上一篇文章我们从背景和设计的角度介绍了 Kubernetes 的 Pod，这篇文章我们来看一下 Pod 如何使用？需要提前说明的一下的是，真正在生产环境中，我们很少会直接去创建 Pod 对象，更多是通过其他控制器来创建，比如 Deployment、Daemonset，在这些控制器中，Pod 将会被定义为其中的一个 template，这个我们后面再说，这一节还是以介绍 Pod 的使用为主。

1. 概览

本文将从一个最简单的 Pod 的例子开始，然后由浅入深地介绍 Pod 的特性。首先下面是一个最简单 Pod 示例。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
```

其中包含的字段包括：

- apiVersion：表示 api 对象的版本（比如 Pod 就是一种 api 对象）；
- kind : 表明 api 对象类型，Pod 对应的 kind 是 `Pod`；
- metadata : 包含一些元信息，比如 `name` 、 `labels` 等；
- spec：定义了 Pod 的一些描述信息，重要信息都是在 spec 这里进行描述的，比如：
    - containers：Pod 中运行的容器的镜像列表，可以包含多个，我们后面细说；
    - affinity：亲和性，在 Pod 调度的时候使用，比如我要把特定的 Pod 调度到特定的节点上，就可以使用这个特性；
    - hostAliases：hosts 条目，会在 Pod 启动的时候注入到 Pod 中；
    - hostIPC：Pod 内的容器使用宿主机的 IPC namespace，默认是 false；
    - hostNetwork：Pod 内的容器使用宿主机的网络 namespace，相当于 Docker 网络中的 host network，默认为 false；
    - hostPID：Pod 内的容器使用宿主机的 pid namespace，也就是同一个进程空间；
    - hostname：指定 Pod 的 hostname；
    - dnsPolicy：Pod 内的容器的 dns 策略；
    - imagePullSecrets：我们有时候使用的镜像是私有的，需要指定用户名密码，就是通过这个字段来指定，一般是将镜像的用户名密码配置成一个 Secret；
    - initContainers：init 容器，Pod 中的一种特殊容器，会先启动。后面会细说；
    - nodeSelector：在调度的时候，有时候我们希望 Pod 被调度到指定的 node 节点上，那么我们就可以使用这个特性；

    - restartPolicy：Pod 内的容器的重启策略，当容器异常退出或者健康检查失败时，kubelet 将根据该字段的值来进行响应的操作，取值包括：
        - Always：容器失效时，由 kubelet 自动重启该容器；
        - OnFailure：当容器终止运行且退出码不为 0 时，由 kubelet 自动重启；
        - Never：不论容器如何失败，kubelet 都不会重启该容器。

其中最重要的是 spec.containers 字段，containers 下面是一个 List，包含一些列的 container。我们看一下 container 支持的重要字段：

- name：启动之后的容器名字；
- image：镜像名称；
- imagePullPolicy：描述 Pod 启动时不同的镜像拉取策略，取值有 `Always`， `Never`、`IfNotPresent`，含义分别为：
    - Always：每次 Pod 启动或者重启，都去镜像中心拉取镜像；
    - Never：每次 Pod 启动或者重启，都不去拉取镜像，而是使用本地的；
    - IfNotPresent：每次 Pod 启动或者重启时，先从本地找有没有该镜像，如果本地存在，则使用本地的，否则去镜像中心拉取；
- command：我们知道镜像可以有一个默认的 EntryPoint，这个 command 相当于我们使用一个新的执行代码覆盖容器默认的 EntryPoint；
- args：参数，和 command 配合使用；
- env：注入到容器内部的环境变量；
- livenessProbe：存活性检测；
- readinessProbe：可用性检测。这个是为了和存活性进行区分的，比如有些进程存在，可能并不代表可以正常对外服务；
- volumeMounts：存储卷挂载点。

我们下面通过几个例子来循序渐进的理解这些特性。

2. command

---

command 的使用场景主要是当容器的默认 entrypoint 一直启动失败时，我们想要进行调试，那么可以用 command 来覆盖容器的默认 entrypoint，比如将 command 设置成功 `/bin/bash` 或者 `sh`。下面我们将 busybox 镜像的默认 entrypoint 覆盖，用一个 shell 命令替换。

```
apiversion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

3. affinity

Kubernetes 的 Pod 的亲和性提供了一种调度上的遍历行，主要包括三种：

- nodeAffinity：描述了 Pod 和 Node 之间的调度关系，比如把 Pod 调度到含有指定的标签的 Node 节点上；
- podAffinity：描述了 Pod 之间的调度关系，比如将某两种 Pod 调度到指定的节点上；
- podAntiAffinity：和 podAffinity 正好相反，这个叫反亲和，比如让某两种 Pod 不要调度到同一个节点。

下面是一个应用 `nodeAffinity` 的 Pod 的例子。其中的 `requiredDuringSchedulingIgnoredDuringExecution` 是 `nodeAffinity` 的一种策略，表示 Pod 必须部署到满足条件的节点上，如果没有满足条件的节点，就不停重试。其中 IgnoreDuringExecution 表示 Pod 部署之后运行的时候，如果节点标签发生变化，不再满足 Pod 指定的条件，Pod 也会继续运行。除此之外，`nodeAffinity` 还支持其他策略，比如：

- requiredDuringSchedulingRequiredDuringExecution：类似 `requiredDuringSchedulingIgnoredDuringExecution`，不过如果节点标签发生了变化，不再满足 pod 指定的条件，则重新选择符合要求的节点。
- preferredDuringSchedulingIgnoredDuringExecution：表示优先部署到满足条件的节点上，如果没有满足条件的节点，就忽略这些条件，按照正常逻辑部署。
- preferredDuringSchedulingIgnoredDuringExecution：表示优先部署到满足条件的节点上，如果没有满足条件的节点，就忽略这些条件，按照正常逻辑部署。其中 RequiredDuringExecution 表示如果后面节点标签发生了变化，满足了条件，则重新调度到满足条件的节点。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  affinity:
    nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: <label-name>
                operator: In
                values:
                - <value>
  containers:
  - name: myapp-container
    image: busybox:latest
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

4. hostAliases

使用过 Docker 的同学应该都有印象，我们不能直接修改 Docker 容器中的 /etc/hosts 文件，所以就有一些奇淫技巧，比如在 Docker 镜像的 entrypoint 中去修改 hosts 文件，这其实是一种非常不规范的做法。

在 Kubernetes 中，我们有时候也要修改 Pod 的 hosts 文件，我们使用的方式是 Kubernetes 中提供了 Pod 的特性：hostAlias，非常简单，下面是我们一个实际的例子。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod-hostalias
  labels:
    app: myapp
spec:
  hostAliases:
  - ip: "1.2.3.4"
    hostnames:
    - "foo.local"
    - "bar.local"
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

我们通过 `kubectl apply` 一下，然后看一下 Pod 内部的 hosts 文件情况，最后一行就是我们添加的条目。

```
kubectl exec myapp-pod-hostalias -n imooc -- cat /etc/hosts

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
10.1.2.134  myapp-pod-hostalias


1.2.3.4 foo.local   bar.local
```

5. Init Container

Pod 可以包含多个容器，有时候我们想要某个或者某几个容器先于其他容器启动，这个时候我们就需要 init containenr。init container 与普通的容器区别在于：

- init containenr 总是运行到完成；
- 每个 init container 运行完成，下一个容器才会运行。如果有多个 init container，则按顺序启动。

如果 Pod 的 init container 运行失败，Kubernetes 会不断地重启该 Pod，知道 init container 成功为止，除非 restartPolicy 值为 Never。init containenr 在 Pod 的 spec 中定义和 containenr 在一个层级，具备的功能和普通容器没有区别。

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app-container
    image: ...
  initContainers:
  - name: init-container-1
    image: ...
  - name: init-container-2
    image: ...
```

init container 的好处

init container 使用与应用容器分离的单独镜像，具有如下优势:

- init container 可以包含一些安装过程中应用容器中不存在的使用工具或个性化代码。例如，没有必要仅仅为了在安装过程中使用类似 `sed`、`awk`、`python`、`dig` 这样的工具而去 `FROM` 一个镜像来生成新的镜像；
- init container 可以安全地运行这些工具，避免这些工具导致应用镜像的安全性降低；
- 应用镜像的创建者和部署者可以各自独立工作，而没有必要联合构建一个单独的应用镜像；
- init container 能以不同于 Pod 内应用容器的文件系统视图运行。因此，Init 容器可具有访问 Secrets 的权限，而应用容器不能够访问；
- 由于 init container 必须在应用容器启动之前运行完成，因此 init container 提供了一种机制来阻塞或延迟应用容器的启动，直到满足了一组先决条件。一旦前置条件满足，Pod 内的所有的应用容器会并行启动。

init container 使用

下面是官方的 init container 的使用示例，这个 Pod 里面的应用容器依赖于两个服务：`myservice` 和 `mydb`，需要这两个服务启动之后，应用容器才能启动。那么我们就可以定义两个 init container 用来检测依赖服务有没有启动：一个等待 `myservice` 启动；一个等待 `mydb` 启动。其中检测的方式是检测 service 对应 dns 域名有没有。

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    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:1.28
```

```
    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"]
```

为了便于演示，下面的对象定义了两个依赖的 service。

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
---
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

为了直观看到 init container 的作用，我们先启动 Pod（我们这里把 demo 启动到一个专门的 namespace `imooc` 里面）。

```
➜  kubectl apply -f myapp.yaml -n imooc
```

然后查看 Pod 状态，从 `STATUS` 字段发现 Pod 启动还卡在 Init 上，也就是 init container，因为我们还没有启动 service。

```
➜  kubectl get pods -n imooc
NAME        READY   STATUS     RESTARTS   AGE
myapp-pod   0/1     Init:0/2   0          59s
```

我们下面使用上面的 yaml 启动两个依赖的 service：`myservice` 和 `mydb`。

```
➜  kubectl apply -f myservice-mydb.yaml -n imooc
service/myservice created
service/mydb created
```

这时候我们再看 Pod 的运行状态，发现 `STATUS` 已经变成 ***Running*** 了。

```
➜  kubectl get pods -n imooc
NAME         READY   STATUS    RESTARTS   AGE
myapp-pod    1/1     Running   0          4m27s
```

6. 总结

本文从具体的一个例子开始介绍 Pod 的使用和 Pod 的一些特性，但是 Pod 的特性实在是太多了，想通过一篇文章介绍完基本不可能，所以希望大家还是可以多实践。

}