

15 | 深入解析Pod对象（二）：使用进阶

2018-09-26 张磊

深入剖析Kubernetes

[进入课程 >](#)



讲述：张磊

时长 21:47 大小 8.73M



你好，我是张磊。今天我和你分享的主题是：深入解析 Pod 对象之使用进阶。

在上一篇文章中，我深入解析了 Pod 的 API 对象，讲解了 Pod 和 Container 的关系。

作为 Kubernetes 项目里最核心的编排对象，Pod 携带的信息非常丰富。其中，资源定义（比如 CPU、内存等），以及调度相关的字段，我会在后面专门讲解调度器时再进行深入的分析。在本篇，我们就先从一种特殊的 Volume 开始，来帮助你更加深入地理解 Pod 对象各个重要字段的含义。

这种特殊的 Volume，叫作 Projected Volume，你可以把它翻译为“投射数据卷”。

备注：Projected Volume 是 Kubernetes v1.11 之后的新特性

这是什么意思呢？


在 Kubernetes 中，有几种特殊的 Volume，它们存在的意义不是为了存放容器里的数据，也不是用来进行容器和宿主机之间的数据交换。这些特殊 Volume 的作用，是为容器提供预先定义好的数据。所以，从容器的角度来看，这些 Volume 里的信息就是仿佛是被 **Kubernetes “投射”（Project）进入容器当中的**。这正是 Projected Volume 的含义。

到目前为止，Kubernetes 支持的 Projected Volume 一共有四种：

1. Secret；
2. ConfigMap；
3. Downward API；
4. ServiceAccountToken。

在今天这篇文章中，**我首先和你分享的是 Secret**。它的作用，是帮你把 Pod 想要访问的加密数据，存放到 Etcd 中。然后，你就可以通过在 Pod 的容器里挂载 Volume 的方式，访问到这些 Secret 里保存的信息了。

Secret 最典型的使用场景，莫过于存放数据库的 Credential 信息，比如下面这个例子：


 复制代码

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: test-projected-volume
5 spec:
6   containers:
7     - name: test-secret-volume
8       image: busybox
9       args:
10        - sleep
11        - "86400"
12       volumeMounts:
13        - name: mysql-cred
14          mountPath: "/projected-volume"
15          readOnly: true
16   volumes:
17     - name: mysql-cred
18       projected:
19         sources:
20           - secret:
21             name: user
```

```
22     - secret:
23       name: pass
```


在这个 Pod 中，我定义了一个简单的容器。它声明挂载的 Volume，并不是常见的 emptyDir 或者 hostPath 类型，而是 projected 类型。而这个 Volume 的数据来源（sources），则是名为 user 和 pass 的 Secret 对象，分别对应的是数据库的用户名和密码。

这里用到的数据库的用户名、密码，正是以 Secret 对象的方式交给 Kubernetes 保存的。完成这个操作的指令，如下所示：

 复制代码

```
1 $ cat ./username.txt
2 admin
3 $ cat ./password.txt
4 c1oudc0w!
5
6 $ kubectl create secret generic user --from-file=./username.txt
7 $ kubectl create secret generic pass --from-file=./password.txt
```

其中，username.txt 和 password.txt 文件里，存放的就是用户名和密码；而 user 和 pass，则是我为 Secret 对象指定的名字。而我想要查看这些 Secret 对象的话，只要执行一条 kubectl get 命令就可以了：

 复制代码

```
1 $ kubectl get secrets
2 NAME          TYPE          DATA   AGE
3 user          Opaque        1       51s
4 pass          Opaque        1       51s
```


当然，除了使用 kubectl create secret 指令外，我也可以直接通过编写 YAML 文件的方式来创建这个 Secret 对象，比如：

 复制代码

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: mysecret
5 type: Opaque
6 data:
7   user: YWRtaW4=
8   pass: MWYyZDFlMmU2N2Rm
```

可以看到，通过编写 YAML 文件创建出来的 Secret 对象只有一个。但它的 data 字段，却以 Key-Value 的格式保存了两份 Secret 数据。其中，“user”就是第一份数据的 Key，“pass”是第二份数据的 Key。


需要注意的是，Secret 对象要求这些数据必须是经过 Base64 转码的，以免出现明文密码的安全隐患。这个转码操作也很简单，比如：

 复制代码

```
1 $ echo -n 'admin' | base64
2 YWRtaW4=
3 $ echo -n '1f2d1e2e67df' | base64
4 MWYyZDFlMmU2N2Rm
```

这里需要注意的是，像这样创建的 Secret 对象，它里面的内容仅仅是经过了转码，而并没有被加密。在真正的生产环境中，你需要在 Kubernetes 中开启 Secret 的加密插件，增强数据的安全性。关于开启 Secret 加密插件的内容，我会在后续专门讲解 Secret 的时候，再做进一步说明。

接下来，我们尝试一下创建这个 Pod：

 复制代码

```
1 $ kubectl create -f test-projected-volume.yaml
```

当 Pod 变成 Running 状态之后，我们再验证一下这些 Secret 对象是不是已经在容器里了：

```
1 $ kubectl exec -it test-projected-volume -- /bin/sh
2 $ ls /projected-volume/
3 user
4 pass
5 $ cat /projected-volume/user
6 root
7 $ cat /projected-volume/pass
8 1f2d1e2e67df
```

从返回结果中，我们可以看到，保存在 Etcd 里的用户名和密码信息，已经以文件的形式出现在了容器的 Volume 目录里。而这个文件的名称，就是 `kubectl create secret` 指定的 Key，或者说是 Secret 对象的 data 字段指定的 Key。

更重要的是，像这样通过挂载方式进入到容器里的 Secret，一旦其对应的 Etcd 里的数据被更新，这些 Volume 里的文件内容，同样也会被更新。其实，**这是 kubelet 组件在定时维护这些 Volume。**

需要注意的是，这个更新可能会有一定的延时。所以在编写应用程序时，在发起数据库连接的代码处写好重试和超时的逻辑，绝对是个好习惯。

与 Secret 类似的是 ConfigMap，它与 Secret 的区别在于，ConfigMap 保存的是不需要加密的、应用所需的配置信息。而 ConfigMap 的用法几乎与 Secret 完全相同：你可以使用 `kubectl create configmap` 从文件或者目录创建 ConfigMap，也可以直接编写 ConfigMap 对象的 YAML 文件。

比如，一个 Java 应用所需的配置文件（.properties 文件），就可以通过下面这样的方式保存在 ConfigMap 里：


```
1 # .properties 文件的内容
2 $ cat example/ui.properties
3 color.good=purple
4 color.bad=yellow
5 allow.textmode=true
6 how.nice.to.look=fairlyNice
7
8 # 从.properties 文件创建 ConfigMap
9 $ kubectl create configmap ui-config --from-file=example/ui.properties
```

```
10
11 # 查看这个 ConfigMap 里保存的信息 (data)
12 $ kubectl get configmaps ui-config -o yaml
13 apiVersion: v1
14 data:
15   ui.properties: |
16     color.good=purple
17     color.bad=yellow
18     allow.textmode=true
19     how.nice.to.look=fairlyNice
20 kind: ConfigMap
21 metadata:
22   name: ui-config
23   ...
```

备注：kubectl get -o yaml 这样的参数，会将指定的 Pod API 对象以 YAML 的方式展示出来。

接下来是 Downward API，它的作用是：让 Pod 里的容器能够直接获取到这个 Pod API 对象本身的信息。

举个例子：

 复制代码

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: test-downwardapi-volume
5   labels:
6     zone: us-est-coast
7     cluster: test-cluster1
8     rack: rack-22
9 spec:
10  containers:
11    - name: client-container
12      image: k8s.gcr.io/busybox
13      command: ["sh", "-c"]
14      args:
15        - while true; do
16          if [[ -e /etc/podinfo/labels ]]; then
17            echo -en '\n\n'; cat /etc/podinfo/labels; fi;
18            sleep 5;
19          done;
20      volumeMounts:
```

```
21     - name: podinfo
22       mountPath: /etc/podinfo
23       readOnly: false
24   volumes:
25     - name: podinfo
26       projected:
27         sources:
28         - downwardAPI:
29             items:
30             - path: "labels"
31               fieldRef:
32                 fieldPath: metadata.labels
```

在这个 Pod 的 YAML 文件中，我定义了一个简单的容器，声明了一个 projected 类型的 Volume。只不过这次 Volume 的数据来源，变成了 Downward API。而这个 Downward API Volume，则声明了要暴露 Pod 的 metadata.labels 信息给容器。


通过这样的声明方式，当前 Pod 的 Labels 字段的值，就会被 Kubernetes 自动挂载成为容器里的 /etc/podinfo/labels 文件。

而这个容器的启动命令，则是不断打印出 /etc/podinfo/labels 里的内容。所以，当我创建了这个 Pod 之后，就可以通过 kubectl logs 指令，查看到这些 Labels 字段被打印出来，如下所示：

 复制代码

```
1 $ kubectl create -f dapi-volume.yaml
2 $ kubectl logs test-downwardapi-volume
3 cluster="test-cluster1"
4 rack="rack-22"
5 zone="us-est-coast"
```

目前，Downward API 支持的字段已经非常丰富了，比如：

 复制代码

```
1 1. 使用 fieldRef 可以声明使用：
2 spec.nodeName - 宿主机名字
3 status.hostIP - 宿主机 IP
4 metadata.name - Pod 的名字
5 metadata.namespace - Pod 的 Namespace
```



```
6 status.podIP - Pod 的 IP
7 spec.serviceAccountName - Pod 的 Service Account 的名字
8 metadata.uid - Pod 的 UID
9 metadata.labels['<KEY>'] - 指定 <KEY> 的 Label 值
10 metadata.annotations['<KEY>'] - 指定 <KEY> 的 Annotation 值
11 metadata.labels - Pod 的所有 Label
12 metadata.annotations - Pod 的所有 Annotation
13
14 2. 使用 resourceFieldRef 可以声明使用:
15 容器的 CPU limit
16 容器的 CPU request
17 容器的 memory limit
18 容器的 memory request
```

上面这个列表的内容，随着 Kubernetes 项目的发展肯定还会不断增加。所以这里列出来的信息仅供参考，你在使用 Downward API 时，还是要记得去查阅一下官方文档。

不过，需要注意的是，Downward API 能够获取到的信息，**一定是 Pod 里的容器进程启动之前就能够确定下来的信息**。而如果你想要获取 Pod 容器运行后才会出现的信息，比如，容器进程的 PID，那就肯定不能使用 Downward API 了，而应该考虑在 Pod 里定义一个 sidecar 容器。

其实，Secret、ConfigMap，以及 Downward API 这三种 Projected Volume 定义的信息，大多还可以通过环境变量的方式出现在容器里。但是，通过环境变量获取这些信息的方式，不具备自动更新的能力。所以，一般情况下，我都建议你使用 Volume 文件的方式获取这些信息。

在明白了 Secret 之后，**我再为你讲解 Pod 中一个与它密切相关的概念：Service Account**。

相信你一定有过这样的想法：我现在有了一个 Pod，我能不能在这个 Pod 里安装一个 Kubernetes 的 Client，这样就可以从容器里直接访问并且操作这个 Kubernetes 的 API 了呢？

这当然是可以的。

不过，你首先要解决 API Server 的授权问题。

Service Account 对象的作用，就是 Kubernetes 系统内置的一种“服务账户”，它是 Kubernetes 进行权限分配的对象。比如，Service Account A，可以只被允许对 Kubernetes API 进行 GET 操作，而 Service Account B，则可以有 Kubernetes API 的所有操作的权限。

像这样的 Service Account 的授权信息和文件，实际上保存在它所绑定的一个特殊的 Secret 对象里的。这个特殊的 Secret 对象，就叫作**ServiceAccountToken**。任何运行在 Kubernetes 集群上的应用，都必须使用这个 ServiceAccountToken 里保存的授权信息，也就是 Token，才可以合法地访问 API Server。

所以说，Kubernetes 项目的 Projected Volume 其实只有三种，因为第四种 ServiceAccountToken，只是一种特殊的 Secret 而已。

另外，为了方便使用，Kubernetes 已经为你提供了一个的默认“服务账户”（default Service Account）。并且，任何一个运行在 Kubernetes 里的 Pod，都可以直接使用这个默认的 Service Account，而无需显示地声明挂载它。

这是如何做到的呢？

当然还是靠 Projected Volume 机制。

如果你查看一下任意一个运行在 Kubernetes 集群里的 Pod，就会发现，每一个 Pod，都已经自动声明一个类型是 Secret、名为 default-token-xxxx 的 Volume，然后自动挂载在每个容器的一个固定目录上。比如：

 复制代码

```
1 $ kubectl describe pod nginx-deployment-5c678cfb6d-1g9lw
2 Containers:
3 ...
4 Mounts:
5   /var/run/secrets/kubernetes.io/serviceaccount from default-token-s8rbq (ro)
6 Volumes:
7   default-token-s8rbq:
8     Type:          Secret (a volume populated by a Secret)
9     SecretName:    default-token-s8rbq
10    Optional:      false
```

这个 Secret 类型的 Volume，正是默认 Service Account 对应的 ServiceAccountToken。所以说，Kubernetes 其实在每个 Pod 创建的时候，自动在它的 spec.volumes 部分添加上了默认 ServiceAccountToken 的定义，然后自动给每个容器加上了对应的 volumeMounts 字段。这个过程对于用户来说是完全透明的。

这样，一旦 Pod 创建完成，容器里的应用就可以直接从这个默认 ServiceAccountToken 的挂载目录里访问到授权信息和文件。这个容器内的路径在 Kubernetes 里是固定的，即：`/var/run/secrets/kubernetes.io/serviceaccount`，而这个 Secret 类型的 Volume 里面的内容如下所示：

 复制代码

```
1 $ ls /var/run/secrets/kubernetes.io/serviceaccount
2 ca.crt namespace token
```

所以，你的应用程序只要直接加载这些授权文件，就可以访问并操作 Kubernetes API 了。而且，如果你使用的是 Kubernetes 官方的 Client 包 (`k8s.io/client-go`) 的话，它还可以自动加载这个目录下的文件，你不需要做任何配置或者编码操作。

这种把 Kubernetes 客户端以容器的方式运行在集群里，然后使用 default Service Account 自动授权的方式，被称作 “InClusterConfig”，也是我最推荐的进行 Kubernetes API 编程的授权方式。

当然，考虑到自动挂载默认 ServiceAccountToken 的潜在风险，Kubernetes 允许你设置默认不为 Pod 里的容器自动挂载这个 Volume。


除了这个默认的 Service Account 外，我们很多时候还需要创建一些我们自己定义的 Service Account，来对应不同的权限设置。这样，我们的 Pod 里的容器就可以通过挂载这些 Service Account 对应的 ServiceAccountToken，来使用这些自定义的授权信息。在后面讲解为 Kubernetes 开发插件的时候，我们将会实践到这个操作。

接下来，我们再看 Pod 的另一个重要的配置：**容器健康检查和恢复机制**。

在 Kubernetes 中，你可以为 Pod 里的容器定义一个健康检查“探针”（Probe）。这样，kubelet 就会根据这个 Probe 的返回值决定这个容器的状态，而不是直接以容器进行

是否运行（来自 Docker 返回的信息）作为依据。这种机制，是生产环境中保证应用健康存活的重要手段。

我们一起来看一个 Kubernetes 文档中的例子。

 复制代码


```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   labels:
5     test: liveness
6   name: test-liveness-exec
7 spec:
8   containers:
9     - name: liveness
10       image: busybox
11       args:
12         - /bin/sh
13         - -c
14         - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
15       livenessProbe:
16         exec:
17           command:
18             - cat
19             - /tmp/healthy
20         initialDelaySeconds: 5
21         periodSeconds: 5
```

在这个 Pod 中，我们定义了一个有趣的容器。它在启动之后做的第一件事，就是在 /tmp 目录下创建了一个 healthy 文件，以此作为自己已经正常运行的标志。而 30 s 过后，它会把这个文件删除掉。

与此同时，我们定义了一个这样的 livenessProbe（健康检查）。它的类型是 exec，这意味着，它会在容器启动后，在容器里面执行一句我们指定的命令，比如：“cat /tmp/healthy”。这时，如果这个文件存在，这条命令的返回值就是 0，Pod 就会认为这个容器不仅已经启动，而且是健康的。这个健康检查，在容器启动 5 s 后开始执行（initialDelaySeconds: 5），每 5 s 执行一次（periodSeconds: 5）。


现在，让我们来**具体实践一下这个过程**。

首先，创建这个 Pod：

 复制代码

```
1 $ kubectl create -f test-liveness-exec.yaml
```


然后，查看这个 Pod 的状态：

 复制代码

```
1 $ kubectl get pod
2 NAME                READY    STATUS    RESTARTS   AGE
3 test-liveness-exec   1/1      Running   0           10s
```


可以看到，由于已经通过了健康检查，这个 Pod 就进入了 Running 状态。

而 30 s 之后，我们再查看一下 Pod 的 Events：

 复制代码

```
1 $ kubectl describe pod test-liveness-exec
```

你会发现，这个 Pod 在 Events 报告了一个异常：

 复制代码

```
1 FirstSeen LastSeen    Count   From              SubobjectPath          Type          Reason
2 -----
3 2s          2s          1    {kubelet worker0} spec.containers{liveness} Warning       Unhea:
```

显然，这个健康检查探测到 /tmp/healthy 已经不存在了，所以它报告容器是不健康的。那么接下来会发生什么呢？

我们不妨再次查看一下这个 Pod 的状态：

```
1 $ kubectl get pod test-liveness-exec
2 NAME                READY    STATUS    RESTARTS   AGE
3 liveness-exec       1/1     Running   1          1m
```

这时我们发现，Pod 并没有进入 Failed 状态，而是保持了 Running 状态。这是为什么呢？

其实，如果你注意到 RESTARTS 字段从 0 到 1 的变化，就明白原因了：这个异常的容器已经被 Kubernetes 重启了。在这个过程中，Pod 保持 Running 状态不变。

需要注意的是：Kubernetes 中并没有 Docker 的 Stop 语义。所以虽然是 Restart（重启），但实际却是重新创建了容器。

这个功能就是 Kubernetes 里的**Pod 恢复机制**，也叫 restartPolicy。它是 Pod 的 Spec 部分的一个标准字段（pod.spec.restartPolicy），默认值是 Always，即：任何时候这个容器发生了异常，它一定会被重新创建。

但一定要强调的是，Pod 的恢复过程，永远都是发生在当前节点上，而不会跑到别的节点上去。事实上，一旦一个 Pod 与一个节点（Node）绑定，除非这个绑定发生了变化（pod.spec.node 字段被修改），否则它永远都不会离开这个节点。这也就意味着，如果这个宿主机宕机了，这个 Pod 也不会主动迁移到其他节点上去。

而如果你想让 Pod 出现在其他的可用节点上，就必须使用 Deployment 这样的“控制器”来管理 Pod，哪怕你只需要一个 Pod 副本。这就是我在第 12 篇文章[《牛刀小试：我的第一个容器化应用》](#)最后给你留的思考题的答案，即一个单 Pod 的 Deployment 与一个 Pod 最主要的区别。

而作为用户，你还可以通过设置 restartPolicy，改变 Pod 的恢复策略。除了 Always，它还有 OnFailure 和 Never 两种情况：

Always：在任何情况下，只要容器不在运行状态，就自动重启容器；

OnFailure: 只在容器 异常时才自动重启容器；

Never: 从来不重启容器。


在实际使用时，我们需要根据应用运行的特性，合理设置这三种恢复策略。

比如，一个 Pod，它只计算 $1+1=2$ ，计算完成输出结果后退出，变成 Succeeded 状态。这时，你如果再用 `restartPolicy=Always` 强制重启这个 Pod 的容器，就没有任何意义了。

而如果你要关心这个容器退出后的上下文环境，比如容器退出后的日志、文件和目录，就需要将 `restartPolicy` 设置为 `Never`。因为一旦容器被自动重新创建，这些内容就有可能丢失掉了（被垃圾回收了）。

值得一提的是，Kubernetes 的官方文档，把 `restartPolicy` 和 Pod 里容器的状态，以及 Pod 状态的对应关系，[总结了非常复杂的一大堆情况](#)。实际上，你根本不需要死记硬背这些对应关系，只要记住如下两个基本的设计原理即可：

1. 只要 Pod 的 `restartPolicy` 指定的策略允许重启异常的容器（比如：`Always`），那么这个 Pod 就会保持 `Running` 状态，并进行容器重启。否则，Pod 就会进入 `Failed` 状态。
2. 对于包含多个容器的 Pod，只有它里面所有的容器都进入异常状态后，Pod 才会进入 `Failed` 状态。在此之前，Pod 都是 `Running` 状态。此时，Pod 的 `READY` 字段会显示正常容器的个数，比如：

 复制代码

```
1 $ kubectl get pod test-liveness-exec
2 NAME          READY    STATUS    RESTARTS   AGE
3 liveness-exec  0/1      Running   1           1m
```


所以，假如一个 Pod 里只有一个容器，然后这个容器异常退出了。那么，只有当 `restartPolicy=Never` 时，这个 Pod 才会进入 `Failed` 状态。而其他情况下，由于 Kubernetes 都可以重启这个容器，所以 Pod 的状态保持 `Running` 不变。

而如果这个 Pod 有多个容器，仅有一个容器异常退出，它就始终保持 `Running` 状态，哪怕即使 `restartPolicy=Never`。只有当所有容器也异常退出之后，这个 Pod 才会进入 `Failed` 状态。


其他情况，都可以以此类推出来。

现在，我们一起回到前面提到的 livenessProbe 上来。

除了在容器中执行命令外，livenessProbe 也可以定义为发起 HTTP 或者 TCP 请求的方式，定义格式如下：

 复制代码

```
1 ...
2 livenessProbe:
3   httpGet:
4     path: /healthz
5     port: 8080
6     httpHeaders:
7     - name: X-Custom-Header
8       value: Awesome
9     initialDelaySeconds: 3
10    periodSeconds: 3
```

 复制代码

```
1 ...
2 livenessProbe:
3   tcpSocket:
4     port: 8080
5     initialDelaySeconds: 15
6     periodSeconds: 20
```

所以，你的 Pod 其实可以暴露一个健康检查 URL（比如 /healthz），或者直接让健康检查去检测应用的监听端口。这两种配置方法，在 Web 服务类的应用中非常常用。

在 Kubernetes 的 Pod 中，还有一个叫 readinessProbe 的字段。虽然它的用法与 livenessProbe 类似，但作用却大不一样。readinessProbe 检查结果的成功与否，决定的这个 Pod 是不是能被通过 Service 的方式访问到，而并不影响 Pod 的生命周期。这部分内容，我会留在讲解 Service 时再重点介绍。

在讲解了这么多字段之后，想必你对 Pod 对象的语义和描述能力，已经有了一个初步的感觉。


这时，你有没有产生这样一个想法：Pod 的字段这么多，我又不可能全记住，Kubernetes 能不能自动给 Pod 填充某些字段呢？

这个需求实际上非常实用。比如，开发人员只需要提交一个基本的、非常简单的 Pod YAML，Kubernetes 就可以自动给对应的 Pod 对象加上其他必要的信息，比如 labels，annotations，volumes 等等。而这些信息，可以是运维人员事先定义好的。

这么一来，开发人员编写 Pod YAML 的门槛，就被大大降低了。

所以，这个叫作 PodPreset（Pod 预设置）的功能已经出现在了 v1.11 版本的 Kubernetes 中。

举个例子，现在开发人员编写了如下一个 pod.yaml 文件：

 复制代码

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: website
5   labels:
6     app: website
7     role: frontend
8 spec:
9   containers:
10    - name: website
11      image: nginx
12      ports:
13        - containerPort: 80
```

作为 Kubernetes 的初学者，你肯定眼前一亮：这不就是我最擅长编写的、最简单的 Pod 嘛。没错，这个 YAML 文件里的字段，想必你现在闭着眼睛也能写出来。

可是，如果运维人员看到了这个 Pod，他一定会连连摇头：这种 Pod 在生产环境里根本不能用啊！

所以，这个时候，运维人员就可以定义一个 PodPreset 对象。在这个对象中，凡是他想在开发人员编写的 Pod 里追加的字段，都可以预先定义好。比如这个 preset.yaml：

```
1 apiVersion: settings.k8s.io/v1alpha1
2 kind: PodPreset
3 metadata:
4   name: allow-database
5 spec:
6   selector:
7     matchLabels:
8       role: frontend
9   env:
10    - name: DB_PORT
11      value: "6379"
12   volumeMounts:
13    - mountPath: /cache
14      name: cache-volume
15   volumes:
16    - name: cache-volume
17      emptyDir: {}
```

在这个 PodPreset 的定义中，首先是一个 selector。这就意味着后面这些追加的定义，只会作用于 selector 所定义的、带有 “role: frontend” 标签的 Pod 对象，这就可以防止 “误伤” 。

然后，我们定义了一组 Pod 的 Spec 里的标准字段，以及对应的值。比如，env 里定义了 DB_PORT 这个环境变量，volumeMounts 定义了容器 Volume 的挂载目录，volumes 定义了一个 emptyDir 的 Volume。

接下来，我们假定运维人员先创建了这个 PodPreset，然后开发人员才创建 Pod：

```
1 $ kubectl create -f preset.yaml
2 $ kubectl create -f pod.yaml
```

这时，Pod 运行起来之后，我们查看一下这个 Pod 的 API 对象：

```
1 $ kubectl get pod website -o yaml
2 apiVersion: v1
```

```
3 kind: Pod
4 metadata:
5   name: website
6   labels:
7     app: website
8     role: frontend
9   annotations:
10    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
11 spec:
12   containers:
13     - name: website
14       image: nginx
15       volumeMounts:
16         - mountPath: /cache
17           name: cache-volume
18       ports:
19         - containerPort: 80
20       env:
21         - name: DB_PORT
22           value: "6379"
23   volumes:
24     - name: cache-volume
25       emptyDir: {}
```

这个时候，我们就可以清楚地看到，这个 Pod 里多了新添加的 labels、env、volumes 和 volumeMount 的定义，它们的配置跟 PodPreset 的内容一样。此外，这个 Pod 还被自动加上了一个 annotation 表示这个 Pod 对象被 PodPreset 改动过。

需要说明的是，**PodPreset 里定义的内容，只会在 Pod API 对象被创建之前追加在这个对象本身上，而不会影响任何 Pod 的控制器的定义。**

比如，我们现在提交的是一个 nginx-deployment，那么这个 Deployment 对象本身是永远不会被 PodPreset 改变的，被修改的只是这个 Deployment 创建出来的所有 Pod。这一点请务必区分清楚。

这里有一个问题：如果你定义了同时作用于一个 Pod 对象的多个 PodPreset，会发生什么呢？

实际上，Kubernetes 项目会帮你合并（Merge）这两个 PodPreset 要做的修改。而如果它们要做的修改有冲突的话，这些冲突字段就不会被修改。

总结

在今天这篇文章中，我和你详细介绍了 Pod 对象更高阶的使用方法，希望通过对这些实例的讲解，你可以更深入地理解 Pod API 对象的各个字段。

而在学习这些字段的同时，你还应该认真体会一下 Kubernetes “一切皆对象”的设计思想：比如应用是 Pod 对象，应用的配置是 ConfigMap 对象，应用要访问的密码则是 Secret 对象。

所以，也就自然而然地有了 PodPreset 这样专门用来对 Pod 进行批量化、自动化修改的工具对象。在后面的内容中，我会为你讲解更多的这种对象，还会和你介绍 Kubernetes 项目如何围绕着这些对象进行容器编排。

在本专栏中，Pod 对象相关的知识点非常重要，它是接下来 Kubernetes 能够描述和编排各种复杂应用的基石所在，希望你能够继续多实践、多体会。

思考题

在没有 Kubernetes 的时候，你是通过什么方法进行应用的健康检查的？Kubernetes 的 livenessProbe 和 readinessProbe 提供的几种探测机制，是否能满足你的需求？

感谢你的收听，欢迎你给我留言，也欢迎分享给更多的朋友一起阅读。

深入剖析 Kubernetes

Kubernetes 原来可以如此简单

张磊

Kubernetes 社区
资深成员与项目维护者



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 深入解析Pod对象（一）：基本概念

下一篇 16 | 编排其实很简单：谈谈“控制器”模型

精选留言 (54)

 写留言



huan

2018-09-26

 19

不实践，就无法理解为什么pod这么设计，这里给了我自己的实践的记录：

<https://github.com/huan9huan/k8s-practice/tree/master/15-pod-advanced>

仅供参考。

展开



snakorse

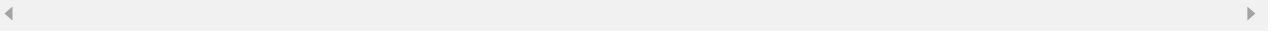
2018-09-26

 6

probe的原理是通过sidecar容器来实现的吗

展开 ▾

作者回复: 不是的, exec



Geek_zz

2018-09-26

👍 6

信息量大, 要多看两遍

展开 ▾



Vincent

2018-10-16

👍 4

按照老师的操作执行这个命令报错了

```
[root@node1 k8s]# kubectl create -f preset.yaml
error: unable to recognize "preset.yaml": no matches for kind "PodPreset" in
version "settings.k8s.io/v1alpha1"
```

...

展开 ▾



huan

2018-09-26

👍 4

目前项目使用了forever之类的wrapper进程来管理工作进程, forever是根据工作进程的状态做重启操作(也可以使用forever的api接口做health检查)。另外, 日志都是使用append的形式, 而不是覆盖, 这样可以不用掩盖错误的发生。

展开 ▾



我来也

2018-09-27

👍 3

在使用PodPreset对象时,发现并未生效,最终才知道是因为当初安装时未启用 Pod Preset. 然后参考

[<https://kubernetes.io/docs/concepts/workloads/pods/podpreset/#enable-pod-preset>] 修改 [/etc/kubernetes/manifests/kube-apiserver.yaml] 中的 spec.containers.command: 修改原[- --runtime-config=api/all=true]为[- --runtime...

展开 ▾

作者回复: 对。新特性需要先启用。



我来也

2018-09-26

👍 3

文章中的代码 dapi-volume.yaml 格式不对,被取消了缩进,导致直接贴出来使用会报错。还有按文章中的命令 `kubectl create secret generic user --from-file=./username.txt`, 在pod中[`kubectl exec -it test-projected-volume -- /bin/sh`]展示的目录不是user,而是username.txt. 可以通过[`kubectl edit secrets user`]手动修改data:下的字段名。

展开 ∨

作者回复: 请仔细看,我在文稿列出来的exec的输出,是第二种方法、也就是写YAML文件方法创建的Secret。而你列出来的是第一种方法创建的Secret。他俩本来就是不一样的。



Anker

2018-09-26

👍 3

Kubernetes 的 livenessProbe 和 readinessProbe 两个在项目中都用到了,最基本的是定时检测服务端口是否存在,毕竟好的是请求服务,例如针对http服务,发起一个请求,查看服务是否正常,因为有时候端口在,服务不一定正常。

展开 ∨



fhchina

2018-09-26

👍 3

是projected volume 1.11以后,不是persistent volume



蜗牛

2018-12-02

👍 2

复习了下容器的检查探针,有几个点还是没太明白,还望老师能解答下:

1. restartPolicy : 这个restartPolicy是重启的Pod的Container,那么重启的时机是根据Container结束时返回的状态码吗?
2. restart 和 probe的关系: Pod某个容器的livenessProbe 返回fail,这个时候Container并没有结束,只是状态检查是失败的,那为什么Container也会重启呢? 这个重启动作是谁发...

展开 ∨

作者回复: 很简单。restart policy 要尊重 liveness probe



龙坤

2018-09-27

👍 2

老师你好，有句话不太明白

原文：“相信你一定有过这样的想法：我现在有了一个 Pod，我能不能在这个 Pod 里安装一个 Kubernetes 的 Client，这样就可以从容器里直接访问并且操作这个 Kubernetes 的 API 了呢？”

...

展开 ▾

作者回复: client library，kubernetes 对外暴露的 api



北卡

2018-09-27

👍 2

我记得 deployment 所创建的 pod restart 策略只支持 always。是我使用的版本太低了吗？

作者回复: deployment 确实是这么要求的啊。不过你可以想想为什么。



W.T

2018-09-26

👍 2

在讲述 livenessProbe 的时候说到：虽然是 Restart（重启），但实际却是重新创建了容器；那之前那个还在运行的 liveness 容器被自动销毁了吗？

展开 ▾

作者回复: 对



假装乐

2018-09-26

👍 2

讲的详细量又大哈

展开 ▾



Tim Zhang

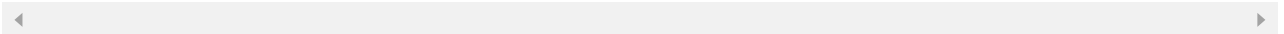
2018-09-26

👍 2

pv是1.11以后？

展开 ▾

作者回复: 再仔细看看.....



QQ怪

2019-04-22

👍 1

我公司也在用livenessprobe和readinessprobe，第一个是就绪探针在确定容器是否已经就绪可以接受流量，而第二个是存活容器来确定容器是否假死，如果遇到假死则开始重启p



Yao1931

2018-10-26

👍 1

Deployment是控制器，有副本数要求，如果恢复策略不仅是always就与它的设计违背了。是这样吗？



Racoon

2018-10-23

👍 1

geektime/sample:v2和geektime/tomcat:7.0 麻烦给一下这两个镜像的公有地址。



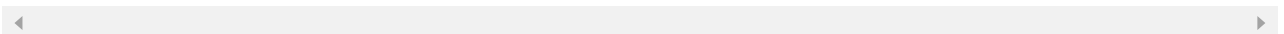
周娄子

2018-09-27

👍 1

podpreset我觉得不要什么情况都用，还是坚持所见即所得。

作者回复: 这是一个场景化的功能，要有批量操作的需求





gogo

2018-09-27

👍 1

老师好，请问java web项目的日志输出到分布式存储里面，怎么方便查看保存好的日志呢

作者回复: 存在elastic search里

