简介

Pod是可以在Kubernetes中创建和管理的最小可部署单元。Pod是一组（一个或多个）容器的打包，这一组容器共享存储、网络；pod中的容器地位均等且一同调度，在共享的上下文中运行。这些容器在业务上是紧密耦合在一起的。Pod就像一台"逻辑主机"为这一组紧密相关的容器提供运行上下文。Pod除了正常运行的业务容器外还可以在启动期间运行Init容器。也可以在集群支持临时容器的情况下，以调试为目的注入临时容器。

Pod的阶段（状态）

通过kubectl get pod -o yaml 查看pod的信息，其中status.phase字段表示该pod的阶段。 通过kubectl describe pod 查看pod详情，其中State字段表示该pod的状态（阶段）。

阶段（状态）	描述
Pending	Pod 已被 Kubernetes 系统接受，但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间。
Running	Pod 已经绑定到了某个节点，Pod 中所有的容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态。
Succeeded	Pod 中的所有容器都已成功终止，并且不会再重启。
Failed	Pod 中的所有容器都已终止，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。这种情况通常是因为与 Pod 所在主机通信失败

容器状态

状态值	描述
Waiting	如果容器并不处在 Running 或 Terminated 状态之一，它就处在 Waiting 状态。处于 Waiting 状态的容器仍在运行它完成启动所需要的操作。使用 kubectl 来查询包含 Waiting 状态的容器的 Pod 时，会看到一个 Reason 字段，其中给出了容器处于等待状态的原因
Running	Running 状态表明容器正在执行状态并且没有问题发生。 如果配置了 postStart 回调，那么该回调已经执行且已完成。 如果使用 kubectl 来查询包含 Running 状态的容器的 Pod 时，也会看到 关于容器进入 Running 状态的信息。
Terminated	处于 Terminated 状态的容器已经开始执行并且正常结束或者因为某些原因失败。如果使用 kubectl 来查询包含 Terminated 状态的容器的 Pod 时，会看到 容器进入此状态的原因、退出代码以及容器执行期间的起止时间。

Pod的定义

配置文件定义

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-pod
  namespace: default
  labels:
    name: myhello-pod
    env: dev
spec:
  restartPolicy: Always
  containers:
  - name: myhello
    image: xlhmzch/hello:1.0.0
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
```

关键词

restartPolicy

容器重启策略，可选值为：Always、Never、OnFailure，默认值为 Always。restartPolicy 适用于 Pod 中的所有容器。当pod中的容器退出时，kubelet 会按指数回退 方式计算重启的延迟（10s、20s、40s、...），其最长延迟为 5 分钟。 一旦某容器执行了 10 分钟并且没有出现问题，kubelet 对该容器的重启回退计时器执行 重置操作。

imagePullPolicy

镜像拉取策略，可选值为：Always、Never、IfNotPresent，默认为：Always
Always: 表示每次都尝试重新拉取镜像
Never:表示从不拉取镜像，仅使用本地镜像
IfNotPresent:表示如果本地有该镜像，这使用该镜像；如果本地没有该镜像这拉取镜像。

command

指定容器启动命令，若未指定该值则默认为镜像中指定的启动命令

args

容器启动命令参数，若未指定则该值默认为容器镜像中指定的启动参数

resources

容器使用资源设置
requests：表示为容器分配最低资源配额
limits：表示容器可使用的最高资源配额，一旦容器资源的使用超出了该配置，那么容器将会被杀死。
CPU资源单位：Kubernetes将1CPU以1000m来表示，CPU的最小资源单位为m，1m表示千分之一CPU。通常一个容器使用的CPU配额为100m~ 300m。
内存资源单位：例如：Ei、Pi、Ti、Gi、Mi、Ki，与通常使用的空间单位一致。

Pod的使用

创建并访问Pod

1. 创建pod

```
kubectl create -f myhello-pod.yaml
```

3. 访问pod中的容器

```
# 通过端口转发来查看该pod是否可以正常提供服务
kubectl port-forward pod/myhello-pod 5000:80

# 通过curl访问，该pod，查看其环境变量、启动参数等信息是否生效
curl http://localhost:5000/print/env
curl http://localhost:5000/print/startup
```

多个应用容器

1. 定义

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-pod
  namespace: default
  labels:
    name: myhello-pod
    env: dev
spec:
  restartPolicy: Always
  containers:
  - name: myhello
    image: xlhmzch/hello:1.0.0
    imagePullPolicy: IfNotPresent
```

```
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
  - name: myredis                    #容器的名称
    image: redis       #容器对应的 Docker Image
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
```

```
# 根据文件删除资源
kubectl delete -f myhello-pod.yaml
# 根据文件应用资源
kubectl apply -f myhello-pod.yaml
```

2. 访问pod中的容器

```
# 通过端口转发来查看该pod是否可以正常提供服务
kubectl port-forward pod/myhello-pod 5001:80 5002:6379 --address 192.168.239.142
--address localhost

# 访问myhello容器
curl http://localhost:5001/ping

# 访问redis，通过新创建一个其他redis，然后通过其他redis的客户端访问上面redis-server
kubectl run myredis --image=redis
# 进入redis交互
kubectl exec -it pod/myredis --  /bin/bash

# 以转发的地址和端口访问
redis-cli -h 192.168.239.142 -p 5002
# 以pod ip 访问
redis-cli -h <podIP> -p 6379
```

Init容器

Init容器与普通容器的区别

Init 容器是一种特殊容器，在 Pod 内的应用容器启动之前运行。Init 容器可以包括一些应用镜像中不存在的实用工具和安装脚本。
每个 Pod 中可以包含多个容器， 应用运行在这些容器里面，同时 Pod 也可以有一个或多个先于应用容器启动的 Init 容器。
如果 Pod 的 Init 容器失败，kubelet 会不断地重启该 Init 容器直到该容器成功为止。 然而，如果 Pod 对应的 restartPolicy 值为 "Never"，并且 Pod 的 Init 容器失败， 则 Kubernetes 会将整个 Pod 状态设置为失败。
Init 容器与普通容器的区别：

1. 它们总是运行到完成，即init容器提供的不是通过守护进程提供服务，而是通过启动容器来执行处理任务，当任务处理完成容器即完成。
2. 每个都必须在下一个启动之前成功完成，多个init容器按定义的顺序一次执行。
3. 只有当所有init容器完成时，Kubernetes才会为Pod初始化应用容器。

Init容器的使用

init 容器的特殊性，依次执行、容器必须来到完成状态、init容器完成后才会启动应用容器

1. 定义

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-pod
  namespace: default
  labels:
    name: myhello-pod
    env: dev
spec:
  restartPolicy: Always
  initContainers:
  - name: init-myservice
    image: busybox
    # 查找命名空间下myservice服务，如果存在则执行成功，如果不存在则一直查找
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox
    # 查找命名空间下mydb服务，如果存在则执行成功，如果不存在则一直查找
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for mydb; sleep 2; done"]
  containers:
  - name: myhello
    image: xlhmzch/hello:1.0.0
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
```

```
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
  - name: myredis                    #容器的名称
    image: redis       #容器对应的 Docker Image
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
```

2. 创建容器

```
# 删除原有pod
kubectl delete -f myhello-pod.yaml
# 重新应用
kubectl apply -f myhello-pod.yaml
```

3. 获取pod，可以看到pod一直处于init阶段

```
 # 获取pod
 kubectl get -f myhello-pod.yaml
```

```
nick@k8s-master1:~/work$ kubectl get -f myhello-init-pod-demo.yaml
NAME                  READY   STATUS     RESTARTS   AGE
myhello-init-pod-demo   0/2     Init:0/2   0          5m34s
nick@k8s-master1:~/work$
```

4. 查看pod详情

```
 kubectl describe -f myhello-pod.yaml
```

输出详细信息，pod状态为Pending，容器状态：仅init-myservice 为running，其他容器均为：Waiting

5. 查看容器日志

```
kubectl logs -f  myhello-pod -c init-myservice
```

6. 创建myservice服务

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

```
kubectl create -f myservice.yaml
```

7. 查看pod状态，容器日志

```
kubectl get -f myhello-pod.yaml

kubectl describe -f myhello-pod.yaml

kubectl logs -f  myhello-pod -c init-mydb
```

8. 创建mydb服务

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

```
kubectl create -f mydb.yaml
```

9. 查看pod状态，容器日志

```
kubectl get -f myhello-pod.yaml

kubectl describe -f myhello-pod.yaml

kubectl logs -f  myhello-pod -c init-mydb
```

以上案例可以看出：

1. init容器按定义顺序依次执行
2. init容器的启动命令必须为有执行结果的命令，否则init容器将无法来到完成状态
3. init容器全部完成之后才会初始化应用程序容器（普通容器）

容器的生命周期处理函数

Kubernetes 支持 postStart 和 preStop 事件。 当一个容器启动后，Kubernetes 将立即发送 postStart 事件；在容器被终结之前， Kubernetes 将发送一个 preStop 事件。容器可以为每个事件指定一个处理程序。

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-pod
  namespace: default
  labels:
    name: myhello-pod
    env: dev
spec:
  restartPolicy: Always
  initContainers:
  - name: init-myservice
    image: busybox
    # 查找命名空间下myservice服务，如果存在则执行成功，如果不存在则一直查找
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox
    # 查找命名空间下mydb服务，如果存在则执行成功，如果不存在则一直查找
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for mydb; sleep 2; done"]
  containers:
  - name: myhello
    image: xlhmzch/hello:1.0.0
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c","echo post start command exec >> /tmp/data"]
      preStop:
        exec:
          command: ["/bin/sh", "-c","echo pre stop command exec >> /tmp/data"]
  - name: myredis                    #容器的名称
    image: redis       #容器对应的 Docker Image
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 6379
```

```
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
```

```
kubectl delete -f myhello-pod.yaml

kubectl apply -f myhello-pod.yaml
```

```
# 使用shell 连接到pod中的容器
kubectl exec -it pod/myhello-pod -c myhello -- sh
# 查看postStart处理函数写入的文本
cat /tmp/data
```

容器的探测

探测类型及使用场景

startupProbe（启动探测）

指示容器中的应用是否已经启动。如果提供了启动探针，则所有其他探针都会被禁用，直到此探针成功为止。探测成功之后，启动探测停止。如果启动探测失败，kubelet 将杀死容器，而容器依其 重启策略进行重启。 如果容器没有提供启动探测，则默认状态为 Success。
使用场景：

1. 容器需要较长时间才能启动就绪的 Pod ，可以指定启动探针。那么可以避免设置一个超长时间间隔的存活探测。例如：容器需要在启动期间加载大型数据、配置文件或执行迁移等

readinessProbe（就绪探测）

指示容器是否准备好为请求提供服务。如果就绪态探测失败， 端点控制器将从与 Pod 匹配的所有服务的端点列表中删除该 Pod 的 IP 地址。 初始延迟之前的就绪态的状态值默认为 Failure。 如果容器不提供就绪态探针，则默认状态为 Success。就绪探测失败时，不会杀死容器进程，只会将端点从service中移除，使其收不到流量。
使用场景：

1. 仅在探测成功时才开始向 Pod 发送请求流量，意味着pod只在探测成功之后才开始接受请求数据。
2. 如果应用程序对后端服务有严格的依赖性，可以同时实现存活态和就绪态探针，存活态探针检测通过后，可以通过就绪态探针会额外检查每个所需的后端服务是否可用

livenessProbe（存活探测）

指示容器是否正在运行。如果存活态探测失败，则 kubelet 会杀死容器， 并且容器将根据其重启策略决定未来。如果容器不提供存活探针， 则默认状态为 Success。
使用场景：

1. 如果希望容器在探测失败时被杀死并重新启动，并指定 restartPolicy 为 "Always" 或 "OnFailure"。

2. 如果容器中的进程能够在遇到问题或不健康的情况下自行崩溃，可以不需要指定存活探针。kubelet 将根据 Pod 的 restartPolicy 自动执行修复操作。

检查机制

exec

在容器内执行指定命令。如果命令退出时**返回码为 0** 则认为诊断成功。

httpGet

对容器的 IP 地址上指定端口和路径执行 HTTP GET 请求。如果**响应的状态码大于等于 200 且小于 400**，则诊断被认为是成功的。

tcpSocket

对容器的 IP 地址上的指定端口执行 TCP 检查。**如果端口打开**，则诊断被认为是成功的。

grpc

使用 gRPC 执行一个远程过程调用。 目标应该实现 gRPC健康检查。 如果响应的**状态是 "SERVING"**，则认为诊断成功。 gRPC 探针是一个 alpha 特性，只有在你启用了 "GRPCContainerProbe" 特性时才能使用

探测结果

Success（成功），容器通过了诊断。
Failure（失败），容器未通过诊断。
Unknown（未知），诊断失败，因此不会采取任何行动。

容器探测案例

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-pod
  namespace: default
  labels:
    name: myhello-pod
    env: dev
spec:
  restartPolicy: Always
  initContainers:
  - name: init-myservice
    image: busybox
    # 查找命名空间下myservice服务，如果存在则执行成功，如果不存在则一直查找
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox
    # 查找命名空间下mydb服务，如果存在则执行成功，如果不存在则一直查找
```

```
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo waiting for mydb; sleep 2; done"]
  containers:
  - name: myhello
    image: xlhmzch/hello:curl
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c","echo post start command exec >> /tmp/data"]
      preStop:
        exec:
          command: ["/bin/sh", "-c","echo pre stop command exec >> /tmp/data"]
    startupProbe:
      exec:
        command: ["/bin/sh", "-c","statuscode=`curl -o /dev/null -s -w %
{http_code} http://localhost/healthz`; [ $$statuscode -le 400 ] || exit 1"]
      # 指定初始化时间，即探针在容器启动之后的延迟时长
      initialDelaySeconds: 5
      # 探测周期，即每隔多长时间探测一次
      periodSeconds: 5
      # 最小连续失败次数，即连续失败多少次表示探测失败
      failureThreshold: 3
      # 最小连续成功次数，即连续探测成功多少次表示探测成功，liveness 和 startup 必须为1
      successThreshold: 1
      # 探测超时时间
      timeoutSeconds: 1
    readinessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
      failureThreshold: 3
      successThreshold: 1
      timeoutSeconds: 1
    livenessProbe:
      httpGet:
```

```
        path: /health
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
      failureThreshold: 3
      successThreshold: 1
      timeoutSeconds: 1
  - name: myredis                    #容器的名称
    image: redis       #容器对应的 Docker Image
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 200Mi
    livenessProbe:
      tcpSocket:
        port: 6379
      initialDelaySeconds: 5
      periodSeconds: 5
      failureThreshold: 3
      successThreshold: 1
      timeoutSeconds: 1
```

Pod的部署

Pod 可通过Deployment、DaemonSet、StatefulSet、Job等方式部署到k8s集群。

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy
  namespace: default
  labels:
    name: myapp-deploy
spec:
  replicas: 5
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      restartPolicy: Always
      containers:
      - name: myhello
```

```
        image: xlhmzch/hello:curl
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        command: ["./app"]
        args: ["--param1=k8s-p1", "--param2=k8s-p2"]
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        lifecycle:
          postStart:
            exec:
              command: ["/bin/sh", "-c","echo post start command exec >>
/tmp/data"]
          preStop:
            exec:
              command: ["/bin/sh", "-c","echo pre stop command exec >>
/tmp/data"]
        startupProbe:
          exec:
            command: ["/bin/sh", "-c","statuscode=`curl -o /dev/null -s -w %
{http_code} http://localhost/healthz`; [ $$statuscode -le 400 ] || exit 1"]
          # 指定初始化时间，即探针在容器启动之后的延迟时长
          initialDelaySeconds: 5
          # 探测周期，即每隔多长时间探测一次
          periodSeconds: 5
          # 最小连续失败次数，即连续失败多少次表示探测失败
          failureThreshold: 3
          # 最小连续成功次数，即连续探测成功多少次表示探测成功，liveness 和 startup 必须为1
          successThreshold: 1
          # 探测超时时间
          timeoutSeconds: 1
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
```

```
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
      - name: myredis                    #容器的名称
        image: redis       #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 6379
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        livenessProbe:
          tcpSocket:
            port: 6379
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
```

```
# 应用资源
kubectl apply -f myapp-deployment.yaml
```

```
# 应用新版配置文件，实现deployment升级
kubectl apply -f myapp-deployment.yaml

# 通过 rollout命令可以实现deployment的发布相关功能（发布状态、发布历史、暂停、继续、回滚等）
kubectl rollout

# 通过 scale 命令可以实现deployment的伸缩
kubectl scale

# 通过 autoscale 可以实现deployment的自动伸缩
kubectl autoscale
```

DaemonSets

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: myapp-ds
  namespace: default
  labels:
    name: myapp-ds
spec:
  selector:
    matchLabels:
      app: myapp-ds
```

```
  template:
    metadata:
      labels:
        app: myapp-ds
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      restartPolicy: Always
      containers:
      - name: myhello
        image: xlhmzch/hello:curl
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        command: ["./app"]
        args: ["--param1=k8s-p1", "--param2=k8s-p2"]
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        env:    # 注入到容器的环境变量
        - name: env1
          value: "k8s-env1"
        - name: env2
          value: "k8s-env2"
        lifecycle:
          postStart:
            exec:
              command: ["/bin/sh", "-c","echo post start command exec >>
/tmp/data"]
          preStop:
            exec:
              command: ["/bin/sh", "-c","echo pre stop command exec >>
/tmp/data"]
        startupProbe:
          exec:
            command: ["/bin/sh", "-c","statuscode=`curl -o /dev/null -s -w %
{http_code} http://localhost/healthz`; [ $$statuscode -le 400 ] || exit 1"]
          # 指定初始化时间，即探针在容器启动之后的延迟时长
          initialDelaySeconds: 5
          # 探测周期，即每隔多长时间探测一次
          periodSeconds: 5
          # 最小连续失败次数，即连续失败多少次表示探测失败
          failureThreshold: 3
          # 最小连续成功次数，即连续探测成功多少次表示探测成功，liveness 和 startup 必须为1
          successThreshold: 1
          # 探测超时时间
          timeoutSeconds: 1
        readinessProbe:
          httpGet:
```

```
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
      - name: myredis                    #容器的名称
        image: redis       #容器对应的 Docker Image
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 6379
        resources:
          limits:
            cpu: 200m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        livenessProbe:
          tcpSocket:
            port: 6379
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
          successThreshold: 1
          timeoutSeconds: 1
```

```
kubectl create -f myapp-ds.yaml
```

```
# 通过 rollout命令可以实现daemonset的发布相关功能（发布状态、发布历史、暂停、继续、回滚等）
kubectl rollout
```

静态pod

静态pod定义在Node的具体文件当中，有kubelet组件直接管理。

```
# 打开kubelet的service文件，找到环境变量 KUBELET_CONFIG_ARGS 指定的--config参数
# --config参数，默认情况下指定为：/var/lib/kubelet/config.yaml 文件
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf

# 打开kubelet配置文件，找到staticPodPath，该字段指定了静态pod的路径
# 默认为 /etc/kubernetes/manifests/
vim /var/lib/kubelet/config.yaml
```

创建静态pod

```
apiVersion: v1
kind: Pod
metadata:
  name: myhello-static-pod
  namespace: default
  labels:
    name: myhello-static-pod
    env: dev
spec:
  containers:
  - name: myhello
    image: xlhmzch/hello:1.0.0
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    command: ["./app"]
    args: ["--param1=k8s-p1", "--param2=k8s-p2"]
    resources:
      limits:
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 200Mi
    env:    # 注入到容器的环境变量
    - name: env1
      value: "k8s-env1"
    - name: env2
      value: "k8s-env2"
```

```
# 在静态pod文件目录新增pod定义文件，kubelet会自动创建新增的静态pod
vim /etc/kubernetes/manifests/myhello-static-pod.yaml

# 删除pod，kubelet组件会重启该pod
kubectl delete pod myhello-static-pod

# 删除静态pod
rm /etc/kubernetes/manifests/myhello-static-pod.yaml
```