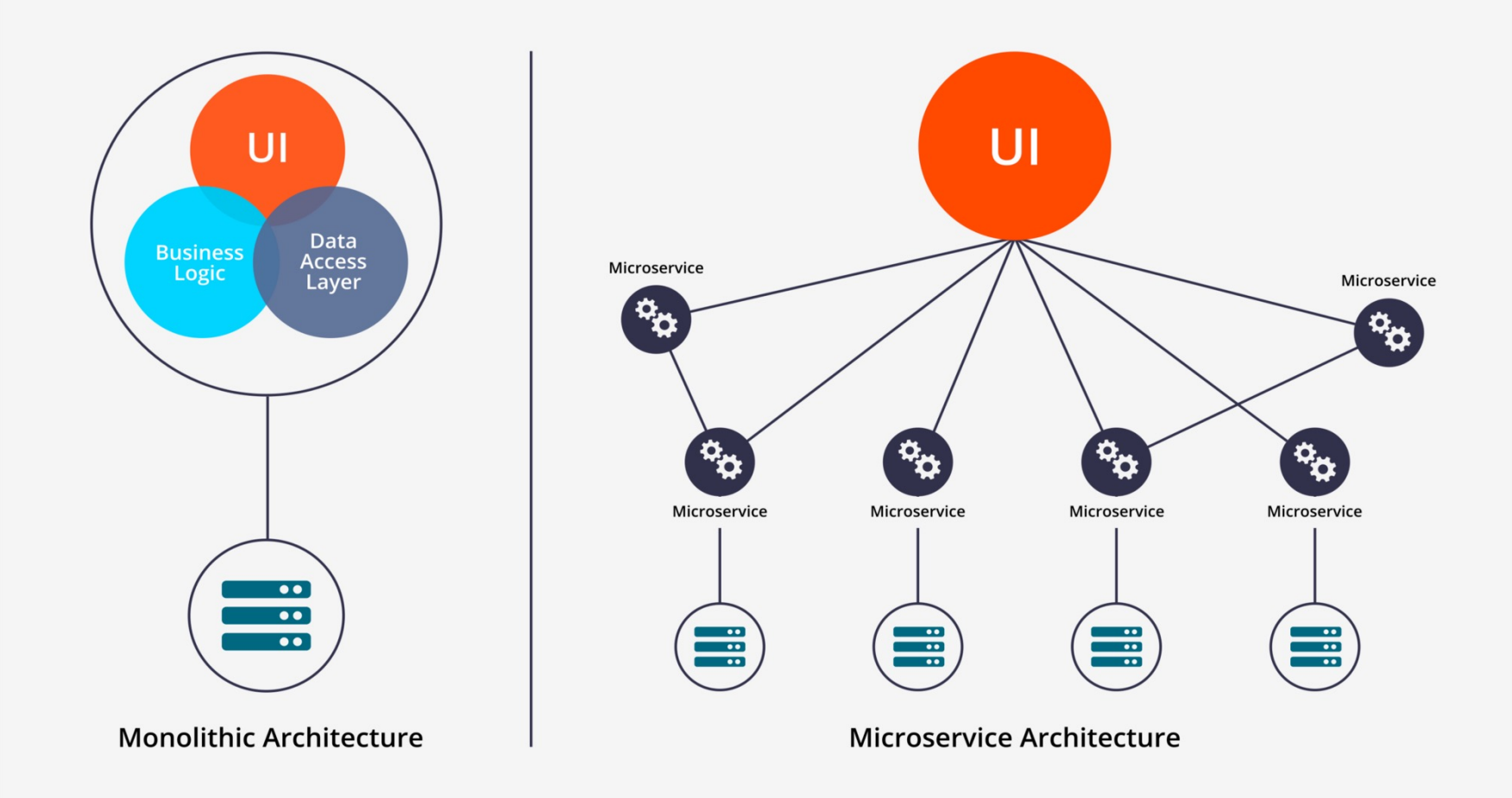
Docker 核心技术

孟凡杰

eBay 资深架构师

传统分层架构 vs 微服务



Docker

极客时间 | 训练营

- 基于 Linux 内核的 Cgroup，Namespace，以及Union FS 等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术，由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。
- 最初实现是基于 LXC，从 0.7 以后开始去除 LXC，转而使用自行开发的 Libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 Containerd。
- Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护，使得 Docker 技术比虚拟机技术更为轻便、快捷。

为什么要用 Docker

更高效的利用系统资源

更快速的启动时间

一致的运行环境

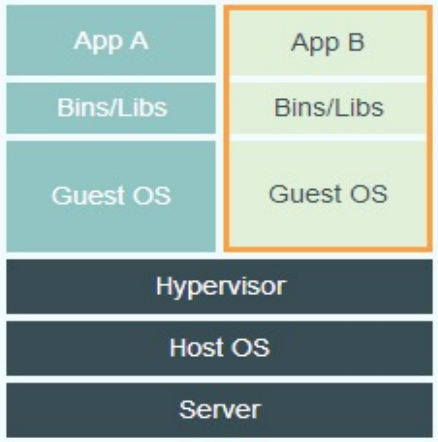
持续交付和部署

更轻松的迁移

更轻松的维护和扩展

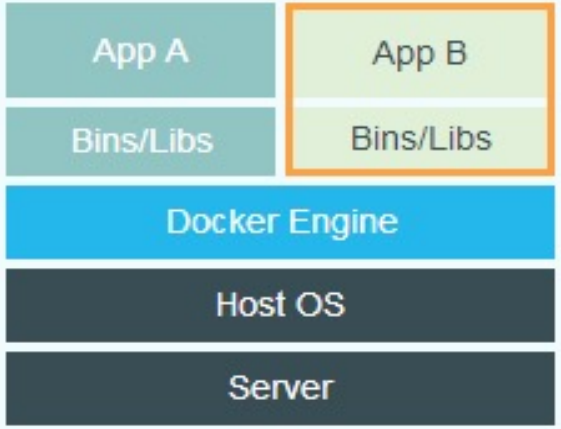
……

虚拟机和容器运行态的对比



Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

容器操作

启动：

- docker run
  - -it 交互
  - -d 后台运行
  - -p 端口映射
  - -v 磁盘挂载
- 启动已终止容器

  docker start
- 停止容器

  docker stop
- 查看容器进程

  docker ps

容器操作

极客时间 | 训练营

- 查看容器细节：

  docker inspect <containerid>

- 进入容器：

  docker attach

  docker exec

- 通过 nsenter：

  PID=$(docker inspect --format "{{ .State.Pid }}" <container>)

  $ nsenter --target $PID --mount --uts --ipc --net --pid

- 拷贝文件至容器内：

  docker cp file1 <containerid>:/file-to-path

初识容器

极客时间 | 训练营

- cat Dockerfile

```
FROM ubuntu
ENV MY_SERVICE_PORT=80
ADD bin/amd64/httpserver /httpserver
ENTRYPOINT /httpserver
```

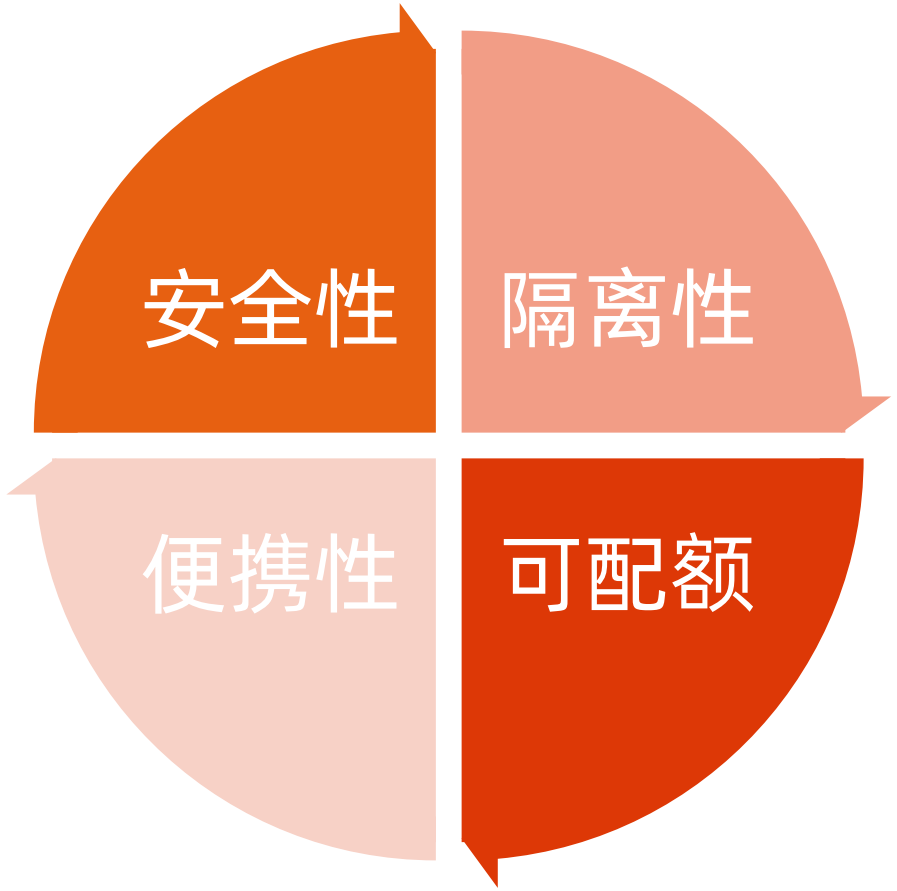
- 将Dockerfile打包成镜像

```
docker build -t cncamp/httpserver:${tag} .
docker push cncamp/httpserver:v1.0
```

- 运行容器

```
docker run -d cncamp/httpserver:v1.0
```

容器主要特性



Namespace

极客时间 | 训练营

- Linux Namespace 是一种 Linux Kernel 提供的资源隔离方案：
  - 系统可以为进程分配不同的 Namespace；
  - 并保证不同的 Namespace 资源独立分配、进程彼此隔离，即不同的 Namespace 下的进程互不干扰 。

Linux 内核代码中 Namespace 的实现

- 进程数据结构

```
struct task_struct {
    ...
    /* namespaces */
    struct nsproxy *nsproxy;
    ...
}
```

- Namespace数据结构

```
struct nsproxy {
    atomic_t count;
    struct uts_namespace *uts_ns;
    struct ipc_namespace *ipc_ns;
    struct mnt_namespace *mnt_ns;
    struct pid_namespace
*pid_ns_for_children;
    struct net       *net_ns;
}
```

Linux 对 Namespace操作方法

- clone

  在创建新进程的系统调用时，可以通过 flags 参数指定需要新建的 Namespace 类型：

  // CLONE_NEWCGROUP / CLONE_NEWIPC / CLONE_NEWNET / CLONE_NEWNS / CLONE_NEWPID / CLONE_NEWUSER / CLONE_NEWUTS

  int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)

- setns

  该系统调用可以让调用进程加入某个已经存在的 Namespace 中：

  Int setns(int fd, int nstype)

- unshare

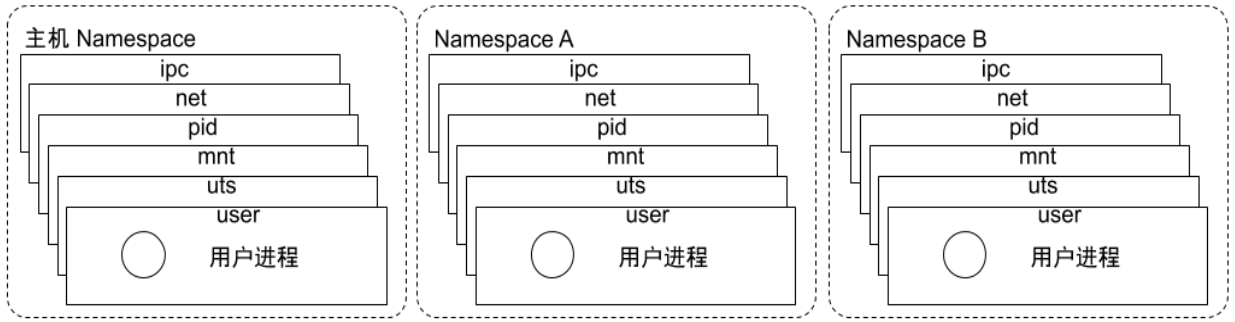
  该系统调用可以将调用进程移动到新的 Namespace 下：

  int unshare(int flags)

隔离性 - Linux Namespace

极客时间 | 训练营

Namespace 类型	隔离资源	Kernel 版本
IPC	System V IPC 和 POSIX 消息队列	2.6.19
Network	网络设备、网络协议栈、网络端口等	2.6.29
PID	进程	2.6.14
Mount	挂载点	2.4.19
UTS	主机名和域名	2.6.19
USR	用户和用户组	3.8



关于 namespace 的常用操作

- 查看当前系统的 namespace：

  lsns -t <type>

- 查看某进程的 namespace：

  ls -la /proc/<pid>/ns/

- 进入某 namespace 运行命令：

  nsenter -t <pid> -n ip addr

Namespace 练习

极客时间 | 训练营

- 在新 network namespace 执行 sleep 指令：

```
unshare -fn sleep 60
```

- 查看进程信息

```
ps -ef|grep sleep
root       32882    4935  0 10:00 pts/0    00:00:00 unshare -fn sleep 60
root       32883   32882  0 10:00 pts/0    00:00:00 sleep 60
```

- 查看网络 Namespace

```
lsns -t net
4026532508 net       2 32882 root unassigned                         unshare
```

- 进入改进程所在 Namespace 查看网络配置，与主机不一致

```
nsenter -t 32882 -n ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Cgroups

- Cgroups （Control Groups）是 Linux 下用于对一个或一组进程进行资源控制和监控的机制；
- 可以对诸如 CPU 使用时间、内存、磁盘 I/O 等进程所需的资源进行限制；
- 不同资源的具体管理工作由相应的 Cgroup 子系统（Subsystem）来实现；
- 针对不同类型的资源限制，只要将限制策略在不同的的子系统上进行关联即可；
- Cgroups 在不同的系统资源管理子系统中以层级树（Hierarchy）的方式来组织管理：每个 Cgroup 都可以包含其他的子 Cgroup，因此子 Cgroup 能使用的资源除了受本 Cgroup 配置的资源参数限制，还受到父 Cgroup 设置的资源限制 。

Linux 内核代码中 Cgroups 的实现

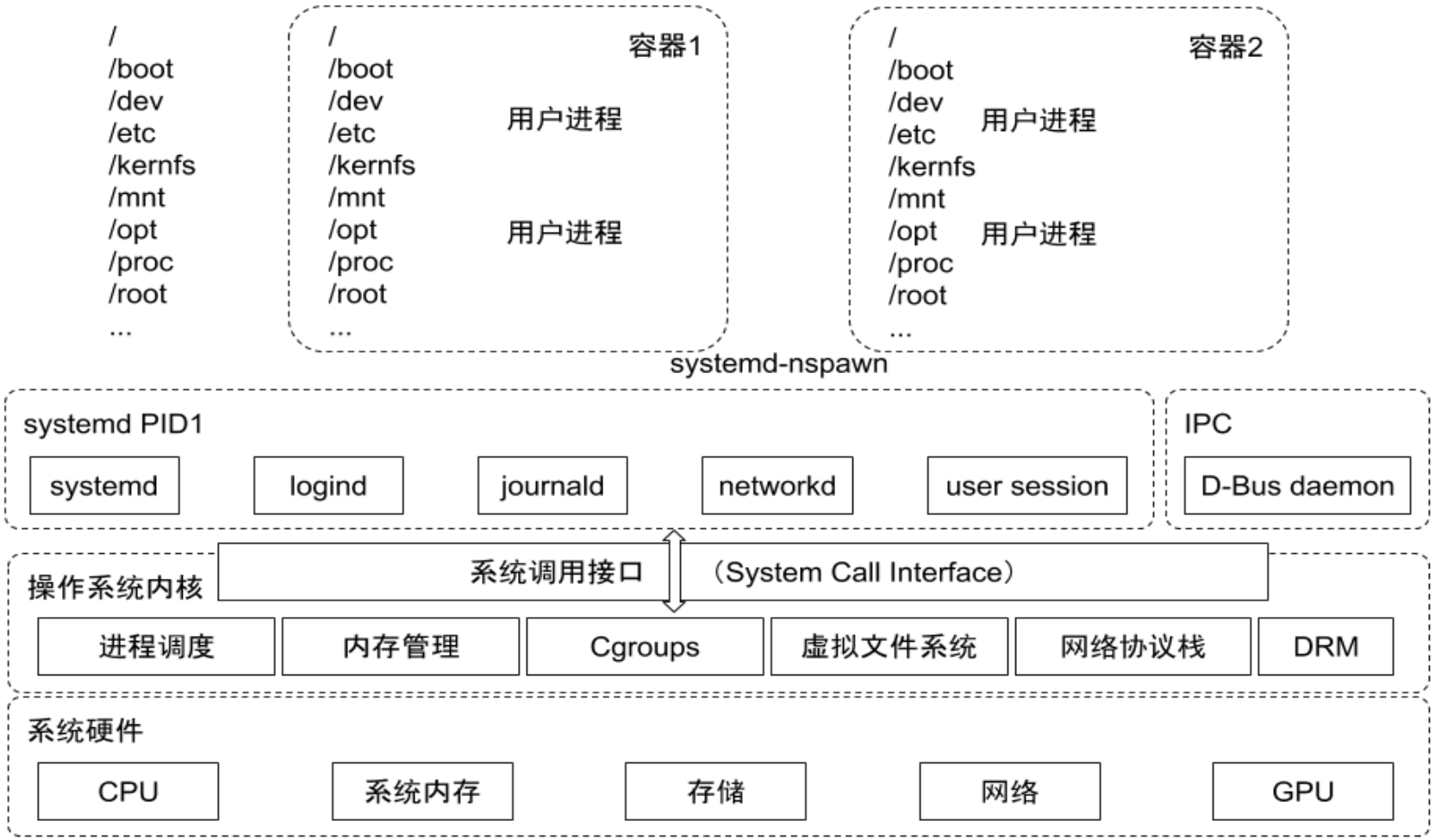
- 进程数据结构

```
struct task_struct
{
    #ifdef CONFIG_CGROUPS
    struct css_set __rcu *cgroups;
    struct list_head cg_list;
    #endif
}
```

- css_set 是 cgroup_subsys_state 对象的集合数据结构

```
struct css_set {
    /*
     * Set of subsystem states, one for each subsystem. This array is
     * immutable after creation apart from the init_css_set during
     * subsystem registration (at boot time).
     */
    struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
};
```

可配额/可度量 - Control Groups (cgroups)



可配额/可度量 - Control Groups (cgroups)

cgroups 实现了对资源的配额和度量。

- blkio：这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及 USB 等等；
- cpu：这个子系统使用调度程序为 cgroup 任务提供 CPU 的访问；
- cpuacct：产生 cgroup 任务的 CPU 资源报告；
- cpuset：如果是多核心的CPU，这个子系统会为 cgroup 任务分配单独的 CPU 和内存；
- devices：允许或拒绝 cgroup 任务对设备的访问；
- freezer：暂停和恢复 cgroup 任务；
- memory：设置每个 cgroup 的内存限制以及产生内存资源报告；
- net_cls：标记每个网络包以供 cgroup 方便使用；
- ns：名称空间子系统；
- pid: 进程标识子系统。

CPU 子系统

cpu.shares：可出让的能获得 CPU 使用时间的相对值。

cpu.cfs_period_us：cfs_period_us 用来配置时间周期长度，单位为 us（微秒）。

cpu.cfs_quota_us：cfs_quota_us 用来配置当前 Cgroup 在 cfs_period_us 时间内最多能使用的 CPU 时间数，单位为 us（微秒）。

cpu.stat：Cgroup 内的进程使用的 CPU 时间统计。

nr_periods：经过 cpu.cfs_period_us 的时间周期数量。

nr_throttled：在经过的周期内，有多少次因为进程在指定的时间周期内用光了配额时间而受到限制。

throttled_time：Cgroup 中的进程被限制使用 CPU 的总用时，单位是 ns（纳秒）。

CPU 子系统练习

极客时间 | 训练营

- 在 cgroup cpu 子系统目录中创建目录结构

cd /sys/fs/cgroup/cpu

mkdir cpudemo

cd cpudemo

- 运行 busyloop
- 执行 top 查看 CPU 使用情况，CPU 占用200%
- 通过 cgroup 限制 CPU

cd /sys/fs/cgroup/cpu/cpudemo

    - 把进程添加到 cgroup 进程配置组

    echo ps -ef|grep busyloop|grep -v grep|awk '{print $2}' > cgroup.procs

    - 设置 cpuquota

    echo 10000 > cpu.cfs_quota_us

- 执行 top 查看 CPU 使用情况，CPU 占用变为 10%

cpuacct 子系统

用于统计 Cgroup 及其子 Cgroup 下进程的 CPU 的使用情况。

- cpuacct.usage

包含该 Cgroup 及其子 Cgroup 下进程使用 CPU 的时间，单位是 ns（纳秒）。

- cpuacct.stat

包含该 Cgroup 及其子 Cgroup 下进程使用的 CPU 时间，以及用户态和内核态的时间。

memory 子系统

- memory.usage_in_bytes

  cgroup下进程使用的内存，包含cgroup及其子cgroup下的进程使用的内存。

- memory.max_usage_in_bytes

  cgroup下进程使用内存的最大值，包含子cgroup的内存使用量。

- memory.limit_in_bytes

  设置Cgroup下进程最多能使用的内存。如果设置为-1，表示对该cgroup的内存使用不做限制。

- memory.oom_control

  设置是否在Cgroup中使用OOM（Out of Memory）Killer，默认为使用。当属于该cgroup的进程使用的内存超过最大的限定值时，会立刻被OOM Killer处理。

memory 子系统练习

极客时间 | 训练营

- 在 cgroup memory 子系统目录中创建目录结构：

```
cd /sys/fs/cgroup/memory
mkdir memorydemo
cd memorydemo
```

- 运行 malloc（在 Linux 机器 make build）；
- 查看内存使用情况；

```
watch 'ps -aux|grep malloc|grep -v grep‘
```

- 通过 cgroup 限制 memory：
    - 把进程添加到 cgroup 进程配置组：

```
echo ps -ef|grep malloc |grep -v grep|awk '{print $2}' > cgroup.procs
```

    - 设置 memory.limit_in_bytes：

```
echo 104960000 > memory.limit_in_bytes
```

- 等待进程被 oom kill。

文件系统

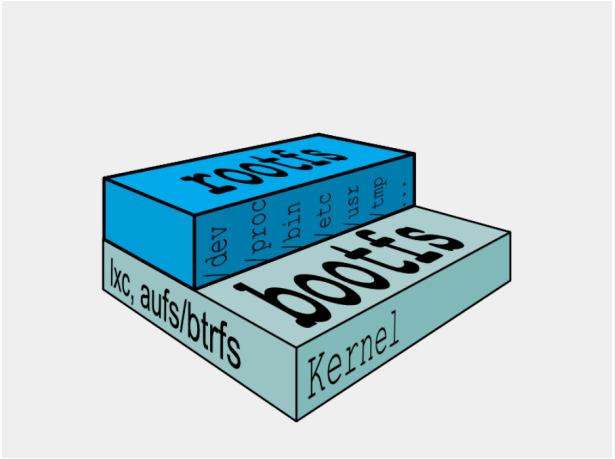
Union FS

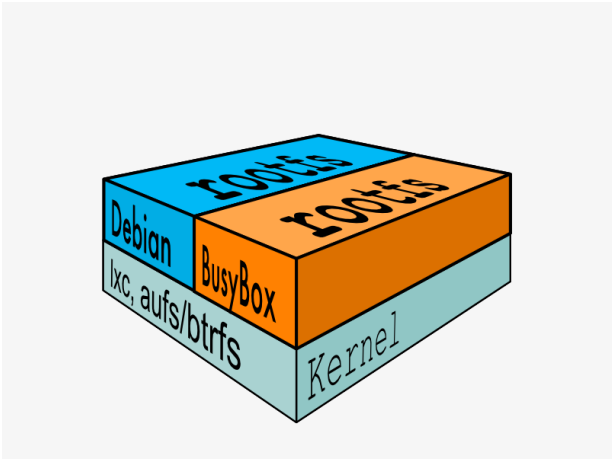
- 将不同目录挂载到同一个虚拟文件系统下 （unite several directories into a single virtual filesystem）的文件系统。
- 支持为每一个成员目录（类似Git Branch）设定 readonly、readwrite 和 whiteout-able 权限。
- 文件系统分层, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。
- 通常 Union FS 有两个用途, 一方面可以将多个disk挂到同一个目录下, 另一个更常用的就是将一个 readonly 的 branch 和一个 writeable 的 branch 联合在一起。

Docker 的文件系统

典型的 Linux 文件系统组成：

- Bootfs（boot file system）
  - Bootloader - 引导加载 kernel，
  - Kernel - 当 kernel 被加载到内存中后 umount bootfs。
- rootfs （root file system）
  - /dev，/proc，/bin，/etc 等标准目录和文件。
  - 对于不同的 linux 发行版, bootfs 基本是一致的，但 rootfs 会有差别。





Docker 启动

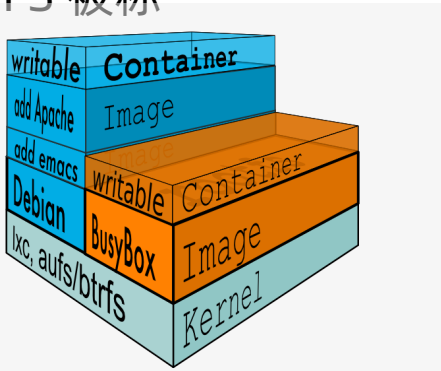
极客时间 | 训练营

Linux

- 在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 “readwrite” 供用户使用。

Docker启动

- 初始化时也是将 rootfs 以 readonly 方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的 rootfs 之上；
- 并且允许再次将下层的 FS(file system) 设定为 readonly 并且向上叠加；
- 这样一组 readonly 和一个 writeable 的结构构成一个 container 的运行时态, 每一个 FS 被称作一个 FS 层。



写操作

由于镜像具有共享特性，所以对容器可写层的操作需要依赖存储驱动提供的写时复制和用时分配机制，以此来支持对容器可写层的修改，进而提高对存储和内存资源的利用率。

- 写时复制

写时复制，即 Copy-on-Write。一个镜像可以被多个容器使用，但是不需要在内存和磁盘上做多个拷贝。在需要对镜像提供的文件进行修改时，该文件会从镜像的文件系统被复制到容器的可写层的文件系统进行修改，而镜像里面的文件不会改变。不同容器对文件的修改都相互独立、互不影响。

- 用时分配

按需分配空间，而非提前分配，即当一个文件被创建出来后，才会分配空间。

容器存储驱动

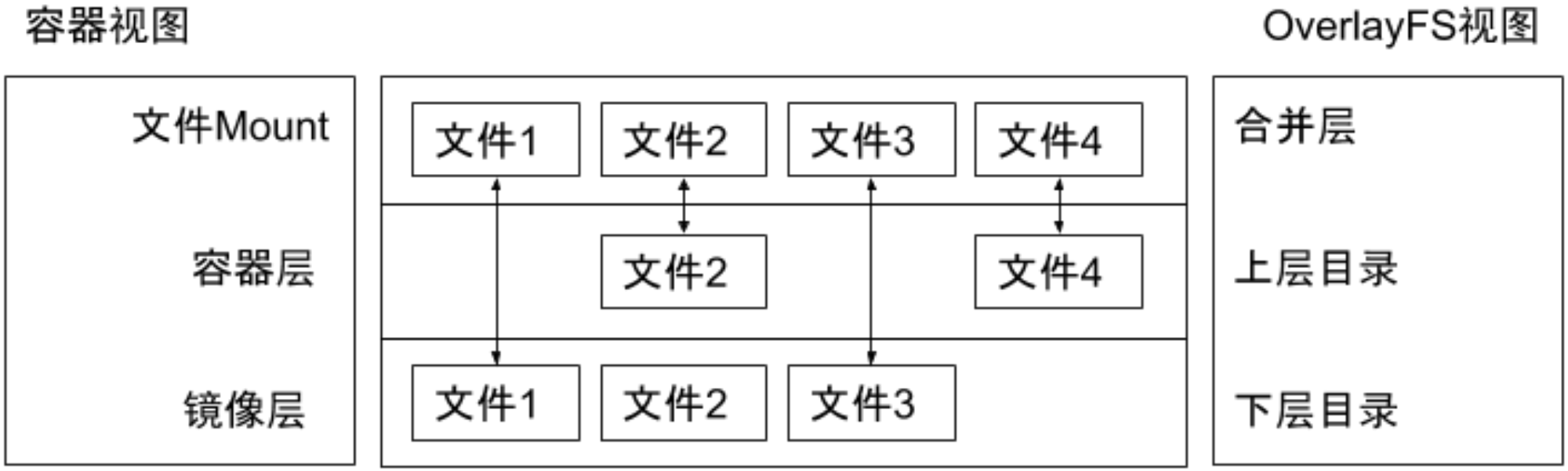
极客时间 | 训练营

存储驱动	Docker	Containerd
AUFS	在 Ubuntu 或者 Debian上支持	不支持
OverlayFS	支持	支持
Device Mapper	支持	支持
Btrfs	社区版本在 Ubuntu 或者 Debian 上支持，企业版本在 SLES 上支持	支持
ZFS	支持	不支持

以 OverlayFS 为例

OverlayFS 也是一种与 AUFS 类似的联合文件系统，同样属于文件级的存储驱动，包含了最初的 Overlay 和更新更稳定的 overlay2。

Overlay 只有两层：upper 层和 Lower 层。Lower 层代表镜像层，upper 层代表容器可写层



OverlayFS 文件系统练习

极客时间 | 训练营

```
$ mkdir upper lower merged work
$ echo "from lower" > lower/in_lower.txt
$ echo "from upper" > upper/in_upper.txt
$ echo "from lower" > lower/in_both.txt
$ echo "from upper" > upper/in_both.txt
$ sudo mount -t overlay overlay -o
lowerdir=`pwd`/lower,upperdir=`pwd`/upper,workdir=`pwd`/work `pwd`/merged
$ cat merged/in_both.txt
$ delete merged/in_both.txt
$ delete merged/in_lower.txt
$ delete merged/in_upper.txt
```

网络

极客时间 | 训练营

Null(--net=None)

- 把容器放入独立的网络空间但不做任何网络配置；
- 用户需要通过运行 docker network 命令来完成网络配置。

Host

- 使用主机网络名空间，复用主机网络。

Container

- 重用其他容器的网络。

Bridge(--net=bridge)

- 使用 Linux 网桥和 iptables 提供容器互联，Docker 在每台主机上创建一个名叫 docker0 的网桥，通过 veth pair 来连接该主机的每一个 EndPoint。

网络

极客时间 | 训练营

Overlay(libnetwork, libkv)

- 通过网络封包实现。

Remote(work with remote drivers)

- Underlay：
  - 使用现有底层网络，为每一个容器配置可路由的网络 IP。
- Overlay：
  - 通过网络封包实现。

Null 模式

极客时间 | 训练营

- Null 模式是一个空实现；
- 可以通过 Null 模式启动容器并在宿主机上通过命令为容器配置网络。

```
mkdir -p /var/run/netns
find -L /var/run/netns -type l -delete
ln -s /proc/$pid/ns/net /var/run/netns/$pid
ip link add A type veth peer name B
brctl addif br0 A
ip link set A up
ip link set B netns $pid
ip netns exec $pid ip link set dev B name eth0
ip netns exec $pid ip link set eth0 up
ip netns exec $pid ip addr add
$SETIP/$SETMASK dev eth0
ip netns exec $pid ip route add default via
$GATEWAY
```

Docker优势

极客时间 | 训练营

封装性：
- 不需要再启动内核，所以应用扩缩容时可以秒速启动。
- 资源利用率高，直接使用宿主机内核调度资源，性能损失小。
- 方便的 CPU、内存资源调整。
- 能实现秒级快速回滚。

镜像增量分发：
由于采用了 Union FS， 简单来说就是支持将不同的目录挂载到同一个虚拟文件系统下，并实现一种 layer 的概念，每次发布只传输变化的部分，节约带宽。

封装性：
- 一键启动所有依赖服务，测试不用为搭建环境犯愁，PE 也不用为建站复杂担心。
- 镜像一次编译，随处使用。
- 测试、生产环境高度一致（数据除外）。

隔离性：
- 应用的运行环境和宿主机环境无关，完全由镜像控制，一台物理机上部署多种环境的镜像测试。
- 多个应用版本可以并存在机器上。

社区活跃：
Docker 命令简单、易用，社区十分活跃，且周边组件丰富。

作业

极客时间 | 训练营

A.思考题：容器的劣势有哪些？

B.试验题：使用memory进行限制。

THANKS