

K8S基础与实践

KUBERNETES基础与实践

- Kubernetes简介
- Kubernetes技术架构
- yamlfile介绍
- Kubernetes基本对象
- 常用Workloads Controller
- 配置管理对象



KUBERNETES简介

- 什么是Kubernetes
- Kubernetes有什么好处
- 为什么用Kubernetes

什么是KUBERNETES

Kubernetes的名字来自希腊语，意思是“舵手”或“领航员”。

K8s是将8个字母“**ubernete**”替换为8的缩写。

Kubernetes始于**Google**2014年的一个项目：**Borg**，它是**Google**内部使用的一个超大规模集群管理系统，它基于容器技术，在**Google**内部使用已有超过10年的历史，目的是实现资源管理自动化，以及跨多个**DC**的资源利用率最大化。

而**Kubernetes**可理解为**Borg**的一个开源版本，是一个基于**Docker**容器技术的分布式架构，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专注在以容器为中心的应用上。

Kubernetes是一个开放的开发平台，无论是**Go**、**Java**或**Python**编写的服务，都可以轻松映射为**Kubernetes**的**Service**，并通过标准的**TCP**通信协议进行交互，因此现有系统可以非常容易的迁移到**Kubernetes**平台上。

KUBERNETES有什么好处

如果系统设计遵循了Kubernetes的设计理念与思想，我们不必费心于服务监控和故障处理，使用Kubernetes可以节约大量的开发成本，可以使更多精力集中于业务本身。

另外Kubernetes提供了强大的自动化运维功能，使系统的后期的运维难度和成本大大降低。

Kubernetes具备完善的集群管理能力，包括多层次的安全防护与准入机制、多租户支撑能力、4/7层服务发现、内置负载均衡器、强大的故障发现与自我修复能力、简单易用的服务部署、升级与回滚、优良的调度策略，以及资源配额管理等。

为什么用KUBERNETES

Docker兴起使服务从单机走向集群，而云加速了这一进程。可预见的几年内，会有大量服务或新系统选择它，不管这些系统运行在本地服务器亦或被托管到公有云上。

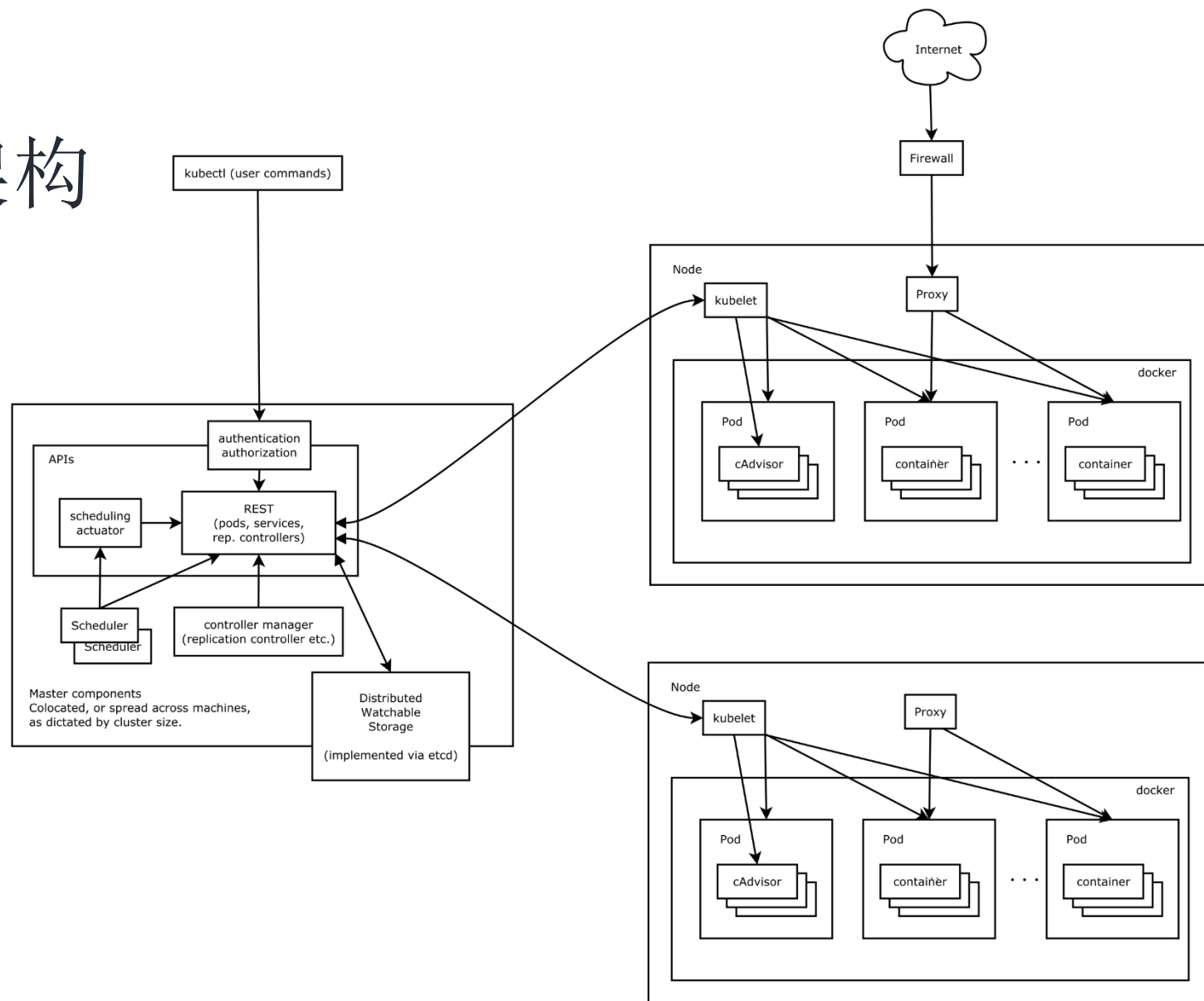
微服务的核心是将巨大的单体应用分解为很多小的互联的微服务，一个微服务背后可能有多个实例副本在支撑运行。每个微服务可以由专门的开发团队或人员开发，开发者可以自由的选择开发技术，这对大规模的团队很有价值，每个微服务可以独立的开发、部署、升级与回滚和扩展，可以使产品具备很高的稳定性，加速产品的迭代。

除了前面对**Kubernetes**的简单介绍之外，**Kubernetes**全面拥抱微服务架构，采用**Kubernetes**可以轻装上阵的开发一个复杂的系统。

举例：以前一个几十人组成的团队且需要不少技术达人一起分工协作才能实现的和运维的分布式系统，在采用**Kubernetes**解决方案之后，只需要一个精悍的小团队就可以轻松应对。在这个团队，一名架构师负责专注于系统中“服务组件”的提炼，几名工程师专注于业务代码的开发，一名系统兼运维工程师负责**Kubernetes**的部署和运维，这并不是我们少做了什么，而是**Kubernetes**帮我们做了很多。

KUBERNETES技术架构

- **Kubernetes**集群包含有节点代理**kubelet**和**Master**组件(APIs, scheduler, etc), 一切都基于分布式的存储系统。下面这张图是**Kubernetes**的架构图。
- 在这张系统架构图中, 我们把服务分为运行在工作节点上的服务和组成集群级别控制板的服务。
- **Kubernetes**节点有运行应用容器必备的服务, 而这些都是受**Master**的控制。
- 每次个节点上当然都要运行**Docker**。**Docker**来负责所有具体的镜像下载和容器运行。



KUBERNETES技术架构

- **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

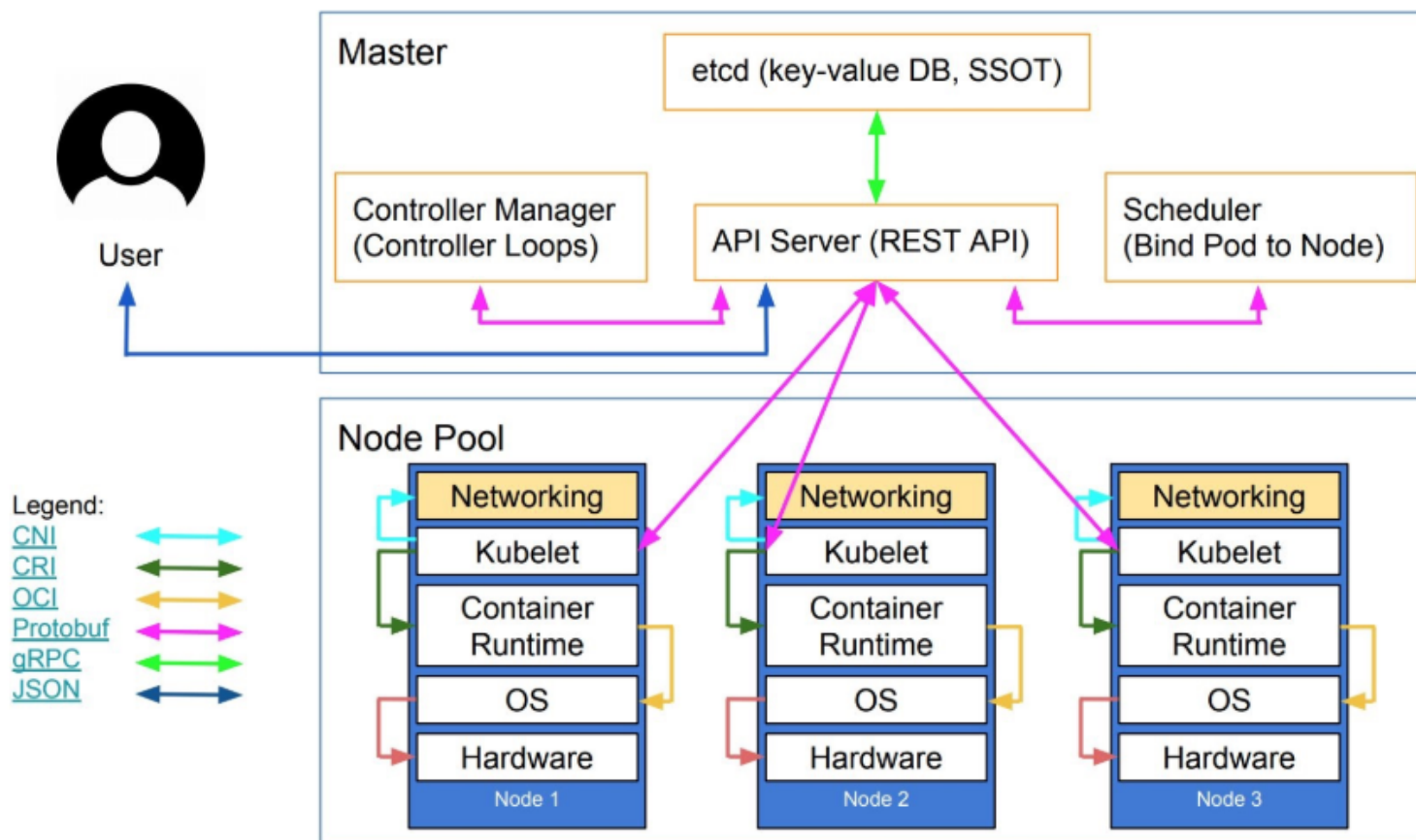
KUBERNETES技术架构

除了核心组件，还有一些推荐的Add-ons：

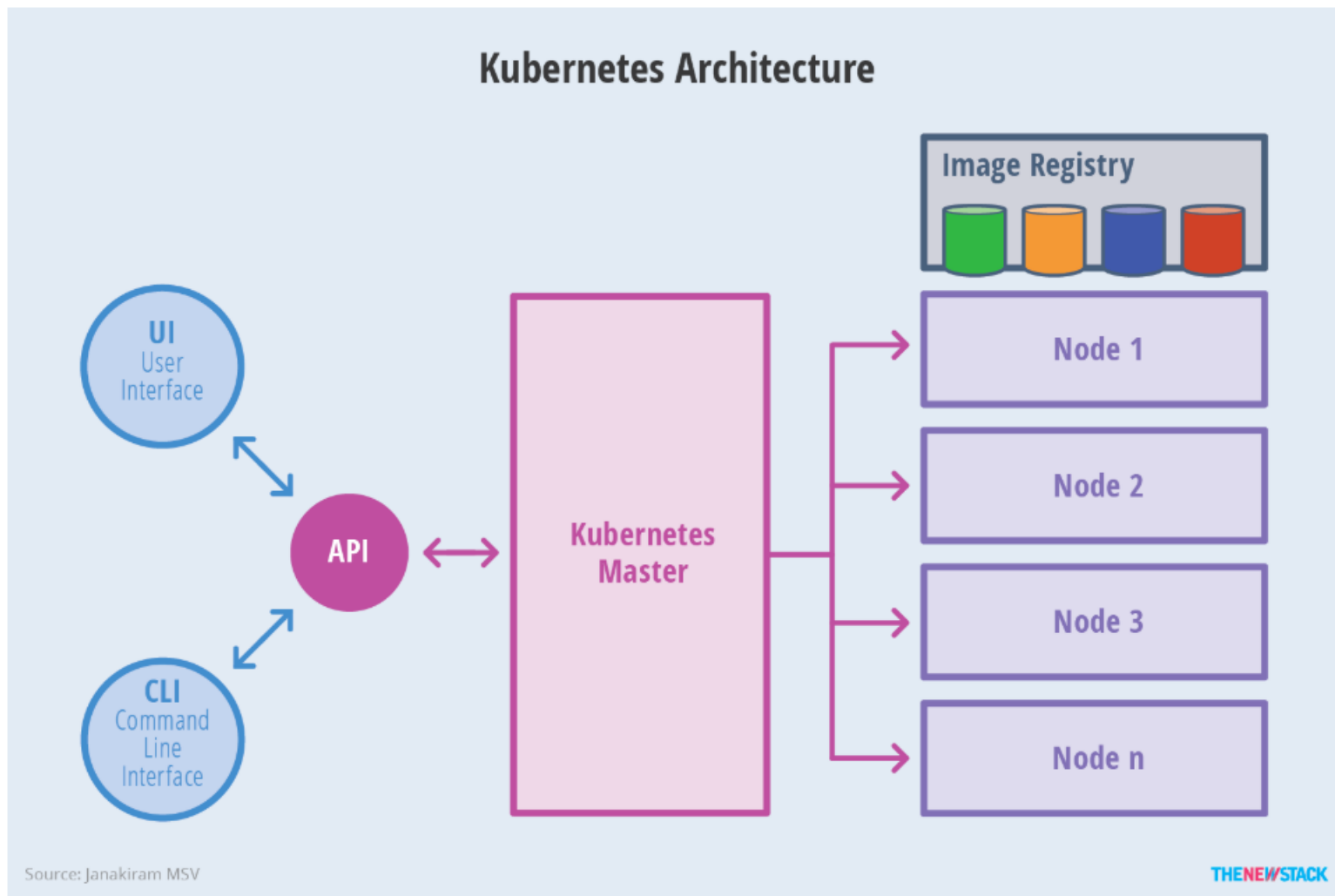
- kube-dns负责为整个集群提供DNS服务
- Ingress Controller为服务提供外网入口
- Heapster提供资源监控
- Dashboard提供GUI
- Fluentd-elasticsearch提供集群日志采集、存储与查询

KUBERNETES技术架构

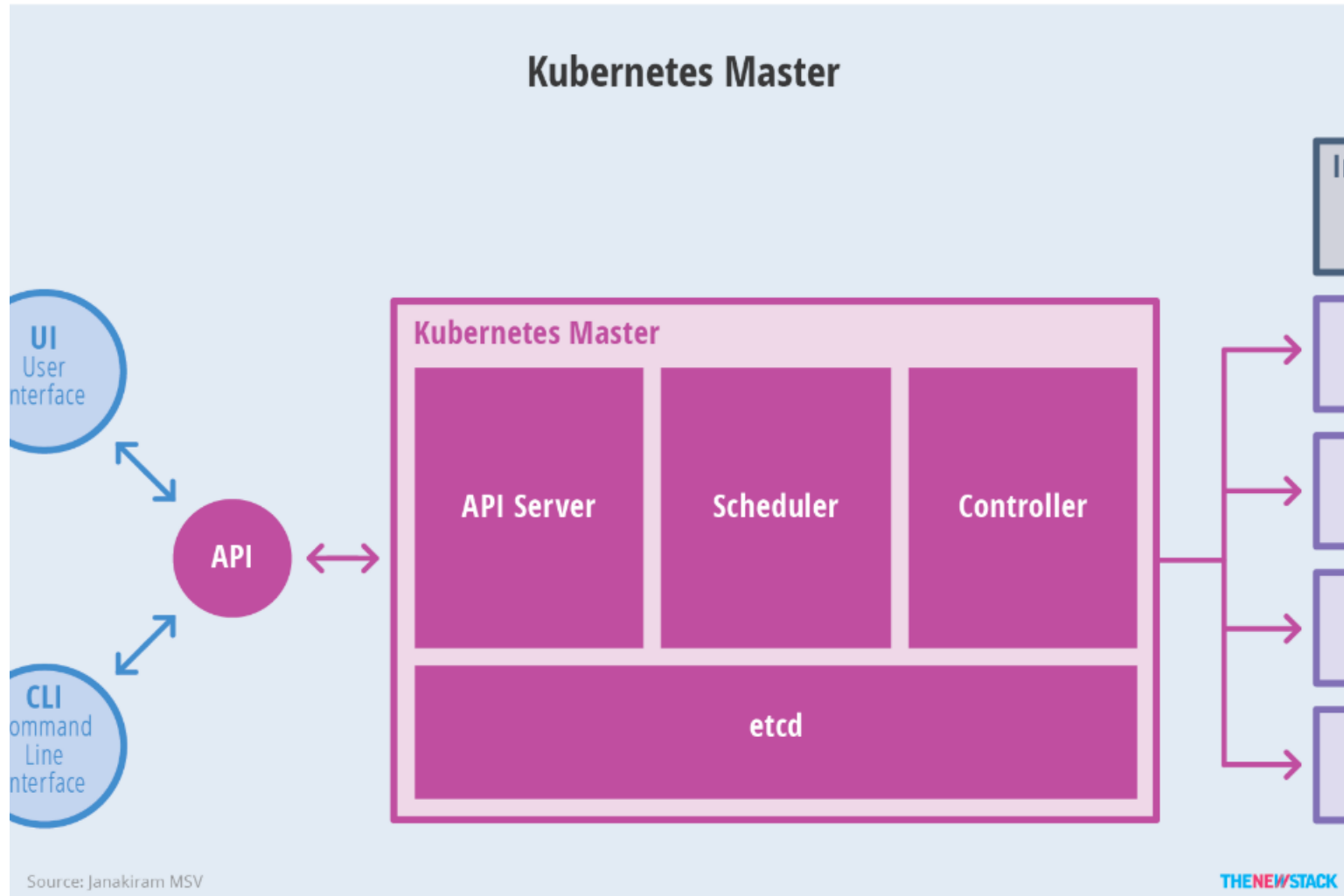
Kubernetes' high-level component architecture



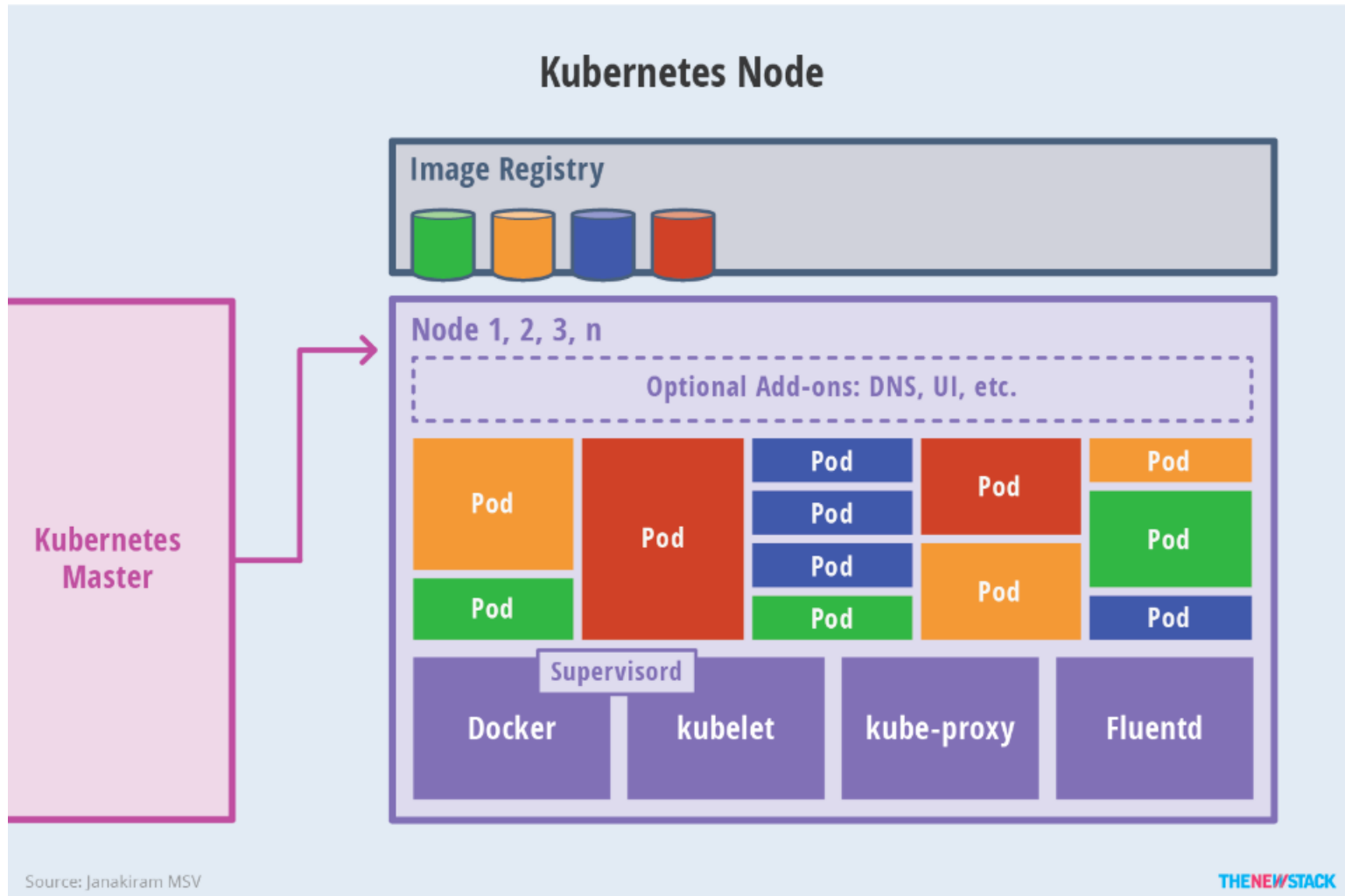
K8S技术架构



KUBERNETES技术架构



KUBERNETES技术架构



KUBERNETES技术架构

- **Kubernetes**设计理念和功能其实就是一个类似**Linux**的分层架构，如下图所示
 - 核心层：**Kubernetes**最核心的功能，对外提供**API**构建高层的应用，对内提供插件式应用执行环境
 - 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、**DNS**解析等）
 - 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩展、动态**Provision**等）以及策略管理（**RBAC**、**Quota**、**PSP**、**NetworkPolicy**等）
 - 接口层：**kubectl**命令行工具、客户端**SDK**以及集群联邦
 - 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
 - **Kubernetes**外部：日志、监控、配置管理、**CI**、**CD**、**Workflow**、**FaaS**、**OTS**应用、**ChatOps**等
 - **Kubernetes**内部：**CRI**、**CNI**、**CVI**、镜像仓库、**Cloud Provider**、集群自身的配置和管理等

核心组件

- ETCD

所有master的持续状态都存在etcd的一个实例中。这可以很好地存储配置数据。因为有watch(观察者)的支持，各部件协调中的改变可以很快被察觉。

- Kubelet

kubelet负责管理pods和它们上面的容器，images镜像、volumes等。

- kube-proxy

每一个节点也运行一个简单的网络代理和负载均衡。正如Kubernetes API里面定义的这些服务也可以在各种终端中以轮询的方式做一些简单的TCP和UDP传输。

服务端点目前是通过DNS或者环境变量(Docker-links-compatible 和 Kubernetes{FOO}_SERVICE_HOST 及 {FOO}_SERVICE_PORT 变量都支持)。这些变量由服务代理所管理的端口来解析。

核心组件

- **kube-apiserver**

API服务提供[Kubernetes API](#)的服务。这个服务试图通过把所有或者大部分的业务逻辑放到不两只的部件中从而使其具有**CRUD**特性。它主要处理**REST**操作，在**etcd**中验证更新这些对象（并最终存储）。

- **kube-controller**

所有其它的集群级别的功能目前都是由控制管理器所负责。例如，端点对象是被端点控制器来创建和更新。这些最终可以被分隔成不同的部件来让它们独自的可插拔。

- **kube-scheduler**

调度器把未调度的**pod**通过**binding api**绑定到节点上。调度器是可插拔的，并且我们期待支持多集群的调度，未来甚至希望可以支持用户自定义的调度器。

KUBERNETES安装及集群搭建

- 本例：以三个节点为例。具体节点安装组件如下，其中etcd为K8S数据库

节点IP地址	角色	安装组件名称
16.187.190.237	Master（管理节点）	Etcd/docker/kube-apiserver/kube-controller-manager/kube-scheduler/flannel/kubelet
16.187.188.247	Node1（计算节点）	Etcd/docker/kubelet/kube-proxy/flannel/kubelet
16.187.188.248	Node2（计算节点）	Etcd/docker/kubelet/kube-proxy/flannel/kubelet

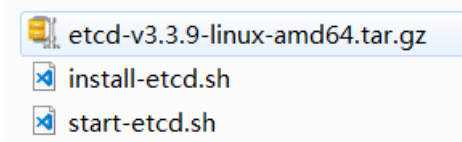
- 在CentOS7系统 以二进制文件部署，所有组件都需要4个步骤：
 - 复制对应的二进制文件到/usr/bin目录下
 - 创建systemd service启动服务文件
 - 创建service 中对应的配置参数文件
 - 将该应用加入到开机自启

KUBERNETES安装及集群搭建

- 安装etcd集群

- 由于K8s集群依赖于etcd集群存在，所以我们需要先安装etcd集群。

课件中我们提供了etcd数据库的安装和启动脚本，还有本例用到的etcd二进制文件



使用方法与安装Docker类似：

```
[root@SGDLITVM0237 k8s-install]# ./install-etcd.sh
Usage: ./install-etcd.sh FILE_NAME_ETCD_TAR_GZ NODEID THISNODEIP NODE1IP NODE2IP NODE3IP
      ./install-etcd.sh etcd-v3.3.9-linux-amd64.tar.gz 01 16.187.190.237 16.187.190.237 16.187.188.247 16.187.188.248
Get etcd binary from: https://github.com/etcd-io/etcd/releases
eg: wget https://github.com/etcd-io/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz
```

Etcd集群需要三台安装好同时启动，单独安装一台然后启动可能会失败。

三台都安装好之后同时运行start-etcd.sh 启动etcd集群

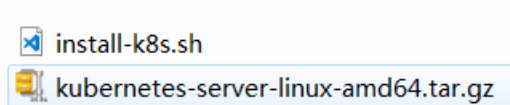
```
## etcd cluster-health
member 1bbd26b716c8af3b is healthy: got healthy result from http://16.187.188.247:2379
member a40c02ced3538529 is healthy: got healthy result from http://16.187.188.248:2379
member bcad8d949237fbeb is healthy: got healthy result from http://16.187.190.237:2379
cluster is healthy
```

KUBERNETES安装及集群搭建

- 安装K8s集群

- 在安装好etcd集群后，我们可以开始安装K8s集群了。

课件中我们提供了K8s集群的安装脚本，还有本例用到的K8s集群相关二进制文件



使用方法与安装Docker类似：

```
[root@SGDLITVM0737 k8s-install]# ./install-k8s.sh
Usage: ./install-k8s.sh FILE_NAME_K8S_TAR_GZ THISNODEIP MASTERIP
      ./install-k8s.sh kubernetes-server-linux-amd64.tar.gz 16.187.190.237 16.187.190.237
Get kubernetes binary from: https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.11.md#v1111
eg: wget https://dl.k8s.io/v1.11.1/kubernetes-server-linux-amd64.tar.gz
```

本例中用到的kubernetes版本是1.11.1，首先确保三台机器Master、Node1和Node2是在同一个网段。

然后先安装Master，再依次安装Node1和Node2。

安装完成后我们就可以通过kubectl get nodes查看集群主机状态了。

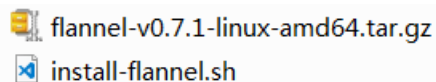
```
[root@SGDLITVM0737 k8s-install]# kubectl get nodes
NAME                STATUS    ROLES    AGE    VERSION
16.187.188.247      Ready    <none>    12m    v1.11.1
16.187.188.248      Ready    <none>    11m    v1.11.1
16.187.190.237      Ready    <none>    14m    v1.11.1
```

KUBERNETES安装及集群搭建

- 安装Flannel集群

- 在安装好K8s集群后，可以开始安装flannel集群了。

课件中我们提供了Flannel集群的安装脚本，还有本例用到的Flannel集群相关二进制文件



使用方法与安装Docker类似：

```
[root@SGDLITVM0737 k8s-install]# ./install-flannel.sh
Usage: ./install-flannel.sh FILE_NAME_ETCD_TAR_GZ THISNODEIP MASTERIP
      ./install-flannel.sh flannel-v0.7.1-linux-amd64.tar.gz 16.187.190.237 16.187.190.237
Get flannel binary from: https://github.com/coreos/flannel/releases
eg: wget https://github.com/coreos/flannel/releases/download/v0.7.1/flannel-v0.7.1-linux-amd64.tar.gz
```

本例中用到的Flannel版本是0.7.1，安装flannel集群的目的是为了规划Pod网络，是集群中承载各个Pod相互通信的网络。

三台依次安装，安装完成后可以通过ip addr命令查看当前主机网络配置，如果docker0网络和flannel网络在同一个IP段，就说明安装成功了。

```
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:16:3a:60:d8 brd ff:ff:ff:ff:ff:ff
    inet 172.17.84.1/24 brd 172.17.84.255 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:16ff:fe3a:60d8/64 scope link
        valid_lft forever preferred_lft forever
66: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN
    link/ether 06:7c:b9:1a:7d:2b brd ff:ff:ff:ff:ff:ff
    inet 172.17.84.0/32 scope global flannel.1
        valid_lft forever preferred_lft forever
    inet6 fe80::47c:b9ff:fe1a:7d2b/64 scope link
        valid_lft forever preferred_lft forever
```

YAMLFILF介绍

YAML是专门用来写配置文件的语言，非常简洁和强大，使用比json更方便。它实质上是一种通用的数据串行化格式。

- YAML语法规则：

- 大小写敏感
- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- “#”表示注释，从这个字符一直到行尾，都会被解析器忽略

YAMLFILE介绍

- 在Kubernetes中，只需要知道两种结构类型即可：
 - Lists
 - Maps
- 使用YAML用于K8s的定义带来的好处包括：
 - 便捷性：不必添加大量的参数到命令行中执行命令
 - 可维护性：YAML文件可以通过源头控制，跟踪每次操作
 - 灵活性：YAML可以创建比命令行更加复杂的结构

YAMLFILE介绍

- **YAML Maps:**

Map顾名思义指的是字典，即一个**Key:Value** 的键值对信息。例如：

```
1  ---
2  apiVersion: v1
3  kind: Pod
```

注：“---”为可选的分隔符，当需要在一个文件中定义多个结构的时候需要使用。
上述内容表示有两个键**apiVersion**和**kind**，分别对应的值为**v1**和**Pod**。

YAMLFILE介绍

Maps的value既能够对应字符串也能够对应一个**Maps**。例如：

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: kube100-site
6    labels:
7      app: web
```

注：上述的YAML文件中，**metadata**这个KEY对应的值为一个**Maps**，而嵌套的**labels**这个KEY的值又是一个**Map**。实际使用中可视情况进行多层嵌套。

YAML处理器根据行缩进来知道内容之间的关联。上述例子中，使用两个空格作为缩进，但空格的数据量并不重要，只是至少要求一个空格并且所有缩进保持一致的空格数。例如，**name**和**labels**是相同缩进级别，因此YAML处理器知道他们属于同一map；它知道**app**是**lables**的值因为**app**的缩进更大。

注意：在YAML文件中绝对不要使用**tab**键

YAMLFILE介绍

- **YAML Lists**

List即列表，说白了就是数组，例如：

```
1  args
2  -beijing
3  -shanghai
4  -shenzhen
5  -guangzhou
```

可以指定任何数量的项在列表中，每个项的定义以破折号（-）开头，并且与父元素之间存在缩进。在JSON格式中，表示如下：

```
1  {
2  |  "args": ["beijing", "shanghai", "shenzhen", "guangzhou"]
3  }
```

YAMLFILE介绍

当然Lists的子项也可以是
Maps, Maps的子项也可以是
List, 例如:

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5  |   name: kube100-site
6  |   labels:
7  |     app: web
8  spec:
9  |   containers:
10 |     - name: front-end
11 |       image: nginx
12 |       ports:
13 |         - containerPort: 80
14 |     - name: flaskapp-demo
15 |       image: jcdemo/flaskapp
16 |       ports: 8080
```

```
1  {
2    "apiVersion": "v1",
3    "kind": "Pod",
4    "metadata": {
5      "name": "kube100-site",
6      "labels": {
7        "app": "web"
8      },
9    },
10   },
11   "spec": {
12     "containers": [{
13       "name": "front-end",
14       "image": "nginx",
15       "ports": [{
16         "containerPort": "80"
17       }]
18     }, {
19       "name": "flaskapp-demo",
20       "image": "jcdemo/flaskapp",
21       "ports": [{
22         "containerPort": "5000"
23       }]
24     }]
25   }
26 }
```

YAMLFILE介绍

- 编写yamlfile创建一个pod

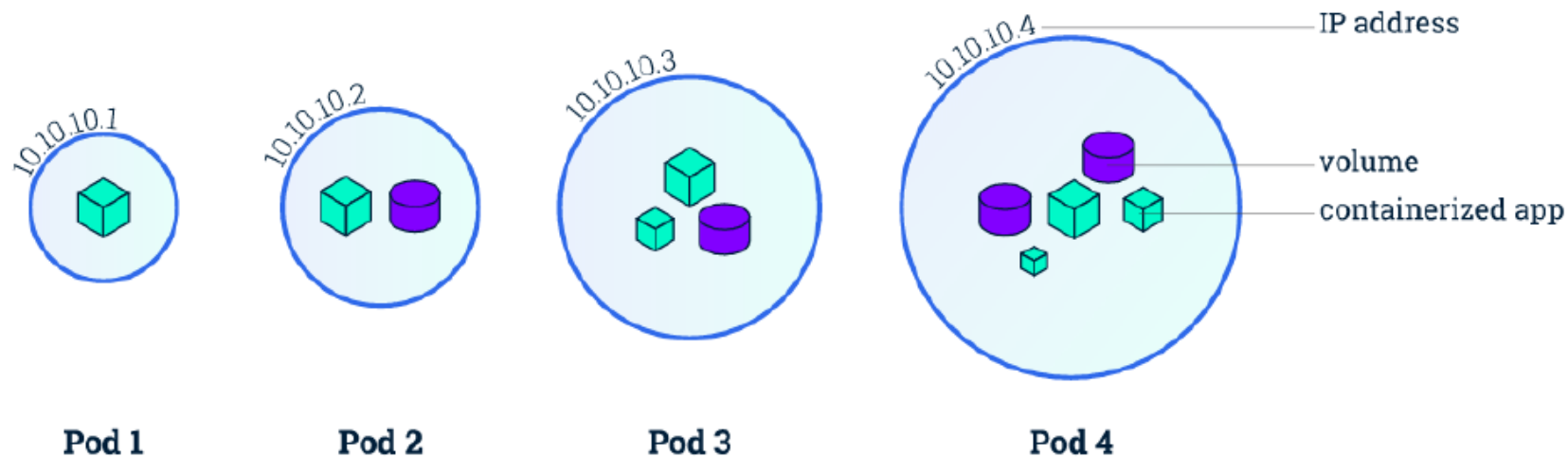
```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: private-registry
6  spec:
7    containers:
8      - name: private-registry
9        image: registry:latest
10       ports:
11         - containerPort: 5000
12           hostPort: 5000
```

```
[root@SGDLITVM0737 yaml_content]# kubectl describe pod private-registry
Name:         private-registry
Namespace:    default
Priority:      0
PriorityClassName: <none>
Node:         16.187.190.237/16.187.190.237
Start Time:   Mon, 08 Oct 2018 00:49:28 +0800
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           172.17.84.2
Containers:
  private-registry:
    Container ID:  docker://5a3b167a60de6bf4e0a1c5ca4c7c99729c1b7f931964b26096c0fdb7f0f2c774
    Image:         registry:latest
    Image ID:      docker-pullable://registry@sha256:5a156ff125e5a12ac7fdec2b90b7e2ae5120fa249cf62248337b6d04abc574c8
    Port:          5000/TCP
    Host Port:     5000/TCP
    State:         Running
      Started:     Mon, 08 Oct 2018 00:49:37 +0800
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-m4m76 (ro)
Conditions:
  Type           Status
  Initialized     True
  Ready          True
  ContainersReady True
  PodScheduled   True
Volumes:
  default-token-m4m76:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-m4m76
    Optional: false
QoS Class:   BestEffort
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
              node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type     Reason      Age   From              Message
  ----     -
  Normal   Scheduled   30s   default-scheduler Successfully assigned default/private-registry to 16.187.190.237
  Normal   Pulling     29s   kubelet, 16.187.190.237 pulling image "registry:latest"
  Normal   Pulled      22s   kubelet, 16.187.190.237 Successfully pulled image "registry:latest"
  Normal   Created     21s   kubelet, 16.187.190.237 Created container
  Normal   Started     21s   kubelet, 16.187.190.237 Started container
```

KUBERNETES基本对象

- 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。



KUBERNETES基本对象--POD

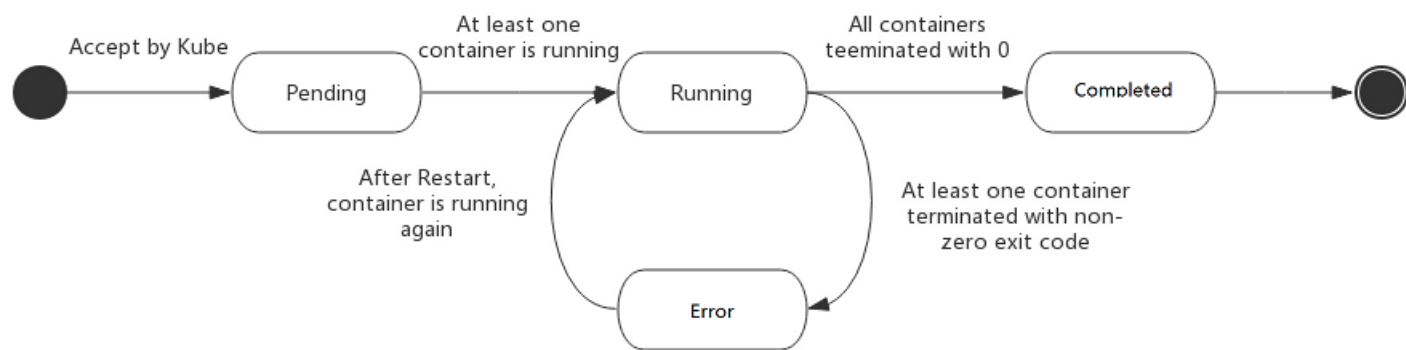
- Pod的特征
 - 包含多个共享IPC、Network和UTC namespace的容器，可直接通过localhost通信
 - 所有Pod内容容器都可以访问共享的Volume，可以访问共享数据
 - 无容错性：直接创建的Pod一旦被调度后就跟Node绑定，即使Node挂掉也不会被重新调度（而是被自动删除），因此推荐使用Deployment、Daemonset等控制器来容错
 - 优雅终止：Pod删除的时候先给其内的进程发送SIGTERM，等待一段时间（grace period）后才强制停止依然还在运行的进程
 - 特权容器（通过SecurityContext配置）具有改变系统配置的权限（在网络插件中大量应用）

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5  |   name: private-registry
6  spec:
7  |   containers:
8  |     - name: private-registry
9  |       image: registry:latest
10 |       ports:
11 |         - containerPort: 5000
12 |           hostPort: 5000
```

KUBERNETES基本对象--POD

- Pod的生命周期

- Pending
- Running
- CrashLoopBackoff
- Unknown
- Error
- Completed



KUBERNETES基本对象--POD

- 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：
 - hostname：
HOSTNAME环境变量保存了该Pod的hostname。
 - 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。
 - 集群中服务的信息：
容器的环境变量中还可以引用容器运行前创建的所有服务的信息，如：
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://172.17.17.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=172.17.17.1
 - 自定义环境变量：
Pod允许自定义环境变量，环境变量的值可以是固定数值或者从其他元素中引用。

```
65  env:
66  - name: REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY
67    value: /var/lib/registry
68  - name: MY_NODE_NAME
69    valueFrom:
70      fieldRef:
71        fieldPath: spec.nodeName
72  - name: MY_POD_NAME
73    valueFrom:
74      fieldRef:
75        fieldPath: metadata.name
76  - name: MY_POD_NAMESPACE
77    valueFrom:
78      fieldRef:
79        fieldPath: metadata.namespace
80  - name: MY_POD_IP
81    valueFrom:
82      fieldRef:
83        fieldPath: status.podIP
```

KUBERNETES基本对象--POD

- 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

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。

```
29     containers:
30     - name: private-registry
31       image: registry:latest
32       imagePullPolicy: Always
33       securityContext:
```


KUBERNETES基本对象--POD

- 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(默认策略)
- Default：优先从kubelet中配置的DNS查询

- 使用主机命名空间

通过设置spec.hostIPC参数为true，使用主机的IPC命名空间，默认为false。

通过设置spec.hostNetwork参数为true，使用主机的网络命名空间，默认为false。

通过设置spec.hostPID参数为true，使用主机的PID命名空间，默认为false。

```
1 ---
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: private-registry
6   annotations:
7     kubernetes.io/ingress-bandwidth: 3M
8     kubernetes.io/egress-bandwidth: 4M
9 spec:
10   hostIPC: false
11   hostPID: false
12   hostNetwork: false
13   hostname: my-registry
14   hostAliases:
15   - ip: "127.0.0.1"
16     hostnames:
17     - "foo.local"
18     - "bar.local"
19   subdomain: test
```

KUBERNETES基本对象--POD

- 设置Pod的hostname

通过spec.hostname参数实现，如果未设置默认使用metadata.name参数的值作为Pod的hostname。

设置Pod的子域名：

通过spec.subdomain参数设置Pod的子域名，默认为空。

指定hostname为busybox-2和subdomain为default-subdomain，完整域名为busybox-2.default-subdomain.default.svc.cluster.local，也可以简写为busybox-2.default-subdomain.default

默认情况下，DNS为Pod生成的A记录格式为pod-ip-address.mynamespace.pod.cluster.local，如1-2-3-4.default.pod.cluster.local
还需要在default namespace中创建一个名为default-subdomain（即subdomain）的headless service，否则其他Pod无法通过完整域名访问到该Pod（只能自己访问到自己）

```
1 ---
2 apiVersion: v1
3 kind: Pod
4 metadata:
5   name: private-registry
6   annotations:
7     kubernetes.io/ingress-bandwidth: 3M
8     kubernetes.io/egress-bandwidth: 4M
9 spec:
10   hostIPC: false
11   hostPID: false
12   hostNetwork: false
13   hostname: my-registry
14   hostAliases:
15   - ip: "127.0.0.1"
16     hostnames:
17     - "foo.local"
18     - "bar.local"
19   subdomain: test
20   initContainers:
21   - name: install
22     image: busybox
23     command:
24     - touch
25     - /var/lib/registry/init.txt
26   volumeMounts:
27   - name: registry-storage
28     mountPath: "/var/lib/registry"
29   containers:
30   - name: private-registry
31     image: registry:latest
32     imagePullPolicy: Always
```

KUBERNETES基本对象--POD

- 资源限制

Kubernetes通过cgroups限制容器的CPU和内存等计算资源，包括requests（请求，调度器保证调度到资源充足的

Node上，如果无法满足会调度失败）和limits（上限）等：

- `spec.containers[].resources.limits.cpu`：CPU上限，可以短暂超过，容器也不会被停止
- `spec.containers[].resources.limits.memory`：内存上限，不可以超过；如果超过，容器可能会被终止或调度到其他资源充足的机器上
- `spec.containers[].resources.requests.cpu`：CPU请求，也是调度CPU资源的依据，可以超过
- `spec.containers[].resources.requests.memory`：内存请求，也是调度内存资源的依据，可以超过；但如果超过，容器可能会在Node内存不足时清理

CPU 的单位是CPU 个数，可以用millicpu (m) 表示少于1个CPU的情况，如500m = 500millicpu = 0.5cpu， 而一个CPU相当于

- AWS 上的一个vCPU
- GCP 上的一个Core
- Azure 上的一个vCore
- 物理机上开启超线程时的一个超线程

内存的单位则包括E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki 等。

```
46 resources:
47   limits:
48     cpu: 1000m
49     memory: 400Mi
50   requests:
51     cpu: 100m
52     memory: 200Mi
```

KUBERNETES基本对象--POD

- 健康检查

微服务由于其松耦合的特性使得各个服务之间的访问基于网络/协议，为了确保容器在部署后确实处在正常运行状态，**Kubernetes**提供了两种探针（**Probe**）来探测容器的状态：

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

Kubernetes支持三种方式来执行探针：

- **exec**：在容器中执行一个命令，如果命令退出码返回0则表示探测成功，否则表示失败
- **tcpSocket**：对指定的容器IP及端口执行一个TCP检查，如果端口是开放的则表示探测成功，否则表示失败
- **httpGet**：对指定的容器IP、端口及路径执行一个HTTP Get请求，如果返回的状态码在[200,400)之间则表示探测成功，否则表示失败

KUBERNETES基本对象--POD

- 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。
- Init Container总在应用容器启动之前运行，并且包含一些应用镜像内不存在的实用工具和安装脚本。
- Init Container独立于应用服务
- 一个Pod中可能有一个或多个先于应用容器启动的Init Container
- Init Container与普通容器非常像，除了以下两点
 - 它们总是运行到完成
 - 顺序执行，每个都必须在下一个启动之前成功完成
- 注意事项：
 - 由于Init Container必须在APP容器启动之前完成，所以可以用于处理服务依赖
 - 如果pod重启了，所有Init Container都需要重新运行
 - Init Container禁止使用readiness/liveness探针
 - readiness/liveness禁止使用lifecycle hook

```
17  apiVersion: v1
18  kind: Pod
19  metadata:
20    name: private-registry
21    namespace: default
22    annotations:
23      kubernetes.io/ingress-bandwidth: 3M
24      kubernetes.io/egress-bandwidth: 4M
25    labels:
26      k8s-app: kube-registry
27  spec:
28    hostIPC: false
29    hostPID: false
30    hostNetwork: false
31    hostname: my-registry
32    hostAliases:
33      - ip: "127.0.0.1"
34        hostnames:
35          - "foo.local"
36          - "bar.local"
37    subdomain: test
38    initContainers:
39      - name: install
40        image: busybox
41        command:
42          - touch
43          - /var/lib/registry/init.txt
44        volumeMounts:
45          - name: registry-storage
46            mountPath: "/var/lib/registry"
47    containers:
```

KUBERNETES基本对象--POD

- **Init Container**用途

- 服务依赖

比如某个服务A依赖于db，**name**可以利用服务A中pod的**Init Container**判断db是否正常提供服务，如果db不能提供服务，设置**Init Container**启动失败，那么Pod中的服务A就不会被启动了。

- 业务无关

Init Container独立于服务，如替换配置文件等服务初始化工作，与业务无关的工作都可以放在其中执行。

- 特殊安全处理

应用镜像因安全原因等，没办法安装或运行的工具，都可以放到**Init Container**中运行

KUBERNETES基本对象--POD

- Container Lifecycle Hooks

容器生命周期钩子（Container Lifecycle Hooks）
监听容器生命周期的特定事件，并在事件发生时执行已注册的回调函数。支持两种钩子：

- **postStart**: 容器创建后立即执行，注意由于是异步执行，它无法保证一定在ENTRYPOINT之前运行。如果失败，容器会被杀死，并根据RestartPolicy决定是否重启
- **preStop**: 容器终止前执行，常用于资源清理。如果失败，容器同样也会被杀死

而钩子的回调函数支持两种方式：

- **exec**: 在容器内执行命令，如果命令的退出状态码是0表示执行成功，否则表示失败
- **httpGet**: 向指定URL发起GET请求，如果返回的HTTP状态码在[200, 400)之间表示请求成功，否则表示失败

```
38   initContainers:
39   □ - name: install
40     image: busybox
41   □ command:
42     - touch
43     - /var/lib/registry/init.txt
44   volumeMounts:
45   □ - name: registry-storage
46     mountPath: "/var/lib/registry"
47   containers:
48   □ - name: private-registry
49     image: registry:latest
50     imagePullPolicy: Always
51   □ securityContext:
52     capabilities:
53     □ add:
54       - NET_ADMIN
55     drop:
56     - KILL
57   □ lifecycle:
58     postStart:
59     □ exec:
60       command: ["touch", "/var/lib/registry/postStart.txt"]
61     preStop:
62     □ exec:
63       command: ["touch", "/var/lib/registry/preStop.txt"]
64   □ resources:
65     limits:
66     □ cpu: 100m
67       memory: 400Mi
68     requests:
69     □ cpu: 100m
70       memory: 200Mi
```

KUBERNETES基本对象--POD

- **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。

KUBERNETES基本对象--POD

- 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：

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: hello-world
5  spec:
6    containers:
7      - name: hello-world-container
8        # The container definition
9        # ...
10     securityContext:
11       privileged: true
```

- 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）。

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: hello-world
5  spec:
6    containers:
7      - name: hello-world-container
8        # The container definition
9        # ...
10     securityContext:
11       selinuxOptions:
12         user: system_u
13         role: object_r
14         type: svirt_sandbox_file_t
15       level: s0:c100,c200
```

KUBERNETES基本对象--POD

- 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PodSecurityPolicyadmission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

```
1  apiVersion: extensions/v1beta1
2  kind: PodSecurityPolicy
3  metadata:
4    name: permissive
5  spec:
6    seLinux:
7      | rule: RunAsAny
8    supplementalGroups:
9      | rule: RunAsAny
10   runAsUser:
11     | rule: RunAsAny
12   fsGroup:
13     | rule: RunAsAny
14   hostPorts:
15     - min: 8000
16       | max: 8080
17   volumes:
18     - '*'
```

KUBERNETES基本对象--POD

- 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。

```
47 containers:
48   - name: private-registry
49     image: registry:latest
50     imagePullPolicy: Always
51   securityContext:
52     capabilities:
53       add:
54         - NET_ADMIN
55       drop:
56         - KILL
57   lifecycle:
58     postStart:
59       exec:
60         command: ["touch", "/var/lib/registry/postStart.txt"]
61     preStop:
62       exec:
63         command: ["touch", "/var/lib/registry/preStop.txt"]
64   resources:
65     limits:
66       cpu: 1000m
67       memory: 400Mi
68     requests:
69       cpu: 100m
70       memory: 200Mi
```

KUBERNETES基本对象--POD

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: private-registry
6  spec:
7    containers:
8  - name: private-registry
9    image: registry:latest
10   env:
11 - name: REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY
12   value: /var/lib/registry
13   ports:
14 - containerPort: 5000
15   hostPort: 5000
16   volumeMounts:
17 - name: registry-storage
18   mountPath: /var/lib/registry
19 volumes:
20 - name: registry-storage
21   emptyDir: {}
```

```
[root@SGDLITVM0852 yamll]# kubectl describe pod private-registry
Name:         private-registry
Namespace:    default
Priority:      0
PriorityClassName: <none>
Node:         16.187.191.97/16.187.191.97
Start Time:   Sun, 07 Oct 2018 20:27:07 +0800
Labels:       <none>
Annotations:  <none>
Status:       Running
IP:           172.17.8.2
Containers:
  private-registry:
    Container ID:  docker://680bef67afe89631f276605f499773680033576a51f31e96a984ca3e6ec8cf86
    Image:         registry:latest
    Image ID:      docker-pullable://registry@sha256:5a156ff125e5a12ac7fdec2b90b7e2ae5120fa249cf62248337b6d04abc574c8
    Port:         5000/TCP
    Host Port:    5000/TCP
    State:        Running
      Started:    Sun, 07 Oct 2018 20:27:22 +0800
    Ready:        True
    Restart Count: 0
    Environment:
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /var/lib/registry
    Mounts:
      /var/lib/registry from registry-storage (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-rx9kv (ro)
  Conditions:
    Type             Status
    Initialized       True
    Ready             True
    ContainersReady   True
    PodScheduled      True
  Volumes:
    registry-storage:
      Type: EmptyDir (a temporary directory that shares a pod's lifetime)
      Medium:
    default-token-rx9kv:
      Type: Secret (a volume populated by a Secret)
      SecretName: default-token-rx9kv
      Optional: false
    QoS Class:       BestEffort
    Node-Selectors:  <none>
    Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                    node.kubernetes.io/unreachable:NoExecute for 300s
  Events:
    Type     Reason      Age   From          Message
    ----     -
    Normal   Scheduled   1m    default-scheduler   Successfully assigned default/private-registry to 16.187.191.97
    Normal   Pulling    1m    kubelet, 16.187.191.97   pulling image "registry:latest"
    Normal   Pulled     1m    kubelet, 16.187.191.97   Successfully pulled image "registry:latest"
```

KUBERNETES基本对象--POD

- RestartPolicy

只持三种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。

```
81     valueFrom:
82     |   fieldRef:
83     |   |   fieldPath: status.podIP
84   ports:
85   - containerPort: 5000
86     hostPort: 5000
87   volumeMounts:
88   - name: registry-storage
89     mountPath: /var/lib/registry
90   volumes:
91   - name: registry-storage
92     emptyDir: {}
93   restartPolicy: Always
```

KUBERNETES基本对象--POD

- 限制网络带宽
 - 可以通过给Pod增加
kubernetes.io/ingressbandwidth和
kubernetes.io/egress-bandwidth这两个
annotation来限制Pod的网络带宽
 - 目前只有kuben~~et~~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。

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: private-registry
6    annotations:
7      kubernetes.io/ingress-bandwidth: 3M
8      kubernetes.io/egress-bandwidth: 4M
9  spec:
10    hostIPC: false
11    hostPID: false
12    hostNetwork: false
13    hostname: my-registry
```

KUBERNETES基本对象--POD

- 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。
- 从v1.7开始，可以通过pod.Spec.HostAliases来增加hosts内容

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: private-registry
6    annotations:
7      | kubernetes.io/ingress-bandwidth: 3M
8      | kubernetes.io/egress-bandwidth: 4M
9  spec:
10   hostIPC: false
11   hostPID: false
12   hostNetwork: false
13   hostname: my-registry
14   hostAliases:
15     - ip: "127.0.0.1"
16       hostnames:
17         - "foo.local"
18         - "bar.local"
19   subdomain: test
```

KUBERNETES基本对象--POD

- **静态Pod**

- 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。

KUBERNETES基本对象--NAMESPACE

- Namespace

Namespace（命名空间）是Kubernetes系统中一个非常重要的概念。

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对象划分为不同的项目组或用户组。

常见的Pod/Service/ReplicaSet和Deployment等都是属于某一个namespace的。

而node/PV/ClusterRole等不属于任何namespace。

Kubernetes集群初始有两个namespace：

- 默认的namespace是default
- 系统的namespace是kube-system

这两个namespace是不可删除的

Namespace名称满足正则表达式`a-z0-9?`，最大长度63位。

v1.7版本增加了kube-public命名空间，该命名空间用来存放公共的信息，一般以ConfigMap的形式存放。

- `kubectl get configmap -n=kube-public`

KUBERNETES基本对象--NAMESPACE

- 创建namespace

`kubectl create namespace mynamespace`

- 查询

`kubectl get namespaces`

Namespace包含两种状态 “Active” 和 “Terminating”，在namespaces删除过程中，namespaces状态被设置成 “Terminating”。

- 删除namespace

`kubectl delete namespace mynamespace`

删除namespace会自动删除所有属于该namespace的资源（Pod/Service/ReplicaSet和Deployment等）

```
[root@SGDLITVM0852 yaml]# kubectl create namespace mynamespace
namespace/mynamespace created
[root@SGDLITVM0852 yaml]# kubectl get namespaces
NAME           STATUS    AGE
default        Active    1d
kube-public    Active    1d
kube-system    Active    1d
mynamespace    Active    7s
[root@SGDLITVM0852 yaml]# kubectl delete namespace mynamespace
namespace "mynamespace" deleted
```

```
Every 2.0s: kubectl get namespaces

NAME           STATUS    AGE
default        Active    1d
kube-public    Active    1d
kube-system    Active    1d
mynamespace    Terminating 3s
```

KUBERNETES基本对象--LABEL

- 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，一个label可以attach到任何资源对象上（如Pod，RC，RS，Deployment，Service等）
- 多个label（它们之间是and关系），如
app=nginx,env=production
- Label定义好后其他对象可以使用labelSelector来选择一组相同label的对象（比如Deployment和service用label来选择一组pod）

```
1  ---
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: kube-registry
6    namespace: default
7    labels:
8      k8s-app: kube-registry
9  spec:
10   ports:
11     - name: registry
12       port: 5000
13       protocol: TCP
14   selector:
15     k8s-app: kube-registry
16  ---
17  apiVersion: v1
18  kind: Pod
19  metadata:
20    name: private-registry
21    namespace: default
22    annotations:
23      kubernetes.io/ingress-bandwidth: 3M
24      kubernetes.io/egress-bandwidth: 4M
25    labels:
26      k8s-app: kube-registry
27  spec:
28    hostIPC: false
29    hostPID: false
30    hostNetwork: false
31    hostname: my-registry
32    hostAliases:
33      - ip: "127.0.0.1"
34        hostnames:
35          - "foo.local"
36          - "bar.local"
37    subdomain: test
```

KUBERNETES基本对象--LABEL

- Label使用

可以使用Label selector进行查询和删选拥有特定label的资源对象

Label selector支持以下两种方式:

- 等式 (= / ==, !=) :

```
[root@SGDLITVM0852 yam1]# kubectl get pods -l k8s-app=kube-registry
NAME                READY   STATUS    RESTARTS   AGE
private-registry    1/1     Running   0           1h
[root@SGDLITVM0852 yam1]# kubectl get pods -l k8s-app!=kube-registry
No resources found.
[root@SGDLITVM0852 yam1]# kubectl get pods -l k8s-app==kube-registry
NAME                READY   STATUS    RESTARTS   AGE
private-registry    1/1     Running   0           1h
```

- 集合 (In, NotIn) :

```
[root@SGDLITVM0852 yam1]# kubectl get pods -l 'k8s-app in (kube-registry)'
NAME                READY   STATUS    RESTARTS   AGE
private-registry    1/1     Running   0           1h
[root@SGDLITVM0852 yam1]# kubectl get pods -l 'k8s-app notin (kube-registry)'
No resources found.
```

```
1  ---
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: kube-registry
6    namespace: default
7    labels:
8      k8s-app: kube-registry
9  spec:
10   ports:
11     - name: registry
12       port: 5000
13       protocol: TCP
14   selector:
15     k8s-app: kube-registry
16  ---
17  apiVersion: v1
18  kind: Pod
19  metadata:
20    name: private-registry
21    namespace: default
22    annotations:
23      kubernetes.io/ingress-bandwidth: 3M
24      kubernetes.io/egress-bandwidth: 4M
25    labels:
26      k8s-app: kube-registry
27  spec:
28    hostIPC: false
29    hostPID: false
30    hostNetwork: false
31    hostname: my-registry
32    hostAliases:
33      - ip: "127.0.0.1"
34        hostnames:
35          - "foo.local"
36          - "bar.local"
37    subdomain: test
```

KUBERNETES基本对象--ANNOTATION

- Annotation

Annotation是以key/value的形式附加与对象的注解

annotations:

key1: value1

key2: value2

主要用于记录一些附加信息，用来辅助应用部署、安全策略以及调度策略等

不同于label用于标示和选择对象

```
16 ---
17 apiVersion: v1
18 kind: Pod
19 metadata:
20   name: private-registry
21   namespace: default
22   annotations:
23     kubernetes.io/ingress-bandwidth: 3M
24     kubernetes.io/egress-bandwidth: 4M
25   labels:
26     k8s-app: kube-registry
27 spec:
28   hostIPC: false
29   hostPID: false
30   hostNetwork: false
31   hostname: my-registry
32   hostAliases:
33   - ip: "127.0.0.1"
34     hostnames:
35     - "foo.local"
36     - "bar.local"
37   subdomain: test
```

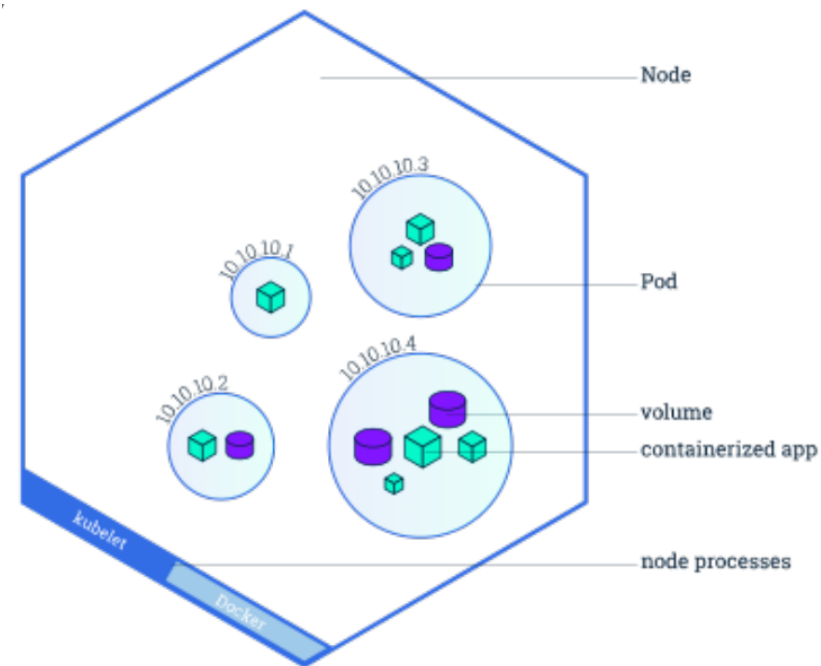
KUBERNETES基本对象--NODE

- node

Node是为Kubernetes集群提供计算能力的对象。

最初成为Minion，后改名为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节点至少应该运行Docker、kubelet和kube-proxy服务。



KUBERNETES基本对象--NODE

- Node特性

Node不像其他资源（如Pod和namespace），node本质上不是Kubernetes来创建的。

虽然可以通过Manifest创建一个node对象，但Kubernetes也只是去检查是否真的有这么一个node，如果检查失败，也不会往上调度Pod。

这个检查是由NodeController来完成的。NodeController负责：

- 维护Node状态
- 与CloudProvider同步Node
- 给Node分配容器CIDR

```
[root@SGDLITVM0852 yaml]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
16.187.191.95	Ready	<none>	2d	v1.11.1
16.187.191.97	Ready	<none>	2d	v1.11.1
16.187.191.99	Ready	<none>	2d	v1.11.1

默认情况下，kubelet在启动时会向master注册自己，并创建Node资源

KUBERNETES基本对象--NODE

- Node的状态

每个Node都包括以下信息：

- 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、容器引擎版本和OS类型等

KUBERNETES基本对象--NODE

- Node的维护
 - `kubectl cordon`: 标志Node为不可调度，但不影响其上正在运行的Pod
 - `kubectl drain`: 标志node不可调度，同时evict Pods
 - 除了mirror pods
 - `--ignore-daemonsets`参数会在evict Pods同时删除由DaemonSet管理的Pods
 - `kubectl delete node`: 删除node

KUBERNETES基本对象--SERVICE

- Service

Service的定义也是通过yaml或json, 比如右边这个yaml文件定义了一个名为kube-registry的服务, 将服务的5000端口转发到default namespace中带有标签k8s-app=kube-registry的Pod的5000端口

```
1  ---
2  apiVersion: v1
3  kind: Service
4  metadata:
5    name: kube-registry
6    namespace: default
7    labels:
8      | k8s-app: kube-registry
9  spec:
10    ports:
11      - name: registry
12        | port: 5000
13        | protocol: TCP
14    selector:
15      | k8s-app: kube-registry
```

KUBERNETES基本对象--SERVICE

- Service的特性

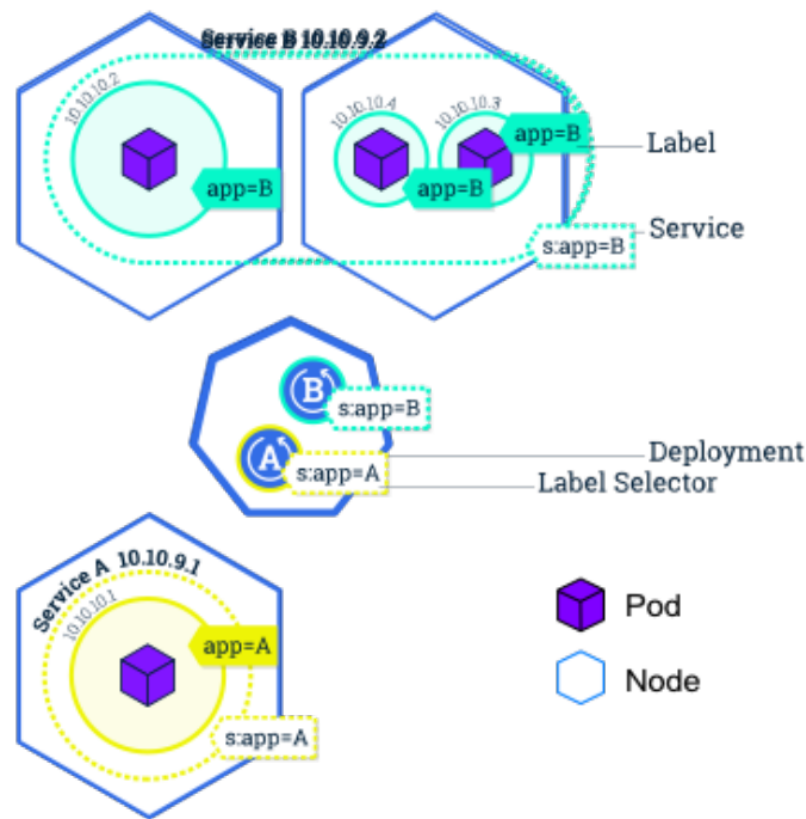
Pod只是一个运行服务的实例，随时可能在一个节点上停止，并在另外一个节点上以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口提供服务。

要稳定提供服务，就需要服务发现和负载均衡。

Service定义了一个服务的访问入口，通过这个入口可以访问其背后一组Pod副本组成的集群实例。

Service是应用服务的抽象，通过Labels为应用服务提供负载均衡和服务发现。

通过LabelSelector进行关联Pod，这些标签匹配的PodIP和端口列表组成Endpoints，由kube-proxy负责将服务的IP负载均衡到这些Endpoint上，从而实现应用的零宕机升级。



KUBERNETES基本对象--SERVICE

- **Service**工作方式

service自动分配了Cluster IP 172.17.17.52

```
[root@SGDLITVM0852 yam1]# kubectl get service kube-registry
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-registry	ClusterIP	172.17.17.52	<none>	5000/TCP	2m

自动创建的endpoint

```
[root@SGDLITVM0852 yam1]# kubectl get endpoints kube-registry
```

NAME	ENDPOINTS	AGE
kube-registry	172.17.20.2:5000	59s

Service自动关联endpoint

```
[root@SGDLITVM0852 yam1]# kubectl describe service kube-registry
```

Name: kube-registry
Namespace: default
Labels: k8s-app=kube-registry
Annotations: <none>
Selector: k8s-app=kube-registry
Type: ClusterIP
IP: 172.17.17.52
Port: registry 5000/TCP
TargetPort: 5000/TCP
Endpoints: 172.17.20.2:5000
Session Affinity: None
Events: <none>

KUBERNETES基本对象--SERVICE

- Service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

- 为Service在每台机器上绑定一个端口，通过<NodeIP>:NodePort来访问该服务

- LoadBalancer

- 在NodePort的基础上，借助CloudProvider创建一个外部负载均衡器，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域名（通过spec.externalName设定）

KUBERNETES基本对象--SERVICE

- 不指定Selector的服务

在创建service的时候也可以不指定Selector，用来将service转发到Kubernetes集群外部的服务（而不是集群内部Pod）。

没有Selector，Endpoint不会自动被创建，目前支持以下两种方法

- 自定义endpoint指定远程服务，即创建同名service和endpoint，在endpoint中设置外部服务IP和端口

```
1  kind: Endpoints
2  apiVersion: v1
3  metadata:
4    name: my-service
5  subsets:
6    - addresses:
7      - ip: 10.0.251.145
8  ports:
9    - port: 6379
```

```
11 kind: Service
12 apiVersion: v1
13 metadata:
14   name: my-service
15 spec:
16   ports:
17     - port: 6379
18       targetPort: 6379
19       protocol: TCP
```

KUBERNETES基本对象--SERVICE

- 通过DNS转发指定远程服务

DNS服务会给<service- name>.<namespace>.svc.cluster.local创建一个CNAME，其值为my.database.example.com

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）

```
1  kind: Service
2  apiVersion: v1
3  metadata:
4    name: my-service
5    namespace: default
6  spec:
7    type: ExternalName
8    externalName: my.database.example.com
```

KUBERNETES基本对象--VOLUME

- Volume

Kubernetes集群中的存储卷跟Docker的存储卷有些类似，单作用范围不太一样

- Docker存储卷作用范围是一个容器
- Kubernetes的存储卷的生命周期和作用范围是一个Pod。
- 每个Pod中的存储卷会由Pod中所有的容器共享

目前Kubernetes支持的存储卷类型有很多

- 云存储：
azureDisk,awsElasticBlockStore,gcePersistentDisk,portworxVolume等
- 分布式存储：Ceph,NFS,GlusterFS等
- 本地存储：hostPath,configMap等

本例中由于演示环境原因，主要对NFS、hostPath和configMap进行演示和说明。

```
1  ---
2  apiVersion: v1
3  kind: Pod
4  metadata:
5    name: private-registry
6  spec:
7    containers:
8  - name: private-registry
9    image: registry:latest
10   env:
11   - name: REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY
12     value: /var/lib/registry
13   ports:
14   - containerPort: 5000
15     hostPort: 5000
16   volumeMounts:
17   - name: registry-storage
18     mountPath: /var/lib/registry
19   volumes:
20   - name: registry-storage
21     emptyDir: {}
```

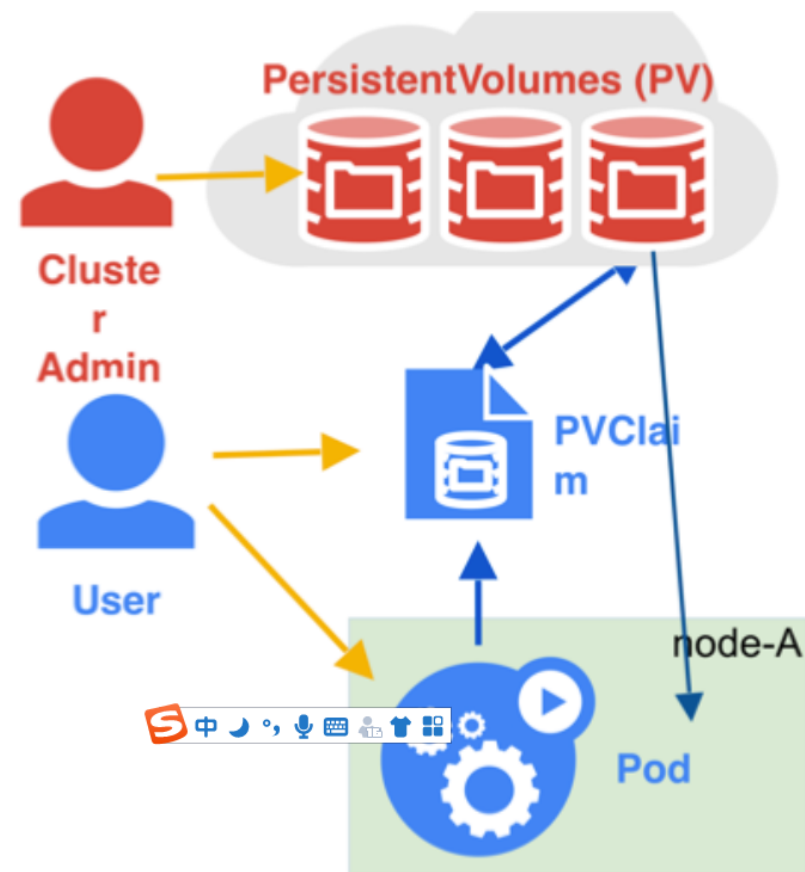

KUBERNETES基本对象--VOLUME

- PV/PVC

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支持它的持久化引擎。

PV和PVC的生命周期可以分为以下5个阶段：

- **Provisioning**，即PV的创建，可以直接创建PV（静态方式），也可以通过**StorageClass**动态创建。
- **Binding**，将PV分配给PVC。
- **Using**，Pod通过PVC使用该Volume。
- **Releasing**，Pod释放Volume并删除PVC
- **Reclaiming**，回收PV，可以设置三种回收策略：
 - **Retain**：保留策略，允许人工处理保留的策略。
 - **Recycle**：将执行清除策略，之后可以被新的PVC使用，即`rm -rf <volume path>/*`（只有NFS和hostPath支持）
 - **Delete**：删除存储资源，需要插件支持。



KUBERNETES基本对象--VOLUME

- PV的状态

根据这5个阶段，Volume的状态有以下4种

- Available，可用
- Bound，已绑定某个claim的资源
- Released，claim已被删除，PVC已解绑，但PV资源还没有被集群回收
- Failed，PV回收操作失败

- 在PVC绑定PV时通常根据两个条件来绑定：

- 存储的大小
- 访问模式

- 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

KUBERNETES基本对象--VOLUME

- 安装NFS server

本例主要介绍NFS作为PV的使用方法，所以在创建PV前需要首先配置一台NFS server并发布一个目录。

本例课件中提供了一个简易的NFS server安装脚本和必

 nfs-utils-1.3.0-0.8.el7.x86_64.rpm	2018/10/12 0:47	RPM 文件	362 KB
 rpcbind-0.2.0-26.el7.x86_64.rpm	2018/10/12 0:47	RPM 文件	55 KB
 setUpNFS.sh	2018/10/12 0:45	SH 文件	4 KB

Examples:

```
./setUpNFS.sh           Default setup.
./setUpNFS.sh /xxx/xxx Set NFS folder.
./setUpNFS.sh /xxx/xxx true 1999 1999 Set NFS folder, userId and groupId.
./setUpNFS.sh -h       Show help.
```

KUBERNETES基本对象--VOLUME

- 创建PV/PVC

通过yaml file 创建PV/PVC

kubectl create -f pv-pvc.yaml

```
[root@SGDLITVM0852 yaml]# kubectl create -f pv-pvc.yaml
persistentvolume/demo-pv created
persistentvolumeclaim/demo-pvc created
[root@SGDLITVM0852 yaml]# kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM                STORAGECLASS  REASON  AGE
demo-pv   5Gi       RWX           Retain          Bound   default/demo-pvc     default       5s
[root@SGDLITVM0852 yaml]# kubectl get pvc --all-namespaces
NAMESPACE  NAME      STATUS  VOLUME  CAPACITY  ACCESS MODES  STORAGECLASS  AGE
default    demo-pvc  Bound   demo-pv  5Gi       RWX           default       10s
```

```
1  kind: PersistentVolume
2  apiVersion: v1
3  metadata:
4    name: demo-pv
5    labels:
6      |  kubernetes.io/cluster-service: "true"
7  spec:
8    capacity:
9      |  storage: 5Gi
10   accessModes:
11     |  - ReadWriteMany
12   persistentVolumeReclaimPolicy: Retain
13   nfs:
14     |  path: /var/vols/demo
15     |  server: 16.187.191.97
16
17  ---
18  kind: PersistentVolumeClaim
19  apiVersion: v1
20  metadata:
21    name: demo-pvc
22    namespace: default
23    labels:
24      |  kubernetes.io/cluster-service: "true"
25  spec:
26    accessModes:
27      |  - ReadWriteMany
28    resources:
29      requests:
30        |  storage: 5Gi
31    selector:
32      matchLabels:
33        |  |  kubernetes.io/cluster-service: "true"
```

KUBERNETES基本对象--VOLUME

- Pod挂载Volume

- hostPath:

Kubernetes允许用户将主机的hostPath挂载到Pod中。

当Pod启动时会在Pod上创建一个Volume，这个Volume可以挂载到Pod中的任何一个容器内，但是要注意以下几点：

- 如果容器内，被挂的目标路径已存在，挂载后，该目录会被覆盖为指定挂载的hostPath内容
- 已挂载的Volume内文件权限与主机上文件一致。
- 当挂载的目标路径在容器内不存在时，会自动创建该目录。
- 但Pod只能挂载它运行所在的主机的hostPath。
- 当一个集群中有多个节点满足Pod运行条件时，Pod可能被创建在任何一个节点上，需要确保该节点上指定的hostPath目录已经存在。

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: test-volume
5  spec:
6    containers:
7      - name: test-volume
8        image: busybox
9        volumeMounts:
10       - mountPath: /mnt
11         name: test-volume
12    volumes:
13      - name: test-volume
14        hostPath:
15          path: /root
16          type: Directory
```

KUBERNETES基本对象--VOLUME

- 挂载PV:

在创建Pod的yaml file中

`volumes.persistentVolumeClaim.claimName`可以指定要挂载的PVC的名字

`volumeMounts`是将Pod中创建的Volume挂载到某个container中:

- `mountPath`是要挂载的容器中的目标路径
- `name`可以指定volumes中创建的卷名
- `subPath`指定将从volumes的某个子目录开始挂载。

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: test-volume1
5  spec:
6    containers:
7    - name: test-volume
8      image: busybox
9      command: [ "/bin/sh", "-c", "sleep 3600" ]
10     volumeMounts:
11     - mountPath: /mnt
12       name: test-pv
13       subPath: test-subpath
14   volumes:
15   - name: test-pv
16     persistentVolumeClaim:
17       claimName: demo-pvc
```

常用的WORKLOADS

- Replication Controller
- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

常用的WORKLOADS--REPLICATION CONTROLLER

- Replication Controller

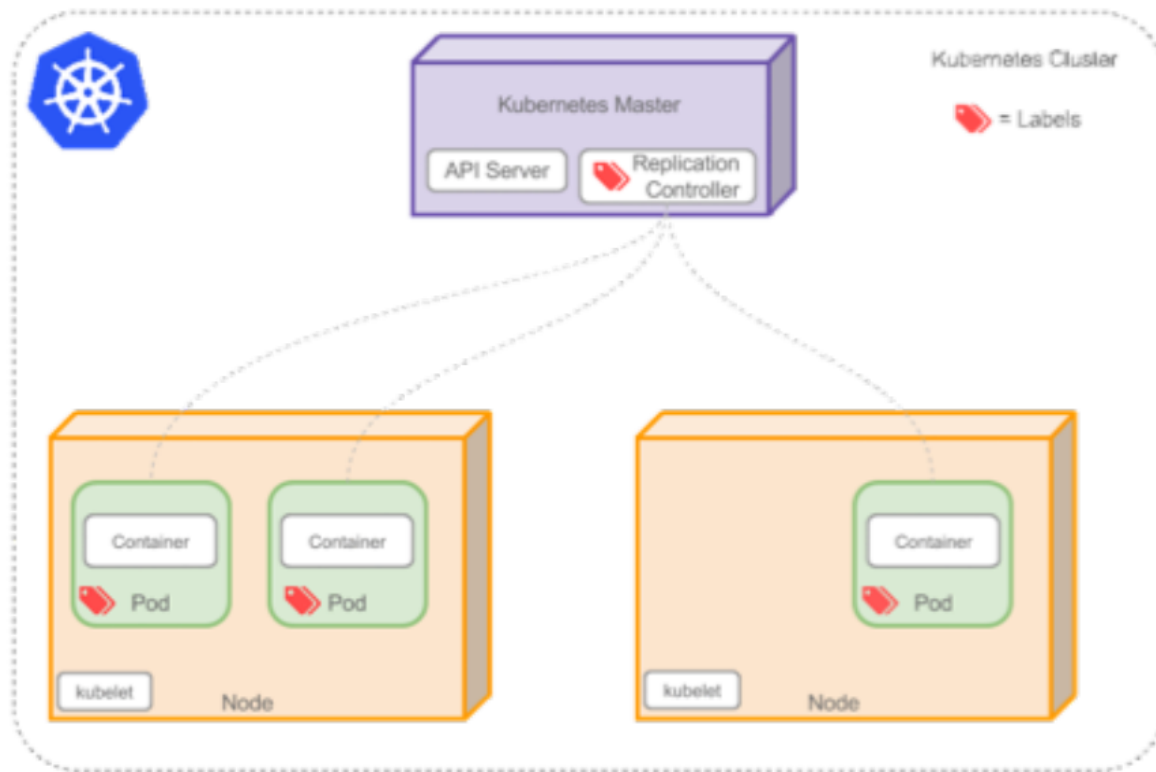
复制控制器（RC），比较早期的技术概念。

RC是Kubernetes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0~多个

- 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就会杀死多出的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。



常用的WORKLOADS--REPLICATION CONTROLLER

Replication Controller主要包含以下部分：

- Pod期待的副本数
- 用于筛选目标Pod的LabelSelector
- Pod数量小于预期副本数量，窗新Pod的template

关于Replication Controller的几点说明：

- 在删除RC的时候，可以不影响任何Pod，使用`kubectl delete rc rcName --cascade=false`命令。
- 原RC删除后，可以创建一个新的RC来替换它，只要旧的RC和新的RC的`spec.selector`相匹配即可。

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
default	nginx-p9d66	1/1	Running	0	2m	172.17.20.4	16.187.191.99
default	nginx-qv5dx	1/1	Running	0	2m	172.17.8.2	16.187.191.97
default	nginx-wllch	1/1	Running	0	2m	172.17.59.2	16.187.191.95

```
1  apiVersion: v1
2  kind: ReplicationController
3  metadata:
4    name: nginx
5  spec:
6    replicas: 3
7    selector:
8      app: nginx
9    template:
10     metadata:
11       name: nginx
12       labels:
13         app: nginx
14     spec:
15       containers:
16       - name: nginx
17         image: nginx
18         ports:
19         - containerPort: 80
```

常用的WORKLOADS--REPLICASET

- **ReplicaSet**

- RS是新一代的RC,提供同样的高可用能力，区别主要在于RS能够支持更多类型的匹配模式，更加灵活。
 - RC仅支持基于等式(Equality-based)的Label Selector
 - 而RS还支持基于集合(Set-based)的Label Selector
- RS对象一般不单独使用，主要作用Deployment协调 Pod创建、删除和更新。除非需要自定义更新编排， 否则建议使用Deployment而不是直接使用RS
 - 当使用Deployment时候，你不必担心创建Pod的 RS,因为可以通过Deployment实现管理RS
 - 这意味着可能永远不需要操作RS对象，而是使用Deployment替代管理

常用的WORKLOADS--DEPLOYMENT

- Deployment

- Deployment表示用户对Kubernetes集群的一次更新操。
- Deployment是一个比RS应用模式更广的API对象，它为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。
 - 创建、更新、滚动升级一个新的服务
 - 滚动升级，实际上是创建一个新的RS，然后逐渐将新的RS中副本数量增加到理想状态，同时将旧的RS中的副本数量逐渐减少到0的复合操作。
- 以Kubernetes的发展方向，未来对所有长期伺服型的业务管理，都会通过Deployment来管理。
- 用Deployment管理RS的一个主要原因是 Deployment支持回滚
- Deployment的典型使用场景如下：
 - 通过 Deployment 对象生成 ReplicaSet，并完成Pod副本创建
 - 通过Deployment升级或回滚Pod
 - 通过Deployment暂停或恢复发布

常用的WORKLOADS--DEPLOYMENT

- Deployment的更新策略

spec.minReadySeconds:新创建的pod状态为Ready持续的时间至少为指定时间，才认为Pod Available(Ready)

spec.strategy:

- Type可以是Recreate、RollingUpdate
- **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
- **rollingUpdate.maxSurge:**可以整数或者百分比，默认为desiredPods数量的25%，Scale Up新的RS时，按照比例计算出允许的MaxSurge，计算时向上取整。
- **rollingUpdate.maxUnavailable:**可以整数或者百分比，默认为desiredPods数量的25%，Scale Down旧的RS时，按照比例计算出允许的maxUnavailable，计算时向下取整。

```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    minReadySeconds: 5
7    strategy:
8      type: RollingUpdate
9      rollingUpdate:
10       maxSurge: 3
11       maxUnavailable: 2
12    replicas: 3
13    template:
14      metadata:
15        labels:
16          app: nginx
17      spec:
18        containers:
19          - name: nginx
20            image: nginx:1.7.9
21            ports:
22              - containerPort: 80
```

常用的WORKLOADS--DEPLOYMENT

- Deployment操作

- 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：

- 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 --cpu-percent=80`

- 暂停、恢复

- `kubectl rollout pause deployment/nginx-deployment`

- `kubectl rollout resume deploy nginx`

- 更新镜像：

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

- `kubectl edit deployment/nginx-deployment`

- 回滚：

- `kubectl rollout status deployment/nginx-deployment`

- `kubectl rollout undo deployment/nginx-deployment`

- `kubectl rollout history deployment/nginx-deployment`

常用的WORKLOADS--DEPLOYMENT

- 查看revision

如果我们创建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。

```
kubectl rollout history deployment/nginx-deployment --revision=1
```

```
[root@SGDLITVM0852 yaml]# kubectl rollout history deployment/nginx-deployment --revision=1
deployments "nginx-deployment" with revision #1
Pod Template:
  Labels:      app=nginx
              pod-template-hash=2315082692
  Annotations: kubernetes.io/change-cause=kubectl create --filename=Deployment-nginx.yaml --record=true
  Containers:
    nginx:
      Image:      nginx:1.7.9
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:      <none>
      Volumes:     <none>
```

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。

常用的WORKLOADS--DEPLOYMENT

- 回退到历史版本

我们可以决定回退当前的rollout到之前的版本:

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

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=1
```

```
[root@SGDLITVM0852 yaml]# kubectl rollout undo deployment/nginx-deployment --to-revision=1  
deployment.extensions/nginx-deployment
```

```
[root@SGDLITVM0852 yaml]# kubectl describe deployment/nginx-deployment  
Name: nginx-deployment  
Namespace: default  
CreationTimestamp: Thu, 11 Oct 2018 21:20:49 +0800  
Labels: app=nginx  
Annotations: deployment.kubernetes.io/revision=6  
             kubernetes.io/change-cause=kubectl create --filename=Deployment-nginx.yaml --record=true  
Selector: app=nginx  
Replicas: 10 desired | 10 updated | 10 total | 10 available | 0 unavailable  
StrategyType: RollingUpdate  
MinReadySeconds: 5  
RollingUpdateStrategy: 2 max unavailable, 3 max surge  
Pod Template:  
  Labels: app=nginx  
  Containers:  
    nginx:  
      Image: nginx:1.7.9  
      Port: 80/TCP  
      Host Port: 0/TCP
```

常用的WORKLOADS--STATEFULSET

- 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）

常用的WORKLOADS--STATEFULSET

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

常用的WORKLOADS--DAEMONSET

- 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

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

常用的WORKLOADS--DAEMONSET

常用操作

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>
```

常用的WORKLOADS--INGRESS CONTROLLER

- 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
- 外部访问的时候，**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

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。

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。

```
1  ---
2  apiVersion: extensions/v1beta1
3  kind: Ingress
4  metadata:
5    name: service-ingress
6    namespace: default
7  spec:
8    rules:
9      - host: foo.bar.com
10      http:
11        paths:
12          - backend:
13              serviceName: s1
14              servicePort: 80
15      - host: bar.foo.com
16      http:
17        paths:
18          - backend:
19              serviceName: s2
20              servicePort: 80
```

常用的WORKLOADS—JOB

- 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

常用的WORKLOADS—JOB

- Job 模式

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+

常用的WORKLOADS—JOB

- Job Spec

- 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

常用的WORKLOADS—JOB

- 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--runtimeconfig=batch/v2alpha1
 - 从v1.8开始，API升级到batch/v1beta1，并在apiserver中默认开启
- 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

配置管理对象--CONFIGMAP

- 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。

- 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 yamlfile`创建。

配置管理对象--CONFIGMAP

- `kubectl create configmap demo-config1 --from-literal=key=value`

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config1 -o json|jq
{
  "apiVersion": "v1",
  "data": {
    "key": "value"
  },
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-10-11T14:58:00Z",
    "name": "demo-config1",
    "namespace": "default",
    "resourceVersion": "4578683",
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config1",
    "uid": "0be1f79c-cd66-11e8-bb3d-005056b02fab"
  }
}
```

- `echo -e "a=b\nc=d" | tee config.env`
- `kubectl create configmap demo-config2 --from-env-file=config.env`

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config2 -o json|jq
{
  "apiVersion": "v1",
  "data": {
    "a": "b",
    "c": "d"
  },
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-10-11T14:59:42Z",
    "name": "demo-config2",
    "namespace": "default",
    "resourceVersion": "4578898",
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config2",
    "uid": "4879b14c-cd66-11e8-bb3d-005056b02fab"
  }
}
```

配置管理对象--CONFIGMAP

- 从key-value字符串创建

```
kubectl create configmap demo-config1 --  
from-literal=key=value
```

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config1 -o json|jq  
{  
  "apiVersion": "v1",  
  "data": {  
    "key": "value"  
  },  
  "kind": "ConfigMap",  
  "metadata": {  
    "creationTimestamp": "2018-10-11T14:58:00Z",  
    "name": "demo-config1",  
    "namespace": "default",  
    "resourceVersion": "4578683",  
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config1",  
    "uid": "0be1f79c-cd66-11e8-bb3d-005056b02fab"  
  }  
}
```

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config2 -o json|jq  
{  
  "apiVersion": "v1",  
  "data": {  
    "a": "b",  
    "c": "d"  
  },  
  "kind": "ConfigMap",  
  "metadata": {  
    "creationTimestamp": "2018-10-11T14:59:42Z",  
    "name": "demo-config2",  
    "namespace": "default",  
    "resourceVersion": "4578898",  
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config2",  
    "uid": "4879b14c-cd66-11e8-bb3d-005056b02fab"  
  }  
}
```

配置管理对象--CONFIGMAP

- 从目录创建

mkdir config

echo a>config/a

echo b>config/b

kubectl create configmap demo-config3 --
from-file=config/

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config3 -o json|jq
{
  "apiVersion": "v1",
  "data": {
    "a": "a\n",
    "b": "b\n"
  },
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-10-11T15:05:54Z",
    "name": "demo-config3",
    "namespace": "default",
    "resourceVersion": "4579687",
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config3",
    "uid": "26540a2b-cd67-11e8-bb3d-005056b02fab"
  }
}
```

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: demo-config
5    namespace: default
6    labels:
7      app: demo
8  data:
9    FQDN: "a.b.c"
10   PORT: "443"
11   PROTOCOL: "https"
```

```
[root@GeorgeF5v4 output]# kubectl get cm demo-config4 -o json|jq
{
  "apiVersion": "v1",
  "data": {
    "FQDN": "a.b.c",
    "PORT": "443",
    "PROTOCOL": "https"
  },
  "kind": "ConfigMap",
  "metadata": {
    "creationTimestamp": "2018-10-11T15:08:10Z",
    "labels": {
      "app": "demo"
    },
    "name": "demo-config4",
    "namespace": "default",
    "resourceVersion": "4579976",
    "selfLink": "/api/v1/namespaces/default/configmaps/demo-config4",
    "uid": "774327be-cd67-11e8-bb3d-005056b02fab"
  }
}
```

配置管理对象--CONFIGMAP

- ConfigMap用途

ConfigMap可以存储文件或key/value，使用中可以直接被yamlfile引用：

- 用作环境变量

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: demo-cm-env
5    namespace: default
6  data:
7    env1: "env-value1"
8    env2: "env-value2"
9
10 ---
11 apiVersion: v1
12 kind: Pod
13 metadata:
14   name: test-pod
15 spec:
16   containers:
17   - name: test-container
18     image: busybox
19     command: [ "/bin/sh", "-c", "env" ]
20     env:
21     - name: ENV1
22       valueFrom:
23         configMapKeyRef:
24           name: demo-cm-env
25           key: env1
26     - name: ENV2
27       valueFrom:
28         configMapKeyRef:
29           name: demo-cm-env
30           key: env2
31   restartPolicy: Never
```

```
[root@SGDLITVM0852 yam1]# kubectl logs test-pod
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://172.17.17.1:443
HOSTNAME=test-pod
SHLV1=1
HOME=/root
KUBE_REGISTRY_PORT_5000_TCP_ADDR=172.17.17.52
KUBE_REGISTRY_PORT_5000_TCP_PORT=5000
KUBE_REGISTRY_PORT_5000_TCP_PROTO=tcp
KUBE_REGISTRY_SERVICE_PORT_REGISTRY=5000
KUBERNETES_PORT_443_TCP_ADDR=172.17.17.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
ENV1=env-value1
ENV2=env-value2
KUBERNETES_PORT_443_TCP_PORT=443
```

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: demo-cm-file
5    namespace: default
6  data:
7    file1: "file1 text\n"
8    file2: "file2 text\n"
9
10 ---
11 apiVersion: v1
12 kind: Pod
13 metadata:
14   name: test-pod-file
15 spec:
16   containers:
17   - name: test-container
18     image: busybox
19     command: [ "/bin/sh", "-c", "cat /etc/config/*" ]
20     volumeMounts:
21     - name: config-volume
22       mountPath: /etc/config
23   volumes:
24   - name: config-volume
25     configMap:
26       name: demo-cm-file
27   restartPolicy: Never
```

```
[root@SGDLITVM0852 yam1]# kubectl logs test-pod-file
file1 text
file2 text
[root@SGDLITVM0852 yam1]#
```

配置管理对象--SECRET

- Secret

Secret类型

Opaque : base64编码格式的Secret, 用来存储密码、密钥等; 但数据也通过base64 - decode解码得到原始数据, 所有加密性很弱。

- `kubernetes.io/dockerconfigjson` : 用来存储私有docker registry的认证信息。
- `kubernetes.io/service-account-token` : 用于被serviceaccount引用。serviceaccount创建时

Kubernetes会默认创建对应的secret。Pod如果使用了serviceaccount, 对应的secret会自动挂载到Pod的`/run/secrets/kubernetes.io/serviceaccount`目录中。

配置管理对象--SECRET

- Secret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-encryptionprovider-config。

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

```
1 kind: EncryptionConfig
2 apiVersion: v1
3 resources:
4   - resources:
5     - secrets
6     providers:
7       - aescbc:
8         keys:
9           - name: key1
10             secret: c2VjcmV0IGlzIHNIY3VyZQ==
11           - name: key2
12             secret: dGhpcyBpcyBwYXNzd29yZA==
13       - identity: {}
14       - aesgcm:
15         keys:
16           - name: key1
17             secret: c2VjcmV0IGlzIHNIY3VyZQ==
18           - name: key2
19             secret: dGhpcyBpcyBwYXNzd29yZA==
20       - secretbox:
21         keys:
22           - name: key1
23             secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
```

配置管理对象—SECRET VS CONFIGMAP

- Secret与ConfigMap对比

相同点:

- key/value的形式
- 属于某个特定的namespace
- 可以导出到环境变量
- 可以通过目录/文件形式挂载(支持挂载所有key和部分key)

不同点:

- Secret可以被ServerAccount关联(使用)
- Secret可以存储register的鉴权信息, 用在ImagePullSecret参数中, 用于拉取私有仓库的镜像
- Secret支持Base64加密
- Secret分为Opaque, kubernetes.io/Service Account, kubernetes.io/dockerconfigjson三种类型, Configmap不区分类型
- Secret文件存储在tmpfs文件系统中, Pod删除后Secret文件也会对应的删除。