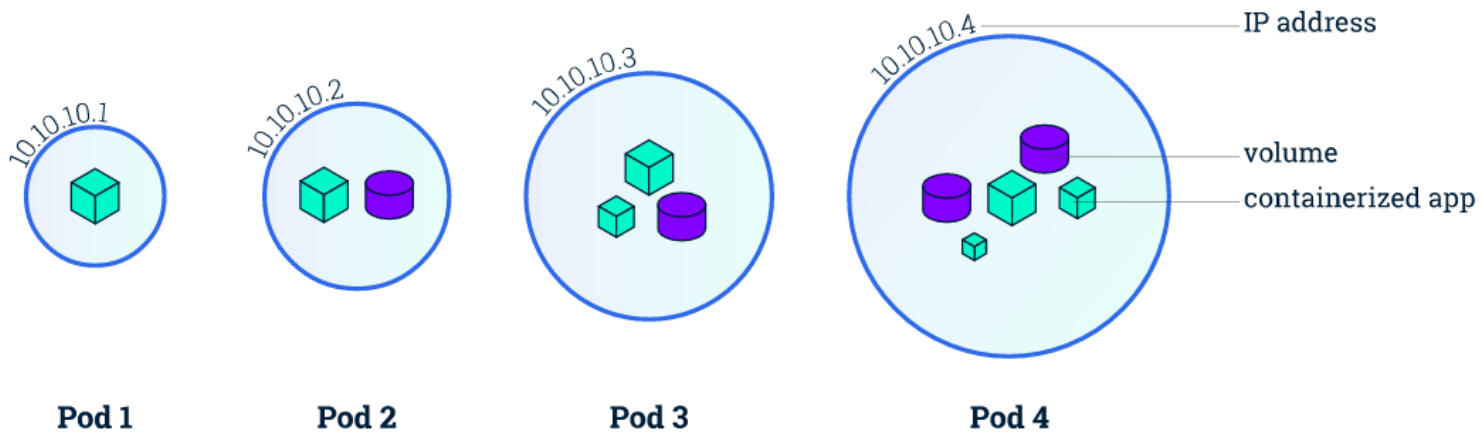
第二部分 Kubernetes对象详解

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。
-

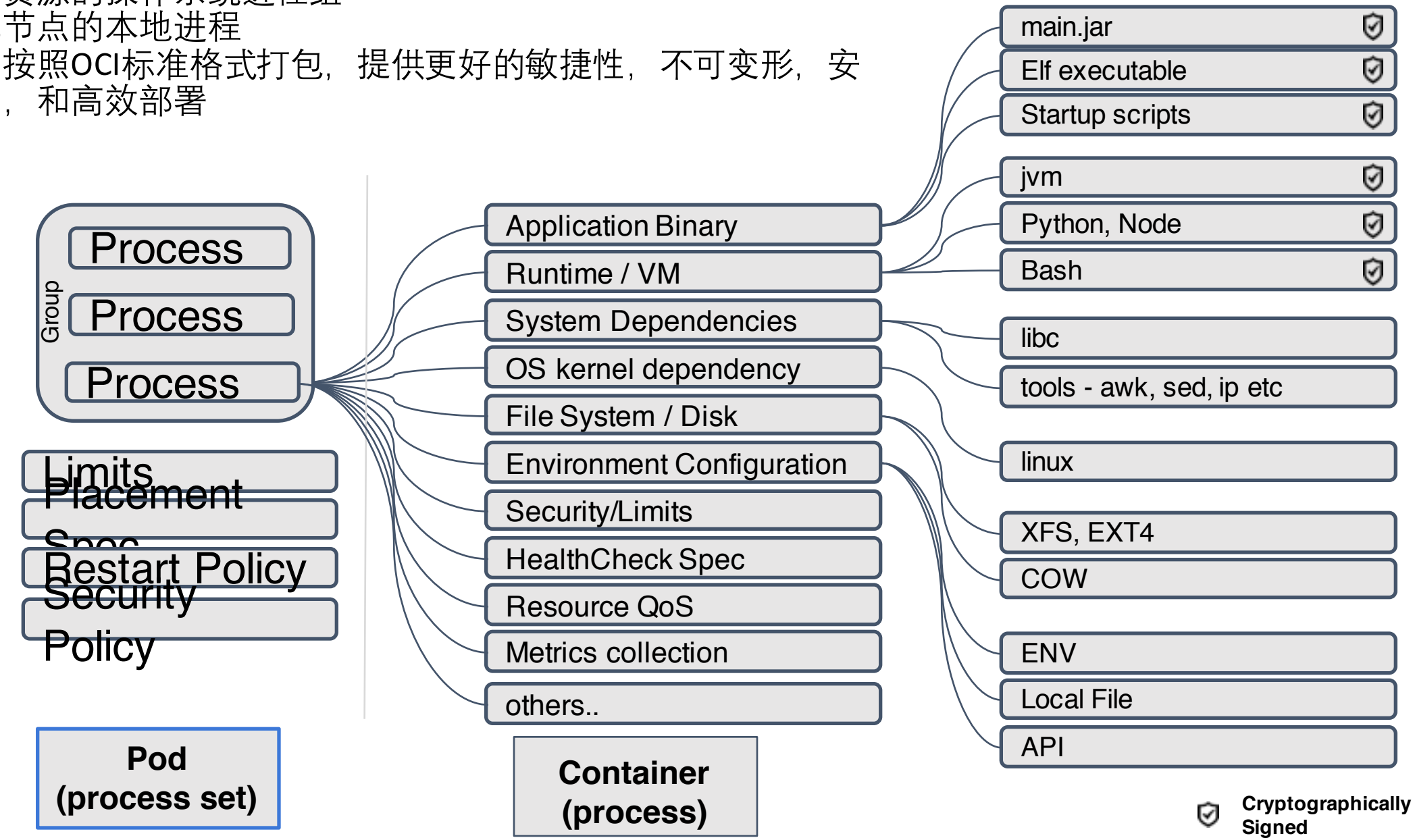


Pod的特征

- 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）

Pod

- 共享资源的操作系统进程组
- 计算节点的本地进程
- 可以按照OCI标准格式打包，提供更好的敏捷性，不可变形，安全性，和高效部署



Pod Spec

```yaml
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: {}
```

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。

sample

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "sh", "-c"]
      args:
      - env
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
```

```
env:
    - name: MY_NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MY_POD_SERVICE_ACCOUNT
      valueFrom:
        fieldRef:
          fieldPath: spec.serviceAccountName
```

```
    - name: MY_CPU_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: requests.cpu
    - name: MY_CPU_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: limits.cpu
    - name: MY_MEM_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: requests.memory
    - name: MY_MEM_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: limits.memory
  restartPolicy: Never
```

(3) 集群中服务的信息

- 容器的环境变量中还可以引用容器运行前创建的所有服务的信息，比如默认的kubernetes服务对应以下环境变量：

```
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.1
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.0.0.1:443 KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
```

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。

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。

使用主机名空间

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostIPC: true
  hostPID: true
  hostNetwork: true
  containers:
  - image: busybox
```

设置Pod的hostname

- 通过spec.hostname参数实现，如果未设置默认使用metadata.name参数的值作为Pod的hostname。
- 设置Pod的子域名
- 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.my-namespace.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）

```
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，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。

资源限制示例

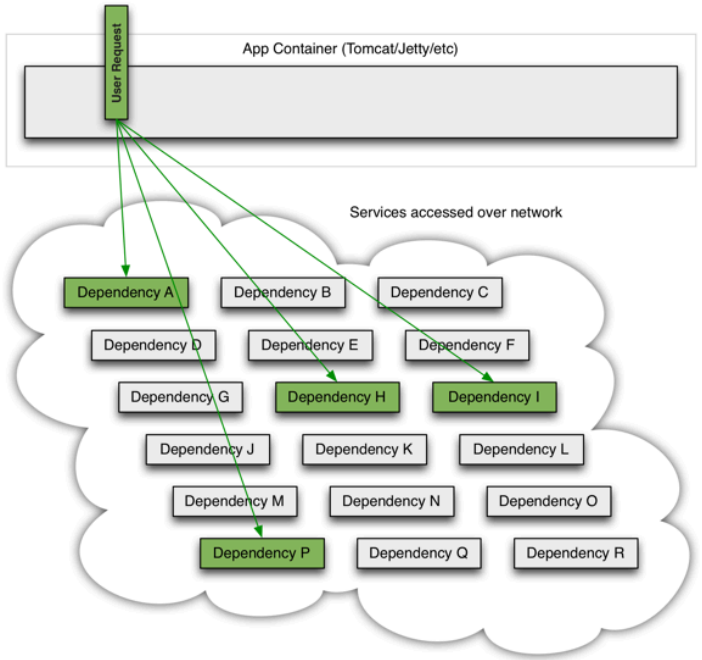
```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        requests:
          cpu: "300m"
          memory: "56Mi"
        limits:
          cpu: "1"
          memory: "128Mi"
```

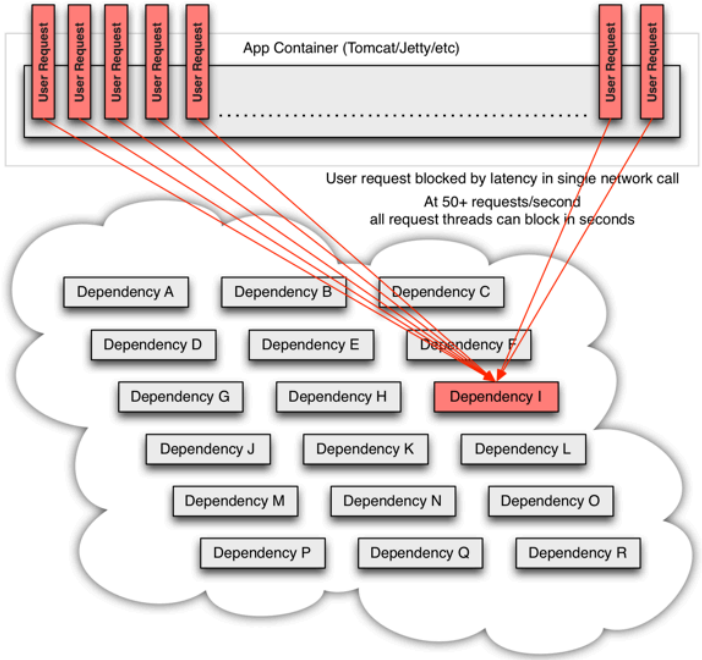
健康检查

- 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
  - SLA
- 对于要发布的服务，需要两个级别的健康检查：
  - 节点健康指标：CPU，内存，网络吞吐；
  - 应用级别健康指标：
    - Selfclosure
    - External
    - Functional
- LB上的TCP/ECV check

服务容错

- 当企业微服务化以后，服务之间会有错综复杂的依赖关系，例如，一个前端请求一般会依赖于多个后端服务。在实际生产环境中，服务往往不是百分百可靠，服务可能会出错或者产生延迟，如果一个应用不能对其依赖的故障进行容错和隔离，那么该应用本身就处在被拖垮的风险中。在一个高流量的网站中，某个单一后端一旦发生延迟，可能在数秒内导致所有资源被耗尽，造成所谓的雪崩效应，严重时可致整个网站瘫痪。





服务容错

- 电路熔断器模式(Circuit Breaker Patten),
  - 原理类似于家里的电路熔断器。在分布式系统中应用电路熔断器模式后，当目标服务慢或者大量超时，调用方能够主动熔断，以防止服务被进一步拖垮；如果情况又好转了，电路又能自动恢复，这就是所谓的弹性容错，系统有自恢复能力。正常状态下，电路处于关闭状态(Closed)，如果调用持续出错或者超时，电路被打开进入熔断状态(Open)，后续一段时间内的所有调用都会被拒绝(Fail Fast)，一段时间以后，保护器会尝试进入半熔断状态(Half-Open)，允许少量请求进来尝试，如果调用仍然失败，则回到熔断状态，如果调用成功，则回到电路闭合状态。
- 舱壁隔离模式(Bulkhead Isolation Pattern)
  - 像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。线程隔离(Thread Isolation)就是舱壁隔离模式的一个例子，假定一个应用程序A调用了Svc1/Svc2/Svc3三个服务，且部署A的容器一共有120个工作线程，采用线程隔离机制，可以给对Svc1/Svc2/Svc3的调用各分配40个线程，当Svc2慢了，给Svc2分配的40个线程因慢而阻塞并最终耗尽，线程隔离可以保证给Svc1/Svc3分配的80个线程可以不受影响，如果没有这种隔离机制，当Svc2慢的时候，120个工作线程会很快全部被对Svc2的调用吃光，整个应用程序会全部慢下来。

服务容错

- 限流(Rate Limiting/Load Shedder)
  - 服务总有容量限制，没有限流机制的服务很容易在突发流量(秒杀，大促)时被冲垮。限流通常指对服务限定并发访问量，比如单位时间只允许100个并发调用，对超过这个限制的请求要拒绝并回退。
- 回退(fallback)
  - 在熔断或者限流发生的时候，应用程序的后续处理逻辑是什么？回退是系统的弹性恢复能力，常见的处理策略有，直接抛出异常，也称快速失败(Fail Fast)，也可以返回空值或缺省值，还可以返回备份数据，如果主服务熔断了，可以从备份服务获取数据。

健康检查

- 为了确保容器在部署后确实处在正常运行状态，Kubernetes提供了两种探针（Probe）来探测容器的状态：
  - LivenessProbe：探测应用是否处于健康状态，如果不健康则删除并重新创建容器
  - ReadinessProbe：探测应用是否启动完成并且处于正常服务状态，如果不正常则不会接收来自Kubernetes Service的流量
- Kubernetes支持三种方式来执行探针：
  - exec：在容器中执行一个命令，如果命令退出码返回0则表示探测成功，否则表示失败
  - tcpSocket：对指定的容器IP及端口执行一个TCP检查，如果端口是开放的则表示探测成功，否则表示失败
  - httpGet：对指定的容器IP、端口及路径执行一个HTTP Get请求，如果返回的状态码在[200,400)之间则表示探测成功，否则表示失败

健康检查示例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
    containers:
    - image: nginx
      imagePullPolicy: Always
      name: http
      livenessProbe:
        httpGet:
          path: /
          port: 80
          httpHeaders:
          - name: X-Custom-Header
            value: Awesome
        initialDelaySeconds: 15
        timeoutSeconds: 1
      readinessProbe:
        exec:
          command:
          - cat
          - /usr/share/nginx/html/index.html
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

```
- name: goproxy
    image: gcr.io/google_containers/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

初始化容器

- 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： 容器创建后立即执行，注意由于是异步执行，它无法保证一定在ENTRYPOINT之前运行。如果失败，容器会被杀死，并根据RestartPolicy决定是否重启
- preStop：容器终止前执行，常用于资源清理。如果失败，容器同样也会被杀死
- 而钩子的回调函数支持两种方式：
- exec：在容器内执行命令，如果命令的退出状态码是0表示执行成功，否则表示失败
- httpGet：向指定URL发起GET请求，如果返回的HTTP状态码在[200, 400)之间表示请求成功，否则表示失败

postStart和preStop钩子示例

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        httpGet:
          path: /
          port: 80
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

使用Capabilities

- 默认情况下，容器都是以非特权容器的方式运行。比如，不能在容器中创建虚拟网卡、配置虚拟网络。
- Kubernetes提供了修改Capabilities的机制，可以按需要给容器增加或删除。

Capabilities示例

```
apiVersion: v1
kind: Pod
metadata:
  name: cap-pod
spec:
  containers:
  - name: friendly-container
    image: "alpine:3.4"
    command: ["/bin/sleep", "3600"]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        drop:
        - KILL
```

限制网络带宽

- 可以通过给Pod增加kubernetes.io/ingress-bandwidth和kubernetes.io/egress-bandwidth这两个annotation来限制Pod的网络带宽
- 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。

```
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
```

限制网络带宽原理

- 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

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
```

Pod spec对host的支持

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

```
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"
```

HugePages

- v1.8+支持给容器分配HugePages，资源格式为hugepages-<size>（如hugepages-2Mi）。使用前要配置
- 开启--feature-gates="HugePages=true"
- 在所有 Node 上面预分配好 HugePage，以便 Kubelet 统计所在 Node 的 HugePage 容量

```
apiVersion: v1
kind: Pod
metadata:
 generateName: hugepages-volume-
spec:
 containers:
 - image: fedora:latest
   command:
   - sleep
   - inf
   name: example
   volumeMounts:
   - mountPath: /hugepages
     name: hugepage
   resources:
     limits:
       hugepages-2Mi: 100Mi
 volumes:
 - name: hugepage
   emptyDir:
     medium: HugePages
```

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。
- 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。

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。

ReplicationController/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。

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）

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 Servic和StatefulSet必须在相同的namespace
  - .cluster.local为Cluster Domain，

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"
 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
```

Statefulset创建的对象

```
# 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                                     CAPACITY   ACCESSMODES   AGE
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
```

Dns 配置

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	Pod DNS	Pod Hostname
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.svc.cluster.local	web-{0..N-1}
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.svc.cluster.local	web-{0..N-1}
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.foo.svc.kube.local	web-{0..N-1}

StatefulSet的更新策略

- 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。
  - 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）。

Statefulset Pod管理策略

- 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）

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

Daemonset更新策略

- 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

- # 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>

Namespace

- Namespace是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。常见的pod, service, replication controller和deployment等都是属于某一个namespace的（默认是default），而node, persistent volume，namespace等资源则不属于任何namespace。
- Namespace常用来隔离不同的用户，比如Kubernetes自带的服务一般运行在kube-system namespace中。

Namespace操作

- kubectl可以通过--namespace或者-n选项指定namespace。如果不指定，默认为default。查看操作下,也可以通过设置--all-namespace=true来查看所有namespace下的资源。
- **查询**
  - $ kubectl get namespaces
  - NAME　　STATUS　AGE
  - default　　Active　11d
  - kube-system　Active　11d
- 注意：namespace包含两种状态"Active"和"Terminating"。在namespace删除过程中，namespace状态被设置成"Terminating"。

创建

- (1) 命令行直接创建
- $ kubectl create namespace new-namespace

- (2) 通过文件创建
- $ cat my-namespace.yaml
- apiVersion: v1
- kind: Namespace
- metadata:
- name: new-namespace

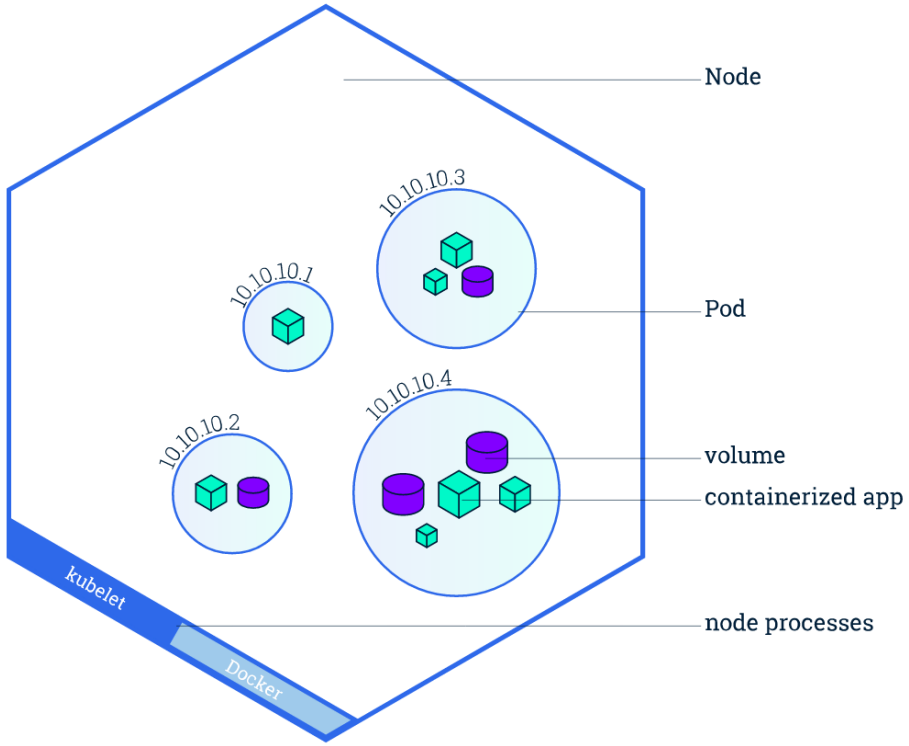
- $ kubectl create -f ./my-namespace.yaml

删除

- $ kubectl delete namespaces new-namespace
- 删除一个namespace会自动删除所有属于该namespace的资源。
- default和kube-system命名空间不可删除。
- PersistentVolume是不属于任何namespace的，但PersistentVolumeClaim是属于某个特定namespace的。
- Event是否属于namespace取决于产生event的对象。
- 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

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yaml所示），但Kubernetes也只是去检查是否真的是有这么一个Node，如果检查失败，也不会往上调度Pod。
- 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。

```
kind: Node
apiVersion: v1
metadata:
  name: 10-240-79-157
  labels:
    name: my-first-k8s-node
```

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

Node维护模式

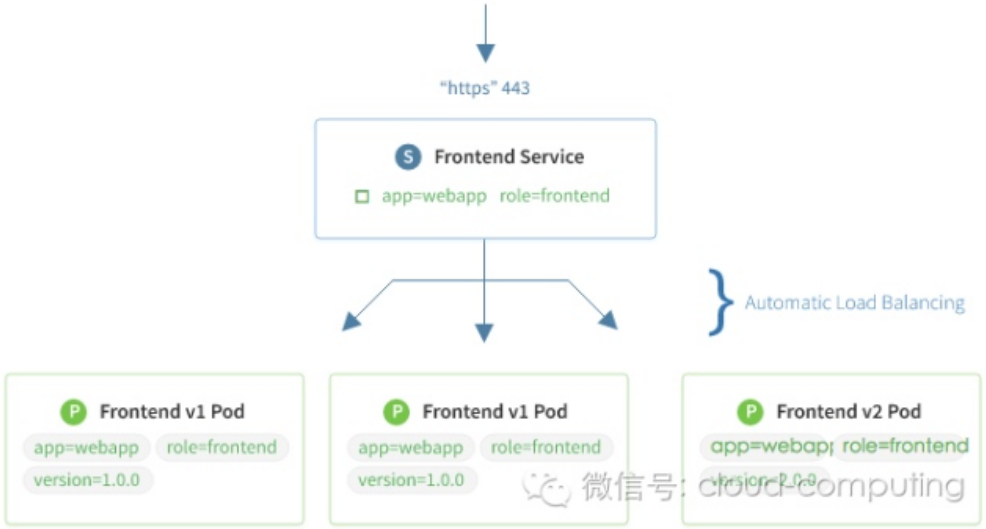
- 标志Node不可调度但不影响其上正在运行的Pod，这种维护Node时是非常有用的
  - kubectl cordon $NODENAME

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.externlName设定）。需要kube-dns版本在1.7以上。
- 另外，也可以将已有的服务以Service的形式加入到Kubernetes集群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

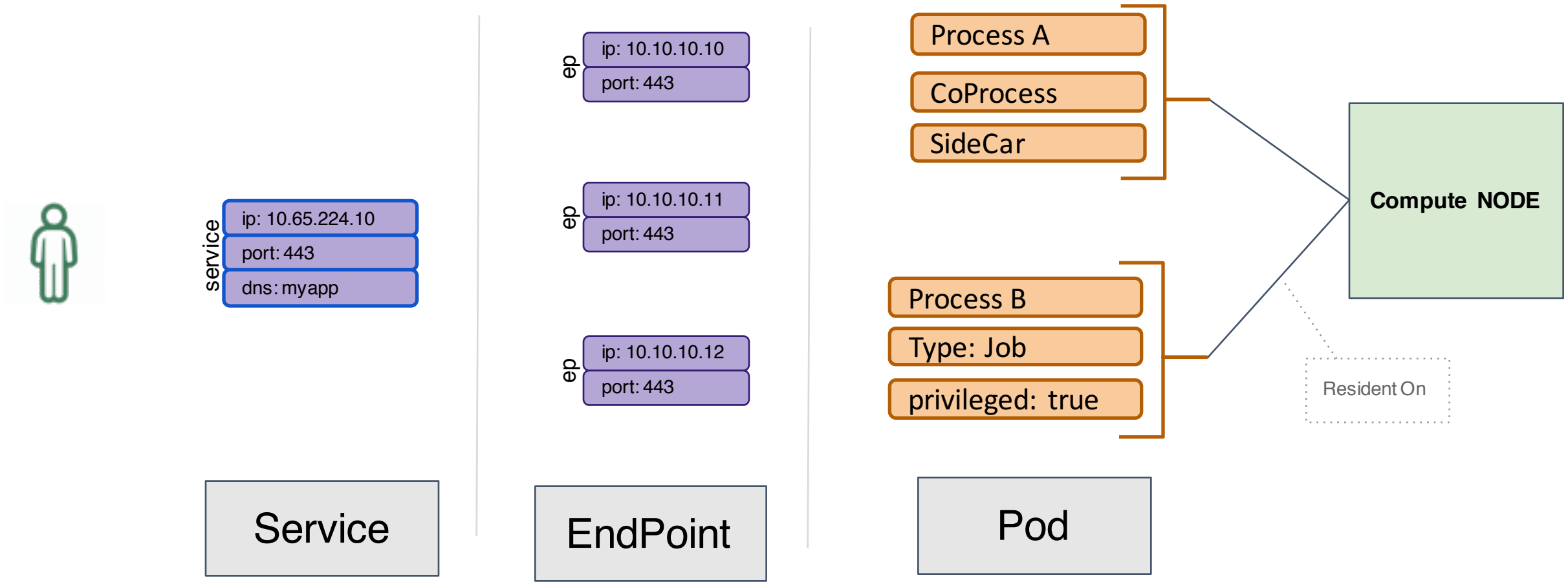
```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx
  name: nginx
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: nginx
  sessionAffinity: None
  type: ClusterIP
```

都发生了什么

```
# 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>
```

Service 和Endpoints



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
- 没有Selector，endpoint不会被自动创建

通过自定义endpoint指定远程服务

- 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: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

通过DNS转发指定远程服务

- 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，通过CNAME记录处理
  - 指定Selectors，通过DNS A记录设置后端endpoint列表

Headless service示例

```
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
```

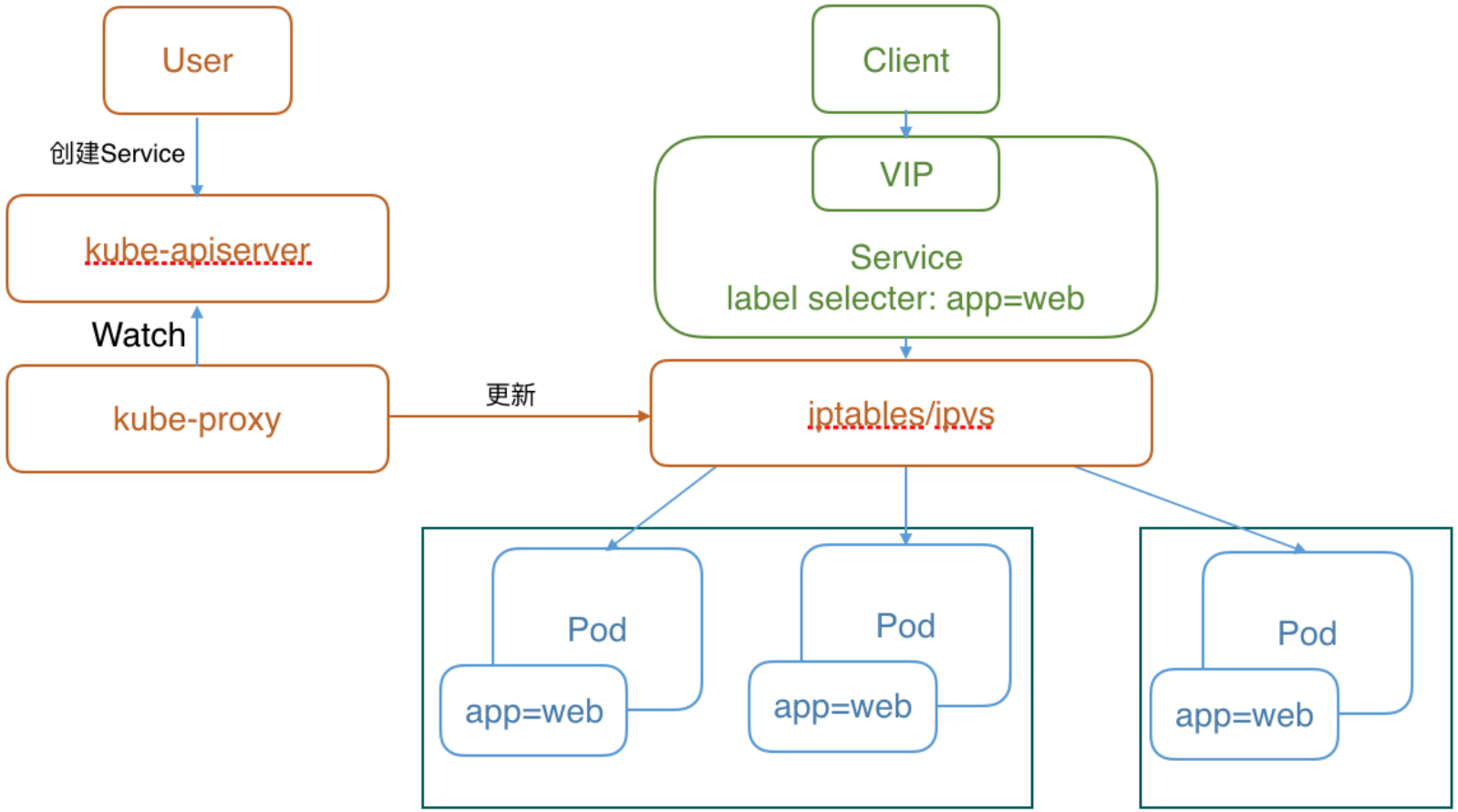
验证headless服务

- # 查询创建的nginx服务
- $ kubectl get service --all-namespaces=true
  - NAMESPACE NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
  - default nginx None <none> 80/TCP 5m
  - kube-system kube-dns 172.26.255.70 <none> 53/UDP,53/TCP 1d
- $ kubectl get pod
  - NAME READY STATUS RESTARTS AGE IP NODE
  - nginx-2204978904-6o5dg 1/1 Running 0 14s 172.26.2.5 10.0.0.2
  - nginx-2204978904-qyilx 1/1 Running 0 14s 172.26.1.5 10.0.0.8
- $ 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

保留源IP

- 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。

工作原理


User
创建Service
kube-apiserver
Watch
kube-proxy
更新
Client
VIP
Service
label selecter: app=web
iptables/ipvs
Pod
app=web
Pod
app=web
Pod
app=web

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示例

- 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。

```
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：
-

Ingress示例

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

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

emptyDir
hostPath
gcePersistentDisk
awsElasticBlockStore
nfs
iscsi
flocker
glusterfs
rbd
cephfs
gitRepo
secret
persistentVolumeClaim
downwardAPI
azureFileVolume
azureDisk
vsphereVolume
Quobyte
PortworxVolume
ScaleIO
FlexVolume
StorageOS
local

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

- subpath可以指定在共享volume里的子目录
- Pod的多个容器使用同一个Volume时，subPath非常有用，避免互相影响

```
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

```
- 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

本地存储配额

- Kubernetes根据storage.kubernetes.io/scratch的大小来调度本地存储空间，而根据storage.kubernetes.io/overlay来调度容器的存储。比如

为容器请求64MB的可写层存储空间

```
apiVersion: v1
kind: Pod
metadata:
  name: ls1
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: busybox
    command: ["df"]
    resources:
      requests:
        storage.kubernetes.io/overlay: 64Mi
```

为empty请求64MB的存储空间

```
apiVersion: v1
kind: Pod
metadata:
  name: ls1
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: busybox
    command: ["df"]
    volumeMounts:
    - name: data
      mountPath: /data
  volumes:
  - name: data
    emptyDir:
      sizeLimit: 64Mi
```

Mount传递

- 在Kubernetes中，Volume Mount默认是私有的，但从v1.8开始，Kubernetes支持配置Mount传递（mountPropagation）。它支持两种选项
- HostToContainer：这是开启MountPropagation=true时的默认模式，等效于rslave模式，即容器可以看到Host上面在该volume内的任何新Mount操作
- Bidirectional：等效于rshared模式，即Host和容器都可以看到对方在该Volume内的任何新Mount操作。该模式要求容器必须运行在特权模式（即securityContext.privileged=true）
- 注意：
- 使用Mount传递需要开启--feature-gates=MountPropagation=true

持久化卷

- PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 提供了方便的持久化卷：PV 提供网络存储资源，而 PVC 请求存储资源。这样，设置持久化的工作流包括配置底层文件系统或者云数据卷、创建持久性数据卷、最后创建 PVC 来将 Pod 跟数据卷关联起来。PV 和 PVC 可以将 pod 和数据卷解耦，pod 不需要知道确切的文件系统或者支持它的持久化引擎。

Volume生命周期

- Volume的生命周期包括5个阶段
  - 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
  - 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，可以保留PV以便下次使用，也可以直接从云存储中删除

Volume的状态

- 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
```

访问模式与回收策略

- 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，即PVC释放卷的时候PV该如何操作）也有三种
  - Retain，不清理, 保留Volume（需要手动清理）
  - Recycle，删除数据，即rm -rf /thevolume/*（只有NFS和HostPath支持）
  - Delete，删除存储资源，比如删除AWS EBS卷（只有AWS EBS, GCE PD, Azure Disk和Cinder支持）

StorageClass

- 上面通过手动的方式创建了一个NFS Volume，这在管理很多Volume的时候不太方便。Kubernetes还提供了StorageClass来动态创建PV，不仅节省了管理员的时间，还可以封装不同类型的存储供PVC选用。
- StorageClass包括四个部分
  - provisioner：指定Volume插件的类型，包括内置插件（如kubernetes.io/glusterfs）和外部插件（如external-storage提供的ceph.com/cephfs）。
  - mountOptions：指定挂载选项，当PV不支持指定的选项时会直接失败。比如NFS支持hard和nfsvers=4.1等选项。
  - parameters：指定provisioner的选项，比如kubernetes.io/aws-ebs支持type、zone、iopsPerGB等参数。
  - reclaimPolicy：指定回收策略，同PV的回收策略。
- 在使用PVC时，可以通过DefaultStorageClass准入控制设置默认StorageClass，即给未设置storageClassName的PVC自动添加默认的StorageClass。而默认的StorageClass带有annotation storageclass.kubernetes.io/is-default-class=true。

修改默认StorageClass

- 取消原来的默认StorageClass
  - kubectl patch storageclass <default-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"}}}'

OpenStack Cinder示例

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  type: fast
  availability:  nova
```

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]}
```

本地数据卷

- 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 v1.7之前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

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-local-pv
  annotations:
    "volume.alpha.kubernetes.io/node-affinity": '{
      "requiredDuringSchedulingIgnoredDuringExecution": {
        "nodeSelectorTerms": [
          { "matchExpressions": [
            { "key": "kubernetes.io/hostname",
              "operator": "In",
              "values": ["example-node"]
            }
          ]}
         ]}
        }',
spec:
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
```

创建PVC：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: example-local-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: local-storage
```

创建Pod，引用PVC：

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: example-local-claim
```

最佳实践

- 推荐为每个存储卷分配独立的磁盘，以便隔离IO请求
- 推荐为每个存储卷分配独立的分区，以便隔离存储空间
- 避免重新创建同名的Node，否则会导致新Node无法识别已绑定旧Node的PV
- 推荐使用UUID而不是文件路径，以避免文件路径误配的问题
- 对于不带文件系统的块存储，推荐使用唯一ID（如/dev/disk/by-id/），以避免块设备路径误配的问题

Deployment

- 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。
- Kubernetes v1.7及以前API版本使用extensions/v1beta1
- Kubernetes v1.8的API版本升级到apps/v1beta2

示例

```
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
```

使用

- 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的典型应用场景

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

Deployment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。

典型用例

- 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。

Deployment Spec

- Pod Template
- Replicas
- Selector
- .spec.strategy
  - .spec.strategy.type==Recreate
    - 在创建出新的Pod之前会先杀掉所有已存在的Pod。
  - .spec.strategy.type==RollingUpdate
    - 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%）。
    - 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%）。
- Progress Deadline Seconds
- Min Ready Seconds
- Rollback To
- Revision
- Revision History Limit

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
- 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

输出

- 输出结果表明我们希望的repalica数是3（根据deployment中的.spec.replicas配置）当前replica数（ .status.replicas）是0, 最新的replica数（.status.updatedReplicas）是0，可用的replica数（.status.availableReplicas）是0。
- 过几秒后再执行get命令，将获得如下输出：
- $ kubectl get deployments NAME DESIRED CURRENT UP-TO-DATE AVAILABLE 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。

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

- $ 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规则

- Deployment可以保证在升级时只有一定数量的Pod是down的。默认的，它会确保至少有比期望的Pod数量少一个的Pod是up状态（最多一个不可用）。
- Deployment同时也可以确保只创建出超过期望数量的一定数量的Pod。默认的，它会确保最多比期望的Pod数量多一个的Pod是up的（最多1个surge）。
- 在未来的Kuberentes版本中，将从1-1变成25%-25%。

Rollover（多个rollout并行）

- 每当Deployment controller观测到有新的deployment被创建时，如果没有已存在的Replica Set来创建期望个数的Pod的话，就会创建出一个新的Replica Set来做这件事。已存在的Replica Set控制label匹配.spec.selector但是template跟.spec.template不匹配的Pod缩容。最终，新的Replica Set将会扩容出.spec.replicas指定数目的Pod，旧的Replica Set会缩容到0。
- 如果你更新了一个的已存在并正在进行中的Deployment，每次更新Deployment都会创建一个新的Replica Set并扩容它，同时回滚之前扩容的Replica Set——将它添加到旧的Replica Set列表，开始缩容。
- 例如，假如你创建了一个有5个niginx:1.7.9 replica的Deployment，但是当还只有3个nginx:1.7.9的replica创建出来的时候你就开始更新含有5个nginx:1.9.1 replica的Deployment。在这种情况下，Deployment会立即杀掉已创建的3个nginx:1.7.9的Pod，并开始创建nginx:1.9.1的Pod。它不会等到所有的5个nginx:1.7.9的Pod都创建完成后才开始执行滚动更新。

回滚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。

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:1.91
- 因为我们创建Deployment的时候使用了—recored参数可以记录命令，我们可以很方便的查看每次revison的变化。

查看revision

- 因为我们创建Deployment的时候使用了—recored参数可以记录命令，我们可以很方便的查看每次revison的变化。
- 可以通过设置.spec.revisonHistoryLimit项来指定deployment最多保留多少revison历史记录。默认的会保留所有的revision；如果将该项设置为0，Deployment就不允许回退了。

```
$ 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

- 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

Deployment扩容

- 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 --cpu-percent=80 deployment "nginx-deployment" autoscaled

暂停和恢复Deployment

- 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。
- 使用以下命令暂停Deployment：
  - $ kubectl rollout pause deployment/nginx-deployment
  - deployment "nginx-deployment" paused
- 然后更新Deplyment中的镜像：
  - $ kubectl set image deploy/nginx nginx=nginx:1.9.1
  - deployment "nginx-deployment" image updated
- 恢复这个Deployment
  - $ kubectl rollout resume deploy nginx
  - deployment "nginx" resumed

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

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+

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

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
- 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

CronJob 示例

kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"

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

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 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Spec文件创建

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

用作环境变量

```yaml
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
```

挂载volume

```
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
```

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

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。
- 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

```
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=
```

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。

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.serviceAccountName为default（除非指定了其他ServiceAccout）
- 验证Pod引用的service account已经存在，否则拒绝创建
- 如果Pod没有指定ImagePullSecrets，则把service account的ImagePullSecrets加到Pod中
- 每个container启动后都会挂载该service account的token和ca.crt到/var/run/secrets/kubernetes.io/serviceaccount/
- $ kubectl exec nginx-3137573019-md1u2 ls /run/secrets/kubernetes.io/serviceaccount
    - ca.crt
    - namespace
    - 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

RBAC Sample

```
# 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
```

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

- 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

- 应用到Pod内所有容器，并且还会影响Volume（包括fsGroup和selinuxOptions）。

```
SELinux Example
 securityContext:
    seLinuxOptions:
      user: system_u
      role: object_r
      type: svirt_sandbox_file_t
      level: s0:c100,c200
# ps auxZ | grep sleep
system_u:object_r:svirt_sandbox_file_t:s0:c100,c200  root 8633 0.1 0.0 4348 636 ? Ss 10:12  0:00 sleep 3600
```

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

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:
  - '*'
```

Hostpath访问白名单

- 定义可访问的主机路径白名单
- 空列表代表无限制

```yaml
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: custom-paths
spec:
  allowedHostPaths:
    # This allows "/foo", "/foo/", "/foo/bar"
etc., but
    # disallows "/fool", "/etc/foo" etc.
    - pathPrefix: "/foo"
```

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

```yaml
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: {}
```

效果

- 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
-

开启资源配额功能

- 首先，在API Server启动时配置ResourceQuota adminssion control
- 然后，在namespace中创建一个ResourceQuota对象

资源配额的类型

- 计算资源，包括cpu和memory
  - cpu, limits.cpu, requests.cpu
  - memory, limits.memory, requests.memory
- 存储资源，包括存储资源的总量以及指定storage class的总量
  - requests.storage：存储资源总量，如500Gi
  - persistentvolumeclaims：pvc的个数
  - .storageclass.storage.k8s.io/requests.storage
  - .storageclass.storage.k8s.io/persistentvolumeclaims
  - requests.ephemeral-storage 和 limits.ephemeral-storage （需要v1.8+）
- 对象数，即可创建的对象的个数
  - pods, replicationcontrollers, configmaps, secrets
  - resourcequotas, persistentvolumeclaims
  - services, services.loadbalancers, services.nodeports

示例

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

```yaml
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

```yaml
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
```

Horizontal Pod Autoscaling

- Horizontal Pod Autoscaling (HPA) 可以根据CPU使用率或应用自定义metrics自动扩展 Pod 数量（支持 replication controller、deployment 和 replica set ）。
- 控制管理器每隔30s（可以通过--horizontal-pod-autoscaler-sync-period修改）查询metrics的资源使用情况
- 支持三种metrics类型
  - 预定义metrics（比如Pod的CPU）以利用率的方式计算
  - 自定义的Pod metrics，以原始值（raw value）的方式计算
  - 自定义的object metrics
- 支持两种metrics查询方式：Heapster和自定义的REST API
- 支持多metrics
-

简单用法

- kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10

完整的HPA示例

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

```yaml
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
```

状态条件

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

```
$ kubectl describe hpa cm-test
Name:                           cm-test
Namespace:                      prom
Labels:                         <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 able to successfully calculate a replica count from pods metric http_requests
  ScalingLimited        False   DesiredWithinRange      the desired replica count is within the acceptible range
Events:
```

Network Policy

- 随着微服务的流行，越来越多的云服务平台需要大量模块之间的网络调用。Kubernetes 在 1.3 引入了Network Policy，Network Policy提供了基于策略的网络控制，用于隔离应用并减少攻击面。它使用标签选择器模拟传统的分段网络，并通过策略控制它们之间的流量以及来自外部的流量。
- 在使用Network Policy时，需要注意
- v1.6以及以前的版本需要在kube-apiserver中开启extensions/v1beta1/networkpolicies
- v1.7版本Network Policy已经GA，API版本为networking.k8s.io/v1
- v1.8版本新增 Egress 和 IPBlock 的支持
- 网络插件要支持 Network Policy，如 Calico、Romana、Weave Net和trireme 等

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。

比如默认拒绝所有Pod之间Ingress通信

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

比如默认拒绝所有Pod之间Egress通信

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

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

```yaml
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
```

什么是Ingress

- 通常情况下，service和pod的IP仅可在集群内部访问。集群外部的请求需要通过负载均衡转发到service在Node上暴露的NodePort上，然后再由kube-proxy通过边缘路由器(edge router)将其转发给相关的Pod或者丢弃。
- 而Ingress就是为进入集群的请求提供路由规则的集合。
- 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示例

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

Ingress Controller职责范围

- 负载均衡配置
  - 四层
  - 七层
- DNS配置
- 边缘路由器配置

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。

增加环境变量和存储卷的PodPreset

```yaml
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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

结果

```
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: {}
```

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 mame>.<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

示例

```
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
```

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

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:
  - "*"
```

迁移到CustomResourceDefinition

- 从kubernetes1.7开始ThirdPartyResources被替换为CustomResourceDefinition
- 1.7版本，两种资源同时支持
- 自1.8版本， ThirdPartyResources将被废弃
- 需要将已有资源做数据迁移