19 | 深入理解StatefulSet（二）：存储状态

19 | 深入理解StatefulSet（二）：存储状态

张磊 2018-10-05

12:51

讲述：张磊 大小：5.89M

你好，我是张磊。今天我和你分享的主题是：深入理解 StatefulSet 之存储状态。

在上一篇文章中，我和你分享了 StatefulSet 如何保证应用实例的拓扑状态，在 Pod 删除和再创建的过程中保持稳定。

而在今天这篇文章中，我将继续为你解读 StatefulSet 对存储状态的管理机制。这个机制，主要使用的是一个叫作 Persistent Volume Claim 的功能。

在前面介绍 Pod 的时候，我曾提到过，要在一个 Pod 里声明 Volume，只要在 Pod 里加上 spec.volumes 字段即可。然后，你就可以在这个字段里定义一个具体类型的 Volume 了，比如：hostPath。

可是，你有没有想过这样一个场景：**如果你并不知道有哪些 Volume 类型可以用，要怎么办呢**？

更具体地说，作为一个应用开发者，我可能对持久化存储项目（比如 Ceph、GlusterFS 等）一窍不通，也不知道公司的 Kubernetes 集群里到底是怎么搭建出来的，我也自然不会编写它们对应的 Volume 定义文件。

所谓“术业有专攻”，这些关于 Volume 的管理和远程持久化存储的知识，不仅超越了开发者的知识储备，还会有暴露公司基础设施秘密的风险。

比如，下面这个例子，就是一个声明了 Ceph RBD 类型 Volume 的 Pod：

```
apiVersion: v1
kind: Pod
metadata:
name: rbd
spec:
containers:
- image: kubernetes/pause
name: rbd-rw
volumeMounts:
- name: rbdpd
mountPath: /mnt/rbd
volumes:
- name: rbdpd
rbd:
monitors:
- '10.16.154.78:6789'
- '10.16.154.82:6789'
- '10.16.154.83:6789'
pool: kube
image: foo
fsType: ext4
readOnly: true
user: admin
```

```
    keyring: /etc/ceph/keyring
    imageformat: "2"
    imagefeatures: "layering"
```

复制代码

其一，如果不懂得 Ceph RBD 的使用方法，那么这个 Pod 里 Volumes 字段，你十有八九也完全看不懂。其二，这个 Ceph RBD 对应的存储服务器的地址、用户名、授权文件的位置，也都被轻易地暴露给了全公司的所有开发人员，这是一个典型的信息被“过度暴露”的例子。

这也是为什么，在后来的演化中，**Kubernetes 项目引入了一组叫作 Persistent Volume Claim（PVC）和 Persistent Volume（PV）的 API 对象，大大降低了用户声明和使用持久化 Volume 的门槛。**

举个例子，有了 PVC 之后，一个开发人员想要使用一个 Volume，只需要简单的两步即可。

第一步：定义一个 PVC，声明想要的 Volume 的属性：

```
    kind: PersistentVolumeClaim
    apiVersion: v1
    metadata:
    name: pv-claim
    spec:
    accessModes:
    - ReadWriteOnce
    resources:
    requests:
    storage: 1Gi
```

复制代码

可以看到，在这个 PVC 对象里，不需要任何关于 Volume 细节的字段，只有描述性的属性和定义。比如，storage: 1Gi，表示我想要的 Volume 大小至少是 1 GiB；accessModes: ReadWriteOnce，表示这个 Volume 的挂载方式是可读写，并且只能被挂载在一个节点上而非被多个节点共享。

> *备注：关于哪种类型的 Volume 支持哪种类型的 AccessMode，你可以查看 Kubernetes 项目官方文档中的详细列表。*

第二步：在应用的 Pod 中，声明使用这个 PVC：

```
apiVersion: v1
kind: Pod
metadata:
name: pv-pod
spec:
containers:
- name: pv-container
image: nginx
ports:
- containerPort: 80
name: "http-server"
volumeMounts:
- mountPath: "/usr/share/nginx/html"
name: pv-storage
volumes:
- name: pv-storage
persistentVolumeClaim:
claimName: pv-claim
```

复制代码

可以看到，在这个 Pod 的 Volumes 定义中，我们只需要声明它的类型是 persistentVolumeClaim，然后指定 PVC 的名字，而完全不必关心 Volume 本身的定义。

这时候，只要我们创建这个 PVC 对象，Kubernetes 就会自动为它绑定一个符合条件的 Volume。可是，这些符合条件的 Volume 又是从哪里来的呢?

答案是，它们来自于由运维人员维护的 PV（Persistent Volume）对象。接下来，我们一起看一个常见的 PV 对象的 YAML 文件：

```
kind: PersistentVolume
apiVersion: v1
metadata:
name: pv-volume
labels:
type: local
```

```
  spec:
  capacity:
  storage: 10Gi
  rbd:
  monitors:
  - '10.16.154.78:6789'
  - '10.16.154.82:6789'
  - '10.16.154.83:6789'
  pool: kube
  image: foo
  fsType: ext4
  readOnly: true
  user: admin
  keyring: /etc/ceph/keyring
  imageformat: "2"
  imagefeatures: "layering"
```

复制代码

可以看到，这个 PV 对象的 spec.rbd 字段，正是我们前面介绍过的 Ceph RBD Volume 的详细定义。而且，它还声明了这个 PV 的容量是 10 GiB。这样，Kubernetes 就会为我们刚刚创建的 PVC 对象绑定这个 PV。

所以，Kubernetes 中 PVC 和 PV 的设计，**实际上类似于“接口”和“实现”的思想**。开发者只要知道并会使用“接口”，即：PVC；而运维人员则负责给“接口”绑定具体的实现，即：PV。

这种解耦，就避免了因为向开发者暴露过多的存储系统细节而带来的隐患。此外，这种职责的分离，往往也意味着出现事故时可以更容易定位问题和明确责任，从而避免“扯皮”现象的出现。

而 PVC、PV 的设计，也使得 StatefulSet 对存储状态的管理成为了可能。我们还是以上一篇文章中用到的 StatefulSet 为例（你也可以借此再回顾一下《深入理解 StatefulSet（一）：拓扑状态》中的相关内容）：

```
  apiVersion: apps/v1
  kind: StatefulSet
  metadata:
  name: web
```

```
spec:
serviceName: "nginx"
replicas: 2
selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx
spec:
containers:
- name: nginx
image: nginx:1.9.1
ports:
- containerPort: 80
name: web
volumeMounts:
- name: www
mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
name: www
spec:
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 1Gi
```

□复制代码

这次，我们为这个 StatefulSet 额外添加了一个 volumeClaimTemplates 字段。从名字就可以看出来，它跟 Deployment 里 Pod 模板（PodTemplate）的作用类似。也就是说，凡是被这个 StatefulSet 管理的 Pod，都会声明一个对应的

PVC；而这个 PVC 的定义，就来自于 volumeClaimTemplates 这个模板字段。更重要的是，这个 PVC 的名字，会被分配一个与这个 Pod 完全一致的编号。

这个自动创建的 PVC，与 PV 绑定成功后，就会进入 Bound 状态，这就意味着这个 Pod 可以挂载并使用这个 PV 了。

如果你还是不太理解 PVC 的话，可以先记住这样一个结论：**PVC 其实就是一种特殊的 Volume**。只不过一个 PVC 具体是什么类型的 Volume，要在跟某个 PV 绑定之后才知道。关于 PV、PVC 更详细的知识，我会在容器存储部分做进一步解读。

当然，PVC 与 PV 的绑定得以实现的前提是，运维人员已经在系统里创建好了符合条件的 PV（比如，我们在前面用到的 pv-volume）；或者，你的 Kubernetes 集群运行在公有云上，这样 Kubernetes 就会通过 Dynamic Provisioning 的方式，自动为你创建与 PVC 匹配的 PV。

所以，我们在使用 kubectl create 创建了 StatefulSet 之后，就会看到 Kubernetes 集群里出现了两个 PVC：

```
$ kubectl create -f statefulset.yaml
$ kubectl get pvc -l app=nginx
NAME STATUS VOLUME CAPACITY ACCESSMODES AGE
www-web-0 Bound pvc-15c268c7-b507-11e6-932f-42010a800002 1Gi RWO 48s
www-web-1 Bound pvc-15c79307-b507-11e6-932f-42010a800002 1Gi RWO 48s
```

□复制代码

可以看到，这些 PVC，都以“<PVC 名字 >-<StatefulSet 名字 >-< 编号 >”的方式命名，并且处于 Bound 状态。

我们前面已经讲到过，这个 StatefulSet 创建出来的所有 Pod，都会声明使用编号的 PVC。比如，在名叫 web-0 的 Pod 的 volumes 字段，它会声明使用名叫 www-web-0 的 PVC，从而挂载到这个 PVC 所绑定的 PV。

所以，我们就可以使用如下所示的指令，在 Pod 的 Volume 目录里写入一个文件，来验证一下上述 Volume 的分配情况：

```
$ for i in 0 1; do kubectl exec web-$i -- sh -c 'echo hello $(hostname) > /usr/share/nginx/html/index.html'; done
```

□复制代码

如上所示，通过 kubectl exec 指令，我们在每个 Pod 的 Volume 目录里，写入了一个 index.html 文件。这个文件的内容，正是 Pod 的 hostname。比如，我们在 web-0 的 index.html 里写入的内容就是 "hello web-0"。

此时，如果你在这个 Pod 容器里访问“http://localhost”，你实际访问到的就是 Pod 里 Nginx 服务器进程，而它会为你返回 /usr/share/nginx/html/index.html 里的内容。这个操作的执行方法如下所示：

```
$ for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
hello web-0
hello web-1
```

□复制代码

现在，关键来了。

如果你使用 kubectl delete 命令删除这两个 Pod，这些 Volume 里的文件会不会丢失呢？

```
$ kubectl delete pod -l app=nginx
pod "web-0" deleted
pod "web-1" deleted
```

□复制代码

可以看到，正如我们前面介绍过的，在被删除之后，这两个 Pod 会被按照编号的顺序被重新创建出来。而这时候，如果你在新创建的容器里通过访问“http://localhost”的方式去访问 web-0 里的 Nginx 服务：

```
# 在被重新创建出来的 Pod 容器里访问 http://localhost
$ kubectl exec -it web-0 -- curl localhost
hello web-0
```

□复制代码

就会发现，这个请求依然会返回：hello web-0。也就是说，原先与名叫 web-0 的 Pod 绑定的 PV，在这个 Pod 被重新创建之后，依然同新的名叫 web-0 的 Pod 绑定在了一起。对于 Pod web-1 来说，也是完全一样的情况。

这是怎么做到的呢？

其实，我和你分析一下 StatefulSet 控制器恢复这个 Pod 的过程，你就可以很容易理解了。

首先，当你把一个 Pod，比如 web-0，删除之后，这个 Pod 对应的 PVC 和 PV，并不会被删除，而这个 Volume 里已经写入的数据，也依然会保存在远程存

储服务里（比如，我们在这个例子里用到的 Ceph 服务器）。

此时，StatefulSet 控制器发现，一个名叫 web-0 的 Pod 消失了。所以，控制器就会重新创建一个新的、名字还是叫作 web-0 的 Pod 来，“纠正”这个不一致的情况。

需要注意的是，在这个新的 Pod 对象的定义里，它声明使用的 PVC 的名字，还是叫作：www-web-0。这个 PVC 的定义，还是来自于 PVC 模板（volumeClaimTemplates），这是 StatefulSet 创建 Pod 的标准流程。

所以，在这个新的 web-0 Pod 被创建出来之后，Kubernetes 为它查找名叫 www-web-0 的 PVC 时，就会直接找到旧 Pod 遗留下来的同名的 PVC，进而找到跟这个 PVC 绑定在一起的 PV。

这样，新的 Pod 就可以挂载到旧 Pod 对应的那个 Volume，并且获取到保存在 Volume 里的数据。

通过这种方式，Kubernetes 的 StatefulSet 就实现了对应用存储状态的管理。

看到这里，你是不是已经大致理解了 StatefulSet 的工作原理呢？现在，我再为你详细梳理一下吧。

首先，StatefulSet 的控制器直接管理的是 Pod。这是因为，StatefulSet 里的不同 Pod 实例，不再像 ReplicaSet 中那样都是完全一样的，而是有了细微区别的。比如，每个 Pod 的 hostname、名字等都是不同的、携带了编号的。而 StatefulSet 区分这些实例的方式，就是通过在 Pod 的名字里加上事先约定好的编号。

其次，Kubernetes 通过 Headless Service，为这些有编号的 Pod，在 DNS 服务器中生成带有同样编号的 DNS 记录。只要 StatefulSet 能够保证这些 Pod 名字里的编号不变，那么 Service 里类似于 web-0.nginx.default.svc.cluster.local 这样的 DNS 记录也就不会变，而这条记录解析出来的 Pod 的 IP 地址，则会随着后端 Pod 的删除和再创建而自动更新。这当然是 Service 机制本身的能力，不需要 StatefulSet 操心。

最后，StatefulSet 还为每一个 Pod 分配并创建一个同样编号的 PVC。这样，Kubernetes 就可以通过 Persistent Volume 机制为这个 PVC 绑定上对应的 PV，从而保证了每一个 Pod 都拥有一个独立的 Volume。

在这种情况下，即使 Pod 被删除，它所对应的 PVC 和 PV 依然会保留下来。所以当这个 Pod 被重新创建出来之后，Kubernetes 会为它找到同样编号的 PVC，挂载这个 PVC 对应的 Volume，从而获取到以前保存在 Volume 里的数据。

这么一看，原本非常复杂的 StatefulSet，是不是也很容易理解了呢？

总结

在今天这篇文章中，我为你详细介绍了 StatefulSet 处理存储状态的方法。然后，以此为基础，我为你梳理了 StatefulSet 控制器的工作原理。

从这些讲述中，我们不难看出 StatefulSet 的设计思想：StatefulSet 其实就是一种特殊的 Deployment，而其独特之处在于，它的每个 Pod 都被编号了。而且，这个编号会体现在 Pod 的名字和 hostname 等标识信息上，这不仅代表了 Pod 的创建顺序，也是 Pod 的重要网络标识（即：在整个集群里唯一的、可被的访问身份）。

有了这个编号后，StatefulSet 就使用 Kubernetes 里的两个标准功能：Headless Service 和 PV/PVC，实现了对 Pod 的拓扑状态和存储状态的维护。

实际上，在下一篇文章的“有状态应用”实践环节，以及后续的讲解中，你就会逐渐意识到，StatefulSet 可以说是 Kubernetes 中作业编排的“集大成者”。

因为，几乎每一种 Kubernetes 的编排功能，都可以在编写 StatefulSet 的 YAML 文件时被用到。

思考题

在实际场景中，有一些分布式应用的集群是这么工作的：当一个新节点加入到集群时，或者老节点被迁移后重建时，这个节点可以从主节点或者其他从节点那里同步到自己所需要的数据。

在这种情况下，你认为是否还有必要将这个节点 Pod 与它的 PV 进行一对一绑定呢？（提示：这个问题的答案根据不同的项目是不同的。关键在于，重建后的节点进行数据恢复和同步的时候，是不是一定需要原先它写在本地磁盘里的数据）