mounted_volume:Z
/var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/volumes/kubernetes.io~secret/default-token-88xxa:/var/run/secrets/kubernetes.io/serviceaccount:ro,Z
/var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/etc-hosts:/etc/hosts
```

对应的volume也都会正确设置SELinux：

```
$ ls -Z /var/lib/kubelet/pods/f734678c-95de-11e6-89b0-42010a8c0002/volumes
drwxr-xr-x. root root unconfined_u:object_r:svirt_sandbox_file_t:s0:c2,c3 kubernetes.io~empty-dir
drwxr-xr-x. root root unconfined_u:object_r:svirt_sandbox_file_t:s0:c2,c3 kubernetes.io~secret
```

Resource Quotas

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

资源配额的启用

首先，在API Server启动时配置ResourceQuota adminssion control；然后在namespace中创建 ResourceQuota 对象即可。

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
- 对象数，即可创建的对象个数
 - 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-container/limits.yaml --namespace=limit-example
limitrange "mylimits" created
$ kubectl describe limits mylimits --namespace=limit-example
Name:      mylimits
Namespace: limit-example
Type      Resource      Min      Max      Default Request      Defa
ult Limit      Max Limit/Request Ratio
----
-----
Pod        cpu            200m     2        -                  -
Pod        memory        6Mi      1Gi      -                  -
Container  cpu            100m     2        200m              300m
Container  memory        3Mi      1Gi      100Mi             200Mi
```

配额范围

每个配额在创建时可以指定一系列的

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

Horizontal Pod Autoscaling

Horizontal Pod Autoscaling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

示例

```
# 创建pod和service
$ kubectl run php-apache --image=gcr.io/google_containers/hpa-example
--requests=cpu=200m --expose --port=80
service "php-apache" created
deployment "php-apache" created

# 创建autoscaler
$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --
max=10
deployment "php-apache" autoscaled
$ kubectl get hpa
```

NAME	REFERENCE	TARGET	CURRENT	MINPOD
S	MAXPODS	AGE		
php-apache	Deployment/php-apache/scale	50%	0%	1
10	18s			

```
# 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.cluster.lo
cal; done
```

过一会就可以看到负载升高了

\$ kubectl get hpa

NAME	REFERENCE	TARGET	CURRENT	MINPOD
S	MAXPODS	AGE		
php-apache	Deployment/php-apache/scale	50%	305%	1
10	3m			

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	CURRENT	MINPOD
S	MAXPODS	AGE		
php-apache	Deployment/php-apache/scale	50%	0%	1
10	11m			

\$ 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`
- 控制管理器的 `--apiserver` 指向[API Server Aggregator](#)
- 在API Server Aggregator中注册自定义的metrics API

注：可以参考k8s.io/metrics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/v2alpha1
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: <some-time>
  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（AbleToScale）以及是否受到限制（ScalingLimited）。

```
$ kubectl describe hpa cm-test
Name:                                cm-test
Namespace:                           prom
Labels:                               <none>
Annotations:                          <none>
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 scale
time was sufficiently old as to warrant a new scale
  ScalingActive                       True    ValidMetricFound                       the HPA was a
ble to successfully calculate a replica count from pods metric http_r
equests
  ScalingLimited                      False   DesiredWithinRange                     the desired r
eplica count is within the acceptable range
Events:
```

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之前，需要注意

- v1.6以及以前的版本需要在apiserver开启 `extensions/v1beta1/networkpolicies`
- v1.7+版本Network Policy已经GA，API版本为 `networking.k8s.io/v1`
- 网络插件要支持Network Policy，如Calico、Romana、Weave Net和trireme等

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之间通信

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
```

而默认允许所有Pod通信的策略为


```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
  ingress:
  - {}
```

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 <namespace> "net.beta.kubernetes.io/network-policy={\"ingress\": {\"isolation\": \"DefaultDeny\"}}"
```

注：目前，Network Policy仅支持Ingress流量控制。

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
```

示例

以calico为例看一下Network Policy的具体用法（使用kubernetes v1.6版本）。

首先配置kubelet使用CNI网络插件

```
kubelet --network-plugin=cni --cni-conf-dir=/etc/cni/net.d --cni-bin-dir=/opt/cni/bin ...
```

安装calio网络插件

```
# 注意修改CIDR，需要跟k8s pod-network-cidr一致，默认为192.168.0.0/16
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml
```

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 run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status
is Pending, pod ready: false

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：

```
# annotate仅适用于v1.6及以前版本，v1.7+需要创建默认拒绝策略
$ kubectl annotate ns default "net.beta.kubernetes.io/network-policy={\"ingress\": {\"isolation\": \"DefaultDeny\"}}\"

$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status
is Pending, pod ready: false

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
```

```
# v1.7中版本号为networking.k8s.io/v1
apiVersion: extensions/v1beta1
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, status
is Pending, pod ready: false
```

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=busybox
/bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status
is Pending, pod ready: false
```

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
# v1.7中版本号为networking.k8s.io/v1
apiVersion: extensions/v1beta1
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
```

Ingress

术语

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网络模型实现群集内的通信的具体实现，比如flannel 和 OVS。
- 服务：使用标签选择器标识一组pod成为的Kubernetes 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将其转发给相关的Pod。

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。

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。

路由到多服务的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

- [traefik ingress](#)提供了一个traefik ingress的实践案例
- [kubernetes/ingress](#)提供了更多的Ingress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 Træfik and Let's Encrypt](#)
- [Kubernetes : Træfik and Let's Encrypt at scale](#)
- [Kubernetes Ingress Controller-Træfik](#)
- [Kubernetes 1.2 and simplifying advanced networking with Ingress](#)

ConfigMap

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。

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.how=very
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.env
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。

[warning] 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.how=very --from-literal=special.type=charm
$ kubectl create configmap env-config --from-literal=log_level=INFO
```

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) $(SPECIAL_TYPE_KEY)" ]
      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.level"
]
      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
```

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。

开启 PodPreset

- 开启API `settings.k8s.io/v1alpha1/podpreset`
- 开启准入控制 `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 version"
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 version"
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
```

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 `<kind name>.<domain>` 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

[warning] ThirdPartyResources在v1.7弃用

ThirdPartyResources已在v1.7弃用，并将在v1.8版本中删除。建议从v1.7开始，迁移到[CustomResourceDefinition \(CRD\)](#)。

下面的例子会创建一

个 `/apis/stable.example.com/v1/namespaces/<namespace>/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 schedule"
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-cluster-
role --user=user1 --user=user2 --group=group1
```

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 schedul
e"
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},ESTABLISHED:{.status.conditions[?(@.type=="Established")].status}'
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 yaml
--export > tpr.yaml

```

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数据是否以前成功，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）

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，TPR将在v1.8中删除。

CRD 示例

下面的例子会创建一

个 `/apis/stable.example.com/v1/namespaces/<namespace>/crontabs/...` 的API

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # version name to use for REST API: /apis/<group>/<version>
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    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/crontabs/m
y-new-cron-object
  uid: 5c82083e-5fbd-11e7-a204-42010a8c0002
spec:
  cronSpec: '* * * * /5'
  image: my-awesome-cron-image
```

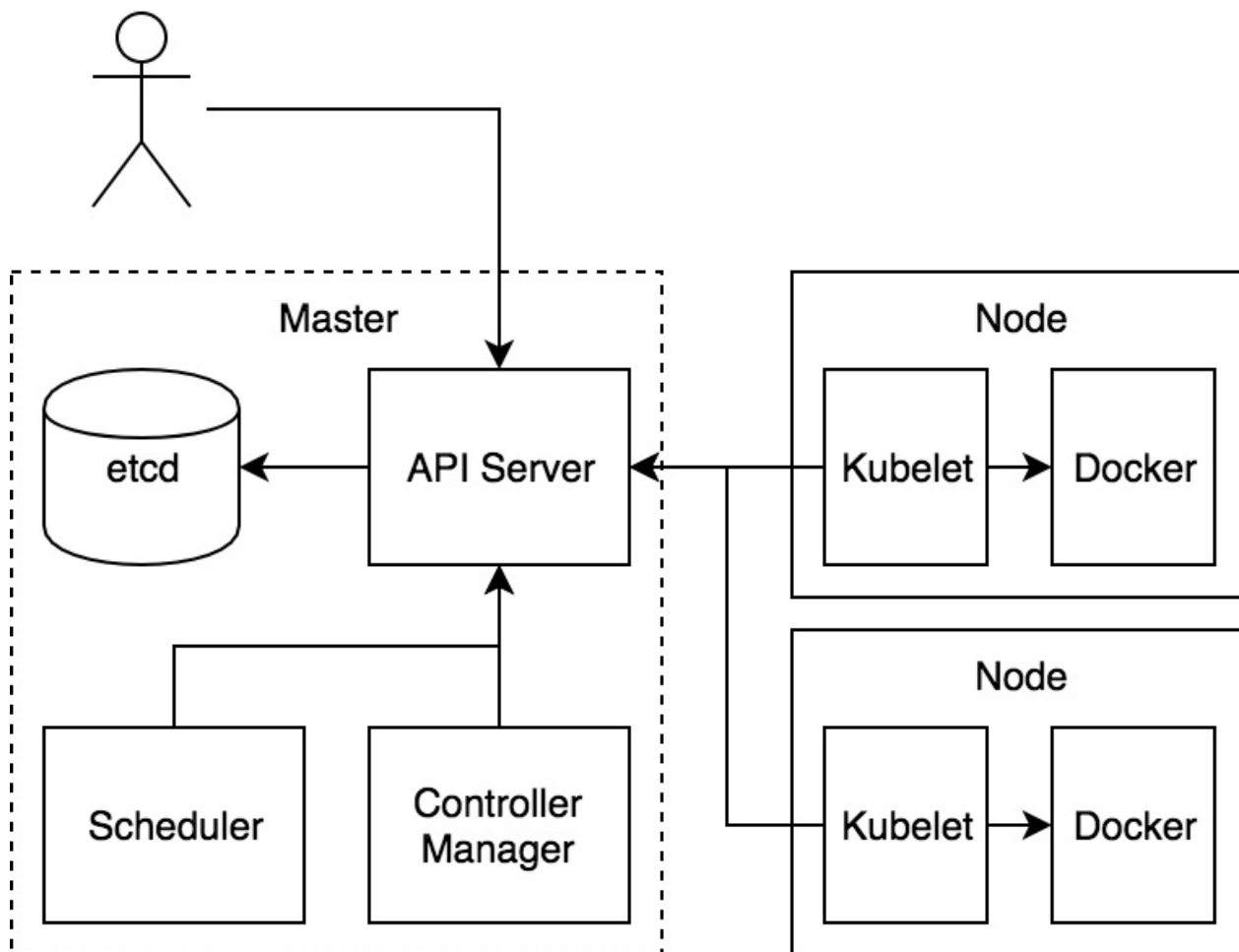
Finalizer

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
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
```

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。

核心组件



Kubernetes主要由以下几个核心组件组成:

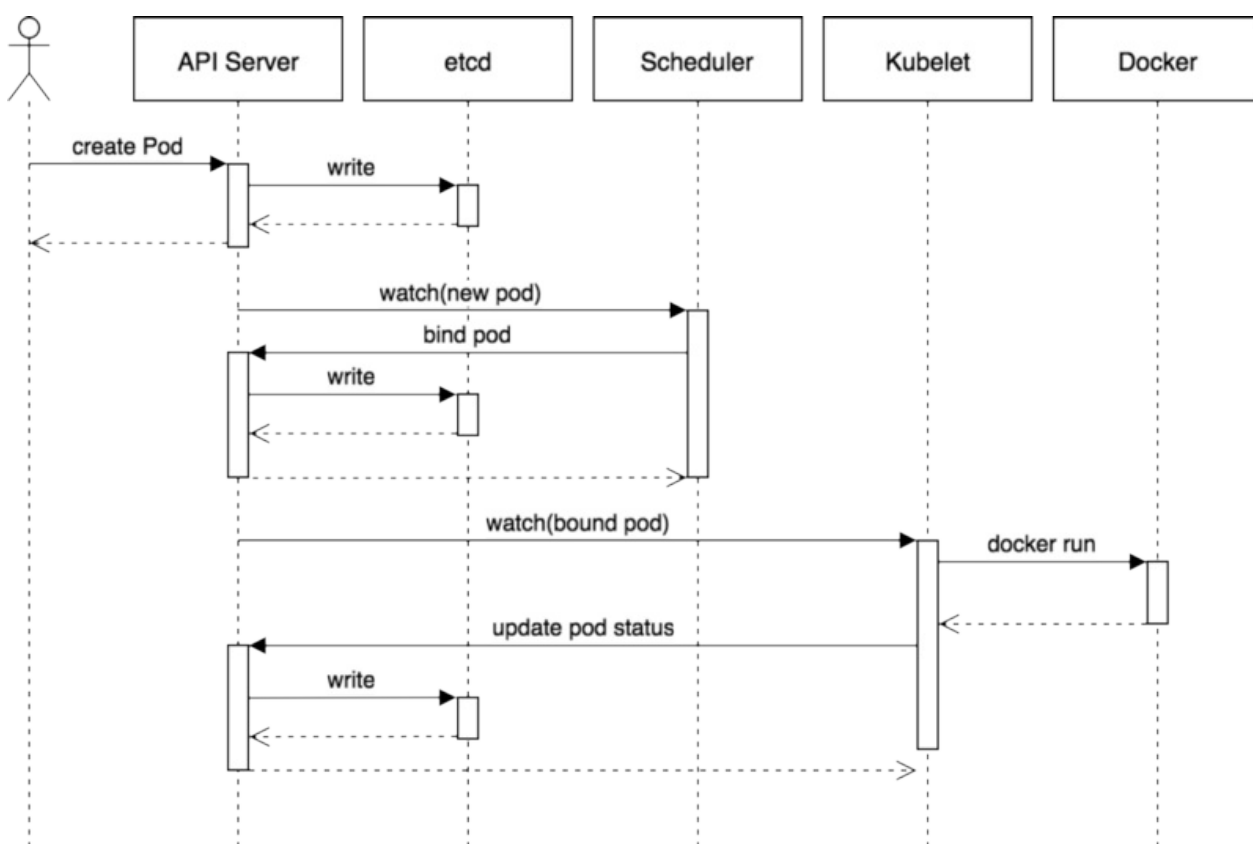
- **etcd**保存了整个集群的状态；
- **apiserver**提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- **controller manager**负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- **scheduler**负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- **kubelet**负责维护容器的生命周期，同时也负责Volume（CVI）和网络（CNI）的管理；
- **Container runtime**负责镜像管理以及Pod和容器的真正运行（CRI）；
- **kube-proxy**负责为Service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

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. scheduler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

参考文档

- [Master-Node communication](#)

- [Core Kubernetes: Jazz Improv over Orchestration](#)

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

当接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又会倍选举为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，想更进一步了解的可以去看源码，这里只简单分析下它的wal日志。

Entry			
type	term	index	data

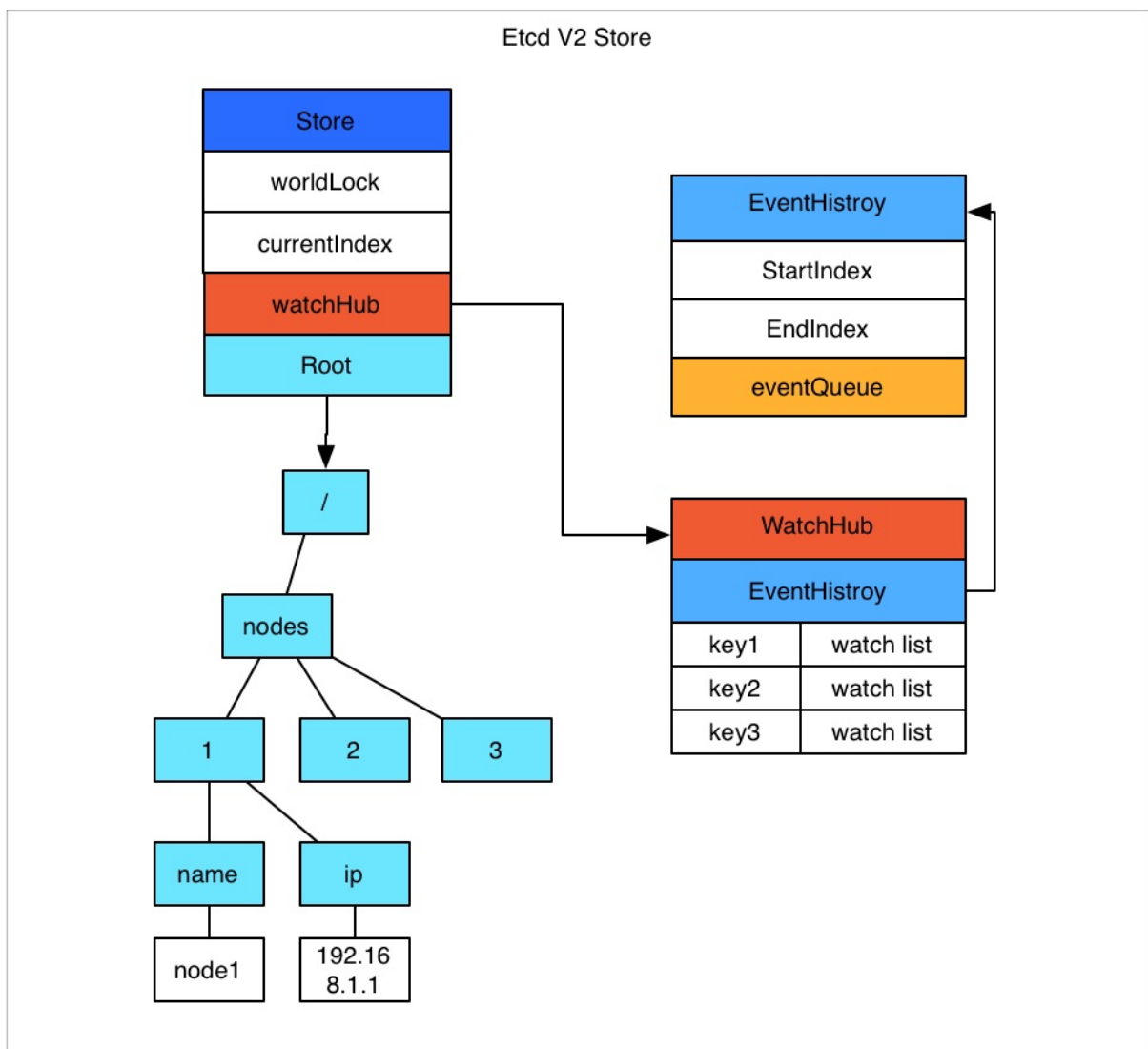
wal日志是二进制的，解析出来后是以上数据结构LogEntry。其中第一个字段type，只有两种，一种是0表示Normal，1表示ConfChange（ConfChange表示 Etcd 本身的配置变更同步，比如有新的节点加入等）。第二个字段是term，每个term代表一个主节点的任期，每次主节点变更term就会变化。第三个字段是index，这个序号是严格有序递增的，代表变更序号。第四个字段是二进制的data，将raft request对象的pb结构整个保存下。Etcd 源码下有个tools/etcd-dump-logs，可以将wal日志dump成文本查看，可以协助分析raft协议。

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。

Etcd v2 存储，Watch以及过期机制



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小于等于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。

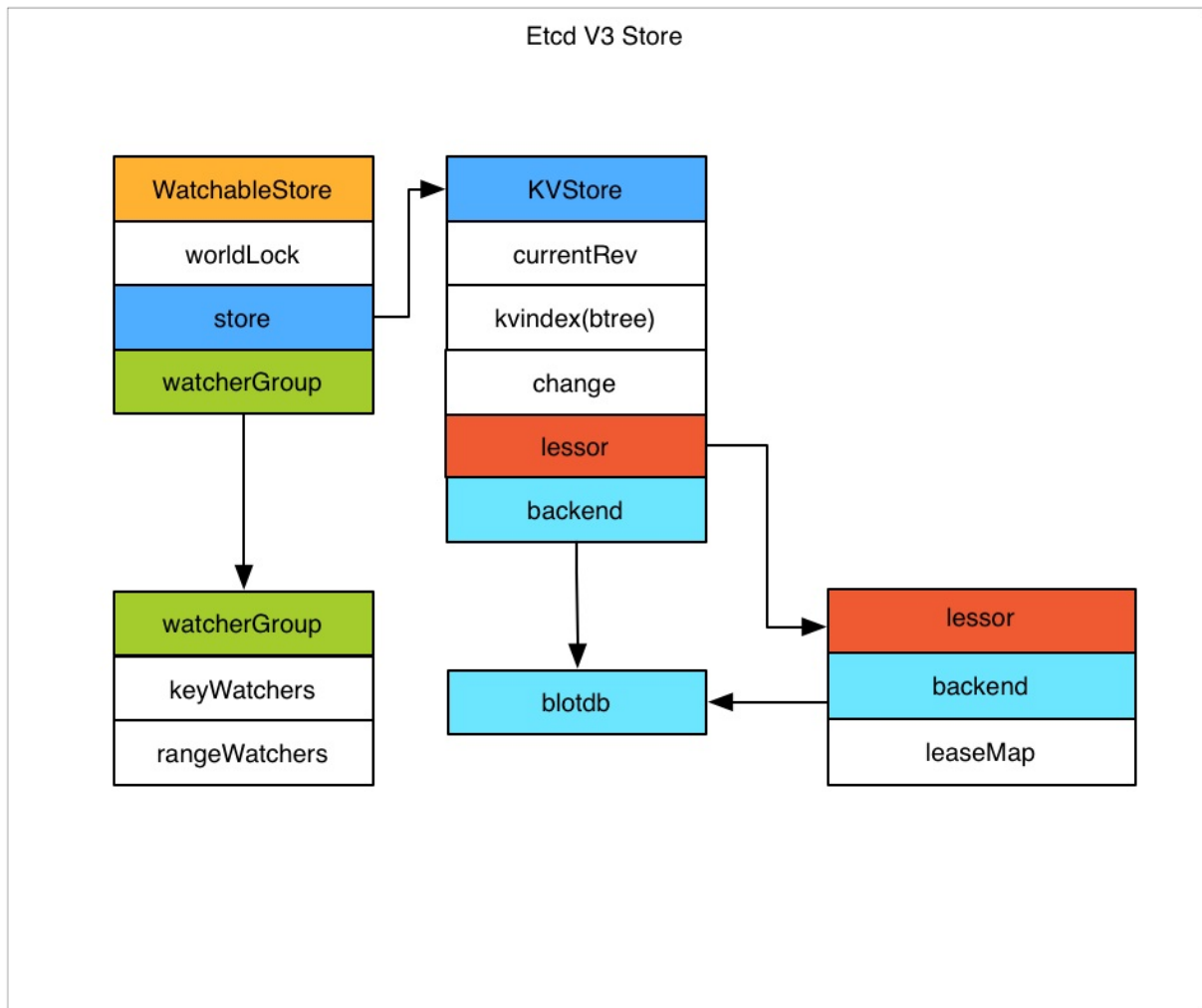
这里有几个影响使用的细节问题：

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好久，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watch的时候，出现了阻塞（每个watch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. watch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



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reversion，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"
```

reversion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reversion，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reversion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 range watchers, 数据结构是一个 IntervalTree（不熟悉的参看文末链接），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syncd，一种是unsyncd，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syncd组中，否则放入unsyncd组。同时 Etcd 会启动一个后台的goroutine持续同步unsyncd的watcher，然后将其迁移到syncd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那些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。[confd](#)

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: Etcserver: 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不能解决容量问题。要想解决容量问题，只能进行分片，但分片后如何使用raft同步数据？只能实现一个 multiple group raft，每个分片的多个副本组成一个虚拟的raft group，通过raft实现数据同步。当前实现了multiple group raft的有 TiKV 和 Cockroachdb，但尚未一个独立通用的。理论上来说，如果有了这套 multiple group raft，后面挂个持久化的kv就是一个分布式kv存储，挂个内存kv就是分布式缓存，挂个lucene就是分布式搜索引擎。当然这只是理论上，要真正实现复杂度还是不小。

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从应用场景到是实现原理的全方位解读](#)

API Server

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
```

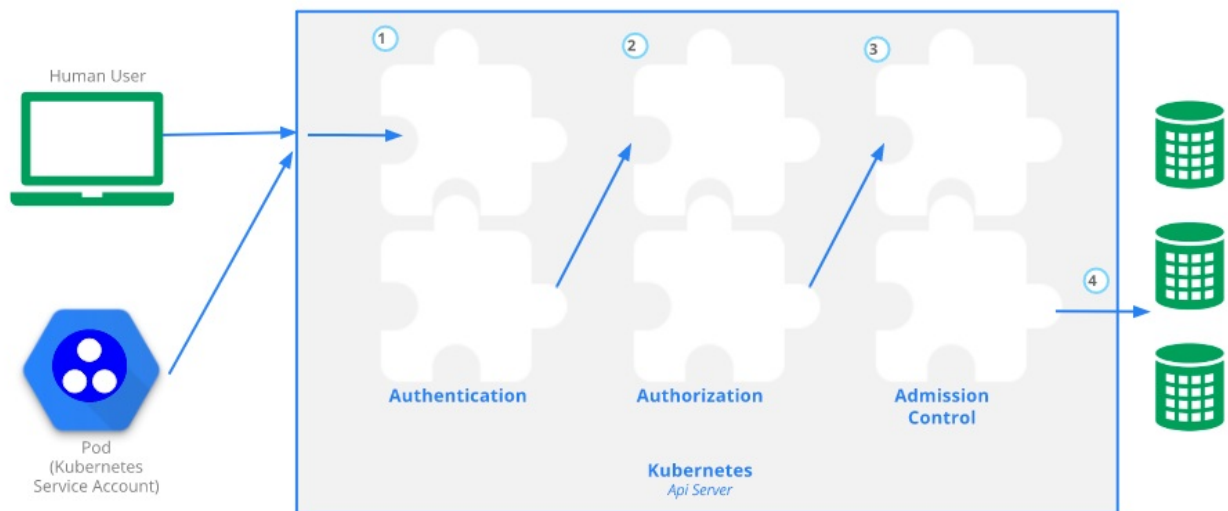
OpenAPI和Swagger

通过 `/swaggerapi` 可以查看Swagger API，`/swagger.json` 查看OpenAPI。

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

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。

[warning] Kubernetes不管理用户

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。

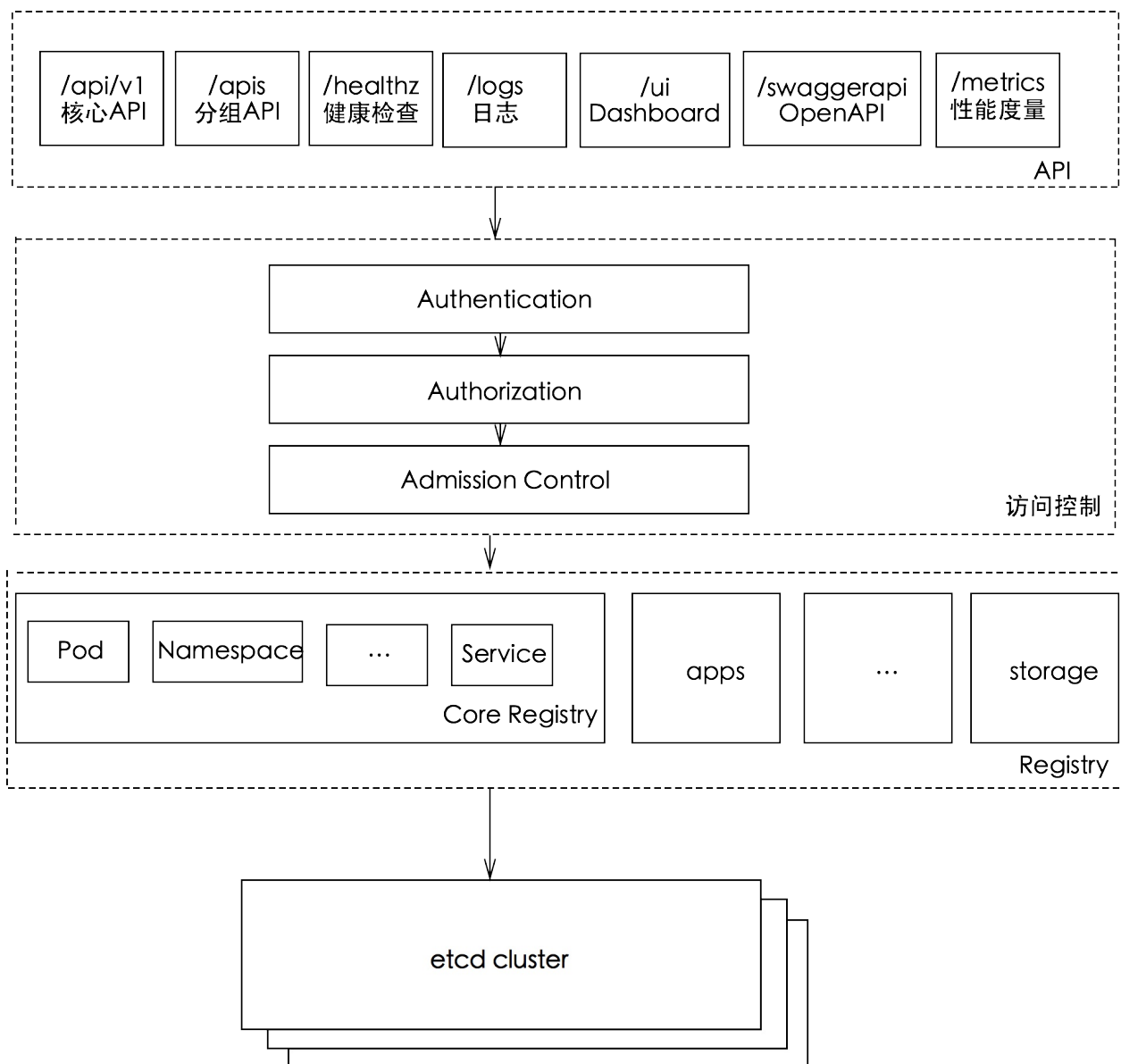
更多准入控制模块的使用方法可以参考[Kubernetes准入控制](#)。

启动apiserver示例

```
kube-apiserver --feature-gates=AllAlpha=true --runtime-config=api/all=true \
  --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-kubelet-client.crt \
  --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt \
  --insecure-port=8080 \
  --admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,
PersistentVolumeLabel,DefaultStorageClass,ResourceQuota,DefaultTolerationSeconds \
  --requestheader-group-headers=X-Remote-Group \
  --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key \
  --secure-port=6443 \
  --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname \
  --service-cluster-ip-range=10.96.0.0/12 \
  --authorization-mode=RBAC \
  --advertise-address=192.168.0.20 --etcd-servers=http://127.0.0.1:2379
```

kube-apiserver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）。



访问API

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

参考文档

- [v1.5 API Reference](#)
- [v1.6 API Reference](#)
- [v1.7 API Reference](#)

API Aggregation

API Aggregation允许在不修改Kubernetes核心代码的同时扩展Kubernetes API。

开启API Aggregation

kube-apiserver增加以下配置

```
--requestheader-client-ca-file=<path to aggregator CA cert>
--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=<path to aggregator proxy cert>
--proxy-client-key-file=<path to aggregator proxy key>
```

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 <resource-name>`，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 <your-domain>
$ apiserver-boot init glide

# 创建资源
$ apiserver-boot create group version resource --group <group> --version <version> --kind <Kind>

# 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 --namespace default --image gcr.io/myrepo/myimage:mytag
$ kubectl create -f sample/<type>.yaml
```

示例

见[sample-apiserver](#)和[apiserver-builder/example](#)。

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。

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"}]`

启动**kube-scheduler**示例

```
kube-scheduler --address=127.0.0.1 --leader-elect=true --kubeconfig=/etc/kubernetes/scheduler.conf
```

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的亲合性要求
- 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]

[warning] 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-scheduler的入口在 `plugin/cmd/kube-scheduler`。

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 microseconds of adding an object to etcd cache
# 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 microseconds of getting an object from etcd cache
# 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


```
kube-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 \  
  --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt \  
  --address=127.0.0.1 \  
  --leader-elect=true \  
  --use-service-account-credentials=true \  
  --controllers=*,bootstrapsigner,tokencleaner \  
  --kubeconfig=/etc/kubernetes/controller-manager.conf \  
  --insecure-experimental-approve-all-kubelet-csrs-for-group=system  
:bootstrappers \  
  --root-ca-file=/etc/kubernetes/pki/ca.crt \  
  --service-account-private-key-file=/etc/kubernetes/pki/sa.key \  
  --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
```

Controller Manager工作原理

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

如何保证高可用

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](https://github.com/kubernetes/cloud-controller-manager/blob/master/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。

如何保证高性能

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](#)。

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。

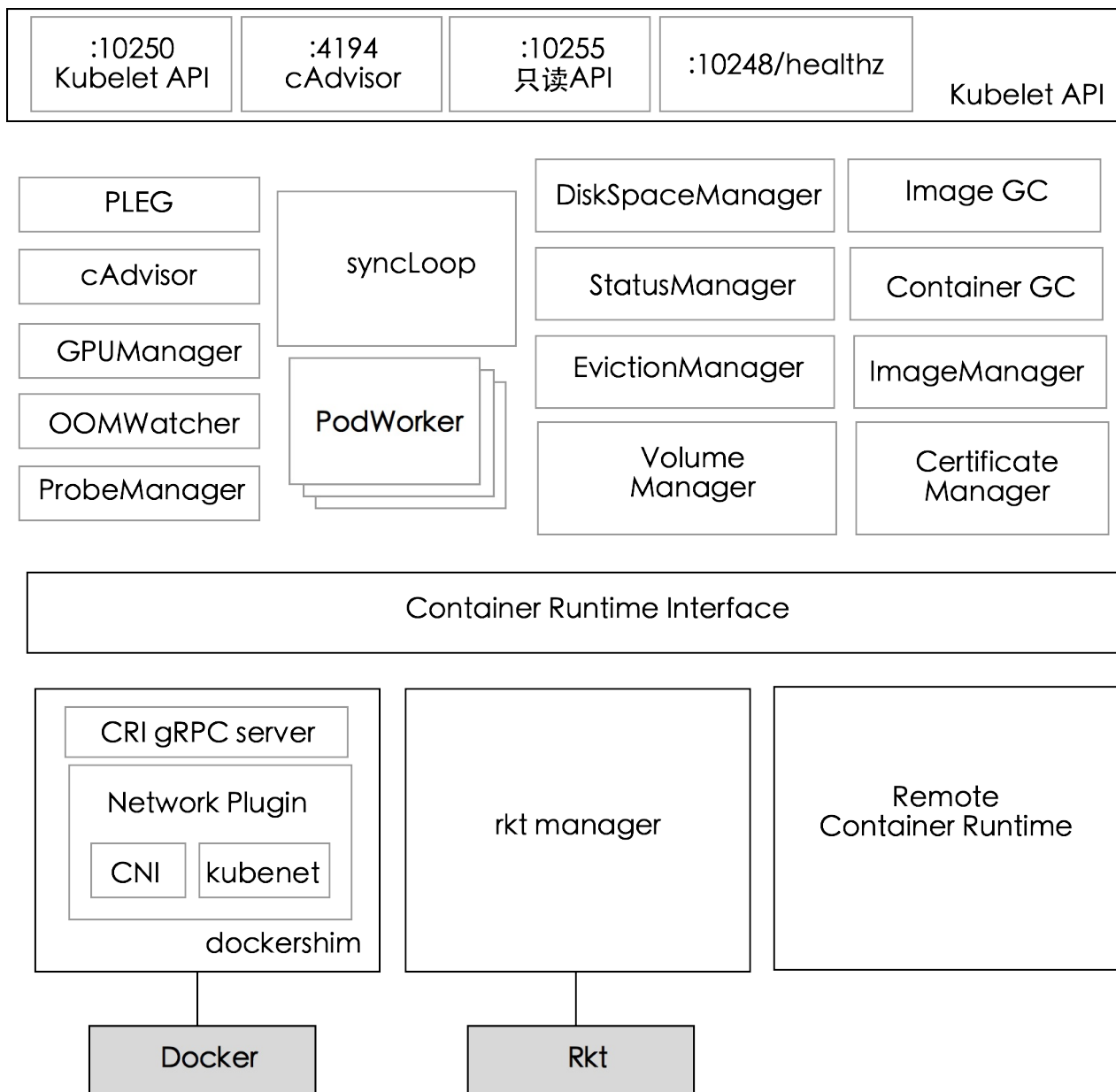
启动kubelet示例

```
/usr/bin/kubelet --kubeconfig=/etc/kubernetes/kubelet.conf \  
  --require-kubeconfig=true \  
  --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 \  
  --feature-gates=AllAlpha=true
```

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

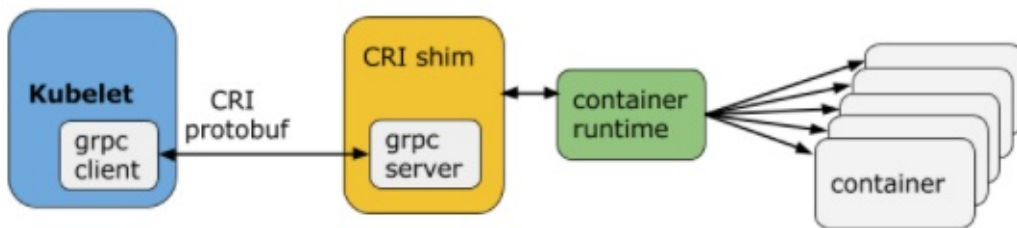


容器运行时

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。

CRI

Container Runtime Interface (CRI)是Kubelet 1.5/1.6中主要负责的一块项目，它重新定义了Kubelet Container Runtime API，将原来完全面向Pod级别的API拆分成面向Sandbox和Containers的API，并分离镜像管理和容器引擎到不同的服务。



Docker

Docker runtime的核心代码在kubelet内部，是最稳定和特性支持最好的Runtime。

开源电子书《[Docker从入门到实践](#)》是docker入门和实践不错的参考。

Hyper

[Hyper](#)是一个基于Hypervisor的容器运行时，为Kubernetes带来了强隔离，适用于多租户和运行不可信容器的场景。

Hyper在Kubernetes的集成项目为frakti，<https://github.com/kubernetes/frakti>，目前已支持Kubernetes v1.6+。

Rkt

rkt是另一个集成在kubelet内部的容器运行时，但也正在迁往CRI的路上，<https://github.com/kubernetes-incubator/rktlet>。

Runc

Runc有两个实现，`cri-o`和`cri-containerd`

- `cri-containerd`，还在开发中
- `cri-o`，已支持Kubernetes v1.6

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（社区有人提到过几万条），大规模下也有性能问题
- **winuserspace**：同userspace，但仅工作在windows上

另外，基于ipvs的方案正在讨论中（见[#44063](#)和[slide](#)），大规模情况下可以大幅提升性能，比如slide里面提供的示例将服务延迟从小时缩短到毫秒级。

Iptables 示例

```

-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 traffic
requiring SNAT" -m mark --mark 0x4000/0x4000 -j MASQUERADE

-A KUBE-SEP-55QZ6T7MF3AHP00B -s 10.244.1.6/32 -m comment --comment "d
efault/http:" -j KUBE-MARK-MASQ
-A KUBE-SEP-55QZ6T7MF3AHP00B -p tcp -m comment --comment "default/htt
p:" -m tcp -j DNAT --to-destination 10.244.1.6:80

-A KUBE-SEP-KJZJRL2KRWMXNR3J -s 10.244.1.5/32 -m comment --comment "d
efault/http:" -j KUBE-MARK-MASQ
-A KUBE-SEP-KJZJRL2KRWMXNR3J -p tcp -m comment --comment "default/htt
p:" -m tcp -j DNAT --to-destination 10.244.1.5:80

-A KUBE-SERVICES -d 10.101.85.234/32 -p tcp -m comment --comment "def
ault/http: cluster IP" -m tcp --dport 80 -j KUBE-SVC-7IMAZDGB20NQNK4Z
-A KUBE-SVC-7IMAZDGB20NQNK4Z -m comment --comment "default/http:" -m
statistic --mode random --probability 0.500000000000 -j KUBE-SEP-KJZJR
L2KRWMXNR3J
-A KUBE-SVC-7IMAZDGB20NQNK4Z -m comment --comment "default/http:" -j
KUBE-SEP-55QZ6T7MF3AHP00B

```

启动kube-proxy示例

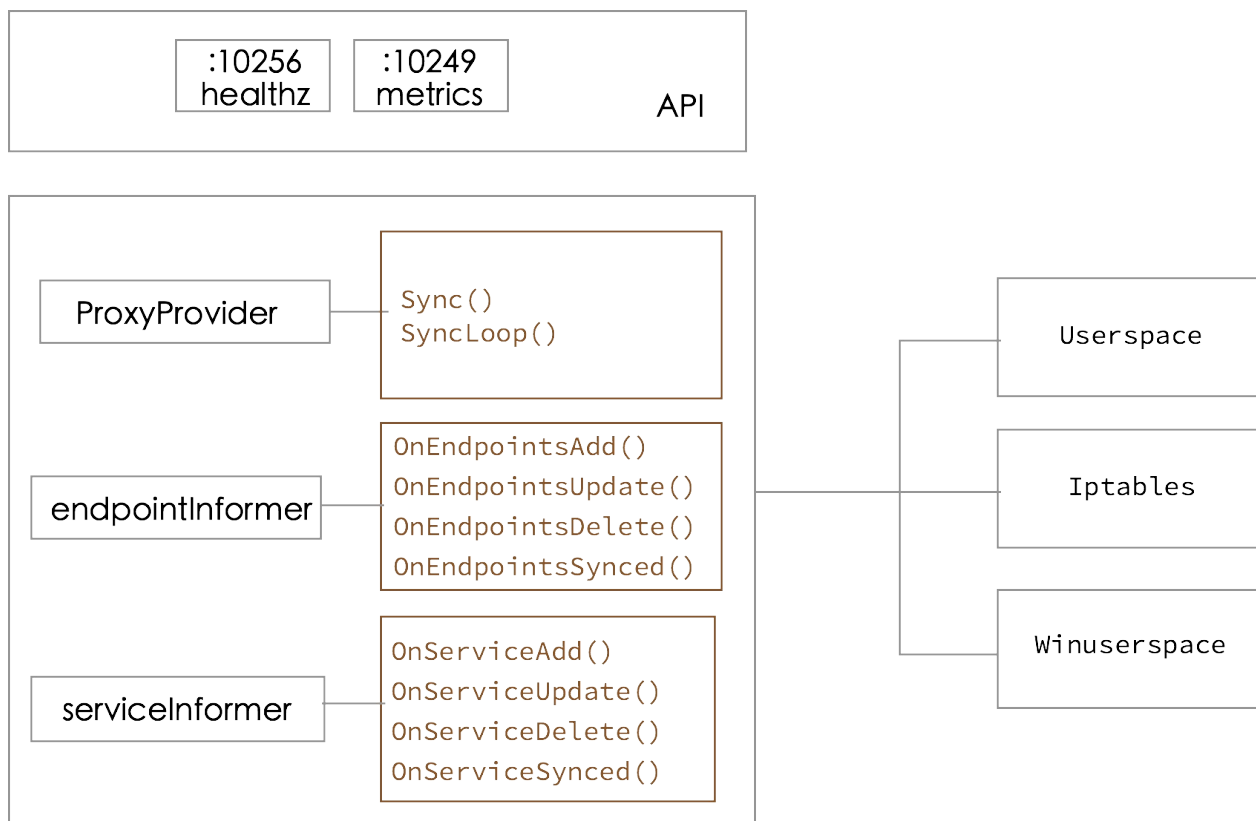
```
kube-proxy --kubeconfig=/var/lib/kube-proxy/kubeconfig.conf
```

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-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等proxier来为服务配置负载均衡（仅支持TCP和UDP）。



kube-dns

kube-dns为Kubernetes集群提供命名服务，一般通过addon的方式部署，从v1.3版本开始，成为了一个内建的自启动服务。

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

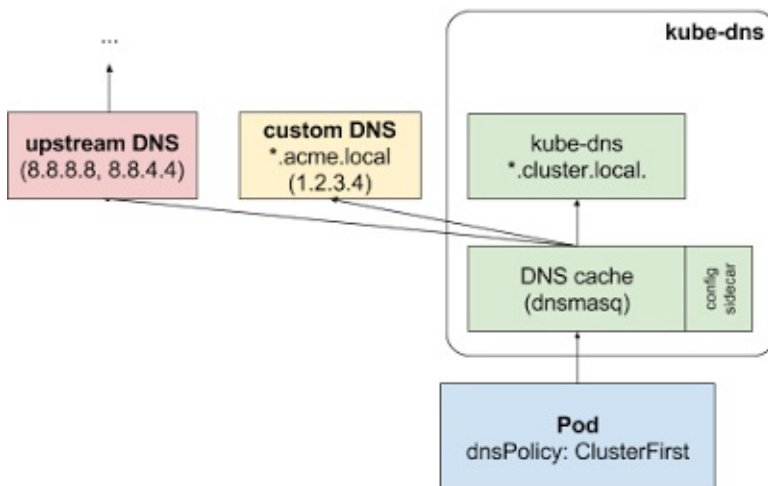
```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

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。



启动kube-dns示例

一般通过addon的方式部署DNS服务，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=/kube-
dns-config --v=2

# dnsmasq container
dnsmasq-nanny -v=2 -logtostderr -configDir=/etc/k8s/dns/dnsmasq-nanny
  -restartDnsmasq=true -- -k --cache-size=1000 --log-facility=- --serv
er=127.0.0.1#10053

# sidecar container
sidecar --v=2 --logtostderr --probe=kubedns,127.0.0.1:10053,kubernete
s.default.svc.cluster.local.,5,A --probe=dnsmasq,127.0.0.1:53,kuberne
tes.default.svc.cluster.local.,5,A
```

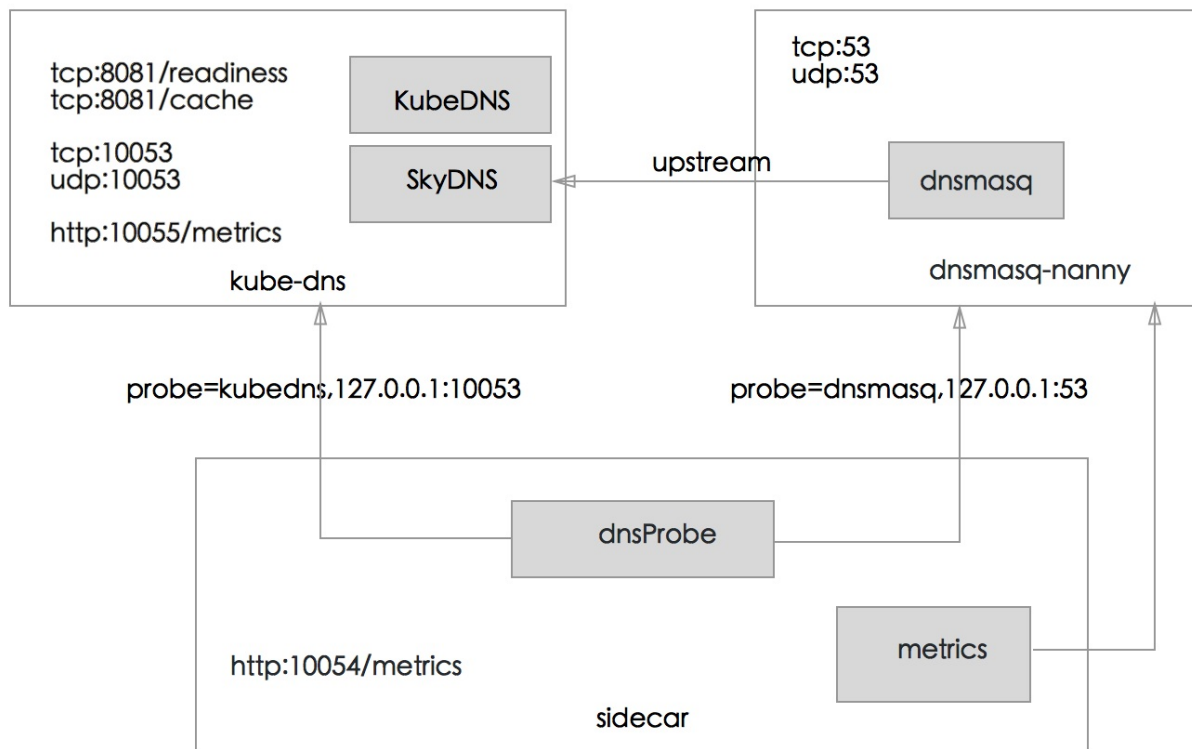
参考文档

- [dns-pod-service](#) 介绍

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）



源码分析

kube-dns的代码已经从kubernetes里面分离出来，放到了

<https://github.com/kubernetes/dns>。

[info] 源码版本

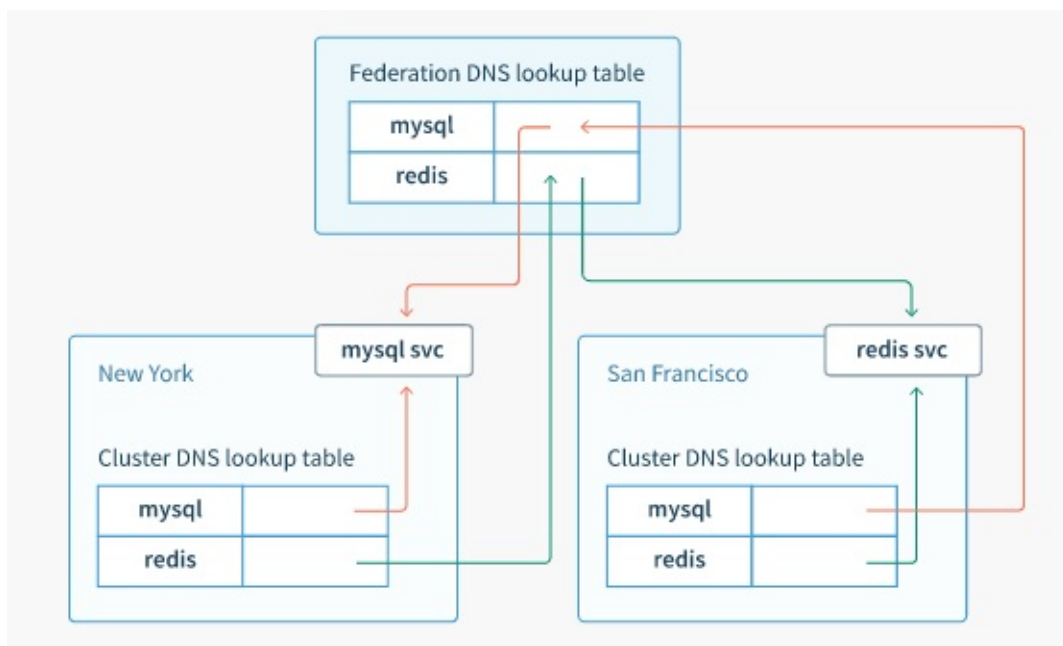
本节内容基于Kubernetes DNS 1.14.2版本。

kube-dns、dnsmasq-nanny和sidecar的代码均是从 `cmd/<cmd-name>/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](https://github.com/skynetservices/skydns)。

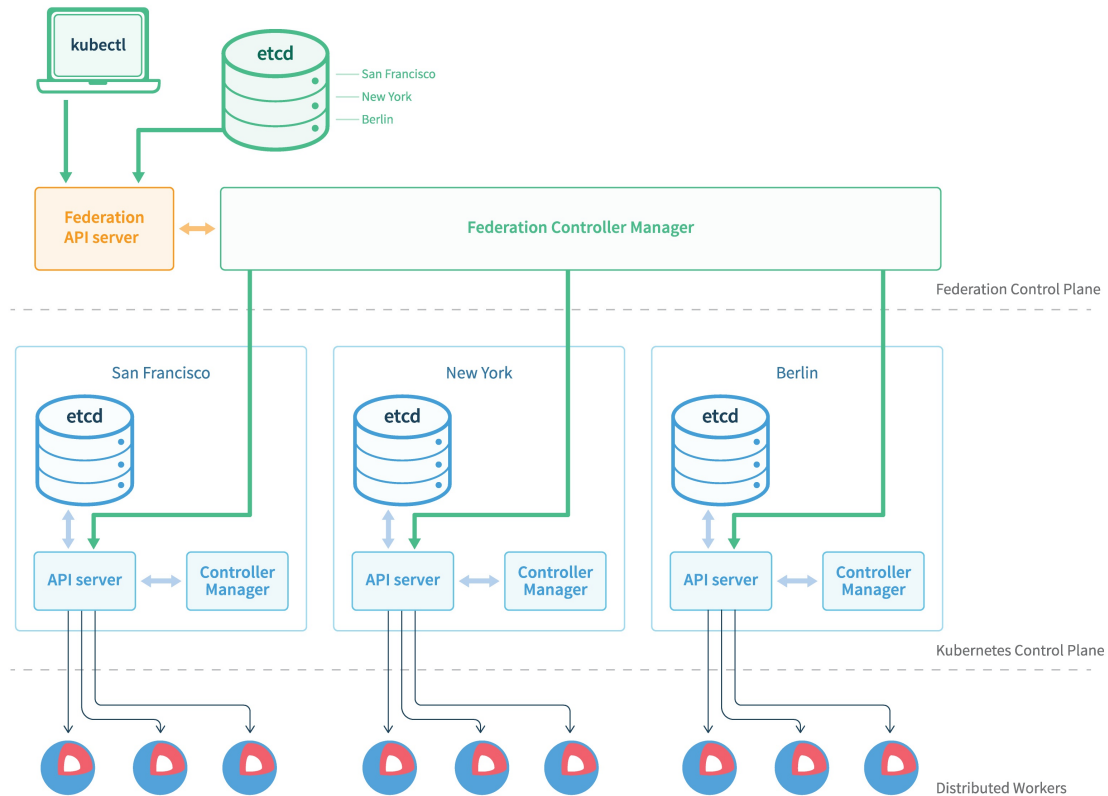
Federation

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部署方法

下载kubefed和kubectl

kubefed下载

```
# Linux
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/kubernetes-client-linux-amd64.tar.gz
tar -xzvf kubernetes-client-linux-amd64.tar.gz

# OS X
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/kubernetes-client-darwin-amd64.tar.gz
tar -xzvf kubernetes-client-darwin-amd64.tar.gz

# Windows
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/kubernetes-client-windows-amd64.tar.gz
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配置名称
  --dns-provider="google-clouddns" \     # DNS服务提供商，还支持aws-rout
e53或coredns
  --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" # federa
tion API server自定义参数
$ 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 stable/etcd-operator
$ helm upgrade --namespace my-namespace --set cluster.enabled=true etcd-operator stable/etcd-operator
```

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.yaml stable/coredns
```

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配置名称
  --dns-provider="coredns" \             # DNS服务提供商，还支持aws-route53或google-clouddns
  --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配置

  --apiserver-arg-overrides="--anonymous-auth=false,--v=4" # federation API server自定义参数
```

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配置名称
  --dns-provider="coredns" \             # DNS服务提供商，还支持aws-route53或google-clouddns
  --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配置

  --apiserver-arg-overrides="--anonymous-auth=false,--v=4" \ # federation API server自定义参数
  --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-context=gondor_needs-no_king
```

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": "pci", "operator": "In", "values": ["true"]}, {"key": "environment", "operator": "NotIn", "values": ["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/kubernetes.github.io/master/docs/tutorials/federation/scheduling-policy-admission.yaml
```

(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.yaml
```

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/kubernetes.github.io/master/docs/tutorials/federation/policy-engine-service.yaml
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tutorials/federation/policy-engine-deployment.yaml
```

(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-scheduling-policy
```

(5) 创建策略

```
wget https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tutorials/federation/policy.rego
kubectl --context=federation -n kube-federation-scheduling-policy create configmap scheduling-policy --from-file=policy.rego
```

(6) 验证策略

```
kubectl --context=federation annotate clusters cluster-name-1 pci-certified=true
kubectl --context=federation create -f https://raw.githubusercontent.com/kubernetes/kubernetes.github.io/master/docs/tutorials/federation/replicaset-example-policy.yaml
kubectl --context=federation get rs nginx-pci -o jsonpath='{.metadata.annotations}'
```

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）

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.yaml

# 添加后端Pod
$ for CLUSTER in asia-east1-c asia-east1-a asia-east1-b \
    europe-west1-d europe-west1-c europe-west1-b \
    us-central1-f us-central1-a us-central1-b us-
central1-c \
    us-east1-d us-east1-c us-east1-b
do
    kubectl --context=$CLUSTER run nginx --image=nginx:1.11.1-alpine --
port=80
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工作原理

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 kubeadm init --pod-network-cidr 10.244.0.0/16 --kubernetes-version latest
```

这个命令会自动

- 系统状态检查
- 生成token
- 生成自签名CA和可独断证书
- 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": "10.244.0.0/16",
        "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

```
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel-rbac.yml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yml
```

weave

```
kubectl apply -f https://git.io/weave-kube-1.6
```

calico


```
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml
```

添加Node

```
token=$(kubeadm token list | grep authentication,signing | awk '{print $1}')
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-apiserer
- 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 <command> --help` 查看子命令的帮助
- `kubectl [command] [PARAMS] -o=<format>` 设置输出格式（如json、yaml、jsonpath等）

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 --password=secret
kubectl config set-cluster local-server --server=http://localhost:8080
kubectl config set-context default-context --cluster=local-server --user=myself --namespace=default
kubectl config use-context default-context
kubectl config view
```

常用命令格式

- 创建：`kubectl run <name> --image=<image>` 或者 `kubectl create -f manifest.yaml`

- 查询：`kubectl get <resource>`
- 更新
- 删除：`kubectl delete <resource> <name>` 或者 `kubectl delete -f manifest.yaml`
- 查询Pod IP：`kubectl get pod <pod-name> -o jsonpath='{.status.podIP}'`
- 容器内执行命令：`kubectl exec -ti <pod-name> sh`
- 容器日志：`kubectl logs [-f] <pod-name>`
- 导出服务：`kubectl expose deploy <name> --port=80`

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	<code>--restart=Never</code>
Replication Controller	<code>--generator=run/v1</code>
Deployment	<code>--restart=Always</code>
Job	<code>--restart=OnFailure</code>
CronJob	<code>--schedule=<cron></code>

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 from pod web-1
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 container by default
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-container from pod 123456-7890
# 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 container in the pod will be chosen
- 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 the first container by default
kubectl exec 123456-7890 date

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

# Switch to raw terminal mode, sends stdin to 'bash' in ruby-container from pod 123456-7890
# 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 container in the pod will be chosen
- 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 port 8888 locally, forwarding to 5000 in the pod
kubectl port-forward mypod 8888:5000
```

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 pod
in the default namespace
kubectl cp /tmp/foo_dir <some-pod>:/tmp/bar_dir

# Copy /tmp/foo local file to /tmp/bar in a remote pod in a specific
container
kubectl cp /tmp/foo <some-pod>:/tmp/bar -c <specific-container>

# Copy /tmp/foo local file to /tmp/bar in a remote pod in namespace
<some-namespace>
kubectl cp /tmp/foo <some-namespace>/<some-pod>:/tmp/bar

# Copy /tmp/foo from a remote pod to /tmp/bar locally
kubectl cp <some-namespace>/<some-pod>:/tmp/foo /tmp/bar
```

Options:

`-c`, `--container=''`: Container name. If omitted, the first container in the pod will be chosen

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。

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!
```

附录

kubectl的安装方法

```
# OS X
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/bin/darwin/amd64/kubectl

# Linux
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/bin/linux/amd64/kubectl

# Windows
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(
curl -s https://storage.googleapis.com/kubernetes-release/release/sta
ble.txt)/bin/windows/amd64/kubectl.exe
```

Kubernetes部署指南

- 单机部署
- 集群部署
 - [kubeadm](#)
 - [kargo](#)
 - [frakti](#)
 - 证书生成示例
- [kubectl](#)客户端
- 在CentOS上部署kubernetes1.6集群
 - (1) 创建TLS证书和秘钥
 - (2) 创建kubeconfig 文件
 - (3) 创建高可用etcd集群
 - (4) 安装kubectl命令行工具
 - (5) 部署高可用master集群
 - (6) 部署node节点
 - (7) 安装kubedns插件
 - (8) 安装dashboard插件
 - (9) 安装heapster插件
 - (10) 安装EFK插件
- 其他部署方法

单机部署

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

首先下载kubectl

```
curl -Lo kubectl https://storage.googleapis.com/kubernetes-release/release/v1.6.4/bin/linux/amd64/kubectl
chmod +x kubectl
```

安装minikube

```
# install minikube
$ brew cask install minikube
$ brew install docker-machine-driver-xhyve
# docker-machine-driver-xhyve need root owner and uid
$ sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xhyve
$ sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xhyve
```

最后启动minikube

```
# start minikube.
# http proxy is required in China
$ minikube start --docker-env HTTP_PROXY=http://proxy-ip:port --docker-env HTTPS_PROXY=http://proxy-ip:port --vm-driver=xhyve
```

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

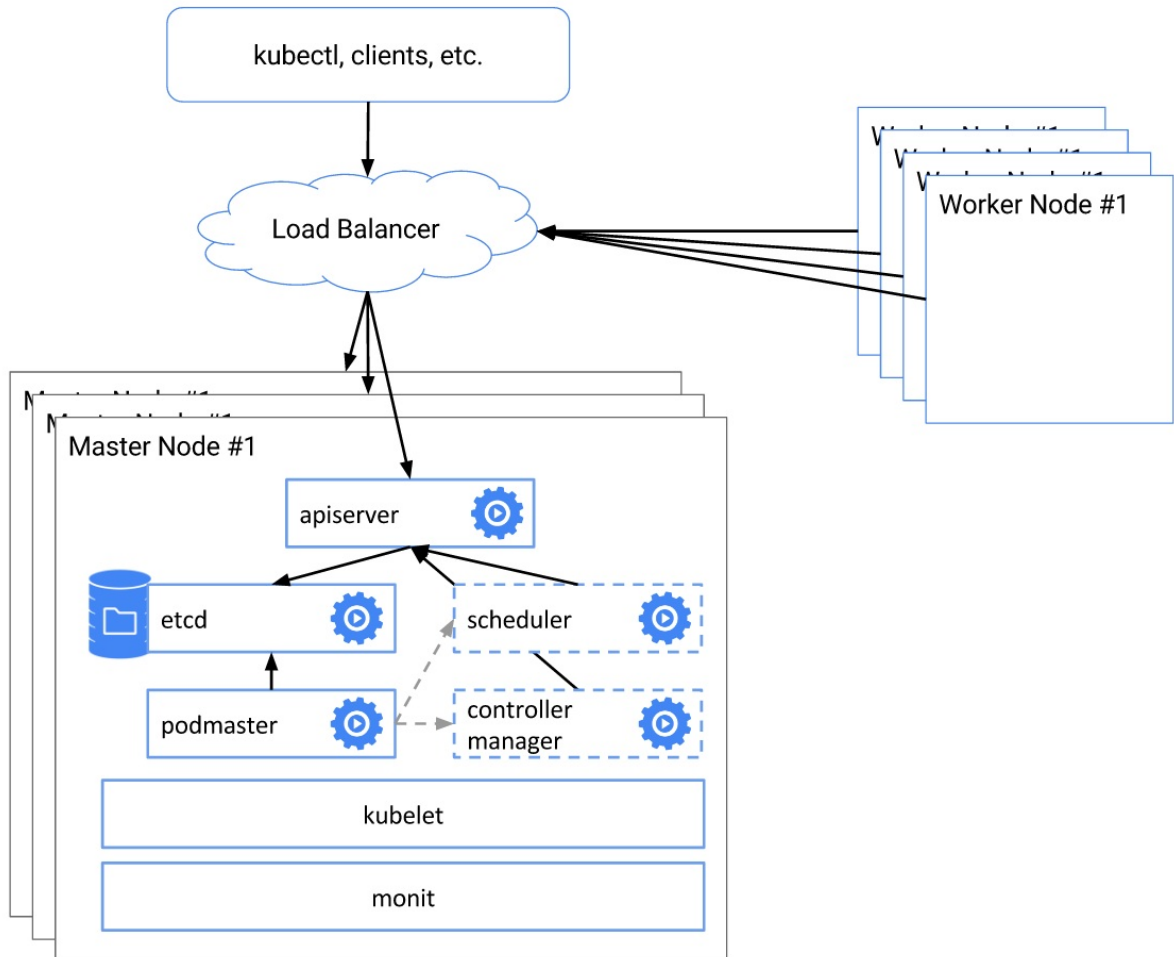
export KUBERNETES_PROVIDER=local
hack/install-etcd.sh
export PATH=$GOPATH/src/k8s.io/kubernetes/third_party/etcd:$PATH
hack/local-up-cluster.sh
```

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/kubect1.sh
```

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 <https://kubernetes.io/docs/admin/high-availability/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}`，既可以由 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<https://kubernetes.io/docs/admin/high-availability/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=10.245.0.0/16 --allocate-node-cidrs=true --service-account-private-key-file=/srv/kubernetes/server.key --v=2 --leader-elect=true
```

把[kube-scheduler.yaml](#)和[kube-controller-manager.yaml](#)(非GCE平台需要适当修改)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。

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

GCE/Azure

在GCE或者Azure上面可以利用cluster脚本方便的部署集群：

```
# gce,aws,gke,azure-legacy,vsphere,openstack-heat,rackspace,libvirt-c
oreos
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来部署Kubernetes集群。

kubeadm

统一化安装脚本

```
# on master
git clone https://github.com/feiskyer/ops
cd ops
kubernetes/setup_kubernetes.sh

# on node
git clone https://github.com/feiskyer/ops
cd ops
export TOKEN=xxxxxx
export MASTER_IP=xx.xx.xx.xx
kubernetes/add_docker_node.sh
```

以下是详细的安装步骤。

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
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key
add -
cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
# Install docker if you don't have it already.
apt-get install -y docker.io
apt-get install -y kubelet kubeadm kubectl kubernetes-cni
systemctl enable docker && systemctl start docker

systemctl enable kubelet && systemctl start kubelet
```

centos

```
cat <<EOF > /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.gpg
EOF
setenforce 0
yum install -y docker kubelet kubeadm kubectl kubernetes-cni
systemctl enable docker && systemctl start docker

systemctl enable kubelet && systemctl start kubelet
```

安装master

```
# --api-advertise-addresses <ip-address>
# for flannel, setup --pod-network-cidr 10.244.0.0/16
kubeadm init kubeadm init --pod-network-cidr 10.244.0.0/16 --kubernet
es-version latest

# enable schedule pods on the master
export KUBECONFIG=/etc/kubernetes/admin.conf
# for v1.5-, use kubectl taint nodes --all dedicated-
kubectl taint nodes --all node-role.kubernetes.io/master:NoSchedule-
```

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

```
#kubectl apply -f https://gist.githubusercontent.com/feiskyer/1e7a95f27c391a35af47881eb20131d7/raw/4266f05355590fa185bc8e50c0f50d2841993d20/flannel.yaml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel-rbac.yml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yml
```

weave

```
# kubectl apply -f https://gist.githubusercontent.com/feiskyer/0b00688584cc7ed9bd9a993adddae5e3/raw/67f3558e32d5c76be38e36ef713cc46deb2a74ca/weave.yaml
kubectl apply -f https://git.io/weave-kube-1.6
```

calico

```
# kubectl apply -f https://gist.githubusercontent.com/feiskyer/0f952c7dadbfcefd2ce81ba7ea24a8ca/raw/92addea398bbc4d4a1dcff8a98c1ac334c8acb26/calico.yaml
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernetes/installation/hosted/kubeadm/1.6/calico.yaml
```

添加Node

```
token=$(kubeadm token list | grep authentication,signing | awk '{print $1}')
kubeadm join --token $token ${master_ip}
```

删除安装

```
kubeadm reset
```

动态升级

kubeadm将在v1.8支持动态升级，目前升级还需要手动操作。

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`

Kubespray 集群安装

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 v1.6.1
- Etcd v3.1.6
- Flannel v0.7.1
- 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/host 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' /etc/
apt/sources.list
$ 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 cowsay p
ython-pip python-netaddr libssl-dev
```

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
```

安装完成后，新增配置档 /etc/kubespray/kubespray.yml，並加入以下内容：

```
$ mkdir /etc/kubespray
$ cat <<EOF > /etc/kubespray/kubespray.yml
kubespray_git_repo: "https://github.com/kubernetes-incubator/kubespra
y.git"
# Logging options
loglevel: "info"
EOF
```

接着用 kubespray cli 来产生 inventory 文件：

```
$ kubespray prepare --masters master1 --etcds master1 --nodes node1 node2 node3
$ cat ~/.kubespray/inventory/inventory.cfg
```

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 flannel
Run kubernetes cluster deployment with the above command ? [Y/n]y
...
master1                : ok=368  changed=89  unreachable=0    failed=0
node1                  : ok=305  changed=73  unreachable=0    failed=0
node2                  : ok=276  changed=62  unreachable=0    failed=0
node3                  : ok=276  changed=62  unreachable=0    failed=0

Kubernetes deployed successfully
```

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:"v1.6.1+coreos.0", GitCommit:"9212f77ed8c169a0afa02e58dce87913c6387b3e", GitTreeState:"clean", BuildDate:"2017-04-04T00:32:53Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.1+coreos.0", GitCommit:"9212f77ed8c169a0afa02e58dce87913c6387b3e", GitTreeState:"clean", BuildDate:"2017-04-04T00:32:53Z", GoVersion:"go1.7.5", Compiler:"gc", Platform:"linux/amd64"}
```

取得当前集群节点状态：

```
$ kubectl get node
NAME                STATUS              AGE      VERSION
master1             Ready,SchedulingDisabled 11m      v1.6.1+coreos.0
node1                Ready               11m      v1.6.1+coreos.0
node2                Ready               11m      v1.6.1+coreos.0
node3                Ready               11m      v1.6.1+coreos.
```

查看当前集群 Pod 状态：


```
$ kubectl get po -n kube-system
```

NAME	READY	STATUS	RESTARTS
AGE			
dnsmasq-975202658-6jj3n	1/1	Running	0
14m			
dnsmasq-975202658-h4rn9	1/1	Running	0
14m			
dnsmasq-autoscaler-2349860636-kfpx0	1/1	Running	0
14m			
flannel-master1	1/1	Running	1
14m			
flannel-node1	1/1	Running	1
14m			
flannel-node2	1/1	Running	1
14m			
flannel-node3	1/1	Running	1
14m			
kube-apiserver-master1	1/1	Running	0
15m			
kube-controller-manager-master1	1/1	Running	0
15m			
kube-proxy-master1	1/1	Running	1
14m			
kube-proxy-node1	1/1	Running	1
14m			
kube-proxy-node2	1/1	Running	1
14m			
kube-proxy-node3	1/1	Running	1
14m			
kube-scheduler-master1	1/1	Running	0
15m			
kubedns-1519522227-thmrh	3/3	Running	0
14m			
kubedns-autoscaler-2999057513-tx14j	1/1	Running	0
14m			
nginx-proxy-node1	1/1	Running	1
14m			
nginx-proxy-node2	1/1	Running	1
14m			
nginx-proxy-node3	1/1	Running	1
14m			

Frakti运行时部署指南

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/allinone.sh | bash
```

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/v0.2
/frakti -o /usr/bin/frakti
chmod +x /usr/bin/frakti
cgroup_driver=$(docker info | awk '/Cgroup Driver/{print $3}')
cat <<EOF > /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 | apt-key
add -
cat <<EOF > /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 <<EOF > /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.gpg
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=remote --container-runtime-endpoint=/var/run/frakti.sock --feature-gates=AllAlpha=true"' /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
systemctl daemon-reload
```

配置Master

```
kubeadm init kubeadm init --pod-network-cidr 10.244.0.0/16 --kubernetes-version latest
```

```
# Optional: enable schedule pods on the master
```

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

```
kubectl taint nodes --all node-role.kubernetes.io/master:NoSchedule-
```

配置Node

```
# get token on master node
```

```
token=$(kubeadm token list | grep authentication,signing | awk '{print $1}')
```

```
# 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](#)

kubectl客户端

kubectl的安装方法

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/release/$(  
curl -s https://storage.googleapis.com/kubernetes-release/release/sta  
ble.txt)/bin/darwin/amd64/kubectl
```

Linux

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(  
curl -s https://storage.googleapis.com/kubernetes-release/release/sta  
ble.txt)/bin/linux/amd64/kubectl
```

Windows

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(  
curl -s https://storage.googleapis.com/kubernetes-release/release/sta  
ble.txt)/bin/windows/amd64/kubectl.exe
```

kubectl使用方法

kubectl的详细使用方法请参考[kubectl指南](#)。

Kubernetes Addons

- [Dashboard](#)
- [Heapster](#)
- [EFK](#)

Kubernetes Dashboard

Kubernetes Dashboard的部署非常简单，只需要运行

```
kubectl create -f https://git.io/kube-dashboard
```

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)
AGE
kubernetes-dashboard              10.101.211.212   <nodes>          80:32729/TCP
1m
$ kubectl -n kube-system describe service kubernetes-dashboard
Name:                               kubernetes-dashboard
Namespace:                           kube-system
Labels:                               app=kubernetes-dashboard
Annotations:                           <none>
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:                                <none>
```

然后就可以通过 `http://nodeIP:32729` 来访问了。

https

通常情况下，建议Dashboard服务以https的方式运行，在访问它之前我们需要将证书导入系统中：

```
openssl pkcs12 -export -in apiserver-kubelet-client.crt -inkey apiserver-kubelet-client.key -out kube.p12
curl -sSL -E ./kube.p12:password -k https://nodeIP:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
```

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。

Heapster

```
git clone https://github.com/kubernetes/heapster
cd heapster
kubectl create -f deploy/kube-config/influxdb/
```

Elasticsearch Fluentd Kibana (EFK)

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

具体使用方法可以参考[Kubernetes日志处理](#)。

Kubernetes配置最佳实践

本文翻译自Kubernetes官方文档[Configuration Best Practices](#)。

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（目前是V1）。
- 在配置文件push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作为数据交换格式，但是YAML格式比起JSON更易读和配置。
- 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（注意，尽管你可以在使用 `kubectl` 命令时指定配置文件目录，你也可以在配置文件目录下执行 `kubectl create` ——查看下面的详细信息）。
- 为了简化和最小化配置，也为了防止错误发生，不要指定不必要的默认配置。例如，省略掉 `ReplicationController` 的 `selector` 和 `label`，如果你希望它们跟 `podTemplate` 中的 `label` 一样的话，因为那些配置默认是 `podTemplate` 的 `label` 产生的。更多信息请查看 [guestbook app](#) 的 `yml` 文件和 [examples](#)。
- 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](#)（没有这个必要

吧？）你也可以在创建Replication Controller的时候不指定replica数量（默认是1），创建service后，在通过Replication Controller来扩容。这样可以在扩容很多个replica之前先确认pod是正常的。

- 除非十分必要的情况下（如运行一个node daemon），不要使用 `hostPort`（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可被调度到的主机的受限了，因为端口冲突。如果是为了调试目的来通过端口访问的话，你可以使用 `kubect proxy and apiserver proxy` 或者 `kubect 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`。

使用Label

- 定义 `labels` 来指定应用或Deployment的 **semantic attributes**。For example, instead of attaching a label to a set of pods to explicitly represent some service (e.g., `service: myservice`), or explicitly representing the replication controller managing the pods (e.g., `controller: mycontroller`), attach labels that identify semantic attributes, such as `{ app: myapp, tier: frontend, phase: test, deployment: v3 }`. This will let you select the object groups appropriate to the context— e.g., a service for all “tier: frontend” pods, or all “test” phase components of app “myapp”. See the `guestbook` app for an example of this approach.

A service can be made to span multiple deployments, such as is done across `rolling updates`, by simply omitting release-specific labels from its selector, rather than updating a service’s selector to match the replication controller’s selector fully.

- 为了滚动升级的方便，在Replication Controller的名字中包含版本信息，例如作为名字的后缀。设置一个 `version` 标签页是很有用的。滚动更新创建一个新的controller而不是修改现有的controller。因此，`version`含混不清的controller名字就可能带来问题。查看`Rolling Update Replication Controller`文档获取更多关于滚动升级命令的信息。

注意 `Deployment` 对象不需要再管理 `replication controller` 的版本名。`Deployment` 中描述了对对象的期望状态，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。（`Deployment`目前是 `extensions`

API Group的一部分)。

- 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中的label的方式将其从一个controller或者service中移除，原来的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。查看 `kubectl label` 命令。

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。

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到底运行的是哪个版本的容器和回滚。

Using kubectl

- 尽量使用 `kubectl create -f <directory>`。`kubectl`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` 命令。
- 使用 `kubectl delete` 而不是 `stop`。`Delete` 是 `stop` 的超集，`stop` 已经被弃用。
- 使用 `kubectl bulk` 操作（通过文件或者label）来get和delete。查看 [label selectors](#) 和 [using labels effectively](#)。
- 使用 `kubectl run` 和 `expose` 命令快速创建只有单个容器的Deployment。查看 [quick start guide](#)中的示例。

Kubernetes插件

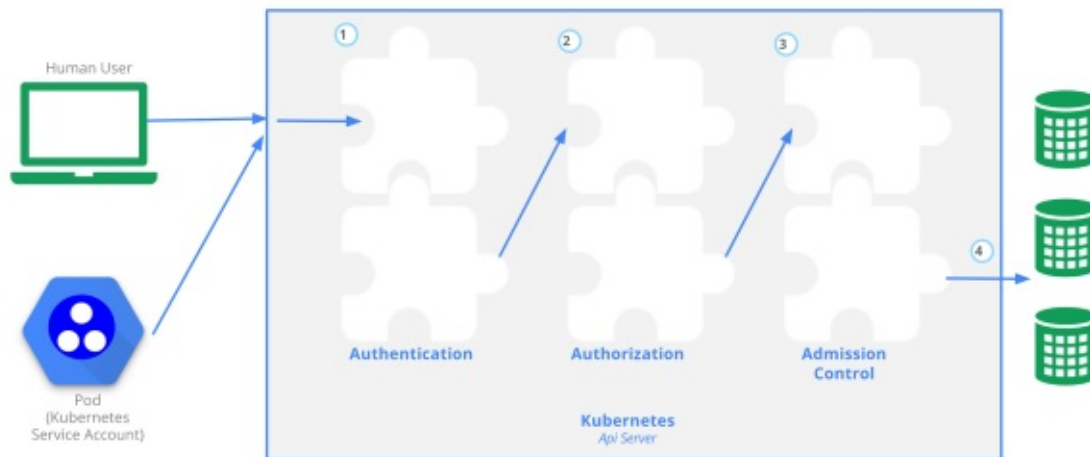
Kubernetes在设计之初就充分考虑了可扩展性，很多资源或操作都可以通过插件来自由扩展，比如认证授权、网络、Volume、容器执行引擎、调度等。

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。

[warning] Kubernetes不管理用户

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=jbeda/O=app1/O=app2"
```

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 Authrization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，需要在请求头中加入Basic Authorization头，如

```
Basic BASE64ENCODED(USER:PASSWORD)
```

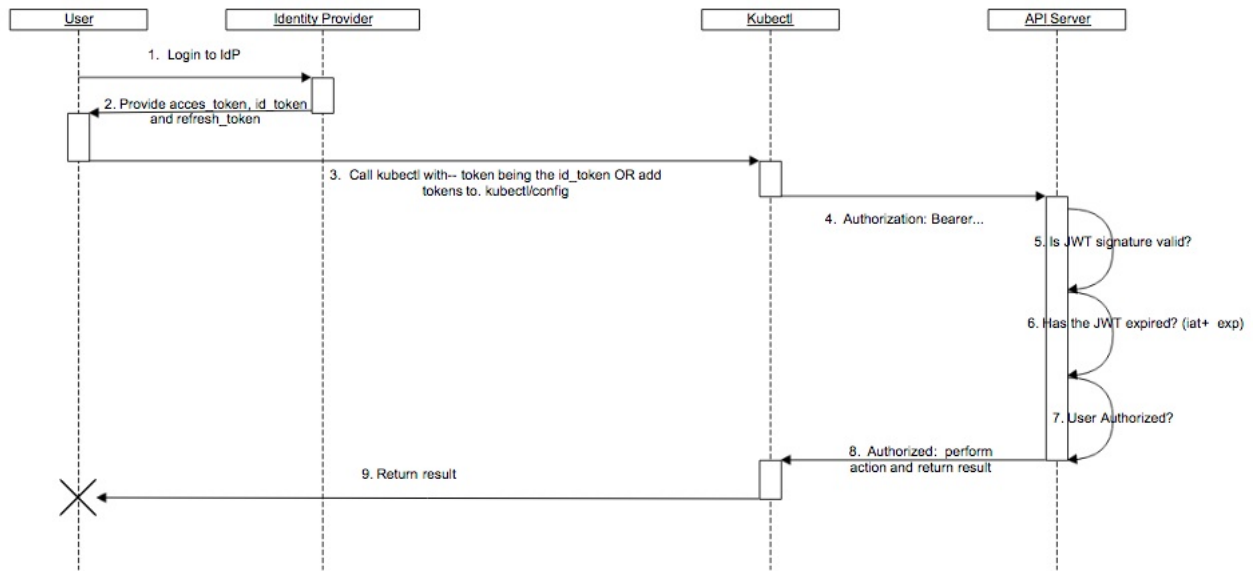
Service Account

ServiceAccount是Kubernetes自动生成的，并会自动挂载到容器的 `/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:
      certificate-authority: /path/to/ca.pem          # CA for verifying the remote service.
      server: https://authn.example.com/authenticate # URL of remote service to query. Must use 'https'.

# 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 webhook plugin to use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API server.
current-context: webhook
contexts:
  - context:
      cluster: name-of-remote-authn-service
      user: name-of-api-server
      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=<AuthURL>`，而https时还需要设置 `--experimental-keystone-ca-file=SOMEFILE`。

[warning] 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`。

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。

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

[info] 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:
      certificate-authority: /path/to/ca.pem      # CA for verifying
the remote service.
      server: https://authz.example.com/authorize # URL of remote ser
vice to query. Must use 'https'.

# 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 webhook pl
ugin to use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API Server.
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:<nodeName>`。

参考文档

- [Authenticating](#)
- [Authorization](#)
- [Bootstrap Tokens](#)
- [Managing Service Accounts](#)

RBAC

在Kubernetes1.6版本中新增角色访问控制机制（Role-Based Access，RBAC）让集群管理员可以针对特定使用者或服务账号的角色，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中的一个亮点是RBAC访问控制机制升级到了beta版本。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。

本文主要关注新特性和最佳实践。

RBAC vs ABAC

目前kubernetes中已经有一系列鉴权机制。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。

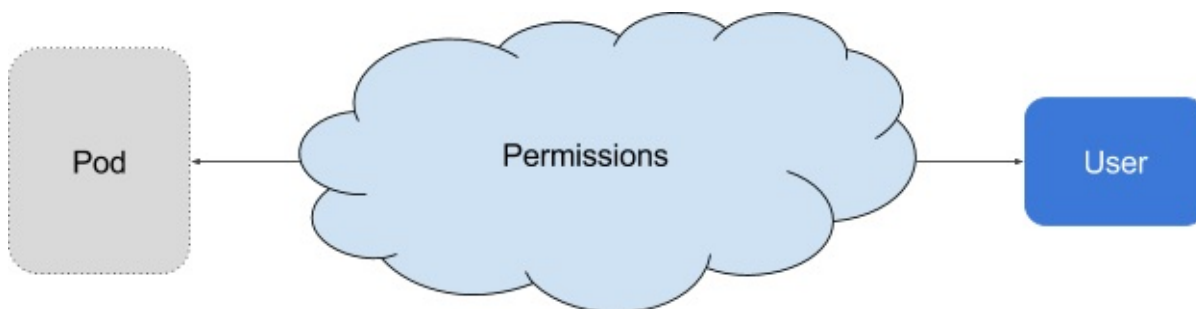
ABAC（Attribute Based Access Control）本来是不错的概念，但是在 Kubernetes 中的实现比较难于管理和理解，而且需要对 Master 所在节点的 SSH 和文件系统权限，而且要使得对授权的变更成功生效，还需要重新启动 API Server。

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。



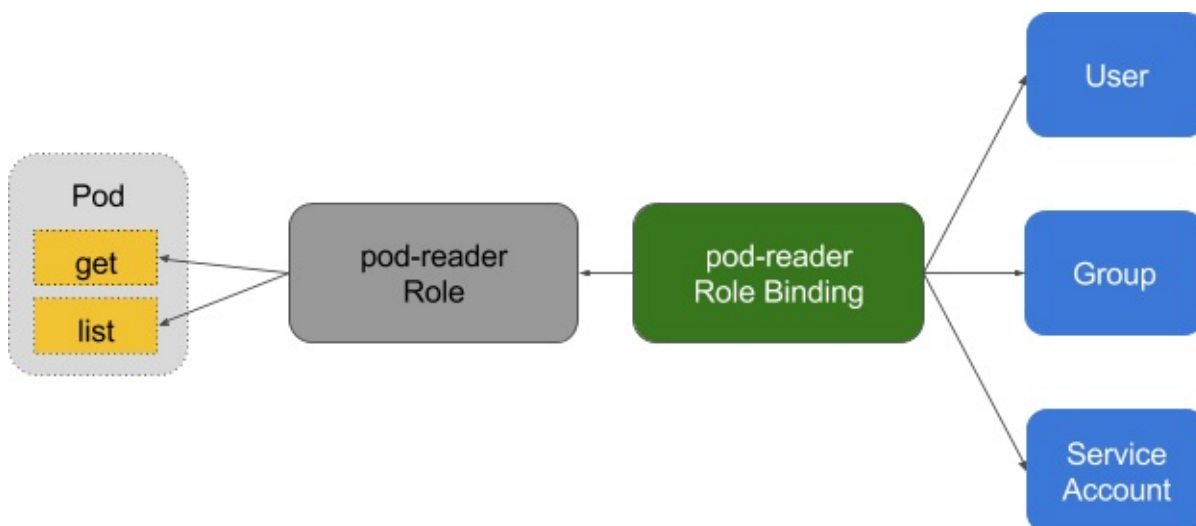
在 RBAC 中定义了两个对象，用于描述在用户和资源之间的连接权限。

role

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，ClusterRole 跟 Role 类似，但是可以在集群中到处使用（Role 是 namespace 一级的）。

role binding

RoleBinding 把角色映射到用户，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。



另外还要考虑 cluster roles 和 cluster role binding。cluster role 和 cluster role binding 方法跟 role 和 role binding 一样，出了它们有更广的 scope。详细差别请访问 [role binding 与 cluster role binding](#)。

Kubernetes 中的 RBAC

RBAC 现在被 Kubernetes 深度集成，并使用他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
```


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。

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 的负载。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].metadata.name}') bash
$ apt-get update && apt-get install -y curl
$ curl -ik \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
  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 已经足够了，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。

参考文档

说明：本文翻译自 [RBAC Support in Kubernetes](#)，转载自 [kubernetes 中文社区](#)，译者催总，[Jimmy Song](#) 做了稍许修改。该文章是 5 天内了解 Kubernetes 1.6 新特性的系列文章之一。

- [RBAC documentation](#)
- [Google Cloud Next talks 1](#)
- [Google Cloud Next talks 2](#)
- [在 Kubernetes Pod 中使用 Service Account 访问 API Server](#)

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+）

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，并将完成后将任务从 `metadata.initializers` 列表中删除。

Initializers的开启方法为

- kube-apiserver配置 `--admission-control=...,Initializer`
- 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
spec:
  initializers:
    # the name needs to be fully qualified, i.e., containing at least
    # two "."
    - name: podimage.example.com
      rules:
        # apiGroups, apiVersion, resources all support wildcard "*".
        # "*" cannot be mixed with non-wildcard.
        - apiGroups:
            - "*"
          apiVersions:
            - v1
          resources:
            - pods
```

注意，如果不需要修改对象的话，建议使用性能更好的
`GenericAdmissionWebhook`。

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 controller
    # Other options: Ignore
    failurePolicy: Fail
    clientConfig:
      caBundle: <pem encoded ca cert that signs the server cert used by the webhook>
      service:
        name: <name of the front-end service>
        namespace: <namespace of the front-end service>
```

推荐配置

对于Kubernetes >= 1.6.0，推荐kube-apiserver开启以下插件

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,Per  
sistentVolumeLabel,DefaultStorageClass,ResourceQuota,DefaultToleratio  
nSeconds
```

Kubernetes 网络

介绍Kubernetes网络模型以及常见插件。

Kubernetes网络

Kubernetes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

- 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](#)~~

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

CNI plugin

安装CNI：

```
cat <<EOF > /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.gpg
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/Documentation/kube-flannel-rbac.yml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yml
```

```
## [Weave Net](weave/index.md)
```

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kernel 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。

```
```sh
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。

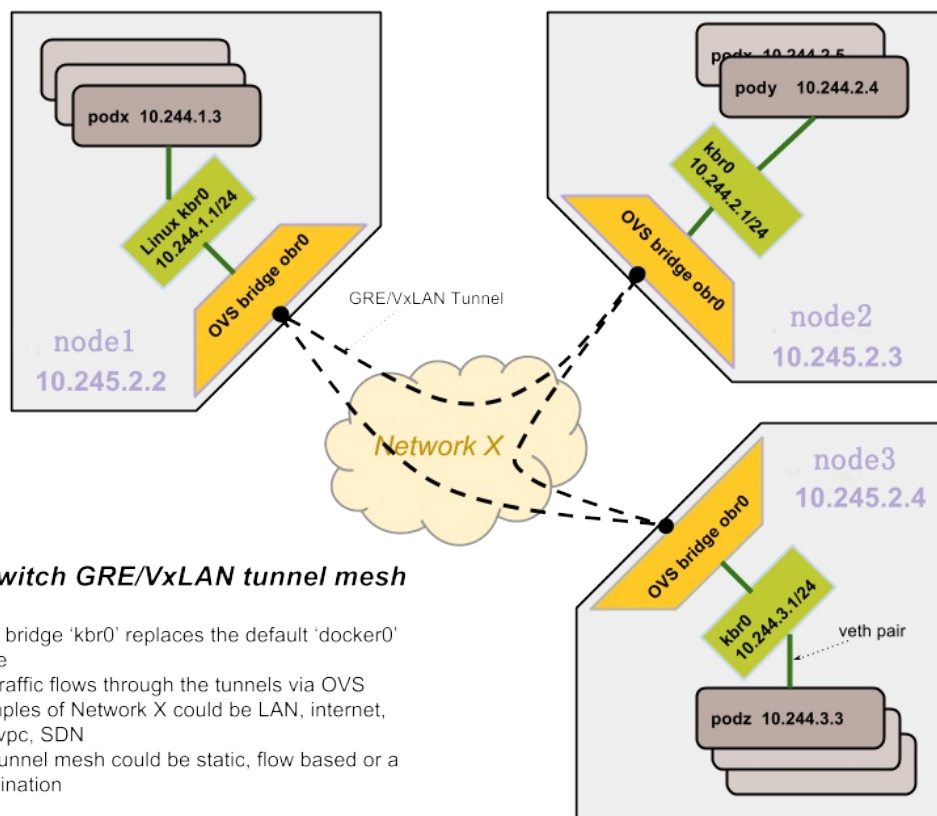
此外，Calico基于iptables还提供了丰富而灵活的网络Policy，保证通过各个节点上的ACLs来提供Workload的多租户隔离、安全组以及其他可达性限制等功能。

```
kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernet
es/installation/hosted/kubeadm/1.6/calico.yaml
```

## 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：

- **Overlay**: 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。

## 其他

### ipvs

目前社区还在推进<https://github.com/kubernetes/kubernetes/issues/17470>，预计v1.7可以有alpha版进来。

### 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

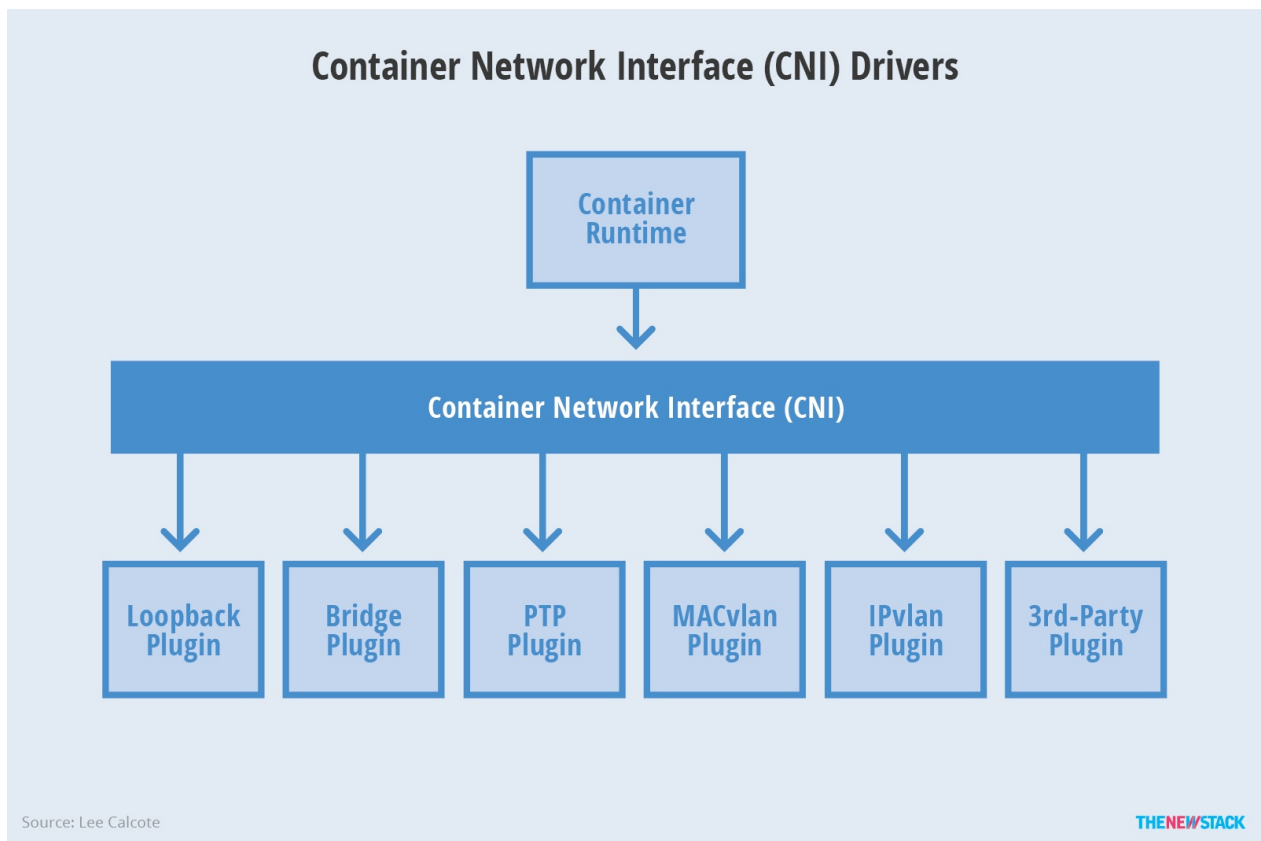
项目主页为<https://github.com/kopeio/kope-routing>。

## CNI (Container Network Interface)

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。



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", "bridge": "cni0", "isGateway": true, "ipMasq": true, "ipam": {"type": "host-local", "subnet": "10.244.1.0/24", "routes": [{"dst": "0.0.0.0/0"}]}}' | sudo CNI_COMMAND=ADD CNI_NETNS=/var/run/netns/a CNI_PATH=./bin CNI_IFNAME=eth0 CNI_CONTAINERID=a CNI_VERSION=0.3.1 ./bin/bridge

$ echo '{"cniVersion": "0.3.1", "type": "IGNORED", "name": "a", "ipam": {"type": "host-local", "subnet": "10.1.2.3/24"}}' | sudo CNI_COMMAND=ADD CNI_NETNS=/var/run/netns/a CNI_PATH=./bin CNI_IFNAME=a CNI_CONTAINERID=a CNI_VERSION=0.3.1 ./bin/host-local
```

## 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。可以借助[Flannel](#)或者Quagga动态路由等来自动配置。

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: <LOOPBACK> 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: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
 link/ether 0a:58:0a:0a:00:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
 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 dadfailed
 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连接的容器之间通信。

```
{
 "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，所有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 <master-dev> <slave-dev> 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 <master-dev> 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kernel 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

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。

## 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portmapping、vm等非常有帮助。

## 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>。

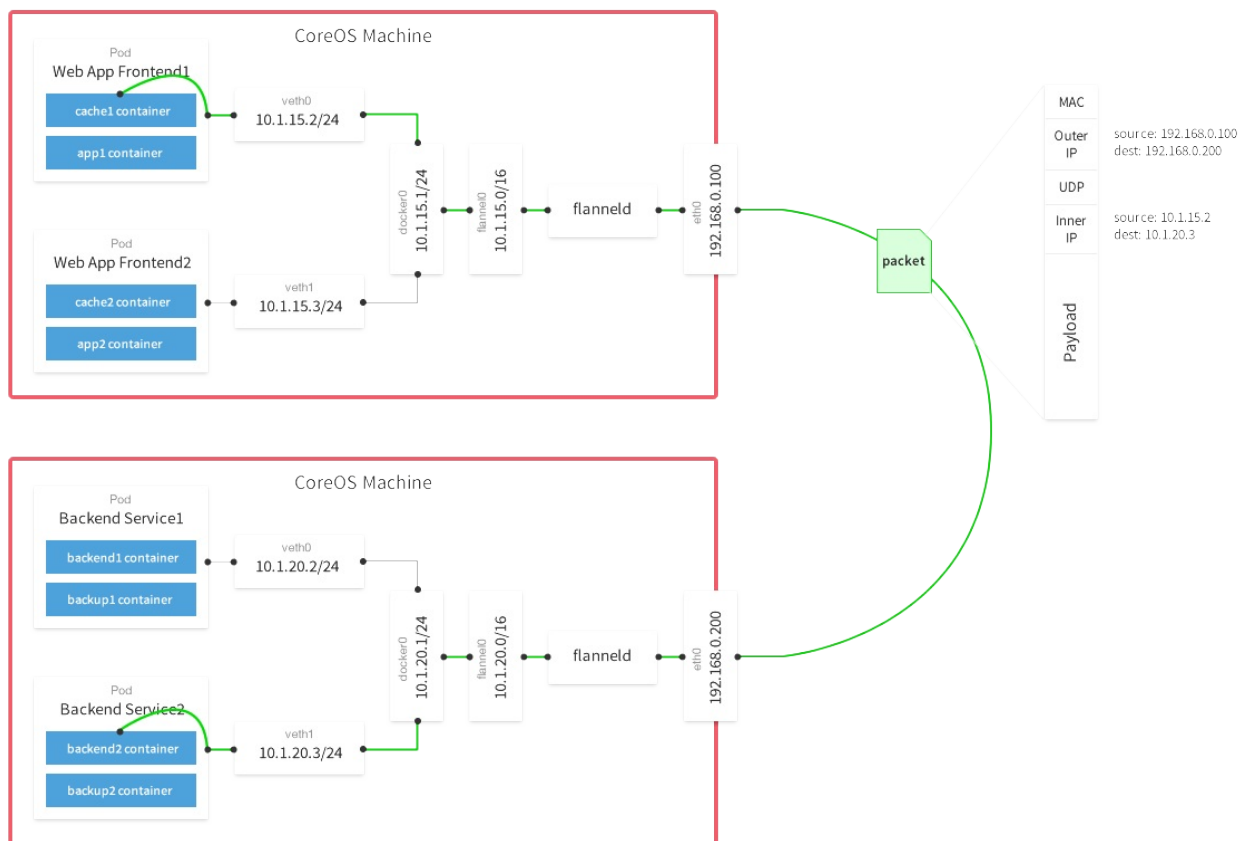
# Flannel

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udp封装，默认使用8285端口
- vxlan：vxlan封装，需要配置VNI，Port（默认8472）和GBP
- host-gw：直接路由的方式
- 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

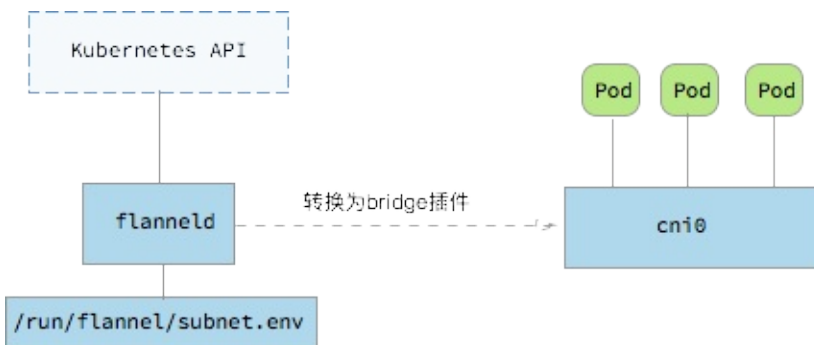
使用flannel前需要配置 `kube-controller-manager --allocate-node-cidrs=true -`  
`-cluster-cidr=10.244.0.0/16`。

```
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel-rbac.yml
kubectl create -f https://github.com/coreos/flannel/raw/master/Documentation/kube-flannel.yml
```

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 --i
p-masq --kube-subnet-mgr
root 9640 9619 0 13:51 ? 00:00:00 /bin/sh -c set -e -x;
cp -f /etc/kube-flannel/cni-conf.json /etc/cni/net.d/10-flannel.conf
; while true; do sleep 3600; done

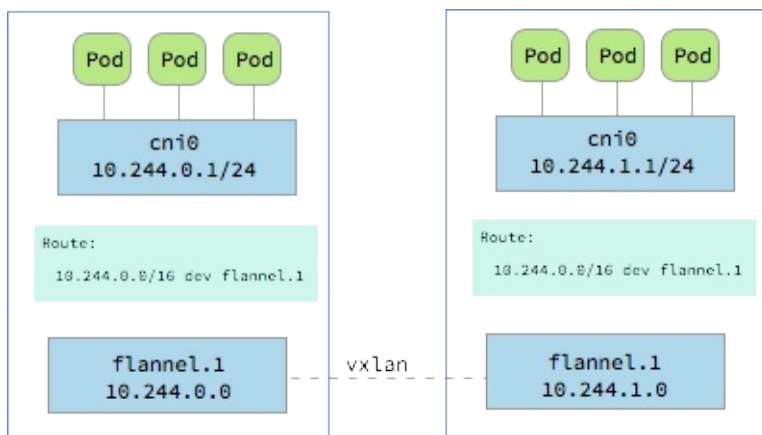
$ cat /etc/cni/net.d/10-flannel.conf
{
 "name": "cbr0",
 "type": "flannel",
 "delegate": {
 "isDefaultGateway": true
 }
}
```



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: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noque
ue state UNKNOWN mode DEFAULT group default
 link/ether 8e:5a:0d:07:0f:0d brd ff:ff:ff:ff:ff:ff promiscuity 0
 vxlan id 1 local 10.146.0.2 dev ens4 srcport 0 0 dstport 8472 nol
earning ageing 300 udpcsum addrgenmode eui64
```



## 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/>

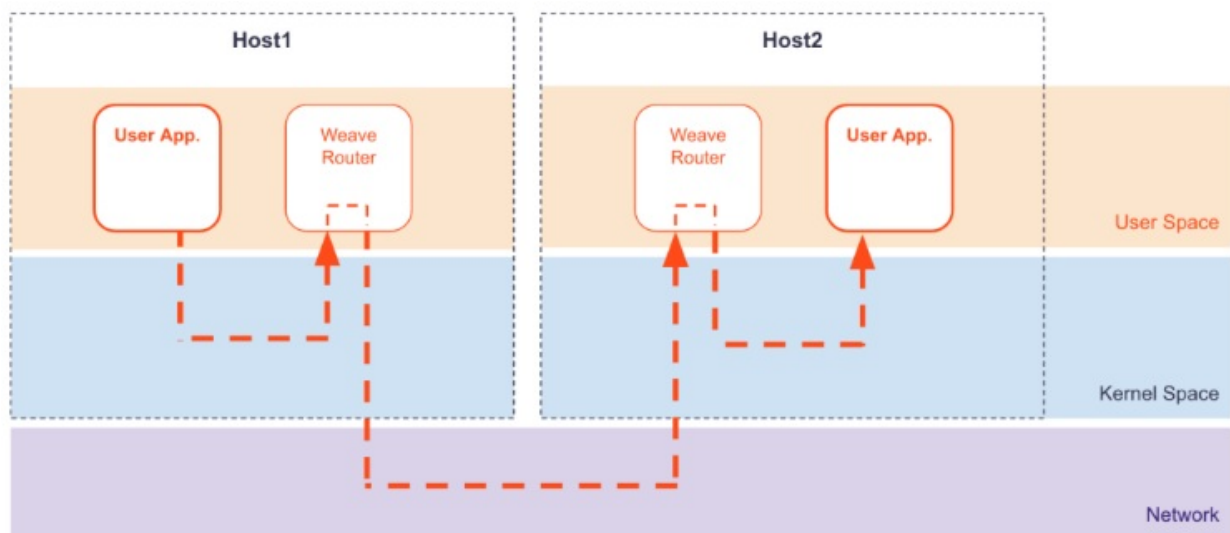
# Weave Net

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：

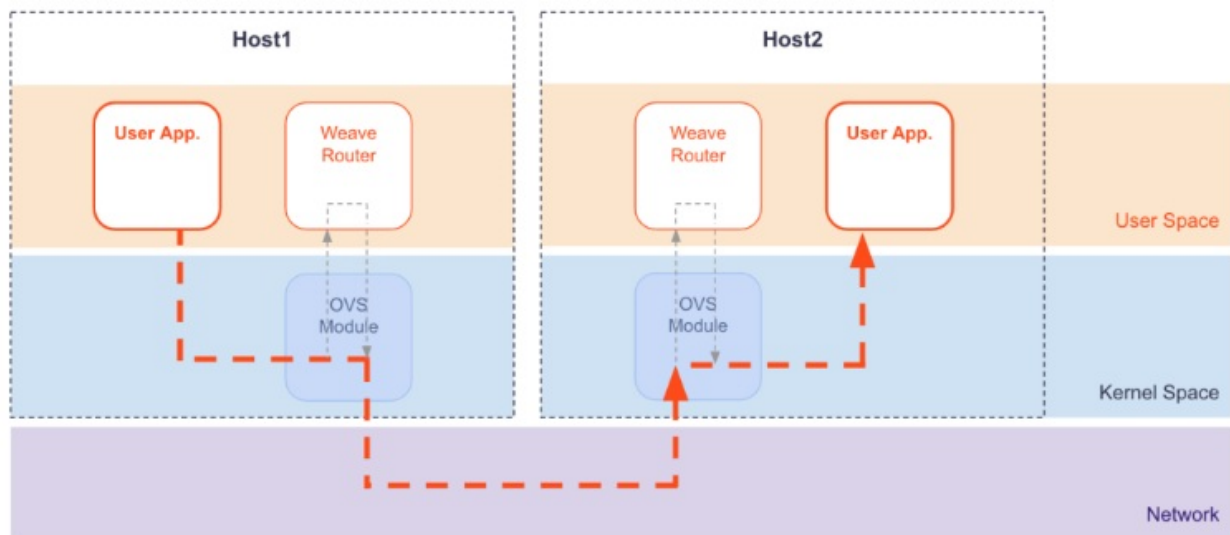
- 运行在user space的sleeve mode：通过pcap设备在Linux bridge上截获数据包并由wRouter完成UDP封装，支持对L2 traffic进行加密，还支持Partial Connection，但是性能损失明显。
- 运行在kernel space的 fastpath mode：即通过OVS的odp封装VxLAN并完成转发，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发，这种方式可以明显地提升吞吐量，但是不支持加密等高级功能。

Sleeve Mode:

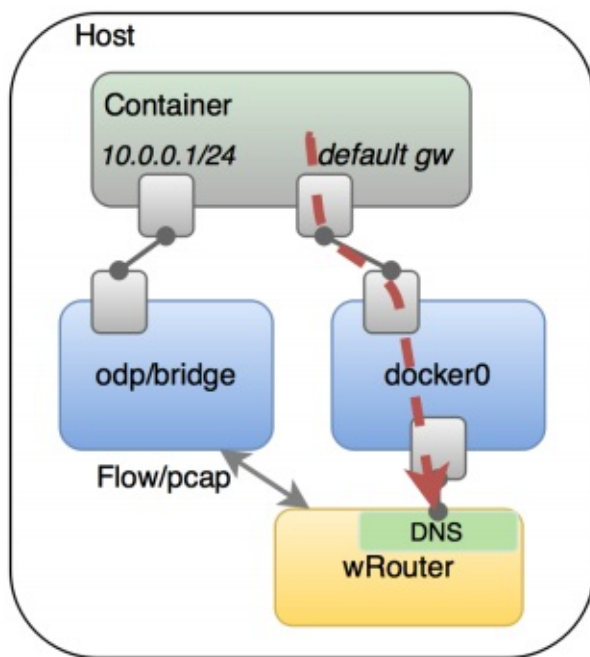


Fastpath Mode:





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

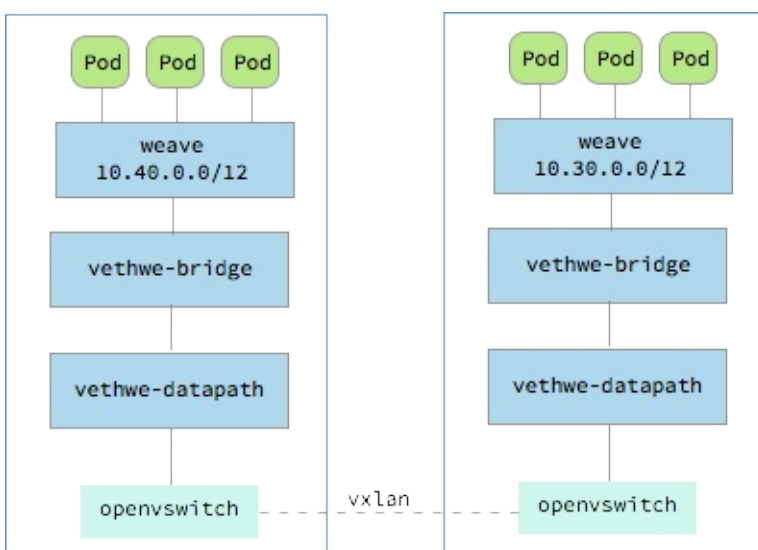
```
kubectl apply -n kube-system -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

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/weave/launch.sh
root 25204 25147 0 16:22 ? 00:00:00 /home/weave/weaver --port=6783 --datapath=datapath --host-root=/host --http-addr=127.0.0.1:6784 --status-addr=0.0.0.0:6782 --docker-api= --no-dns --db-prefix=/weavedb/weave-net --ipalloc-range=10.32.0.0/12 --nickname=ubuntu-0 --ipalloc-init consensus=2 --conn-limit=30 --expect-npc 10.146.0.2 10.146.0.3
root 25669 25654 0 16:22 ? 00:00:00 /usr/bin/weave-npc
```

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

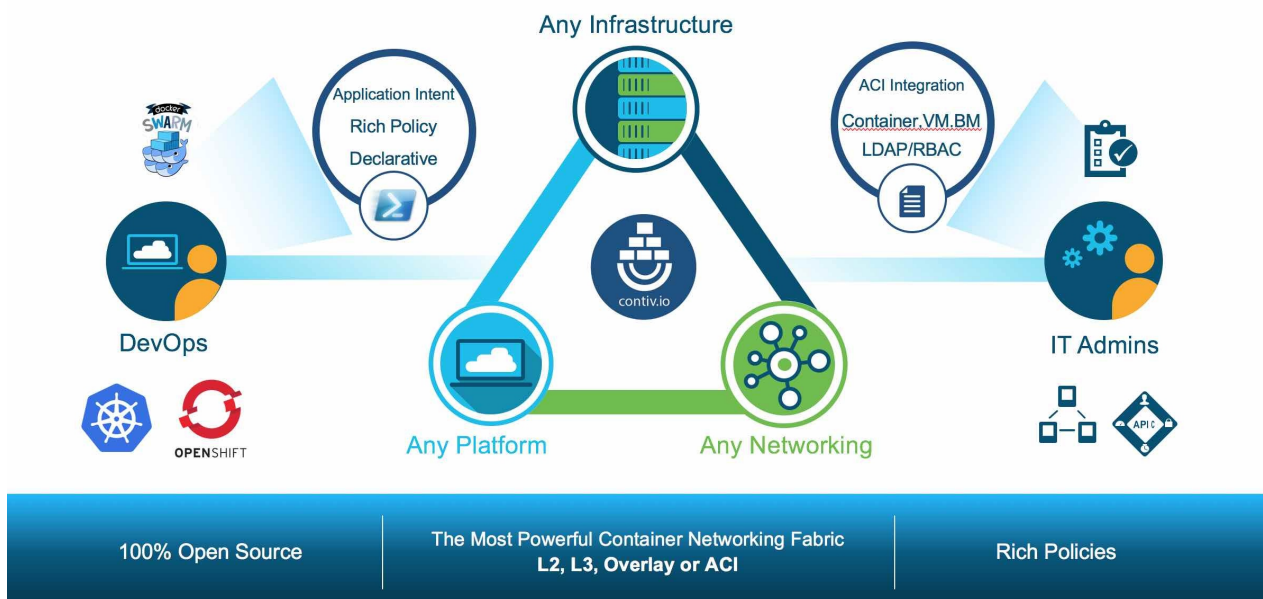
- UDP模式性能损失较大

### 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/>

## Contiv

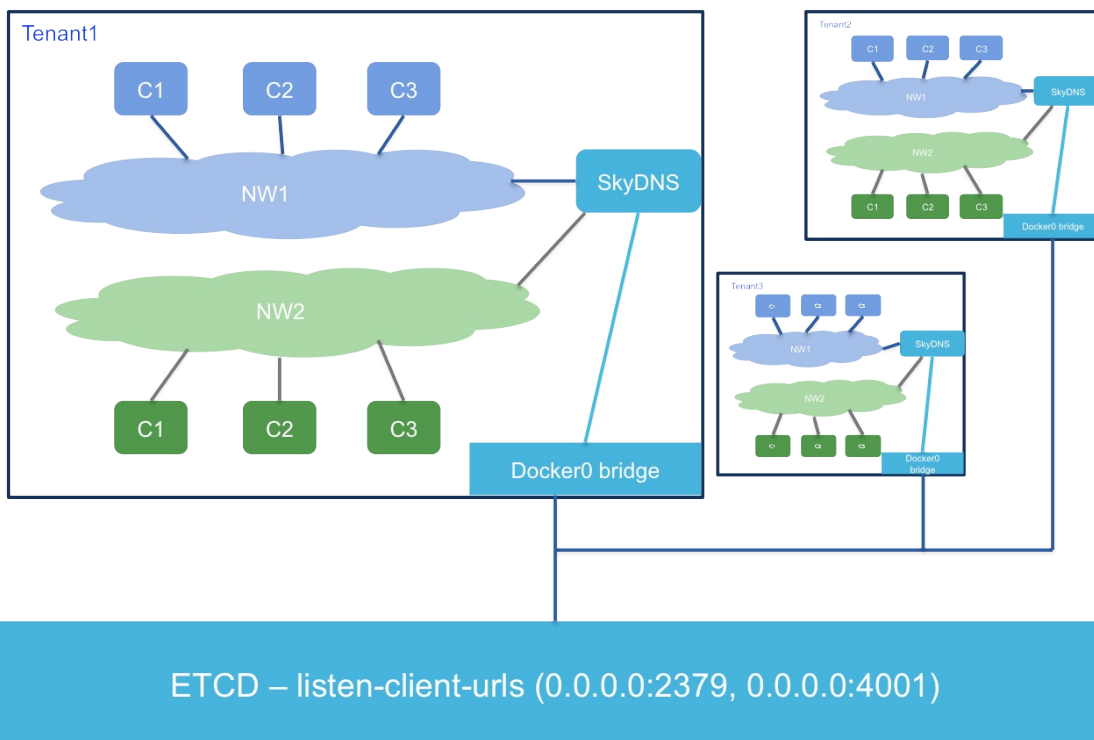
**Contiv**是思科开源的容器网络方案，是一个用于跨虚拟机、裸机、公有云或私有云的异构容器部署的开源容器网络架构，并与主流容器编排系统集成。**Contiv**最主要的优势是直接提供了多租户网络，并支持L2(VLAN), L3(BGP), Overlay (VXLAN)以及思科自家的ACI。



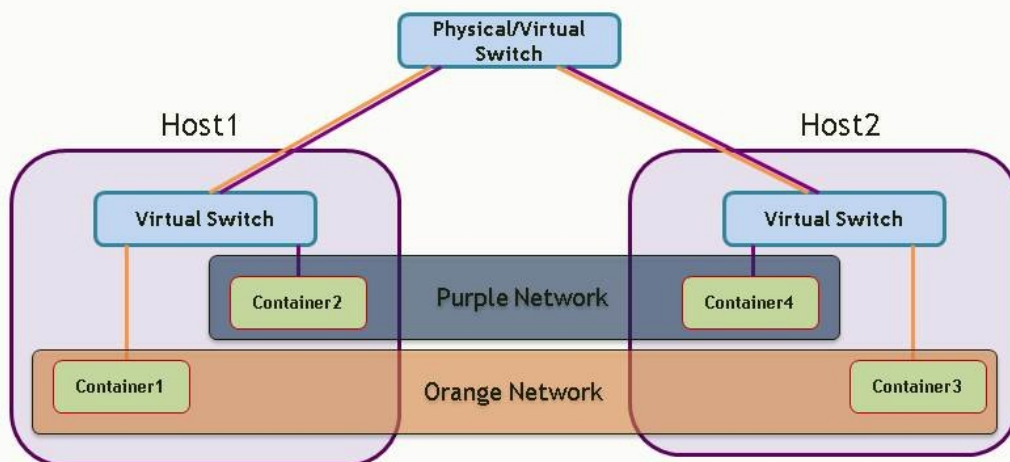
### 主要特征

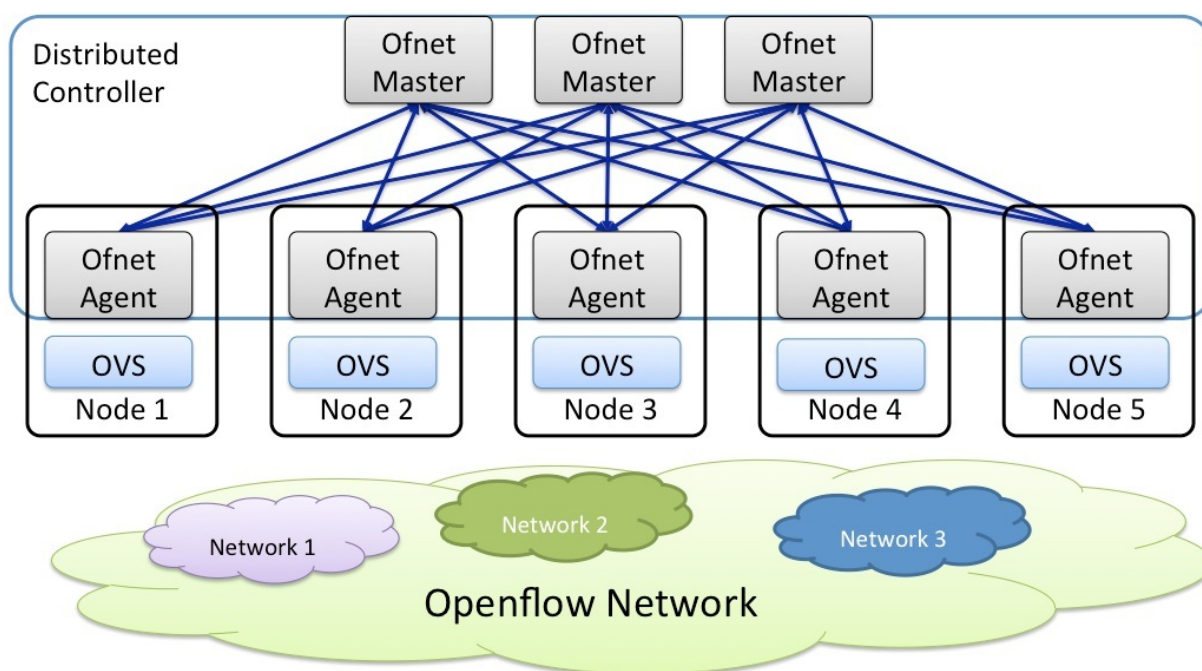
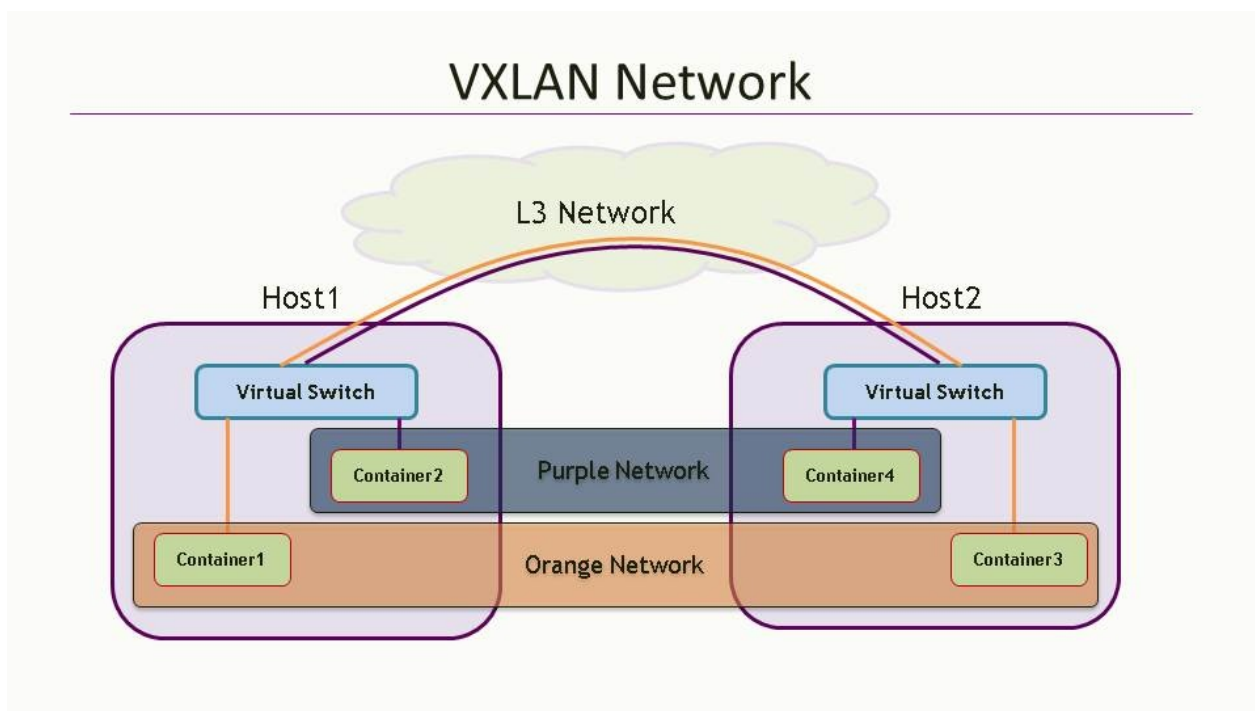
- 原生的Tenant支持，一个Tenant就是一个virtual routing and forwarding (VRF)
- 两种网络模式
  - L2 VLAN Bridged
  - Routed network, e.g. vxlan, BGP, ACI
- Network Policy，如Bandwidth, Isolation等

## Host



## VLAN Network





## 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/$VERSION/contiv-$VERSION.tgz
tar xf contiv-$VERSION.tgz
cd ~/contiv/contiv-$VERSION/install/k8s
netctl --netmaster http://$netmaster:9999 global set --fwd-mode routing

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=10.100.100.215-10.100.100.220/27 --gateway=10.100.100.193 vlan3280
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="65000" --neighbor-as="65000" --neighbor="30.30.30.2"
netctl bgp create devstack-78 --router-ip="30.30.30.78/24" --as="65000" --neighbor-as="65000" --neighbor="30.30.30.2"
netctl bgp create devstack-71 --router-ip="30.30.30.79/24" --as="65000" --neighbor-as="65000" --neighbor="30.30.30.2"

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](#)



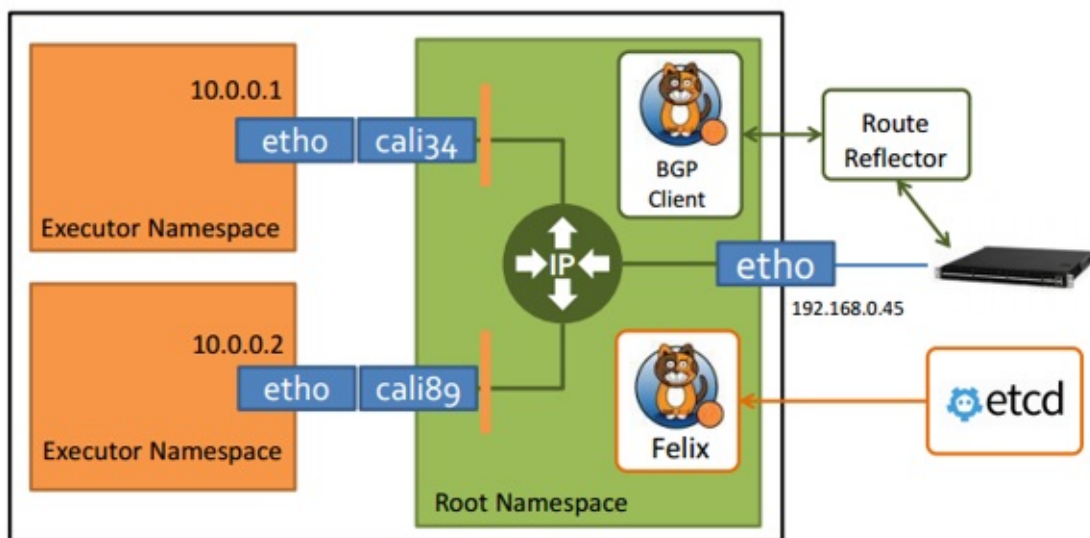
## Calico

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

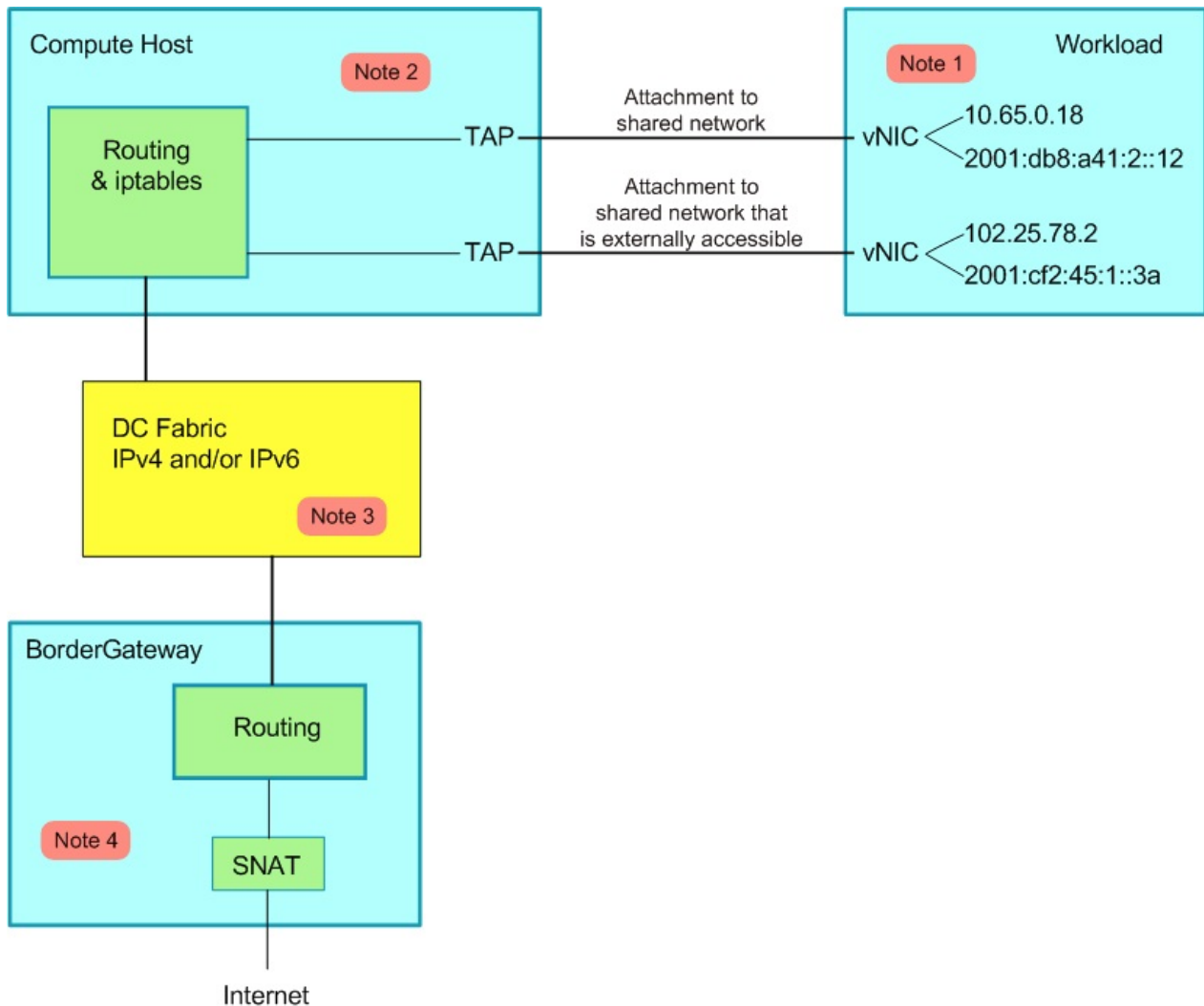


Calico主要由Felix、etcd、BGP client以及BGP Route Reflector组成

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。



## 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<http://docs.projectcalico.org/v2.1/getting-started/docker/>。

## Calico Kubernetes

见<http://docs.projectcalico.org/v2.1/getting-started/kubernetes/>。

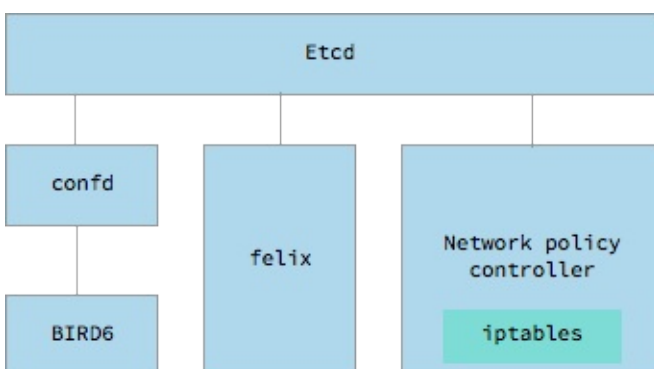
For Kubeadm 1.5 with Kubernetes 1.5.x:

```
kubectl apply -f http://docs.projectcalico.org/v2.3/getting-started/kubernet
es/installation/hosted/kubeadm/1.5/calico.yaml
```

For Kubeadm 1.6 with Kubernetes 1.6.x:

```
kubectl apply -f http://docs.projectcalico.org/v2.3/getting-started/kubernet
es/installation/hosted/kubeadm/1.6/calico.yaml
```

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/local
/bin/etcd --name=calico --data-dir=/var/etcd/calico-data --advertise-
client-urls=http://$CALICO_ETCD_IP:6666 --listen-client-urls=http://0
.0.0.0:6666 --listen-peer-urls=http://0.0.0.0:6667
```

```
root 9038 9012 0 14:51 ? 00:00:01 /usr/local/bin/etcd -
-name=calico --data-dir=/var/etcd/calico-data --advertise-client-urls
=http://10.146.0.2:6666 --listen-client-urls=http://0.0.0.0:6666 --li
sten-peer-urls=http://0.0.0.0:6667
```

```
root 9326 9325 0 14:51 ? 00:00:00 bird6 -R -s /var/run/
calico/bird6.ctl -d -c /etc/calico/confd/config/bird6.cfg
```

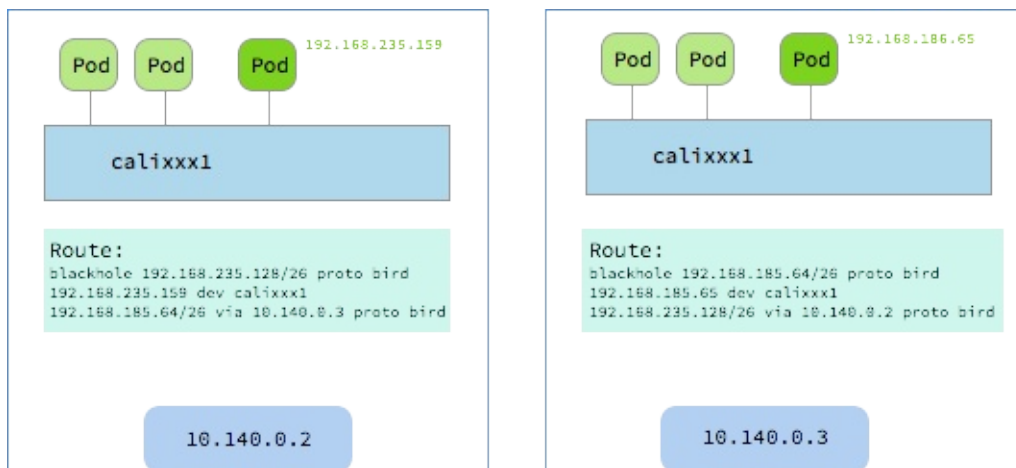
```
root 9327 9322 0 14:51 ? 00:00:00 confd -confdir=/etc/c
alico/confd -interval=5 -watch --log-level=debug -node=http://10.96.2
32.136:6666 -client-key= -client-cert= -client-ca-keys=
```

```
root 9328 9324 0 14:51 ? 00:00:00 bird -R -s /var/run/c
alico/bird.ctl -d -c /etc/calico/confd/config/bird.cfg
```

```
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": "<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

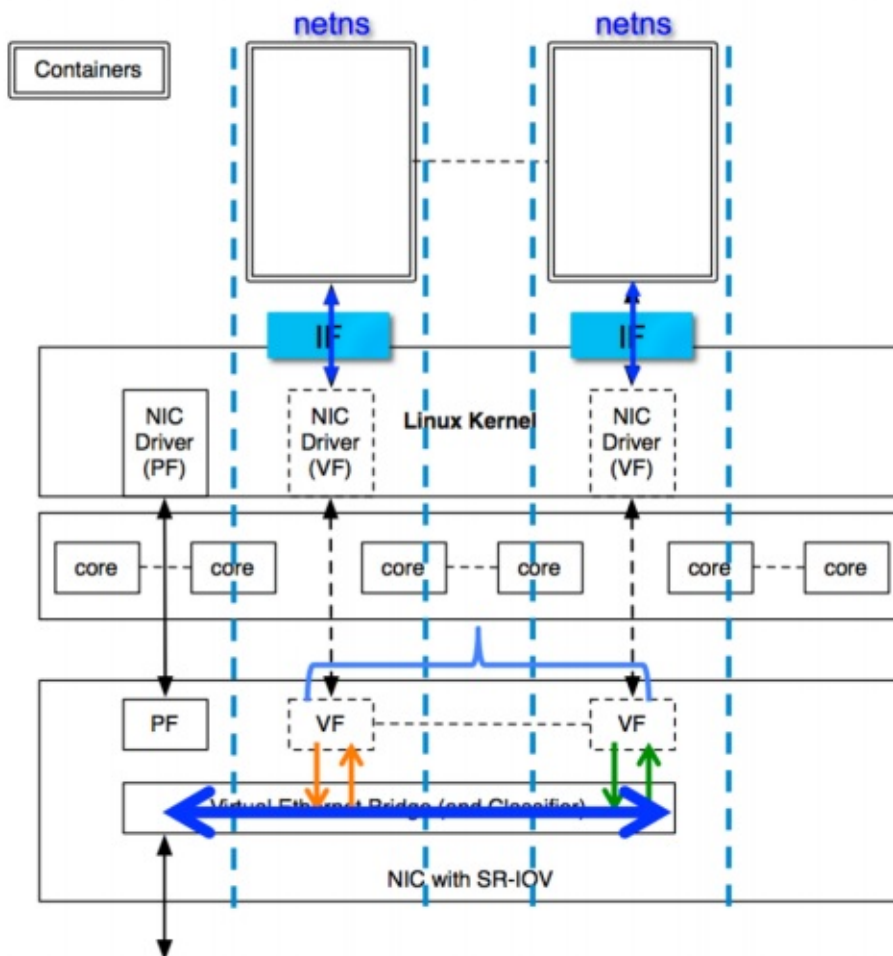
- <https://xuxinkun.github.io/2016/07/22/cni-cnm/>
- <https://www.projectcalico.org/>
- <http://blog.dataman-inc.com/shurenyun-docker-133/>

# 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 created, starting with "eth", e.g. eth4
```

## 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](https://github.com/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

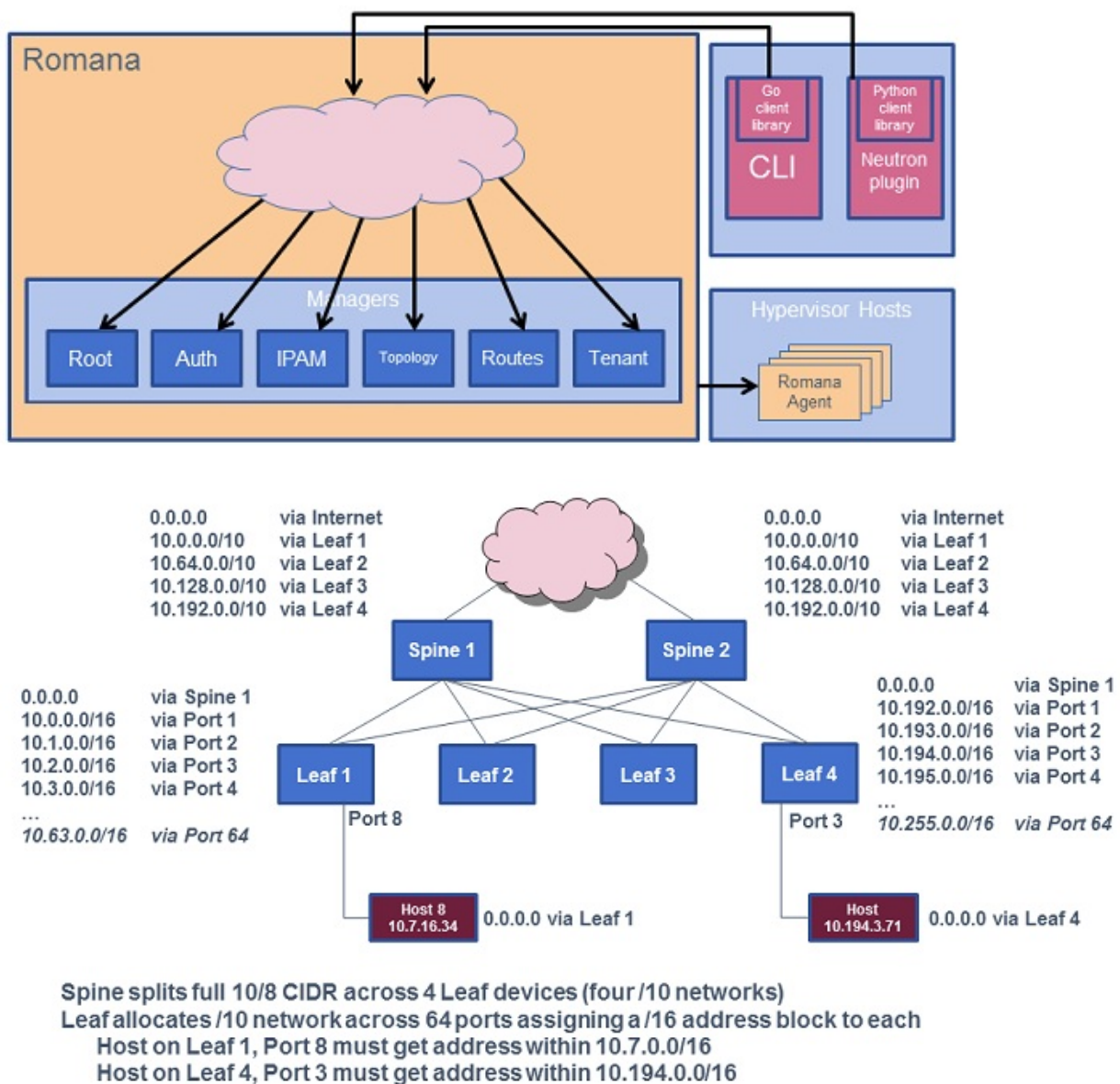
### 参考文档

- <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。

## 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



Bit location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Field	10/8 Net Mask								Host ID Bits (8)								Tenant ID Bits (8)								Segment ID and IID							
Capacity	0	0	0	0	1	0	1	0	Up to 255 Hosts								Up to 255 Tenants								255 Endpoints for each Tenant							

Example:

	Bits	Length	Location	Purpose
10/8 Network	8	8	1-8	10/8 Network
Hosts	8	16	9-16	Up to 255 Hosts
Tenants	8	24	17-24	Up to 255 Tenants
Segments	4	28	25-28	Up to 16 Segments per Tenant
Endpoints	4	32	29-32	Up to 16 Endpoints per Segment

Host 1	ID	CIDR or IP
Physical Addr		192.168.0.10
Host	1	10.1/16
Tenant	1	10.1.1/24
Segment	1	10.1.1.16/28
Pod 1	11	10.1.1.27
Pod 2	14	10.1.1.40
Tenant	2	10.1.2/24
Segment	1	10.1.2.16/28
Pod 1	4	10.1.2.20
Pod 2	8	10.1.2.24

Host 2	ID	CIDR or IP
Physical Addr		192.168.0.11
Host	2	10.2/16
Tenant	1	10.2.1/24
Segment	1	10.2.1.16/28
Pod 1	4	10.2.1.20
Pod 2	5	10.2.1.21
Tenant	1	10.2.1/24
Segment	2	10.2.1.32/28
Pod 1	9	10.2.1.41
Pod 2	12	10.2.1.44

Host 3	ID	CIDR or IP
Physical Addr		192.168.0.12
Host	3	10.3/16
Tenant	1	10.3.1/24
Segment	1	10.3.1.16/28
Pod 1	4	10.3.1.20
Pod 2	5	10.3.1.21
Tenant	2	10.3.2/24
Segment	1	10.3.2.32/28
Pod 1	9	10.3.2.25
Pod 2	12	10.3.2.28

## 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/>
- <https://github.com/romana/romana>
- <http://romana.io/how/background/>

# 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

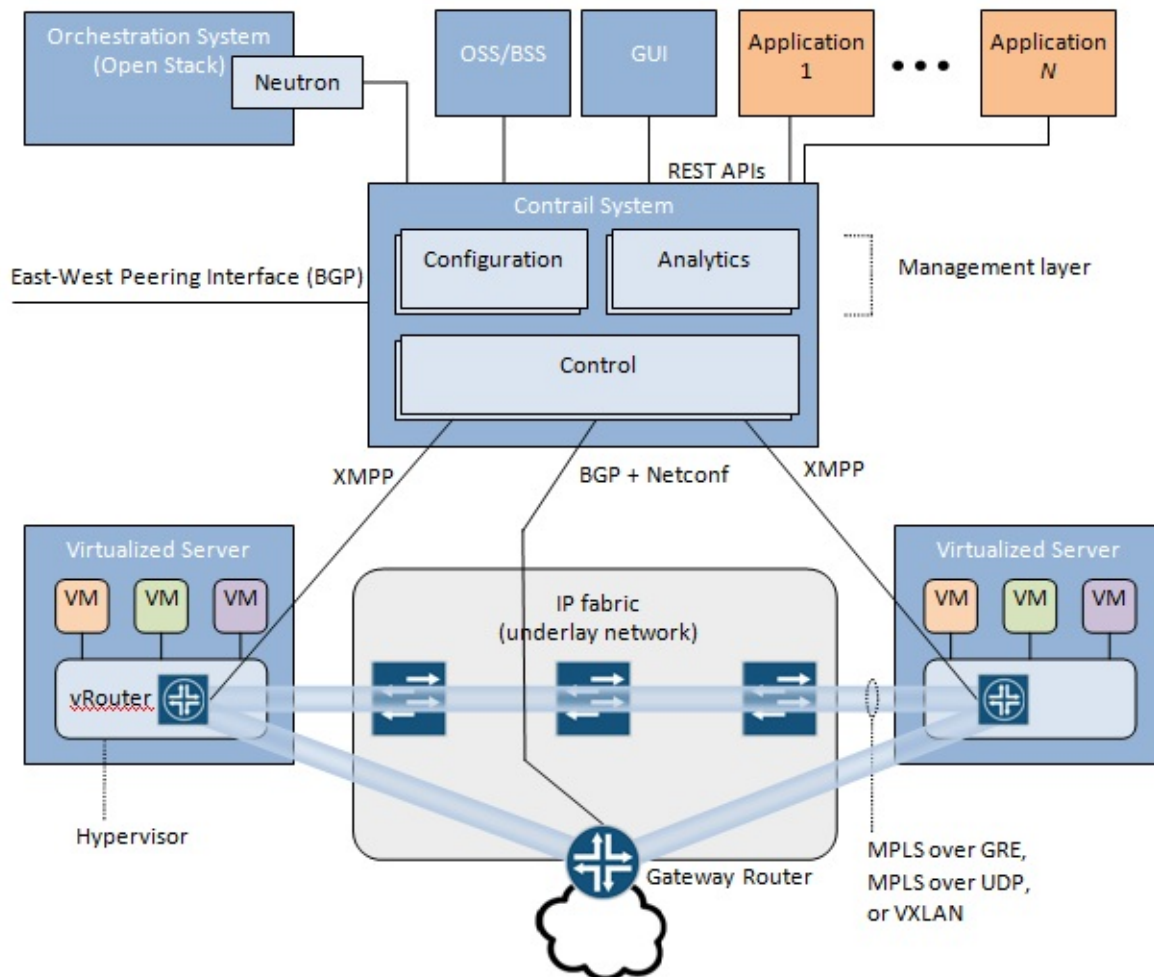
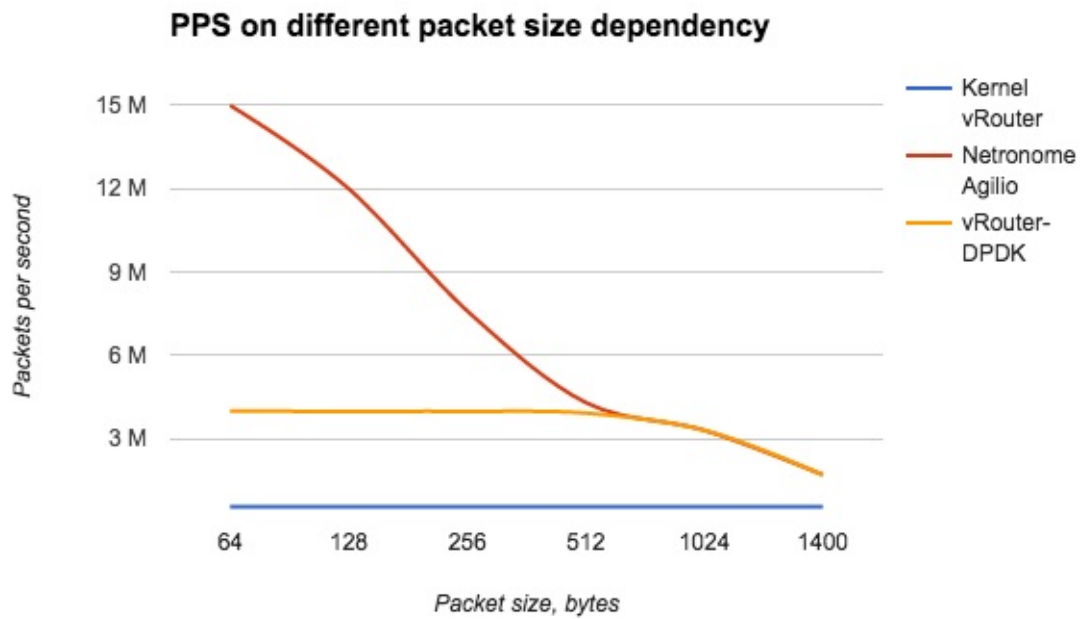


Figure 1: OpenContrail System Overview

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/>

## 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portmapping、vm等非常有帮助。

## 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

### 示例

下面的例子展示了bridge+portmap插件的用法。

```
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}
 }
]
}
```

```
export CAP_ARGS='{
 "portMappings": [
 {
 "hostPort": 9090,
 "containerPort": 80,
 "protocol": "tcp",
 "hostIP": "127.0.0.1"
 }
]
}'

ip netns add test
CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cni tool add mynet /var/run/netns/test
{
```

```
"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": {}
}
```

```
iptables-save | grep 10.244.10.7
-A CNI-DN-be1eedf7a76853f303ebd -d 127.0.0.1/32 -p tcp -m tcp --dport
9090 -j DNAT --to-destination 10.244.10.7:80
-A CNI-SN-be1eedf7a76853f303ebd -s 127.0.0.1/32 -d 10.244.10.7/32 -p
tcp -m tcp --dport 80 -j MASQUERADE
```

```
CNI_PATH=/opt/cni/bin NETCONFPATH=/root ./cnitool del mynet /var/ru
n/netns/test
```

## Volume插件扩展

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实现自己的Volume插件。

## FlexVolume

实现一个FlexVolume包括两个步骤

- 实现FlexVolume插件接口，包括 `init/attach/detach/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/<vendor~driver>/<driver>` 目录中

而在使用flexVolume时，需要指定卷的driver，格式为 `<vendor~driver>/<driver>`，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"
```



## 使用glusterfs做持久化存储

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 glusterfs-rdma glusterfs-geo-replication glusterfs-devel

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的模式有很多中，包括以下几种：

- 分布卷（默认模式）：即DHT, 也叫 分布卷: 将文件已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：[CentOS7安装GlusterFS](#)。

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-test-001.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data force

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-size
1024MB
```

## Kubernetes 中配置glusterfs

官方的文档

见：<https://github.com/kubernetes/kubernetes/tree/master/examples/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，因此这一步可以省去。

## 配置 endpoints

```
$ curl -O https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-endpoints.json
```

```
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/master/examples/volumes/glusterfs/glusterfs-service.json
```

```
service.json 里面查找的是 endpoints 的名称与端口，端口默认配置为 1，我改成了1990
```

```
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://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-pod.json
```

```
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	AGE
glusterfs	1/1	Running	0	1m

```
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 1073741824 0 1073741824 0% 172.20.0.
113:k8s-volume 1.0T 0 1.0T 0% /var/lib/kubelet/pods/3de9fc69-
30b7-11e7-bfbd-8af1e3a7c5bd/volumes/kubernetes.io~glusterfs/glusterfs
vol
```

## 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 STATUS
CLAIM STORAGECLASS REASON AGE
gluster-dev-volume 8Gi RWX Retain Availab
le 3s
```

## 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 ACCESSMOD
ES STORAGECLASS AGE
glusterfs-nginx Bound gluster-dev-volume 8Gi RWX
 4s
```

## 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-volume
172.20.0.113:k8s-volume 1.0T 0 1.0T 0% /usr/share/nginx
x/html

创建文件 测试
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- touch /usr/share/nginx/html/index.html

$ kubectl exec -it nginx-dm-3698525684-g0mvt -- ls -lt /usr/share/nginx/html/index.html
-rw-r--r-- 1 root root 0 May 4 11:36 /usr/share/nginx/html/index.html

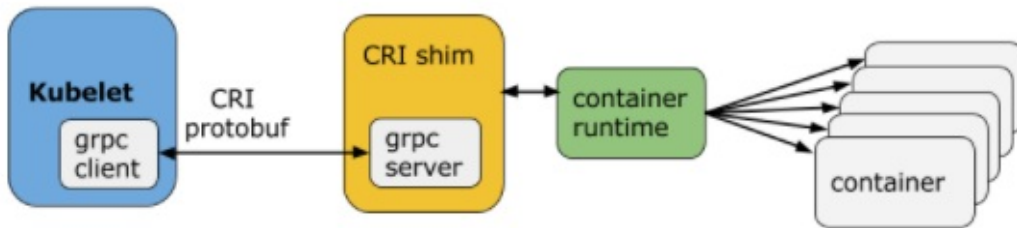
验证 glusterfs
因为我们使用分布卷，所以可以看到某个节点中有文件
[root@sz-pg-oam-docker-test-001 ~] ls /opt/gfs_data/
[root@sz-pg-oam-docker-test-002 ~] ls /opt/gfs_data/
index.html
[root@sz-pg-oam-docker-test-003 ~] ls /opt/gfs_data/
```

## 参考

- [CentOS 7 安装 GlusterFS](#)
- [GlusterFS with kubernetes](#)

## Container Runtime Interface

Container Runtime Interface (CRI)是Kubelet 1.5/1.6中主要负责的一块项目，它重新定义了Kubelet Container Runtime API，将原来完全面向Pod级别的API拆分成面向Sandbox和Containers的API，并分离镜像管理和容器引擎到不同的服务。



CRI最早从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Runtime，如frakti、cri-o的alpha支持。v1.7版本新增了containerd的alpha支持，而frakti和cri-o则升级到beta支持。

### CRI接口

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

- v1.7: [pkg/kubelet/apis/cri/v1alpha1/runtime](#)
- v1.6: [pkg/kubelet/api/v1alpha1/runtime](#)

Kubelet作为CRI的客户端，而Runtime维护者则需要实现CRI服务端，并在启动kubelet时将其传入：

```
kubelet --container-runtime=remote --container-runtime-endpoint=/var/run/frakti.sock ..
```

## 如何开发新的Container Runtime

开发新的Container Runtime只需要实现CRI 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）。

具体的实现方法可以参考下面已经支持的Container Runtime列表。

## 目前支持的**Container Runtime**

目前，有多家厂商都在基于CRI集成自己的容器引擎，其中包括

- Docker: 核心代码依然保留在kubelet内部（[pkg/kubelet/dockershim](#)），依然是最稳定和特性支持最好的Runtime
- HyperContainer: <https://github.com/kubernetes/frakti>，支持Kubernetes v1.6/v1.7，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
- Rkt: <https://github.com/kubernetes-incubator/rktlet>，开发中
- Runc有两个实现，cri-o和cri-containerd
  - [cri-containerd](#)，支持kubernetes v1.7。
  - [cri-o](#)，支持Kubernetes v1.6/v1.7，底层运行时支持runc和intel clear container。
- Mirantis: <https://github.com/Mirantis/virtlet>，直接管理libvirt虚拟机，镜像须是qcow2格式
- Infranetes: <https://github.com/apporbit/infranetes>，直接管理IaaS平台虚拟机，如GCE、AWS等

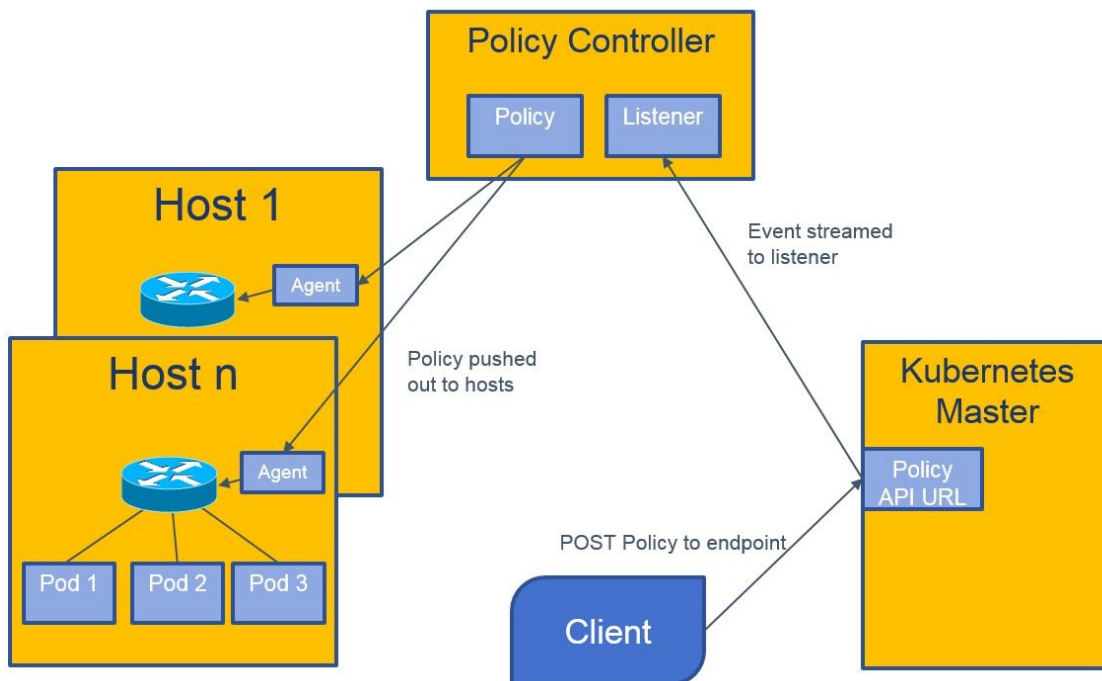
## Network Policy扩展

**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](#)
- [Tireme](#)
- [OpenContrail](#)

## Network Policy使用方法

具体Network Policy的使用方法可以参考[这里](#)。



# Ingress Controller扩展

[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和提供访问入口。

## 如何开发Ingress Controller扩展

[kubernetes/ingress](#)提供了一个Ingress Controller的基本框架，可以在此基础上方便的开发新的Ingress Controller。

## 常见Ingress Controller

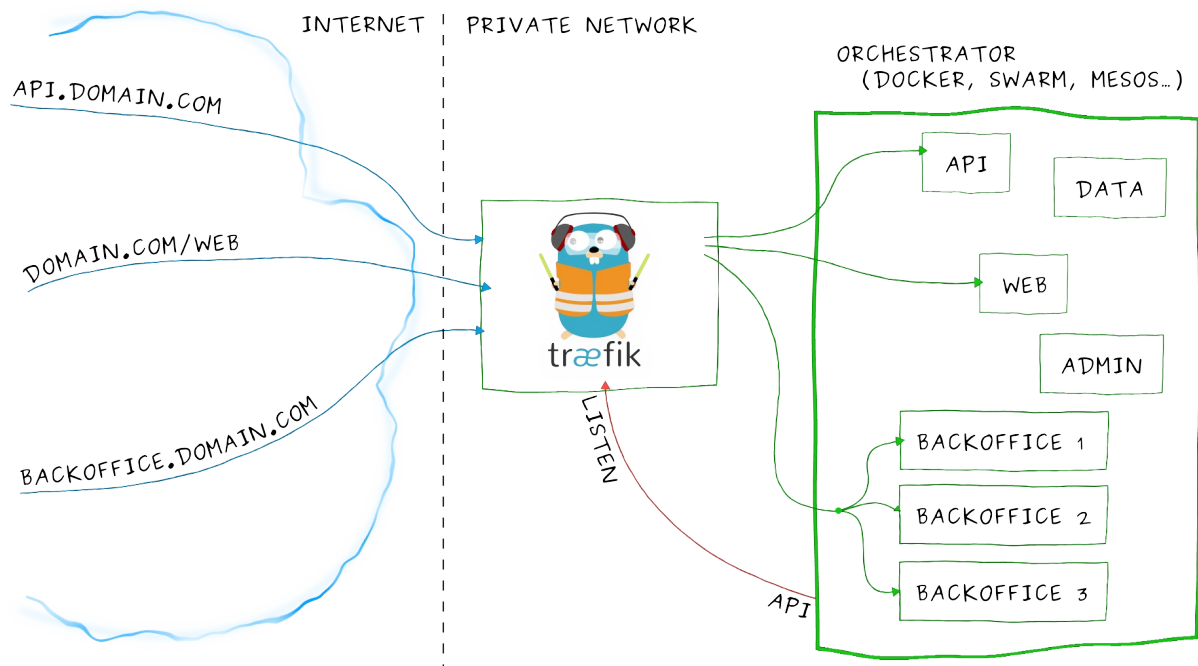
- [Ingress Examples](#)
- [Dummy controller backend](#)
- [HAProxy Ingress controller](#)
- [Linkerd](#)
- [traefik](#)
- [AWS Application Load Balancer Ingress Controller](#)
- [kube-ingress-aws-controller](#)
- [Voyager: HAProxy Ingress Controller](#)

## Ingress使用方法

具体Ingress的使用方法可以参考[这里](#)。

# Traefik ingress

**Traefik**是一个开源的反向代理和负载均衡工具，它监听后端的变化并自动更新服务配置。



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, Etc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

本章内容包括

- 安装Traefik ingress
- 分布式负载测试
- 网络和集群性能测试
- 边缘节点配置

# 安装traefik ingress

## 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等负载均衡方向代理服务器，其中还包括规则定义，即URL的路由信息，路由信息得的刷新由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。

## 部署Traefik

### 介绍traefik

Traefik是一款开源的反向代理与负载均衡工具。它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, Mesos/Marathon, Mesos, Kubernetes, Consul, Etcd, Zookeeper, BoltDB, Rest API等等后端模型。

以下配置文件可以在[kubernetes-handbook](#)GitHub仓库中的[manifests/traefik-ingress/](#)目录下找到。

### 创建ingress-rbac.yaml

将用于service account验证。

```
apiVersion: v1
kind: ServiceAccount
metadata:
 name: ingress
 namespace: kube-system

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
 name: ingress
subjects:
 - kind: ServiceAccount
 name: ingress
 namespace: kube-system
roleRef:
 kind: ClusterRole
 name: cluster-admin
 apiGroup: rbac.authorization.k8s.io
```

创建名为 **traefik-ingress** 的**ingress**，文件名traefik.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: traefik-ingress
spec:
 rules:
 - host: traefik.nginx.io
 http:
 paths:
 - path: /
 backend:
 serviceName: my-nginx
 servicePort: 80
 - host: traefik.frontend.io
 http:
 paths:
 - path: /
 backend:
 serviceName: frontend
 servicePort: 80
```

这其中的 `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，service将会接受path这个路径，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的名字和端口自行修改，有新service增加时，修改该文件后可以使用 `kubectl replace -f traefik.yaml` 来更新。

我们现在集群中已经有两个service了，一个是nginx，另一个是官方的 `guestbook` 例子。

## 创建Deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: traefik-ingress-lb
 namespace: kube-system
 labels:
 k8s-app: traefik-ingress-lb
spec:
 template:
 metadata:
 labels:
 k8s-app: traefik-ingress-lb
 name: traefik-ingress-lb
 spec:
 terminationGracePeriodSeconds: 60
 hostNetwork: true
 restartPolicy: Always
 serviceAccountName: ingress
 containers:
 - image: traefik
 name: traefik-ingress-lb
 resources:
 limits:
 cpu: 200m
 memory: 30Mi
 requests:
 cpu: 100m
 memory: 20Mi
 ports:
 - name: http
 containerPort: 80
 hostPort: 80
 - name: admin
 containerPort: 8580
 hostPort: 8580
 args:
 - --web
 - --web.address=:8580
 - --kubernetes
```

注意我们这里用的是Deploy类型，没有限定该pod运行在哪个主机上。Traefik的端口是8580。

## Traefik UI

```
apiVersion: v1
kind: Service
metadata:
 name: traefik-web-ui
 namespace: kube-system
spec:
 selector:
 k8s-app: traefik-ingress-lb
 ports:
 - name: web
 port: 80
 targetPort: 8580

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: traefik-web-ui
 namespace: kube-system
spec:
 rules:
 - host: traefik-ui.local
 http:
 paths:
 - path: /
 backend:
 serviceName: traefik-web-ui
 servicePort: web
```

配置完成后就可以启动traefik ingress了。

```
kubectl create -f .
```

我查看到traefik的pod在 172.20.0.115 这台节点上启动了。

访问该地址 <http://172.20.0.115:8580/> 将可以看到dashboard。

 Providers Health v1.2.3 / morbier Documentation [traefik.io](https://traefik.io)

kubernetes

traefik-ui.local/

Route	Rule
/	PathPrefix:/
traefik-ui.local	Host:traefik-ui.local

http Backend:traefik-ui.local/ PassHostHeader Priority:1

traefik.frontend.io/

Route	Rule
/	PathPrefix:/
traefik.frontend.io	Host:traefik.frontend.io

http Backend:traefik.frontend.io/ PassHostHeader Priority:1

traefik.nginx.io/

Route	Rule
/	PathPrefix:/
traefik.nginx.io	Host:traefik.nginx.io

http Backend:traefik.nginx.io/ PassHostHeader Priority:1

traefik-ui.local/

Server	URL	Weight
traefik-ingress-lb-4237248072-4009r	http://172.20.0.115:8580	1

Load Balancer: wrr

traefik.frontend.io/

Server	URL	Weight
frontend-1289468719-6l4v7	http://172.30.60.11:80	1
frontend-1289468719-sfkvb	http://172.30.71.5:80	1
frontend-1289468719-vg4zz	http://172.30.94.9:80	1

Load Balancer: wrr

traefik.nginx.io/

Server	URL	Weight
my-nginx-2096504489-1jg9c	http://172.30.60.5:80	1
my-nginx-2096504489-1vl95	http://172.30.94.6:80	1

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。

## 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/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
 body {
 width: 35em;
 margin: 0 auto;
 font-family: Tahoma, Verdana, Arial, sans-serif;
 }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
nginx.org.

Commercial support is available at
nginx.com.</p>

<p>Thank you for using nginx.</p>
</body>
</html>
```

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。

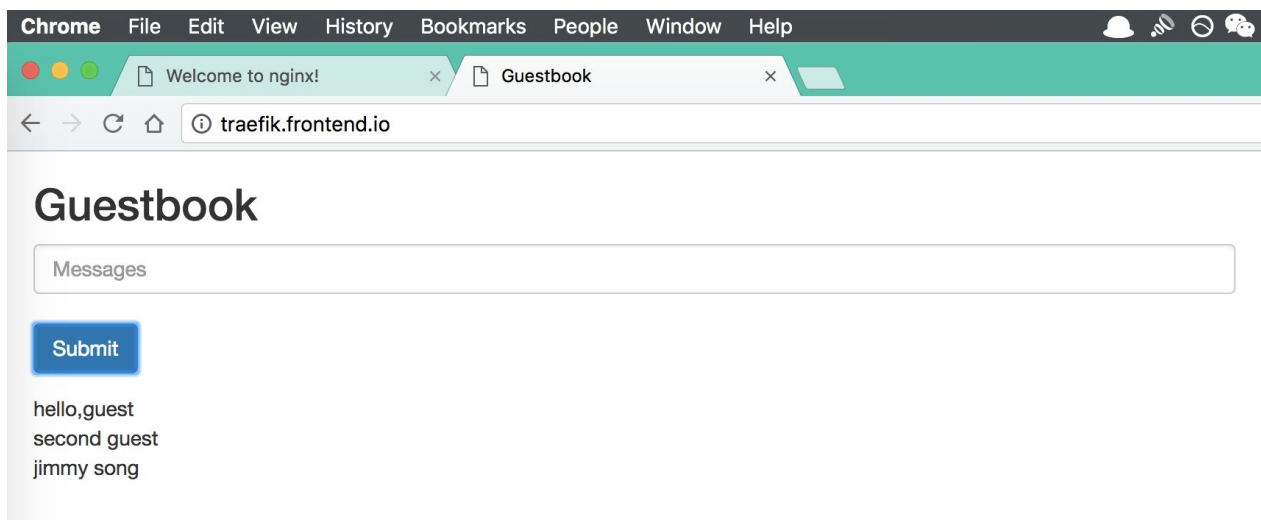
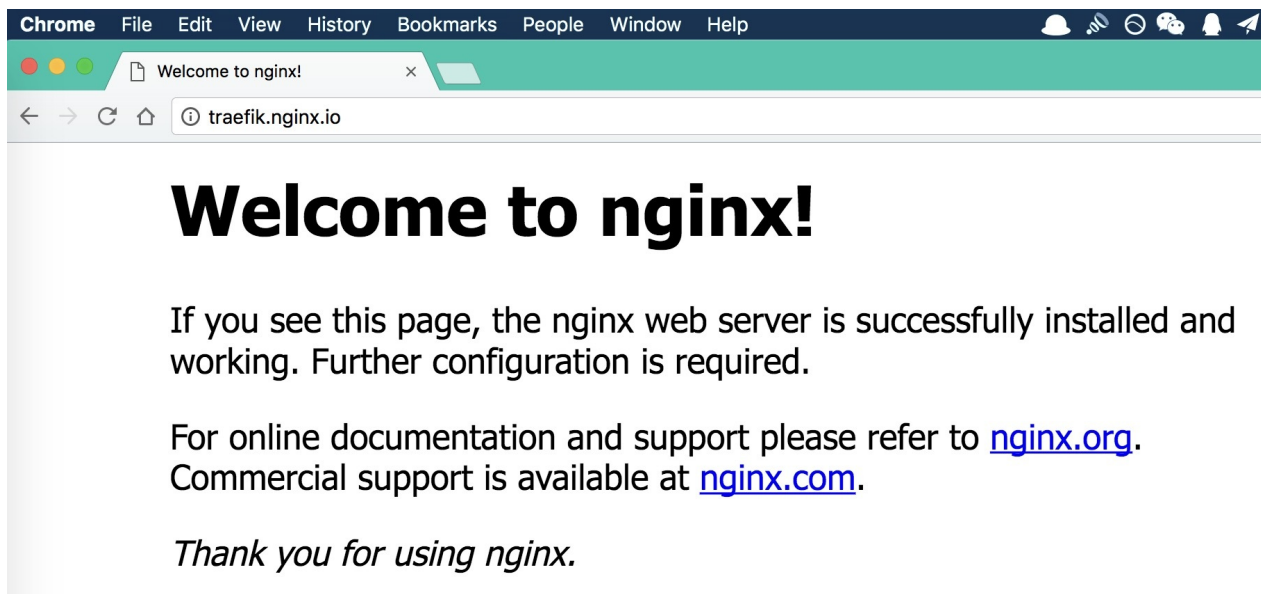
在其中加入：

```
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。

修改hosts后就可以在kubernetes集群外访问以上两个service，如下图：



## 参考文档

- [Traefik-kubernetes 初试](#)
- [Traefik 简介](#)
- [Guestbook example](#)

## 分布式负载测试

该教程描述如何在Kubernetes中进行分布式负载均衡测试，包括一个web应用、docker镜像和Kubernetes controllers/services。更多资料请查看[Distributed Load Testing Using Kubernetes](#)。

## 准备

不需要GCE及其他组件，你只需要有一个kubernetes集群即可。

如果你还没有kubernetes集群，可以参考[kubernetes-handbook](#)部署一个。

## 部署Web应用

`sample-webapp` 目录下包含一个简单的web测试应用。我们将其构建为docker镜像，在kubernetes中运行。你可以自己构建，也可以直接用这个我构建好的镜像 `index.tenxcloud.com/jimmy/k8s-sample-webapp:latest`。

在kubernetes上部署sample-webapp。

```
$ cd kubernetes-config
$ kubectl create -f sample-webapp-controller.yaml
$ kubectl create -f sample-webapp-service.yaml
```

## 部署Locust的Controller和Service

`locust-master` 和 `locust-work` 使用同样的docker镜像，修改controller中 `spec.template.spec.containers.env` 字段中的value为你 `sample-webapp` service的名字。

```
- name: TARGET_HOST
 value: http://sample-webapp:8000
```

## 创建Controller Docker镜像（可选）

`locust-master` 和 `locust-work` `controller`使用的都是 `locust-tasks` `docker`镜像。你可以直接下载 `gcr.io/cloud-solutions-images/locust-tasks`，也可以自己编译。自己编译大概要花几分钟时间，镜像大小为820M。

```
$ docker build -t index.tenxcloud.com/jimmy/locust-tasks:latest .
$ docker push index.tenxcloud.com/jimmy/locust-tasks:latest
```

注意：我使用的是时速云的镜像仓库。

每个`controller`的`yml`的 `spec.template.spec.containers.image` 字段指定的是我的镜像：

```
image: index.tenxcloud.com/jimmy/locust-tasks:latest
```

## 部署locust-master

```
$ kubectl create -f locust-master-controller.yaml
$ kubectl create -f locust-master-service.yaml
```

## 部署locust-worker

Now deploy `locust-worker-controller` :

```
$ kubectl create -f locust-worker-controller.yaml
```

你可以很轻易的给`work`扩容，通过命令行方式：

```
``ba sh $ kubectl scale --replicas=20 replicationcontrollers locust-worker
```

当然你也可以通过WebUI : Dashboard - Workloads - Replication Controllers - \*\*ServiceName\*\* - Scale来扩容。

![dashboard-scale](images/dashbaord-scale.jpg)

#### 配置Traefik

参考[kubernetes的traefik ingress安装](https://github.com/feiskyer/kubernetes-handbook/blob/master/practice/service-discovery-lb/traefik-ingress-installation.md)，在`ingress.yaml`中加入如下配置：

```
```Yaml
- host: traefik.locust.io
  http:
    paths:
      - path: /
        backend:
          serviceName: locust-master
          servicePort: 8089
```

然后执行 `kubect1 replace -f ingress.yaml` 即可更新traefik。

通过Traefik的dashboard就可以看到刚增加的 `traefik.locust.io` 节点。

Route	Rule
/	PathPrefix:/
traefik.guestbook.io	Host:traefik.guestbook.io

httpBackend:traefik.guestbook.io/PassHostHeaderPriority:1

traefik.locust.io/

Route	Rule
/	PathPrefix:/
traefik.locust.io	Host:traefik.locust.io

httpBackend:traefik.locust.io/PassHostHeaderPriority:1

traefik.nginx.io/

Route	Rule
/	PathPrefix:/
traefik.nginx.io	Host:traefik.nginx.io

httpBackend:traefik.nginx.io/PassHostHeaderPriority:1

Server	URL	Weight
frontend-1289468719-4565s	http://172.30.94.9:80	1
frontend-1289468719-rf8rw	http://172.30.71.3:80	1
frontend-1289468719-s4m6p	http://172.30.94.10:80	1

Load Balancer: wrr

traefik.locust.io/

Server	URL	Weight
locust-master-p8vpn	http://172.30.94.3:8089	1

Load Balancer: wrr

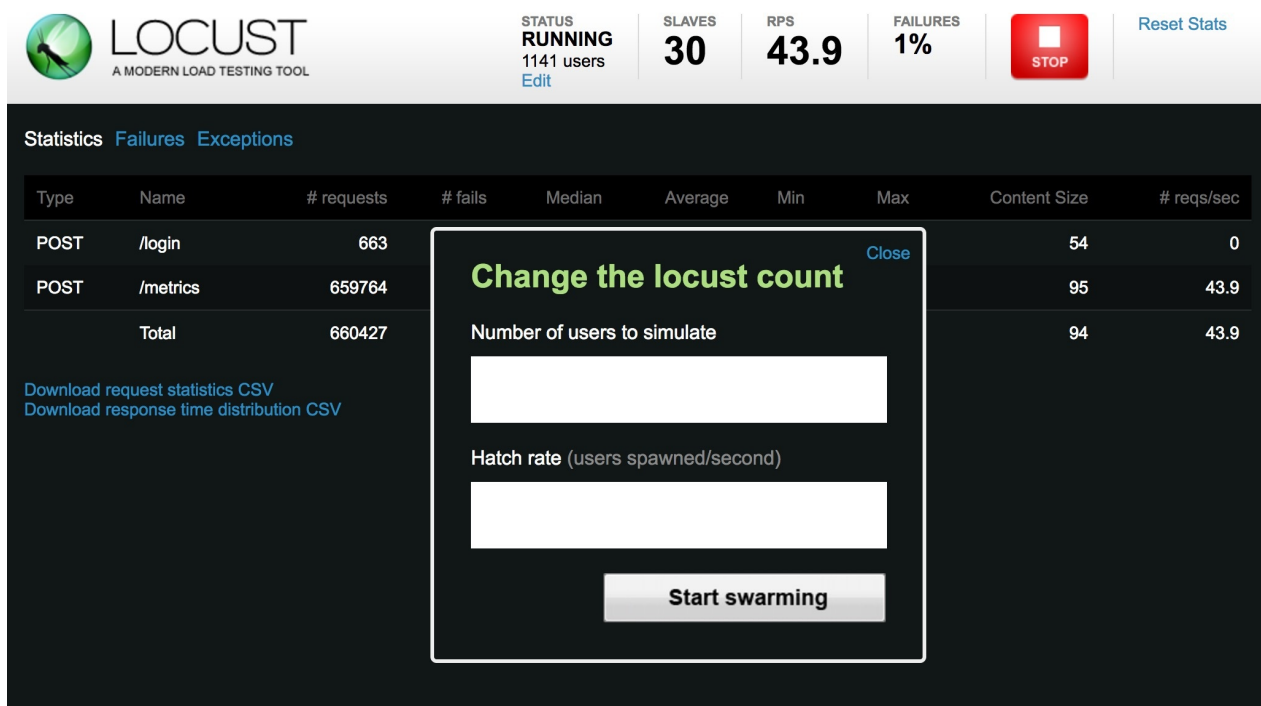
traefik.nginx.io/

Server	URL	Weight
my-nginx-2096504489-6l1zs	http://172.30.71.11:80	1
my-nginx-2096504489-9jh91	http://172.30.71.10:80	1
my-nginx-2096504489-9s58t	http://172.30.94.8:80	1
my-nginx-2096504489-gt621	http://172.30.71.12:80	1
my-nginx-2096504489-mp4gt	http://172.30.94.11:80	1

Load Balancer: wrr

执行测试

打开 `http://traefik.locust.io` 页面，点击 `Edit` 输入伪造的用户数和用户每秒发送的请求个数，点击 `Start Swarming` 就可以开始测试了。

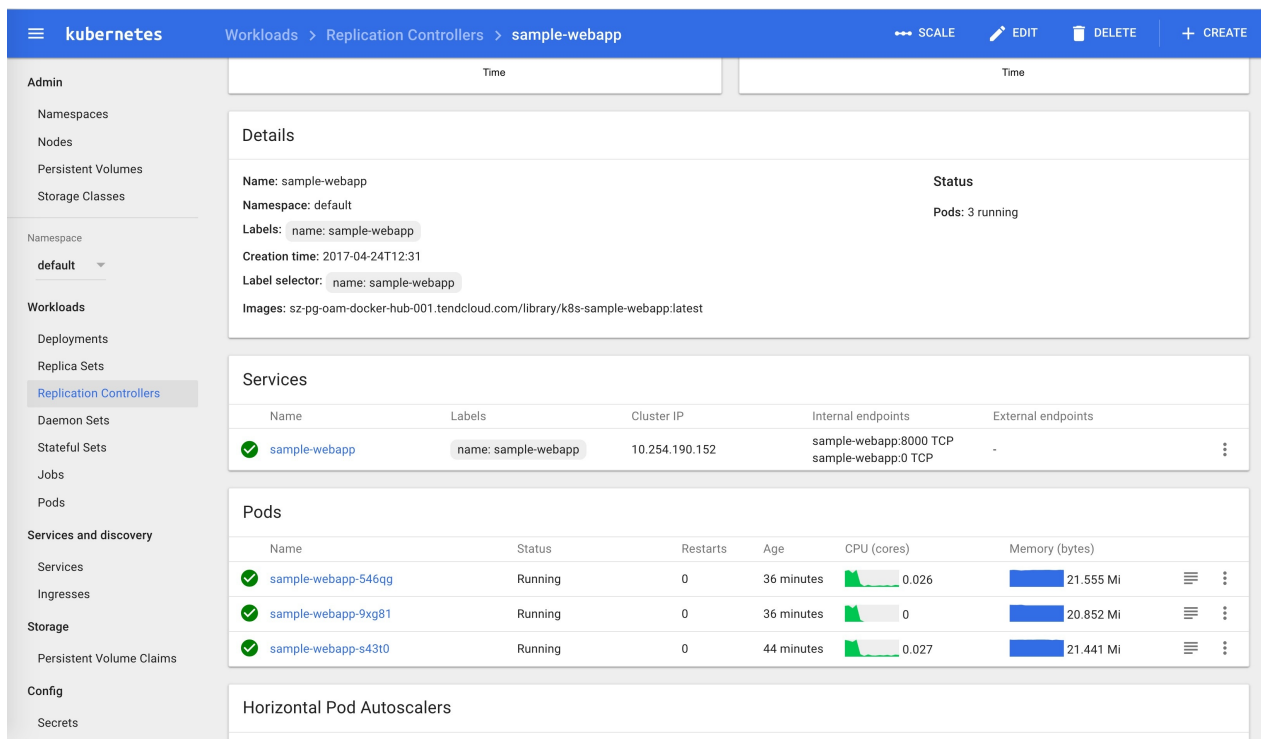


The Locust interface shows a top status bar with the following metrics: STATUS RUNNING (1141 users), SLAVES 30, RPS 43.9, and FAILURES 1%. A red STOP button and a Reset Stats link are also present. Below the status bar, there are tabs for Statistics, Failures, and Exceptions. The Statistics tab is active, displaying a table of request statistics. A modal window titled 'Change the locust count' is open, allowing users to adjust the number of users to simulate and the hatch rate (users spawned/second). The modal includes a 'Start swarming' button.

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/login	663						54	0
POST	/metrics	659764						95	43.9
Total		660427						94	43.9

Download request statistics CSV
Download response time distribution CSV

在测试过程中调整 `sample-webapp` 的pod个数（默认设置了1个pod），观察pod的负载变化情况。



The Kubernetes dashboard shows the 'sample-webapp' Replication Controller. The 'Details' section displays the following information: Name: sample-webapp, Namespace: default, Labels: name: sample-webapp, Creation time: 2017-04-24T12:31, Label selector: name: sample-webapp, and Images: sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-sample-webapp:latest. The 'Status' section shows 'Pods: 3 running'. The 'Services' section shows a single service named 'sample-webapp' with Cluster IP 10.254.190.152 and internal endpoints 'sample-webapp:8000 TCP' and 'sample-webapp:0 TCP'. The 'Pods' section shows three running pods: 'sample-webapp-546qg' (36 minutes, 0.026 CPU), 'sample-webapp-9xg81' (36 minutes, 0 CPU), and 'sample-webapp-s43t0' (44 minutes, 0.027 CPU). The 'Horizontal Pod Autoscalers' section is empty.

从一段时间的观察中可以看到负载被平均分配给了3个pod。

在locust的页面中可以实时观察也可以下载测试结果。



LOCUST
A MODERN LOAD TESTING TOOL

STATUS
RUNNING
1180 users
[Edit](#)

SLAVES
30

RPS
16.5

FAILURES
1%

[Reset Stats](#)

[Statistics](#) [Failures](#) [Exceptions](#)

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/login	664	4	4	111	7	15034	54	0
POST	/metrics	660469	5609	5	122	3	15836	95	16.5
Total		661133	5613	5	122	3	15836	94	16.5

[Download request statistics CSV](#)
[Download response time distribution CSV](#)

Kubernetes网络和集群性能测试

准备

测试环境

在以下几种环境下进行测试：

- Kubernetes集群node节点上通过Cluster IP方式访问
- Kubernetes集群内部通过service访问
- Kubernetes集群外部通过traefik ingress暴露的地址访问

测试地址

Cluster IP: 10.254.149.31

Service Port : 8000

Ingress Host : traefik.sample-webapp.io

测试工具

- [Locust](#)：一个简单易用的用户负载测试工具，用来测试web或其他系统能够同时处理的并发用户数。
- curl
- [kubemark](#)
- 测试程序：sample-webapp，源码见Github [kubernetes的分布式负载测试](#)

测试说明

通过向 `sample-webapp` 发送curl请求获取响应时间，直接curl后的结果为：

```
$ curl "http://10.254.149.31:8000/"  
Welcome to the "Distributed Load Testing Using Kubernetes" sample web  
app
```

网络延迟测试

场景一、Kubernetes集群node节点上通过Cluster IP访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://10.254.149.31:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.000	0.003	0.003
2	0.000	0.002	0.002
3	0.000	0.002	0.002
4	0.000	0.002	0.002
5	0.000	0.002	0.002
6	0.000	0.002	0.002
7	0.000	0.002	0.002
8	0.000	0.002	0.002
9	0.000	0.002	0.002
10	0.000	0.002	0.002

平均响应时间：**2ms**

时间指标说明

单位：秒

time_connect：建立到服务器的 TCP 连接所用的时间

time_starttransfer：在发出请求之后，Web 服务器返回数据的第一个字节所用的时间

time_total：完成请求所用的时间

场景二、Kubernetes 集群内部通过service访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://sample-webapp:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.004	0.006	0.006
2	0.004	0.006	0.006
3	0.004	0.006	0.006
4	0.004	0.006	0.006
5	0.004	0.006	0.006
6	0.004	0.006	0.006
7	0.004	0.006	0.006
8	0.004	0.006	0.006
9	0.004	0.006	0.006
10	0.004	0.006	0.006

平均响应时间：**6ms**

场景三、在公网上通过traefik ingress访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://traefik.sample-webapp.io" >>result
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.043	0.085	0.085
2	0.052	0.093	0.093
3	0.043	0.082	0.082
4	0.051	0.093	0.093
5	0.068	0.188	0.188
6	0.049	0.089	0.089
7	0.051	0.113	0.113
8	0.055	0.120	0.120
9	0.065	0.126	0.127
10	0.050	0.111	0.111

平均响应时间：**110ms**

测试结果

在这三种场景下的响应时间测试结果如下：

- Kubernetes集群node节点上通过Cluster IP方式访问：2ms
- Kubernetes集群内部通过service访问：6ms
- Kubernetes集群外部通过traefik ingress暴露的地址访问：110ms

注意：执行测试的`node`节点/`Pod`与`service`所在的`pod`的距离（是否在同一台主机上），对前两个场景可能会有一定影响。

网络性能测试

网络使用flannel的vxlan模式。

使用iperf进行测试。

服务端命令：

```
iperf -s -p 12345 -i 1 -M
```

客户端命令：

```
iperf -c ${server-ip} -p 12345 -i 1 -t 10 -w 20K
```

场景一、主机之间

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	598 MBytes	5.02 Gbits/sec
[3]	1.0- 2.0 sec	637 MBytes	5.35 Gbits/sec
[3]	2.0- 3.0 sec	664 MBytes	5.57 Gbits/sec
[3]	3.0- 4.0 sec	657 MBytes	5.51 Gbits/sec
[3]	4.0- 5.0 sec	641 MBytes	5.38 Gbits/sec
[3]	5.0- 6.0 sec	639 MBytes	5.36 Gbits/sec
[3]	6.0- 7.0 sec	628 MBytes	5.26 Gbits/sec
[3]	7.0- 8.0 sec	649 MBytes	5.44 Gbits/sec
[3]	8.0- 9.0 sec	638 MBytes	5.35 Gbits/sec
[3]	9.0-10.0 sec	652 MBytes	5.47 Gbits/sec
[3]	0.0-10.0 sec	6.25 GBytes	5.37 Gbits/sec

场景二、不同主机的Pod之间(使用flannel的vxlan模式)

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[3]	1.0- 2.0 sec	345 MBytes	2.89 Gbits/sec
[3]	2.0- 3.0 sec	361 MBytes	3.03 Gbits/sec
[3]	3.0- 4.0 sec	397 MBytes	3.33 Gbits/sec
[3]	4.0- 5.0 sec	405 MBytes	3.40 Gbits/sec
[3]	5.0- 6.0 sec	410 MBytes	3.44 Gbits/sec
[3]	6.0- 7.0 sec	404 MBytes	3.39 Gbits/sec
[3]	7.0- 8.0 sec	408 MBytes	3.42 Gbits/sec
[3]	8.0- 9.0 sec	451 MBytes	3.78 Gbits/sec
[3]	9.0-10.0 sec	387 MBytes	3.25 Gbits/sec
[3]	0.0-10.0 sec	3.85 GBytes	3.30 Gbits/sec

场景三、Node与非同主机的Pod之间（使用flannel的vxlan模式）

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[3]	1.0- 2.0 sec	420 MBytes	3.53 Gbits/sec
[3]	2.0- 3.0 sec	434 MBytes	3.64 Gbits/sec
[3]	3.0- 4.0 sec	409 MBytes	3.43 Gbits/sec
[3]	4.0- 5.0 sec	382 MBytes	3.21 Gbits/sec
[3]	5.0- 6.0 sec	408 MBytes	3.42 Gbits/sec
[3]	6.0- 7.0 sec	403 MBytes	3.38 Gbits/sec
[3]	7.0- 8.0 sec	423 MBytes	3.55 Gbits/sec
[3]	8.0- 9.0 sec	376 MBytes	3.15 Gbits/sec
[3]	9.0-10.0 sec	451 MBytes	3.78 Gbits/sec
[3]	0.0-10.0 sec	3.98 GBytes	3.42 Gbits/sec

场景四、不同主机的**Pod**之间（使用**flannel**的**host-gw**模式）

[ID]	Interval	Transfer	Bandwidth
[5]	0.0- 1.0 sec	530 MBytes	4.45 Gbits/sec
[5]	1.0- 2.0 sec	576 MBytes	4.84 Gbits/sec
[5]	2.0- 3.0 sec	631 MBytes	5.29 Gbits/sec
[5]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[5]	4.0- 5.0 sec	627 MBytes	5.26 Gbits/sec
[5]	5.0- 6.0 sec	578 MBytes	4.85 Gbits/sec
[5]	6.0- 7.0 sec	584 MBytes	4.90 Gbits/sec
[5]	7.0- 8.0 sec	571 MBytes	4.79 Gbits/sec
[5]	8.0- 9.0 sec	564 MBytes	4.73 Gbits/sec
[5]	9.0-10.0 sec	572 MBytes	4.80 Gbits/sec
[5]	0.0-10.0 sec	5.68 GBytes	4.88 Gbits/sec

场景五、**Node**与非同主机的**Pod**之间（使用**flannel**的**host-gw**模式）

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	570 MBytes	4.78 Gbits/sec
[3]	1.0- 2.0 sec	552 MBytes	4.63 Gbits/sec
[3]	2.0- 3.0 sec	598 MBytes	5.02 Gbits/sec
[3]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[3]	4.0- 5.0 sec	590 MBytes	4.95 Gbits/sec
[3]	5.0- 6.0 sec	594 MBytes	4.98 Gbits/sec
[3]	6.0- 7.0 sec	598 MBytes	5.02 Gbits/sec
[3]	7.0- 8.0 sec	606 MBytes	5.08 Gbits/sec
[3]	8.0- 9.0 sec	596 MBytes	5.00 Gbits/sec
[3]	9.0-10.0 sec	604 MBytes	5.07 Gbits/sec
[3]	0.0-10.0 sec	5.75 GBytes	4.94 Gbits/sec

网络性能对比综述

使用Flannel的**vxlan**模式实现每个pod一个IP的方式，会比宿主机直接互联的网络性能损耗30%~40%，符合网上流传的测试结论。而flannel的host-gw模式比起宿主机互连的网络性能损耗大约是10%。

Vxlan会有一个封包解包的过程，所以会对网络性能造成较大的损耗，而host-gw模式是直接使用路由信息，网络损耗小，关于host-gw的架构请访问[Flannel host-gw architecture](#)。

Kubernetes的性能测试

参考[Kubernetes集群性能测试](#)中的步骤，对kubernetes的性能进行测试。

我的集群版本是Kubernetes1.6.0，首先克隆代码，将kubernetes目录复制到 `$GOPATH/src/k8s.io/` 下然后执行：

```

$ ./hack/generate-bindata.sh
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes
Generated bindata file : test/e2e/generated/bindata.go has 13498 test
/e2e/generated/bindata.go lines of lovely automated artifacts
No changes in generated bindata file: pkg/generated/bindata.go
/usr/local/src/k8s.io/kubernetes
$ make WHAT="test/e2e/e2e.test"
...
+++ [0425 17:01:34] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test
/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:01:34] Building go targets for linux/amd64:
    test/e2e/e2e.test
$ make ginkgo
+++ [0425 17:05:57] Building the toolchain targets:
    k8s.io/kubernetes/hack/cmd/teststale
    k8s.io/kubernetes/vendor/github.com/jteeuwen/go-bindata/go-bindat
a
+++ [0425 17:05:57] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test
/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:05:58] Building go targets for linux/amd64:
    vendor/github.com/onsi/ginkgo/ginkgo

$ export KUBERNETES_PROVIDER=local
$ export KUBECTL_PATH=/usr/bin/kubect1
$ go run hack/e2e.go -v -test --test_args="--host=http://172.20.0.11
3:8080 --ginkgo.focus=\[Feature:Performance\]" >>log.txt

```

测试结果

```

Apr 25 18:27:31.461: INFO: API calls latencies: {
  "apicalls": [
    {
      "resource": "pods",
      "verb": "POST",

```



```
    "latency": {
      "Perc50": 2148000,
      "Perc90": 13772000,
      "Perc99": 14436000,
      "Perc100": 0
    }
  },
  {
    "resource": "services",
    "verb": "DELETE",
    "latency": {
      "Perc50": 9843000,
      "Perc90": 11226000,
      "Perc99": 12391000,
      "Perc100": 0
    }
  },
  ...
Apr 25 18:27:31.461: INFO: [Result:Performance] {
  "version": "v1",
  "dataItems": [
    {
      "data": {
        "Perc50": 2.148,
        "Perc90": 13.772,
        "Perc99": 14.436
      },
      "unit": "ms",
      "labels": {
        "Resource": "pods",
        "Verb": "POST"
      }
    },
    ...
  ],
  ...
2.857: INFO: Running AfterSuite actions on all node
Apr 26 10:35:32.857: INFO: Running AfterSuite actions on node 1

Ran 2 of 606 Specs in 268.371 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 604 Skipped PASS

Ginkgo ran 1 suite in 4m28.667870101s
Test Suite Passed
```

从kubemark输出的日志中可以看到**API calls latencies**和**Performance**。

日志里显示，创建**90个pod**用时**40秒**以内，平均创建每个**pod**耗时**0.44秒**。

不同type的资源类型API请求耗时分布

Resource	Verb	50%	90%	99%
services	DELETE	8.472ms	9.841ms	38.226ms
endpoints	PUT	1.641ms	3.161ms	30.715ms
endpoints	GET	931µs	10.412ms	27.97ms
nodes	PATCH	4.245ms	11.117ms	18.63ms
pods	PUT	2.193ms	2.619ms	17.285ms

从 `log.txt` 日志中还可以看到更多详细请求的测试指标。

kubernetes

Workloads > Replication Controllers

+ CREATE

Admin

Namespaces

Nodes

Persistent Volumes

Storage Classes

Namespaces

e2e-tests-load-30-nodepods-1-v

Workloads

Deployments

Replica Sets

Replication Controllers

Daemon Sets

Stateful Sets

Jobs

Pods

Services and discovery

Services

Ingresses

Storage

Persistent Volume Claims

Config

Secrets

Replication Controllers

1 - 10 of 13

<

>

Name	Labels	Pods	Age	Images	
load-medium-1	name: load-medium-1	30 / 30	44 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-1	name: load-small-1	5 / 5	43 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-10	name: load-small-10	5 / 5	42 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-11	name: load-small-11	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-12	name: load-small-12	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-2	name: load-small-2	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-3	name: load-small-3	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-4	name: load-small-4	5 / 5	39 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-5	name: load-small-5	5 / 5	41 seconds	gcr.io/google_containers/serve_hostname:v1.4	
load-small-6	name: load-small-6	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4	

注意事项

测试过程中需要用到docker镜像存储在GCE中，需要翻墙下载，我没看到哪里配置这个镜像的地址。该镜像副本已上传时速云：

用到的镜像有如下两个：

- gcr.io/google_containers/pause-amd64:3.0
- gcr.io/google_containers/serve_hostname:v1.4

时速云镜像地址：

- index.tenxcloud.com/jimmy/pause-amd64:3.0
- index.tenxcloud.com/jimmy/serve_hostname:v1.4

将镜像pull到本地后重新打tag。

Locust测试

请求统计

Method	Name	# requests	# failures	Median response time	Average response time	Min response time
POST	/login	5070	78	59000	80551	11218
POST	/metrics	5114232	85879	63000	82280	29518
None	Total	5119302	85957	63000	82279	11218

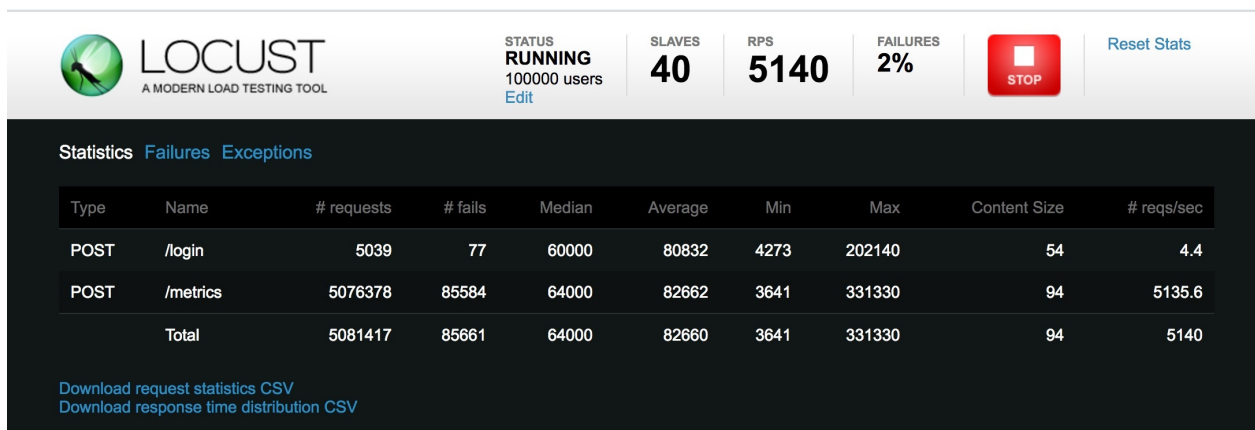
响应时间分布

Name	# requests	50%	66%	75%	80%	90%	95%
POST /login	5070	59000	125000	140000	148000	160000	16600
POST /metrics	5114993	63000	127000	142000	149000	160000	16600
None Total	5120063	63000	127000	142000	149000	160000	16600

以上两个表格都是瞬时值。请求失败率在2%左右。

Sample-webapp起了48个pod。

Locust模拟10万用户，每秒增长100个。



参考

[基于 Python 的性能测试工具 locust \(与 LR 的简单对比\)](#)

[Locust docs](#)

[python用户负载测试工具：locust](#)

[Kubernetes集群性能测试](#)

[CoreOS是如何将Kubernetes的性能提高10倍的](#)

[Kubernetes 1.3 的性能和弹性 —— 2000 节点，60,000 Pod 的集群](#)

[运用Kubernetes进行分布式负载测试](#)

[Kubemark User Guide](#)

[Flannel host-gw architecture](#)

边缘节点配置

前言

为了配置kubernetes中的traefik ingress的高可用，对于kubernetes集群以外只暴露一个访问入口，需要使用keepalived排除单点问题。本文参考了[kube-keepalived-vip](#)，但并没有使用容器方式安装，而是直接在node节点上安装。

定义

首先解释下什么叫边缘节点（Edge Node），所谓的边缘节点即集群内部用来向集群外暴露服务能力的节点，集群外部的服务通过该节点来调用集群内部的服务，边缘节点是集群内外交流的一个Endpoint。

边缘节点要考虑两个问题

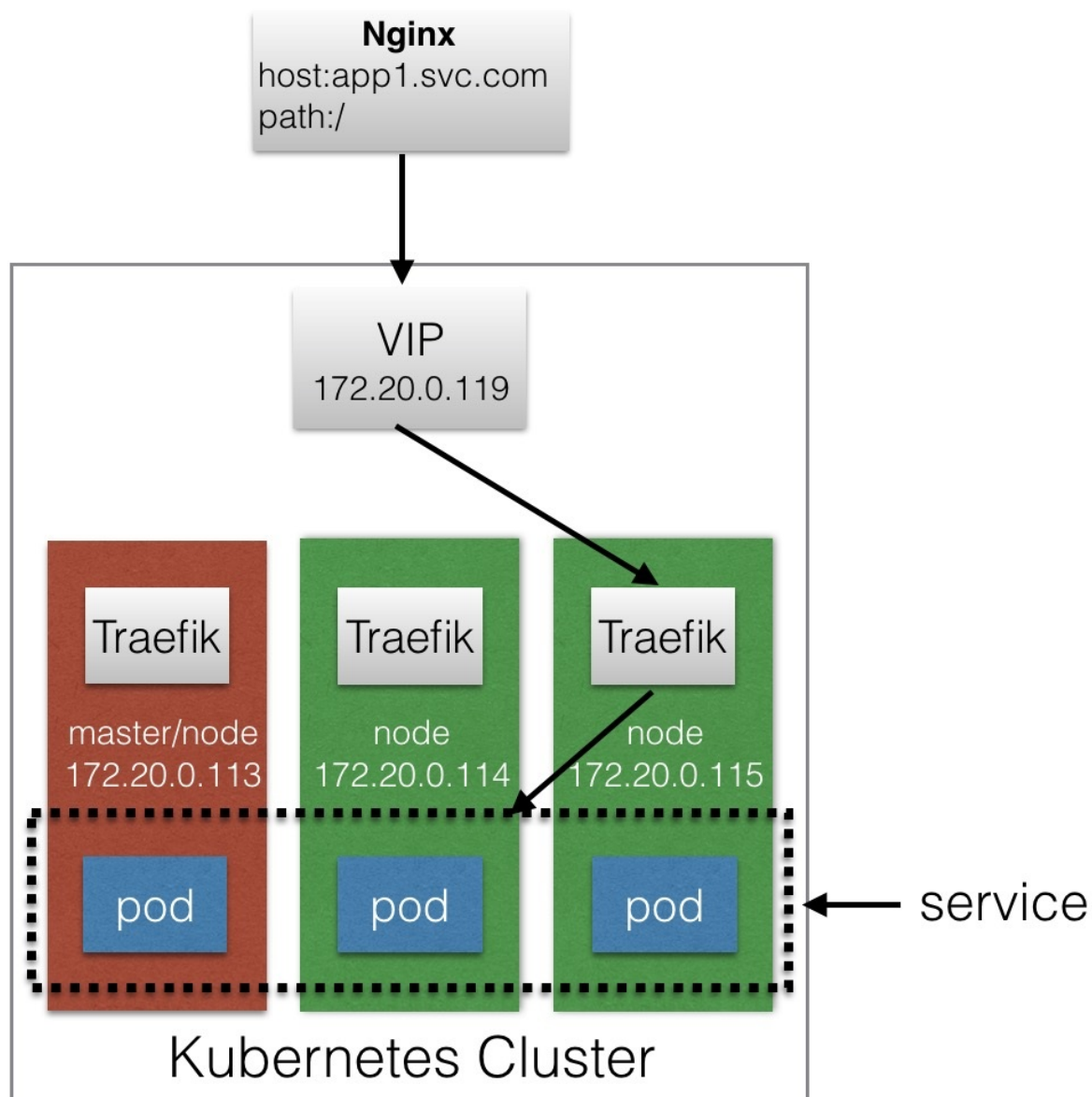
- 边缘节点的高可用，不能有单点故障，否则整个kubernetes集群将不可用
- 对外的一致暴露端口，即只能有一个外网访问IP和端口

架构

为了满足边缘节点的以上需求，我们使用[keepalived](#)来实现。

在Kubernetes集群外部配置nginx来访问边缘节点的VIP。

选择Kubernetes的三个node作为边缘节点，并安装keepalived。



准备

复用kubernetes测试集群的三台主机。

172.20.0.113

172.20.0.114

172.20.0.115

安装

使用keepalived管理VIP，VIP是使用IPVS创建的，IPVS已经成为linux内核的模块，不需要安装

LVS的工作原理请参

考：<http://www.cnblogs.com/codebean/archive/2011/07/25/2116043.html>

不使用镜像方式安装了，直接手动安装，指定三个节点为边缘节点（Edge node）。

因为我们的测试集群一共只有三个node，所有在三个node上都要安装keepalived和ipvsadmin。

```
yum install keepalived ipvsadm
```

配置说明

需要对原先的traefik ingress进行改造，从以Deployment方式启动改成DaemonSet。还需要指定一个与node在同一网段的IP地址作为VIP，我们指定成172.20.0.119，配置keepalived前需要先保证这个IP没有被分配。。

- Traefik以DaemonSet的方式启动
- 通过nodeSelector选择边缘节点
- 通过hostPort暴露端口
- 当前VIP漂移到了172.20.0.115上
- Traefik根据访问的host和path配置，将流量转发到相应的service上

配置keepalived

参考[基于keepalived 实现VIP转移，lvs，nginx的高可用](#)，配置keepalived。

keepalived的官方配置文档见：<http://keepalived.org/pdf/UserGuide.pdf>

配置文件 `/etc/keepalived/keepalived.conf` 文件内容如下：

```
! Configuration File for keepalived

global_defs {
    notification_email {
        root@localhost
    }
    notification_email_from kaadmin@localhost
```

```
smtp_server 127.0.0.1
smtp_connect_timeout 30
router_id LVS_DEVEL
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        172.20.0.119
    }
}

virtual_server 172.20.0.119 80{
    delay_loop 6
    lb_algo loadbalance
    lb_kind DR
    nat_mask 255.255.255.0
    persistence_timeout 0
    protocol TCP

    real_server 172.20.0.113 80{
        weight 1
        TCP_CHECK {
            connect_timeout 3
        }
    }
    real_server 172.20.0.114 80{
        weight 1
        TCP_CHECK {
            connect_timeout 3
        }
    }
    real_server 172.20.0.115 80{
        weight 1
```



```
TCP_CHECK {  
    connect_timeout 3  
}  
}  
}
```

Realserver 的IP和端口即traefik供外网访问的IP和端口。

将以上配置分别拷贝到另外两台node的 `/etc/keepalived` 目录下。

我们使用转发效率最高的 `lb_kind DR` 直接路由方式转发，使用TCP_CHECK来检测 `real_server`的health。

启动**keepalived**

```
systemctl start keepalived
```

三台node都启动了keepalived后，观察eth0的IP，会在三台node的某一台上发现一个VIP是172.20.0.119。

```
$ ip addr show eth0  
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP  
qlen 1000  
    link/ether f4:e9:d4:9f:6b:a0 brd ff:ff:ff:ff:ff:ff  
    inet 172.20.0.115/17 brd 172.20.127.255 scope global eth0  
        valid_lft forever preferred_lft forever  
    inet 172.20.0.119/32 scope global eth0  
        valid_lft forever preferred_lft forever
```

关掉拥有这个VIP主机上的keepalived，观察VIP是否漂移到了另外两台主机的其中之一上。

改造Traefik

在这之前我们启动的traefik使用的是deployment，只启动了一个pod，无法保证高可用（即需要将pod固定在某一台主机上，这样才能对外提供一个唯一的访问地址），现在使用了keepalived就可以通过VIP来访问traefik，同时启动多个traefik的pod保证高可用。

配置文件 `traefik.yaml` 内容如下：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
        - image: traefik
          name: traefik-ingress-lb
          resources:
            limits:
              cpu: 200m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
          ports:
            - name: http
              containerPort: 80
              hostPort: 80
            - name: admin
              containerPort: 8580
              hostPort: 8580
          args:
            - --web
            - --web.address=:8580
            - --kubernetes
      nodeSelector:
        edgenode: "true"
```

注意，我们使用了 `nodeSelector` 选择边缘节点来调度traefik-ingress-lb运行在它上面，所有你需要使用：

```
kubectl label nodes 172.20.0.113 edgenode=true
kubectl label nodes 172.20.0.114 edgenode=true
kubectl label nodes 172.20.0.115 edgenode=true
```

给三个node打标签。

查看DaemonSet的启动情况：

```
$ kubectl -n kube-system get ds
NAME                                DESIRED    CURRENT    READY    UP-TO-DATE    AVAIL
ABLE    NODE-SELECTOR
traefik-ingress-lb    3          3          3        3            3
          edgenode=true    2h
```

现在就可以在外网通过172.20.0.119:80来访问到traefik ingress了。

参考

[kube-keepalived-vip](#)

<http://www.keepalived.org/>

[keepalived工作原理与配置说明](#)

[LVS简介及使用](#)

[基于keepalived 实现VIP转移，lvs，nginx的高可用](#)

minikube Ingress

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	AGE
default-http-backend-5374j	1/1	Running	0	1m
kube-addon-manager-minikube	1/1	Running	0	2m
kube-dns-268032401-rhrx6	3/3	Running	0	1m
kubernetes-dashboard-xh74p	1/1	Running	0	2m
nginx-ingress-controller-78mk6	1/1	Running	0	1m

创建Ingress

首先启用一个echo server服务

```
$ kubectl run echoserver --image=gcr.io/google_containers/echoserver:1.4 --port=8080
$ 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 <<EOF | kubectl create -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 /etc/hosts
```

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 <<EOF | kubectl create -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
```

Cloud Provider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

Kubernetes内置的Cloud Provider包括

- GCE
- AWS
- Azure
- Mesos
- OpenStack
- CloudStack
- Ovirt
- Photon
- Rackspace
- Vsphere

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进一步重构 `cloud-controller-manager`，解耦了Controller Manager与Cloud Controller的代码逻辑

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](#)。

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 $SERVER get pods -o json | jq '
.items[] | select(.spec.schedulerName == "my-scheduler") | select(.spec.nodeName == null) | .metadata.name' | tr -d ' ');
    do
        NODES=$(kubectl --server $SERVER get nodes -o json | jq '.items[].metadata.name' | tr -d ' ')
        NUMNODES=${#NODES[@]}
        CHOSEN=${NODES[$[ $RANDOM % $NUMNODES ]]}
        curl --header "Content-Type:application/json" --request POST
        --data '{"apiVersion":"v1", "kind": "Binding", "metadata": {"name": "'$PODNAME'"}, "target": {"apiVersion": "v1", "kind": "Node", "name": "'$CHOSEN'"}' http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/
        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
```

Kubernetes应用管理

本章介绍Kubernetes manifest以及应用的管理方法。

目前最常用的是手动管理manifest，比如kubernetes github代码库就提供了很多的manifest示例

- <https://github.com/kubernetes/kubernetes/tree/master/examples>
- <https://github.com/kubernetes/contrib>
- <https://github.com/kubernetes/ingress>

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

- **Helm**提供了一些常见应用的模版
- **operator**则提供了一种有状态应用的管理模式
- **Deis**在Kubernetes之上提供了一个PaaS平台
- **Draft**是微软Deis团队开源的容器应用开发辅助工具，可以帮助开发人员简化容器应用程序的开发流程
- **Kompose**是一个将docker-compose配置转换成Kubernetes manifests的工具

一般准则

- 分离构建和运行环境
- 使用 `dumb-init` 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 app .

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 dock
er run busybox sleep 10000
root       14221  0.1  0.0   1188     4 ?        Ss   13:25   0:00 slee
p 10000

# 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 slee
p 10000
```

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 /bin/sh -c "sleep 10000"
```

参考文档

- [Kubernetes Production Patterns](#)

服务滚动升级

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（Horizontal 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的是输出到客户端的
}

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${TAG} main.go
    docker build -t sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG} .

# Push to tenxcloud
push:
    docker push sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG}

# 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/hello:v
1
          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-test=sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:v2
```

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 --image
sz-pg-oam-docker-hub-001.tendcloud.com/library/zeppelin:0.7.1
Created zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b
Scaling up zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b from
0 to 1, scaling down zeppelin-controller from 1 to 0 (keep 1 pods ava
ilable, don't exceed 2 pods)
Scaling zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b up to 1
Scaling zeppelin-controller down to 0
Update succeeded. Deleting old controller: zeppelin-controller
Renaming zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b to zepp
elin-controller
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](#)

Kubernetes应用管理--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   AGE
quieting-warthog-mysql             Pending   1s
```

```
==> v1/Service
NAME                                CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
quieting-warthog-mysql             10.3.253.105  <none>         3306/TCP   1s

==> extensions/v1beta1/Deployment
NAME                                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
quieting-warthog-mysql             1          1          1              0            1s

NOTES:
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
quieting-warthog-mysql.default.svc.cluster.local

To get your root password run:

    kubectl get secret --namespace default quieting-warthog-mysql -o
    jsonpath="{.data.mysql-root-password}" | base64 --decode; echo

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=Never
    -- bash -il

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[Helm命令参考](#)。

Helm工作原理

见[Helm工作原理](#)。

链接

Helm文档

- <https://github.com/kubernetes/helm>
- <https://github.com/kubernetes/charts>

第三方Helm repository

- <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/mgoodness/kube-prometheus-charts>
- <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

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	STATUS
CHART	NAMESPACE		
quieting-warthog	1	Tue Feb 21 20:13:02 2017	DEPLOYED
mysql-0.2.5	default		

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
CAPACITY	ACCESSMODES	AGE
quieting-warthog-mysql	Bound	pvc-90af9bf9-f80d-11e6-930a-42010af00102
8Gi	RWO	9m

```
==> v1/Service
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
quieting-warthog-mysql	10.3.253.105	<none>	3306/TCP	9m

```
==> extensions/v1beta1/Deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
quieting-warthog-mysql	1	1	1	1	9m

NOTES:

MySQL can be accessed via port 3306 on the following DNS name from within your cluster:

```
quieting-warthog-mysql.default.svc.cluster.local
```

To get your root password run:

```
kubectl get secret --namespace default quieting-warthog-mysql -o
jsonpath="{.data.mysql-root-password}" | base64 --decode; echo
```

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=Never
-- bash -il
```

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.storage.g
oogleapis.com/

# 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命令参考

completion	Generate bash autocompletions script
create	create a new chart with the given name
delete	given a release name, delete the release from Kubernetes
dependency	manage a chart's dependencies
fetch	download a chart from a repository and (optionally) unpack it in local directory
get	download a named release
history	fetch release history
home	displays the location of HELM_HOME
init	initialize Helm on both client and server
inspect	inspect a chart
install	install a chart archive
lint	examines a chart for possible issues
list	list releases
package	package a chart directory into a chart archive
repo	add, list, remove, update, and index chart repositories
reset	uninstalls Tiller from a cluster
rollback	roll back a release to a previous revision
search	search for a keyword in charts
serve	start a local http web server
status	displays the status of the named release
test	test a release
upgrade	upgrade a release
verify	verify that a chart at the given path has been signed and is valid
version	print the client/server version information

Flags:

--debug	enable verbose output
--home string	location of your Helm config. Overrides \$HELM_HOME (default "~/helm")
--host string	address of tiller. Overrides \$HELM_HOST
--kube-context string	name of the kubeconfig context to use
--tiller-namespace string	namespace of tiller (default "kube-system")

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Dockerfile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 --name=nginx-ingress
# 等待ingress controller配置完成，并记下外网IP
$ kubectl --namespace kube-system get services -w nginx-ingress-nginx-ingress-controller
```

[info] 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":"feisky@email.com"}' | base64)
# 注意修改registry.org和basedomain
$ draft init --set registry.url=docker.io,registry.org=feisky,registry.authtoken=${token},basedomain=app.feisky.xyz
```

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      r
requirements.txt
$ 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: sha256:5178d22192c2b8b4e1140a3bae9021ee0e808d754b4310014745c1
1f03fcc61b
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:de7e97d0d889b4cdb81ae4b97
2097d759c59e06e
....
de7e97d0d889b4cdb81ae4b972097d759c59e06e: digest: sha256:7ee10c1a56ce
d4f854e7934c9d4a1722d331d7e9bf8130c1a01d6adf7aed6238 size: 2840
--> 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!
```


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。



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Service/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	AGE
etcd-operator-334633986-3nzk1	1/1	Running	0	31s

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(S)
example-etcd-cluster            None             <none>            2379/TCP,
2380/TCP
example-etcd-cluster-client    10.105.90.190   <none>            2379/TCP

# kubectl get pod
NAME                                READY    STATUS    RESTARTS   AGE
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](#)

相关示例

- <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

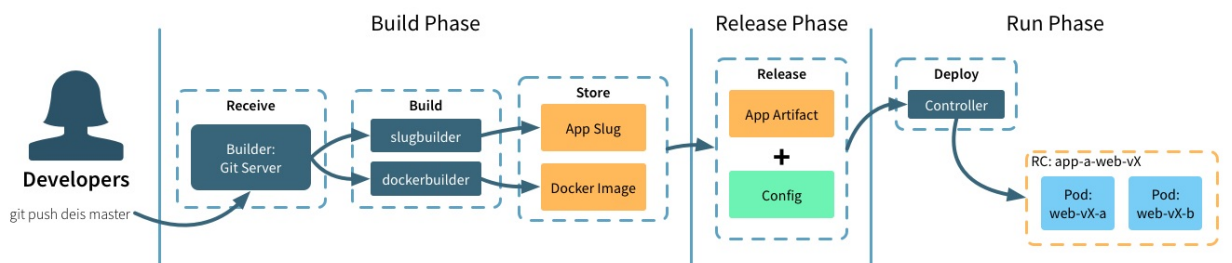
- **StatefulSets**：StatefulSets为有状态服务提供了DNS、持久化存储等，而Operator可以自动处理服务失效、备份、重配置等复杂的场景。
- **Puppet**：Puppet是一个静态配置工具，而Operator则可以实时、动态地保证应用处于预期状态
- **Helm**：Helm是一个打包工具，可以将多个应用打包到一起部署，而Operator则可以认为是Helm的补充，用来动态保证这些应用的正常运行

参考资料

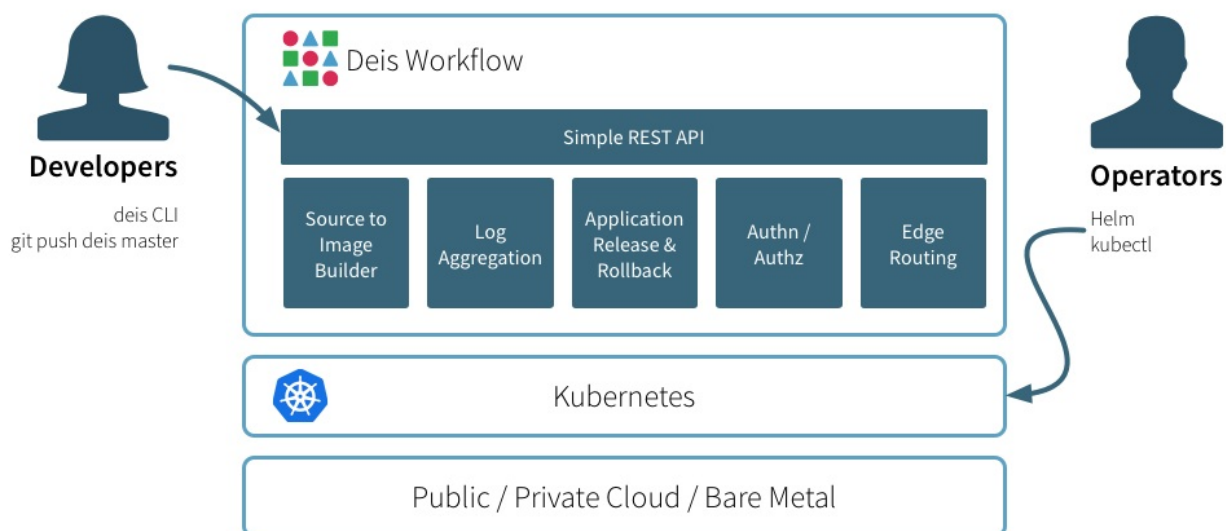
- [Kubernetes Operators](#)

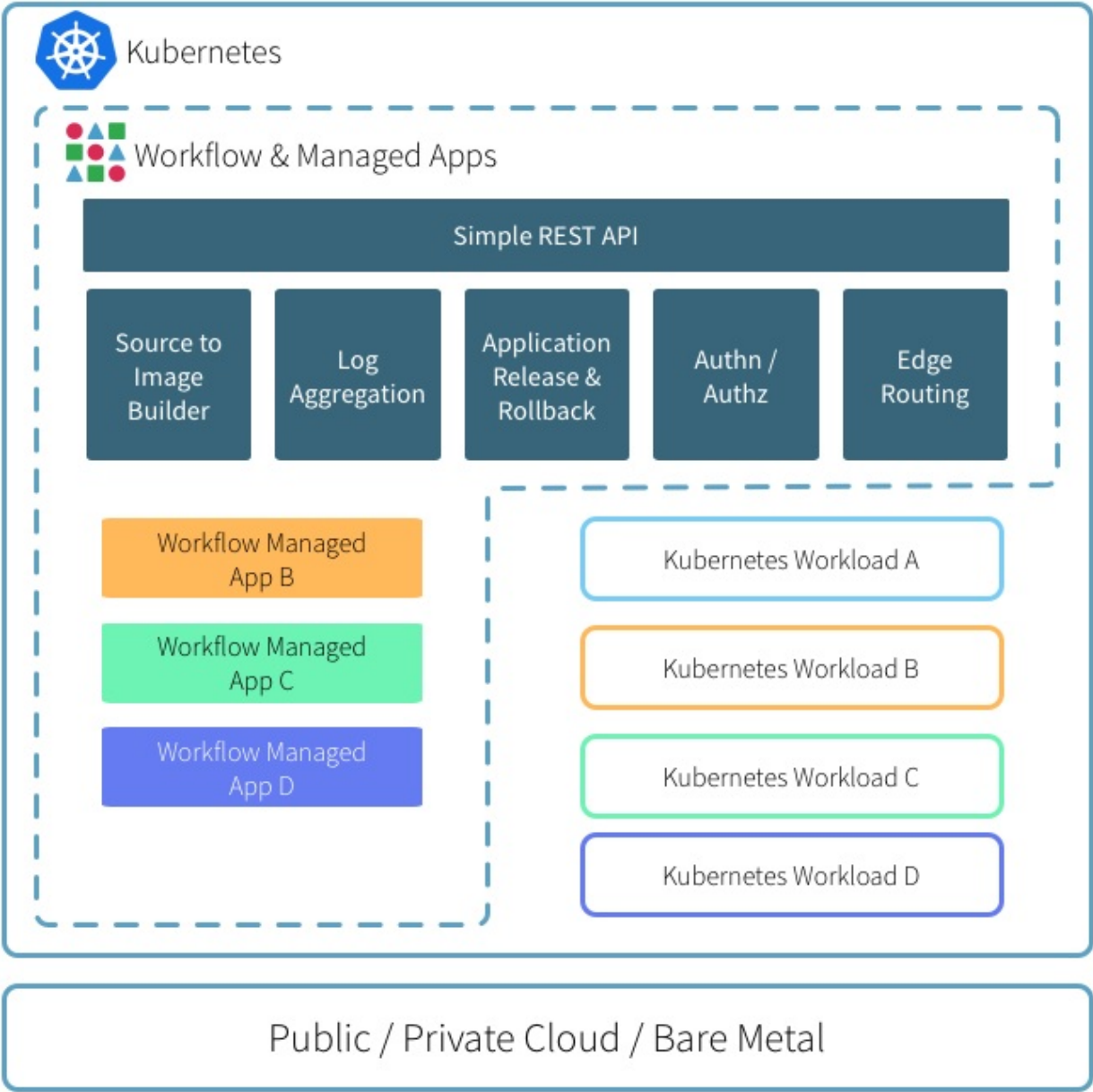
Deis workflow

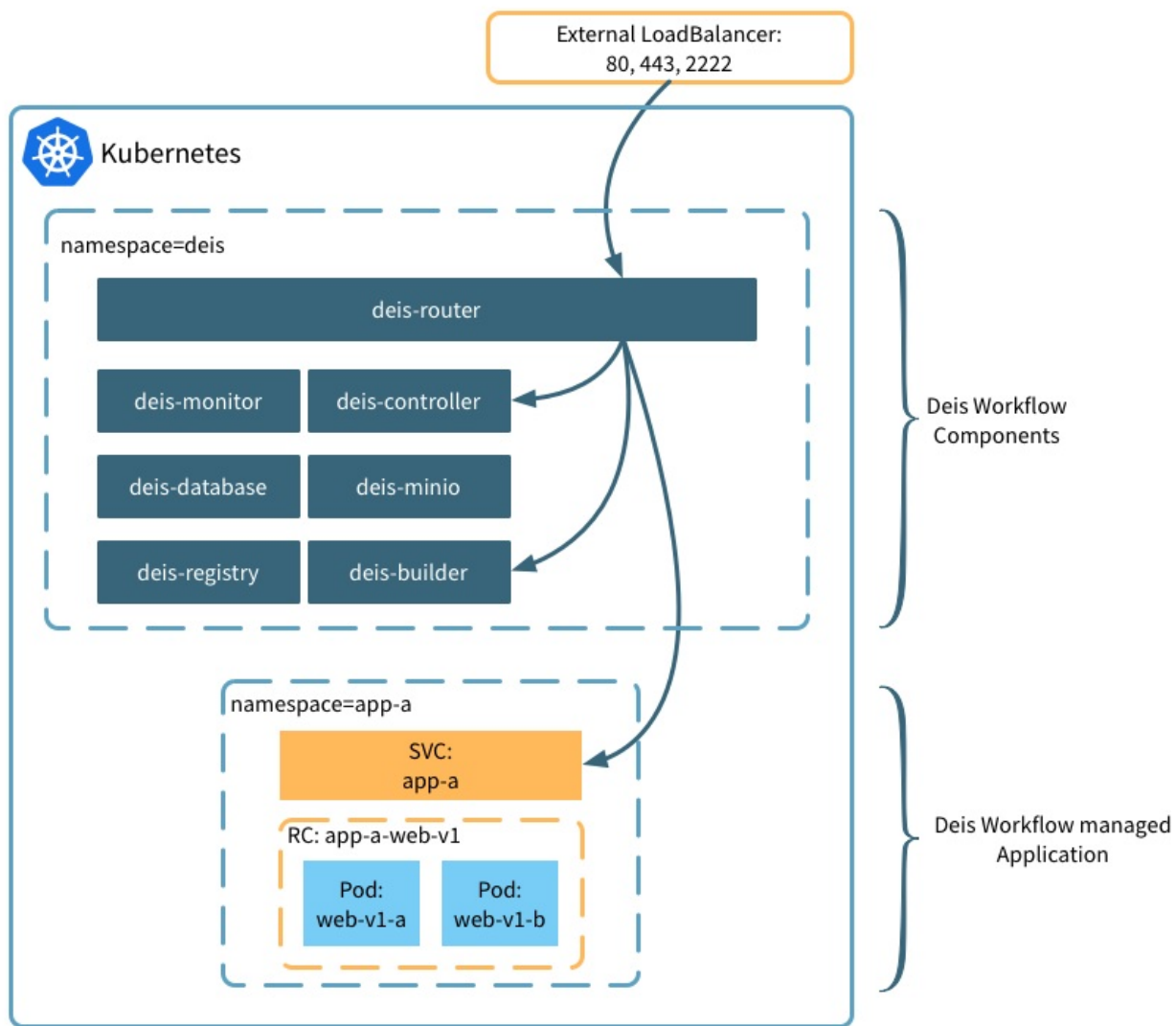
Deis workflow是基于Kubernetes的PaaS管理平台，进一步简化了应用的打包、部署和服务发现。



Deis 架构







Deis安装部署

首先需要部署一套kubernetes（比如minikube，GKE等，记得启用 `KUBE_ENABLE_CLUSTER_DNS=true`），并配置好本机的kubectl客户端，然后运行以下脚本安装deis：


```
# install deis v2 (workflow)
curl -sSL http://deis.io/deis-cli/install-v2.sh | bash
mv deis /usr/local/bin/

# install helm
wget https://storage.googleapis.com/kubernetes-helm/helm-v2.2.1-linux-amd64.tar.gz
tar zxvf helm-v2.2.1-linux-amd64.tar.gz
mv linux-amd64/helm /usr/local/bin/
rm -rf linux-amd64 helm-v2.2.1-linux-amd64.tar.gz
helm init

# deploy helm components
helm repo add deis https://charts.deis.com/workflow
helm install deis/workflow --namespace deis
kubectl --namespace=deis get pods
```

Deis 基本使用

注册用户并登录

```
deis register deis-controller.deis.svc.cluster.local
deis login deis-controller.deis.svc.cluster.local
deis perms:create newuser --admin
```

部署应用

注意，**deis**的大部分操作命令都需要在应用的目录中（即下面的 **example-dockerfile-http**）。

```
git clone https://github.com/deis/example-dockerfile-http.git
cd example-dockerfile-http
docker build -t deis/example-dockerfile-http .
docker push deis/example-dockerfile-http

# create app
deis create example-dockerfile-http --no-remote
# deploy app
deis pull deis/example-dockerfile-http:latest

# query application status
deis info
```

扩展应用

```
$ deis scale cmd=3
$ deis ps
=== example-dockerfile-http Processes
--- cmd:
example-dockerfile-http-cmd-4246296512-08124 up (v2)
example-dockerfile-http-cmd-4246296512-401fv up (v2)
example-dockerfile-http-cmd-4246296512-fx3w3 up (v2)
```

也可以配置自动扩展

```
deis autoscale:set example-dockerfile-http --min=3 --max=8 --cpu-percent=75
```

这样，就可以通过Kubernetes的DNS来访问应用了（配置了外网负载均衡后，还可以通过负载均衡来访问服务）：

```
$ curl example-dockerfile-http.example-dockerfile-http.svc.cluster.local
Powered by Deis
```

域名和路由

```
# 注意设置CNMAE记录到原来的地址
deis domains:add hello.bacongobbler.com

dig hello.deisapp.com
deis routing:enable
```

这实际上是在deis-router的nginx配置中增加了 virtual hosts :

```
server {
    listen 8080;
    server_name ~^example-dockerfile-http\.(?<domain>.+)$;
    server_name_in_redirect off;
    port_in_redirect off;
    set $app_name "example-dockerfile-http";
    vhost_traffic_status_filter_by_set_key example-dockerfile-http application::*;

    location / {
        proxy_buffering off;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $access_scheme;
        proxy_set_header X-Forwarded-Port $forwarded_port;
        proxy_redirect off;
        proxy_connect_timeout 30s;
        proxy_send_timeout 1300s;
        proxy_read_timeout 1300s;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

        proxy_pass http://10.0.0.224:80;
    }
}

server {
    listen 8080;
    server_name hello.bacongobbler.com;
    server_name_in_redirect off;
    port_in_redirect off;
```

```
    set $app_name "example-dockerfile-http";
    vhost_traffic_status_filter_by_set_key example-dockerfile-http
application::*;

    location / {
        proxy_buffering off;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $access_scheme;
        proxy_set_header X-Forwarded-Port $forwarded_port;
        proxy_redirect off;
        proxy_connect_timeout 30s;
        proxy_send_timeout 1300s;
        proxy_read_timeout 1300s;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_pass http://10.0.0.224:80;
    }
}
```

参考文档

- <https://github.com/deis/workflow>
- <https://deis.com/workflow/>

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/releases/download/v0.5.0/kompose-linux-amd64 -o kompose

# macOS
$ curl -L https://github.com/kubernetes-incubator/kompose/releases/download/v0.5.0/kompose-darwin-amd64 -o kompose

# Windows
$ curl -L https://github.com/kubernetes-incubator/kompose/releases/download/v0.5.0/kompose-windows-amd64.exe -o kompose.exe

# 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 PersistentVolumeClaims **for** your Dockerized application.

If you need different kind of resources, use the **'kompose convert'** and **'kubectl create -f'** commands instead.

```
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 **'kubectl get deployment,svc,pods,pvc'** **for** details.

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
```

Istio

Istio是Google、IBM和Lyft联合开源的微服务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的主要特性包括：

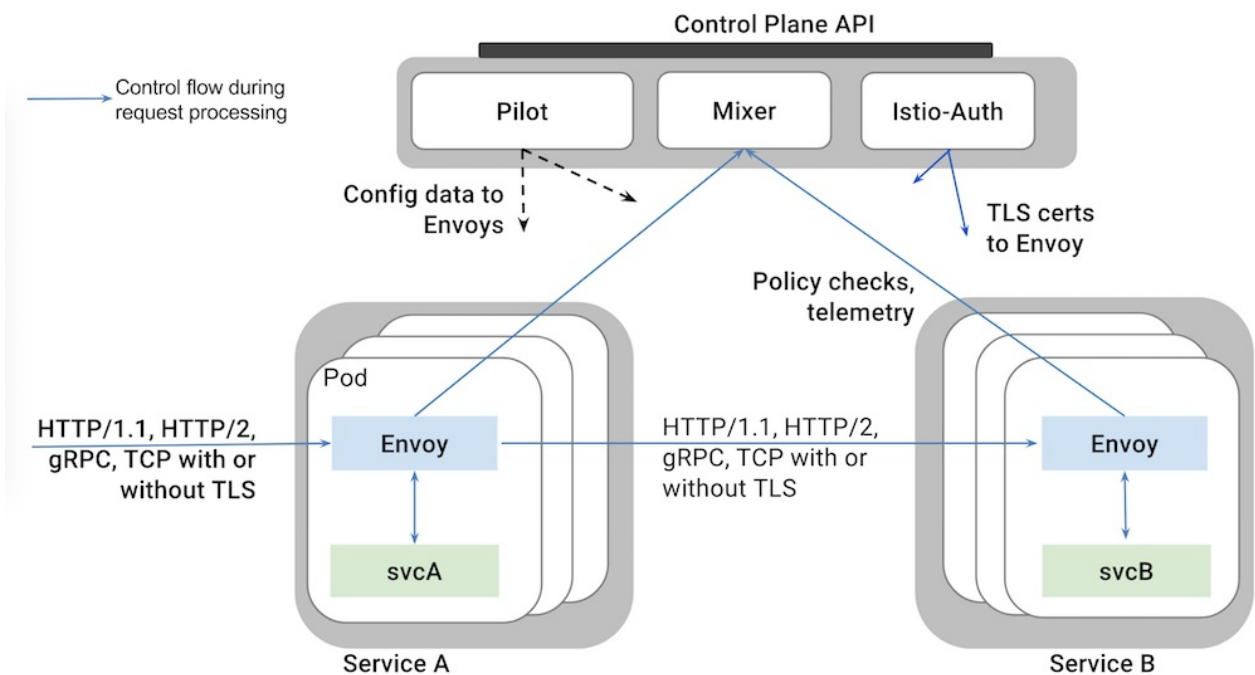
- HTTP、gRPC和TCP网络流量的自动负载均衡
- 丰富的路由规则，细粒度的网络流量行为控制
- 流量加密、服务间认证，以及强身份声明
- 全范围（Fleet-wide）策略执行
- 深度遥测和报告

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（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这些智能代理，并动态执行策略

Istio架构可以如下图所示



主要由以下组件构成

- **Envoy**：Lyft开源的高性能代理总线，支持动态服务发现、负载均衡、TLS终止、

HTTP/2和gPRC代理、健康检查、性能测量等功能。Envoy以sidecar的方式部署在相关的服务的Pod中。

- **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
- **Pilot**：用户和Istio的接口，验证用户提供的配置和路由策略并发送给Istio组件，管理Envoy示例的生命周期
- **Istio-Auth**：提供服务间和终端用户的认证机制

安装

Istio目前仅支持Kubernetes，在部署Istio之前需要先部署好Kubernetes集群并配置好kubectl客户端。

下载Istio

```
curl -L https://git.io/getIstio | sh -  
cd istio-0.1.6/  
cp bin/istioctl /usr/local/bin/
```

创建RBAC角色和绑定

```
$ kubectl apply -f install/kubernetes/istio-rbac-beta.yaml  
clusterrole "istio-pilot" created  
clusterrole "istio-ca" created  
clusterrole "istio-sidecar" created  
rolebinding "istio-pilot-admin-role-binding" created  
rolebinding "istio-ca-role-binding" created  
rolebinding "istio-ingress-admin-role-binding" created  
rolebinding "istio-sidecar-role-binding" created
```

如果碰到下面的错误

```
Error from server (Forbidden): error when creating "install/kubernetes/istio-rbac-beta.yaml": clusterroles.rbac.authorization.k8s.io "istio-pilot" is forbidden: attempt to grant extra privileges: [[[*] [istio.io] [istioconfigs] [] []] {[*] [istio.io] [istioconfigs.istio.io] [] []}] {[*] [extensions] [thirdpartyresources] [] []}] {[*] [extensions] [thirdpartyresources.extensions] [] []}] {[*] [extensions] [ingresses] [] []}] {[*] [] [configmaps] [] []}] {[*] [] [endpoints] [] []}] {[*] [] [pods] [] []}] {[*] [] [services] [] []}] user=&{user@example.org [...]
```

需要给用户授予admin权限

```
kubectl create clusterrolebinding myname-cluster-admin-binding --clusterrole=cluster-admin --user=myname@example.org
```

部署Istio核心服务

两种方式（选择其一执行）

- 禁止Auth： `kubectl apply -f install/kubernetes/istio.yaml`
- 启用Auth： `kubectl apply -f install/kubernetes/istio-auth.yaml`

部署Prometheus、Grafana和Zipkin插件

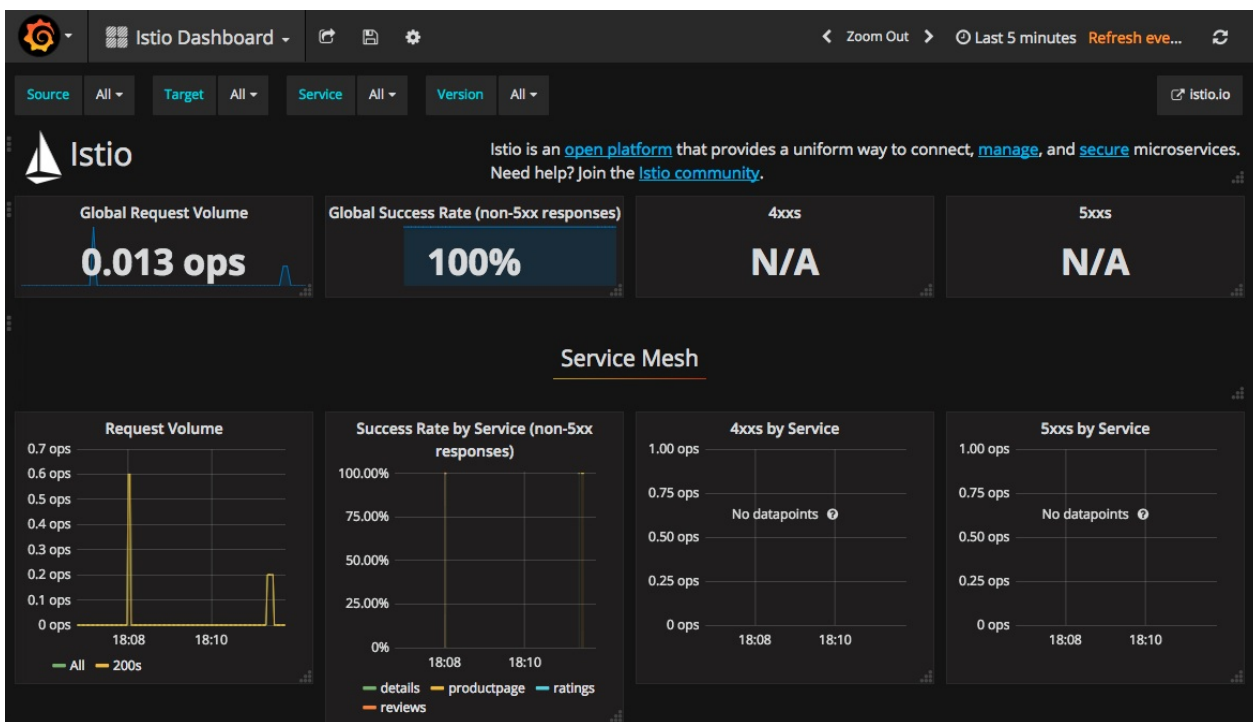
```
kubectl apply -f install/kubernetes/addons/prometheus.yaml
kubectl apply -f install/kubernetes/addons/grafana.yaml
kubectl apply -f install/kubernetes/addons/servicegraph.yaml
kubectl apply -f install/kubernetes/addons/zipkin.yaml
```

等一会所有Pod启动后，可以通过NodePort或负载均衡服务的外网IP来访问这些服务。
比如通过NodePort方式，先查询服务的NodePort

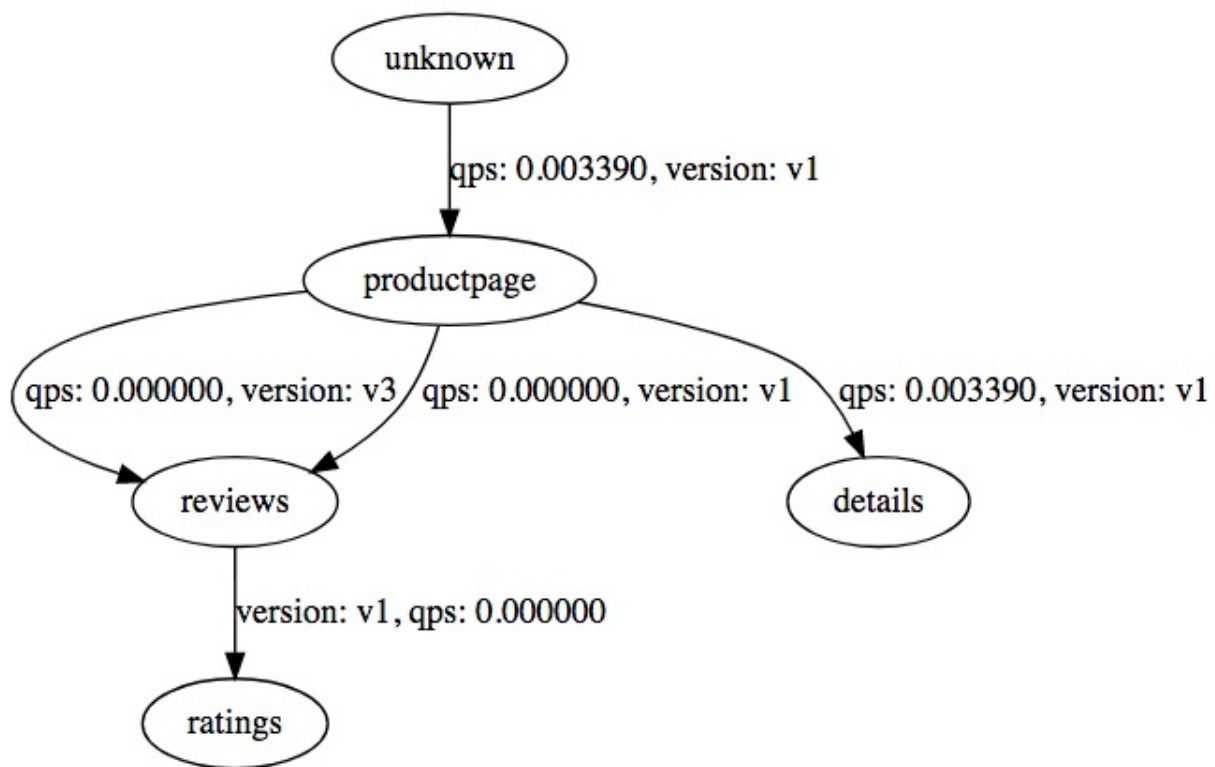
```
$ kubectl get svc grafana -o jsonpath='{.spec.ports[0].nodePort}'
32070
$ kubectl get svc servicegraph -o jsonpath='{.spec.ports[0].nodePort}'

31072
$ kubectl get svc zipkin -o jsonpath='{.spec.ports[0].nodePort}'
30032
$ kubectl get svc prometheus -o jsonpath='{.spec.ports[0].nodePort}'
30890
```

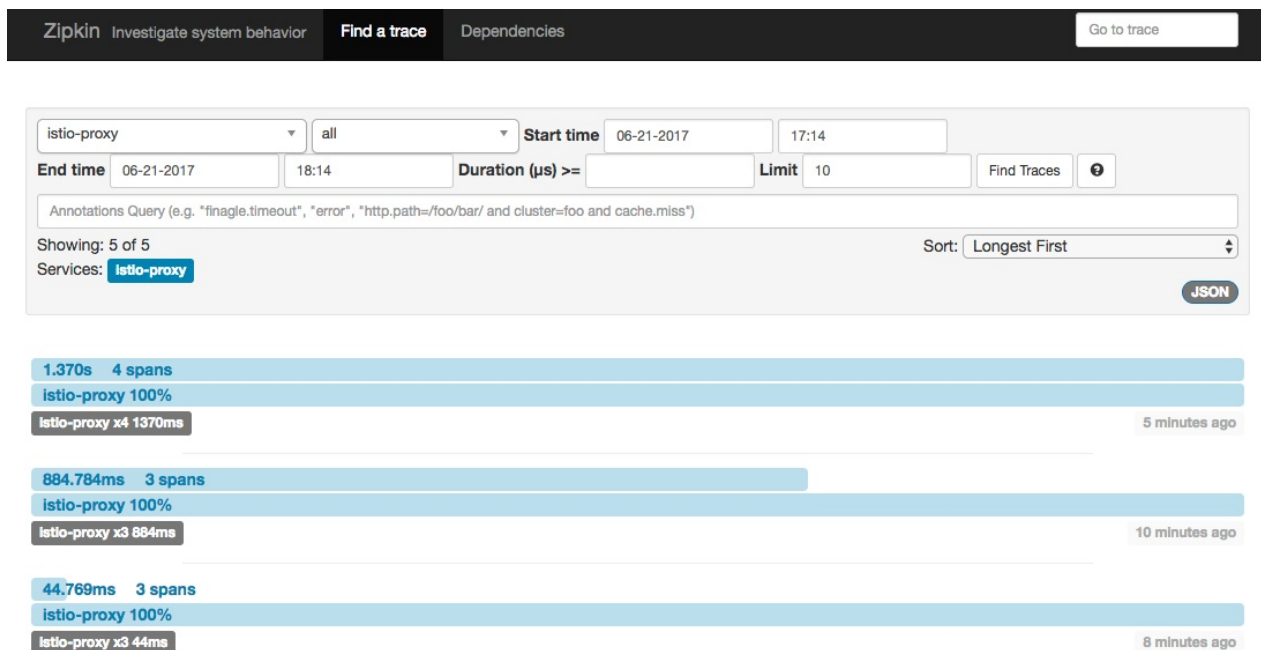
通过 `http://<kubernetes-ip>:32070/dashboard/db/istio-dashboard` 访问 Grafana 服务



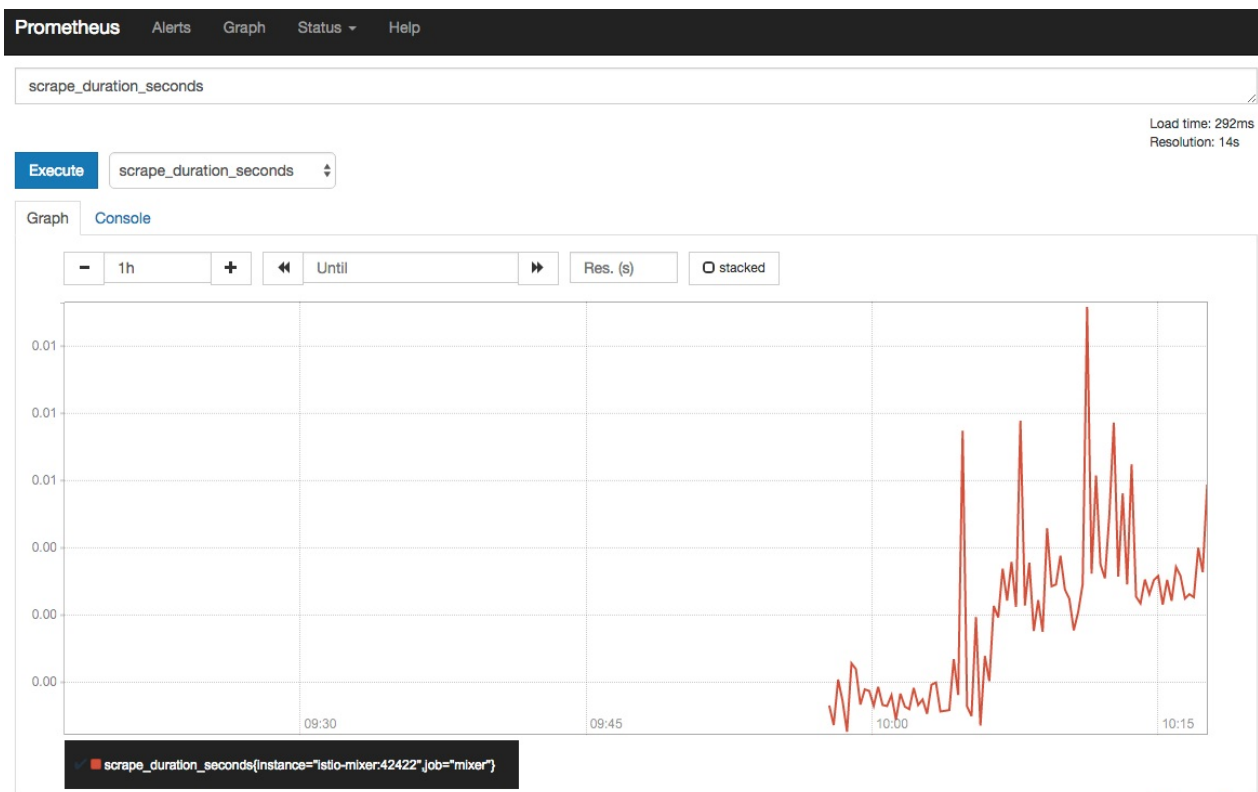
通过 `http://<kubernetes-ip>:31072/dotviz` 访问 ServiceGraph 服务，展示服务之间调用关系图



通过 <http://<kubernetes-ip>:30032> 访问Zipkin跟踪页面



通过 <http://<kubernetes-ip>:30890> 访问Prometheus页面



部署示例应用

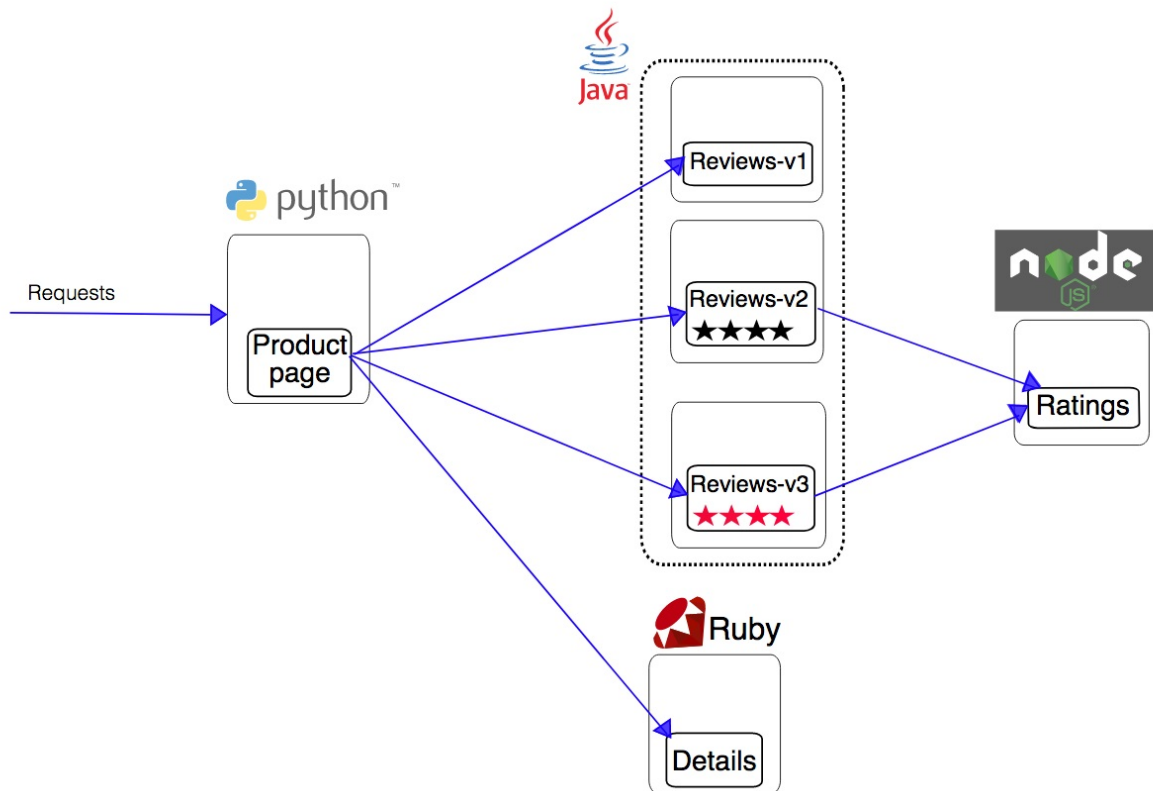
在部署应用时，需要通过 `istioctl kube-inject` 给Pod自动插入Envoy容器，即

```
kubectl create -f <(istioctl kube-inject -f <your-app-spec>.yaml)
```

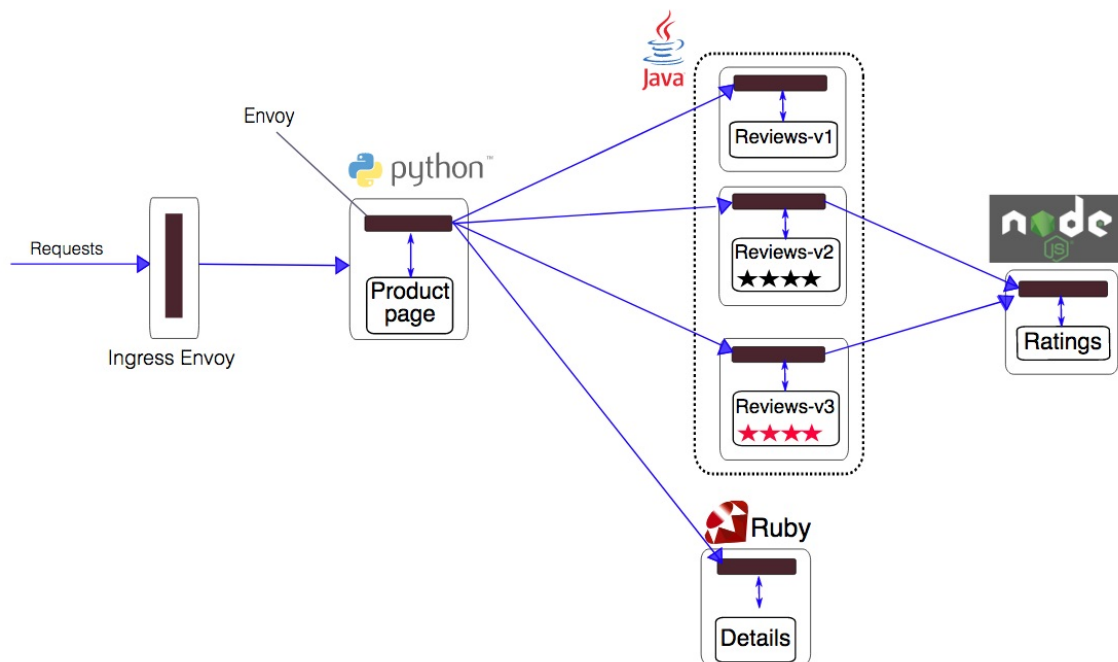
比如Istio提供的BookInfo示例：

```
kubectl apply -f <(istioctl kube-inject -f samples/apps/bookinfo/bookinfo.yaml)
```

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Ingress地址 `http://<ingress-address>/productpage` 来访问BookInfo应用

```
$ kubectl describe ingress
Name:          gateway
Namespace:     default
Address:       192.168.0.77
Default backend: default-http-backend:80 (10.8.0.4:8080)
Rules:
  Host      Path      Backends
  ----      -
  *
            /productpage    productpage:9080 (<none>)
            /login      productpage:9080 (<none>)
            /logout     productpage:9080 (<none>)
Annotations:
Events:      <none>
```

BookInfo Sample

Sign in

The Comedy of Errors

Wikipedia Summary: The Comedy of Errors is one of **William Shakespeare's** early plays. It is his shortest and one of his most farcical comedies, with a major part of the humour coming from slapstick and mistaken identity, in addition to puns and word play.

Book Details

Paperback:
200 pages
Publisher:
PublisherA
Language:
English
ISBN-10:
1234567890
ISBN-13:
123-1234567890

An extremely entertaining play by Shakespeare. The slapstick humour is refreshing!

— Reviewer1 Affiliation1

★★★★★

Absolutely fun and entertaining. The play lacks thematic depth when compared to other plays by Shakespeare.

— Reviewer2 Affiliation2

★★★★☆

参考文档

- <https://istio.io/>
- [Istio - A modern service mesh](#)
- <https://lyft.github.io/envoy/>

Kubernetes 实践

Kubernetes 实践及常用技巧。

Kubernetes 监控

[info] Kubernetes 监控注意事项

- tags和labels变的非常重要，是应用分组的重要依据
- 多维度的监控，包括集群内的所有节点、容器、容器内的应用以及Kubernetes自身
- 天生的分布式，对复杂应用的监控聚合是个挑战

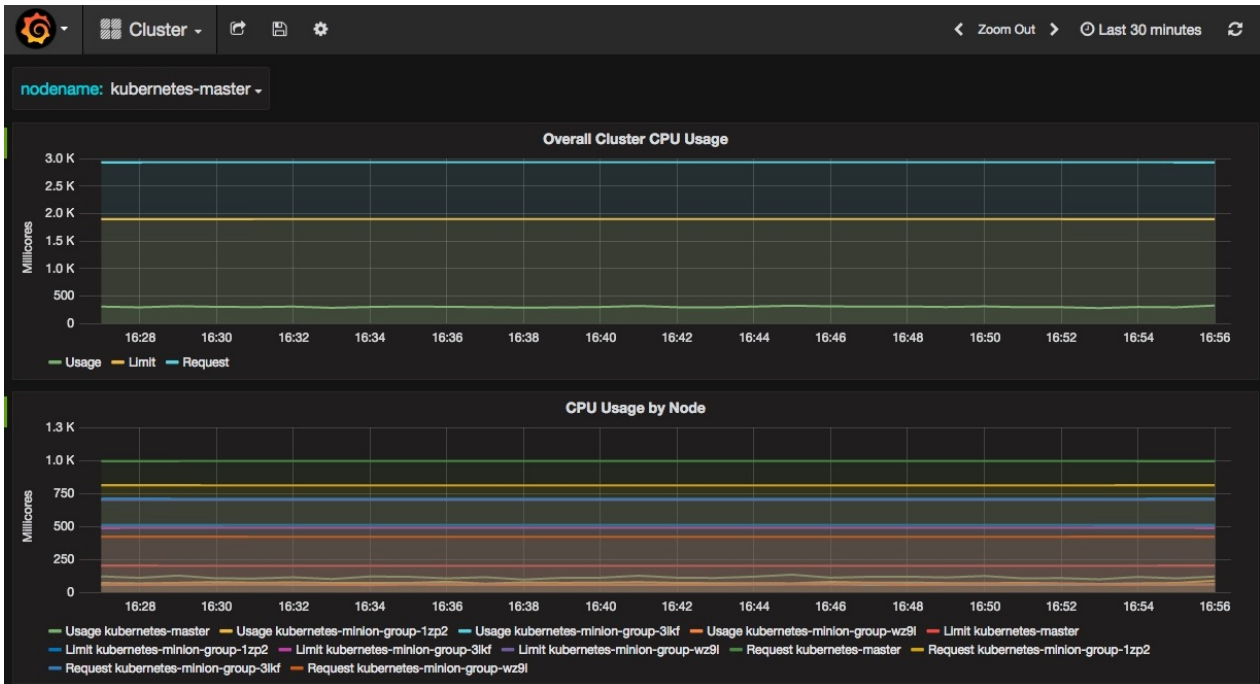
cAdvisor

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 and 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`）。



InfluxDB和Grafana

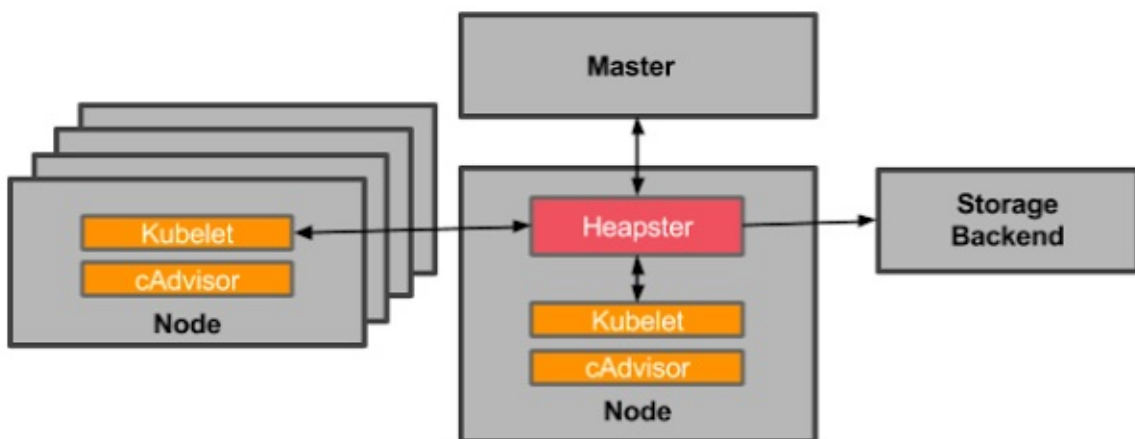
InfluxDB是一个开源分布式时序、事件和指标数据库；而Grafana则是InfluxDB的dashboard，提供了强大的图表展示功能。它们常被组合使用展示图表化的监控数据。



Heapster

前面提到的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。

Heapster从kubelet提供的API采集节点和容器的资源占用：



另外，Heapster的 `/metrics` API提供了Prometheus格式的数据。

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/namespaces/kube-system/services/heapster
KubeDNS is running at https://kubernetes-master/api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://kubernetes-master/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
Grafana is running at https://kubernetes-master/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
InfluxDB is running at https://kubernetes-master/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb
```

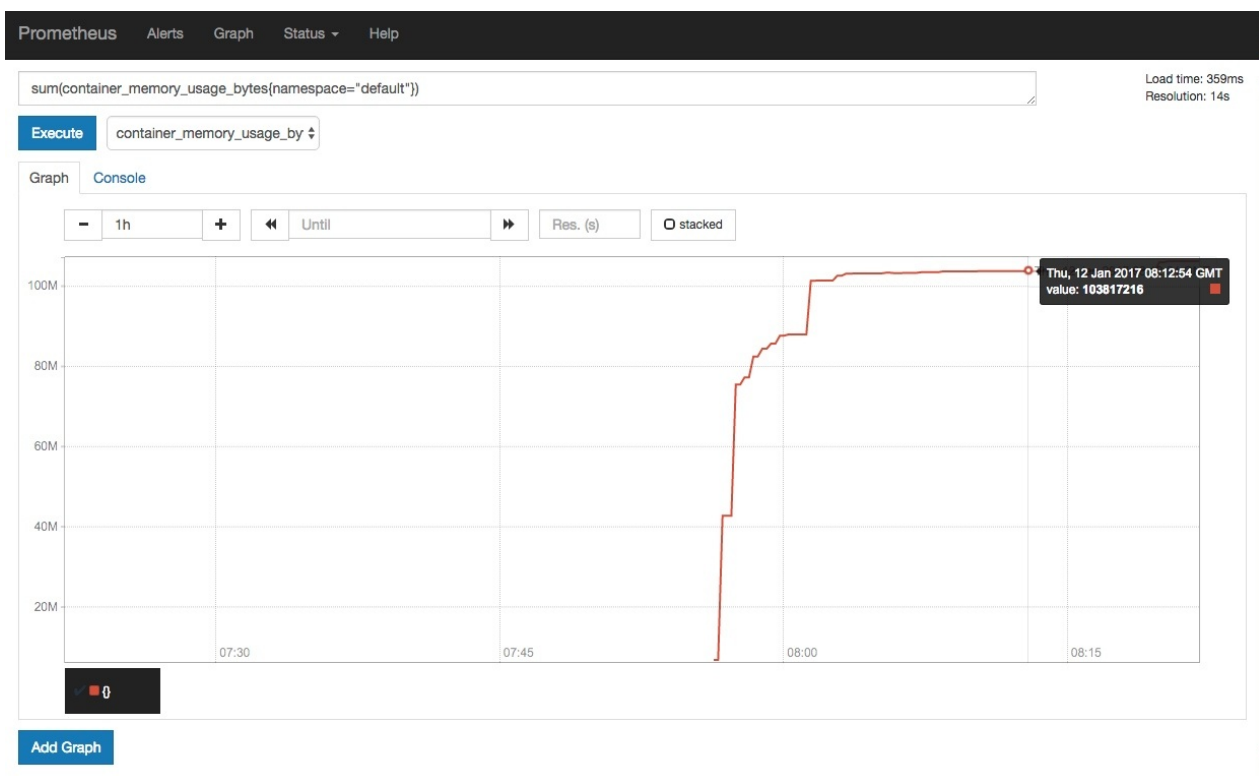
如果这些服务没有自动部署的话，可以根据[cluster/addons](#)来添加需要的服务。

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。

使用Prometheus监控Kubernetes需要配置好数据源，一个简单的示例是[prometheus.yml](#)：

```
kubectl create -f http://feisky.xyz/kubernetes/monitor/prometheus.txt
```



推荐使用[Prometheus Operator](#)或[Prometheus Chart](#)来部署和管理Prometheus。

其他容器监控系统

- [Sysdig](#)
- [CoScale](#)
- [Datadog](#)
- [Sematext](#)

Kubernetes 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。

应用日志收集

在进行日志收集的过程中，我们首先想到的是使用Logstash，因为它是ELK stack中的重要成员，但是在测试过程中发现，Logstash是基于JDK的，在没有产生日志的情况单纯启动Logstash就大概要消耗**500M**内存，在每个Pod中都启动一个日志收集组件的情况下，使用logstash有点浪费系统资源，经人推荐我们选择使用**Filebeat**替代，经测试单独启动Filebeat容器大约会消耗**12M**内存，比起logstash相当轻量级。

方案选择

Kubernetes官方提供了EFK的日志收集解决方案，但是这种方案并不适合所有的业务场景，它本身就有一些局限性，例如：

- 所有日志都必须是out前台输出，真实业务场景中无法保证所有日志都在前台输出
- 只能有一个日志输出文件，而真实业务场景中往往有多个日志输出文件
- Fluentd并不是常用的日志收集工具，我们更习惯用logstash，现使用filebeat替代
- 我们已经有自己的ELK集群且有专人维护，没有必要再在kubernetes上做一个日志收集服务

基于以上几个原因，我们决定使用自己的ELK集群。

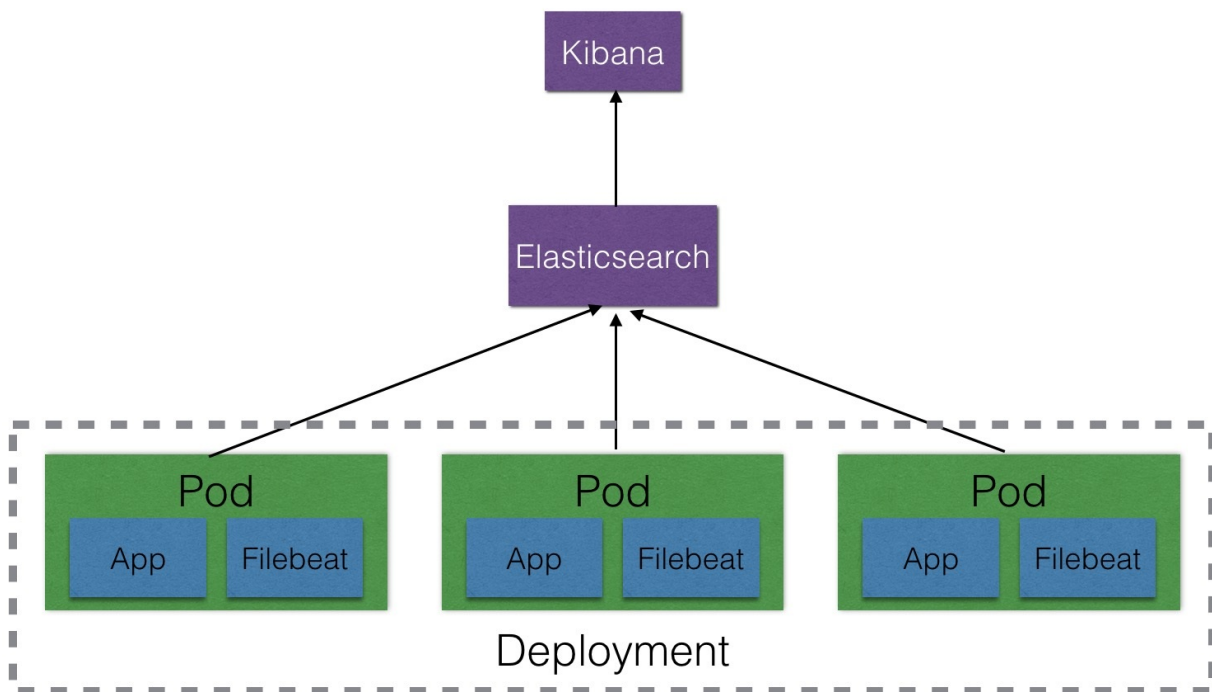
Kubernetes集群中的日志收集解决方案

编号	方案	优点	缺点
1	每个app的镜像中都集成日志收集组件	部署方便，kubernetes的yaml文件无须特别配置，可以为每个app自定义日志收集配置	强耦合，不方便应用和日志收集组件升级和维护且会导致镜像过大
2	单独创建一个日志收集组件跟app的容器一起运行在同一个pod中	低耦合，扩展性强，方便维护和升级	需要对kubernetes的yaml文件进行单独配置，略显繁琐
3	将所有的Pod的日志都挂载到宿主机上，每台主机上单独起一个日志收集Pod	完全解耦，性能最高，管理起来最方便	需要统一日志收集规则，目录和输出方式

综合以上优缺点，我们选择使用方案二。

该方案在扩展性、个性化、部署和后期维护方面都能做到均衡，因此选择该方案。

日志收集 / filebeat



我们创建了自己的logstash镜像，镜像地址

为：`index.tenxcloud.com/jimmy/filebeat:5.4.0`

测试

我们部署一个应用filebeat来收集日志的功能测试。

创建应用yaml文件 `filebeat-test.yaml`。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: filebeat-test
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
      labels:
        k8s-app: filebeat-test
    spec:
      containers:
        - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/filebea
```

```
t:5.4.0
  name: filebeat
  volumeMounts:
  - name: app-logs
    mountPath: /log
  - name: filebeat-config
    mountPath: /etc/filebeat/
  - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/analyti
cs-docker-test:Build_8
    name : app
    ports:
    - containerPort: 80
    volumeMounts:
    - name: app-logs
      mountPath: /usr/local/TalkingData/logs
  volumes:
  - name: app-logs
    emptyDir: {}
  - name: filebeat-config
    configMap:
      name: filebeat-config
---
apiVersion: v1
kind: Service
metadata:
  name: filebeat-test
  labels:
    app: filebeat-test
spec:
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    run: filebeat-test
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: filebeat-config
data:
  filebeat.yml: |
```



```
filebeat.prospectors:
- input_type: log
  paths:
    - "/log/*"
    - "/log/usermange/common/*"
output.elasticsearch:
  hosts: ["172.23.5.255:9200"]
username: "elastic"
password: "changeme"
index: "filebeat-test"
```

说明

该文件中包含了配置文件filebeat的配置文件的ConfigMap，因此不需要再定义环境变量。

当然你也可以不同ConfigMap，通过传统的传递环境变量的方式来配置filebeat。

例如对filebeat的容器进行如下配置：

```
containers:
- image: sz-pg-oam-docker-hub-001.tendcloud.com/library/filebeat:5.4.0
  name: filebeat
  volumeMounts:
    - name: app-logs
      mountPath: /log
  env:
    - name: PATHS
      value: "/log/*"
    - name: ES_SERVER
      value: 172.23.5.255:9200
    - name: INDEX
      value: logstash-docker
    - name: INPUT_TYPE
      value: log
```

目前使用这种方式会有个问题，及时 `PATHS` 只能传递单个目录，如果想传递多个目录需要修改filebeat镜像的 `docker-entrypoint.sh` 脚本，对该环境变量进行解析增加filebeat.yml文件中的PATHS列表。

推荐使用ConfigMap，这样filebeat的配置就能够更灵活。

注意事项

- 将app的 /usr/local/TalkingData/logs 目录挂载到filebeat的 /log 目录下。
- 该文件可以在 manifests/test/filebeat-test.yaml 找到。
- 我使用了自己的私有镜像仓库，测试时请换成自己的应用镜像。
- Filebeat的环境变量的值配置

```
ENV ES_SERVER 172.23.5.255:9200
ENV INDEX filebeat-test
ENV INPUT_TYPE log
ENV ES_USERNAME elastic
ENV ES_PASSWORD changeme
```

创建应用

部署Deployment

```
kubectl create -f filebeat-test.yaml
```

查看 http://172.23.5.255:9200/_cat/indices 将可以看到列表有这样的indices：

```
green open filebeat-2017.05.17          1qatsSajSYqAV42_XYwLsQ 5 1
1189      0      1mb    588kb
```

访问Kibana的web页面，查看 filebeat-2017.05.17 的索引，可以看到logstash收集到了app日志。

Kibana

Discover filebeat*

Selected Fields

- _source
- _message
- _index

Available Fields

- @timestamp
- _id
- _score
- _type
- beat.hostname
- beat.name
- beat.version
- input_type
- offset
- type

May 17th 2017, 15:41:34.573 - May 17th 2017, 15:56:34.573 — by 30 seconds

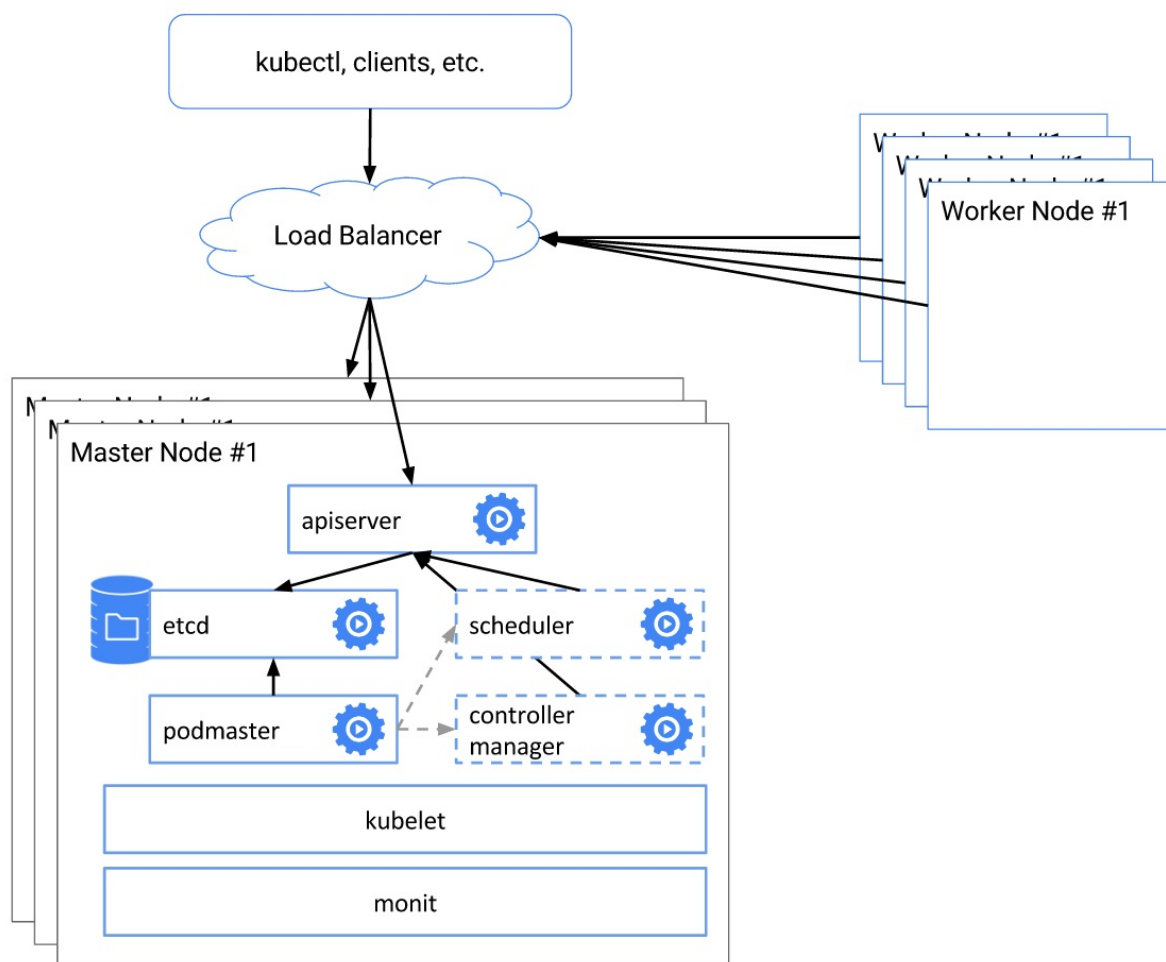
Time	source	message	_index
May 17th 2017, 15:55:13.192	/log/api.1 og	at java.net.Socket.<init>(Socket.java:375) ~[na:1.6.0_26]	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	at java.net.Socket.connect(Socket.java:478) ~[na:1.6.0_26]	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	at com.mysql.jdbc.MySQLIO.<init>(MySQLIO.java:294) ~[mysql-connector-java-5.1.17.jar:na]	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	... 154 common frames omitted	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	at java.net.Socket.<init>(Socket.java:218) ~[na:1.6.0_26]	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	2017-05-17 15:55:08.210 [net.sf.ehcache.CacheManager@1e4218cb] INFO net.sf.ehcache.util.UpdateChecker- New update(s) found: 2.6.5 [http://www.terracotta.org/confluence/display/release/Release+Notes+Ehcache+Core+2.6]. Please check http://ehcache.org for the latest version.	filebeat- 2017.05. 17
May 17th 2017, 15:55:13.192	/log/api.1 og	at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:257) ~[mysql-connector-java-5.1.17.jar:na]	filebeat- 2017.05. 17

Kubernetes HA

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
- apiserver负载均衡
- controller manager、scheduler和cluster autoscaler自动选主（有且仅有一个运行实例）

如下图所示



etcd 集群

从 <https://discovery.etcd.io/new?size=3> 获取token后，把 <https://kubernetes.io/docs/admin/high-availability/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}`，既可以由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。

apiserver

把<https://kubernetes.io/docs/admin/high-availability/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等为master节点配置负载均衡。

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=10.245.0.0/16 --allocate-node-cidrs=true --service-account-private-key-file=/srv/kubernetes/server.key --v=2 --leader-elect=true
```

把[kube-scheduler.yaml](#)和[kube-controller-manager.yaml](#)(非GCE平台需要适当修改)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Kubernetes v1.6还不支持IPVS模式）。

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

- <https://kubernetes.io/docs/admin/high-availability/>
- <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](#)

调试运行中的容器

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:webserver --cap-add sys_admin --cap-add sys_ptrace alpine sh
/ # 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/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.5/community/x86_64/APKINDEX.tar.gz
v3.5.1-34-g1d3b13bd53 [http://dl-cdn.alpinelinux.org/alpine/v3.5/main]
v3.5.1-29-ga981b1f149 [http://dl-cdn.alpinelinux.org/alpine/v3.5/community]
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
<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 (LISTEN)
```

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 webserver
```

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的一个子功能，通过 `kubectl port-forward` 可以将本地端口转发到指定的Pod。

使用方法

```
# Listen on ports 5000 and 6000 locally, forwarding data to/from ports 5000 and 6000 in the pod
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
```

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。

在Kubernetes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

使用方法

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 device_l
ib.list_local_devices()
    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: 0000:0
0:04.0"
]
```

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

```
NVIDIA_GPU_NAME=$(nvidia-smi --query-gpu=gpu_name --format=csv,nohead  
er --id=0)  
source /etc/default/kubelet  
KUBELET_OPTS="$KUBELET_OPTS --node-labels='alpha.kubernetes.io/nvidia  
-gpu-name=$NVIDIA_GPU_NAME'"  
echo "KUBELET_OPTS=$KUBELET_OPTS" > /etc/default/kubelet
```

然后，在创建Pod时设置Node Affinity

```
kind: pod  
apiVersion: v1  
metadata:  
  annotations:  
    scheduler.alpha.kubernetes.io/affinity: >  
    {  
      "nodeAffinity": {  
        "requiredDuringSchedulingIgnoredDuringExecution": {  
          "nodeSelectorTerms": [  
            {  
              "matchExpressions": [  
                {  
                  "key": "alpha.kubernetes.io/nvidia-gpu-name",  
                  "operator": "In",  
                  "values": ["Tesla K80", "Tesla P100"]  
                }  
              ]  
            }  
          ]  
        }  
      }  
    }  
spec:  
  containers:  
    - image: gcr.io/tensorflow/tensorflow:latest-gpu  
      name: gpu-container-1  
      command: ["python"]  
      args: ["-u", "-c", "import tensorflow"]  
      resources:  
        limits:  
          alpha.kubernetes.io/nvidia-gpu: 2
```


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 Unc
orr. ECC |
| Fan   Temp   Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Co
mpute M. |
|=====+=====+=====+
+-----+
|    0   Tesla K80           Off   | 0000:00:04.0     Off  |
|    0   |
| N/A    74C    P0      80W / 149W |      0MiB / 11439MiB |    100%
Default |
+-----+-----+-----+
+-----+
+-----+
| Processes:                                                       GP
U Memory |
| GPU       PID    Type    Process name                       Us
age      |
|=====+=====+=====+
+-----+
| 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/ubuntu1604/x86_64/cuda-repo-ubuntu1604_8.0.61-1_amd64.deb
    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/lib64/libcudnn*
```

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 Unc
orr. ECC |
| Fan   Temp   Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Co
mpute M. |
|=====+=====+=====+
=====|
|    0   Tesla K80           Off   | 0000:00:04.0     Off  |
|    0 |
| N/A    74C    P0      80W / 149W |      0MiB / 11439MiB |      100%
Default |
+-----+-----+-----+
+-----+

+-----+
+-----+
| Processes:                                     GP
U Memory |
| GPU       PID    Type    Process name                       Us
age      |
|=====+=====+=====+
=====|
| No running processes found
|
+-----+
+-----+

```

容器安全

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

请参考[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还在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=1
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.min_p
mtu=1000,kernel.msgmax=1 2 3
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 $\geq$ 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/<container_name>` 来设置。
可选的值包括

- `runtime/default` : 使用Container Runtime的默认配置
- `localhost/<profile_name>` : 使用已加载到内核的AppArmor profile

```
$ sudo apparmor_parser -q <<EOF
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
    #include <abstractions/base>

    file,

    # Deny all file writes.
    deny /** w,
}
EOF'

$ kubectl create -f /dev/stdin <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-deny-write
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo '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 non-zero exit code: Error executing in Docker Container: 1
```

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/<container name>` : 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/<profile-name>` : 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
```

Kubernetes 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-b2f7-4ba9b1e43b53" ip="127.0.0.1" method="GET" user="admin" groups="\system:masters\", \"system:authenticated\" as "<self>" asgroups="<lookup>" namespace="default" uri="/api/v1/namespaces/default/pods"
2017-03-21T03:57:09.108403639-04:00 AUDIT: id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" response="200"
```

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-policy.yaml
```

其中，设计策略的配置格式为

```
rules:
  # Don't log watch requests by the "system:kube-proxy" on endpoints
  or services
  - 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 paths.
  - 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" namespace.
      # The empty string "" can be used to select non-namespaced resources.
      namespaces: ["kube-system"]
```

```
# Log configmap and secret changes in all other namespaces at the M
etadata level.
- level: Metadata
  resources:
  - group: "" # core API group
    resources: ["secrets", "configmaps"]

# Log all other resources in core and extensions at the Request lev
el.
- level: Request
  resources:
  - group: "" # core API group
  - group: "extensions" # Version of group should NOT be included.

# A catch-all rule to log all other requests at the Metadata level.
- 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-ec8
ef575696d" stage="RequestReceived" ip="10.2.1.3" method="update" user
="system:serviceaccount:kube-system:default" groups="\system:service
accounts\","\system:serviceaccounts:kube-system\","\system:authentica
ted\" as="<self>" asgroups="<lookup>" namespace="kube-system" uri="/
api/v1/namespaces/kube-system/endpoints/kube-controller-manager" resp
onse="<deferred>"
2017-06-15T21:50:50.259470834Z AUDIT: id="591e9fde-6a98-46f6-b7bc-ec8
ef575696d" stage="ResponseComplete" ip="10.2.1.3" method="update" use
r="system:serviceaccount:kube-system:default" groups="\system:servic
eaccounts\","\system:serviceaccounts:kube-system\","\system:authentica
ted\" as="<self>" asgroups="<lookup>" namespace="kube-system" uri="
/api/v1/namespaces/kube-system/endpoints/kube-controller-manager" res
ponse="200"
```

- 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 the
remote service.
      server: https://audit.example.com/audit # URL of remote service
to query. Must use 'https'.

# 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 webhook pl
ugin to use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API server.
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/namespaces/kube-public/rolebindings/system:controller:bootstrap-signer",
      "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
      }
    }
  ]
}
```

Kubernetes开发环境

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
# 由于社区还没有验证最新版本的Docker，因而这里安装一个较老一点的版本
sh -c 'echo "deb https://apt.dockerproject.org/repo ubuntu-$(lsb_release -cs) main" > /etc/apt/sources.list.d/docker.list'
curl -fsSL https://apt.dockerproject.org/gpg | sudo apt-key add -
apt-key fingerprint 58118E89F3A912897C070ADBF76221572C52609D
apt-get update
apt-get -y install "docker-engine=1.13.1-0~ubuntu-$(lsb_release -cs)"

# 安装etcd
curl -L https://github.com/coreos/etcd/releases/download/v3.1.8/etcd-v3.1.8-linux-amd64.tar.gz -o etcd-v3.1.8-linux-amd64.tar.gz && tar xzvf etcd-v3.1.8-linux-amd64.tar.gz && /bin/cp -f etcd-v3.1.8-linux-amd64/{etcd,etcdctl} /usr/bin && rm -rf etcd-v3.1.8-linux-amd64*

# 安装Go
curl -sL https://storage.googleapis.com/golang/go1.8.3.linux-amd64.tar.gz | tar -C /usr/local -zxvf -
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/k8s.io/kubernetes
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kubectI之后就可以开始使用了：

```
export KUBECONFIG=/var/run/kubernetes/admin.kubeconfig
cluster/kubect1.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
```

e2e测试

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\sfoward
ing'
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Feature:Secur
ityContext'
```

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，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PR拉取到本地来测试，比如拉取PR #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PR非常多，这个课程可能会很慢）：

```
fetch = +refs/pull/*:refs/remotes/origin/pull/*
```

其他参考

- 编译release版： `make quick-release`
- 测试命令：[kubernetes test-infra](#)。
- [Kubernetes TestGrid](#)，包含所有的测试历史
- [Kuberentes Submit Queue Status](#)，包含所有的PR状态以及正在合并的PR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PR列表（需要github登录）
- [Jenkins Logs](#)和[Prow Status](#)，包含所有PR的jenkins测试日志

Kubernetes 测试

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 flag set
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_ARGS='
-run ^TestValidatePod$'

# Runs tests that match the regex ValidatePod|ValidateConfigMap in pkg/api/validation
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_ARGS='
-run ValidatePod\|ValidateConfigMap$'
```

或者直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/api/validation -run ^TestValidatePod
$
```

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 # Ad
d to PATH
```

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 the verbose flag set.
make test-integration KUBE_GOFLAGS="-v" KUBE_TEST_ARGS="-run ^TestPodUpdateActiveDeadlineSeconds$"
```

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 down
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\scli\n\t\s\[k8s\.io\]\sKubectl\srolling\-\update\sshould\s\support\srolling\-\update\sto\s\same\simage\s\[Conformance\]'
```

跳过测试用例

```
go run hack/e2e.go -- -v --test --test_args="--ginkgo.skip=Pods.*env"
```

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="--ginkgo
.skip=\[Serial\]"

# Run tests in parallel, skip any that must be run serially and keep
the test namespace if test failed
GINKGO_PARALLEL=y go run hack/e2e.go -- -v --test --test_args="--ginkgo
.skip=\[Serial\] --delete-namespace-on-failure=false"
```

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 cluster.
Useful for
# cleaning up after a failed test or viewing logs. Use -v to avoid su
ppressing
# kubectl output.
go run hack/e2e.go -- -v -ctl='get events'
go run hack/e2e.go -- -v -ctl='delete pod foobar'
```

Federation 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 -- -v
-build

# 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=\[Feature
:Federation\]"

# Don't forget to teardown everything down
go run hack/e2e.go -- -v --down
```

可以用 `cluster/log-dump.sh <directory>` 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={{if (exists . "status" "containerStatuses")}}{{range .status.containerStatuses}}{{if eq .name "nginx"}}{{.image}}{{end}}{{end}}{{end}}'
```

参考文档

- [Kubernetes testing](#)
- [End-to-End Testing](#)
- [Node e2e test](#)
- [How to write e2e test](#)
- [Coding Conventions](#)

Kubernetes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。

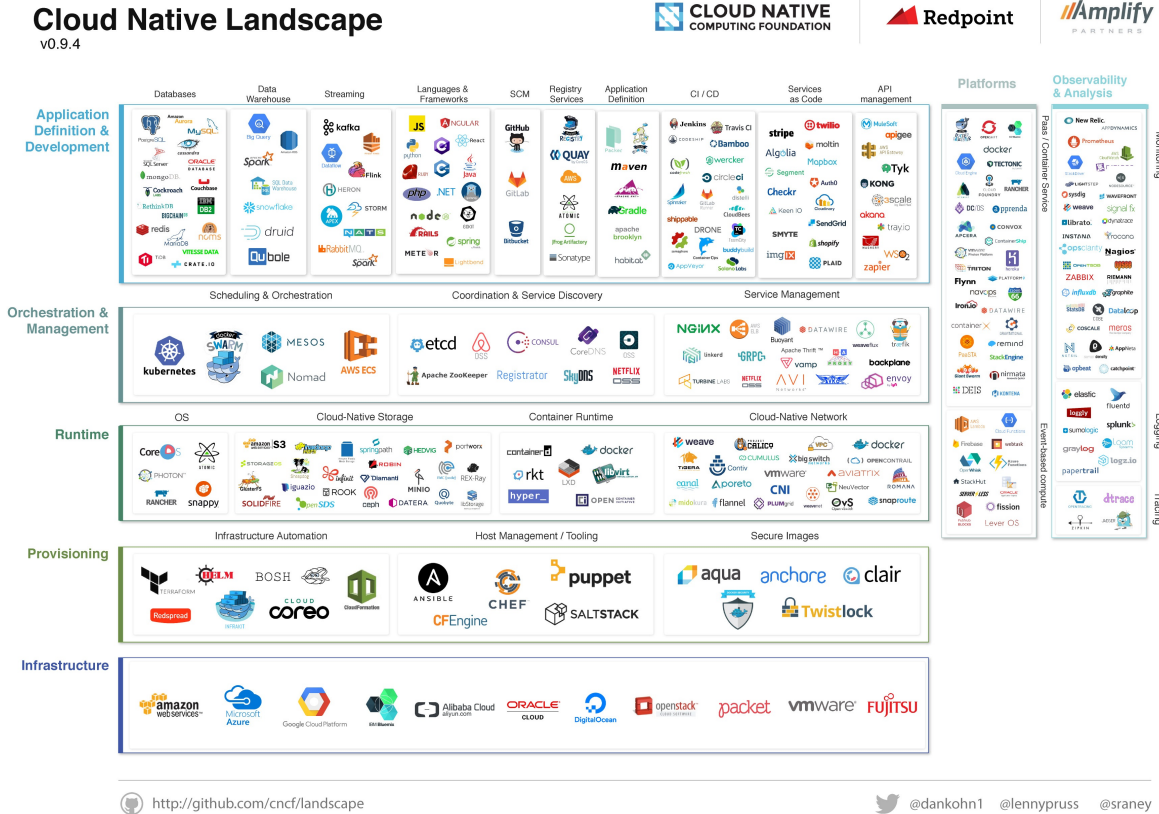
如果在社区贡献中碰到问题，可以参考以下指南

- [Contributing guidelines](#)
- [Kubernetes Developer Guide](#)
- [Special Interest Groups](#)
- [Feature Tracking and Backlog](#)
- [Community Expectations](#)

附录

参考文档以及一些实用的资源链接。

Kubernetes ecosystem



- <http://kubernetes.io/partners/>
- K8s distributions and SaaS offerings
 - <http://openshift.com>
 - <https://tectonic.com/>
 - <http://rancher.com/kubernetes/>
 - <https://www.infoq.com/news/2016/11/apprenda-kubernetes-ket>
 - <https://github.com/samsung-cnct/kraken>
 - <https://www.mirantis.com/solutions/container-technologies/>
 - <https://www.ubuntu.com/cloud/kubernetes>
 - <https://platform9.com/products-2/kubernetes/>
 - <https://kubermatic.io/en/>
 - <https://stackpoint.io/#/>
 - <http://gravitational.com/telekube/>
 - <https://kcluster.io/>
 - <http://www.stratoscale.com/products/kubernetes-as-a-service/>

- <https://giantswarm.io/product/>
- <https://cloud.google.com/container-engine/>
- <https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes>
- <http://blog.kubernetes.io/2016/11/bringing-kubernetes-support-to-azure.html>
- <http://thenewstack.io/huawei-launches-kubernetes-based-container-engine/>
- <http://blogs.univa.com/2016/05/univa-announces-navops-command-for-managing-enterprise-container-workload-on-kubernetes-distributions/>
- <https://supergiant.io/>
- <https://diamanti.com/products/>
- <http://www.vmware.com/company/news/releases/vmw-newsfeed-VMware-Introduces-Kubernetes-as-a-Service-on-Photon-Platform.2104598.html>
- <http://mantl.io/>
- <https://github.com/hyperhq/hypernetes>
- <https://www.joyent.com/containerpilot>
- PaaS on Kubernetes
 - Openshift
 - Deis Workflow
 - Gondor/Kel
 - WSO2
 - Rancher
 - Kumoru
- Serverless implementations
 - Funktion
 - [Fission](#)
 - Kubeless
 - OpenWhisk
 - Iron.io
- Application frameworks
 - [Spring Cloud](#)
- API Management
 - Apigee
 - [Kong](#)
 - Apiman
- Data processing
 - Pachyderm
 - Heron

- Package managers
 - Helm
 - [KPM](#)
- Configuration
 - Kompose
 - Jsonnet
 - [Spread](#)
 - [K8comp](#)
 - [Ktmpl](#)
 - [Konfd](#)
 - [kenv](#)
 - [kubediff](#)
 - [Habitat](#)
 - [Puppet](#)
 - [Ansible](#)
- Application deployment orchestration
 - [ElasticKube](#)
 - [AppController](#)
 - [Broadway](#)
 - [Kb8or](#)
 - [IBM UrbanCode](#)
 - [nulecule](#)
 - [Deployment manager](#)
- API/CLI adaptors
 - [Kubebot](#)
 - [StackStorm](#)
 - [Kubefuse](#)
 - [Ksql](#)
 - [kubectld](#)
- UIs / mobile apps
 - [Cabin](#)
 - [Cockpit](#)
- CI/CD
 - [Jenkins plugin](#)
 - Wercker
 - Shippable
 - GitLab

- [cloudmunch](#)
 - [Kontinuous](#)
 - [Kit](#)
 - [Spinnaker](#)
- Developer platform
 - [Fabric8](#)
 - [Spring Cloud integration](#)
 - [goPaddle](#)
 - [VAMP](#)
- Secret generation and management
 - [Vault controller](#)
 - [kube-lego](#)
 - [k8sec](#)
- [Client libraries](#)
- Autoscaling
 - [Kapacitor](#)
- Monitoring
 - Sysdig
 - Datadog
 - Sematext
 - Prometheus
 - Snap
 - [Satellite](#)
 - [Netsil](#)
 - [Weave Scope](#)
 - [AppFormix](#)
- Logging
 - Sematext
 - [Sumo Logic](#)
- RPC
 - Grpc
 - [Micro](#)
- Load balancing
 - Nginx Plus
 - Traefik
 - Service mesh
 - Envoy

- Linkerd
- Amalgam8
- WeaveWorks
- Networking
 - WeaveWorks
 - Tigera
 - [OpenContrail](#)
 - Nuage
 - [Kuryr](#)
 - [Contiv](#)
- Storage
 - Flocker
 - [Portworx](#)
 - REX-Ray
 - [Torus](#)
 - Hedvig
 - [Quobyte](#)
 - [NetApp](#)
 - [Datera](#)
 - [Ceph](#)
 - [Gluster](#)
- Database/noSQL
 - [CockroachDB](#)
 - [Cassandra](#) / [DataStax](#)
 - [MongoDB](#)
 - [Hazelcast](#)
 - [Crate](#)
 - [Vitess](#)
 - [Minio](#)
- Container runtimes
 - containerd
 - Rkt
 - CRI-O (OCI)
 - Hyper.sh/frakti
- Security
 - [Tirame](#)
 - [Aquasec](#)

- [Twistlock](#)
- Appliances
 - Diamanti
 - Redapt
- Cloud providers
 - GKE/GCE
 - AWS
 - Azure
 - Digital Ocean
 - CenturyLink
 - Rackspace
 - VMWare
 - Openstack
 - Cloudstack
- Managed Kubernetes
 - Platform9
 - Gravitational
 - [KCluster](#)
- VMs on Kubernetes
 - Openstack
 - Redhat
- Other
 - [Netflix OSS](#)
 - [Kube-monkey](#)
 - [Kubecraft](#)

Play with Kubernetes

[Play with Kubernetes](#)提供了一个免费的Kubernetes体验环境，每次创建的集群最长可以使用4小时。

集群初始化

打开[play-with-k8s.com](#)，点击"ADD NEW INSTANCE"，并在新INSTANCE（名字为node1）的TERMINAL中初始化kubernetes master：

```
$ kubeadm init --apiserver-advertise-address $(hostname -i)
Initializing machine ID from random generator.
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production clusters.
[init] Using Kubernetes version: v1.7.0
[init] Using Authorization modes: [Node RBAC]
[preflight] Skipping pre-flight checks
[certificates] Generated CA certificate and key.
[certificates] Generated API server certificate and key.
[certificates] API Server serving cert is signed for DNS names [node1
  kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 10.0.1.3]
[certificates] Generated API server kubelet client certificate and key.
[certificates] Generated service account token signing key and public key.
[certificates] Generated front-proxy CA certificate and key.
[certificates] Generated front-proxy client certificate and key.
[certificates] Valid certificates and keys now exist in "/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/controller-manager.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/scheduler.conf"
```



```
[apiclient] Created API client, waiting for the control plane to become ready
[apiclient] All control plane components are healthy after 25.001152 seconds
[token] Using token: 35e301.77277e7cafee013c
[apiconfig] Created RBAC rules
[addons] Applied essential addon: kube-proxy
[addons] Applied essential addon: kube-dns
```

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

You should now deploy a pod network to the cluster.

Run `"kubectl apply -f [podnetwork].yaml"` with one of the options listed at:

<http://kubernetes.io/docs/admin/addons/>

You can now join any number of machines by running the following on each node as root:

```
kubeadm join --token 35e301.77277e7cafee013c 10.0.1.3:6443
```

Waiting for api server to startup.....

```
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
daemonset "kube-proxy" configured
```

注意：记住输出中的 `kubeadm join --token 35e301.77277e7cafee013c 10.0.1.3:6443` 命令，后面会用来添加新的节点。

配置 kubectl

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

初始化网络

```
kubectl apply -n kube-system -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

创建Dashboard

```
curl -L -s https://git.io/kube-dashboard | sed 's/targetPort: 9090/targetPort: 9090\n  type: LoadBalancer/' | kubectl apply -f -
```

稍等一会，在页面上方会显示Dashborad服务的端口号，点击端口号就可以访问Dashboard页面。

添加新的节点

点击"ADD NEW INSTANCE"，并在新INSTANCE的TERMINAL中输入前面第一步记住的 `kubeadm join` 命令，如

```
kubeadm join --token 35e301.77277e7cafee013c 10.0.1.3:6443
```

回到node1的TERMINAL，输入 `kubectl get node` 即可查看所有Node的状态。等所有Node状态都变成Ready后，整个Kubernetes集群都搭建好了。

FAQ

1. Failed to start ContainerManager failed to initialise top level QOS containers #43856

重启kubelet时报错，目前的解决方法是：

- 1.在docker.service配置中增加的 `--exec-opt native.cgroupdriver=systemd` 配置。
- 2.手动删除slice（貌似不管用）
- 3.重启主机，这招最管用☺

```
for i in $(systemctl list-unit-files --no-legend --no-pager -l | grep --color=never -o *.slice | grep kubepod);do systemctl stop $i;done
```

上面的几种方法在该bug修复前只有重启主机管用，该bug已于2017年4月27日修复，merge到了master分支，见<https://github.com/kubernetes/kubernetes/pull/44940>

2. High Availability of Kube-apiserver #19816

API server的HA如何实现？或者说这个master节点上的服务 `api-server` 、 `scheduler` 、 `controller` 如何实现HA？目前的解决方案是什么？

目前的解决方案是api-server是无状态的可以启动多个，然后在前端再加一个nginx或者ha-proxy。而scheduler和controller都是直接用容器的方式启动的。

3. Kubelet启动时Failed to start ContainerManager systemd version does not support ability to start a slice as transient unit

CentOS系统版本7.2.1511

kubelet启动时报错systemd版本不支持start a slice as transient unit。

尝试升级CentOS版本到7.3，看看是否可以修复该问题。

与[kubeadm init waiting for the control plane to become ready on CentOS 7.2 with kubeadm 1.6.1 #228](#)类似。

另外有一个使用systemd管理kubelet的[proposal](#)。

4.kube-proxy报错kube-proxy[2241]: E0502 15:55:13.889842 2241 conntrack.go:42] conntrack returned error: error looking for path of conntrack: exec: "conntrack": executable file not found in \$PATH

导致的现象

kubedns启动成功，运行正常，但是service之间无法解析，kubernetes中的DNS解析异常

解决方法

CentOS中安装 `conntrack-tools` 包后重启kubernetes集群即可。

5. Pod stuck in terminating if it has a privileged container but has been scheduled to a node which doesn't allow privilege issue#42568

当pod被调度到无法权限不足的node上时，pod一直处于pending状态，且无法删除pod，删除时一直处于terminating状态。

kubelet中的报错信息

```
Error validating pod kube-keepalived-vip-1p62d_default(5d79ccc0-3173-11e7-bfbd-8af1e3a7c5bd) from api, ignoring: spec.containers[0].securityContext.privileged: Forbidden: disallowed by cluster policy
```

6.PVC中对Storage的容量设置不生效

使用[glusterfs做持久化存储](#)文档中我们构建了PV和PVC，当时给 `glusterfs-nginx` 的PVC设置了8G的存储限额，`nginx-dm` 这个Deployment使用了该PVC，进入该Deployment中的Pod执行dd测试可以看到创建了9个size为1G的block后无法继续创建了，已经超出了8G的限额：

```
$ dd if=/dev/zero of=test bs=1G count=10
dd: error writing 'test': No space left on device
10+0 records in
9+0 records out
10486870016 bytes (10GB) copied, 14.6692s, 715MB/s
```

参考文档

- [Persistent Volume](#)
- [Resource Design Proposals](#)

参考文档

- [Kubernetes官方网站](#)
- [Kubernetes文档](#)
- [Kubernetes course](#)
- [CNCF项目贡献统计](#)
- [Kubernetes github metrics](#)
- [Kubernetes submit queue](#)
- [Github public data](#)
- [Kubernetes the hard way](#)
- [Kubernetes Bootcamp](#)
- [Design patterns for container-based distributed systems](#)
- [Awesome Kubernetes](#)
- [Awesome Docker](#)