12 | 牛刀小试：我的第一个容器化应用

12 | 牛刀小试：我的第一个容器化应用

张磊 2018-09-19

13:15

讲述：张磊 大小：6.07M

你好，我是张磊。今天我和你分享的主题是：牛刀小试之我的第一个容器化应用。

在上一篇文章《从 0 到 1：搭建一个完整的 Kubernetes 集群》中，我和你一起部署了一套完整的 Kubernetes 集群。这个集群虽然离生产环境的要求还有一定差距（比如，没有一键高可用部署），但也可以当作是一个准生产级别的 Kubernetes 集群了。

而在这篇文章中，我们就来扮演一个应用开发者的角色，使用这个 Kubernetes 集群发布第一个容器化应用。

在开始实践之前，我先给你讲解一下 Kubernetes 里面与开发者关系最密切的几个概念。

作为一个应用开发者，你首先要做的，是制作容器的镜像。这一部分内容，我已经在容器基础部分《白话容器基础（三）：深入理解容器镜像》重点讲解过了。

而有了容器镜像之后，你需要按照 Kubernetes 项目的规范和要求，将你的镜像组织为它能够“认识”的方式，然后提交上去。

那么，什么才是 Kubernetes 项目能“认识”的方式呢？

这就是使用 Kubernetes 的必备技能：编写配置文件。

> *备注：这些配置文件可以是 YAML 或者 JSON 格式的。为方便阅读与理解，在后面的讲解中，我会统一使用 YAML 文件来指代它们。*

Kubernetes 跟 Docker 等很多项目最大的不同，就在于它不推荐你使用命令行的方式直接运行容器（虽然 Kubernetes 项目也支持这种方式，比如：kubectl run），而是希望你用 YAML 文件的方式，即：把容器的定义、参数、配置，统统记录在一个 YAML 文件中，然后用这样一句指令把它运行起来：

```
$ kubectl create -f 我的配置文件
```

□复制代码

这么做最直接的好处是，你会有一个文件能记录下 Kubernetes 到底“run”了什么。比如下面这个例子：

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
spec:
selector:
matchLabels:
app: nginx
replicas: 2
template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
```

```
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

□复制代码

像这样的一个 YAML 文件，对应到 Kubernetes 中，就是一个 API Object（API 对象）。当你为这个对象的各个字段填好值并提交给 Kubernetes 之后，Kubernetes 就会负责创建出这些对象所定义的容器或者其他类型的 API 资源。

可以看到，这个 YAML 文件中的 Kind 字段，指定了这个 API 对象的类型（Type），是一个 Deployment。

所谓 Deployment，是一个定义多副本应用（即多个副本 Pod）的对象，我在前面的文章中（也是第 9 篇文章《从容器到容器云：谈谈 Kubernetes 的本质》）曾经简单提到过它的用法。此外，Deployment 还负责在 Pod 定义发生变化时，对每个副本进行滚动更新（Rolling Update）。

在上面这个 YAML 文件中，我给它定义的 Pod 副本个数 (spec.replicas) 是：2。

而这些 Pod 具体的又长什么样子呢?

为此，我定义了一个 Pod 模版（spec.template），这个模版描述了我想要创建的 Pod 的细节。在上面的例子里，这个 Pod 里只有一个容器，这个容器的镜像（spec.containers.image）是 nginx:1.7.9，这个容器监听端口（containerPort）是 80。

关于 Pod 的设计和用法我已经在第 9 篇文章《从容器到容器云：谈谈 Kubernetes 的本质》中简单的介绍过。而在这里，你需要记住这样一句话：

> *Pod 就是 Kubernetes 世界里的“应用”；而一个应用，可以由多个容器组成。*

需要注意的是，像这样使用一种 API 对象（Deployment）管理另一种 API 对象（Pod）的方法，在 Kubernetes 中，叫作“控制器”模式（controller pattern）。在我们的例子中，Deployment 扮演的正是 Pod 的控制器的角色。关于 Pod 和控制器模式的更多细节，我会在后续编排部分做进一步讲解。

你可能还注意到，这样的每一个 API 对象都有一个叫作 Metadata 的字段，这个字段就是 API 对象的“标识”，即元数据，它也是我们从 Kubernetes 里找到这个对象的主要依据。这其中最主要使用到的字段是 Labels。

顾名思义，Labels 就是一组 key-value 格式的标签。而像 Deployment 这样的控制器对象，就可以通过这个 Labels 字段从 Kubernetes 中过滤出它所关心的被控制对象。

比如，在上面这个 YAML 文件中，Deployment 会把所有正在运行的、携带“app: nginx”标签的 Pod 识别为被管理的对象，并确保这些 Pod 的总数严格等于两个。

而这个过滤规则的定义，是在 Deployment 的“spec.selector.matchLabels”字段。我们一般称之为：Label Selector。

另外，在 Metadata 中，还有一个与 Labels 格式、层级完全相同的字段叫 Annotations，它专门用来携带 key-value 格式的内部信息。所谓内部信息，指的是对这些信息感兴趣的，是 Kubernetes 组件本身，而不是用户。所以大多数 Annotations，都是在 Kubernetes 运行过程中，被自动加在这个 API 对象上。

一个 Kubernetes 的 API 对象的定义，大多可以分为 Metadata 和 Spec 两个部分。前者存放的是这个对象的元数据，对所有 API 对象来说，这一部分的字段和格式基本上是一样的；而后者存放的，则是属于这个对象独有的定义，用来描述它所要表达的功能。

在了解了上述 Kubernetes 配置文件的基本知识之后，我们现在就可以把这个 YAML 文件“运行”起来。正如前所述，你可以使用 kubectl create 指令完成这个操作：

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

□复制代码

然后，通过 kubectl get 命令检查这个 YAML 运行起来的状态是不是与我们预期的一致：

```
$ kubectl get pods -l app=nginx
NAME READY STATUS RESTARTS AGE
nginx-deployment-67594d6bf6-9gdvr 1/1 Running 0 10m
nginx-deployment-67594d6bf6-v6j7w 1/1 Running 0 10m
```

□复制代码

kubectl get 指令的作用，就是从 Kubernetes 里面获取（GET）指定的 API 对象。可以看到，在这里我还加上了一个 -l 参数，即获取所有匹配 app: nginx 标签的 Pod。需要注意的是，**在命令行中，所有 key-value 格式的参数，都使用“=”而非“:”表示。**

从这条指令返回的结果中，我们可以看到现在有两个 Pod 处于 Running 状态，也就意味着我们这个 Deployment 所管理的 Pod 都处于预期的状态。

此外， 你还可以使用 kubectl describe 命令，查看一个 API 对象的细节，比如：

```
$ kubectl describe pod nginx-deployment-67594d6bf6-9gdvr
```

```
Name: nginx-deployment-67594d6bf6-9gdvr
Namespace: default
Priority: 0
PriorityClassName: <none>
Node: node-1/10.168.0.3
Start Time: Thu, 16 Aug 2018 08:48:42 +0000
Labels: app=nginx
pod-template-hash=2315082692
Annotations: <none>
Status: Running
IP: 10.32.0.23
Controlled By: ReplicaSet/nginx-deployment-67594d6bf6
...
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Scheduled 1m default-scheduler Successfully assigned
default/nginx-deployment-67594d6bf6-9gdvr to node-1
Normal Pulling 25s kubelet, node-1 pulling image "nginx:1.7.9"
Normal Pulled 17s kubelet, node-1 Successfully pulled image
"nginx:1.7.9"
Normal Created 17s kubelet, node-1 Created container
Normal Started 17s kubelet, node-1 Started container
```

□复制代码

在 kubectl describe 命令返回的结果中，你可以清楚地看到这个 Pod 的详细信息，比如它的 IP 地址等等。其中，有一个部分值得你特别关注，它就是 **Events（事件）。**

在 Kubernetes 执行的过程中，对 API 对象的所有重要操作，都会被记录在这个对象的 Events 里，并且显示在 kubectl describe 指令返回的结果中。

比如，对于这个 Pod，我们可以看到它被创建之后，被调度器调度（Successfully assigned）到了 node-1，拉取了指定的镜像（pulling image），然后启动了 Pod 里定义的容器（Started container）。

所以，这个部分正是我们将来进行 Debug 的重要依据。**如果有异常发生，你一定要第一时间查看这些 Events**，往往可以看到非常详细的错误信息。

接下来，如果我们要对这个 Nginx 服务进行升级，把它的镜像版本从 1.7.9 升级为 1.8，要怎么做呢?

很简单，我们只要修改这个 YAML 文件即可。

```
...
spec:
containers:
- name: nginx
image: nginx:1.8 # 这里被从 1.7.9 修改为 1.8
ports:
- containerPort: 80
```

复制代码

可是，这个修改目前只发生在本地，如何让这个更新在 Kubernetes 里也生效呢?

我们可以使用 kubectl replace 指令来完成这个更新：

```
$ kubectl replace -f nginx-deployment.yaml
```

复制代码

不过，在本专栏里，我推荐你使用 kubectl apply 命令，来统一进行 Kubernetes 对象的创建和更新操作，具体做法如下所示：

```
$ kubectl apply -f nginx-deployment.yaml
# 修改 nginx-deployment.yaml 的内容
$ kubectl apply -f nginx-deployment.yaml
```

复制代码

这样的操作方法，是 Kubernetes“声明式 API”所推荐的使用方法。也就是说，作为用户，你不必关心当前的操作是创建，还是更新，你执行的命令始终是 kubectl apply，而 Kubernetes 则会根据 YAML 文件的内容变化，自动进行具体的处理。

而这个流程的好处是，它有助于帮助开发和运维人员，围绕着可以版本化管理的 YAML 文件，而不是“行踪不定”的命令行进行协作，从而大大降低开发人员和运维人员之间的沟通成本。

举个例子，一位开发人员开发好一个应用，制作好了容器镜像。那么他就可以在应用的发布目录里附带上一个 Deployment 的 YAML 文件。

而运维人员，拿到这个应用的发布目录后，就可以直接用这个 YAML 文件执行 kubectl apply 操作把它运行起来。

这时候，如果开发人员修改了应用，生成了新的发布内容，那么这个 YAML 文件，也就需要被修改，并且成为这次变更的一部分。

而接下来，运维人员可以使用 git diff 命令查看到这个 YAML 文件本身的变化，然后继续用 kubectl apply 命令更新这个应用。

所以说，如果通过容器镜像，我们能够保证应用本身在开发与部署环境里的一致性的话，那么现在，Kubernetes 项目通过这些 YAML 文件，就保证了应用的“部署参数”在开发与部署环境中的一致性。

而当应用本身发生变化时，开发人员和运维人员可以依靠容器镜像来进行同步；当应用部署参数发生变化时，这些 YAML 文件就是他们相互沟通和信任的媒介。

以上，就是 Kubernetes 发布应用的最基本操作了。

接下来，我们再在这个 Deployment 中尝试声明一个 Volume。

在 Kubernetes 中，Volume 是属于 Pod 对象的一部分。所以，我们就需要修改这个 YAML 文件里的 template.spec 字段，如下所示：

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
spec:
selector:
matchLabels:
app: nginx
replicas: 2
template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
image: nginx:1.8
ports:
```

```
- containerPort: 80
volumeMounts:
- mountPath: "/usr/share/nginx/html"
name: nginx-vol
volumes:
- name: nginx-vol
emptyDir: {}
```

□复制代码

可以看到，我们在 Deployment 的 Pod 模板部分添加了一个 volumes 字段，定义了这个 Pod 声明的所有 Volume。它的名字叫作 nginx-vol，类型是 emptyDir。

那什么是 emptyDir 类型呢?

它其实就等同于我们之前讲过的 Docker 的隐式 Volume 参数，即：不显式声明宿主机目录的 Volume。所以，Kubernetes 也会在宿主机上创建一个临时目录，这个目录将来就会被绑定挂载到容器所声明的 Volume 目录上。

> *备注：不难看到，Kubernetes 的 emptyDir 类型，只是把 Kubernetes 创建的临时目录作为 Volume 的宿主机目录，交给了 Docker。这么做的原因，是 Kubernetes 不想依赖 Docker 自己创建的那个 _data 目录。*

而 Pod 中的容器，使用的是 volumeMounts 字段来声明自己要挂载哪个 Volume，并通过 mountPath 字段来定义容器内的 Volume 目录，比如：/usr/share/nginx/html。

当然，Kubernetes 也提供了显式的 Volume 定义，它叫做 hostPath。比如下面的这个 YAML 文件：

```
...
volumes:
- name: nginx-vol
hostPath:
path: /var/data
```

□复制代码

这样，容器 Volume 挂载的宿主机目录，就变成了 /var/data。

在上述修改完成后，我们还是使用 kubectl apply 指令，更新这个 Deployment:

```
$ kubectl apply -f nginx-deployment.yaml
```

□复制代码

接下来，你可以通过 kubectl get 指令，查看两个 Pod 被逐一更新的过程：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-deployment-5c678cfb6d-v5dlh 0/1 ContainerCreating 0 4s
nginx-deployment-67594d6bf6-9gdvr 1/1 Running 0 10m
nginx-deployment-67594d6bf6-v6j7w 1/1 Running 0 10m
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-deployment-5c678cfb6d-lg9lw 1/1 Running 0 8s
nginx-deployment-5c678cfb6d-v5dlh 1/1 Running 0 19s
```

□复制代码

从返回结果中，我们可以看到，新旧两个 Pod，被交替创建、删除，最后剩下的就是新版本的 Pod。这个滚动更新的过程，我也会在后续进行详细的讲解。

然后，你可以使用 kubectl describe 查看一下最新的 Pod，就会发现 Volume 的信息已经出现在了 Container 描述部分：

```
...
Containers:
nginx:
Container ID: docker://07b4f89248791c2aa47787e3da3cc94b48576cd173018356a6ec8db2b6041343
Image: nginx:1.8
...
Environment: <none>
Mounts:
/usr/share/nginx/html from nginx-vol (rw)
...
Volumes:
nginx-vol:
Type: EmptyDir (a temporary directory that shares a pod's lifetime)
```

□复制代码

> *备注：作为一个完整的容器化平台项目，Kubernetes 为我们提供的 Volume 类型远远不止这些，在容器存储章节里，我将会为你详细介绍这部分内容。*

最后，你还可以使用 kubectl exec 指令，进入到这个 Pod 当中（即容器的 Namespace 中）查看这个 Volume 目录：

```
$ kubectl exec -it nginx-deployment-5c678cfb6d-lg9lw -- /bin/bash
# ls /usr/share/nginx/html
```

□复制代码

此外，你想要从 Kubernetes 集群中删除这个 Nginx Deployment 的话，直接执行：

```
$ kubectl delete -f nginx-deployment.yaml
```

□复制代码

就可以了。

总结

在今天的分享中，我通过一个小案例，和你近距离体验了 Kubernetes 的使用方法。

可以看到，Kubernetes 推荐的使用方式，是用一个 YAML 文件来描述你所要部署的 API 对象。然后，统一使用 kubectl apply 命令完成对这个对象的创建和更新操作。

而 Kubernetes 里“最小”的 API 对象是 Pod。Pod 可以等价为一个应用，所以，Pod 可以由多个紧密协作的容器组成。

在 Kubernetes 中，我们经常会看到它通过一种 API 对象来管理另一种 API 对象，比如 Deployment 和 Pod 之间的关系；而由于 Pod 是“最小”的对象，所以它往往都是被其他对象控制的。这种组合方式，正是 Kubernetes 进行容器编排的重要模式。

而像这样的 Kubernetes API 对象，往往由 Metadata 和 Spec 两部分组成，其中 Metadata 里的 Labels 字段是 Kubernetes 过滤对象的主要手段。

在这些字段里面，容器想要使用的数据卷，也就是 Volume，正是 Pod 的 Spec 字段的一部分。而 Pod 里的每个容器，则需要显式的声明自己要挂载哪个 Volume。

上面这些基于 YAML 文件的容器管理方式，跟 Docker、Mesos 的使用习惯都是不一样的，而从 docker run 这样的命令行操作，向 kubectl apply YAML 文件这样的声明式 API 的转变，是每一个容器技术学习者，必须要跨过的第一道门槛。

所以，如果你想要快速熟悉 Kubernetes，请按照下面的流程进行练习：

首先，在本地通过 Docker 测试代码，制作镜像；
然后，选择合适的 Kubernetes API 对象，编写对应 YAML 文件（比如，Pod，Deployment）；
最后，在 Kubernetes 上部署这个 YAML 文件。

更重要的是，在部署到 Kubernetes 之后，接下来的所有操作，要么通过 kubectl 来执行，要么通过修改 YAML 文件来实现，**就尽量不要再碰 Docker 的命令行了**。

思考题

在实际使用 Kubernetes 的过程中，相比于编写一个单独的 Pod 的 YAML 文件，我一定会推荐你使用一个 replicas=1 的 Deployment。请问，这两者有什么区别呢？