Docker 核心架构...

从本篇起，我们将正式进入本专栏第四部分“架构篇”的内容。这个部分会涉及到 Docker 的架构及各个组件间的分工协作，对 Docker Plugin 的扩展，Docker 的日志和监控实践，以及在使用 Docker 时可能遇到的问题及排查思路等内容。

本篇，我会为你从较高的层次来介绍 Docker 的核心架构，之后两篇则会深入其中，做更为细致的讲解。

之后 Docker Plugin 实践、日志、监控实践等都会是完整的实践。让我们正式进入这个部分的学习。

Docker 的整体架构

在前面内容中，我分别为你介绍了“容器篇”，“镜像篇”和“CI/CD 篇”，其中对于 Docker 的架构只是粗略的进行过一些描述，比如 Docker 整体而言是 C/S 架构，平时我们使用的 docker 命令是其 CLI 工具，而我们所启动的 dockerd 是其运行在后台的 daemon。

本篇我们换个角度，看看 Docker 更为具体的架构，以及其包含的组件。

首先，我们看看在安装完 Docker 之后，我们装了哪些东西，可以直接装完 Docker 后查看，也可以直接以 `docker:dind` 的容器为例。（注意：**以下内容仅限于在 Linux 环境中**）

在装完 Docker 后，我们主要就安装了三类内容。

docker 相关：

```
docker                  # docker cli
dockerd                 # docker daemon
docker-init             # docker init 程序
docker-proxy            # docker userland proxy
```

containerd 相关：

```
containerd              # 容器运行时
containerd-shim         # shim 的功能之后介绍
ctr                     # containerd cli
```

runc 相关：

```
runc                    # OCI 运行时
```

在确认了 Docker 实际安装了哪些东西之后，我们看看如何将它们组织起来。

在前面内容中，我们已经知道了 docker 和 dockerd 分别是 Docker 的 CLI 和 Daemon，本篇中就暂且跳过；我们看看比较大块的两个东西，一个是 containerd，一个是 runc，看看这两部分与 Docker 到底有什么关系。

Docker 的核心组件

containerd

containerd 最初是从 Docker 中拆分出来的，后来被 Docker 捐给了 CNCF 基金会，现在已经从 CNCF 毕业，整体发展良好。

这里我们启动个容器来具体看看它和 Docker 之间的关系：

复制

```
/ # docker run --rm -d redis:alpine
8c4c58b96e372cc7dcf7bdab9f0a046bb9136e2139e19972dcceebe3c258adb6
/ # docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
8c4c58b96e37        redis:alpine        "docker-entrypoint.s…"   2 seconds ago
/ # ps -ef |grep containerd
   60 root      0:16 containerd --config /var/run/docker/containerd/containerd.tom
  561 root      0:00 containerd-shim -namespace moby -workdir /var/lib/docker/cont
  634 root      0:00 grep containerd
```

可以看到，启动容器后会出现一个 containerd-shim 的进程。我们使用 ctr 这个 containerd 的 CLI 工具来连接 containerd 进程看看：

复制

```
/ # ctr -n moby -a /var/run/docker/containerd/containerd.sock  container ls
CONTAINER                                                           IMAGE    RUNTI
8c4c58b96e372cc7dcf7bdab9f0a046bb9136e2139e19972dcceebe3c258adb6    -        io.co
```

以上命令是用于查看 containerd 正在运行的容器列表的。从这里可以看出 Docker 在启动容器后，会将其交由 containerd 来执行，或者换句话说， containerd 是 Docker 的运行时。查看其进程树：

复制

```
/ # pstree
-+= 00001 root dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0.0:237
 \-+= 00060 root containerd --config /var/run/docker/containerd/containerd.toml --
   \-+= 00561 root containerd-shim -namespace moby -workdir /var/lib/docker/contai
     \--= 00578 #999 redis-server
```

可以看到是 containerd 调用了 containerd-shim，再之后才真正运行容器。

runc

```
/ # runc --root /run/docker/runtime-runc/moby/ list
ID                                                                 PID         STA
8c4c58b96e372cc7dcf7bdab9f0a046bb9136e2139e19972dcceebe3c258adb6   578         run
```

复制

runc 是在 Docker 与其他公司共同成立 OCI 组织的时候，将自己的 libcontainer 运行时捐献出去发展而来的。我们可以通过上述命令查看当前在运行的容器。

同时，我们还可以使用它与运行时的容器进行交互：

```
/ # runc  --root /run/docker/runtime-runc/moby/ exec 8c4c58b96e372cc7dcf7bdab9f0a0
-cli  ping
PONG
```

复制

你可能会好奇，为何 runc 可以这么顺畅的与这些容器进行交互。

这是因为最终的这些容器，都是由 runc 创建的，而 runc 则是由 containerd-shim 进行调用的。本篇我们先跳过代码，直接通过实践来看。我们直接使用 containerd 来启动一个容器：

复制

```
# 先下载镜像
/ # ctr -n moby -a /var/run/docker/containerd/containerd.sock image pull docker.io
docker.io/library/redis:alpine:
index-sha256:ee13953704783b284c080b5b0abe4620730728054f5c19e9488d7a97ecd312c5:
manifest-sha256:ae9dd3bbe42bf13bc318af4af2842b323465312392b96d44893895e8a0438565:
layer-sha256:b2eb22a0b7db31a2b1e2b105bf445ef69f2b80a0957cc66d9d27ca32ef9dc8e8:
layer-sha256:60027bdc030cbd93c908afd40e3ff420f18c77247e59753e57427263bdc84ef5:
layer-sha256:592c37d15428bbf75740a87ea79d97e07aac0e7945ff2c2c9f191d3cb0572982:
layer-sha256:b70a614994bf61cd50e30f4a5539943ea7839d5508434bd5fcf734179bb4f990:
layer-sha256:c5ccbdf10203a49131c170f17a9aea9fed5b9b13b745ffbdb92e31586804050f:
config-sha256:a49ff3e0d85f0b60ddf225db3c134ed1735a3385d9cc617457b21875673da2f0:
layer-sha256:89d9c30c1d48bac627e5c6cb0d1ed1eec28e7dbdfbcc04712e4c79c0f83faf17:
elapsed: 3.0 s
unpacking linux/amd64 sha256:ee13953704783b284c080b5b0abe4620730728054f5c19e9488d7
done

# 启动容器
/ # ctr -n moby -a /var/run/docker/containerd/containerd.sock run -d   --rm docker

# 查看进程树
/ # pstree
-+= 00001 root dockerd --host=unix:///var/run/docker.sock --host=tcp://0.0.0.0:237
 \-+= 00060 root containerd --config /var/run/docker/containerd/containerd.toml --
   |-+= 01150 root containerd-shim -namespace moby -workdir /var/lib/docker/contai
   | \--= 01166 #999 redis-server
   \-+= 00561 root containerd-shim -namespace moby -workdir /var/lib/docker/contai
     \--= 00578 #999 redis-server
```

可以看到，容器中的进程是 containerd-shim 的子进程，也验证了我们前面的说法。

总结

本篇，我为你从较上层的角度来介绍了 Docker 相关的组件，主要是介绍了 Docker 启动容器时，调用的 containerd、containerd-shim 及 runc。

由于本篇是首次对 Docker 相关组件的介绍，所以没有涉及到源码相关的内容，主要是通过实践来证明的。下一篇，我会为你介绍本篇尚未介绍的其他相关组件，会涉及到部分 Docker 的源码。再之后一篇，会为你从整体上将所有这些组件组织起来，带你了解容器完整的生命周期中，这些组件是如何分工协作的。