|  |  |
| --- | --- |
|  |  |
|  |

**课 程 报 告**

**容器技术原理与平台研究**

**李斌**

V1.0

**青岛大学计算机科学技术学院**

2021年 6月 14日

**前言**

在电子计算机刚出现时，由于硬件成本高昂，人们试图寻找能够多用户共享计算资源的方式，以提高资源利用率和降低成本。在20世纪60年代，基于硬件技术的主机虚拟化技术出现了。一台物理主机可以被划分为若干个小的机器，每个机器的硬件互不共享，并可以安装各自的操作系统来使用。20世纪90年代后期，X86架构的硬件虚拟化技术逐渐兴起，可在同一台物理机上隔离多个操作系统实例，带来了很多的优点，目前绝大多数的数据中心都采用了硬件虚拟化技术。

虽然硬件虚拟化提供了分隔资源的能力，但是采用虚拟机方式隔离应用程序时，效率往往较低，毕竟还要在每个虚拟机中安装或复制一个操作系统实例，然后把应用部署到其中。因此人们探索出一种更轻量的方案——操作系统虚拟化，使面向应用的管理更便捷。所谓操作系统虚拟化，就是由操作系统创建虚拟的系统环境，使应用感知不到其他应用的存在，仿佛在独自占有全部的系统资源，从而实现应用隔离的目的。在这种方式中不需要虚拟机，也能够实现应用彼此隔离，由于应用是共享同一个操作系统实例的，因此比虚拟机更节省资源，性能更好。操作系统虚拟化在不少系统里面也被称为容器（Container），下面也会以容器来指代操作系统虚拟化。

**一．概述**

**1.1基本概念(技术指标和术语)**

容器的实现原理，主要包括命名空间（Namespace）和控制组

**命名空间**

命名空间是Linux操作系统内核的一种资源隔离方式，使不同的进程具有不同的系统视图。系统视图就是进程能够感知到的系统环境，如主机名、文件系统、网络协议栈、其他用户和进程等。使用命名空间后，每个进程都具备独立的系统环境，进程间彼此感觉不到对方的存在，进程之间相互隔离。目前，Linux中的命名空间共有6种，可以嵌套使用。

**控制组**

命名空间实现了进程隔离功能，但由于各个命名空间中的进程仍然共享同样的系统资源，如CPU、磁盘I/O、内存等，所以如果某个进程长时间占用某些资源，其他命名空间里的进程就会受到影响，这就是“吵闹的邻居（noisy neighbors）”现象。因此，命名空间并没有完全达到进程隔离的目的。为此，Linux内核提供了控制组（Control Groups，cgroups）功能来处理这个问题。

Linux把进程分成控制组，给每组里的进程都设定资源使用规则和限制。在发生资源竞争时，系统会根据每个组的定义，按照比例在控制组之间分配资源。控制组可设定规则的资源包括CPU、内存、磁盘I/O和网络等。通过这种方式，就不会出现某些进程无限度抢占其他进程资源的情况。

Linux系统通过命名空间设置进程的可见且可用资源，通过控制组规定进程对资源的使用量，这样隔离进程的虚拟环境（即容器）就建立起来了。

**1.2技术发展简介**

**1.2.1技术发展历史**

操作系统虚拟化最早出现在2000年，FreeBSD 4.0推出了Jail。Jail加强和改进了用于文件系统隔离的chroot环境。到了2004年，Sun公司发布了Solaris 10的Containers，包括Zones和Resource management两部分。Zones实现了命名空间隔离和安全访问控制，Resource management实现了资源分配控制。2007年，Control Groups（简称cgroups）进入Linux内核，可以限定和隔离一组进程所使用的资源（包括CPU、内存、I/O和网络等）。

2013年，Docker公司发布Docker开源项目，提供了一系列简便的工具链来使用容器。毫不夸张地说，Docker公司率先点燃了容器技术的火焰，拉开了云原生应用变革的帷幕，促进容器生态圈一日千里地发展。截至2020年，Docker Hub中的镜像累计下载了1300亿次，用户创建了约600万个容器镜像库。从这些数据可以看到，用户正在以惊人的速度从传统模式切换到基于容器的应用发布和运维模式。

2015年，OCI（Open Container Initiative）作为Linux基金会项目成立，旨在推动开源技术社区制定容器镜像和运行时规范，使不同厂家的容器解决方案具备互操作能力。同年还成立了CNCF，目的是促进容器技术在云原生领域的应用，降低用户开发云原生应用的门槛。创始会员包括谷歌、红帽、Docker、VMware等多家公司和组织。

CNCF成立之初只有一个开源项目，就是后来大名鼎鼎的Kubernetes。Kubernetes是一个容器应用的编排工具，最早由谷歌的团队研发，后来开源并捐赠给了CNCF成为种子项目。由于Kubernetes是厂家中立的开源项目，开源后得到了社区用户和开发者的广泛参与和支持。到了2018年，Kubernetes已成为容器编排领域事实上的标准，并成为首个CNCF的毕业（graduated）项目。2020年8月，CNCF旗下的开源项目增加到了63个，包括原创于中国的Harbor等项目。

从容器的发展历程可以看到，容器在出现的早期并没有得到人们的广泛关注，主要原因是当时开放的云计算环境还没出现或者未成为主流。2010年之后，随着IaaS、PaaS和SaaS等云平台逐渐成熟，用户对云端应用开发、部署和运维的效率不断重视，重新发掘了容器的价值，最终促成了容器技术的盛行。

**1.2.2技术发展现状**

1） Dokcer 历经5年发展，公司揭露了今年新的Docker年度数据报告，从2013年3月PyCon大会上，Docker首度亮相之后，至今在Docker上的容器镜像下载次数已经超过了370亿次，容器化的应用有高达350万个。

2）目前在LinkedIn网站上的Docker相关职缺也有15,000个。全球活跃的Docker使用者社群已有200多个，包括台湾也有。全球使用企业版Docker EE的企业顾客目前则约有450家。

3）Docker功能的进展不多，主要有拥抱Kubernetes，在Docker产品中可以和Swarm并用。其次重要的新功能是增加了RBAC角色存取控管机制，这也是企业想要的安全机制。

**1.2.3技术发展未来趋势**

**更多更好的Windows容器**

用于Windows的原生Docker容器自从2016年出现就存在有较大问题。从那时起，业界对Windows容器的热情可能就已经降低了，我认为部分原因在于Windows容器使用起来不如 [Linux](https://www.linuxprobe.com/) 容器那样令人激动了。  
但是我也不认为Windows容器已经停止发展了，展望未来，尤其是在推出Windows新的服务器版本时，我预计微软在自己的操作系统是要持续支持原生容器以及加强对使用基于Windows集群的Kubernetes管理的支持。

**更多的运行时**

Docker是使容器成名的运行时。但是自从Docker亮相以来，也已经出现了许多其它的容器运行时。随着越来越多的组织为了满足特定的需求而推出自己的运行时，预计这个列表将会继续增长。尽管如此，我还是认为大多数工作负载将继续依赖于如今占主导地位的运行时，譬如Docker和Containerd。

**Kubernetes失去了发展动力**

Kubernetes的受欢迎程度是毋庸置疑的，但随着越来越多的Kubernetes应用，我怀疑这个平台将不再像现在这样酷了，毕竟，开发人员曾认为Heroku是有史以来最伟大的应用，云化虚拟机可以解决地球上的所有问题。现如今这些技术虽然仍被广泛使用，但是看起来已经变得平凡无奇了。当然，这并不是说Kubernetes将被另一种编排机制所代替。我当然不认为Kubernetes会消失，我只是不认为每个人对五年后的Kubernetes仍然会感到兴奋。顺便提一下，关于服务网格，我也是持同样的看法

**容器安全变成商品**

出于种种原因，容器需要一种新的安全解决方法。截至目前，容器安全及其特有的挑战一直是IT安全人员讨论的焦点 ，更不用说一些公司了。这有很好的理由，要知道，容器安全是件很特别的事情。但是，如果容器需要扩展，则必须通过更广泛的安全流程来解决其安全需求。换句话说，我们必须使用相同的系统和策略来保护我们用于一切的容器。要做到这样，容器安全必须商品化。未来我怀疑将不会再关注集装箱安全的特殊性，或者不再去寻求主要用于保护容器安全的平台。相反的是，我们将容器集成到了其他的安全工作流程中去了。

**二.技术资源**

**2.1研究机构**

dotCloud，[中国科学院自动化研究所](https://www.liepin.com/company/8809978/" \o "中国科学院自动化研究所" \t "/Users/blackhole6/Documents\\x/_blank)，SUN公司，SWsoft公司，CloudFoundry

**2.2国内外厂商**

华为云，阿里云，腾讯云，京东云，博云

**2.3国内外网站、会议与技术刊物**

会议：DockerCon

1. **关键技术**

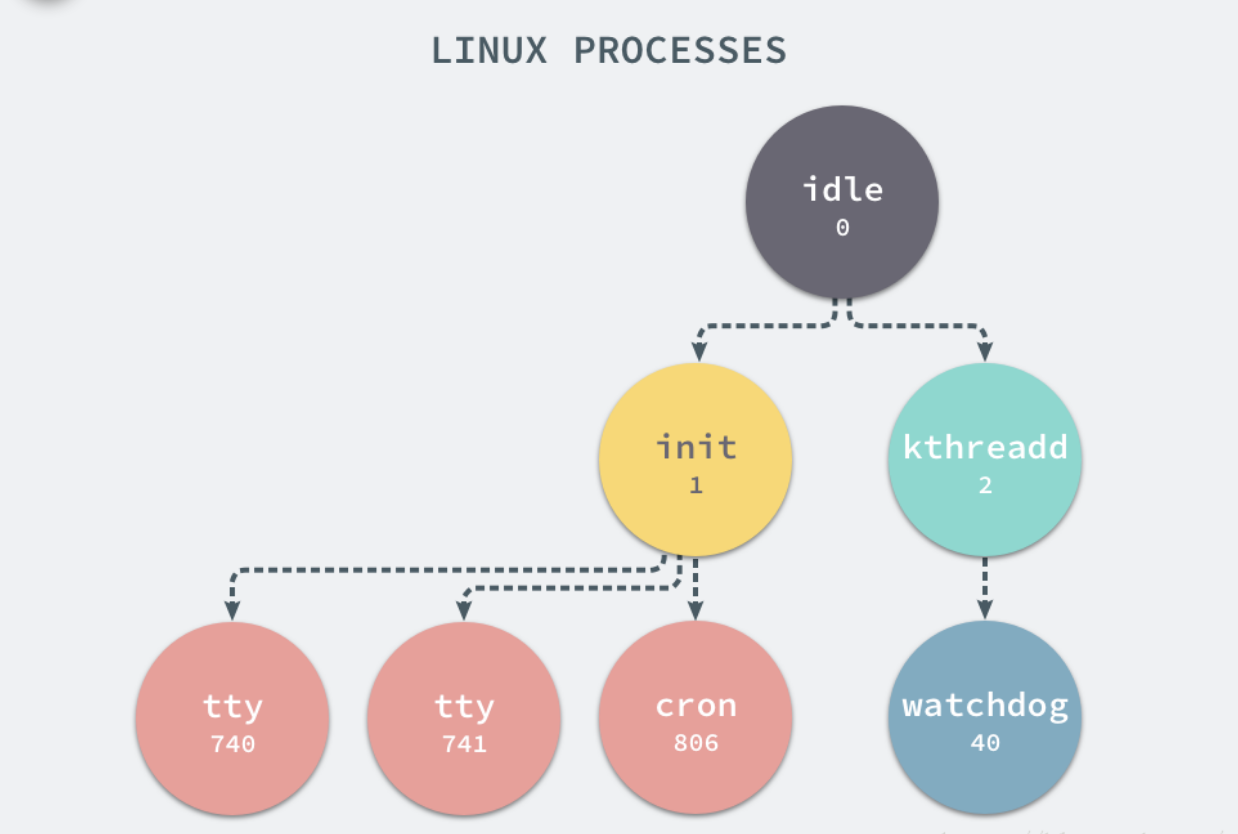
**3.1Namespaces**

命名空间可以有效地帮助Docker分离进程树、网络接口、挂载点以及进程间通信等资源。Linux 的命名空间机制提供了以下七种不同的命名空间，包括 CLONE\_NEWCGROUP、CLONE\_NEWIPC、CLONE\_NEWNET、CLONE\_NEWNS、CLONE\_NEWPID、CLONE\_NEWUSER 和 CLONE\_NEWUTS，通过这七个选项我们能在创建新的进程时设置新进程应该在哪些资源上与宿主机器进行隔离。

## **3.2进程**

1. $ ps -ef
2. UID        PID  PPID  C STIME TTY          TIME CMD
3. root         1     0  0 Apr08 ?        00:00:09 /sbin/init
4. root         2     0  0 Apr08 ?        00:00:00 [kthreadd]
5. root         3     2  0 Apr08 ?        00:00:05 [ksoftirqd/0]
6. root         5     2  0 Apr08 ?        00:00:00 [kworker/0:0H]
7. root         7     2  0 Apr08 ?        00:07:10 [rcu\_sched]
8. root        39     2  0 Apr08 ?        00:00:00 [migration/0]

当前机器上有很多的进程正在执行，在上述进程中有两个非常特殊，一个是 pid 为 1 的 /sbin/init 进程，另一个是 pid 为 2 的 kthreadd 进程，这两个进程都是被 Linux 中的上帝进程 idle 创建出来的，其中前者负责执行内核的一部分初始化工作和系统配置，也会创建一些类似 getty 的注册进程，而后者负责管理和调度其他的内核进程。

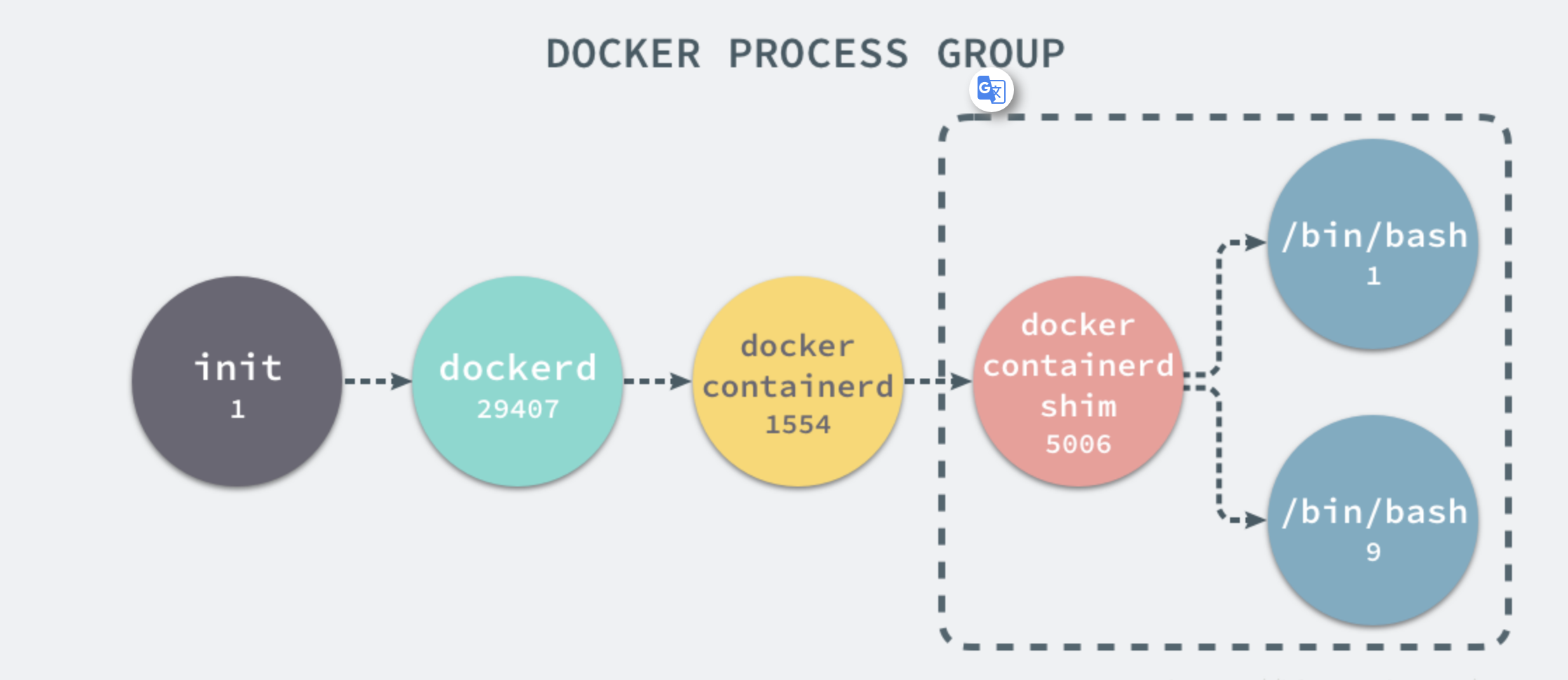


如果我们在当前的 Linux 操作系统下运行一个新的 Docker 容器，并通过 exec 进入其内部的 bash 并打印其中的全部进程，我们会得到以下的结果：

1. root@iZ255w13cy6Z:~# docker run -it -d ubuntu
2. b809a2eb3630e64c581561b08ac46154878ff1c61c6519848b4a29d412215e79
3. root@iZ255w13cy6Z:~# docker exec -it b809a2eb3630 /bin/bash
4. root@b809a2eb3630:/# ps -ef
5. UID        PID  PPID  C STIME TTY          TIME CMD
6. root         1     0  0 15:42 pts/0    00:00:00 /bin/bash
7. root         9     0  0 15:42 pts/1    00:00:00 /bin/bash
8. root        17     9  0 15:43 pts/1    00:00:00 ps -ef

当前的 Docker 容器成功将容器内的进程与宿主机器中的进程隔离，如果我们在宿主机器上打印当前的全部进程时，会得到下面三条与 Docker 相关的结果：

在当前的宿主机器上，可能就存在由上述的不同进程构成的进程树：



这就是在使用 clone(2) 创建新进程时传入 CLONE\_NEWPID 实现的，也就是使用 Linux 的命名空间实现进程的隔离，Docker 容器内部的任意进程都对宿主机器的进程一无所知。

1. containerRouter.postContainersStart
2. └── daemon.ContainerStart
3. └── daemon.createSpec
4. └── setNamespaces
5. └── setNamespace

Docker 的容器就是使用上述技术实现与宿主机器的进程隔离，当我们每次运行 docker run 或者 docker start 时，都会在下面的方法中创建一个用于设置进程间隔离的 Spec：

1. func (daemon \*Daemon) createSpec(c \*container.Container) (\*specs.Spec, error) {
2. s := oci.DefaultSpec()
3. // ...
4. if err := setNamespaces(daemon, &s, c); err != nil {
5. return nil, fmt.Errorf("linux spec namespaces: %v", err)
6. }
7. return &s, nil
8. }

**3.3网络**

如果 Docker 的容器通过 Linux 的命名空间完成了与宿主机进程的网络隔离，但是却有没有办法通过宿主机的网络与整个互联网相连，就会产生很多限制，所以 Docker 虽然可以通过命名空间创建一个隔离的网络环境，但是 Docker 中的服务仍然需要与外界相连才能发挥作用。每一个使用 docker run 启动的容器其实都具有单独的网络命名空间，Docker 为我们提供了四种不同的网络模式，Host、Container、None 和 Bridge 模式。

1. **技术难点试验**

## **4.1Docker的部署**

1）下载所需软件：

1.docker-ce.x86、 docker-ce-cli.x86\_64、 containerd.io.x86、container-selinux.noarch

软件包下载： https://mirros.aliyun.com/(阿里云开源镜像网站)

解决软件依赖性：pkgs.org

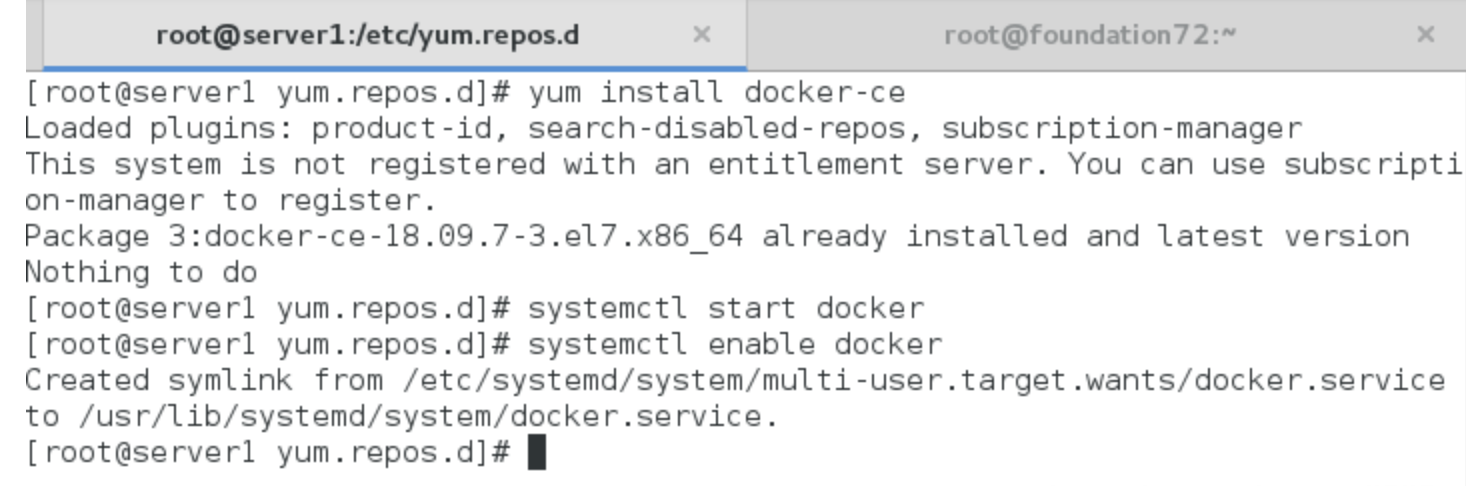
2）将下载好的软件统一放在一个文件夹内，配置docker 的yum仓库

1. vim /etc/yum.repos.d/docker.repo
2. [docker]
3. name=docker-ce
4. baseurl=http://172.25.33.250/pub/docs/docker
5. gpgcheck=0



3）下载docker，并开启

1. yum install docker-ce
2. systemctl start docker
3. systemctl enable docker



1. 下载好docker，查看具体信息

系认会自动产生一个docker网络接口

1. [root@server1 yum.repos.d]# ip addr show
2. 1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
3. link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4. inet 127.0.0.1/8 scope host lo
5. valid\_lft forever preferred\_lft forever
6. inet6 ::1/128 scope host
7. valid\_lft forever preferred\_lft forever
8. 2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP group default qlen 1000
9. link/ether 52:54:00:3b:fd:24 brd ff:ff:ff:ff:ff:ff
10. inet 172.25.33.1/24 brd 172.25.33.255 scope global eth0
11. valid\_lft forever preferred\_lft forever
12. inet6 fe80::5054:ff:fe3b:fd24/64 scope link
13. valid\_lft forever preferred\_lft forever
14. 3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
15. link/ether 02:42:43:ae:3a:71 brd ff:ff:ff:ff:ff:ff
16. inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
17. valid\_lft forever preferred\_lft forever

查看docker安装版本信息

1. [root@server1 ~]# docker version
2. Client:
3. Version:           18.09.7
4. API version:       1.39
5. Go version:        go1.10.8
6. Git commit:        2d0083d
7. Built:             Thu Jun 27 17:56:06 2019
8. OS/Arch:           linux/amd64
9. Experimental:      false
10. Server: Docker Engine - Community
11. Engine:
12. Version:          18.09.7
13. API version:      1.39 (minimum version 1.12)
14. Go version:       go1.10.8
15. Git commit:       2d0083d
16. Built:            Thu Jun 27 17:26:28 2019
17. OS/Arch:          linux/amd64
18. Experimental:     false

docker info : 显示 Docker 系统信息，包括镜像和容器数

1. [root@server1 ~]# docker info
2. Containers: 13
3. Running: 9
4. Paused: 0
5. Stopped: 4
6. Images: 21
7. Server Version: 18.09.7
8. Storage Driver: overlay2
9. Backing Filesystem: xfs
10. Supports d\_type: true
11. Native Overlay Diff: true
12. Logging Driver: json-file
13. Cgroup Driver: cgroupfs
14. Plugins:
15. Volume: local
16. Network: bridge host macvlan null overlay
17. Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
18. Swarm: inactive
19. Runtimes: runc
20. Default Runtime: runc
21. Init Binary: docker-init
22. containerd version: bb71b10fd8f58240ca47fbb579b9d1028eea7c84
23. runc version: 2b18fe1d885ee5083ef9f0838fee39b62d653e30
24. init version: fec3683
25. Security Options:
26. seccomp
27. Profile: default
28. Kernel Version: 3.10.0-862.el7.x86\_64
29. Operating System: Red Hat Enterprise Linux Server 7.5 (Maipo)
30. OSType: linux
31. Architecture: x86\_64
32. CPUs: 1
33. Total Memory: 1.953GiB
34. Name: server1
35. ID: LWQC:RSPE:2FRR:YDYZ:NO4V:O4RL:KPS6:BCYH:Z6LC:QM6E:7UBS:ARM6
36. Docker Root Dir: /var/lib/docker
37. Debug Mode (client): false
38. Debug Mode (server): false
39. Registry: https://index.docker.io/v1/
40. Labels:
41. Experimental: false
42. Insecure Registries:
43. 172.25.33.1
44. 127.0.0.0/8
45. Live Restore Enabled: false
46. Product License: Community Engine

docker常用命令

1. docker load -i 镜像名                       *#导入镜像*
2. docker run -i *--name 容器名  镜像名         #创建容器以镜像为模版*
3. docker ps                                  *#查看运行的容器状态*
4. docker ps -a                              *#查看容器状态（包或不活跃的容器）*
5. docker attach vm1                         *#连接容器*
6. docker top vm1                            *#查看容器进程*
7. docker logs vm1                           *#查看容器指令输出  -f参数可以实时查看*
8. docker inspect vm1                        *# 查看容器详情*
9. docker stats  vm1                         *#查看容器资源使用率*
10. docker diff vm1                           *#查看容器修改*
11. docker stop vm1                           *#停止容器*
12. docker start vm1                          *# 启动容器*
13. docker kill vm1                           *#强制杀死容器进程*
14. docker restart vm1                        *#重启容器*
15. docker rm   vm1                           *#删除容器*
16. docker pause/uppause  vm1                 *#暂停/恢复容器*

docker 对镜像的所有操作命令

1. docker load -i 镜像名                       *#导入镜像*
2. docker run -i *--name 容器名  镜像名         #创建容器以镜像为模版*
3. docker ps                                  *#查看运行的容器状态*
4. docker ps -a                              *#查看容器状态（包或不活跃的容器）*
5. docker attach vm1                         *#连接容器*
6. docker top vm1                            *#查看容器进程*
7. docker logs vm1                           *#查看容器指令输出  -f参数可以实时查看*
8. docker inspect vm1                        *# 查看容器详情*
9. docker stats  vm1                         *#查看容器资源使用率*
10. docker diff vm1                           *#查看容器修改*
11. docker stop vm1                           *#停止容器*
12. docker start vm1                          *# 启动容器*
13. docker kill vm1                           *#强制杀死容器进程*
14. docker restart vm1                        *#重启容器*
15. docker rm   vm1                           *#删除容器*
16. docker pause/uppause  vm1                 *#暂停/恢复容器*

docker 对容器的所有操作命令

1. [root@server1 ~]# docker container
2. Usage: docker container COMMAND
3. Manage containers
4. Commands:
5. attach      Attach local standard input, output, and error streams to a running container
6. commit      Create a new image from a container's changes
7. cp          Copy files/folders between a container and the local filesystem
8. create      Create a new container
9. diff        Inspect changes to files or directories on a container's filesystem
10. exec        Run a command in a running container
11. export      Export a container's filesystem as a tar archive
12. inspect     Display detailed information on one or more containers
13. kill        Kill one or more running containers
14. logs        Fetch the logs of a container
15. ls          List containers
16. pause       Pause all processes within one or more containers
17. port        List port mappings or a specific mapping for the container
18. prune       Remove all stopped containers
19. rename      Rename a container
20. restart     Restart one or more containers
21. rm          Remove one or more containers
22. run         Run a command in a new container
23. start       Start one or more stopped containers
24. stats       Display a live stream of container(s) resource usage statistics
25. stop        Stop one or more running containers
26. top         Display the running processes of a container
27. unpause     Unpause all processes within one or more containers
28. update      Update configuration of one or more containers
29. wait        Block until one or more containers stop, then print their exit codes

docker 对volume的所有操作命令

1. [root@server1 ~]# docker volume
2. Usage: docker volume COMMAND
3. Manage volumes
4. Commands:
5. create      Create a volume
6. inspect     Display detailed information on one or more volumes
7. ls          List volumes
8. prune       Remove all unused local volumes
9. rm          Remove one or more volumes

## **4.2Docker的运行**

以game2048为例

1）获取game2048镜像，并导入：

1. [root@server1 ~]# docker load -i game2048.tar    *##导入镜像*
2. 011b303988d2: Loading layer   5.05MB/5.05MB
3. 36e9226e74f8: Loading layer  51.46MB/51.46MB
4. 192e9fad2abc: Loading layer  3.584kB/3.584kB
5. 6d7504772167: Loading layer  4.608kB/4.608kB
6. 88fca8ae768a: Loading layer  629.8kB/629.8kB
7. Loaded image: game2048:latest
8. [root@server1 ~]# docker images    *##查看存在的镜像*
9. REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
10. game2048            latest              19299002fdbe        2 years ago         55.5MB
11. 创建容器，并进行端口映射

1. [root@server1 ~]# docker run -d --name vm1 -p 80:80 game2048   *##-d打入后台 -p制定端口 宿主机的端口：docker端口*
2. 1d4fab18792eb4faf2147d10b06bd823de7c61f5e9805da72ecaa2fb28567118
3. [root@server1 ~]# docker ps     *##查看docker进程*
4. CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                         NAMES
5. 1d4fab18792e        game2048            "/bin/sh -c 'sed -i …"   15 seconds ago      Up 14 seconds       0.0.0.0:80->80/tcp, 443/tcp   vm1
6. [root@server1 ~]# netstat -tnlp   *##会看到开了一个80端口*
7. Active Internet connections (only servers)
8. Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
9. tcp        0      0 0.0.0.0:22              0.0.0.0:\*               LISTEN      859/sshd
10. tcp        0      0 127.0.0.1:25            0.0.0.0:\*               LISTEN      883/master
11. tcp6       0      0 :::80                   :::\*                    LISTEN      1743/docker-proxy
12. tcp6       0      0 :::22                   :::\*                    LISTEN      859/sshd
13. tcp6       0      0 ::1:25                  :::\*                    LISTEN      883/master



3）浏览器访问：172.25.33.1，运行成功

