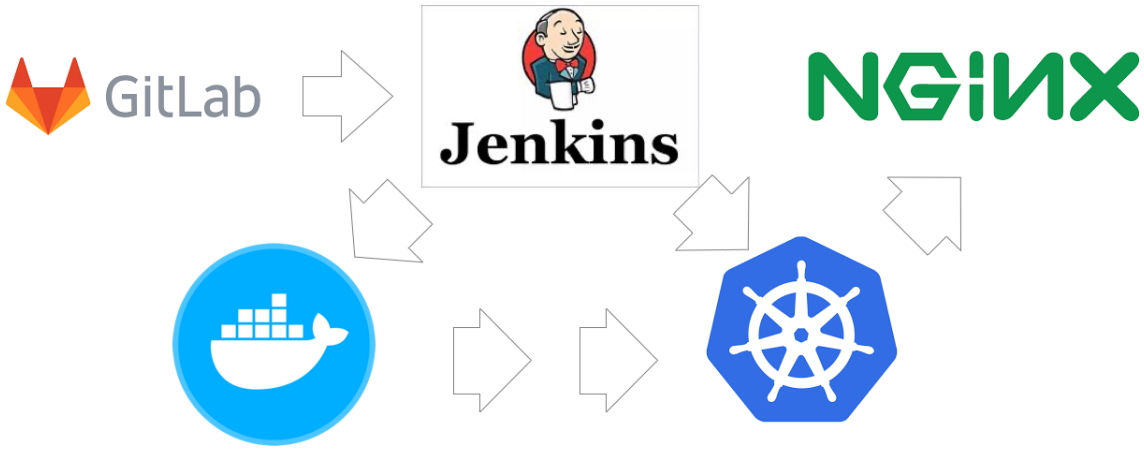
容器虚拟化技术和自动化部署



资源清单-pod进阶

1. 资源清单格式

1.2简介

资源清单有5个顶级的字段组成：apiVersion、kind、metadata、spec、status。

```
apiVersion: group/apiversion # 如果没有给定 group 名称，那么默认为 core，可以使用 kubectl apiversions # 获取当前 k8s 版本上所有的 apiVersion 版本信息( 每个版本可能不同 )
kind: #资源类别
metadata:  #资源元数据
    name
    namespace
    lables
    annotations # 主要目的是方便用户阅读查找
spec: # 期望的状态（disired state）
status: # 当前状态，本字段有 Kubernetes 自身维护，用户不能去定义
```

1.2资源的 apiVersion 版本信息

使用kubectl命令可以查看apiVersion的各个版本信息

```
kubectl api-versions
```

1.3获取字段设置帮助文档

```
kubectl explain pod
```

1.4字段配置格式类型

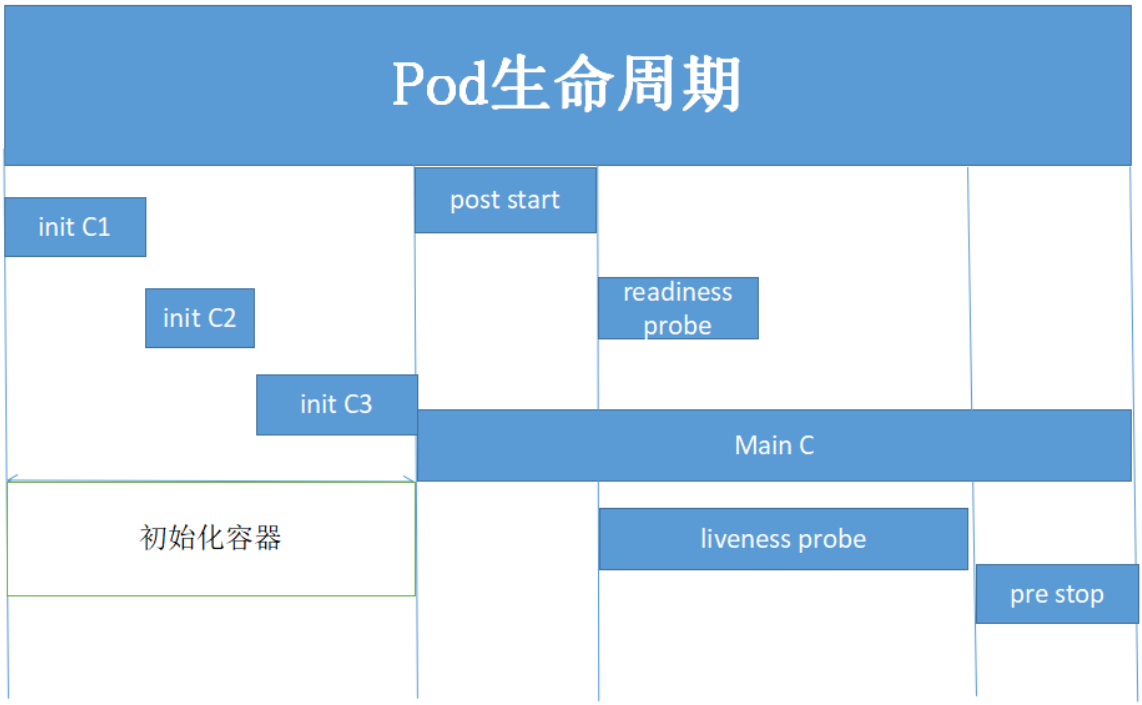
资源清单中大致可以分为如下几种类型：

<map[String]string> <[]string> <[]Object>

```
apiVersion <string> #表示字符串类型
metadata <Object> #表示需要嵌套多层字段
labels <map[string]string> #表示由k:v组成的映射
finalizers <[]string> #表示字串列表
ownerReferences <[]Object> #表示对象列表
hostPID <boolean> #布尔类型
priority <integer> #整型
name <string> -required- #如果类型后面接 -required-，表示为必填字段
```

2.pod声明周期

2.0案例



2.0.0实验需要准备镜像

```
docker pull busybox:1.32.0
docker pull nginx:1.17.10-alpine
```

2.0.1initC案例

initC特点：

1. initC总是运行到成功完成为止。
2. 每个initC容器都必须在下一个initC启动之前成功完成。
3. 如果initC容器运行失败，K8S集群会不断的重启该pod，直到initC容器成功为止。
4. 如果pod对应的restartPolicy为never，它就不会重新启动。

pod/initcpod.yml文件,***需要准备busybox:1.32.0镜像***

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: busybox:1.32.0
      imagePullPolicy: IfNotPresent
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
    - name: init-myservice
      image: busybox:1.32.0
      imagePullPolicy: IfNotPresent
      command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
    - name: init-mydb
      image: busybox:1.32.0
      command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

pod/initcservice1.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

pod/initcservice2.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

执行命令

```
先查看pod启动情况
kubectl get pods

详细查看pod启动情况
kubectl describe pod myapp-pod

查看myapp-pod中的第一个initContainer日志
kubectl logs myapp-pod -c init-myservice

运行init-myservice服务
kubectl apply -f initcservice1.yml

查看init-myservice服务运行情况
kubectl get svc

查看myapp-pod运行情况，需要耐心等一会，会发现pod的第一个init已经就绪
kubectl get pods


运行init-mydb服务
kubectl apply -f initcservice2.yml

查看init-myservice服务运行情况
kubectl get svc

查看myapp-pod运行情况，需要耐心等一会，会发现pod的两个init已经就绪，pod状态为ready
kubectl get pod -w
```

2.0.2readinessProbe(就绪检测)

容器就绪检测案例，***需要准备nginx:1.17.10-alpine镜像。***

pod/readinessprobepod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: readinessprobe-pod
  labels:
    app: readinessprobe-pod
spec:
  containers:
    - name: readinessprobe-pod
      image: nginx:1.17.10-alpine
      imagePullPolicy: IfNotPresent
      readinessProbe:
        httpGet:
          port: 80
          path: /index1.html
        initialDelaySeconds: 1
        periodSeconds: 3
  restartPolicy: Always
```

执行命令

```
创建pod
kubectl apply -f readinessprobepod.yml

检查pod状态，虽然pod状态显示running但是ready显示0/1，因为就绪检查未通过
kubectl get pods

查看pod详细信息，文件最后一行显示readiness probe failed。。。。
kubectl describe pod readinessprobe-pod

进入pod内部，因为是alpine系统，需要使用sh命令
kubectl exec -it readinessprobe-pod sh

进入容器内目录
cd /usr/share/nginx/html/

追加一个index1.html文件
echo "welcome lagou" >> index1.html

退出容器，再次查看pod状态，pod已经正常启动
exit
kubectl get pods
```

2.0.3livenessProbe(存活检测)

容器存活检测，***需要准备busybox:1.32.0镜像***

pod/livenessprobepod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: livenessprobe-pod
  labels:
    app: livenessprobe-pod
spec:
  containers:
    - name: livenessprobe-pod
      image: busybox:1.32.0
      imagePullPolicy: IfNotPresent
      command: ["/bin/sh","-c","touch /tmp/livenesspod ; sleep 30; rm -rf
/tmp/livenesspod; sleep
      3600"]
      livenessProbe:
        exec:
          command: ["test","-e","/tmp/livenesspod"]
        initialDelaySeconds: 1
        periodSeconds: 3
  restartPolicy: Always

```

执行命令

```
创建pod
kubectl apply -f livenessprobepod.yml

监控pod状态变化,容器正常启动
kubectl get pod -w

等待30秒后，发现pod的RESTARTS值从0变为1.说明pod已经重启一次。
```

2.0.4livenessprobe案例二

容器存活检测案例，***需要准备nginx:1.17.10-alpine镜像。***

pod/livenessprobenginxpod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: livenessprobenginx-pod
  labels:
    app: livenessprobenginx-pod
spec:
  containers:
    - name: livenessprobenginx-pod
      image: nginx:1.17.10-alpine
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 80
          name: nginxhttpget
      livenessProbe:
        httpGet:
          port: 80
          path: /index.html
        initialDelaySeconds: 1
        periodSeconds: 3
        timeoutSeconds: 10
  restartPolicy: Always
```

执行命令

```
创建pod
kubectl apply -f livenessprobenginxpod.yml

查看pod状态
kubectl get pod

查看容器IP，访问index.html页面。index.html页面可以正常访问。
kubectl get pod -o wide
curl 10.81.58.199

进入容器内部
kubectl exec -it livenessprobenginx-pod sh
```

```
删除index.html文件,退出容器
rm -rf //usr/share/nginx/html/index.html
exit



再次监控pod状态，等待一段时间后，发现pod的RESTARTS值从0变为1.说明pod已经重启一次。
kubectl get pod -w

进入容器删除文件一条命令执行rm -rf命令后退出容器。
kubectl exec -it livenessprobenginx-pod -- rm -rf
/usr/share/nginx/html/index.html

再次监控pod状态，等待一段时间后，发现pod的RESTARTS值从1变为2.说明pod已经重启一次。
kubectl get pod -w

因为liveness监控index.html页面已经被删除，所以pod需要重新启动，重启后又重新创建nginx
镜像。nginx镜像中默认有index.html页面。
```

2.0.5livenessprobe案例三

容器存活检测案例，***需要准备nginx:1.17.10-alpine镜像。***

pod/livenessprobenginxpod2.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: livenessprobenginx-pod2
  labels:
    app: livenessprobenginx-pod2
spec:
  containers:
    - name: livenessprobenginx-pod2
      image: nginx:1.17.10-alpine
      imagePullPolicy: IfNotPresent
      livenessProbe:
        tcpSocket:
          #监测8080端口，如果8080端口没有反馈信息，重启pod
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 3
        timeoutSeconds: 5
  restartPolicy: Always
```

执行命令

```
创建pod
kubectl apply -f livenessprobenginxpod2.yml

查看pod状态
kubectl get pod -w

存活检测监听8080端口，8080端口没有反馈信息后重启pod，RESTARTS值从0变为1
```

2.0.6钩子函数案例

postStart函数,***需要准备busybox:1.32.0镜像***

pod/lifeclepod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: lifecle-pod1
  labels:
    app: lifecle-pod1
spec:
  containers:
    - name: lifecle-pod1
      image: busybox:1.32.0
      imagePullPolicy: IfNotPresent
      lifecycle:
        postStart:
          exec:
            #创建/lagou/k8s/目录，在目录下创建index.html
            command: ['mkdir','-p','/lagou/k8s/index.html']
      command: ['sh','-c','sleep 5000']
  restartPolicy: Always
```

执行命令

```
创建pod
kubectl apply -f lifeclepod1.yml

查看pod状态
kubectl get pod

进入容器内部，查看是否创建了/lagou/k8s/index.html文件
kubectl exec -it lifecle-pod1 sh

cd /data/web
ls
```

3.总结pod声明周期

pod对象自从创建开始至终止退出的时间范围称为生命周期，在这段时间中，pod会处于多种不同的状态，并执行一些操作；其中，创建主容器为必须的操作，其他可选的操作还包括运行初始化容器（init container）、容器启动后钩子（start hook）、容器的存活性探测（liveness probe）、就绪性探测（readiness probe）以及容器终止前钩子（pre stop hook）等，这些操作是否执行则取决于pod的定义 。

3.1pod的相位

使用`kubectl get pods`命令,STATUS被称之为相位(phase)。

无论是手动创建还是通过控制器创建pod，pod对象总是应该处于其生命进程中以下几个相位之一：

- pending：apiserver创建了pod资源对象并存入etcd中，但它尚未被调度完成或者仍处于下载镜像的过程中
- running：pod已经被调度至某节点，并且所有容器都已经被kubelet创建完成
- succeeded：pod中的所有容器都已经成功终止并且不会被重启
- failed：所有容器都已经终止，但至少有一个容器终止失败，即容器返回了非0值的退出状态或已经被系统终止。
- unknown：apiserver无法正常获取到pod对象的状态信息，通常是由于其无法于所在工作节点的kubelet通信所致。

pod的相位是在其生命周期中的宏观概念，而非对容器或pod对象的综合汇总，而且相位的数量和含义被严格界定。

3.2pod的创建过程

pod是k8s的基础单元，以下为一个pod资源对象的典型创建过程：

1. 用户通过kubectl或其他api客户端提交pod spec给api server
2. api server尝试着将pod对象的相关信息存入etcd中，待写入操作执行完成，api server即会返回确认信息至客户端。
3. api server开始反映etcd中的状态变化
4. 所有的k8s组件均使用watch机制来跟踪检查api server上的相关变动
5. kube-scheduler通过其watch觉察到api server创建了新的pod对象但尚未绑定至任何工作节点
6. kube-scheduler为pod对象挑选一个工作节点并将结果信息更新至api server
7. 调度结果信息由api server更新至etcd，而且api server也开始反映此pod对象的调度结果
8. pod被调度到目标工作节点上的kubelet尝试在当前节点上调用docker启动容器，并将容器的结果状态回送至api server
9. api server将pod状态信息存入etcd中
10. 在etcd确认写入操作成功完成后，api server将确认信息发送至相关的kubelet。

3.3pod生命周期中的重要行为

除了创建应用容器之外，用户还可以为pod对象定义其生命周期中的多种行为，如初始化容器、存活性探测及就绪性探测等。

1、初始化容器

初始化容器即应用程序的主容器启动之前要运行的容器，常用于为主容器执行一些预置操作，它们具有两种典型特征

1. 初始化容器必须运行完成直至结束，若某初始化容器运行失败，那么k8s需要重启它直到成功完成
2. 每个初始化容器都必须按定义的顺序串行运行

有不少场景都需要在应用容器启动之前进行部分初始化操作，例如，等待其他相关联组件服务可用、基于环境变量或配置模板为应用程序生成配置文件、从配置中心获取配置等。初始化容器的典型应用需求具体包含如下几种。

1. 用于运行特定的工具程序，出于安全等反面的原因，这些程序不适于包含在主容器镜像中
2. 提供主容器镜像中不具备的工具程序或自定义代码
3. 为容器镜像的构建和部署人员提供了分离、独立工作的途径，使得它们不必协同起来制作单个镜像文件
4. 初始化容器和主容器处于不同的文件系统视图中，因此可以分别安全地使用敏感数据，例如secrets资源
5. 初始化容器要先于应用容器串行启动并运行完成，因此可用于延后应用容器的启动直至其依赖的条件得到满足

pod资源的spec.initContainers字段以列表的形式定义可用的初始容器，其嵌套可用字段类似于spec.containers。

3.4生命周期钩子函数

容器生命周期钩子使它能够感知其自身生命周期管理中的事件，并在相应的时刻到来时运行由用户指定的处理程序代码。k8s为容器提供了两种生命周期钩子：

- postStart：于容器创建完成之后立即运行的钩子处理器（handler），不过k8s无法确保它一定会于容器中的entrypoint之前运行
- preStop：于容器终止操作之前立即运行的钩子处理器，它以同步的方式调用，因此在其完成之前会阻塞删除容器的操作调用。

钩子处理器的实现方法由Exec和HTTP两种，前一种在钩子事件触发时直接在当前容器中运行由用户定义的命令，后一种则是在当前容器中向某url发起http请求。postStart和preStop处理器定义在spec.lifecycle嵌套字段中。

3.5容器探测

容器探测时pod对象生命周期中的一项重要的日常任务，它是kubelet对容器周期性执行的健康状态诊断，诊断操作由容器的处理器进行定义。k8s支持三种容器探针用于pod探测：

- ExecAction：在容器中执行一个命令，并根据其返回的状态码进行诊断的操作称为Exec探测，状态码为0表示成功，否则即为不健康状态
- TCPSocketAction：通过与容器的某TCP端口尝试建立连接进行诊断，端口能够成功打开即为正常，否则为不健康状态。
- HTTPGetAction：通过向容器IP地址的某指定端口的指定path发起HTTP GET请求进行诊断，响应码大于等于200且小于400时即为成功。

任何一种探测方式都可能存在三种结果：

- success(成功)：容器通过了诊断
- failure(失败)：容器未通过了诊断
- unknown(未知)：诊断失败，因此不会采取任何行动

kubelet可在活动容器上执行两种类型的检测：

(***livenessProbe***)存活性检测：用于判定容器是否处于运行状态，一旦此类检测未通过，kubelet将杀死容器并根据restartPolicy决定是否将其重启；未定义存活性检测的容器的默认状态未success

(***readinessProbe***)就绪性检测：用于判断容器是否准备就绪并可对外提供服务；未通过检测的容器意味着尚未准备就绪，端点控制器会将其IP从所有匹配到此pod对象的service对象的端点列表中移除；检测通过之后，会再次将其IP添加至端点列表中。

3.6容器的重启策略

容器程序发生奔溃或容器申请超出限制的资源等原因都可能会导致pod对象的终止，此时是否应该重建该pod对象则取决于其重启策略（restartPolicy）属性的定义：

- Always：但凡pod对象终止就将其重启，此为默认设定
- OnFailure：尽在pod对象出现错误时方才将其重启
- Never：从不重启。

restartPolicy适用于pod对象中的所有容器，而且它仅用于控制在同一节点上重新启动pod对象的相关容器。首次需要重启的容器，将在其需要时立即进行重启，随后再次需要重启的操作将由kubelet延迟一段时间后进行，且反复的重启操作的延迟时长以此为10s、20s、40s、80s、160s和300s，300s是最大延迟时长。事实上，一旦绑定到一个节点，pod对象将永远不会重新绑定到另一个节点，它要么被重启，要么终止，直到节点发生故障或被删除。

3.7pod的终止过程

当用户提交删除请求之后，系统就会进行强制删除操作的宽限期倒计时，并将TERM信息发送给pod对象的每个容器中的主进程。宽限期倒计时结束后，这些进程将收到强制终止的KILL信号，pod对象随即也将由api server删除，如果在等待进程终止的过程中，kubelet或容器管理器发生了重启，那么终止操作会重新获得一个满额的删除宽限期并重新执行删除操作。

一个典型的pod对象终止流程具体如下：

1. 用户发送删除pod对象的命令
2. api服务器中的pod对象会随着时间的推移而更新，在宽限期内（默认30s），pod被视为dead
3. 将pod标记为terminating状态
4. 与第三步同时运行，kubelet在监控到pod对象转为terminating状态的同时启动pod关闭过程
5. 与第三步同时运行，端点控制器监控到pod对象的关闭行为时将其从所有匹配到此端点的service资源的端点列表中移除
6. 如果当前pod对象定义了preStop钩子处理器，则在其标记为terminating后即会以同步的方式启动执行；若宽限期结束后，preStop仍未执行结束，则第二步会被重新执行并额外获取一个时长为2s的小宽限期
7. pod对象中的容器进程收到TERM信号
8. 宽限期结束后，若存在任何一个仍在运行的进程，那么pod对象即会收到SIGKILL信号
9. kubelet请求api server将此pod资源的宽限期设置为0从而完成删除操作，它变得对用户不再可见。

默认情况下，所有删除操作的宽限期都是30s，不过，kubectl delete命令可以使用“--grace-period=”选项自定义其时长，若使用0值则表示直接强制删除指定的资源，不过此时需要同时使用命令“--forece”选项。