深入剖析容器

> 这是本专栏的第二部分：容器篇，共 6 篇，帮助大家由浅入深的认识和掌握容器。前面，我为你介绍了容器生命周期管理和资源管理相关的内容，让你对容器有了更加灵活的控制。本篇，我来为你深入剖析容器，从本质上理解到底容器是什么。

在正式开始之前，我们总结一下到目前为止的学习：

- 使用 Docker 相关的一些命令可以完成对容器生命周期的管理，包括启动、暂停、停止、删除等；
- 通过 `docker update` 命令可热更新容器的一些配置，包括 CPU，内存等资源限制；
- 在宿主机上可使用 `docker top` 和 `docker stats` 命令拿到容器的一些状态，并且也可通过访问 /sys/fs/cgroup 下的一些特定目录或文件，得到容器的相关信息。

现在，我们来更进一步，对容器进行深入剖析。

容器是什么

在前面我们一直都在用 Docker 启动和管理容器，第一部分中也提到了关于容器技术和 Docker 的发展历程，我们不妨看看 Docker 对容器的定义是什么。

引用 Docker 官网对容器的一个定义：

> What is a Container?
>
> A standardized unit of software.

容器是什么？一个软件的**标准化单元**。

我们来分析下这个定义，首先是**软件**，跟容器相关的是软件而不是硬件，而我们也知道软件主要分为系统软件和应用软件，而容器中运行的程序并非系统软件，它实际运行在宿主机上，与宿主机上的其他进程共用一个内核，这也是容器与传统虚拟机的一个很大区别。

再者，**标准化单元**，刚才我们已经说了，容器内运行的程序并非系统软件，每个软件运行都需要有必要的环境，包括一些 lib 库之类的，而如何能在复杂的环境中做到“标准化”呢？显然，**隔离**是一个最佳选择。将程序及其所需的环境 /lib 库之类的组织在一起，并与系统环境隔离，这就很容易做到“标准化”了。

说白了，**容器其实是在一台机器上的“一组”进程，当然这组进程可能只有一个；它们有相同的特性，同样所受的限制也相同；另外既然叫做容器，很自然地我们认为它们与外界可以进行隔离/应该有一个分界线。**

在本专栏的第一部分，我为你介绍过 Docker 和容器技术的发展历程。本篇中，我们不妨从进程的角度来深入容器内部。

获取容器底层信息

Docker 为我们提供了一个很方便的命令 `docker inspect`，使用此命令可以得到容器的底层信息。我们先启动一个 Alpine Linux 的容器，然后用此命令查看该容器的信息：

复制

```
(MoeLove) ➜  ~ docker run --rm -it alpine
/ #
```

打开另一个终端窗口，执行以下命令：

复制

```
(MoeLove) ➜  ~ docker inspect $(docker ps -ql)
[
    {
        "Id": "2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e78cf6cda956a7",
        "Path": "/bin/sh",
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 9911,
            "ExitCode": 0,
            "Error": "",
            ...
        },
        ...
        "Image": "sha256:055936d3920576da37aa9bc460d70c5f212028bda1c08c0879aedf03d
        "HostnamePath": "/var/lib/docker/containers/2bc8701945480c70df09a4cca36909
        "HostsPath": "/var/lib/docker/containers/2bc8701945480c70df09a4cca36909b67
        "Name": "/sleepy_austin",
        "Driver": "overlay2",
        "Platform": "linux",
        "Config": {
            "Hostname": "2bc870194548",
            "Domainname": "",
            "User": "",
            "AttachStdin": true,
            "AttachStdout": true,
            "AttachStderr": true,
            "Tty": true,
            "OpenStdin": true,
            "StdinOnce": true,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
            ],
            "Cmd": [
                "/bin/sh"
            ],
            "Image": "alpine",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
```

```
        },
        ...
    }
]
```

篇幅原因，省略了其中部分输出。我们看到这个命令的输出是一个列表，所以它支持同时查看多个容器的信息，我们现在关注的重点在于 State 字段，其中包含了 Pid 等信息。

当然你也会因此发现，State 字段内的信息完全符合我们在《容器生命周期管理》中的介绍。

接下来我们直接取这个 Pid 的内容即可。通常我们会使用 grep 命令从该输出中过滤信息，但 Docker 给我们提供了更加便捷的方式：

复制

```
(MoeLove) ➜  ~ docker inspect --format '{{.State.Pid}}' $(docker ps -ql)
9911
```

`docker inspect` 命令有一个 `--format` 选项，可支持 Go 语言模板的字符串。可以直接通过上述命令得到容器的实际 Pid。当然，Go 语言模板也支持各种命令和表达式，可直接在官方文档学习，后续内容中将不再特别进行说明。

/proc 文件系统

既然要从进程的角度对容器进行深入的了解，那就不得不提 /proc 文件系统了，它是一种虚拟的文件系统。该文件系统挂载于 /proc 目录下，包含了各种用于展示内核信息的文件，并且允许进程通过普通的 I/O 调用来直接读取，当然有些内容也可以直接进行修改。

另外，之所以称它为虚拟文件系统，是因为它并不真正存在于磁盘中，而是由内核进行创建更新和管理的。

那我们想要了解的容器进程，在 /proc 文件系统中是一种怎样的形式呢？

/proc 文件系统对于进程都提供了一个 /proc/$PID 的目录，其中包含着进程的详细信息。现在来查看下刚才容器进程的相关信息。

复制

```
(MoeLove) ➜  ~ sudo ls  /proc/9911/
attr        comm             fd        loginuid   mountstats  oom_score_adj  scheds
autogroup   coredump_filter  fdinfo    map_files  net         pagemap        sessio
auxv        cpuset           gid_map   maps       ns          personality    setgro
cgroup      cwd              io        mem        numa_maps   projid_map     smaps
clear_refs  environ          latency   mountinfo  oom_adj     root           smaps_
cmdline     exe              limits    mounts     oom_score   sched           stack
```

可以看到 /proc 文件系统内包含的内容很多，这里我们先忽略掉一些常规内容，重点分析与容器相关的部分。感兴趣可以可以自行了解下 /proc 文件系统。

cgroup 文件

这其中有一个 cgroup 文件，我们看看其中的内容：

复制

```
(MoeLove) ➜  ~ sudo cat  /proc/9911/cgroup
11:perf_event:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4a
10:freezer:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82
9:blkio:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e78
8:devices:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e
7:net_cls,net_prio:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287d
6:pids:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e78c
5:cpu,cpuacct:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4a
4:cpuset:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e7
3:hugetlb:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e
2:memory:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e7
1:name=systemd:/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4
0::/system.slice/docker-2bc8701945480c70df09a4cca36909b67221b4287df7f4af82e78cf6cd
```

这个文件的内容有三列，第一列是个数字，暂且不谈；第二列看着很像一些特定的属性，比如 Memory 和 CPU 之类的；第三列则很像是目录结构，`/system.slice/docker-容器的ID`。

还记得在上一篇最后我提供的，用于在宿主机上查看容器内存限制的命令么？是否感觉很熟悉？如果遗忘了，可以再回过头去复习一下。

另一个重要是一个名为 ns 的目录：

复制

```
(MoeLove) ➜  ~ sudo ls -l --time-style='+' /proc/9911/ns
total 0
lrwxrwxrwx. 1 root root 0  cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx. 1 root root 0  ipc -> 'ipc:[4026532722]'
lrwxrwxrwx. 1 root root 0  mnt -> 'mnt:[4026532720]'
lrwxrwxrwx. 1 root root 0  net -> 'net:[4026532726]'
lrwxrwxrwx. 1 root root 0  pid -> 'pid:[4026532723]'
lrwxrwxrwx. 1 root root 0  pid_for_children -> 'pid:[4026532723]'
lrwxrwxrwx. 1 root root 0  user -> 'user:[4026531837]'
lrwxrwxrwx. 1 root root 0  uts -> 'uts:[4026532721]'
```

可以看到其中的内容都是一些链接，这里我们先暂时不对它进行展开，有个大概印象即可。后续内容会再进行深入。

进程树

我们使用 pstree 命令查看下容器内进程的进程树。

```
(MoeLove) ➜  ~ pstree -Aaps 9911
systemd,1 --switched-root --system --deserialize 33
  `-containerd,1130
      `-containerd-shim,9892 -namespace moby -workdir...
          `-sh,9911
```

复制

可以看到容器内进程从根开始依次为：

```
systemd(1)---containerd(1130)---containerd-shim(9892)---sh(9911)
```

复制

这是从另一个侧面反映了容器的创建过程，但在本篇中，只需要你对它们有个大致印象即可。

当我们在容器内运行程序的时候又会是什么样呢？比如下面的例子。

在容器内运行 sha256sum 程序：

```
/ # sha256sum /dev/zero
```

复制

在宿主机上再次查看进程树：

```
(MoeLove) ➜  ~ pstree -Aps 9911
systemd(1)---containerd(1130)---containerd-shim(9892)---sh(9911)---sha256sum(26985
```

复制

可以看到在容器内运行的进程，都是容器进程的子进程。由于 Linux 上的父子进程有一定的继承关系，所以大多数的能力和限制也都会被继承下来。

换个角度来看，这自然也相当于是一个以容器进程开始的分组。

总结

本篇，我为你从进程的角度介绍了容器的一些细节，但本篇并没有过于深入原理，只是通过 `docker inspect` 和 Linux 的 /proc 文件系统为你做了一些铺垫，方便后续内容的学习。

其中 `docker inspect` 命令配合 `--format` 参数的 Go 模板，可以很方便的完成很多需要容器底层信息的常规需求。当然你也可以使用 jq 工具解析其返回的 JSON 数据进行处理。

/proc 文件系统，以及其中的 cgroup 和 ns 将会对理解和掌握容器的核心原理有很大帮助。

下一篇，我将为你解密容器的核心技术，将本篇遗留的问题进行解答。

相关内容：

- 软件的定义