

深入浅出容器技术

马震 2022-08-26

会议签到二维码



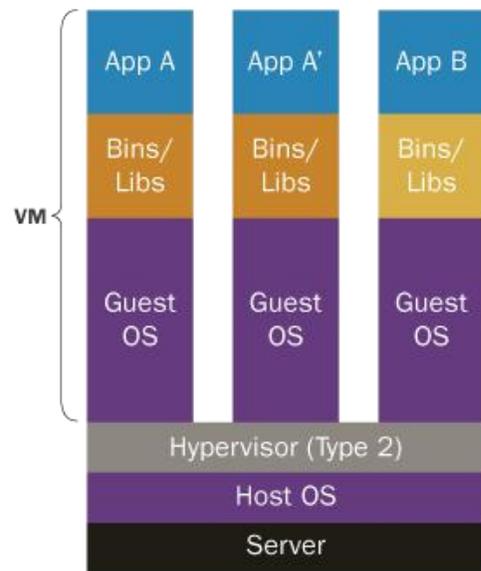
会议时间: 2022.08.26 16:15-17:45

地点: B1001

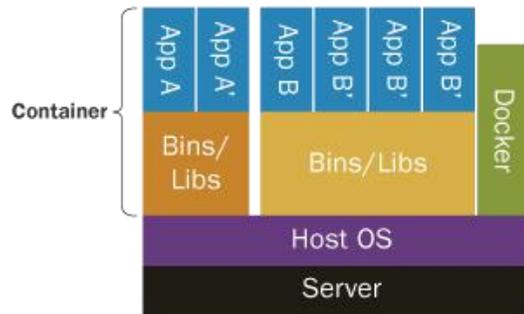
容器不是虚拟机

- 容器的本质就是一个（被隔离和受限的）进程
 - 使用namespaces对进程进行隔离
 - 使用cgroups限制进程使用的资源
 - 使用 capabilities, AppArmor, Seccomp进行安全限制
- 每个虚拟机都有独立的操作系统内核
- 容器共享同一个操作系统内核

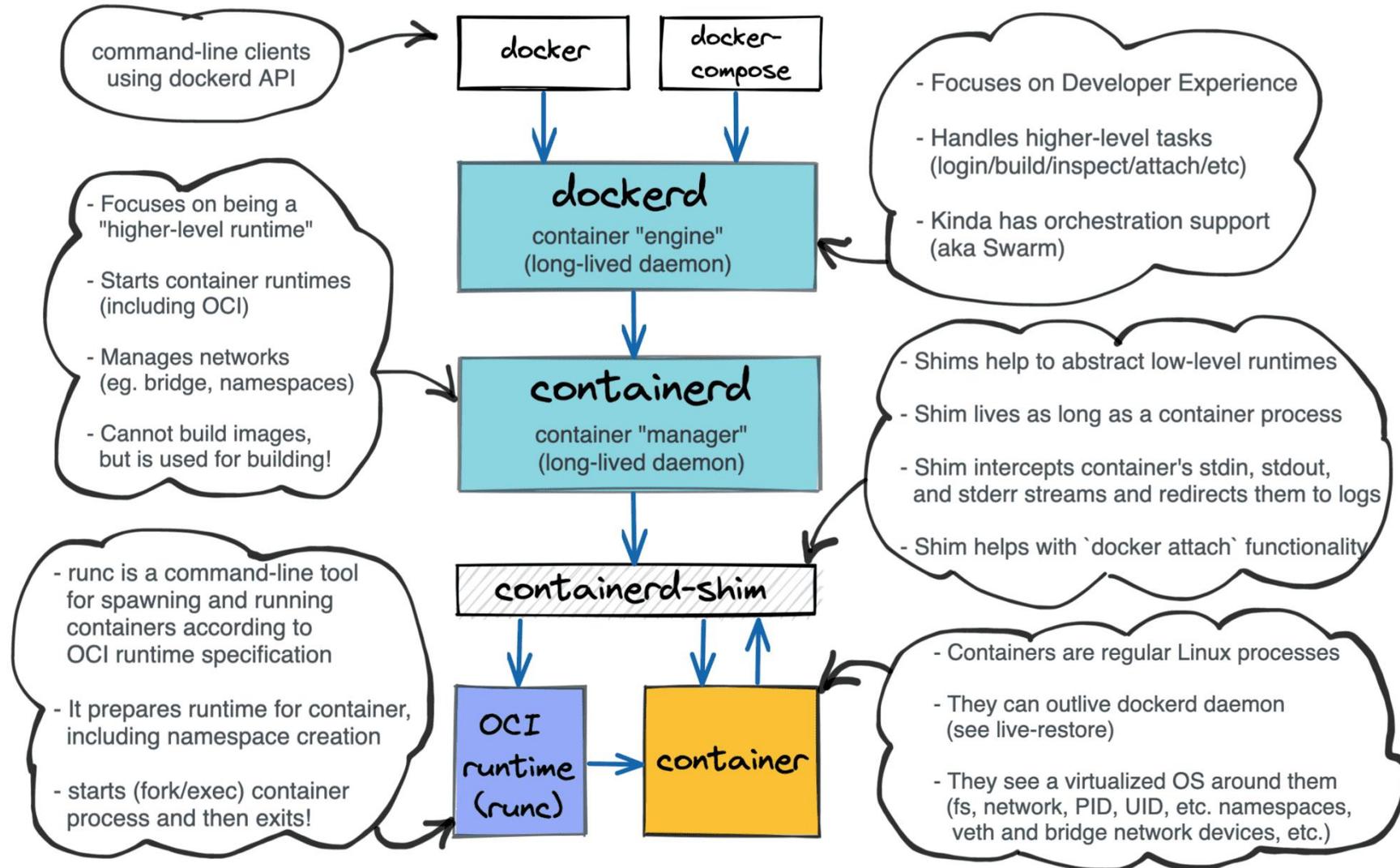
Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries



Docker的分层架构



容器运行时

- OCI([Open Container Initiative](#)) 制定了容器运行时规范[OCI runtime specification](#).
- 使用最广泛的容器运行时 [runc](#)
 - 最初是Docker的一部分
 - runc是OCI运行时规范的参考实现
- crun
 - C语言编写的OCI容器运行时

runc

- runc是一个命令行工具，启动容器进程
- 需要向runc提供一个bundle
 - Linux root file system
 - OCI运行时规范的配置文件
- runc将准备所有必要的Linux资源，最主要的是namespaces和cgroups，并在容器化环境中启动指定进程

每个容器镜像都必须包含OS吗

- 一般都是基于Ubuntu、centos、alpine等发行版构建镜像
- 容器只是一个Linux进程
- 镜像可以只包含一个可执行文件
 - 基于scratch镜像创建
 - 使用一个静态链接的二进制文件
 - 不依赖系统中的一些共享库
 - 基于Linux内核就可以运行



```
# cat Dockerfile
FROM scratch
COPY hello /
CMD ["/hello"]

# docker build -t hello .
# docker run hello
Hello World!
```

使用 dive 查看 hello 镜像

```
| ● Layers |-----| Current Layer Contents |-----|
| Cmp  Size  Command |-----| Permission  UID:GID  Size  Filetree |-----|
| 964 kB  FROM 8ca35db78ee6e2d |-----| -rwxr-xr-x  0:0  964 kB  └─ hello |-----|
|
| Layer Details |-----|
|
| Tags: (unavailable)
| Id: 8ca35db78ee6e2d124411ddeb22f9781da61315bad65893c81e55a7e7dd82
| b95
| Digest: sha256:a0b05635d26c23b9338d21c666fd5de6ce71f13318dc0c6c325631
| 95891c1758
| Command:
| #(nop) COPY file:33e5ec938f8c8b4aa0aa7f6503e5d32cc76cfa96b13bda838f25
| 705724cf6bc8 in /
|
| Image Details |-----|
|
| Total Image size: 964 kB
| Potential wasted space: 0 B
| Image efficiency score: 100 %
|
| Count  Total Space  Path
|
| ^C Quit | Tab Switch view | ^F Filter | ^L Show layer changes | ^A Show aggregated changes
```

使用 [dive](#) 查看 nginx 镜像

Layers			● Current Layer Contents			
Cmp	Size	Command	Permission	UID:GID	Size	Filetree
80 MB	FROM	1c65531519b901e	drwxr-xr-x	0:0	5.3 MB	bin
61 MB	set -x	&& addgroup --system --gid 101 nginx && ad	drwxr-xr-x	0:0	0 B	boot
1.2 kB	#(nop)	COPY file:65504f71f5855ca017fb64d502ce873a31b2e0de	drwxr-xr-x	0:0	0 B	dev
2.0 kB	#(nop)	COPY file:0b866ff3fc1ef5b03c4e6c8c513ae014f691fb05	drwxr-xr-x	0:0	153 kB	etc
1.0 kB	#(nop)	COPY file:0fd5fca330dcd6a7de297435e32af634f29f7132	drwxr-xr-x	0:0	0 B	home
4.6 kB	#(nop)	COPY file:09a214a3e07c919af2fb2d7c749ccbc446b8c10e	drwxr-xr-x	0:0	8.4 MB	lib
			drwxr-xr-x	0:0	0 B	lib64
			drwxr-xr-x	0:0	0 B	media
			drwxr-xr-x	0:0	0 B	mnt
			drwxr-xr-x	0:0	0 B	opt
			drwxr-xr-x	0:0	0 B	proc
			drwxr-xr-x	0:0	732 B	root
			drwxr-xr-x	0:0	0 B	run
			drwxr-xr-x	0:0	4.0 MB	sbin
			-rwxr-xr-x	0:0	65 kB	agetty
			-rwxr-xr-x	0:0	35 kB	badblocks
			-rwxr-xr-x	0:0	35 kB	blkdiscard
			-rwxr-xr-x	0:0	121 kB	blkid
			-rwxr-xr-x	0:0	76 kB	blkzone
			-rwxr-xr-x	0:0	68 kB	blockdev
			-rwxr-xr-x	0:0	47 kB	chcpu
			-rwxr-xr-x	0:0	39 kB	ctrlaltdel
			-rwxr-xr-x	0:0	239 kB	debugfs
			-rwxr-xr-x	0:0	31 kB	dumpe2fs
			-rwxr-xr-x	0:0	348 kB	e2fsck
			-rwxr-xr-x	0:0	39 kB	e2image
			-rwxrwxrwx	0:0	0 B	e2label → tune2fs
			-rwxrwxrwx	0:0	0 B	e2mmpstatus → dumpe2fs
			-rwxr-xr-x	0:0	7.3 kB	e2scrub
			-rwxr-xr-x	0:0	5.4 kB	e2scrub_all
			-rwxr-xr-x	0:0	23 kB	e2undo
			-rwxr-xr-x	0:0	15 kB	findfs
			-rwxr-xr-x	0:0	56 kB	fsck
			-rwxr-xr-x	0:0	43 kB	fsck.cramfs
			-rwxrwxrwx	0:0	0 B	fsck.ext2 → e2fsck

Layer Details

Tags: (unavailable)
Id: 1c65531519b901e0508c411808f7dc2497a6549926bda748896f59aaddc4e
ed4
Digest: sha256:92a4e8a3140f7a04a0e5a15793adef2d0e8889ed306a8f95a6cfb6
7cecb5f212
Command:
#(nop) ADD file:0eae0dca665c7044bf242cb1fc92cb8ea744f5af2dd376a558c90
bc47349aefe in /

Image Details

Total Image size: 142 MB
Potential wasted space: 3.8 MB
Image efficiency score: 98 %

Count	Total Space	Path
2	1.7 MB	/var/cache/debconf/templates.dat
2	1.6 MB	/var/cache/debconf/templates.dat-old
2	203 kB	/var/lib/dpkg/status
2	203 kB	/var/lib/dpkg/status-old
2	59 kB	/var/log/lastlog
2	22 kB	/var/cache/debconf/config.dat
2	21 kB	/var/cache/debconf/config.dat-old
2	16 kB	/etc/ld.so.cache
2	14 kB	/var/lib/apt/extended_states

^C Quit | Tab Switch view | ^F Filter | Space Collapse dir | ^Space Collapse all dir | ^A Added | ^R Removed | ^M Modified | ^U Unmodified | ^B Attr

Nginx和debian镜像

- Nginx镜像基于 debian:bullseye-slim构建
- debian:bullseye-slim 只是把根文件系统复制到一个空的文件夹
- 将Debian用户空间程序和host的内核相结合，容器表现的像一个完整的操作系统

```
FROM debian:bullseye-slim

LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION 1.23.1
ENV NJS_VERSION 0.7.6
ENV PKG_RELEASE 1~bullseye
...
```

snappify.io

```
FROM scratch
ADD rootfs.tar.xz /
CMD ["bash"]
```

snappify.io

不使用 镜像运 行容器

- runc只需要一个普通的目录，里面至少有一个可执行文件和一个config.json
- 这个组合被称为bundle

```
# tree
.
├── config.json
└── rootfs
    └── hello
# cat config.json
{
  "ociVersion": "1.0.2-dev",
  "process": {
    "terminal": false,
    "user": {
      "uid": 0,
      "gid": 0
    },
    "args": [
      "/hello"
    ],
    "env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "TERM=xterm"
    ],
    "cwd": "/",
    "capabilities": {
      ...
    }
  }

# runc create mycontainer
# runc start mycontainer
Hello world!
# runc list
ID          PID    STATUS    BUNDLE          CREATED          OWNER
mycontainer 0      stopped  /root/docker/my_container  2022-08-22T06:10:03.102507972Z  root
# runc delete mycontainer
```

为什么需要镜像

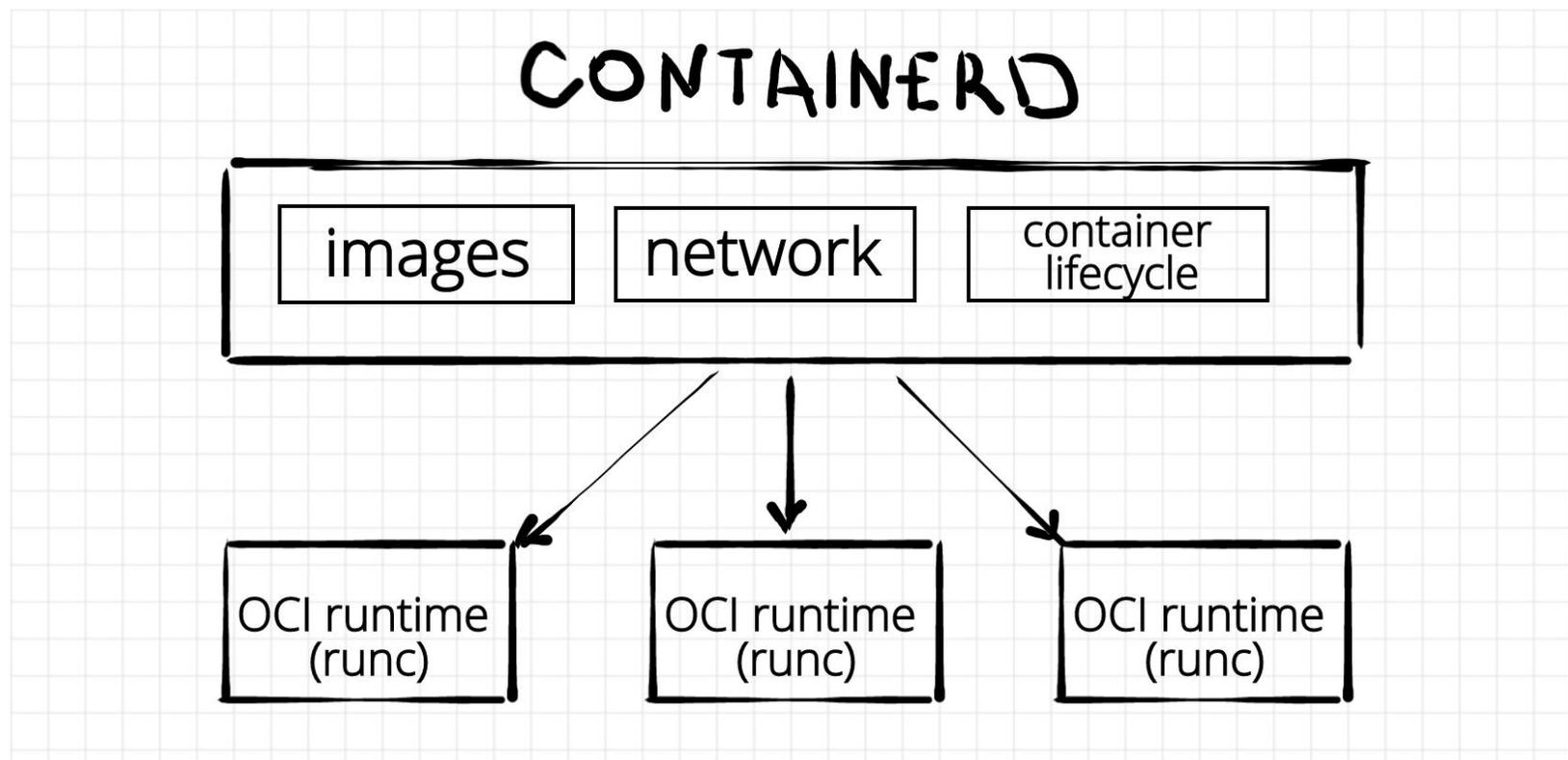
- 镜像是Docker的主要创新
 - 分层结构
 - 层的不可变性
- 镜像解决了高效构建和存储的问题
 - 消除文件的重复，缩短构建时间，减少空间需求，
 - 由于层的不可变性，我们只需要解压一次镜像，然后根据需要多次加载
 - 唯一可变的层是最上面的层

容器运行时

- 使用runc可以启动任意多的容器
- 需要一个容器管理器，管理多个容器
 - 跟踪容器的状态
 - 失败时重启容器
 - 容器终止时释放资源
 - 拉取镜像
 - 容器网络配置

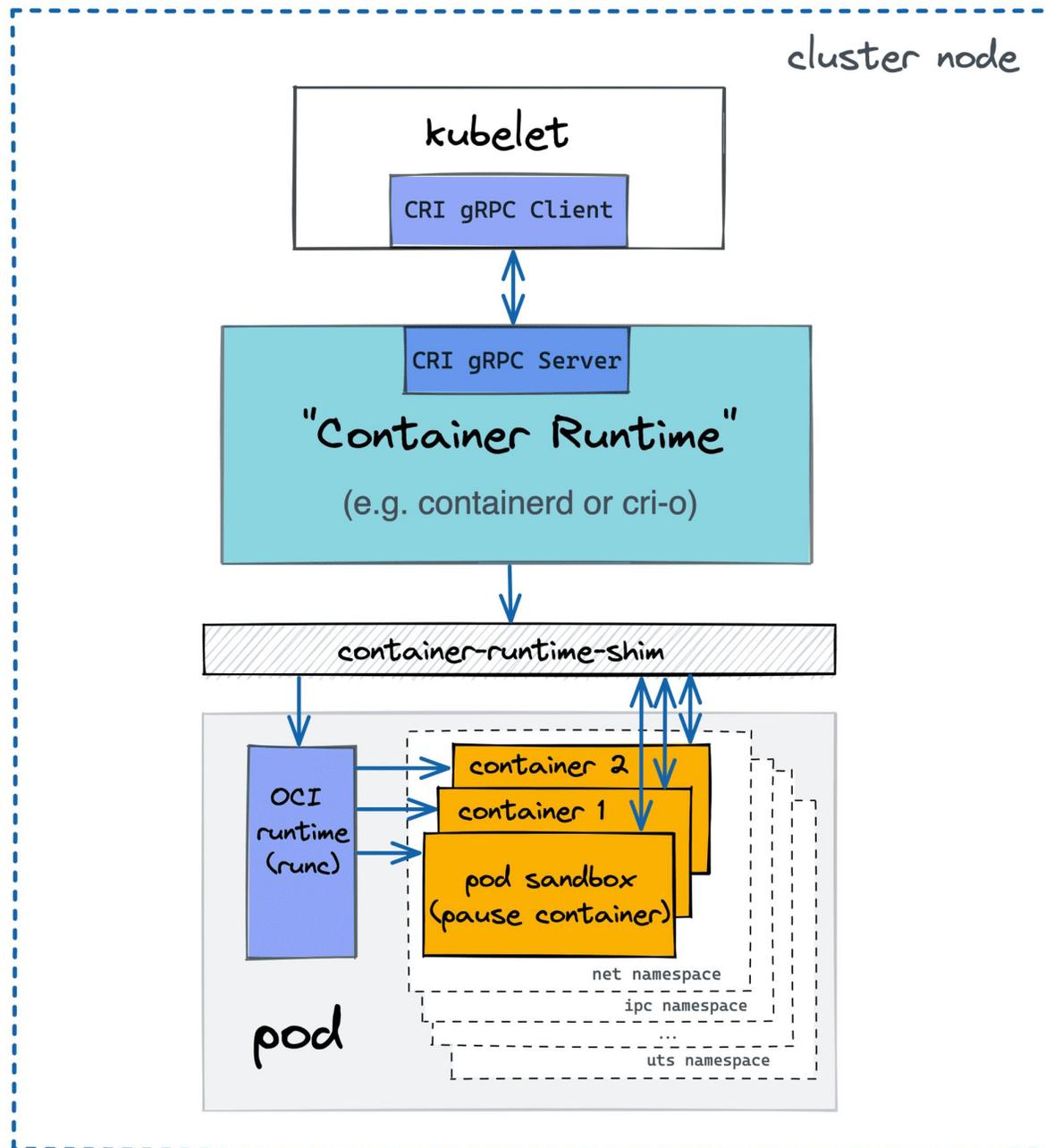
容器管理器

- 在单个主机上进行容器生命周期管理的软件
- 常见的容器管理器
 - containerd
 - cri-o
- runc是一个命令行工具，containerd是一个守护进程



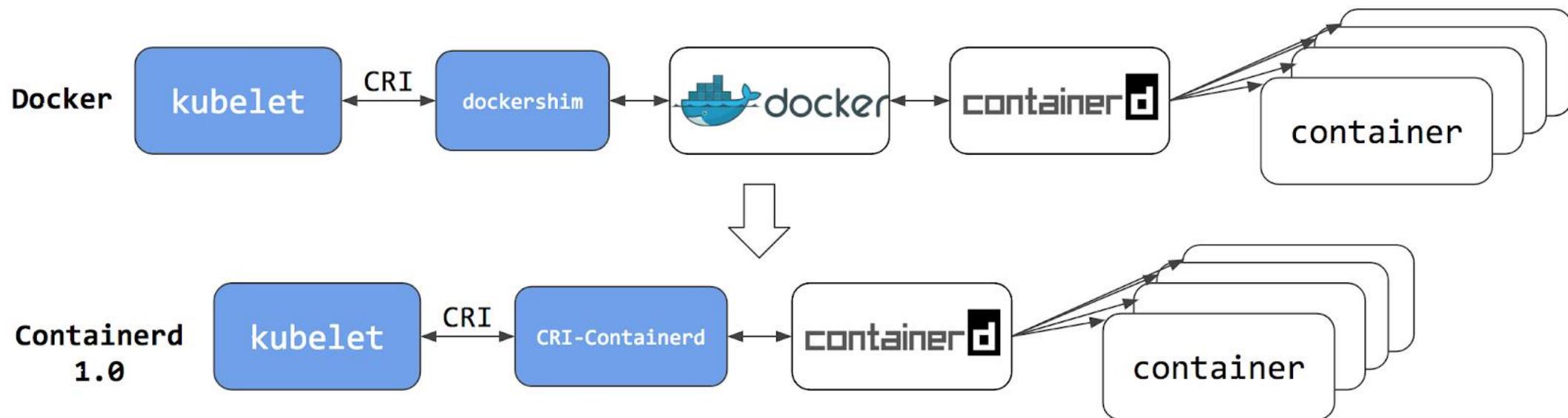
容器管理器

- Container Runtime Interface (CRI) 任何符合 CRI 的容器管理器都可以无缝接入进 Kubernetes
- Kubernetes 通过 CRI 使用不同的容器管理器



容器管理器

- Kubernetes已去除对docker的依赖



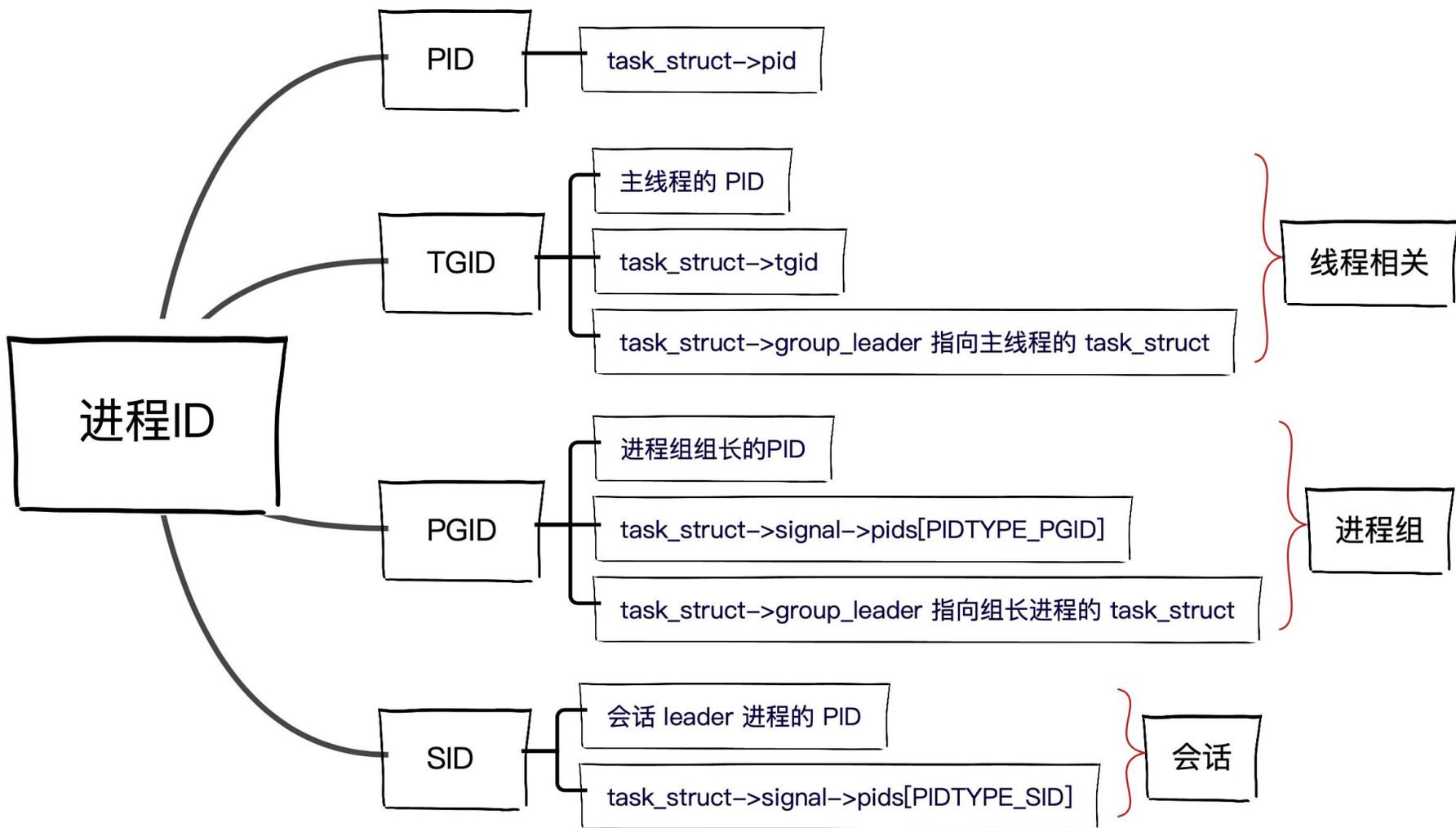
Runtime shim

- 容器管理器可以由于崩溃、更新或其他一些原因而被重新启动，不能直接从容器管理器进程中启动runc
- 需要一个与容器进程一样长时间运行的辅助进程，Runtime shim
- shim是一个轻量级的守护程序，它启动runc并控制容器进程
- shim与容器进程紧密结合，但与管理器的进程完全分离
- 容器和容器管理器之间的所有通信都是通过 shim 进行的

shim的职责

- 启动runc并处理容器创建错误
- 将容器的stdout和stderr定向到日志文件
- 跟踪并报告容器的退出代码。容器进程将被reparent到shim，这样shim可以收到容器进程终止的通知
- 提供attach到运行中容器的服务

进程ID



进程间关系

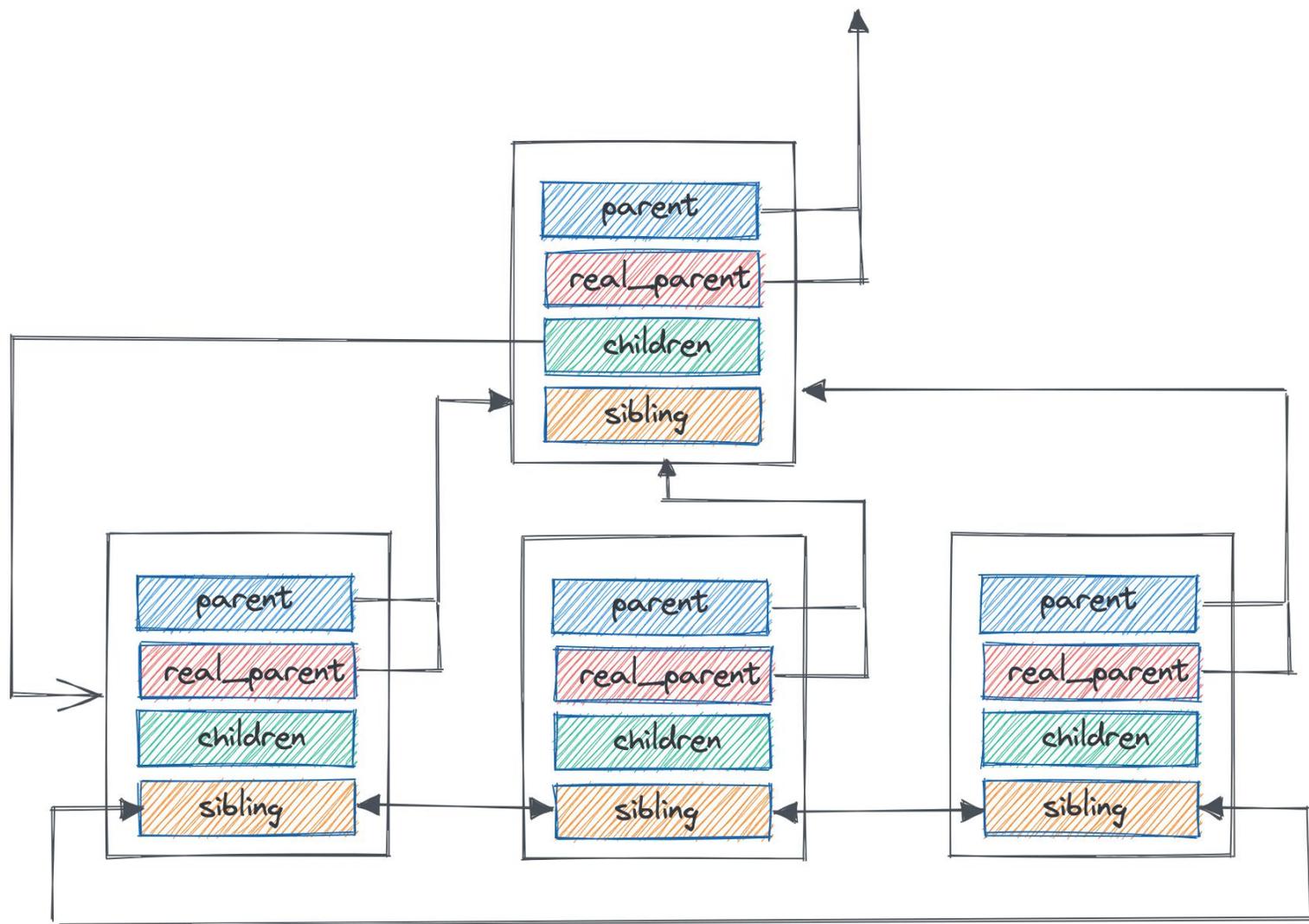
- 内核中所有进程的 `task_struct` 会形成多种组织关系
 - 亲属关系
 - 会话和进程组关系

亲属关系

- 进程通过 `fork()` 创建一个子进程，就形成父子关系，`fork` 出多个子进程，那么这些子进程间属于兄弟关系
- 当一个进程创建了子进程后，它应该通过系统调用 `wait()` 或者 `waitpid()` 等待子进程结束，回收子进程的资源。而子进程在结束时，会向它的父进程发送 `SIGCHLD` 信号。因此父进程还可以注册 `SIGCHLD` 信号的处理函数，异步回收资源。
- 如果父进程提前结束，那么子进程将把1号进程 `init` 作为父进程。
- 进程都有父进程，负责进程结束后的资源回收。在子进程退出且父进程完成回收前，子进程变成僵尸进程。僵尸进程持续的时间通常比较短，在父进程回收它的资源后就会消亡。如果父进程没有处理子进程的终止，那么子进程就会一直处于僵尸状态。

亲属关系

- 进程描述符 `task_struct` 的 `parent` 指向父进程，`children` 指向子进程链表的头部，`sibling` 把当前进程插入到兄弟链表中
- 通常情况下，`real_parent` 和 `parent` 是一样的。如果在 `bash` 上使用 `GDB` 来 `debug` 一个进程，这时候进程的 `parent` 是 `GDB`，进程的 `real_parent` 是 `bash`



会话、进程组关系

- Linux 系统中可以有多个会话 (session) ，每个会话可以包含多个进程组，每个进程组可以包含多个进程。
- 会话是用户登录系统到退出的所有活动，从登录到结束前创建的所有进程都属于这次会话。登录后第一个被创建的进程 (通常是 shell) ，被称为会话 leader
- 进程组用于作业控制。一个终端上可以启动多个进程组。
- 会话有一个前台进程组，还可以有一个或多个后台进程组。只有前台进程可以从终端接收输入，也只有前台进程才被允许向终端输出。

会话、进程组关系

- 通过 `cat | head` 创建了第一个进程组，包含 `cat` 和 `head` 两个进程。
- 按下 `Ctrl + z`，会发送信号 `SIGTSTP` 给前台进程组的所有进程，该信号的缺省行为是暂停作业执行。暂停的作业会让出终端，并且进程不会再被调度，直到它们收到 `SIGCONT` 信号恢复执行。
- 通过 `ps j | more` 创建了另一个进程组，包含 `ps` 和 `more` 两个进程。

```
$ cat | head
hello
hello
^Z
[1]+ 已停止      cat | head
$ ps j | more
  PPID    PID    PGID    SID  TTY          TPGID  STAT   UID    TIME  COMMAND
    1522   1532   1532   1532 pts/0        1762  Ss     1000   0:00  -bash
    1532   1760   1760   1532 pts/0        1762  T      1000   0:00  cat
    1532   1761   1760   1532 pts/0        1762  T      1000   0:00  head
    1532   1762   1762   1532 pts/0        1762  R+     1000   0:00  ps j
    1532   1763   1762   1532 pts/0        1762  S+     1000   0:00  more
```

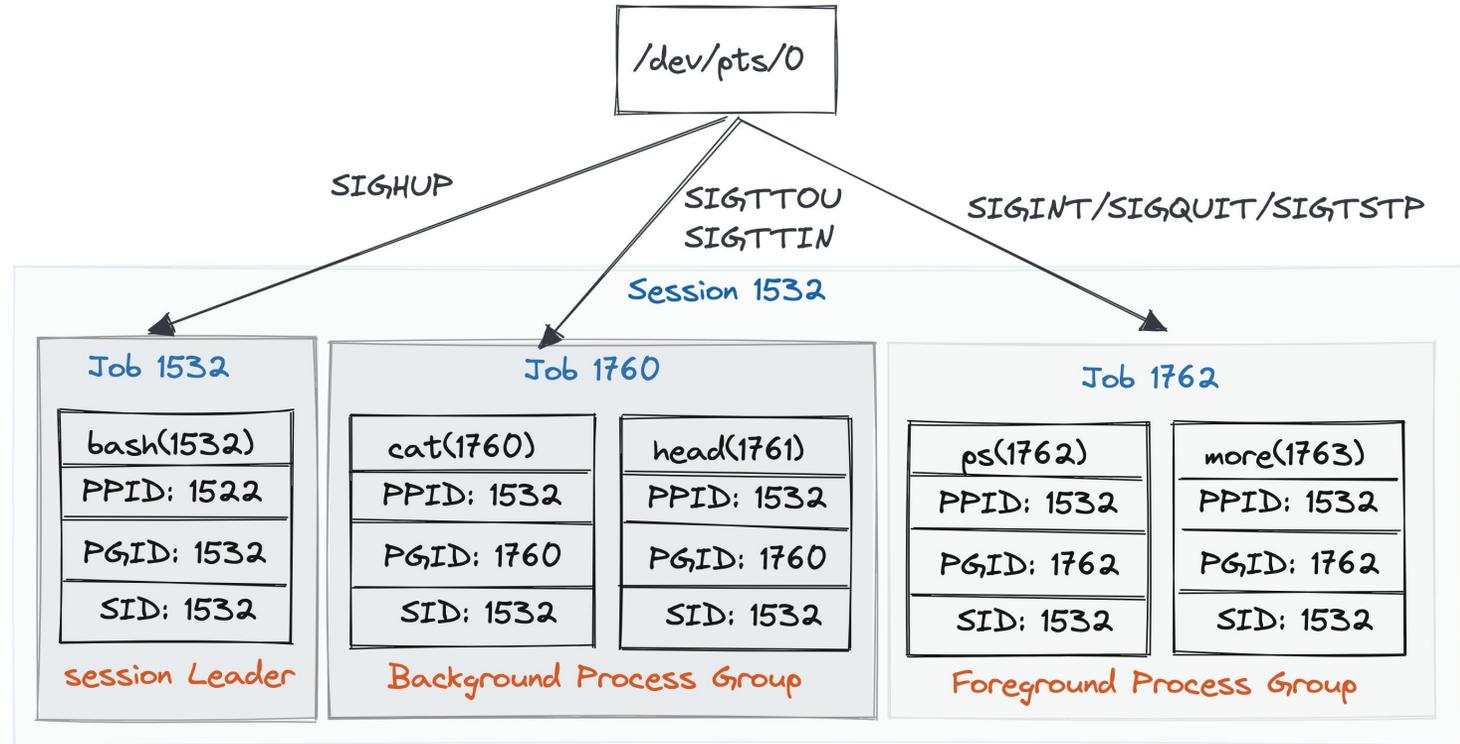
会话、进程组关系

```
# ps 的参数 j 表示用任务格式显示进程。输出中的 STAT 列是进程的状态码
D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not reaped by its parent

# 进程除了大写字母代表的进程状态，还跟着一个附加符号
s : 进程是会话 leader 进程
+ : 进程位于前台进程组中
```

会话、进程组关系

- bash 是这个会话的 leader 进程，它的 PID、PGID 和 SID 相同，都是 1532。这个会话其他所有进程的 SID 也都是 1532。
- cat | head 进程组的 PGID 是 1760
- ps j | more 进程组的 PGID 是 1762
- 用管道连接的进程包含在同一个进程组中，每个进程组内第一个进程成为 Group Leader，并以 Group Leader 的 PID 作为组内进程的 PGID。
- 会话有一个前台进程组，还可以有一个或多个后台进程组，只有前台作业可以从终端读写数据。



如何创建一个守护进程

- 普通进程的stdin、stdout和stderr被连接到会话的控制终端
- 启动守护进程一般使用 [double fork technique](#)
 - fork -> setsid -> fork
 - 主要目的是将守护进程与会话的控制终端分离

shim的demo实现

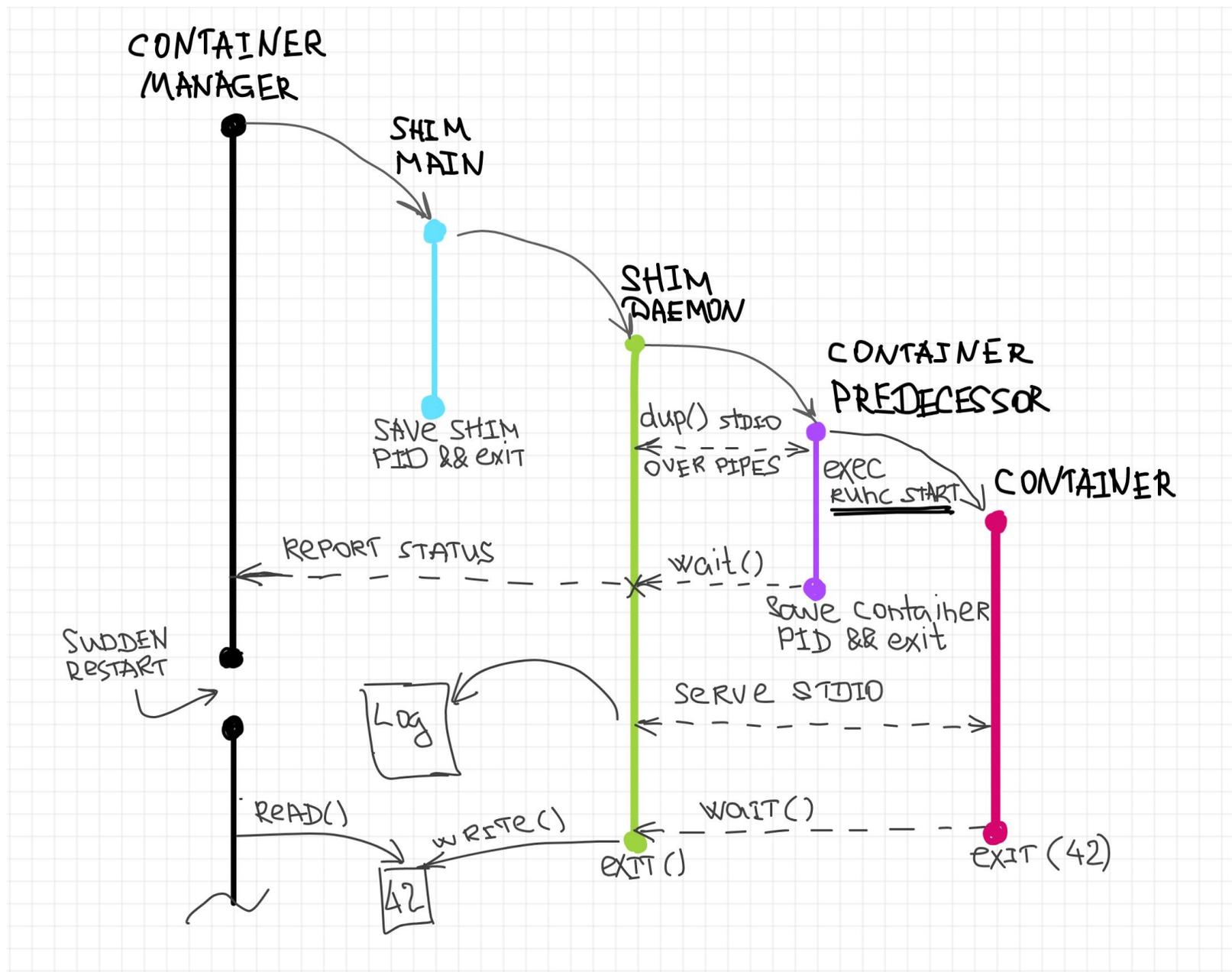
- 前台进程fork出shim守护进程，然后立即退出
- shim守护进程通过创建一个新的session，并将stdio重定向到/dev/null，使stdio从父进程中分离出来
- 创建 pipe，用来将shim与容器的输出连接起来
- 守护进程再fork出一个子进程，即容器进程的前身。这个进程用提供的参数（bundle目录、config.json等）执行runc create 创建容器进程
- 将pipe设置为容器进程前身的stdio，在容器创建成功后，也设置为容器进程的stdio
- 守护进程等待容器进程前身终止，然后将该操作的状态报告给容器管理器。
- 守护进程开始为容器的stdio提供服务，同时也在等待容器的终止

```
int main() {
    if (int shim_pid = fork()) {
        // write shim_pid to file
        exit(0)
    }
    // redirect shim stdio to /dev/null
    dup2("/dev/null", STDIN_FILENO)
    dup2("/dev/null", STDOUT_FILENO)
    dup2("/dev/null", STDERR_FILENO)
    // create a new session
    setsid()
    // prepare pipe_out and pipe_err
    pipe_out = pipe()
    pipe_err = pipe()

    int runtime_pid = fork()
    if (runtime_pid == 0) {
        // redirect runc stdion to pipes
        dup2("/dev/null", STDIN_FILENO)
        dup2(pipe_out, STDOUT_FILENO)
        dup2(pipe_err, STDERR_FILENO)

        exec("runc create --bundle <...>")
    }
    // wait for the runtime process termination:
    waitpid(runtime_pid)
    // ... start serving container process ...
}
```

shim的demo实现



shim 演示

- docker run -d nginx
- ps axfo pid,ppid,command

```
2850634      1 /usr/bin/containerd-shim-runc-v2 -namespace moby -id ee0069f40fe9ee2df211e5176a631
2850655 2850634  \_ nginx: master process nginx -g daemon off;
2850714 2850655  \_ nginx: worker process
2850715 2850655  \_ nginx: worker process
```

- shim成为守护进程， parent PID被重新指定为PID 1

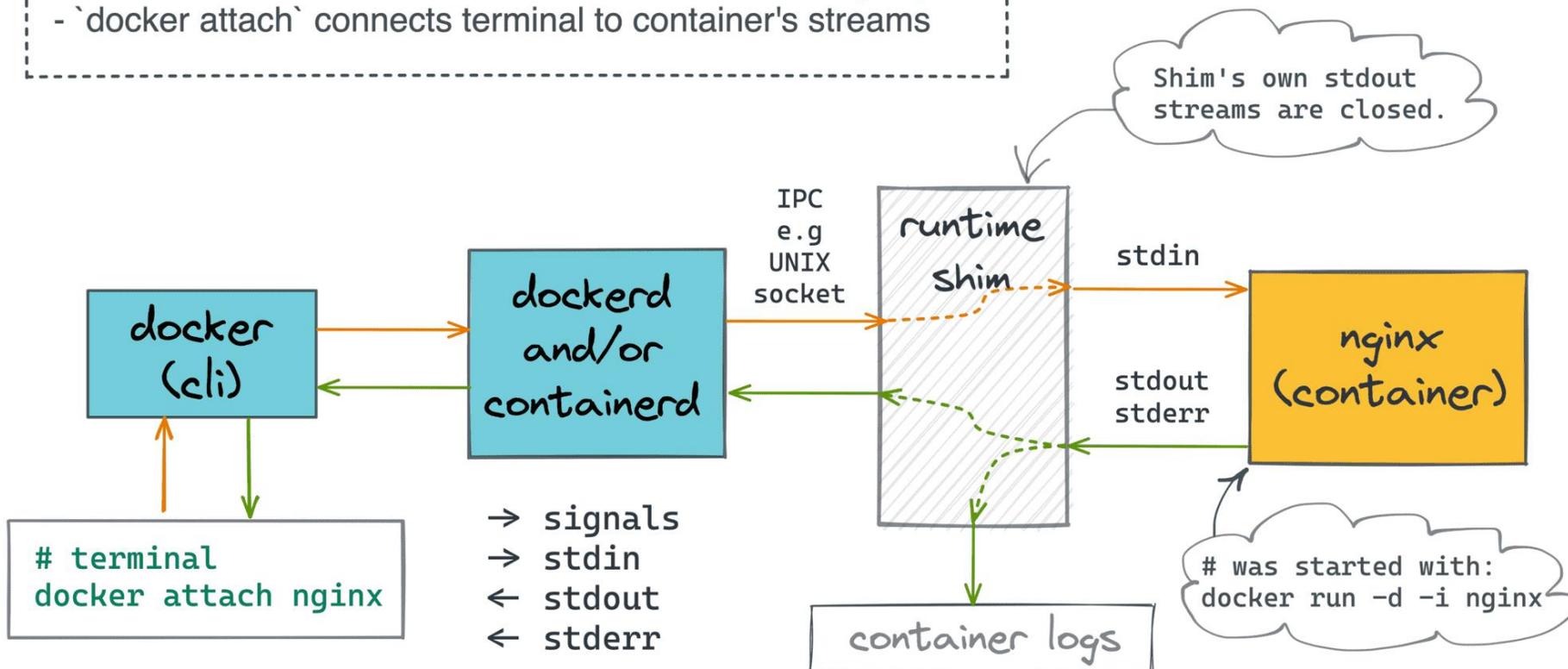
shim演示

- shim进程从容器的stdout和stderr中读取内容，将它们存储到日志
- 默认情况下，shim会关闭容器的stdin，但如果在docker run命令中设置了-i参数，stdin会保持开放
- shim充当了服务器的角色，可以通过shim暴露的端口使用RPC访问
- 当你连接到shim时，它会将容器的stdout和stderr内容回传给你
- 如果一个容器是以交互式模式 (-i) 创建的，那么当你attach到容器后，你在终端中输入的所有内容都将被发送到容器的stdin中

attach的实现

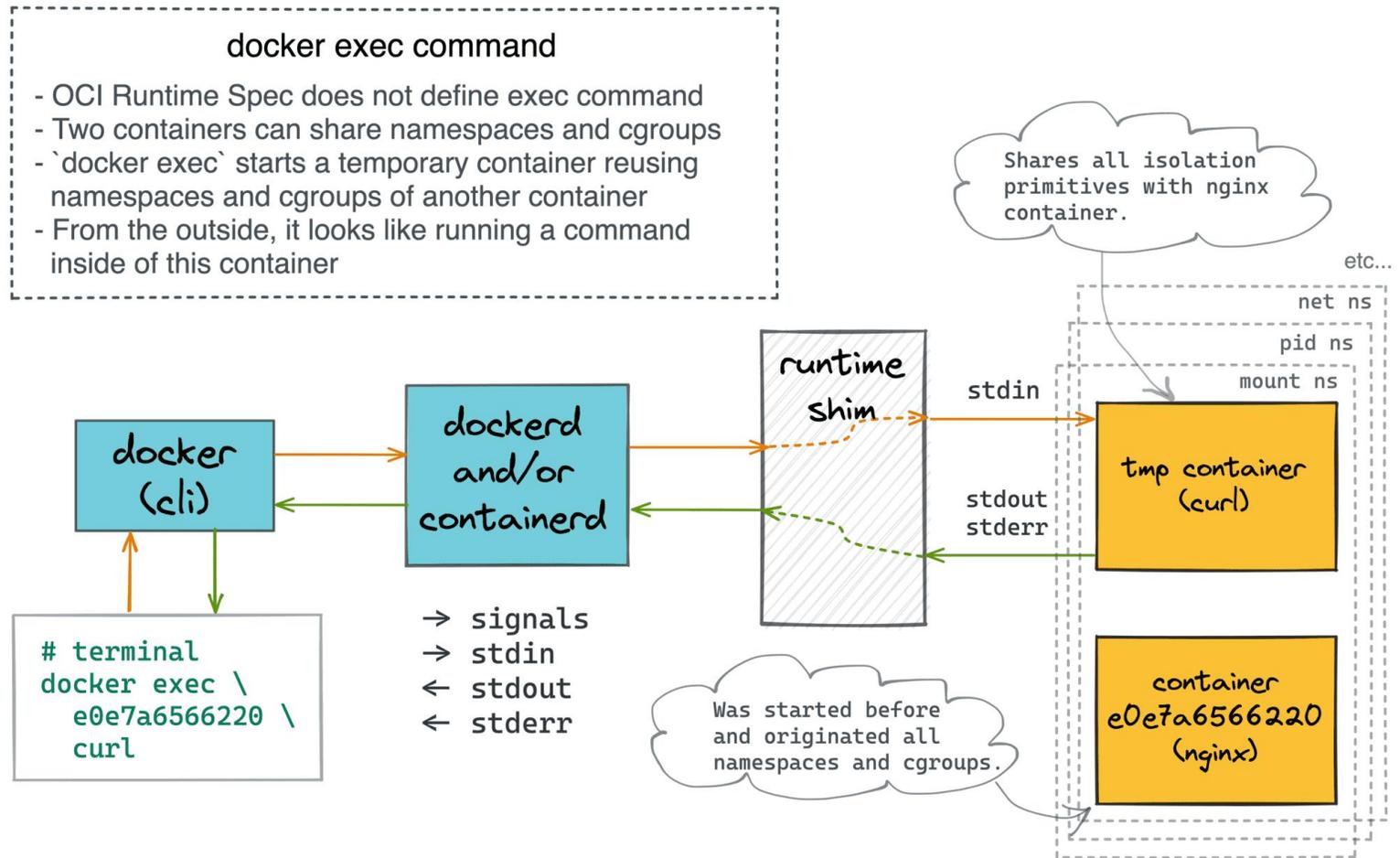
docker attach command

- There is a regular process inside a container
- Every process has stdio streams
- Even in detached mode, container's streams are kept open
- `docker attach` connects terminal to container's streams



exec的实现

- exec命令启动的是一个全新的容器
- 新容器和目标容器共享同样的namespaces和同样的cgroups层次结构
- 从外面看，感觉就像在一个现有的容器内运行一个命令



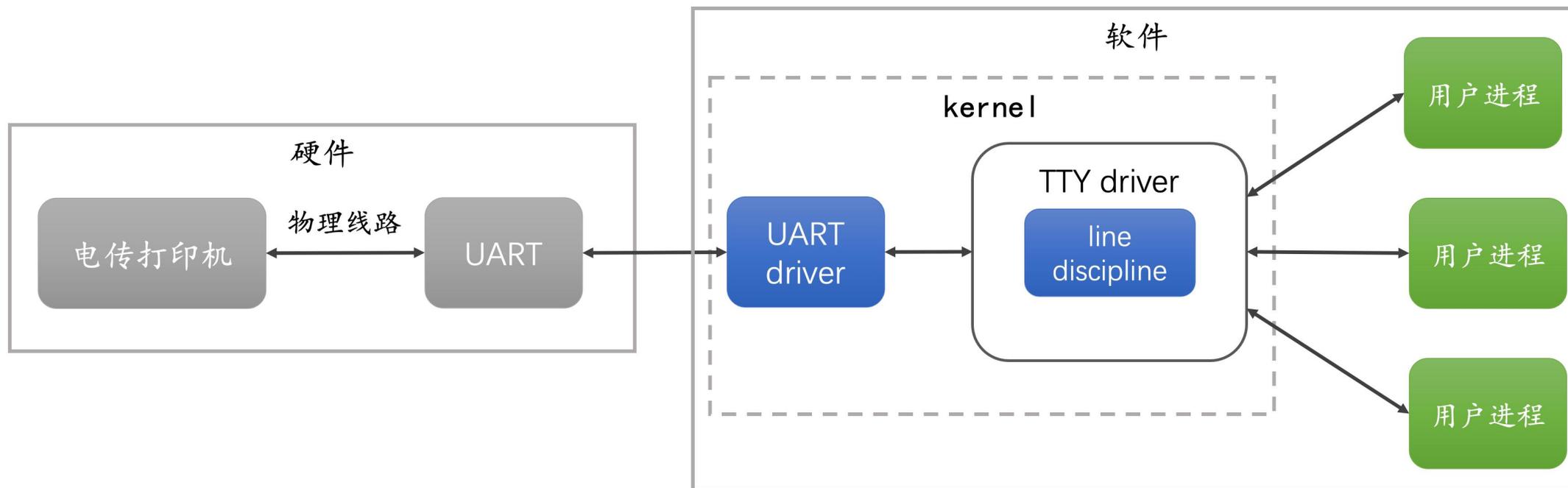
-t 参数的作用是什么

- --tty , -t Allocate a pseudo-TTY

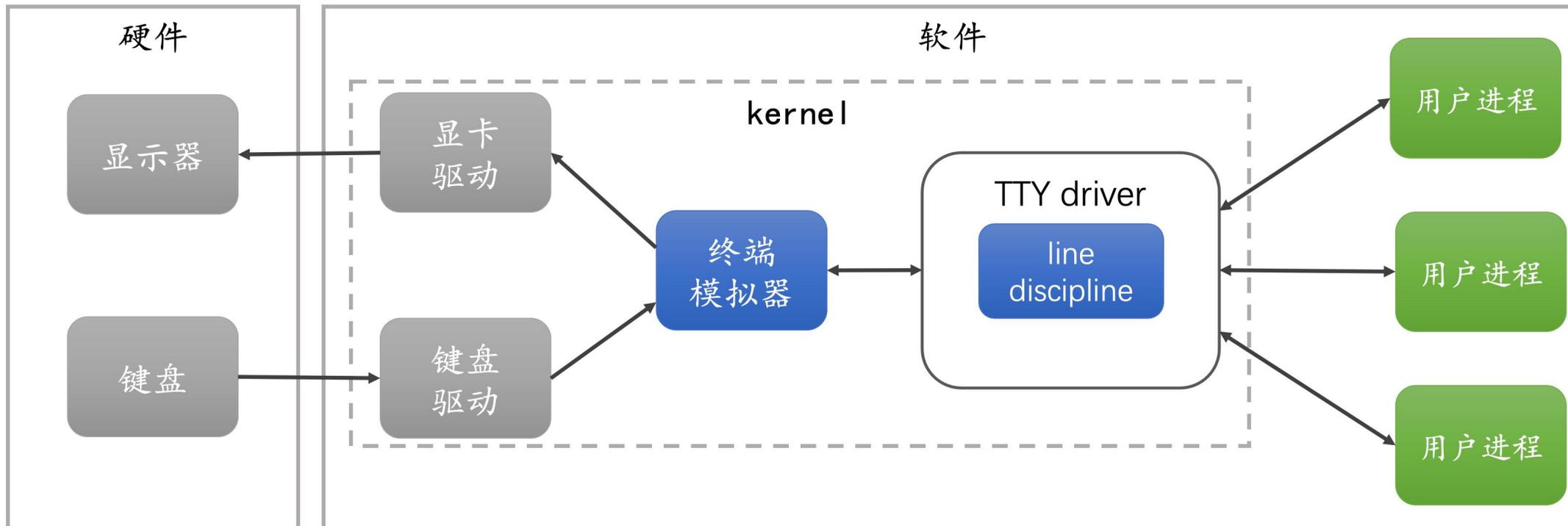


```
# docker run -i -t --name mycontainer ubuntu
root@e3a51aef2a7d:/# echo hello world
hello world
root@e3a51aef2a7d:/#
```

Linux终端

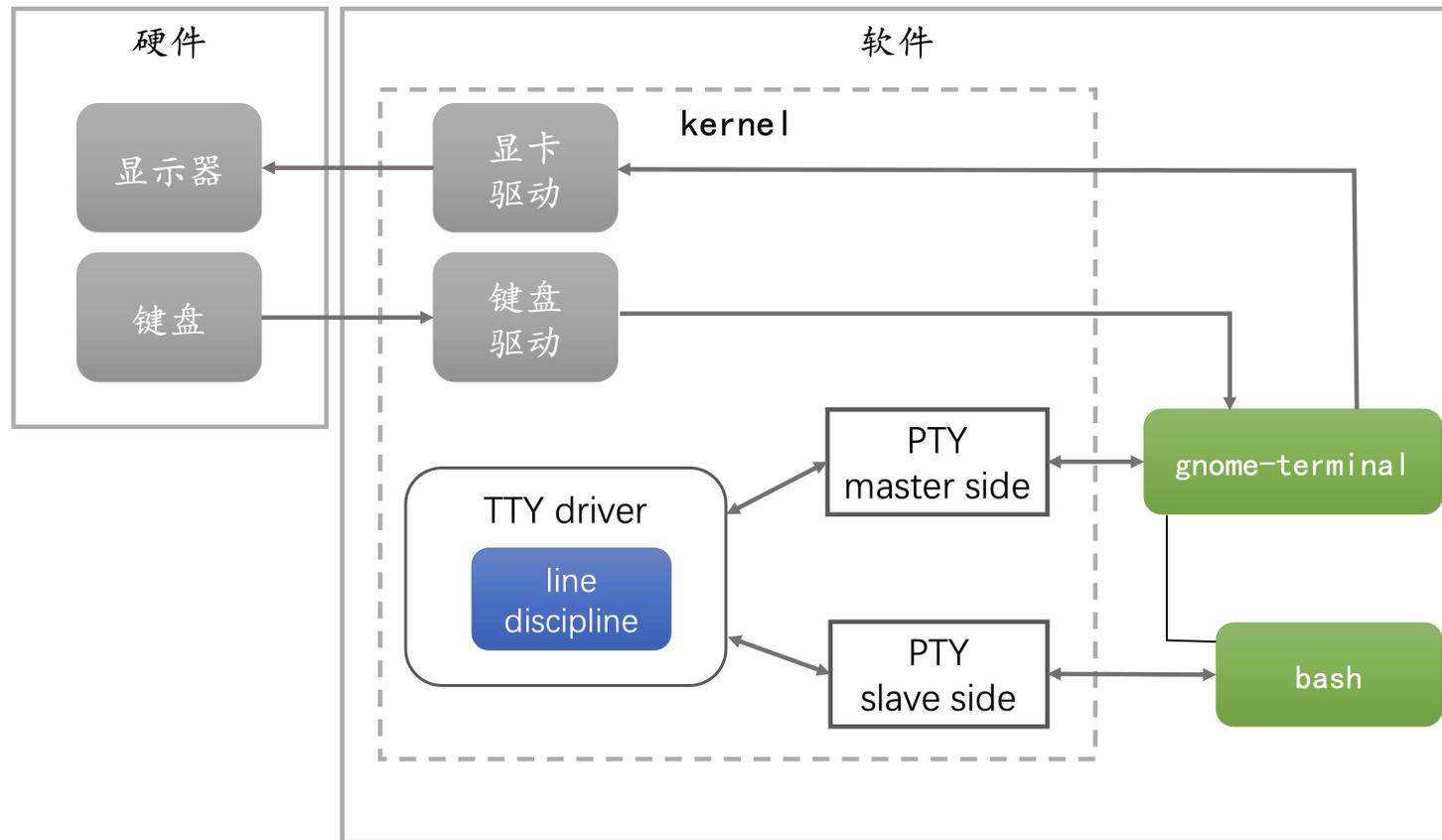


终端模拟器(terminal emulator)



伪终端 (pseudo terminal, PTY)

- PTY 是通过打开特殊的设备文件 /dev/ptmx 创建，由一对双向的字符设备构成，称为 PTY master 和 PTY slave
- terminal 持有 PTY master，通过 PTY master 接收或发送字符到 PTY slave
- terminal 会 fork 一个 shell 子进程，并让 shell 持有 PTY slave
- shell 通过 PTY slave 接收字符，解释执行，并输出处理结果

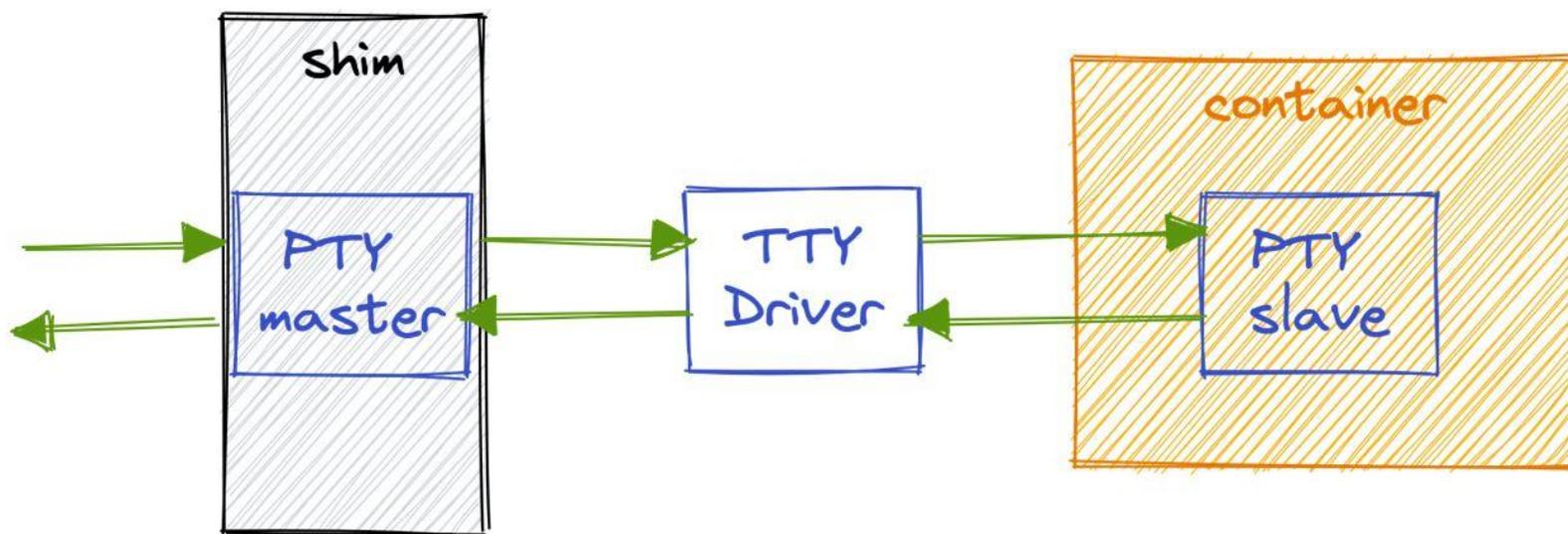


为什么需要PTY

- 实际上我们需要的是TTY driver
- 当用户启动一个普通程序时，它的stdio将被连接到PTY
 - 用户在PTY输入的所有内容都会进入程序的标准输入
 - 程序打印的所有内容都会返回到PTY
- PTY负责信号处理
 - 当用户在键盘上按下Ctrl-C时，通过TTY driver转换为SIGINT信号，发给前台进程
- PTY负责转换特殊字符（如退格、擦除字、清空行）
- PTY负责对字符进行缓冲
 - 按下回车键时，缓冲的数据才会传递给前台进程

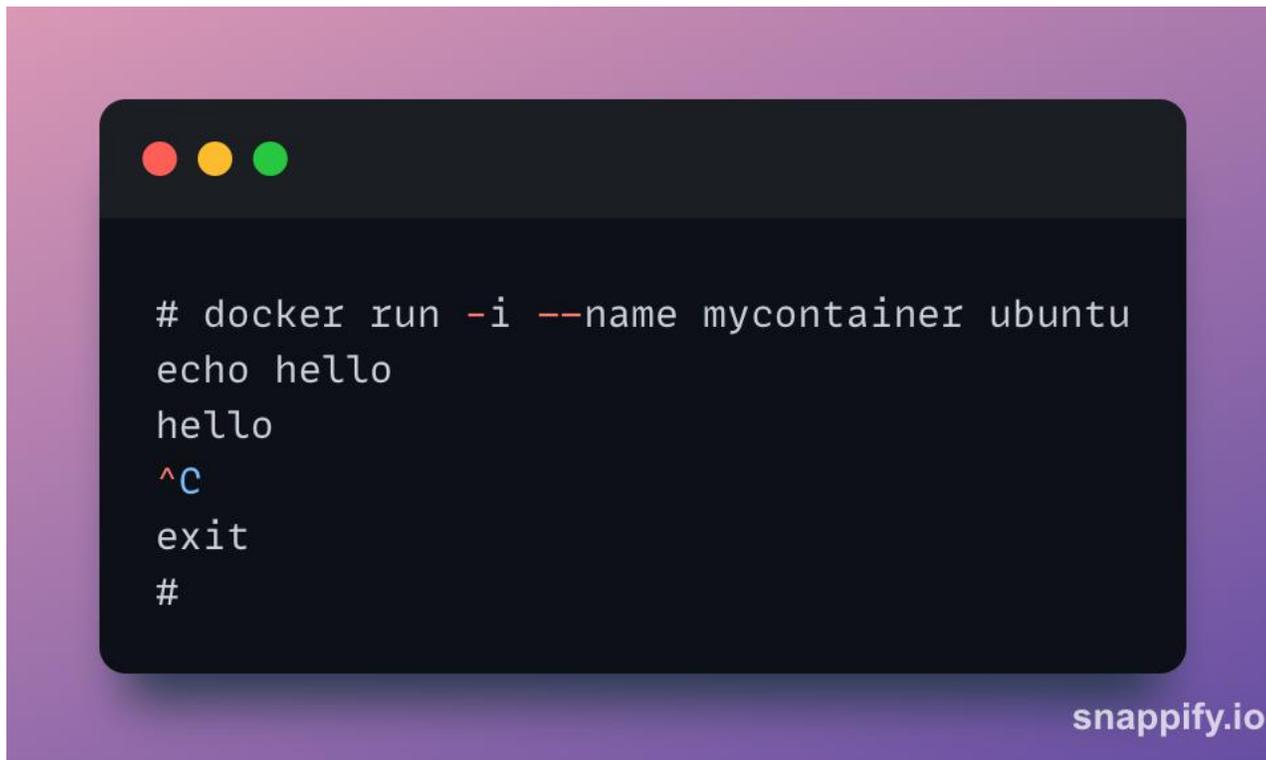
-t 参数的作用

- -t 参数让容器进程持有PTY slave, shim持有PTY master
- 当执行attach的时候
 - stdin的路径: PTY master -> TTY driver-> PTY slave ->容器进程
 - stdout/stderr则是相反方向



不加-t 参数会怎么样?

- 没有终端提示符
- 仍然能接收输入(-i 打开了 stdin)
- 仍然能执行命令并输出
- 没有TTY driver, 不能将Ctrl-C转换为信号

A terminal window with a dark background and light text. The window has three colored window control buttons (red, yellow, green) at the top left. The text inside the terminal shows a Docker command being executed, followed by the output of the command. The command is '# docker run -i --name mycontainer ubuntu echo hello'. The output is 'hello'. Then, a blue '^C' is entered, followed by 'exit' and a '#' prompt.

```
# docker run -i --name mycontainer ubuntu echo hello
hello
^C
exit
#
```

snappify.io

To Be Continued

- 容器网络
- [Containers unplugged](#)
 - Containers unplugged: Linux namespaces
 - Containers unplugged: understanding user namespaces
 - Containers unplugged: An introduction to control groups
 - Containers unplugged: Using seccomp to limit the kernel attack surface
 - Containers unplugged: The Linux capabilities model
 - Containers unplugged: Privileged programs



天燕讲堂第3期 《深入浅出容器技术》
满意度调研



感谢您的参与!