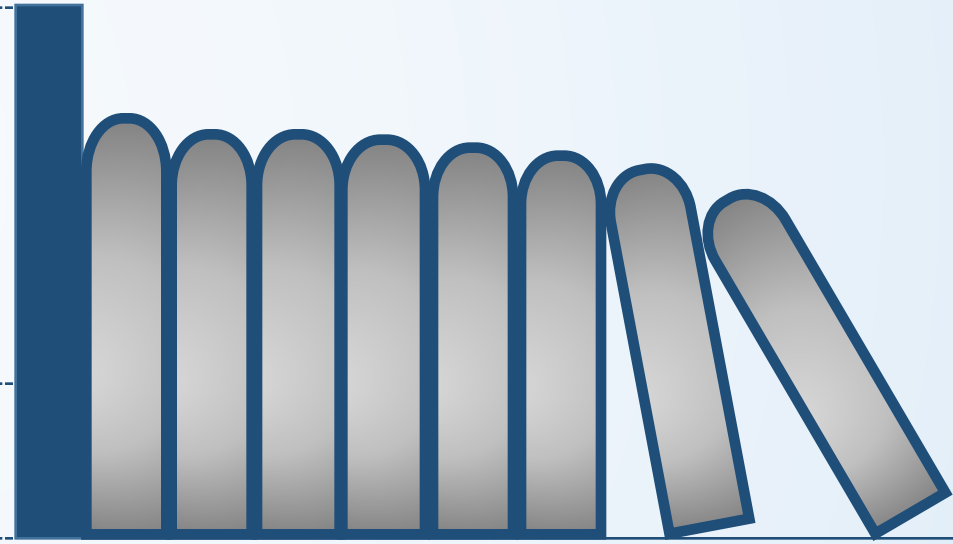
Pod基础与进阶

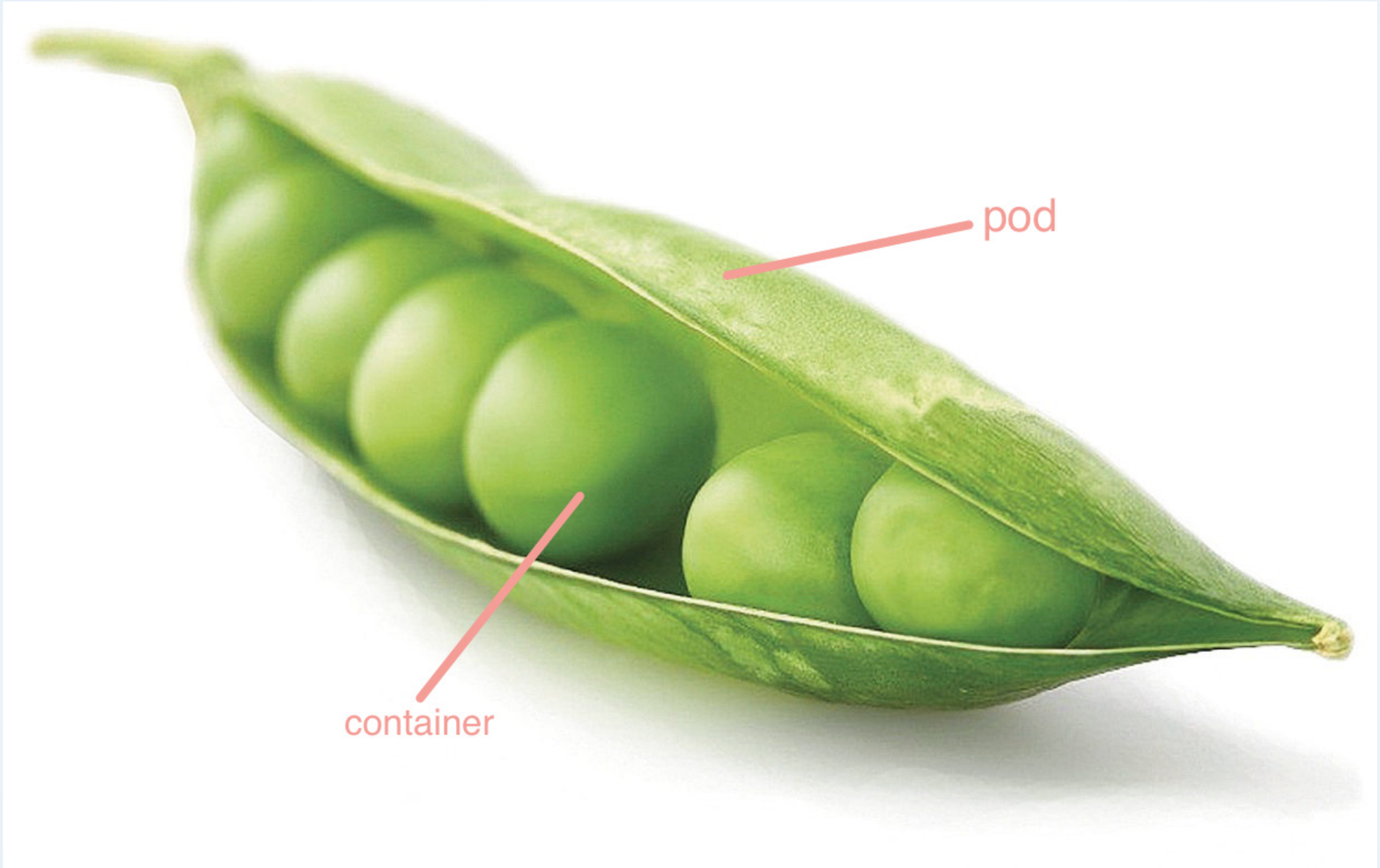
主讲人：宋小金



目录

预期收获

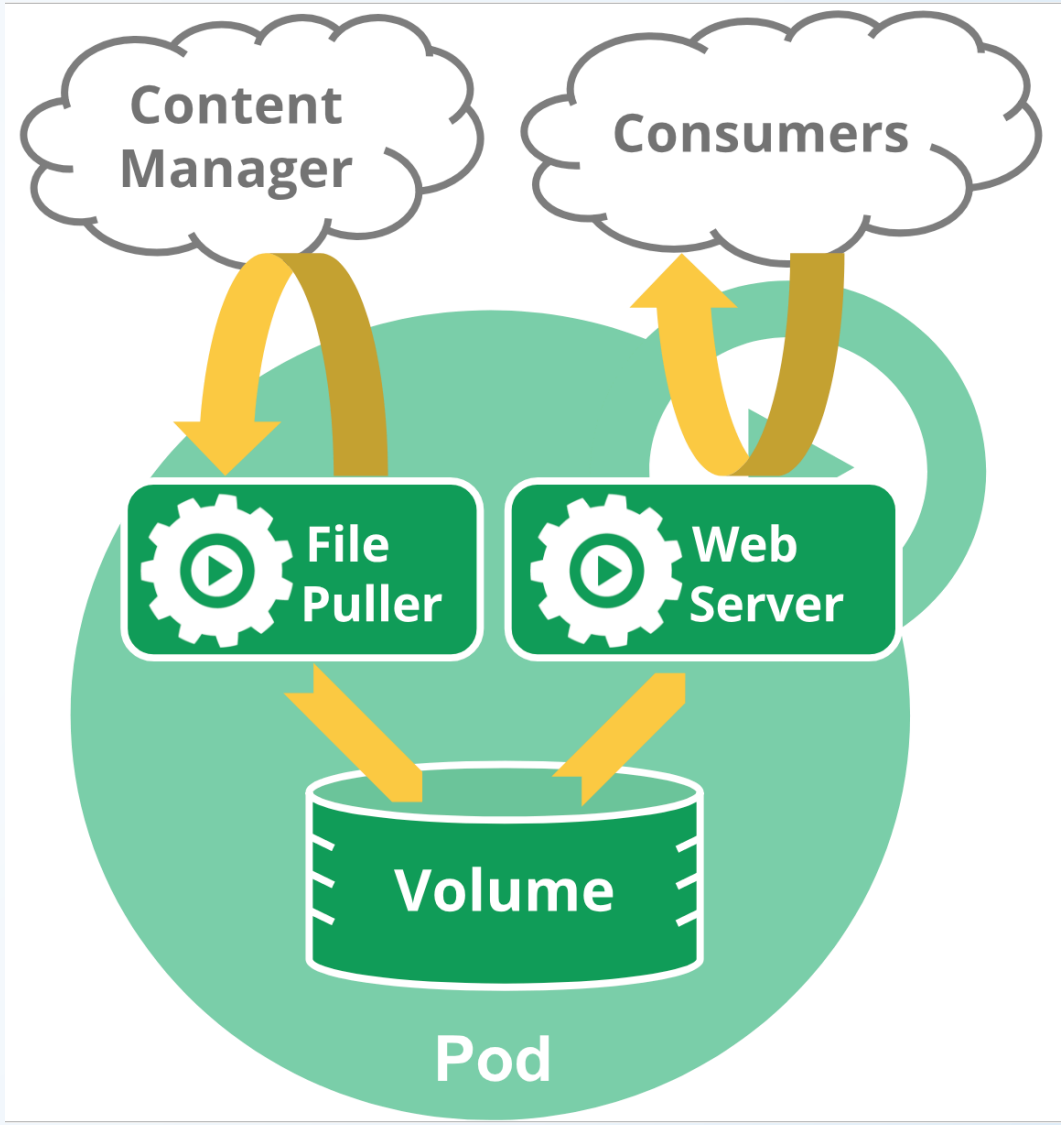
- 了解Pod的常见操作
- 了解静态Pod的用途
- 了解Pod的自我恢复
- 服务初始化与依赖处理


pod
container

Pod定义与操作

Pod介绍

- Pod
  - 一组容器：一组功能相关的容器的组合
  - 共享存储：同一个Pod内的多个容器可共享存储
  - 最小单位：K8S调度和作业运行的基本单位（Scheduler调度，Kubelet运行）
  - 共享Network Namespace：同一个pod里的容器共享同一个网络命名空间（Other container模式），可通过localhost（或127.0.0.1）互相通信。



Pod定义与操作

Pod介绍

Q: 如果pod的*生命周期是短暂*的或不稳定的，那么怎么才能持久化容器数据，即使在pod被销毁或者重启到其他机器存在？
A: Kubernetes支持卷(Volume， Persistent Volumes)的概念， 可以使用持久化的卷类型。

Q: 既然使用Kubernetes集群，那么我们会预期同一个服务可能会有大量实例副本分别分布在不同Node节点上。那么如何*创建大批量的实例副本*。
A ： 可以手动创建单个pod？亦或但是也可以使用Replication Controller/Replica Set/Deployment使用pod模板创建出多份拷贝，下文介绍。

Q: 如果pod是短暂的，那么重启时*IP地址可能会改变*，那么怎么才能从前端容器正确可靠地指向后台容器呢？
A: 这时可以使用Service，后面会详细介绍。

Pod的共享资源

Pods提供两种共享资源：

- 网络
- 存储

Pod的共享资源

Pod网络：

每个pod被分配一个*独立的IP地址*，pod中的每个*容器共享网络命名空间*，包括*IP地址和网络端口*。pod内的容器可以使用*localhost*相互通信。当pod中的容器与pod 外部通信时，他们必须协调如何使用共享网络资源（如端口）。

存储：

pod可以指定一组共享存储volumes。pod中的*所有容器都可以访问共享volumes*，允许这些容器共享数据。volumes 还用于*pod 中的数据持久化*，以防其中一个容器需要重新启动而丢失数据。有关Kubernetes如何在pod中实现共享存储的更多信息，请参考Volumes。

操作Pod

定义pod的资源定义文件
nginx_pod.yaml：

```yaml
apiVersion: v1
kind: pod //表示是一个pod定义
metadata:
  name: nginx //pod名称
  labels:
    name: nginx //pod的标签
spec: //pod容器组定义在spec中
  containers:
  - name: nginx //container名称
    image: nginx //用到的镜像
    ports:
    - containerPort: 80 //容器暴露的端口
```

type定义

object定义

spec定义

单容器Pod

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

多容器Pod

```
apiVersion: v1
kind: Pod
metadata:
 name: test-pod
spec:
  containers:
  - name: tomcat
    image: tomcat
    ports:
    - containerPort: 8080
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

操作Pod

创建pod

```
$ kubectl create -f nginx_pod.yaml
pod "nginx" created
```

列出pod

```
$ kubectl get po nginx
NAME      READY     STATUS              RESTARTS   AGE
nginx     0/1       ContainerCreating   0          36s
```

查看pod日志

```
$ kubectl logs nginx
Error from server (BadRequest): container "nginx" in pod
 "nginx" is waiting to start: ContainerCreating
```

再次查看pod，pod已经处于*Running*状态

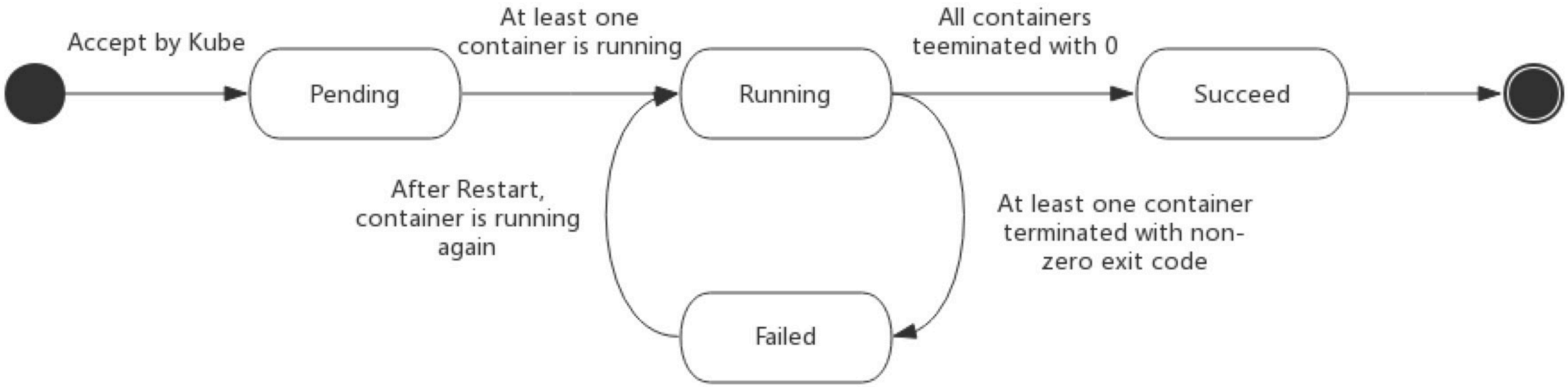
```
$ kubectl get po nginx
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          55s
```

显示pod详细信息

```
$ kubectl describe po nginx
Name:           nginx
Namespace:      default
Node:           bjo-ep-dep-040.dev.fwmrm.net/192.168.17.140
Start Time:     Sat, 21 Apr 2018 06:15:50 +0000
Labels:         name=nginx
Annotations:    <none>
Status:         Running
IP:             10.244.1.211
Containers:
  nginx:
```

Pod的生命周期

Pod的生命周期示意图，从图中可以看到Pod状态的变化



Pod的生命周期

- **Pending**：挂起，Pod已被Kubernetes系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括Pod被调度的时间和通过网络下载镜像的时间
- **Running**：运行中，该Pod已经绑定到了一个节点上，Pod中所有的容器都已被创建。至少有一个容器正在运行，或者正处于启动或重启状态
- **Succeeded**：Pod中的所有容器都被成功终止
- **Failed**：失败，Pod中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非0状态退出或者被系统终止
- **Unknown**：因为某些原因无法取得Pod的状态，通常是因为与Pod所在主机通信失败
- **CrashLoopBackoff**：Pod循环重启崩溃，通常是容器中的应用崩溃造成

静态Pod

静态Pod介绍

在Kubernetes中有一个***DaemonSet类型的pod***，这种类型的pod可以在*某个节点上长期运行*，这种类型的pod就是静态pod。静态pod直接由某个节点上的***kubelet程序进行管理，不需要api server介入***，静态pod也不需要关联任何RC，完全是由***kubelet程序来监控***，当kubelet发现静态pod停止掉的时候，重新启动静态pod。

静态Pod

静态Pod创建

静态Pod有两种创建方式：

- 配置文件
- 通过HTTP。

通过HTTP创建静态pods

Kubelet周期地从 *–manifest-url* 参数指定的地址下载文件，并且把它翻译成JSON/YAML格式的pod定义。此后的操作方式与 *–pod-manifest-path*相同，kubelet会不时地重新下载该文件，当文件变化时对应地终止或启动静态pod。

静态Pod

静态Pod创建

以配置文件形式为例，以静态pod的方式启动一个Nginx的Web服务器：
首先查看kubelet对pod manifest文件的设置路径，如未设置可自行选择一个目录，然后加入*pod-manifest-path*，重启kubelet服务。

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

```yaml
apiVersion: v1
kind: pod
metadata:
  name: static-nginx-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
```

静态Pod

查看静态Pod：

通过下面命令*kubectl get po*查看，发现静态pod已经启动：

```
kubectl get po -o wide | grep static-nginx-web
static-nginx-web-bjo-ep-dep-039.dev.fwmrm.net   1/1   Running   0   1m   10.244.0.28   bjo-ep-dep-039.dev.fwmrm.net
```

删除静态Pod：

移除 /etc/kubernetes/manifests/static-web.yaml后，再次查看pod，发现原静态pod已经查询不到了。

ImagePullPolicy

- 支持三种ImagePullPolicy，在配置文件中通过imagePullPolicy字体设置
  - 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。

RestartPolicy

在Pod中的容器可能会由于异常等原因导致其终止退出，Kubernetes提供了重启策略以重启容器。重启策略对同一个Pod的所有容器起作用，容器的重启由Node上的kubelet执行。Pod支持三种重启策略，在配置文件中通过restartPolicy字段设置重启策略

- 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。
- （2）自定义变量
  - 模板里配置注入

环境变量使用例子

```
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
    -name: testKey
        value: testValue
  restartPolicy: Never
```

DNS解析配置

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

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
```

使用主机名空间

- 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
```

资源设置

- 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。

资源设置

```
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"
```

容器探针(Probe)

对线上业务来说，保证服务的正常稳定是重中之重，对故障服务的及时处理避免影响业务以及快速恢复一直是开发运维的难点。Kubernetes提供了健康检查服务，对于*检测到故障服务会被及时自动下线*，以及通过*重启服务的方式使服务自动恢复*。

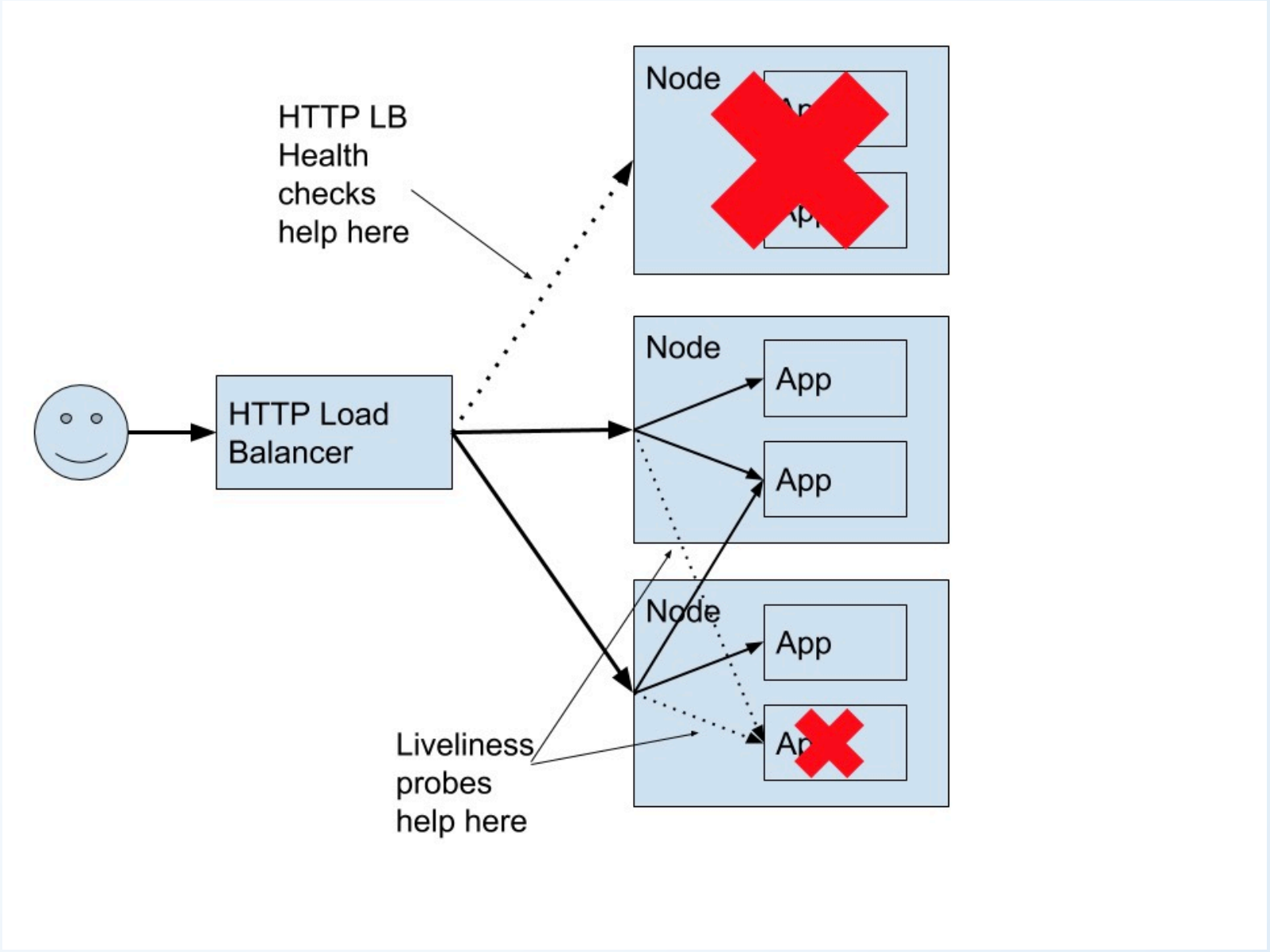
容器探针(Probe)

使用Liveness及Readness探针

- Liveness探针：主要用于判断*Container是否处于运行状态*，比如当服务crash或者死锁等情况发生时，*kubelet会kill掉Container*，然后根据其设置的restart policy进行相应操作（可能会在本机重新启动Container，或者因为设置Kubernetes QoS，本机没有资源情况下会被分发的其他机器上重新启动）。
- Readness探针：主要用于*判断服务是否已经正常工作*，如果服务没有加载完成或工作异常，服务所在的Pod的IP*地址会从服务的Endpoints中被移除*，也就是说，当服务没有ready时，会将其从服务的load balancer中移除，不会再接受或响应任何请求。

容器探针(Probe)

Liveness探针图示：



容器探针(Probe)

健康检查总结

探针检查结果分为3种情况：

- 成功（Success）：通过检查。
- 失败（Failure）：检查失败。
- 未知（Unknown）：检查未知，需要人工干预。

探针类型	说明	通过健康检查标准
ExecAction	Container内部执行shell命令	shell命令返回0
TCPSocketAction	通过Container的IP、port执行tcp进行检查	port是否打开
HTTPGetAction	通过Container的IP、port、path，用HTTP Get请求进行检查	200<=返回值<400

容器探针(Probe)

服务可用性与自动恢复

- 如果服务的健康检查（readiness）失败，*故障的服务实例从service endpoint中下线*，外部请求将不会再转发到该服务上，一定程度上保证正在提供的服务的正确性，如果服务自我恢复了（比如网络问题），会自动重新加入service endpoint对外提供服务。
- 另外，如果设置了Container（liveness）的探针，对故障服务的Container（liveness）的探针同样会失败，*container会被kill掉*，并根据原设置的container重启策略，系统倾向于在其原所在的机器上重启该container、或其他机器重新创建一个pod。

由于上面的机制，整个服务实现了*自身高可用与自动恢复*。

容器探针(Probe)

Liveness 探针

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - args:
    - /server
    image: k8s.gcr.io/liveness
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
    name: liveness
```

容器探针最佳实践

- 可以使用命名的ContainerPort作为HTTP或TCP liveness检查， Probe port指定容器的Port名即可
- 建议对全部服务同时设置readiness和liveness健康检查
- 通过TCP对端口检查要小心，未必准确，进程存在端口能通，服务未必正常
- Probe的配置是非常灵活的，要结合容器里部署的业务属性综合考虑

可用的控制器

有三种如下可用的控制器：

- 使用 Job *运行预期会终止的 Pod*，例如批量计算。Job 仅适用于重启策略为 OnFailure 或 Never 的 Pod
- 对预期不会终止的 Pod *使用 ReplicationController、ReplicaSet 和 Deployment*，例如 Web 服务器。 ReplicationController 仅适用于具有 restartPolicy 为 Always 的 Pod。
- 提供特定于机器的系统服务，*使用 DaemonSet* 为每台机器运行一个 Pod

所有这三种类型的控制器都包含一*个 PodTemplate*。建议创建适当的控制器，让它们来创建 Pod，而不是直接自己创建 Pod。这是因为*单独的 Pod 在机器故障的情况下没有办法自动复原*，而控制器却可以。

Pod初始化容器

理解Init Container

- 一个pod中可以有*一或多个*Init Container。 Pod的中多个*Init Container启动顺序为*yaml文件中的描述顺序，且*串行方式*启动，下一个Init/app Container必须等待上一个Init Container完成后方可启动。如Init Container启动失败，后续的init Container和应用Container将不会被执行启动命令

- 由于Init Container必须要在pod状态变为Ready之前完成，*所以其不需要readiness探针*。另外在资源的requests与limits上与普通Container有细微差别。除以上2点外， Init Container与普通Container并无明显区别。

Pod初始化

Init Container处理服务依赖

可利用Init Container来判断app Container中被依赖的服务是否成功启动。如被依赖的app Container服务启动失败，那么利用Init Container启动失败可以阻止后续app Container服务的启动，如下例：

```
spec:
  initContainers:
  - name: init-serviceA
    image: registry.docker.dev.fwmrm.net/busybox:latest
    command: ['sh', '-c', "curl --connect-timeout 3 --max-time 5 --retry 10 --retry-delay 5 --retry-max-time 60 serviceB:port
B/pathB/"]
  containers:
```

Pod初始化

Init Container处理服务依赖

可利用Init Container来判断app Container中被依赖的服务是否成功启动。

```
spec:
  initContainers:
  - name: init-serviceA
    image: registry.docker.dev.fwmrm.net/busybox:latest
    command: ['sh', '-c', "curl --connect-timeout 3 --max-time 5 --retry 10 --retry-delay 5 --retry-max-time 60 serviceB:port
B/pathB/"]
  containers:
```

如果启动serviceA Pod时，serviceB还没有ready，pod会处于Init状态。

```
NAME                  READY   STATUS    RESTARTS   AGE     IP            NODE
serviceA-3071943788-8x27q     0/1     Init:0/1   0          20s     10.244.1.172   bjo-ep-svc-017.dev.fwmrm.net
```

Pod初始化

Init Container处理服务依赖

通过kubectl describe po查看,可以看出app Container的启动时在init Container启动并成功完成后

```
Events:
FirstSeen LastSeen    Count   From                SubObjectPath          Type      Reason  Message
--------- --------    -----   ----                -------------          --------  ------  -------
25s       25s     1   default-scheduler                          Normal    ScheduledSuccessfully assigned serviceA-3071943788-g03
wt to bjo-ep-dep-040.dev.fwmrm.net
25s       25s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net              Normal    SuccessfulMountVolume   MountVolume.
SetUp succeeded for volume "serviceA-config-volume"
25s       25s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net              Normal    SuccessfulMountVolume   MountVolume.
SetUp succeeded for volume "default-token-2c9j1"
24s       24s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.initContainers{init-myservice} Normal    Pulling pulling im
age "registry.docker.dev.fwmrm.net/ui-search-solr-data:latest"
24s       24s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.initContainers{init-myservice} Normal    Pulled  Successfu
lly pulled image "registry.docker.dev.fwmrm.net/busybox:latest"
24s       24s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.initContainers{init-myservice} Normal    Created Created
 container
24s       24s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.initContainers{init-myservice} Normal    Started Started c
ontainer
20s       20s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.containers{is}       Normal    Pulling pulling image "regist
ry.docker.dev.fwmrm.net/infra/is:latest"
20s       20s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.containers{is}       Normal    Pulled  Successfully pulled i
mage "registry.docker.dev.fwmrm.net/infra/is:latest"
20s       20s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.containers{is}       Normal    Created Created container
19s       19s     1   kubelet, bjo-ep-dep-040.dev.fwmrm.net   spec.containers{is}       Normal    Started Started container
```

Pod初始化

Init Container处理服务依赖

查看docker Container log， init Container正在按照预先的设定，每3秒轮询验证serviceB健康检查点serviceB:portB/pathB/

```
$ docker logs 4fd58bf54f76

waiting for serviceB service
waiting for serviceB service
```

等待一段时间后，再次通过kubectl get po -o wide查看pod处于Running状态

NAME	READY	STATUS	RESTARTS	AGE	IP	.I NODE
serviceA-3071943788-g03wt	1/1	Running	0	1m	10.244.2.68	bjo-ep-dep-040.dev.fwmrm.net

容器生命周期内的Hook

Hook分类

Kubernetes为容器在其生命周期内提供了两种钩子（hook），分别是postStart与preStop两种事件：

- PostStart：在*容器启动之后*，PostStart hook会立即被执行，但需要注意的是，容器里的ENTRYPOINT与PostStart hook的执行顺序谁先谁后并不确定。
- PreStop：在*容器被终止之前被执行*，采用一种阻塞式的方式，也就是必须在PreStop hook执行完毕之后，销毁容器的动作才会被执行。

容器生命周期内的Hook

Hook Handler分类

容器通过实现和注册一个handler来实现对hook的访问，有2种类型的hook handlers：

- **Exec**：执行一个Shell命令。
- **HTTP**: 对一个endpoint执行http请求。

容器生命周期内的Hook

postStrart,
preStop举例

```
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

验证Hook是
否执行

```
$ kubectl exec -it lifecycle-demo cat /usr/share/message
Hello from the postStart handler
```

Capabilities

- 在docker run命令中，我们可以通过*--cap-add*和*--cap-drop*来给容器添加Linux Capabilities。
- Kubernetes通过在*Pod.spec.containers.sercurityContext.capabilities*中配置容器待add和drop的Capabilities，最终借助docker container Capabilities的能力，完成容器的Capabilities权限控制。

Capabilities

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
  - name: friendly-container
    image: "alpine:3.4”
    command: ["/bin/echo", "hello", "world"]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        drop:
        - KILL
```

Pod的完整模板详解

```
apiVersion: v1              //版本
kind: pod                   //类型，pod
metadata:                   //元数据
  name: String              //元数据，pod的名字
  namespace: String         //元数据，pod的命名空间
  labels:                   //元数据，标签列表
    - name: String          //元数据，标签的名字
  annotations:              //元数据,自定义注解列表
    - name: String          //元数据,自定义注解名字
spec:                       //pod中容器的详细定义
  containers:               //pod中的容器列表，可以有多个容器
  - name: String
    image: String           //容器中的镜像
    imagesPullPolicy: [Always|Never|IfNotPresent]//获取镜像的策略
    command: [String]       //容器的启动命令列表（不配置的话使用镜像内部的命令）
    args: [String]          //启动参数列表
    workingDir: String      //容器的工作目录
    volumeMounts:           //挂载到到容器内部的存储卷设置
    - name: String
      mountPath: String
      readOnly: boolean
    ports:                  //容器需要暴露的端口号列表
    - name: String
      containerPort: int    //容器要暴露的端口
      hostPort: int         //容器所在主机监听的端口（容器暴露端口映射到宿主机的端口）
      protocol: String
    env:                    //容器运行前要设置的环境列表
    - name: String
      value: String
    resources:              //资源限制
      limits:
        cpu: Srting
        memory: String
      requeste:
        cpu: String
        memory: String
```

Pod的完整模板详解

```
37      livenessProbe:          //pod内容器健康检查的设置
38        exec:
39          command: [String]
40        httpGet:              //通过httpget检查健康
41          path: String
42          port: number
43          host: String
44          scheme: Srtring
45          httpHeaders:
46          - name: Stirng
47            value: String
48        tcpSocket:            //通过tcpSocket检查健康
49          port: number
50        initialDelaySeconds: 0//首次检查时间
51        timeoutSeconds: 0     //检查超时时间
52        periodSeconds: 0      //检查间隔时间
53        successThreshold: 0
54        failureThreshold: 0
55        securityContext:      //安全配置
56          privileged: falae
57      restartPolicy: [Always|Never|OnFailure]//重启策略
58      nodeSelector: object    //节点选择
59      imagePullSecrets:
60      - name: String
61      hostNetwork: false      //是否使用主机网络模式，默认否
62    volumes:                  //在该pod上定义共享存储卷
63    - name: String
64      meptyDir: {}
65      hostPath:
66        path: string
67      secret:                 //类型为secret的存储卷
68        secretName: String
69        item:
70        - key: String
71          path: String
72      configMap:              //类型为configMap的存储卷
73        name: String
74        items:
75        - key: String
76          path: String
```

Pod实操演示

参照附件：10　Kubernetes常用对象.txt

课程回顾

已学知识要点

- *Pod的定义和操作*
- *Pod的生命周期包括哪几个阶段*
- *Pod初始化与服务依赖处理*
- *容器生命周期内Hook功能*