Pod 使用进阶

前面我们学习了 Pod 的整个生命周期，接下来我们将要来学习关于 Pod 的更多使用方式方法。

Pod 资源配置

> 本节测试最好别用 Kind 创建的集群，使用 Kubeadm 创建的真实集群更好。

实际上上面几个步骤就是影响一个 Pod 生命周期的大的部分，但是还有一些细节也会在 Pod 的启动过程进行设置，比如在容器启动之前还会为当前的容器设置分配的 CPU、内存等资源，我们知道我们可以通过 CGroup 来对容器的资源进行限制，同样的在 Pod 中我们也可以直接配置某个容器的使用的 CPU 或者内存的上限，那么 Pod 是如何来使用和控制这些资源的分配的呢？

首先对于 CPU，我们知道计算机里 CPU 的资源是按**时间片**的方式来进行分配的，系统里的每一个操作都需要 CPU 的处理，所以，哪个任务要是申请的 CPU 时间片越多，那么它得到的 CPU 资源就越多，这个很容易理解。

CPU 资源是以 **CPU** 单位度量的，Kubernetes 中的而一个 CPU 等同于：

- 1 个 AWS vCPU
- 1 个 GCP 核心
- 1 个 Azure vCore
- 裸机上具有超线程能力的英特尔处理器上的 1 个超线程

小数值也是可以使用的，一个请求 0.5 CPU 的容器会获得请求 1 个 CPU 的容器的 CPU 的一半。我们也可以使用后缀 m 表示毫，例如 100m CPU、100 milliCPU 和 0.1 CPU 都相同，需要注意精度不能超过 1m。

CPU 请求只能使用绝对数量，而不是相对数量。0.1 在单核、双核或 48 核计算机上的 CPU 数量值是一样的。

Kubernetes 集群中的每一个节点可以通过操作系统的命令来确认本节点的 CPU 内核数量，然后将这个数量乘以 1000，得到的就是节点总 CPU 总毫数。比如一个节点有四核，那么该节点的 CPU 总毫量为 4000m，如果你要使用一半的 CPU，则你要求的是 4000 * 0.5 = 2000m。在 Pod 里面我们可以通过下面的两个参数来限制和请求 CPU 资源：

- spec.containers[].resources.limits.cpu：CPU 上限值，可以短暂超过，容器也不会被停止
- spec.containers[].resources.requests.cpu：CPU 请求值，Kubernetes 调度算法里的依据值，可以超过

这里需要明白的是，如果 resources.requests.cpu 设置的值大于集群里每个节点的最大可用 CPU 核心数，那么这个 Pod 将无法调度，因为没有节点能满足它。

到这里应该明白了，requests 是用于集群调度使用的资源，而 limits 才是真正的用于资源限制的配置，如果你需要保证的你应用优先级很高，也就是资源吃紧的情况下最后再杀掉你的 Pod，那么可以将 requests 和 limits 的值设置成一致，在后面服务质量章节的时候会具体讲解。

比如，现在我们定义一个 Pod，给容器的配置如下的资源：

```yaml
# pod-resource-demo1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-demo1
spec:
  containers:
    - name: resource-demo1
      image: nginx
      ports:
```

```
        - containerPort: 80
      resources:
        requests:
          memory: 50Mi
          cpu: 50m
        limits:
          memory: 100Mi
          cpu: 100m
```

这里，CPU 我们给的是 50m，也就是 0.05core，这 0.05core 也就是占了 1 个 CPU 里的 5% 的资源时间。而限制资源是给的是 100m，但是需要注意的是 CPU 资源是可压缩资源，也就是容器达到了这个设定的上限后，容器性能会下降，但是不会终止或退出。比如我们直接创建上面这个 Pod：

```
☸ ➜ kubectl apply -f pod-resource-demo1.yaml
```

创建完成后，我们可以看到 Pod 被调度到 node1 这个节点上：

```
☸ ➜ kubectl get pods -o wide
NAME                    READY   STATUS    RESTARTS       AGE    IP              NODE
   NOMINATED NODE   READINESS GATES
resource-demo1          1/1     Running   0              80s    10.244.1.170    node1
  <none>           <none>
```

然后我们到 node1 节点上去查看 Pod 里面启动的 resource-demo1 这个容器：

```
☸ ➜ crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME
                        ATTEMPT             POD ID
5eed25df7a805       605c77e624ddb       2 minutes ago       Running
resource-demo1                      0                   1d7a28e7439c0
# ......
```

我们可以去查看下主容器的信息：

```
☸ ➜  crictl inspect 142292e178f94
crictl inspect 5eed25df7a805
{
  "status": {
    "id": "5eed25df7a805751e3e40a52d5284d72be88aa91237afef2ba6446919ed13cca",
    "metadata": {
      "attempt": 0,
      "name": "resource-demo1"
    },
# ......
      "linux": {
        "resources": {
          "devices": [
            {
              "allow": false,
              "access": "rwm"
```

```
            }
          ],
          "memory": {
            "limit": 104857600
          },
          "cpu": {
            "shares": 51,
            "quota": 10000,
            "period": 100000
          }
        },
        "cgroupsPath": "kubepods-burstable-
pod2346a6e1_c23e_4eac_99c7_5bee2af3b214.slice:cri-
containerd:5eed25df7a805751e3e40a52d5284d72be88aa91237afef2ba6446919ed13cca",
#  ......
```

实际上我们就可以看到这个容器的一些资源情况，Pod 上的资源配置最终也还是通过底层的容器运行时去控制 CGroup 来实现的，我们可以进入如下目录查看 CGroup 的配置，该目录就是 CGroup 父级目录，而 CGroup 是通过文件系统来进行资源限制的，所以我们上面限制容器的资源就可以在该目录下面反映出来：

```
# kubepods-burstable-pod2346a6e1_c23e_4eac_99c7_5bee2af3b214.slice:cri-
containerd:5eed25df7a805751e3e40a52d5284d72be88aa91237afef2ba6446919ed13cca
☸ ➜ cd /sys/fs/cgroup/cpu/kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
pod2346a6e1_c23e_4eac_99c7_5bee2af3b214.slice/
☸ ➜ ll
total 0
-rw-r--r-- 1 root root 0 Dec  8 15:20 cgroup.clone_children
--w--w--w- 1 root root 0 Dec  8 15:20 cgroup.event_control
-rw-r--r-- 1 root root 0 Dec  8 15:20 cgroup.procs
-r--r--r-- 1 root root 0 Dec  8 15:20 cpuacct.stat
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpuacct.usage
-r--r--r-- 1 root root 0 Dec  8 15:20 cpuacct.usage_percpu
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpu.rt_period_us
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpu.rt_runtime_us
-rw-r--r-- 1 root root 0 Dec  8 15:20 cpu.shares
-r--r--r-- 1 root root 0 Dec  8 15:20 cpu.stat
-rw-r--r-- 1 root root 0 Dec  8 15:20 notify_on_release
-rw-r--r-- 1 root root 0 Dec  8 15:20 tasks
☸ ➜ cat cpu.cfs_quota_us
10000
```

其中 `cpu.cfs_quota_us` 就是 CPU 的限制值，如果要查看具体的容器的资源，我们也可以进入到容器目录下面去查看即可。

最后我们了解下内存这块的资源控制，内存的单位换算比较简单：

`1 MiB = 1024 KiB`，内存这块在 Kubernetes 里一般用的是 `Mi` 单位，当然你也可以使用 `Ki`、`Gi` 甚至 `Pi`，看具体的业务需求和资源容量。

> 这里注意的是 MiB ≠ MB，MB 是十进制单位，MiB 是二进制，平时我们以为 MB 等于 1024KB，其实 1MB=1000KB， 1MiB 才等于 1024KiB。中间带字母 i 的是国际电工协会（IEC）定的，走 1024 乘积；KB、MB、GB 是国际单位制，走 1000 乘积。

这里要注意的是，内存是不可压缩性资源，如果容器使用内存资源到达了上限，那么会 OOM，造成内存溢出，容器就会终止和退出。我们也可以通过上面的方式去通过查看 CGroup 文件的值来验证资源限制。

超过容器限制的内存

当节点拥有足够的可用内存时，容器可以使用其请求的内存。但是，容器不允许使用超过其限制的内存。如果容器分配的内存超过其限制，该容器会成为被终止的候选容器，如果容器继续消耗超出其限制的内存，则终止容器。如果终止的容器可以被重启，则 kubelet 会重新启动它，就像其他任何类型的运行时失败一样。

如下所示我们创建一个 Pod，尝试分配超出其限制的内存，该容器的内存请求为 50 MiB，内存限制为 100 MiB：

```yaml
# memory-request-limit-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-1
spec:
  containers:
    - name: memory-demo-1-ctr
      image: polinux/stress
      resources:
        requests:
          memory: "50Mi"
        limits:
          memory: "100Mi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

在上面资源清单文件的 args 部分中的配置，表示该容器会尝试使用 250 MiB 的内存，这远高于我们声明的 100 MiB 的限制。我们直接应用上面的资源对象来查看下会出现什么状态：

```
☸ ➜ kubectl apply -f memory-request-limit-1.yaml
```

创建后查看 Pod 的详细信息：

```
☸ ➜ kubectl get pod memory-demo-1
```

此时，容器可能正在运行或被杀死。重复前面的命令，直到容器被杀掉：

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-1   0/1       OOMKilled   1          24s
```

获取容器更详细的状态信息，可以看出由于内存溢出（OOM），容器已被杀掉：

```
☸ ➜ kubectl describe pod memory-demo-1
Name:         memory-demo-1
```

```
Namespace:    default
# ......
    State:          Terminated
      Reason:       OOMKilled
      Exit Code:    1
      Started:      Thu, 08 Dec 2022 15:42:39 +0800
      Finished:     Thu, 08 Dec 2022 15:42:40 +0800
    Ready:          False
    Restart Count:  3
    Limits:
      memory:  100Mi
    Requests:
      memory:     50Mi
# ......
```

被 kill 掉后 kubelet 会重启它，所以当我们多次运行下面的命令，可以看到容器在反复的被杀死和重启：

```
☸ ➜ kubectl get pod memory-demo-1
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-1   0/1       OOMKilled   1          24s
☸ ➜ kubectl get pod memory-demo-1
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-1   1/1       Running     2          40s
```

这就是因为实际需要的内存已经大大超过 limit 限制的了，所以会被 OOMKill 掉，然后 kubelet 重启它又继续被 Kill。

超过整个节点容量的内存

内存请求和限制是与容器关联的，Pod 的内存请求是 Pod 中所有容器的内存请求之和。同理，Pod 的内存限制是 Pod 中所有容器的内存限制之和。

Pod 的调度是基于 requests 值的，只有当节点拥有足够满足 Pod 内存请求的内存时，才会将 Pod 调度至节点上运行。

如下所示我们创建一个 Pod，其内存请求超过了你集群中的任意一个节点所拥有的内存。在该 Pod 的资源清单文件中，拥有一个请求 1000 GiB 内存的容器，这应该超过了你集群中任何节点的容量。

```yaml
# memory-request-limit-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-2-ctr
      image: polinux/stress
      resources:
        requests:
          memory: "1000Gi"
```

```
        limits:
          memory: "1000Gi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

同样直接创建该 Pod 即可：

```
☸ ➜ kubectl apply -f memory-request-limit-2.yaml
```

查看 Pod 状态：

```
☸ ➜ kubectl get pod memory-demo-2
```

输出结果显示：Pod 处于 PENDING 状态，这意味着，该 Pod 没有被调度至任何节点上运行，并且它会无限期的保持该状态：

```
☸ ➜ kubectl get pod memory-demo-2
NAME            READY   STATUS    RESTARTS   AGE
memory-demo-2   0/1     Pending   0          6s
```

查看关于 Pod 的详细信息，包括事件：

```
☸ ➜ kubectl describe pod memory-demo-2
```

输出结果显示由于节点内存不足，该容器无法被调度：

```
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  41s   default-scheduler  0/2 nodes are available: 1
Insufficient memory, 1 node(s) had taint {node-role.kubernetes.io/master: }, that the
pod didn't tolerate.
```

给 Pod 分配扩展资源

除了我们经常使用的 CPU 和内存之外，其实我们也可以自己定制扩展资源，要请求扩展资源，需要在你的容器清单中包括 resources.requests 字段。扩展资源可以使用任何完全限定名称，只是不能使用 *.kubernetes.io/，比如 example.com/foo 就是有效的格式，其中 example.com 可以被替换为你组织的域名，而 foo 则是描述性的资源名称。

扩展资源类似于内存和 CPU 资源。一个节点拥有一定数量的内存和 CPU 资源，它们被节点上运行的所有组件共享，该节点也可以拥有一定数量的 foo 资源，这些资源同样被节点上运行的所有组件共享。此外我们也可以创建请求一定数量 foo 资源的 Pod。

假设一个节点拥有一种特殊类型的磁盘存储，其容量为 800 GiB，那么我们就可以为该特殊存储创建一个名称，如 example.com/special-storage，然后你就可以按照一定规格的块（如 100 GiB）对其进行发布。在这种情况下，你的节点将会通知它拥有八个 example.com/special-storage 类型的资源。

```
Capacity:
 ...
 example.com/special-storage: 8
```

如果你想要允许针对特殊存储任意（数量）的请求，你可以按照 1 字节大小的块来发布特殊存储。在这种情况下，你将会发布 800Gi 数量的 example.com/special-storage 类型的资源。

```
Capacity:
 ...
 example.com/special-storage:  800Gi
```

然后，容器就能够请求任意数量（多达 800Gi）字节的特殊存储。

扩展资源对 Kubernetes 是不透明的。Kubernetes 不知道扩展资源含义相关的任何信息。 Kubernetes 只了解一个节点拥有一定数量的扩展资源。 扩展资源必须以整形数量进行发布。 例如，一个节点可以发布 4 个 dongle 资源，但是不能发布 4.5 个。

在 Pod 中分配扩展资源之前，我们还需要将该扩展资源发布到节点上去，我们可以直接发送一个 HTTP PATCH 请求到 Kubernetes API server 来完成该操作，假设你的一个节点上带有四个 course 资源，下面是一个 PATCH 请求的示例，该请求为你的节点发布四个 course 资源。

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "add",
    "path": "/status/capacity/ydzs.io~1course",
    "value": "4"
  }
]
```

> 注意：Kubernetes 不需要了解 course 资源的含义和用途，前面的 PATCH 请求告诉 Kubernetes 你的节点拥有四个你称之为 course 的东西。

启动一个代理，然后就可以很容易地向 Kubernetes API server 发送请求：

```
☸ ➜ kubectl proxy
```

在另一个命令窗口中，发送 HTTP PATCH 请求，用你的节点名称替换 <your-node-name>：

```
☸ ➜ curl --header "Content-Type: application/json-patch+json" \
 --request PATCH \
 --data '[{"op": "add", "path": "/status/capacity/ydzs.io~1course", "value": "4"}]' \
 http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

在前面的请求中，~1 为 patch 路径中 / 符号的编码，输出显示该节点的 course 资源容量（capacity）为 4：

```
"capacity": {
    "cpu": "16",
    "ephemeral-storage": "41218252Ki",
    "hugepages-1Gi": "0",
    "hugepages-2Mi": "0",
    "memory": "32771184Ki",
    "pods": "110",
    "ydzs.io/course": "4"
}
```

描述你的节点也可以看到对应的资源数据：

```
☸ ➜ kubectl describe node <your-node-name>
# ......
Capacity:
  cpu:                16
  ephemeral-storage:  41218252Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             32771184Ki
  pods:               110
  ydzs.io/course:     4
Allocatable:
  cpu:                16
  ephemeral-storage:  37986740981
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             32668784Ki
  pods:               110
  ydzs.io/course:     4
# ......
```

同样如果要移出该扩展资源，则发布如下所示的 PATCH 请求即可。

```
☸ ➜ curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "remove", "path": "/status/capacity/ydzs.io~1course"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

验证 course 资源的发布已经被移除：

```
☸ ➜ kubectl describe node <your-node-name> | grep course
```

正常应该看不到任何输出了。

请求扩展资源

现在我们就可以创建请求一定数量 course 资源的 Pod 了。比如我们这里有一个如下所示的资源清单文件：

```yaml
# resource-extended-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo
spec:
  containers:
    - name: extended-resource-demo-ctr
      image: nginx
      resources:
        requests:
          ydzs.io/course: 3
        limits:
          ydzs.io/course: 3
```

在该资源清单文件中我们配置请求了 3 个名为 ydzs.io/course 的扩展资源，同样直接创建该资源对象即可：

```
☸ ➜ kubectl apply -f resource-extended-demo.yaml
```

检查 Pod 是否运行正常：

```
☸ ➜ kubectl get pod extended-resource-demo
NAME                      READY   STATUS    RESTARTS   AGE
extended-resource-demo    1/1     Running   0          33s
```

可以看到该 Pod 可以正常运行，因为目前的扩展资源是满足调度条件的，所以可以正常调度。

同样的我们再创建一个类似的新的 Pod，资源清单文件如下所示：

```yaml
# resource-extended-demo2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo-2
spec:
  containers:
    - name: extended-resource-demo-2-ctr
      image: nginx
      resources:
        requests:
          ydzs.io/course: 2
        limits:
          ydzs.io/course: 2
```

该 Pod 的容器请求了 2 个 course 扩展资源，Kubernetes 将不能满足该资源的请求，因为上面的 Pod 已经使用了 4 个可用 course 中的 3 个。

我们可以尝试创建该 Pod：

```
☸ ➜ kubectl apply -f resource-extended-demo2.yaml
```

创建后查看 Pod 的状态：

```
☸ ➜ kubectl get pod extended-resource-demo-2
NAME                        READY   STATUS    RESTARTS   AGE
extended-resource-demo-2    0/1     Pending   0          92s
```

可以看到当前 Pod 是处于 Pending 状态的，描述下 Pod：

```
☸ ➜ kubectl describe pod extended-resource-demo-2
# ......
                            node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason            Age                 From               Message
  ----     ------            ----                ----               -------
  Warning  FailedScheduling  31s (x4 over 2m35s)  default-scheduler  0/2 nodes are
available: 1 Insufficient ydzs.io/course, 1 node(s) had taint {node-
role.kubernetes.io/master: }, that the pod didn't tolerate.
```

输出结果表明 Pod 不能被调度，因为没有一个节点上存在两个可用的 course 资源了。

静态 Pod

在 Kubernetes 集群中除了我们经常使用到的普通的 Pod 外，还有一种特殊的 Pod，叫做 Static Pod，也就是我们说的静态 Pod，静态 Pod 有什么特殊的地方呢？

静态 Pod 直接由节点上的 kubelet 进程来管理，不通过 master 节点上的 apiserver。无法与我们常用的控制器 Deployment 或者 DaemonSet 进行关联，它由 kubelet 进程自己来监控，当 pod 崩溃时会重启该 pod，kubelet 也无法对他们进行健康检查。静态 pod 始终绑定在某一个 kubelet 上，并且始终运行在同一个节点上。kubelet 会自动为每一个静态 pod 在 Kubernetes 的 apiserver 上创建一个镜像 Pod，因此我们可以在 apiserver 中查询到该 pod，但是不能通过 apiserver 进行控制（例如不能删除）。

创建静态 Pod 有两种方式：配置文件和 HTTP 两种方式

配置文件

配置文件就是放在特定目录下的标准的 JSON 或 YAML 格式的 pod 定义文件。用 kubelet --pod-manifest-path=<the directory> 来启动 kubelet 进程，kubelet 定期的去扫描这个目录，根据这个目录下出现或消失的 YAML/JSON 文件来创建或删除静态 pod。

比如我们在 node1 这个节点上用静态 pod 的方式来启动一个 nginx 的服务，配置文件路径为：

```
☸ ➜ cat /var/lib/kubelet/config.yaml
......
staticPodPath: /etc/kubernetes/manifests  # 和命令行的 pod-manifest-path 参数一致
......
```

打开这个文件我们可以看到其中有一个属性为 staticPodPath 的配置，其实和命令行的 --pod-manifest-path 配置是一致的，所以如果我们通过 kubeadm 的方式来安装的集群环境，对应的 kubelet 已经配置了我们的静态 Pod 文件的路径，默认地址为 /etc/kubernetes/manifests，所以我们只需要在该目录下面创建一个标准的 Pod 的 JSON 或者 YAML 文件即可，如果你的 kubelet 启动参数中没有配置上面的 --pod-manifest-path 参数的话，那么添加上这个参数然后重启 kubelet 即可：

```
☸ ➜ cat <<EOF >/etc/kubernetes/manifests/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    app: static
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
EOF
```

通过 HTTP 创建静态 Pods

kubelet 周期地从 -manifest-url= 参数指定的地址下载文件，并且把它翻译成 JSON/YAML 格式的 pod 定义。此后的操作方式与 -pod-manifest-path= 相同，kubelet 会不时地重新下载该文件，当文件变化时对应地终止或启动静态 pod。

kubelet 启动时，由 --pod-manifest-path= 或 --manifest-url= 参数指定的目录下定义的所有 pod 都会自动创建，例如，我们示例中的 static-web：

```
☸ ➜ nerdctl -n k8s.io ps
CONTAINER ID    IMAGE                                                   COMMAND
    CREATED           STATUS    PORTS    NAMES
6add7aa53969    docker.io/library/nginx:latest                          "/docker-
entrypoint.…"    43 seconds ago    Up
......
```

现在我们通过 kubectl 工具可以看到这里创建了一个新的镜像 Pod：

```
☸ ➜ kubectl get pods
NAME              READY   STATUS    RESTARTS   AGE
static-web-node1  1/1     Running   0          109s
```

静态 pod 的标签会传递给镜像 Pod，可以用来过滤或筛选。 需要注意的是，我们不能通过 API 服务器来删除静态 pod（例如，通过 kubectl 命令），kubelet 不会删除它。

```
☸ ➜ kubectl delete pod static-web-node1
pod "static-web-node1" deleted
☸ ➜ kubectl get pods
NAME               READY   STATUS    RESTARTS   AGE
static-web-node1   1/1     Running   0          4s
```

静态 Pod 的动态增加和删除

运行中的 kubelet 周期扫描配置的目录（我们这个例子中就是 /etc/kubernetes/manifests）下文件的变化，当这个目录中有文件出现或消失时创建或删除 pods：

```
☸ ➜ mv /etc/kubernetes/manifests/static-web.yaml /tmp
# sleep 20
☸ ➜ nerdctl -n k8s.io ps
// no nginx container is running
☸ ➜ mv /tmp/static-web.yaml  /etc/kubernetes/manifests
# sleep 20
☸ ➜ nerdctl -n k8s.io ps
CONTAINER ID    IMAGE                                     COMMAND
    CREATED          STATUS    PORTS    NAMES
902be9190538    docker.io/library/nginx:latest            "/docker-
entrypoint.…"    14 seconds ago    Up
......
```

其实我们用 kubeadm 安装的集群，master 节点上面的几个重要组件都是用静态 Pod 的方式运行的，我们登录到 master 节点上查看 /etc/kubernetes/manifests 目录：

```
☸ ➜ ls /etc/kubernetes/manifests/
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml  kube-scheduler.yaml
```

现在明白了吧，这种方式也为我们将集群的一些组件容器化提供了可能，因为这些 Pod 都不会受到 apiserver 的控制，不然我们这里 kube-apiserver 怎么自己去控制自己呢？万一不小心把这个 Pod 删掉了呢？所以只能有 kubelet 自己来进行控制，这就是我们所说的静态 Pod。

Downward API

前面我们从 Pod 的原理到生命周期介绍了 Pod 的一些使用，作为 Kubernetes 中最核心的资源对象、最基本的调度单元，我们可以发现 Pod 中的属性还是非常繁多的，前面我们使用过一个 volumes 的属性，表示声明一个数据卷，我们可以通过命令 kubectl explain pod.spec.volumes 去查看该对象下面的属性非常多，前面我们只是简单使用了 hostPath 和 emptyDir{} 这两种模式，其中还有一种模式叫做 downwardAPI，这个模式和其他模式不一样的地方在于它不是为了存放容器的数据也不是用来进行容器和宿主机的数据交换的，而是让 Pod 里的容器能够直接获取到这个 Pod 对象本身的一些信息。

目前 Downward API 提供了两种方式用于将 Pod 的信息注入到容器内部：

- 环境变量：用于单个变量，可以将 Pod 信息和容器信息直接注入容器内部
- Volume 挂载：将 Pod 信息生成为文件，直接挂载到容器内部中去

环境变量

我们通过 Downward API 来将 Pod 的 IP、名称以及所对应的 namespace 注入到容器的环境变量中去，然后在容器中打印全部的环境变量来进行验证，对应资源清单文件如下：

```yaml
# env-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
  namespace: kube-system
spec:
  containers:
    - name: env-pod
      image: busybox
      command: ["/bin/sh", "-c", "env"]
      env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

我们可以看到上面我们使用了一种新的方式来设置 env 的值：valueFrom，由于 Pod 的 name 和 namespace 属于元数据，是在 Pod 创建之前就已经定下来了的，所以我们可以使用 metata 就可以获取到了，但是对于 Pod 的 IP 则不一样，因为我们知道 Pod IP 是不固定的，Pod 重建了就变了，它属于状态数据，所以我们使用 status 这个属性去获取。另外除了使用 fieldRef 获取 Pod 的基本信息外，还可以通过 resourceFieldRef 去获取容器的资源请求和资源限制信息。

我们直接创建上面的 Pod：

```
☸ ➜ kubectl apply -f env-pod.yaml
pod "env-pod" created
```

Pod 创建成功后，我们可以查看日志：

```
☸ ➜ kubectl logs env-pod -n kube-system |grep POD
kubectl logs -f env-pod -n kube-system
POD_IP=10.244.1.121
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBE_DNS_SERVICE_PORT_DNS_TCP=53
HOSTNAME=env-pod
SHLVL=1
```

```
HOME=/root
KUBE_DNS_SERVICE_HOST=10.96.0.10
KUBE_DNS_PORT_9153_TCP_ADDR=10.96.0.10
KUBE_DNS_PORT_9153_TCP_PORT=9153
KUBE_DNS_PORT_9153_TCP_PROTO=tcp
KUBE_DNS_SERVICE_PORT=53
KUBE_DNS_PORT=udp://10.96.0.10:53
POD_NAME=env-pod
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBE_DNS_PORT_53_TCP_ADDR=10.96.0.10
KUBERNETES_PORT_443_TCP_PORT=443
KUBE_DNS_SERVICE_PORT_METRICS=9153
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBE_DNS_PORT_9153_TCP=tcp://10.96.0.10:9153
KUBE_DNS_PORT_53_UDP_ADDR=10.96.0.10
KUBE_DNS_PORT_53_TCP_PORT=53
KUBE_DNS_PORT_53_TCP_PROTO=tcp
KUBE_DNS_PORT_53_UDP_PORT=53
KUBE_DNS_SERVICE_PORT_DNS=53
KUBE_DNS_PORT_53_UDP_PROTO=udp
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
POD_NAMESPACE=kube-system
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
KUBE_DNS_PORT_53_TCP=tcp://10.96.0.10:53
KUBE_DNS_PORT_53_UDP=udp://10.96.0.10:53
```

我们可以看到 Pod 的 IP、NAME、NAMESPACE 都通过环境变量打印出来了。

上面打印 Pod 的环境变量可以看到有很多内置的变量，其中大部分是系统自动为我们添加的，Kubernetes 会把当前命名空间下面的 Service 信息通过环境变量的形式注入到 Pod 中去：

```
$ kubectl get svc -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   4d21h
```

Volume 挂载

Downward API除了提供环境变量的方式外，还提供了通过 Volume 挂载的方式去获取 Pod 的基本信息。接下来我们通过Downward API将 Pod 的 Label、Annotation 等信息通过 Volume 挂载到容器的某个文件中去，然后在容器中打印出该文件的值来验证，对应的资源清单文件如下所示：

```yaml
# volume-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
  namespace: kube-system
```

```yaml
  labels:
    k8s-app: test-volume
    node-env: test
  annotations:
    own: youdianzhishi
    build: test
spec:
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: labels
            fieldRef:
              fieldPath: metadata.labels
          - path: annotations
            fieldRef:
              fieldPath: metadata.annotations
  containers:
    - name: volume-pod
      image: busybox
      args:
        - sleep
        - "3600"
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
```

我们将元数据 labels 和 annotaions 以文件的形式挂载到了 /etc/podinfo 目录下，创建上面的 Pod：

```
☸ ➜ kubectl create -f volume-pod.yaml
pod "volume-pod" created
```

创建成功后，我们可以进入到容器中查看元信息是不是已经存入到文件中了：

```
☸ ➜ kubectl exec -it volume-pod /bin/sh -n kube-system
/ # ls /etc/podinfo/
annotations  labels
/ # cat /etc/podinfo/labels
k8s-app="test-volume"
/ # cat /etc/podinfo/labels
k8s-app="test-volume"
node-env="test"/ #
```

我们可以看到 Pod 的 Labels 和 Annotations 信息都被挂载到 /etc/podinfo 目录下面的 labels 和 annotations 文件了。

目前，Downward API 支持的字段已经非常丰富了，比如：

1. 使用 fieldRef 可以声明使用：

- spec.nodeName：宿主机名字
- status.hostIP：宿主机 IP

- metadata.name：Pod 的名字
- metadata.namespace：Pod 的 Namespace
- status.podIP：Pod 的 IP
- spec.serviceAccountName：Pod 的 Service Account 的名字
- metadata.uid：Pod 的 UID
- metadata.labels['<KEY>']：指定 <KEY> 的 Label 值
- metadata.annotations['<KEY>']：指定 <KEY> 的 Annotation 值
- metadata.labels：Pod 的所有 Label
- metadata.annotations：Pod 的所有 Annotation

2. 使用 resourceFieldRef 可以声明使用：

- 容器的 CPU limit
- 容器的 CPU request
- 容器的 memory limit
- 容器的 memory request

> 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 容器来获取了。

在实际应用中，如果你的应用有获取 Pod 的基本信息的需求，一般我们就可以利用 Downward API 来获取基本信息，然后编写一个启动脚本或者利用 initContainer 将 Pod 的信息注入到我们容器中去，然后在我们自己的应用中就可以正常的处理相关逻辑了。

除了通过 Downward API 可以获取到 Pod 本身的信息之外，其实我们还可以通过映射其他资源对象来获取对应的信息，比如 Secret、ConfigMap 资源对象，同样我们可以通过环境变量和挂载 Volume 的方式来获取他们的信息，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，都建议使用 Volume 文件的方式获取这些信息，因为通过 Volume 的方式挂载的文件在 Pod 中会进行热更新。