Docker

Akhia[1]

2020 年 10 月 7 日

[1]E-mail:akhialomgir362856@gmail.com

目录

1 基础 **3**

1.1 简介 4

1.2 CGroup 4

1.3 namespace 4

1.4 unionFS 5

1.4.1 boot file system 6

1.4.2 root file system 6

Chapter 1

基础

1.1 简介

Docker是容器化技术的具体技术实现之一，采用go语言开发。容器本质上讲就是运行在操作系统上的一个进程，只不过加入了对资源的隔离和限制。而Docker是基于容器的这个设计思想，基于linux Container技术实现的核心管理引擎。为什么资源的隔离和限制在云时代更加重要？在默认情况下，一个操作系统里所有运行的进程共享CPU和内存资源，如果程序设计不当，某进程出现死循环可能会耗尽CPU资源，或者由于内存泄漏消耗掉大部分系统资源，所以进程的资源隔离技术是非常必要的。 Docker 不是一项新的技术发明，linux操作系统本身从操作系统层面就支持虚拟化技术，叫做 linux container，也就是大家到处能看到的LXC的全称。

lXC的三大特色：**CGroup**，**namespace** 和 **unionFS**。

1.2 CGroup

CGroups 全称 control group，用来限定一个进程的资源使用，由 linux 内核支持，可以限制和隔离linux进程组 (process groups) 所使用的物理资源，比如**cpu，内存，磁盘和网络IO**，是 linux container 技术的物理基础。

1.3 namespace

另一个维度的资源隔离技术，大家可以把这个概念和我们熟悉的 C++ 里的 namespace 相对照。如果 CGroup 设计出来的目的是为了隔离上面描述的**物理资源**，那么 namespace 则用来隔离 **PID(进程ID), IPC, network 等系统资源**。

容器分配给特定的 namespace，每个 namespace 里面的资源对其他 namespace 都是透明的。不同 container 内的进程属于不同的 namespace，彼此透明，互不干扰。

我们能够为 UID = n 的用户，虚拟化一个 namespace 出来，在这个 namespace 里面，该用户具备 root 权限，但是在宿主机上，该 UID =n 的用户还是一个普通用户，也感知不到自己其实不是一个真的 root 用户这件

事。同样的方式可以通过 namespace 虚拟化进程树。在每一个 namespace 内部，每一个用户都拥有一个属于自己的 init 进程， pid = 1，对于该用户来说，仿佛他独占一台物理的 linux 服务器。

对于每一个命名空间，从用户看起来，应该像一台单独的linux计算机一样，有自己的init进程(PID为1)，其他进程的PID依次递增，A和B空间都有PID为1的init进程，子容器的进程映射到父容器的进程上，父容器可以知道每一个子容器的运行状态，而子容器与子容器之间是隔离的。从图中我们可以看到，进程3在父命名空间里面PID 为3，但是在子命名空间内，他就是1.也就是说用户从子命名空间 A 内看进程3就像 init 进程一样，以为这个进程是自己的初始化进程，但是从整个 host 来看，他其实只是3号进程虚拟化出来的一个空间而已。

父容器有两个子容器，父容器的命名空间里有两个进程， id 分别为 3 和 4, 映射到两个子命名空间后，分别成为其 init 进程，这样命名空间A和B的用户都认为自己独占整台服务器。

linux操作系统到目前为止支持的六种namespace：

1. mnt(mount points, filesystems)
2. pid(process)
3. net(network stack)
4. ipc(System V IPC)
5. uts(hostname)
6. user(UIDs)

1.4 unionFS

unionFS 可以把文件系统上多个目录内容联合挂载到同一个目录下，而目录的物理位置是分开的。要理解 unionFS，我们首先要认识 **bootfs** 和 **rootfs**。

1.4.1 boot file system

（bootfs）：包含操作系统boot loader 和 kernel。用户不会修改这个文件系统。一旦启动完成后，整个linux内核加载进内存，之后bootfs会被卸载掉，从而释放出内存。同样内核版本的不同的 linux 发行版，其bootfs都是一致的。

1.4.2 root file system

（rootfs）：包含典型的目录结构，包括 /dev, /proc, /bin, /etc, /lib, /usr, and /tmp

就是我下面这张图里的这些文件夹：等再加上要运行用户应用所需要的所有配置文件，二进制文件和库文件。这个文件系统在不同的 linux 发行版中是不同的。而且用户可以对这个文件进行修改。 linux 系统在启动时，roofs 首先会被挂载为只读模式，然后在启动完成后被修改为读写模式，随后它们就可以被修改了。不同的 linux 版本，实现 unionFS 的技术可能不一样，使用命令 docker info 查看。

实际上这就是 Docker 容器镜像分层实现的技术基础。如果我们浏览 Docker hub，能发现大多数镜像都不是从头开始制作，而是从一些 base 镜像基础上创建，比如 debian 基础镜像。而新镜像就是从基础镜像上一层层叠加新的逻辑构成的。这种分层设计，一个优点就是资源共享。

想象这样一个场景，一台宿主机上运行了100个基于 debian base 镜像的容器，难道每个容器里都有一份重复的 debian 拷贝呢？这显然不合理；借助 linux 的 unionFS，宿主机只需要在磁盘上保存一份 base 镜像，内存中也只需要加载一份，就能被所有基于这个镜像的容器共享。当某个容器修改了基础镜像的内容，比如 /bin文件夹下的文件，这时其他容器的/bin文件夹是否会发生变化呢？根据容器镜像的写时拷贝技术，某个容器对基础镜像的修改会被限制在单个容器内。这就是我们接下来要学习的容器 Copy-on-Write 特性。

容器镜像由多个镜像层组成，所有镜像层会联合在一起组成一个统一的文件系统。如果不同层中有一个相同路径的文件，比如 /text，上层的 /text 会覆盖下层的 /text，也就是说用户只能访问到上层中的文件 /text。

假设我有如下这个 dockerfile：

```
FROM debian
RUn apt-get install emacs
RUn apt-get install apache2
CMD "/bin/bash"
```

执行docker build . 生成的容器镜像如下：

当用docker run启动这个容器时，实际上在镜像的顶部添加了一个新的可写层。这个可写层也叫容器层。容器启动后，其内的应用所有对容器的改动，文件的增删改操作都只会发生在容器层中，对容器层下面的所有只读镜像层没有影响。