14 | 深入解析Pod对象（一）：基本概念

14 | 深入解析Pod对象（一）：基本概念

张磊 2018-09-24

□

11:47

讲述：张磊 大小：5.41M

你好，我是张磊。今天我和你分享的主题是：深入解析 Pod 对象之基本概念。

在上一篇文章中，我详细介绍了 Pod 这个 Kubernetes 项目中最重要的概念。而在今天这篇文章中，我会和你分享 Pod 对象的更多细节。

现在，你已经非常清楚：Pod，而不是容器，才是 Kubernetes 项目中的最小编排单位。将这个设计落实到 API 对象上，容器（Container）就成了 Pod 属性里的一个普通的字段。那么，一个很自然的问题就是：到底哪些属性属于 Pod 对象，而又有哪些属性属于 Container 呢？

要彻底理解这个问题，你就一定要牢记我在上一篇文章中提到的一个结论：Pod 扮演的是传统部署环境里“虚拟机”的角色。这样的设计，是为了使用户从传统环

境（虚拟机环境）向 Kubernetes（容器环境）的迁移，更加平滑。

而如果你能把 Pod 看成传统环境里的“机器”、把容器看作是运行在这个“机器”里的“用户程序”，那么很多关于 Pod 对象的设计就非常容易理解了。

比如，**凡是调度、网络、存储，以及安全相关的属性，基本上是 Pod 级别的。**

这些属性的共同特征是，它们描述的是“机器”这个整体，而不是里面运行的“程序”。比如，配置这个“机器”的网卡（即：Pod 的网络定义），配置这个“机器”的磁盘（即：Pod 的存储定义），配置这个“机器”的防火墙（即：Pod 的安全定义）。更不用说，这台“机器”运行在哪个服务器之上（即：Pod 的调度）。

接下来，我就先为你介绍 Pod 中几个重要字段的含义和用法。

NodeSelector：是一个供用户将 Pod 与 Node 进行绑定的字段，用法如下所示：

```
apiVersion: v1
kind: Pod
...
spec:
nodeSelector:
disktype: ssd
```

复制代码

这样的一个配置，意味着这个 Pod 永远只能运行在携带了“disktype: ssd”标签（Label）的节点上；否则，它将调度失败。

NodeName：一旦 Pod 的这个字段被赋值，Kubernetes 项目就会被认为这个 Pod 已经经过了调度，调度的结果就是赋值的节点名字。所以，这个字段一般由调度器负责设置，但用户也可以设置它来“骗过”调度器，当然这个做法一般是在测试或者调试的时候才会用到。

HostAliases：定义了 Pod 的 hosts 文件（比如 /etc/hosts）里的内容，用法如下：

```
apiVersion: v1
kind: Pod
...
spec:
```

```
hostAliases:
- ip: "10.1.2.3"
hostnames:
- "foo.remote"
- "bar.remote"
...
```

□复制代码

在这个 Pod 的 YAML 文件中，我设置了一组 IP 和 hostname 的数据。这样，这个 Pod 启动后，/etc/hosts 文件的内容将如下所示：

```
cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1 localhost
...
10.244.135.10 hostaliases-pod
10.1.2.3 foo.remote
10.1.2.3 bar.remote
```

□复制代码

其中，最下面两行记录，就是我通过 HostAliases 字段为 Pod 设置的。需要指出的是，在 Kubernetes 项目中，如果要设置 hosts 文件里的内容，一定要通过这种方法。否则，如果直接修改了 hosts 文件的话，在 Pod 被删除重建之后，kubelet 会自动覆盖掉被修改的内容。

除了上述跟“机器”相关的配置外，你可能也会发现，**凡是跟容器的 Linux Namespace 相关的属性，也一定是 Pod 级别的。**这个原因也很容易理解：Pod 的设计，就是要让它里面的容器尽可能多地共享 Linux Namespace，仅保留必要的隔离和限制能力。这样，Pod 模拟出的效果，就跟虚拟机里程序间的关系非常类似了。

举个例子，在下面这个 Pod 的 YAML 文件中，我定义了 shareProcessNamespace=true：

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
```

```
shareProcessNamespace: true
containers:
- name: nginx
image: nginx
- name: shell
image: busybox
stdin: true
tty: true
```

复制代码

这就意味着这个 Pod 里的容器要共享 PID Namespace。

而在这个 YAML 文件中，我还定义了两个容器：一个是 nginx 容器，一个是开启了 tty 和 stdin 的 shell 容器。

我在前面介绍容器基础时，曾经讲解过什么是 tty 和 stdin。而在 Pod 的 YAML 文件里声明开启它们俩，其实等同于设置了 docker run 里的 -it（-i 即 stdin，-t 即 tty）参数。

如果你还是不太理解它们俩的作用的话，可以直接认为 tty 就是 Linux 给用户提供的一个常驻小程序，用于接收用户的标准输入，返回操作系统的标准输出。当然，为了能够在 tty 中输入信息，你还需要同时开启 stdin（标准输入流）。

于是，这个 Pod 被创建后，你就可以使用 shell 容器的 tty 跟这个容器进行交互了。我们一起实践一下：

```
$ kubectl create -f nginx.yaml
```

复制代码

接下来，我们使用 kubectl attach 命令，连接到 shell 容器的 tty 上：

```
$ kubectl attach -it nginx -c shell
```

复制代码

这样，我们就可以在 shell 容器里执行 ps 指令，查看所有正在运行的进程：

```
$ kubectl attach -it nginx -c shell
/ # ps ax
PID USER TIME COMMAND
1 root 0:00 /pause
8 root 0:00 nginx: master process nginx -g daemon off;
```

```
14 101 0:00 nginx: worker process
15 root 0:00 sh
21 root 0:00 ps ax
```

□复制代码

可以看到，在这个容器里，我们不仅可以看到它本身的 ps ax 指令，还可以看到 nginx 容器的进程，以及 Infra 容器的 /pause 进程。这就意味着，整个 Pod 里的每个容器的进程，对于所有容器来说都是可见的：它们共享了同一个 PID Namespace。

类似地，**凡是 Pod 中的容器要共享宿主机的 Namespace，也一定是 Pod 级别的定义**，比如：

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
hostNetwork: true
hostIPC: true
hostPID: true
containers:
- name: nginx
image: nginx
- name: shell
image: busybox
stdin: true
tty: true
```

□复制代码

在这个 Pod 中，我定义了共享宿主机的 Network、IPC 和 PID Namespace。这就意味着，这个 Pod 里的所有容器，会直接使用宿主机的网络、直接与宿主机进行 IPC 通信、看到宿主机里正在运行的所有进程。

当然，除了这些属性，Pod 里最重要的字段当属“Containers”了。而在上一篇文章中，我还介绍过“Init Containers”。其实，这两个字段都属于 Pod 对容器的定义，内容也完全相同，只是 Init Containers 的生命周期，会先于所有的 Containers，并且严格按照定义的顺序执行。

Kubernetes 项目中对 Container 的定义，和 Docker 相比并没有什么太大区别。我在前面的容器技术概念入门系列文章中，和你分享的 Image（镜像）、Command（启动命令）、workingDir（容器的工作目录）、Ports（容器要开发的端口），以及 volumeMounts（容器要挂载的 Volume）都是构成 Kubernetes 项目中 Container 的主要字段。不过在这里，还有这么几个属性值得你额外关注。

首先，是 ImagePullPolicy 字段。它定义了镜像拉取的策略。而它之所以是一个 Container 级别的属性，是因为容器镜像本来就是 Container 定义中的一部分。

ImagePullPolicy 的值默认是 Always，即每次创建 Pod 都重新拉取一次镜像。另外，当容器的镜像是类似于 nginx 或者 nginx:latest 这样的名字时，ImagePullPolicy 也会被认为 Always。

而如果它的值被定义为 Never 或者 IfNotPresent，则意味着 Pod 永远不会主动拉取这个镜像，或者只在宿主机上不存在这个镜像时才拉取。

其次，是 Lifecycle 字段。它定义的是 Container Lifecycle Hooks。顾名思义，Container Lifecycle Hooks 的作用，是在容器状态发生变化时触发一系列“钩子”。我们来看这样一个例子：

```
apiVersion: v1
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"]
```

□复制代码

这是一个来自 Kubernetes 官方文档的 Pod 的 YAML 文件。它其实非常简单，只是定义了一个 nginx 镜像的容器。不过，在这个 YAML 文件的容器（Containers）部分，你会看到这个容器分别设置了一个 postStart 和 preStop 参数。这是什么意思呢？

先说 postStart 吧。它指的是，在容器启动后，立刻执行一个指定的操作。需要明确的是，postStart 定义的操作，虽然是在 Docker 容器 ENTRYPOINT 执行之后，但它并不严格保证顺序。也就是说，在 postStart 启动时，ENTRYPOINT 有可能还没有结束。

当然，如果 postStart 执行超时或者错误，Kubernetes 会在该 Pod 的 Events 中报出该容器启动失败的错误信息，导致 Pod 也处于失败的状态。

而类似地，preStop 发生的时机，则是容器被杀死之前（比如，收到了 SIGKILL 信号）。而需要明确的是，preStop 操作的执行，是同步的。所以，它会阻塞当前的容器杀死流程，直到这个 Hook 定义操作完成之后，才允许容器被杀死，这跟 postStart 不一样。

所以，在这个例子中，我们在容器成功启动之后，在 /usr/share/message 里写入了一句“欢迎信息”（即 postStart 定义的操作）。而在这个容器被删除之前，我们则先调用了 nginx 的退出指令（即 preStop 定义的操作），从而实现了容器的“优雅退出”。

在熟悉了 Pod 以及它的 Container 部分的主要字段之后，我再和你分享一下**这样一个的 Pod 对象在 Kubernetes 中的生命周期**。

Pod 生命周期的变化，主要体现在 Pod API 对象的**Status 部分**，这是它除了 Metadata 和 Spec 之外的第三个重要字段。其中，pod.status.phase，就是 Pod 的当前状态，它有如下几种可能的情况：

1. Pending。这个状态意味着，Pod 的 YAML 文件已经提交给了 Kubernetes，API 对象已经被创建并保存在 Etcd 当中。但是，这个 Pod 里有些容器因为某种原因而不能被顺利创建。比如，调度不成功。
2. Running。这个状态下，Pod 已经调度成功，跟一个具体的节点绑定。它包含的容器都已经创建成功，并且至少有一个正在运行中。
3. Succeeded。这个状态意味着，Pod 里的所有容器都正常运行完毕，并且已经退出了。这种情况在运行一次性任务时最为常见。
4. Failed。这个状态下，Pod 里至少有一个容器以不正常的状态（非 0 的返回码）退出。这个状态的出现，意味着你得想办法 Debug 这个容器的应用，比如查看 Pod 的 Events 和日志。
5. Unknown。这是一个异常状态，意味着 Pod 的状态不能持续地被 kubelet 汇报给 kube-apiserver，这很有可能是主从节点（Master 和 Kubelet）间的通信出现了问题。

更进一步地，Pod 对象的 Status 字段，还可以再细分出一组 Conditions。这些细分状态的值包括：PodScheduled、Ready、Initialized，以及 Unschedulable。它们主要用于描述造成当前 Status 的具体原因是什么。

比如，Pod 当前的 Status 是 Pending，对应的 Condition 是 Unschedulable，这就意味着它的调度出现了问题。

而其中，Ready 这个细分状态非常值得我们关注：它意味着 Pod 不仅已经正常启动（Running 状态），而且已经可以对外提供服务了。这两者之间（Running 和 Ready）是有区别的，你不妨仔细思考一下。

Pod 的这些状态信息，是我们判断应用运行情况的重要标准，尤其是 Pod 进入了非“Running”状态后，你一定要能迅速做出反应，根据它所代表的异常情况开始跟踪和定位，而不是去手忙脚乱地查阅文档。

总结

在今天这篇文章中，我详细讲解了 Pod API 对象，介绍了 Pod 的核心使用方法，并分析了 Pod 和 Container 在字段上的异同。希望这些讲解能够帮你更好地理解和记忆 Pod YAML 中的核心字段，以及这些字段的准确含义。

实际上，Pod API 对象是整个 Kubernetes 体系中最核心的一个概念，也是后面我讲解各种控制器时都要用到的。

在学习完这篇文章后，我希望你能仔细阅读 $GOPATH/src/k8s.io/kubernetes/vendor/k8s.io/api/core/v1/types.go 里，type Pod struct ，尤其是 PodSpec 部分的内容。争取做到下次看到一个 Pod 的 YAML 文件时，不再需要查阅文档，就能做到把常用字段及其作用信手拈来。

而在下一篇文章中，我会通过大量的实践，帮助你巩固和进阶关于 Pod API 对象核心字段的使用方法，敬请期待吧。

思考题

你能否举出一些 Pod（即容器）的状态是 Running，但是应用其实已经停止服务的例子？相信 Java Web 开发者的亲身体会会比较多吧。