容器底层技术揭秘
&
我的自研容器引擎Cocker(C语言)

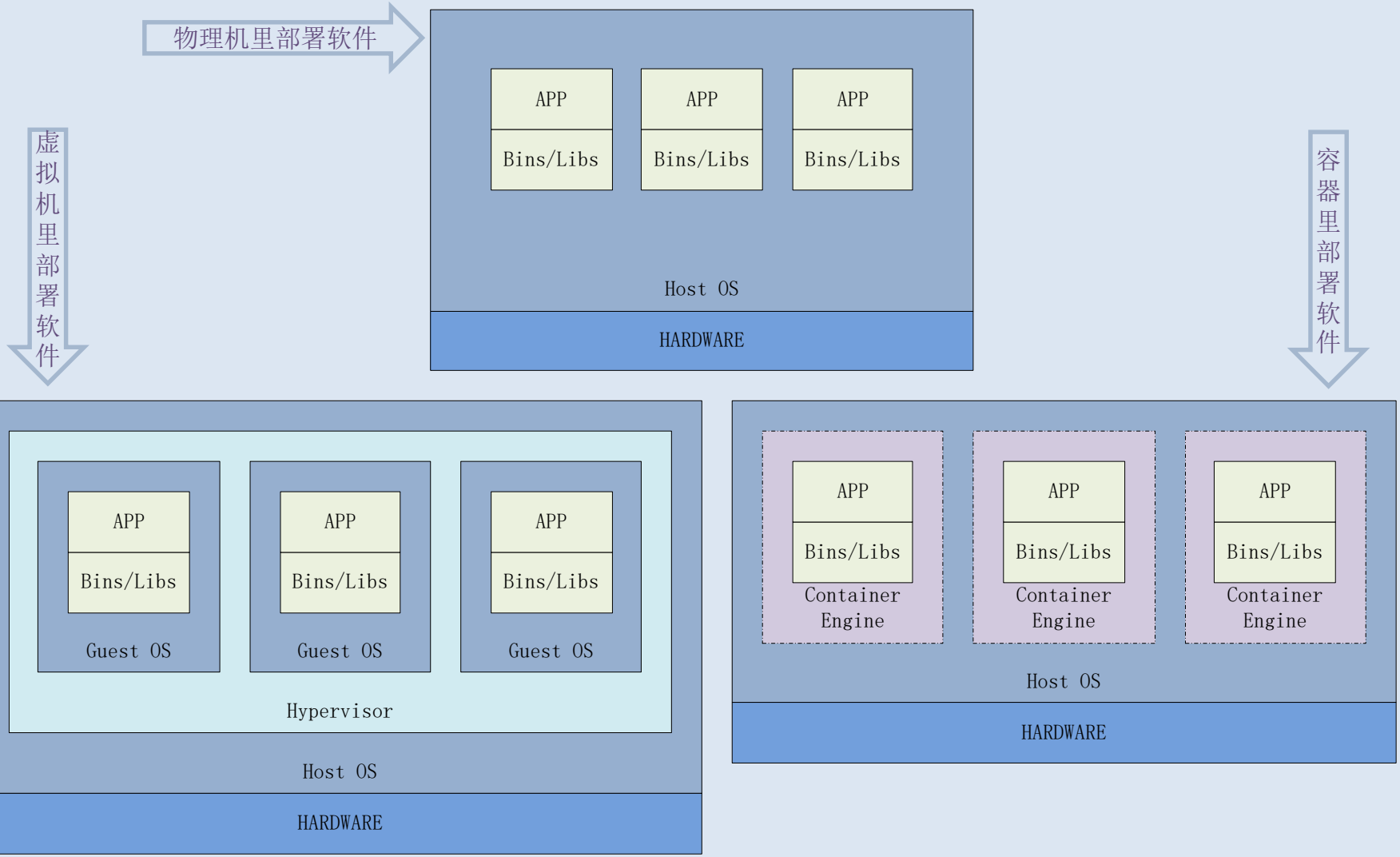
厉华 2018

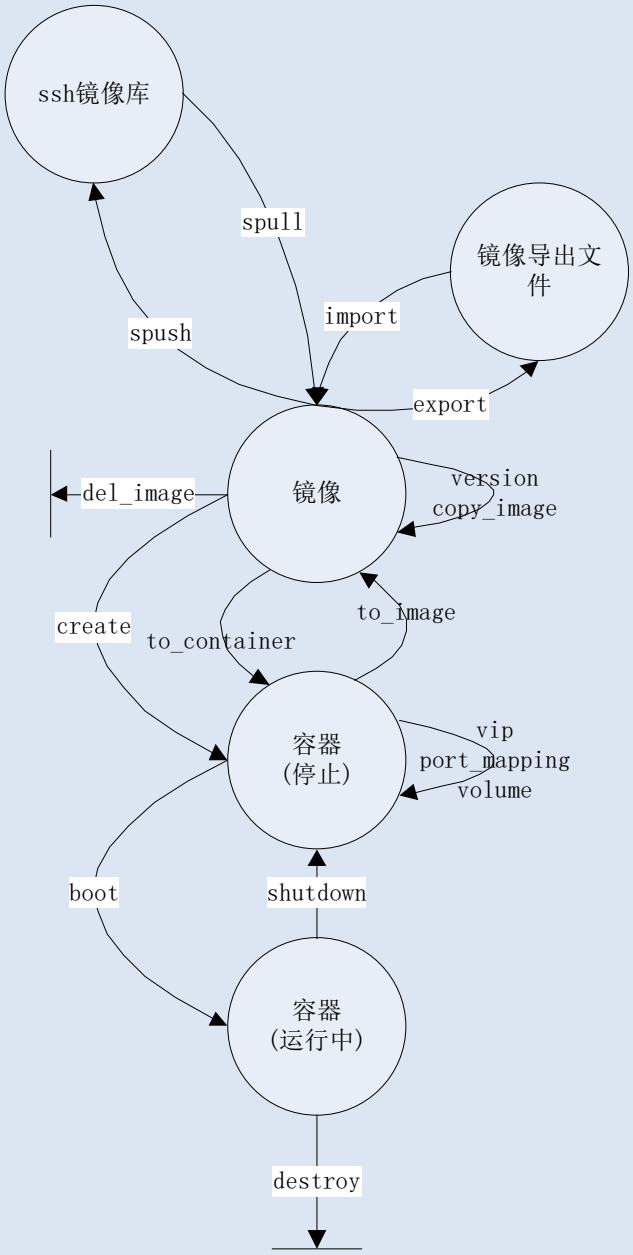
容器是什么？
容器就是将软件所需的完整环境（包括可执行程序、库文件、配置文件等）打包成一个独立包裹，以便于在目标系统中快捷、隔离的部署运行。

镜像是什么？
镜像之于容器，好比类之于对象。

Docker是什么？
是Docker公司推出的Go语言开发实现的容器引擎。

举个栗子：我想装个MySQL玩玩
× 用rpm/yum安装发行版自带的二进制包安装？版本不够新！
× 官网下载最新源代码自己编译？太麻烦！
× 网上别人帮你编译好的最新二进制安装包？万一别人加入了system(“rm –rf /”);你的操作系统就得重装了！
√ 现在流行的做法是，从容器厂商镜像库里pull一个MySQL镜像导入到自己的本地镜像库，通过容器引擎把容器内的MySQL主控进程运行起来即可！
“快捷”体现在运行容器其实就是启动一个进程。
“隔离”体现在容器里的进程、文件系统、IPC、网络等和外面主机是不同“位面”，无法互相影响，无法影响容器外环境。
“隔离”还体现在哪天你不想玩了，停止容器、删除镜像即可，不怕MySQL可执行程序和库文件散落在/bin、/usr/lib等目录中清理不干净。


物理机里部署软件
APP
Bins/Libs
APP
Bins/Libs
APP
Bins/Libs
Host OS
HARDWARE
虚拟机里部署软件
APP
Bins/Libs
Guest OS
APP
Bins/Libs
Guest OS
APP
Bins/Libs
Guest OS
Hypervisor
Host OS
HARDWARE
容器里部署软件
APP
Bins/Libs
Container Engine
APP
Bins/Libs
Container Engine
APP
Bins/Libs
Container Engine
Host OS
HARDWARE


ssh镜像库
spull
镜像导出文件
import
spush
export
镜像
del_image
version
copy_image
create
to_container
to_image
容器
(停止)
vip
port_mapping
volume
boot
shutdown
容器
(运行中)
destroy

*** 一致的运行环境**

容器的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，“这段代码在我机器上没问题啊”理由终结者。

*** 更快速的启动时间**

可以做到秒级、甚至毫秒级的启动时间，启动一个容器本质上讲其实就是启动了一个本地进程。

*** 隔离性**

避免服务器资源/环境受到其他用户/应用的影响。

*** 弹性伸缩，快速扩展**

交易量变大时，挑几个系统压力较轻的服务器，多运行几个容器即可。当交易量变小时…

*** 迁移方便**

把应用打包成独立镜像，可以很轻易的部署到其它相同内核的机器里运行，而不用担心外部运行环境的变化导致应用无法正常运行的情况。

*** 持续交付和部署**

开发完后把容器转换成镜像，丢到测试环境里启动起来，测试完后丢到生产环境里启动起来。别忘了先停止老版本容器哦。

chroot

namespace(IPC,Network,Mount,PID,User,UTS)

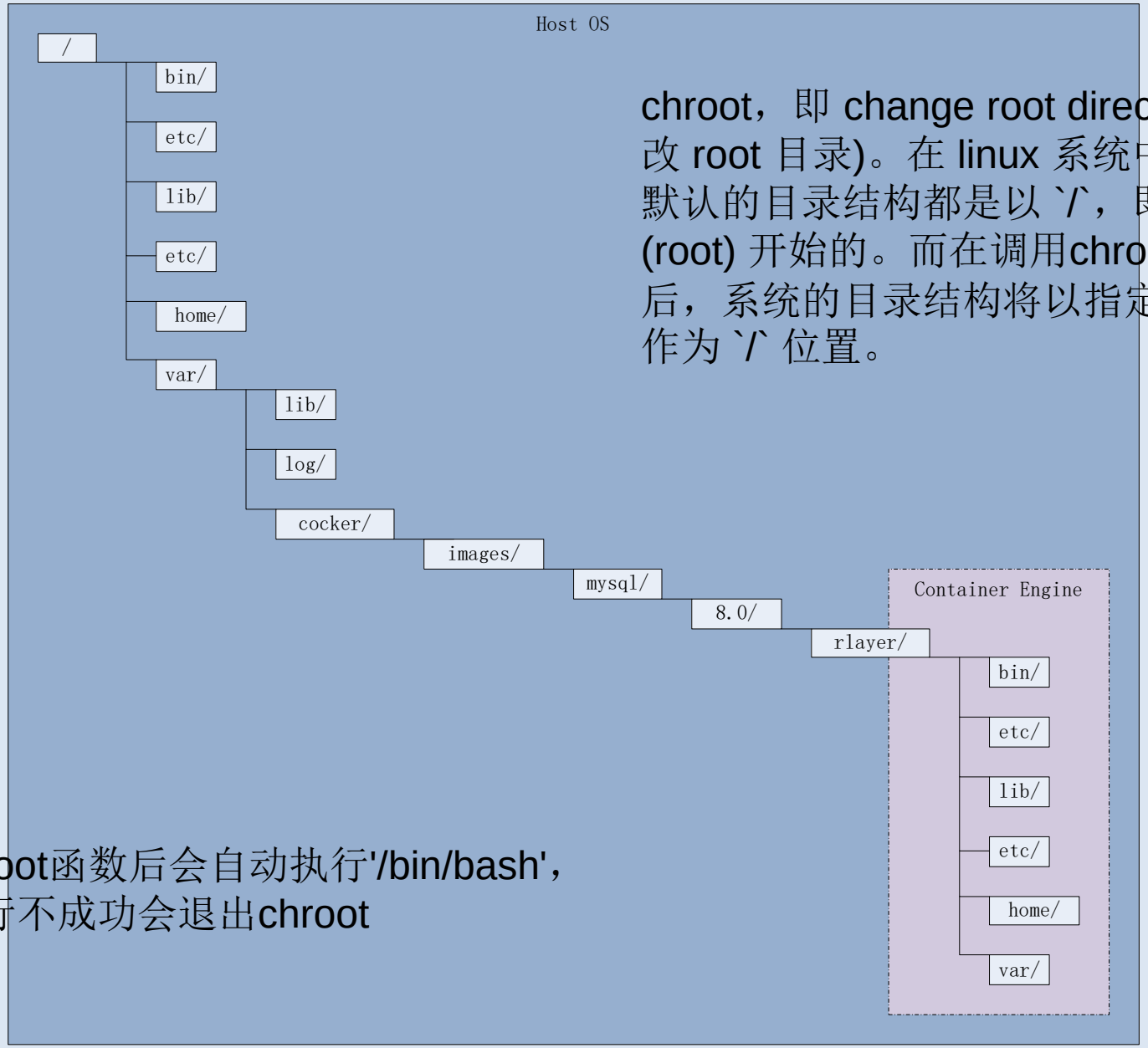
cgroup(CPU,MEM,…)

分层文件系统(overlayfs)与磁盘卷挂接点

伪终端(pts)

网络模型(network bridge,iptables,…)

基础镜像



chroot，即 change root directory (更改 root 目录)。在 linux 系统中，系统默认的目录结构都是以 `/`，即是以根 (root) 开始的。而在调用chroot函数之后，系统的目录结构将以指定的位置作为 `/` 位置。

调用chroot函数后会自动执行'/bin/bash'，
如果执行不成功会退出chroot

114.215.179.129 - SecureCRT

文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

114.215.179.129

```
CHROOT(2)                  Linux Programmer's Manual                 CHROOT(2)

NAME
       chroot - change root directory

SYNOPSIS
       #include <unistd.h>

       int chroot(const char *path);

DESCRIPTION
       chroot()  changes the root directory of the calling process to that specified in path.  This directory will be
       used for pathnames beginning with /.  The root directory is inherited by all children of the calling  process.

       Only a privileged process (Linux: one with the CAP_SYS_CHROOT capability) may call chroot().

       This call changes an ingredient in the pathname resolution process and does nothing else.

       This  call  does  not change the current working directory, so that after the call '.' can be outside the tree
       rooted at '/'.  In particular, the superuser can escape from a "chroot jail" by doing:

           mkdir foo; chroot foo; cd ..

       This call does not close open file descriptors, and such file descriptors may allow access  to  files  outside
       the chroot tree.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.

ERRORS
       Depending on the file system, other errors can be returned.  The more general errors are listed below:

       EACCES Search permission is denied on a component of the path prefix.  (See also path_resolution(7).)

       EFAULT path points outside your accessible address space.

       EIO    An I/O error occurred.

       ELOOP  Too many symbolic links were encountered in resolving path.
:
```

就绪 ssh2: AES-256-CTR 40, 2 40行,131列 Linux 大写 数字

clone() 函数

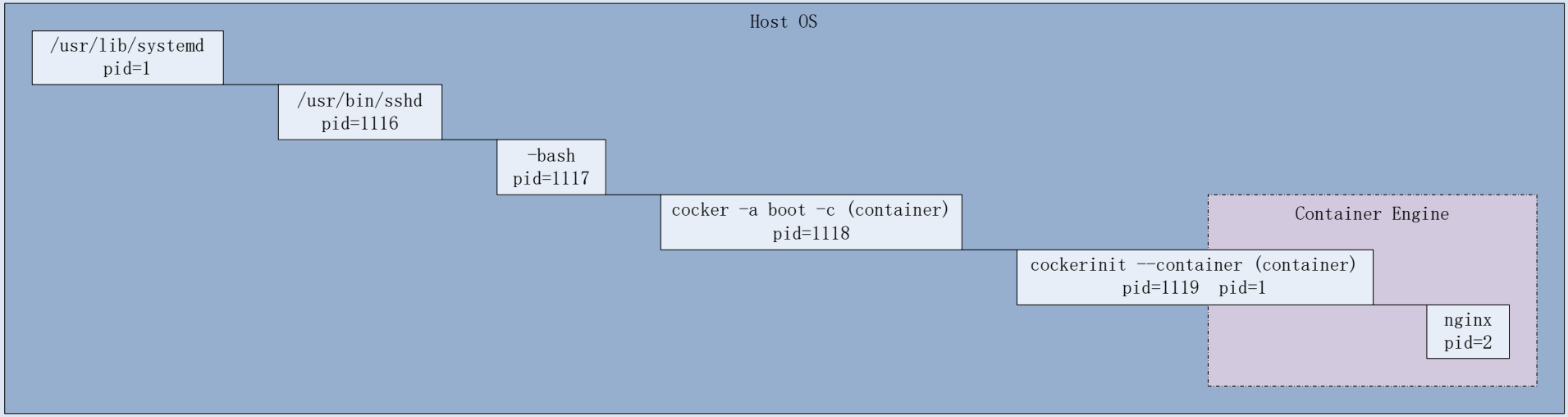
我们可以通过 clone() 在创建新进程的同时创建 namespace。clone() 在 C 语言库中的声明如下：

```
/* Prototype for the glibc wrapper function */
#define _GNU_SOURCE
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

实际上，clone() 是在 C 语言库中定义的一个封装(wrapper)函数，它负责建立新进程的堆栈并且调用对编程者隐藏的 clone() 系统调用。Clone() 其实是 linux 系统调用 fork() 的一种更通用的实现方式，它可以通过 flags 来控制使用多少功能。一共有 20 多种 CLONE_ 开头的 falg(标志位) 参数用来控制 clone 进程的方方面面(比如是否与父进程共享虚拟内存等)，下面我们只介绍与 namespace 相关的 4 个参数：

- fn：指定一个由新进程执行的函数。当这个函数返回时，子进程终止。该函数返回一个整数，表示子进程的退出代码。
- child_stack：传入子进程使用的栈空间，也就是把用户态堆栈指针赋给子进程的 esp 寄存器。调用进程(指调用 clone() 的进程)应该总是为子进程分配新的堆栈。
- flags：表示使用哪些 CLONE_ 开头的标志位，与 namespace 相关的有CLONE_NEWIPC、CLONE_NEWNET、CLONE_NEWNS、CLONE_NEWPID、CLONE_NEWUSER、CLONE_NEWUTS 和 CLONE_NEWCGROUP。
- arg：指向传递给 fn() 函数的参数。

```
int DoAction_boot( struct CockerEnvironment *env )
{
    …
    pid = clone( CloneEntry , stack_bottom+sizeof(stack_bottom) , CLONE_NEWNS | CLONE_NEWPID | CLONE_NEWIPC |
CLONE_NEWUTS | CLONE_NEWNET , (void*)env ) ;
    …
}

static int CloneEntry( void *pv )
{
    struct CockerEnvironment    *env = (struct CockerEnvironment *)pv ;
    …
    nret = chroot( mount_target ) ;
    argc = 0 ;
    argv[argc++] = "/bin/cockerinit" ;
    argv[argc++] = "cockerinit" ;
    argv[argc++] = "--container" ;
    argv[argc++] = env->cmd_para.__container ;
    argv[argc++] = NULL ;
    nret = execv( argv[0] , argv+1 ) ;
    exit(9);
}
```


Host OS
/usr/lib/systemd
pid=1
/usr/bin/sshd
pid=1116
-bash
pid=1117
cocker -a boot -c (container)
pid=1118
Container Engine
cockerinit --container (container)
pid=1119 pid=1
nginx
pid=2

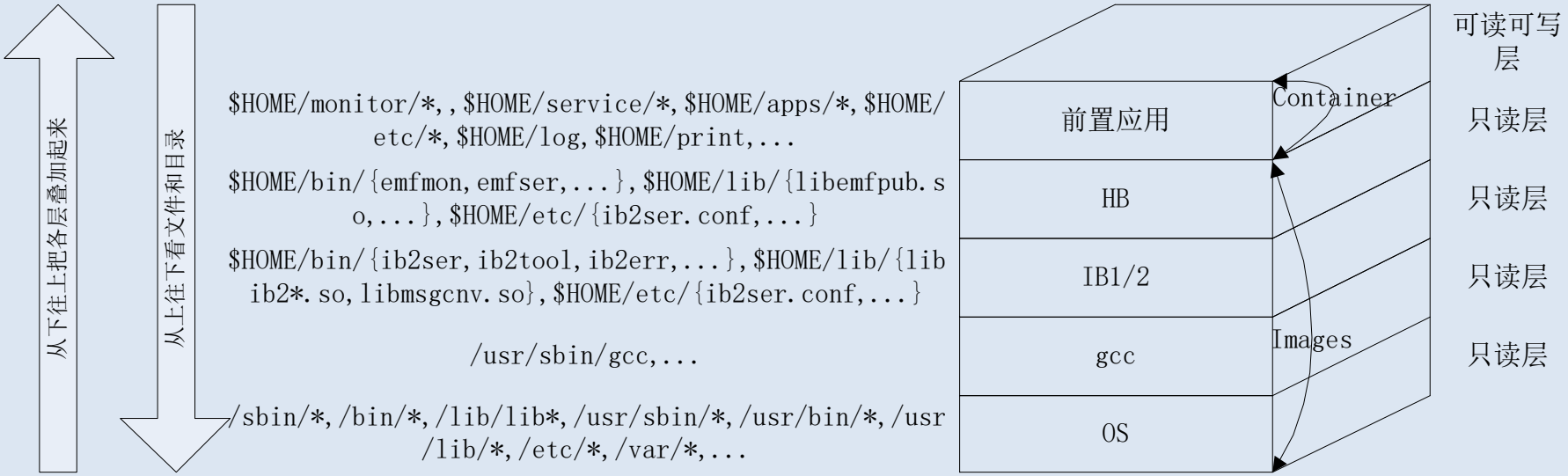
Name	flags in System call ‘clone‘	Remark	Support version
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues	since Linux 2.6.19 rhel6
Network	CLONE_NEWNET	network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net and /sys/class/net directory trees, sockets, etc	since Linux 2.6.24 rhel6
Mount	CLONE_NEWNS	Mount points	since Linux 2.4.19 rhel3
PID	CLONE_NEWPID	Process IDs	since Linux 2.6.24 rhel6
User	CLONE_NEWUSER	User and group IDs	started in Linux 2.6.23 and completed in Linux 3.8 rhel6~rhel7
UTS	CLONE_NEWUTS	Hostname and NIS domain name	since Linux 2.6.19 rhel6

cpus	
/sys/fs/cgroup/cpuset/cocker_${container}/cpuset.cpus	0-1
/sys/fs/cgroup/cpuset/cocker_${container}/cpuset.mem	0-1
/sys/fs/cgroup/cpuset/cocker_${container}/tasks	(cockerinit's pid)

cpu_quota	
/sys/fs/cgroup/cpu,cpuacct/cocker_${container}/cpu.cfs_period_us	1,000,000
/sys/fs/cgroup/cpu,cpuacct/cocker_${container}/cpu.cfs_quota_us	500,000
/sys/fs/cgroup/cpu,cpuacct/cocker_${container}/tasks	(cockerinit's pid)

mem_limit	
/sys/fs/cgroup/memory/cocker_${container}/memory.limit_in_bytes	1024*1024*1024
/sys/fs/cgroup/memory/cocker_${container}/memory.memsw.limit_in_bytes	1024*1024*1024
/sys/fs/cgroup/memory/cocker_${container}/tasks	(cockerinit's pid)

```
static int CloneEntry( void *pv )
{
    struct CockerEnvironment    *env = (struct CockerEnvironment *)pv ;
    …
    if( env->cgroup_enable )
    {
        if( env->cmd_para.__cpus )
        {
            …
            nret = WriteFileLine( env->cmd_para.__cpus , cgroup_cpuset_cpus_file , sizeof(cgroup_cpuset_cpus_file) ,
"%s/cpuset/cocker_%s/cpuset.cpus" , CGROUP_PATH , env->cmd_para.__container ) ;
            …
        }
        if( env->cmd_para.__cpu_quota )
        {
            nret = WriteFileLine( "1000000" , cgroup_cpu_cfs_period_us_file , sizeof(cgroup_cpu_cfs_period_us_file) ,
"%s/cpu,cpuacct/cocker_%s/cpu.cfs_period_us" , CGROUP_PATH , env->cmd_para.__container ) ;
            …
        }
        if( env->cmd_para.__mem_limit )
        {
            …
            nret = WriteFileLine( env->cmd_para.__mem_limit , cgroup_memory_limit_in_bytes_file ,
sizeof(cgroup_memory_limit_in_bytes_file) , "%s/memory/cocker_%s/memory.limit_in_bytes" , CGROUP_PATH , env-
>cmd_para.__container ) ;
            …
        }
    }
    …
    nret = chroot( mount_target ) ;
    …
}
```


从下往上把各层叠加起来
从上往下看文件和目录
$HOME/monitor/*,,$HOME/service/*,$HOME/apps/*,$HOME/etc/*,$HOME/log,$HOME/print,...
$HOME/bin/{emfmon,emfser,...},$HOME/lib/{libemfpub.so,...},$HOME/etc/{ib2ser.conf,...}
$HOME/bin/{ib2ser,ib2tool,ib2err,...},$HOME/lib/{libib2*.so,libmsgcnv.so},$HOME/etc/{ib2ser.conf,...}
/usr/sbin/gcc,...
/sbin/*,/bin/*,/lib/lib*,/usr/sbin/*,/usr/bin/*,/usr/lib/*,/etc/*,/var/*,...
前置应用
HB
IB1/2
gcc
OS
Container
Images
可读可写层
只读层
只读层
只读层
只读层

容器底层技术 - 分层文件系统与磁盘卷挂接点

rhel-7.4-x86_64:1.0.0

```
${COCKER_HOME}/
    images/
        rhel-7.4-x86_64/
            1.0.0/
                rlayer/
                    bin/
                    lib/
                    etc/
                    var/
                    .../
```

ssh:1.0.0

```
${COCKER_HOME}/
    images/
        ssh/
            1.0.0/
                rlayer/
                    bin/
                    lib/
                    etc/
                    var/
                    .../
```

ib2:2.0.0

```
${COCKER_HOME}/
    images/
        ib2/
            2.0.0/
                rlayer/
                    bin/
                    lib/
                    etc/
                    var/
                    .../
```

logpipe:1.0.9

```
${COCKER_HOME}/
    images/
        logpipe/
            1.0.9/
                rlayer/
                    bin/
                    lib/
                    so/
                    etc/
                    .../
```

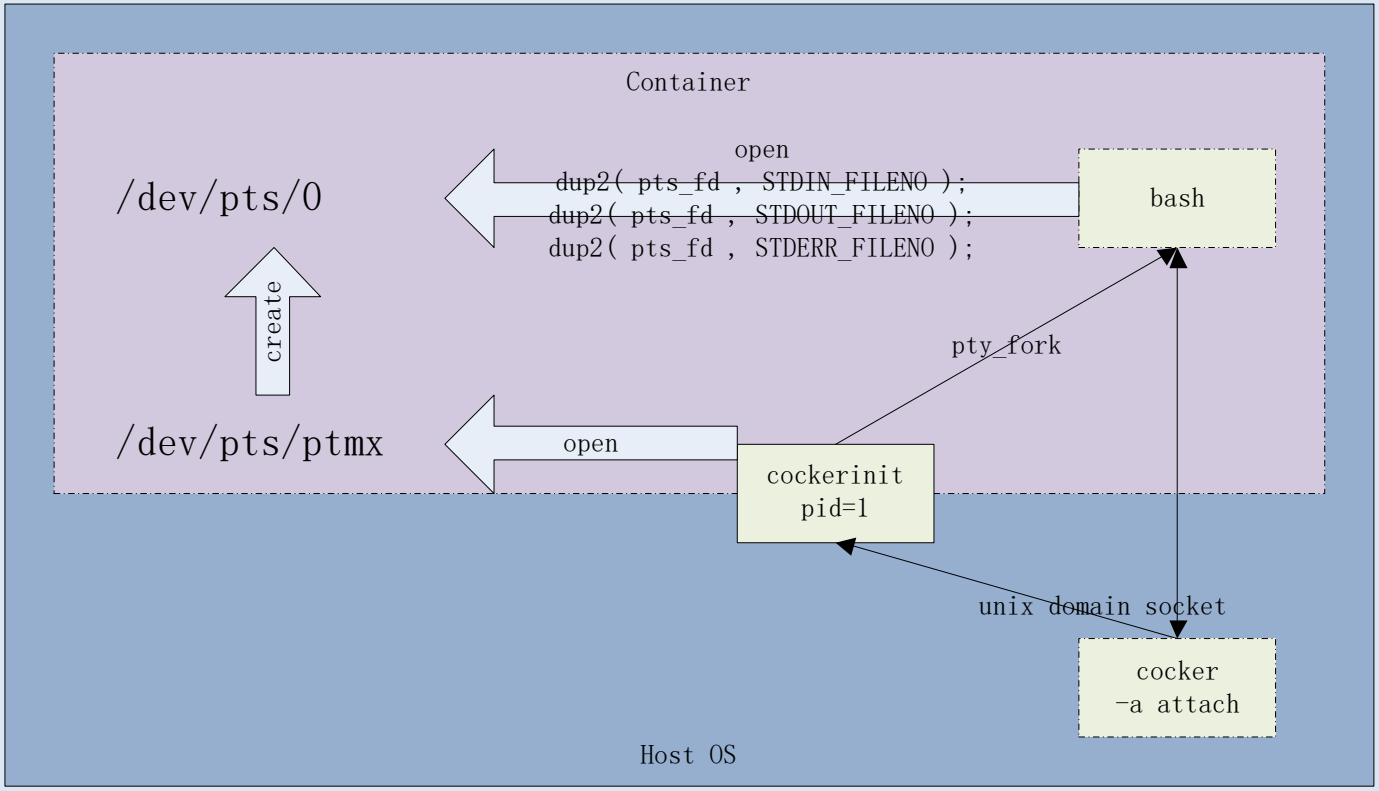
herons

```
${COCKER_HOME}/
    containers/
        herons/
            image,volume,hostname,net,netns,vip,port_mapping
            merged/
            work/
            rwlayer/
                herons/
                    etc/
                    shbin/
                    log/
                    print/
                    modules/ -> (host)/var/cocker/herons/modules
                    .../
```

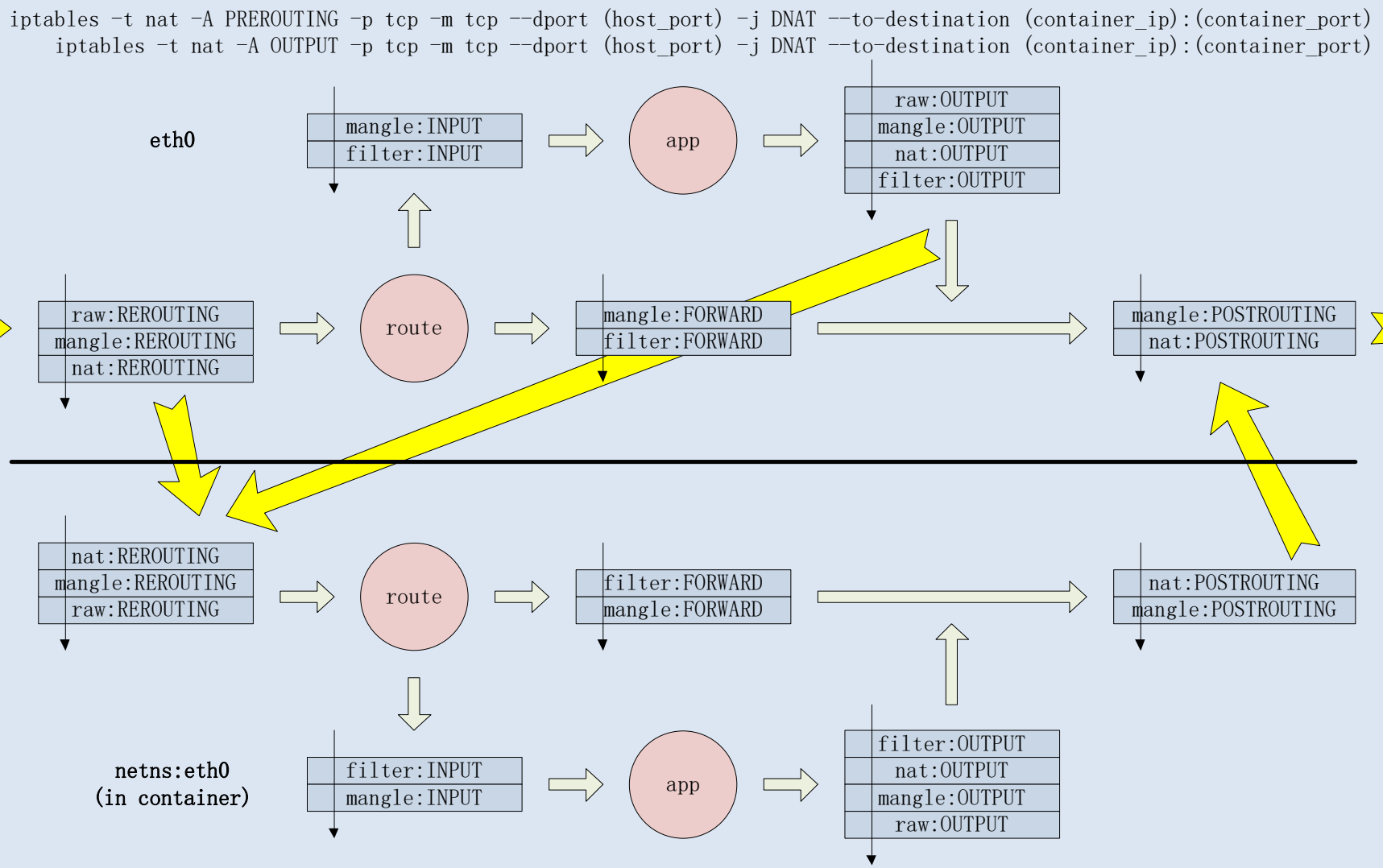
```
...
mount(
"overlay"
, "${COCKER_HOME}/container/herons/merged"
,"overlay"
, MS_MGC_VAL
,
"lower=${COCKER_HOME}/images/logpipe/1.0.9/rlayer:${COCKER_HOME}/images/logpipe/1.0.9/rlayer:${COCKER_HOME}/images/logpipe/1.0.9/rlayer:${COCKER_HOME}/images/logpipe/1.0.9/rlayer,upperdir=${COCKER_HOME}/container/herons/rwlayer,workdir=${COCKER_HOME}/container/herons//workdir"
);
...
chroot( "${COCKER_HOME}/container/herons/merged" ) ;
...
mount( "proc" , "/proc" , "proc" , MS_MGC_VAL , NULL );
...
mount( "devpts" , "/dev/pts" , "devpts" , MS_MGC_VAL , NULL );
...
execl( "/bin/cockerinit" , "cockerinit" , "-c" , "herons" , NULL ) ;
...
```

--volume hsbl:/home/hsbl/log -> ${COCKER_HOME}/volumes/hsbl/log
--volume hsbl:/home/hsbl/print -> ${COCKER_HOME}/volumes/hsbl/print
--volume hsbl:/home/hsbl/msg -> ${COCKER_HOME}/volumes/hsbl/msg

```
static int CloneEntry( void *pv )
{
    struct CockerEnvironment    *env = (struct CockerEnvironment *)pv ;
    …
    TrimEnter( image );
    memset( mount_data , 0x00 , sizeof(mount_data) );
    len = 0 ;
    remain_len = sizeof(mount_data)-1 ;
    nret = InvertLowerDirs( env , image , mount_data , & len , & remain_len ) ;
    l = snprintf( mount_data+len , remain_len , "upperdir=%s/rwlayer,workdir=%s/workdir" , env->container_path_base , env->container_path_base ) ;
    len += l ;
    remain_len -= l ;
    }
    nret = mount( "overlay" , mount_target , "overlay" , MS_MGC_VAL , (void*)mount_data ) ;
    …
    Snprintf( container_volume_file , sizeof(container_volume_file) , "%s/volume" , env->container_path_base ) ;
    container_volume_fp = fopen( container_volume_file , "r" ) ;
    while(1)
    {
        if( fgets( file_buffer , sizeof(file_buffer) , container_volume_fp ) == NULL )
            break;
        volume.host_path = file_buffer ;
        p = strchr( volume.host_path , ':' ) ;
        volume.container_path = p + 1 ;
        (*p) = '\0' ;
        len = snprintf( mount_volume , sizeof(mount_volume)-1 , "%s/merged%s" , env->container_path_base , volume.container_path ) ;
        nret = mount( volume.host_path , mount_volume , NULL , MS_MGC_VAL|MS_BIND , NULL ) ;
    }
    fclose( container_volume_fp );
    …
    nret = chroot( mount_target ) ;
    …
}
```


Container
open
dup2(pts_fd , STDIN_FILENO);
dup2(pts_fd , STDOUT_FILENO);
dup2(pts_fd , STDERR_FILENO);
/dev/pts/0
bash
create
pty_fork
/dev/pts/ptmx
open
cockerinit
pid=1
unix domain socket
cocker
-a attach
Host OS

```
pid_t pty_fork( struct termios *p_origin_termios , struct winsize *p_origin_winsize , int *p_ptm_fd )
{
    …
    ptm_fd = open( "/dev/pts/ptmx" , O_RDWR ) ;
    nret = ptsname_r( ptm_fd , pts_pathfilename , sizeof(pts_pathfilename) ) ;
    nret = chmod( pts_pathfilename , S_IRUSR|S_IWUSR|S_IWGRP ) ;
    lock = 0 ;
    nret = ioctl( ptm_fd , TIOCSPTLCK , & lock ) ;
    …
    pid = fork() ;
    if( pid == -1 )
    {
    }
    else if( pid == 0 )
    {
        close( ptm_fd );
        nret = setsid() ;
        pts_fd = open( pts_pathfilename , O_RDWR ) ;
        nret = ioctl( pts_fd , TIOCSCTTY , NULL ) ;
        nret = ioctl( pts_fd , TIOCSWINSZ , p_origin_winsize ) ;
        nret = dup2( pts_fd , STDIN_FILENO ) ;
        nret = dup2( pts_fd , STDOUT_FILENO ) ;
        nret = dup2( pts_fd , STDERR_FILENO ) ;
        return 0;
    }
    else
    {
        return pid;
    }
}
```


iptables -t nat -A PREROUTING -p tcp -m tcp --dport (host_port) -j DNAT --to-destination (container_ip):(container_port)
iptables -t nat -A OUTPUT -p tcp -m tcp --dport (host_port) -j DNAT --to-destination (container_ip):(container_port)
eth0
mangle:INPUT
filter:INPUT
app
raw:OUTPUT
mangle:OUTPUT
nat:OUTPUT
filter:OUTPUT
raw:REROUTING
mangle:REROUTING
nat:REROUTING
route
mangle:FORWARD
filter:FORWARD
mangle:POSTROUTING
nat:POSTROUTING
nat:REROUTING
mangle:REROUTING
raw:REROUTING
route
filter:FORWARD
mangle:FORWARD
nat:POSTROUTING
mangle:POSTROUTING
netns:eth0
(in container)
filter:INPUT
mangle:INPUT
app
filter:OUTPUT
nat:OUTPUT
mangle:OUTPUT
raw:OUTPUT
iptables -t nat -A POSTROUTING -o (host_eth) -j MASQUERADE

```
int DoAction_boot( struct CockerEnvironment *env )
{
    …
    nret = ReadFileLine( vip , sizeof(vip) , container_vip_file , sizeof(container_vip_file) , "%s/vip" , env->container_path_base ) ;
    nret = ReadFileLine( port_mapping , sizeof(port_mapping)-1 , container_port_mapping_file , sizeof(container_port_mapping_file) , "%s/port_mapping" , env->container_path_base ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip link add %s type veth peer name %s" , env->veth1_name , env->veth0_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "brctl addif %s %s" , env->netbr_name , env->veth1_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ifconfig %s 0.0.0.0" , env->veth1_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip link set %s netns %s" , env->veth0_name , env->netns_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip netns exec %s ip link set %s name %s" , env->netns_name , env->veth0_name , env->veth0_sname ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip netns exec %s ifconfig %s %s" , env->netns_name , env->veth0_sname , vip ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip netns exec %s route add default gw %s netmask 0.0.0.0 dev %s" , env->netns_name , env->netbr_ip , env->veth0_sname ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ifconfig %s up" , env->veth1_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip netns exec %s ifconfig %s up" , env->netns_name , env->veth0_sname ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "ip netns exec %s ifconfig lo up" , env->netns_name ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "iptables -t nat -A POSTROUTING -o %s -j MASQUERADE" , env->host_eth_name ) ;
    p = strtok( port_mapping , "," ) ;
    while( p )
    {
        nret = SnprintfAndSystem( cmd , sizeof(cmd) , "iptables -t nat -A PREROUTING -p tcp -m tcp --dport %d -j DNAT --to-destination %s:%d" , env->src_port , vip , env->dst_port ) ;
        nret = SnprintfAndSystem( cmd , sizeof(cmd) , "iptables -t nat -A OUTPUT -p tcp -m tcp --dport %d -j DNAT --to-destination %s:%d" , env->src_port , vip , env->dst_port ) ;
        p = strtok( NULL , "," ) ;
    }
    …
    pid = clone( CloneEntry , stack_bottom+sizeof(stack_bottom) , CLONE_NEWNS|CLONE_NEWPID|CLONE_NEWIPC|CLONE_NEWUTS|CLONE_NEWNET , (void*)env ) ;
    …
}
```

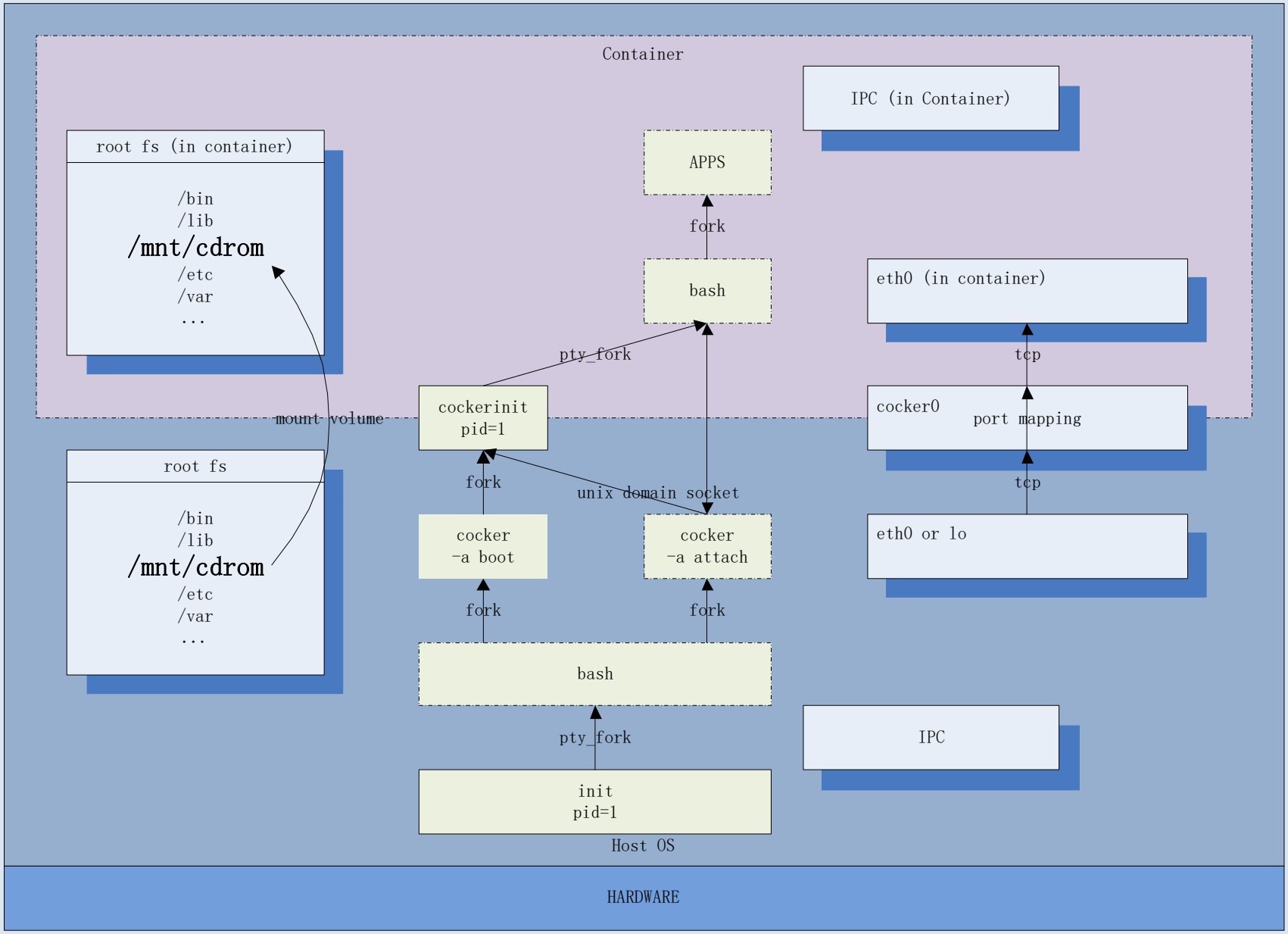
```
static int CloneEntry( void *pv )
{
    …
    len = snprintf( netns_path , sizeof(netns_path)-1 , "/var/run/netns/%s" , env->netns_name ) ;
    netns_fd = open( netns_path , O_RDONLY ) ;
    nret = setns( netns_fd , CLONE_NEWNET ) ;
    close( netns_fd );
    …
    nret = chroot( mount_target ) ;
    …
}
```

```
int DoAction_install_test( struct CockerEnvironment *env )
{
    nret = SnprintfAndMakeDir( version_path_base , sizeof(version_path_base)-1 , "%s/test" , env->images_path_base ) ;
    nret = SnprintfAndMakeDir( env->image_path_base , sizeof(env->image_path_base)-1 , "%s/%s" , version_path_base , env->cmd_para.__version ) ;
    nret = SnprintfAndMakeDir( image_rlayer_path , sizeof(image_rlayer_path)-1 , "%s/rlayer" , env->image_path_base ) ;
    nret = SnprintfAndSystem( cmd , sizeof(cmd) , "cocker_install_test.sh %s" , image_rlayer_path ) ;
    return 0;
}
```

```
cd ${IMAGE_RLAYER_PATH_BASE}
mkdir -p -m 755 bin sbin lib lib64 usr etc root dev proc mnt var tmp
mkdir -p -m 755 mnt/cdrom

# install /bin and /lib64
cd /bin
cp bash tty which locale env ls cat echo cp mv rm pwd cd mkdir rmdir clear ps grep more id uname hostname vi vim awk sed tr file ldd top netstat ping telnet ssh nc mount umount ${IMAGE_RLAYER_PATH_BASE}/bin/
cp cockerinit ${IMAGE_RLAYER_PATH_BASE}/bin/
cd ${IMAGE_RLAYER_PATH_BASE}/bin/
cocker_ldd_and_cp_lib64.sh
```

```
# install /sbin and /lib64
cd /sbin
cp ifconfig route ip iptables ${IMAGE_RLAYER_PATH_BASE}/sbin/
cd ${IMAGE_RLAYER_PATH_BASE}/sbin/
cocker_ldd_and_cp_lib64.sh

# install /etc/profile and $HOME/.profile
cp /bin/cocker_etc_profile_template.sh ${IMAGE_RLAYER_PATH_BASE}/etc/profile
cp /bin/cocker_profile_template.sh ${IMAGE_RLAYER_PATH_BASE}/root/.profile

# install /etc/passwd and /etc/group
echo "root:x:0:0:root:/root:/bin/bash" >${IMAGE_RLAYER_PATH_BASE}/etc/passwd
echo "root:x:0:" >${IMAGE_RLAYER_PATH_BASE}/etc/group

# install /dev/pts
mknod -m 600 ${IMAGE_RLAYER_PATH_BASE}/dev/console c 5 1
mknod -m 600 ${IMAGE_RLAYER_PATH_BASE}/dev/initctl p
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/full c 1 7
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/null c 1 3
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/zero c 1 5
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/random c 1 8
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/urandom c 1 9
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/ptmx c 5 2
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/tty c 5 0
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/tty0 c 4 0
mkdir -p ${IMAGE_RLAYER_PATH_BASE}/dev/pts

# install /usr/share/terminfo
mkdir -p ${IMAGE_RLAYER_PATH_BASE}/usr/share
cp -rf /usr/share/terminfo ${IMAGE_RLAYER_PATH_BASE}/usr/share/
```

```
...
supermin5 -v --prepare bash coreutils -o supermin.d
supermin5 -v --prepare bash coreutils which man-db procps-ng hostname vim-minimal file iputils openssh-clients nmap-ncat util-linux tar wget curl net-tools iproute iptables iptables-services sysvinit-tools time bc m4 expect less findutils sysstat iotop ksh tcsh ftp zip gzip at shadow-utils dos2unix psmisc cronie telnet initscripts sudo cups-client lsof tcpdump sysctl glibc-common binutils valgrind strace yum -o supermin.d
supermin5 -v --build --format chroot supermin.d -o appliance.d

IMAGE_RLAYER_PATH_BASE="appliance.d"

mknod -m 600 ${IMAGE_RLAYER_PATH_BASE}/dev/console c 5 1
mknod -m 600 ${IMAGE_RLAYER_PATH_BASE}/dev/initctl p
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/full c 1 7
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/null c 1 3
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/zero c 1 5
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/random c 1 8
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/urandom c 1 9
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/ptmx c 5 2
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/tty c 5 0
mknod -m 666 ${IMAGE_RLAYER_PATH_BASE}/dev/tty0 c 4 0
mkdir -p ${IMAGE_RLAYER_PATH_BASE}/dev/pts

cp /bin/cockerinit ${IMAGE_RLAYER_PATH_BASE}/bin/
cp /lib64/libcocker_util.so ${IMAGE_RLAYER_PATH_BASE}/lib64/

cp /etc/resolv.conf ${IMAGE_RLAYER_PATH_BASE}/etc/

mkdir -p /mnt/cdrom

cd appliance.d && tar --numeric-owner -cvzf ../${IMAGE_FILENAME} * && cd ..

rm -rf appliance.d supermin.d
...
```


Container
IPC (in Container)
root fs (in container)
/bin
/lib
/mnt/cdrom
/etc
/var
...
APPS
fork
bash
eth0 (in container)
pty_fork
tcp
cockerinit
pid=1
cocker0
port mapping
mount volume
root fs
/bin
/lib
/mnt/cdrom
/etc
/var
...
fork
unix domain socket
tcp
cocker
-a boot
cocker
-a attach
eth0 or lo
fork
fork
bash
pty_fork
IPC
init
pid=1
Host OS
HARDWARE

```
int DoAction_boot( struct CockerEnvironment *env )
{
    创建容器内外一对虚拟网卡
    把容器外网卡与网桥cocker0绑定
    设置容器外网卡地址为'0.0.0.0'
    把容器内网卡加入到网络名字空间中
    把容器内网卡改名为简名'eth0'
    设置容器内网卡地址
    增加容器内网卡缺省路由
    点亮容器外网卡
    点亮容器内网卡
    设置宿主机网卡iptables链POSTROUTING表nat的动作为修改源地址
    遍历所有虚拟端口映射配置
        设置宿主机iptables链PREROUTING表nat的目标端口转向容器内端口
        设置宿主机iptables链OUTPUT表nat的目标端口转向容器内端口
    调用clone创建容器根进程并等待结束
}

static int CloneEntry( void *pv )
{
    设置网络名字空间
    设置uid
    设置gid
    设置hostname
    挂接分层文件系统
    挂接磁盘卷
    设置cgroup系统资源限制
    更改'/'目录
    挂接'/proc'文件系统
    挂接'/dev/pts'文件系统
    更改当前目录为'/root'
    更新程序映像为'/bin/cockerinit'
}
```

1. 概述

1.1. cocker是什么

`cocker`是我个人用C语言完全自研的容器引擎（对标`Docker`、`阿里Pouch`），主要解决如下工作场景中的痛点：

- 原生支持多进程架构的容器使用模式，无须引入第三方组件。
- 按虚拟主机方式管理容器，交互式构建镜像，写过复杂Dockerfile的人都深恶痛绝。
- 镜像多版本共存管理。
- （更多...）

2.1.3. 准备cocker源码

下载cocker源码包，解开，进入

```
# tar xvzf cocker-X.X.X.tar.gz
# cd cocker-X.X.X
```

或克隆cocker源码库，进入

```
# git clone https://gitee.com/calvinwilliams/cocker
# cd cocker
```

or

```
# git clone https://github.com/calvinwilliams/cocker
# cd cocker
```

2.1.4. 编译安装

注意：如果你在非root用户编译源码，确认`sudo`无需输入密码。

清理中间文件

```
# make -f makefile.Linux clean
```

编译并安装到系统目录里

注意：如果你在非root用户编译源码，前面加上`sudo -E`。

```
# make -f makefile.Linux install
```

3.1. cocker指令

不带选项执行`cocker`将得到所有指令和选项列表

```
# cocker
USAGE : cocker -v
               -s images
               -s containers
               -a create (-m|--image) (image[:version])[,(image[:version])]... [ create options ] [ (-c|--container)
               -a boot (-c|--container) (container) [ cgroup options ] [ (-t|--attach) | (-e|--exec) (cmd|"program p
               -a attach (-c|--container) (container)
               -a shutdown (-c|--container) (container) [ (-f|--forcely) ]
               -a kill (-c|--container) (container) [ (-f|--forcely) ]
               -a destroy (-c|--container) (container) [ (-f|--forcely) ] [ (-h|--shutdown) ]
               -a version (-m|--image) (image[:version]) [ --version (version) ]
               -a vip (-c|--container) (container) --vip (ip)
               -a port_mapping (-c|--container) (container) --port-mapping (src_port:dst_port)
               -a volume (-c|--container) (container) --volume (host_path[:container_path])[ ...]
               -a to_image --from-container (container) [ --verion (verion) ] --to-image (image)
               -a to_container --from-image (image[:version]) (-m|--image) (image[:version])[,(image[:version])]...
               -a copy_image --from-image (image[:version]) --to-image (image[:version])
               -a del_image (-m|--image) (image[:version])
               -a import --image-file (file)
               -a export (-m|--image) (image[:version])
               -s ssearch --srepo (user@host)
               -a install_test
create options : [ --volume (host_path:container_path) ][ --volume ... ] [ --host (hostname) ] [ --net (BRIDGE|HOST|
cgroup options : [ --cpus [(cpu_num,...)|(cpu_num-cpu_num2)] ] [ --cpu-quota (percent%) ] [ --mem-limit (num|numM) ]
  enable debug : [ (-d|--debug) ]
```

注意：首次执行`cocker`会自动创建镜像主目录`images`、容器主目录`containers`。

3.1.2. 查询镜像列表

使用`cocker`指令`-s images`查询镜像主目录里的所有镜像。

```
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            _          2018-11-10T09:21:12 24 MB
```

镜像目录层次为`镜像名/版本号/镜像目录文件内容`。如果没有版本号，版本号目录名为`_`。

镜像名格式推荐`(个人名或组织名)=(软件名)-(软件版本号)`。版本号格式推荐`x.y.z`。

我们可以使用指令`-a install_test`创建不同版本的测试镜像，多版本共存管理。

```
# cocker -a install_test --version "1.0.0"
OK
# cocker -a install_test --version "1.1.0"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            _          2018-11-10T09:21:12 24 MB
test                            1.0.0      2018-11-14T07:20:06 24 MB
test                            1.1.0      2018-11-14T07:20:17 24 MB
```

3.1.3. 由镜像创建容器

使用`cocker`指令`-a create`由一个或多个镜像叠加创建容器。

```
# cocker -a create -m test --host test --net BRIDGE --vip 166.88.0.2 --port-mapping 19527:9527 -c test
OK
```

`-m（镜像列表）`:指定镜像列表，镜像名可以以`（镜像名）(:版本号）`格式指定版本号，本指令中允许不指定版本号，`cocker`会自动挑选一个最大版本号的镜像。多个镜像之间用`，`分隔。

`--host（主机名）`:设置容器内的主机名。

`--net（网络模型）`:设置容器网络模型，见前面网络模型章节。

`--vip（ip）`:如果网络模型为`BRIDGE`，设置容器内的网卡IP。

`--port-mapping（网络端口映射列表）`:如果网络模型为`BRIDGE`，设置外部或宿主机访问容器的网络端口映射列表，端口映射格式为`（宿主机端口）:（容器端口）`。多个端口映射之间用`，`分隔。

`-c（容器名）`:指定容器名，若不指定，与镜像同名。建议指定。

除了以上示例中用到的选项，以下为其它可选选项：

`--host-eth（网卡名）`:指定宿主机对外网卡名。这在多物理网卡时使用。

`--volume（磁盘卷映射列表）`:磁盘卷映射用于宿主机与容器之间目录共享，设置格式为`（宿主机目录:容器目录）`。多个磁盘卷映射使用各自的选项键前缀`--volume`。

`-b`:容器创建完后立即启动。（后可追加所有启动容器选项）

3.1.4. 查询容器列表

使用`cocker`指令`-s containers`查询容器主目录中的所有容器以及状态。

```
# cocker -s containers
container_id         image                hostname    net         netns             size       status
-------------------------------------------------------------------------------------------------------
test                 test                 test        BRIDGE      nns098F6BCD46     0 B        STOPED
```

3.1.5. 启动容器

使用`cocker`指令`-a boot`查询容器主目录中的所有容器以及状态。

```
# cocker -a boot -c test
OK
```

3.1.6. 连接容器

如果使用`cockerinit`作为根进程启动容器，使用`cocker`指令`-a attch`连接至容器，`cockerinit`打开一个会话连接到容器中的伪终端。也可叠加ssh镜像在容器内启动ssh服务器，利用ssh连接至容器。

```
# cocker -a attach -c test
connect to container ok
--- Welcome to cocker contrainer ---

[root@test /root]
```

3.1.7. 停止容器

使用`cocker`指令`-a shutdown`停止容器。

```
# cocker -a shutdown -c test
OK
```

```
# cocker -s containers
container_id         image                hostname    net        netns             size       status
-------------------------------------------------------------------------------------------------------
test                 test                 test        BRIDGE     nns098F6BCD46     0 B        STOPED
```

3.1.8. 杀死容器

使用`cocker`指令`-a kill`强杀容器。

3.1.9. 销毁容器

使用`cocker`指令`-a destroy`销毁容器。

注意：销毁容器后容器内所有修改将丢失。

```
# cocker -a destroy -c test
OK
```

3.1.10. 修改镜像属性

3.1.10.1. 修改版本号

使用`cocker`指令`-a version`修改镜像版本号。

```
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            _          2018-11-10T09:21:12 24 MB
# cocker -a version -m test --version "1.0.1"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            1.0.1      2018-11-10T09:21:12 24 MB
# cocker -a version -d -m "test:1.0.1" --version "1.0.2"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            1.0.2      2018-11-10T09:21:12 24 MB
# cocker -a version -d -m "test:1.0.2"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            _          2018-11-10T09:21:12 24 MB
```

3.1.11. 修改容器属性

3.1.11.1. 修改VIP

使用 `cocker` 指令 `-a vip` 修改容器内网卡IP。

注意：必须容器停止后才能修改。

```
# cocker -a vip --vip 166.88.0.3 -c test
OK
```

3.1.11.2. 修改容器端口映射

使用 `cocker` 指令 `-a port_mapping` 修改容器网络端口映射。

注意：必须容器停止后才能修改。

```
# cocker -a port_mapping --port-mapping 19528:9528 -c test
OK
```

3.1.11.3. 修改外挂卷映射

使用 `cocker` 指令 `-a volume` 修改容器磁盘卷映射。

注意：必须容器停止后才能修改。

```
# cocker -a volume --volume "/tmp:/tmp" --volume "/mnt/cdrom:/mnt/cdrom" -c test
OK
```

3.1.12. 镜像转换为容器

当需要修改镜像内文件时可先把镜像转换为容器，修改完后转换回镜像。

使用`cocker`指令`-a to_container`转换指定镜像为容器。

```
# cocker -a to_container --from-image test --host test --net BRIDGE --vip 166.88.0.2 --port-mapping 19527:9527 --to-
OK
```

注意：几乎可使用所有指令`-a create`的选项。

3.1.13. 容器转换为镜像

当想把某一容器打包成镜像，可使用此指令。

使用`cocker`指令`-a to_image`转换指定容器为镜像。

注意：转换的容器必须是停止的。

```
# cocker -a to_image --from-container test --to-image test
OK
```

3.1.14. 复制镜像

使用 `cocker` 指令 `-a copy_image` 可复制镜像。

```
# cocker -a copy_image --from-image test --to-image "test2:1.0.0"
OK
```

3.1.15. 删除镜像

使用 `cocker` 指令 `-a del_image` 可删除镜像。

```
# cocker -a del_image -m "test2:1.0.0"
OK
```

3.1.16. 导出镜像

使用 `cocker` 指令 `-a export` 可导出镜像为镜像打包文件。

```
# cocker -a export -m "test:1.1.0"
OK
```

3.1.17. 导入镜像

使用 `cocker` 指令 `-a import` 可从镜像打包文件导入镜像库。

注意：镜像打包文件名扩展名必须是 `.cockerimage` 。

```
# cocker -a del_image -m "test:1.1.0"
OK
# cocker -a import --image-file "test:1.1.0.cockerimage"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            1.1.0      2018-11-14T08:53:13 24 MB
```

3.1.18. 上传镜像到ssh镜像库

ssh镜像库是利用ssh服务器来搭建镜像库。首先安装ssh服务器，创建镜像库用户，从客户端产生公钥文件分发给镜像库以方便免密登录。

```
# ssh-keygen -t rsa
...
# ssh-copy-id -i ~/.ssh/id_rsa.pub cockerimages@192.168.6.74
```

使用 `cocker` 指令 `-s ssearch` 可查看ssh镜像库里的镜像列表。

```
# cocker -s ssearch --srepo "cockerimages@192.168.6.74"
OK
```

注意： `cocker` 保存ssh镜像库地址配置 `cockerimages@192.168.6.74` 。

还能加上子串通配选项 `--match` 。

```
# cocker -s ssearch --match test
```

使用 `cocker` 指令 `-a spush` 上传镜像到ssh镜像库。

```
# cocker -a spush -m "test:1.0.0"
OK
# cocker -s ssearch --match test
cocker -s ssearch
image_id                                        modify_datetime     size
-------------------------------------------------------------------------
test:1.0.0                                      2018-11-14T9:05:48  11 MB
```

3.1.19. 从ssh镜像库下载镜像

使用`cocker`指令`-a spull`从ssh镜像库下载镜像。

```
#
cocker -a del_image -m "test:1.0.0"
OK
# cocker -a spull -m "test:1.0.0"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
test                            1.0.0      2018-11-14T09:09:04 24 MB
```

3.3.3. 交互式构建G6镜像

G6是本人以前独立自研的负载均衡器（源码托管地址：开源中国、github），不少公司在使用，下面介绍交互式构建G6镜像。

```
# cocker -a create -m "calvin=rhel-7.4-x86_64,calvin=yum,calvin=sshd" --host G6 --volume "/mnt/cdrom:/mnt/cdrom" --n
OK
# cocker -a boot -c "calvin=G6" -t
[root@G6 /root]
```

在容器内配置好G6，在我的环境里这样配置

```
[root@G6 /root] yum install -y git
[root@G6 /root] yum install -y make
[root@G6 /root] yum install -y gcc
...（这里仅作演示，所以把git、gcc都安装在一起了）
[root@G6 /root] mkdir src && cd src
[root@G6 /root/src] git clone https://gitee.com/calvinwillisms/G6 && cd G6
[root@G6 /root/src/G6] cd src
[root@G6 /root/src/G6/src] make -f makefile.Linux install
...
[root@G6 /root/src/G6/src] mkdir ~/etc/ && vi ~/etc/G6.conf
admin_rule_id G 127.0.0.1:* - 127.0.0.1:8600 ;
my_rule RR *:* - 0.0.0.0:222 > 166.88.0.2:22 ;
[root@G6 /root/src/G6/src] cd ~
[root@G6 /root] nohup /usr/sbin/sshd -D &
[root@G6 /root] G6
G6 v1.0.6 build Nov 26 2018 14:28:18
TCP Bridge && Load-Balance Dispenser
Copyright by calvin 2016
USAGE : G6 -f (config_pathfilename) [ -t (forward_thread_size) ] [ -s (forward_session_size) ] [ --log-level (DEBUG|
[root@G6 /root] G6 -f ~/etc/G6.conf
```

另外开一屏连接G6容器

```
# ssh root@166.88.0.2 -p 2222
root@166.88.0.2's password: （root密码前面被重置成root了）
Last login: Mon Nov 26 13:28:07 2018 from 192.168.6.7
-bash-4.2#
```

转换容器为G6镜像

```
[root@G6 /root] ps -ef
...
[root@G6 /root] ps -ef | grep -v grep | grep "G6 -f" | awk '{if($3==1)print $2}' | xargs kill
[root@G6 /root] ps -ef | grep -v grep | grep -w sshd | awk '{print $2}' | xargs kill
[root@G6 /root] exit
logout
# cocker -a shutdown -c "calvin=G6"
OK
# cocker -s containers
container_id         image                hostname   net        netns            size       status
-------------------------------------------------------------------------------------------------------
calvin=G6            calvin=rhel-7.4-x86_64,calvin=yum,calvin=sshd G6         BRIDGE     nns2513F44178    373 MB
# cocker -a to_image --from-container "calvin=G6" --version "1.0.0" --to-image "calvin=G6"
OK
# cocker -s containers
# cocker -s images
image_id                       version    modify_datetime     size
-------------------------------------------------------------------
calvin=rhel-7.4-x86_64         1.0.0      2018-11-25T09:55:25 271 MB
calvin=yum                     1.0.0      2018-11-25T10:16:59 24 MB
calvin=sshd                    1.0.0      2018-11-26T09:16:59 335 MB
calvin=G6                      1.0.0      2018-11-26T10:01:48 373 MB
```

3.3.4. 镜像配置实例化容器并启动

还是拿本人的 `G6` 做例子，把镜像 `calvin=G6` 转化为容器，把配置文件改成模板后再转化回镜像

```
# cocker -a to_container --from-image "calvin=G6" -m "calvin=rhel-7.4-x86_64" --host G6 --to-container "calvin=G6"
OK
# cocker -a boot -c "calvin=G6" -t
[root@G6 /root] cd etc
[root@G6 /root/etc] mv G6.conf G6.conf.tpl
[root@G6 /root/etc] vi G6.conf.tpl
admin_rule G 127.0.0.1:* - 127.0.0.1:${ADMIN_PORT} ;
my_rule RR *:* - 0.0.0.0:${FORWARD_PORT} > ${DEST_IP}:${DEST_PORT} ;
[root@G6 /root/etc] vi ../bin/sshd.start
nohup /usr/sbin/sshd -D &
[root@G6 /root/etc] chmod +x ../bin/sshd.start
[root@G6 /root/etc] exit
logout
# cocker -a shutdown -c "calvin=G6"
OK
# cocker -a to_image --from-container "calvin=G6" --version "1.1.0" --to-image "calvin=G6"
OK
# cocker -s images
image_id                        version    modify_datetime     size
-------------------------------------------------------------------
calvin=rhel-7.4-x86_64          1.0.0      2018-11-27T08:00:07 273 MB
calvin=yum                      1.0.0      2018-11-26T09:10:43 24 MB
calvin=sshd                     1.0.0      2018-11-26T09:17:12 335 MB
calvin=G6                       1.1.0      2018-11-27T08:03:33 373 MB
```

最后用镜像创建容器，配置实例化，启动服务

```
# cocker -a create -m "calvin=rhel-7.4-x86_64,calvin=sshd,calvin=G6" --host G6 --net BRIDGE --vip 166.88.0.2 --port-
OK
# cocker -a boot -c "G6"
OK
# vi G6.conf.map
${ADMIN_PORT}   8600
${FORWARD_PORT} 222
${DEST_IP}      166.88.0.2
${DEST_PORT}    22
# cocker -a rplfile -c "G6" --template-file "/root/etc/G6.conf.tpl" --mapping-file "G6.conf.map" --instance-file "/r
OK
# cocker -a run -c "G6" --cmd "cat /root/etc/G6.conf"
admin_rule G 127.0.0.1:* - 127.0.0.1:8600 ;
my_rule RR *:* - 0.0.0.0:222 > 166.88.0.2:22 ;
# cocker -a run -c "G6" --cmd "nohup /usr/sbin/sshd -D"
nohup: ignoring input and appending output to 'nohup.out'
# cocker -a run -c "G6" --cmd "G6 -f /root/etc/G6.conf"
OK
```

cocker -a rplfile -c "G6" --template-file "/root/etc/G6.conf.tpl" --mapping-file "G6.conf.map" --instance-file "/root/etc/G6.conf"

另外开一屏连接G6容器

```
# ssh root@166.88.0.2 -p 2222
root@166.88.0.2's password: （root密码前面被重置成root了）
Last login: Mon Nov 26 13:28:07 2018 from 192.168.6.7
-bash-4.2# exit
```

用完后关闭服务，最后停止和销毁容器

```
# cocker -a run -c "G6" --cmd "ps -ef | grep -v grep | grep 'G6 -f' | awk '{if($3==1)print $2}' | xargs kill"
# cocker -a run -c "G6" --cmd "ps -ef | grep -v grep | grep -w sshd | awk '{print $2}' | xargs kill"
# cocker -a shutdown -c G6
```

3.3.5. 单进程启动容器

拿前面的 `G6` 容器来演示像 `Docker` 那样单进程启动容器

```
# cocker -a create -m "calvin=rhel-7.4-x86_64,calvin=sshd,calvin=G6" --host G6 --net BRIDGE --vip 166.88.0.2 --port-
OK
```

用完后停止并销毁容器

```
# cocker -a destroy -c G6 -h
OK
```

cocker -a create -m "calvin=rhel-7.4-x86_64,calvin=sshd,calvin=G6" --host G6 --net BRIDGE --vip 166.88.0.2 --port-mapping "8600:8600,2222:222" -c "G6" -b -e "/root/bin/G6 -f /root/etc/G6.conf --no-daemon" -d

习题：分别简述容器底层用到的技术。

https://gitee.com/calvinwilliams/cocker

https://github.com/calvinwilliams/cocker

Thank You

^_^

Question && Answer && Suggest