操作系统安全

Operating system security

[第15次课] 容器技术与安全

授课教师：涂碧波

授课时间：2024年6月7日

内容概要

- 容器与虚拟机
- 容器技术
- 容器安全

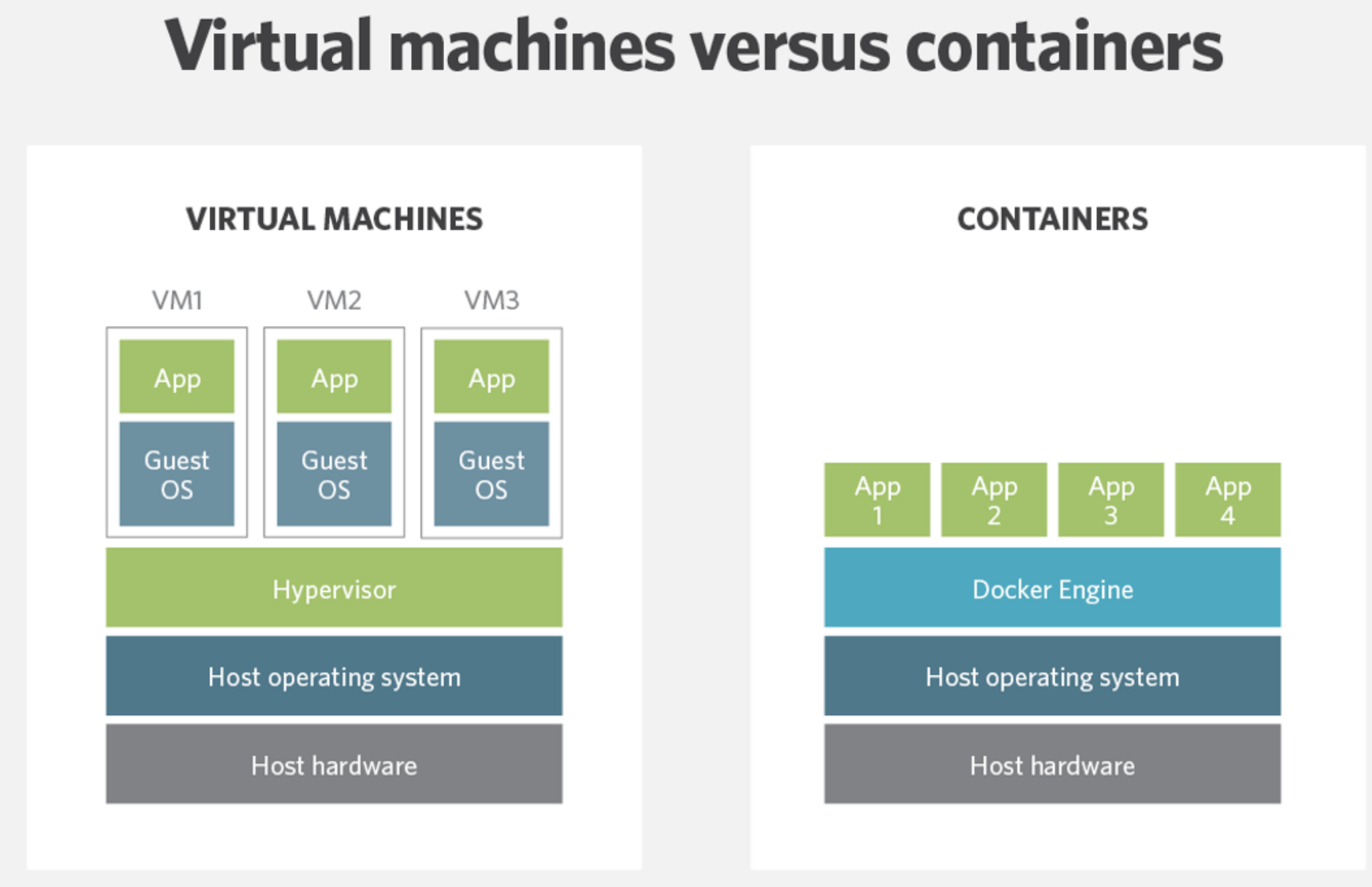
内容概要

- 容器与虚拟机
- 容器技术
- 容器安全

容器概述

- 容器是轻量级的操作系统虚拟化技术，可以让应用运行在一个资源隔离的环境中。
- 应用程序运行所必需的组件打包成一个镜像可以复用。
- 每个容器包含一个独享的完整用户环境空间，并且一个容器内的变动不会影响其它容器的运行环境。
- 容器之间共享同一个系统内核，同一个库文件可被多个容器使用，从而提升资源使用率

容器与虚拟机（1）


Virtual machines versus containers
VIRTUAL MACHINES
VM1
VM2
VM3
App
App
App
Guest OS
Guest OS
Guest OS
Hypervisor
Host operating system
Host hardware
CONTAINERS
App 1
App 2
App 3
App 4
Docker Engine
Host operating system
Host hardware

容器与虚拟机（2）

- 容器与虚拟机拥有着类似的使命：
  - 对应用程序及其关联性进行隔离，从而构建起一套能够随处运行的自容纳单元。
  - 容器与虚拟机还摆脱了对物理硬件的需求，允许我们更为高效地使用计算资源，从而提升能源效率与成本效益。
- 虚拟机会将虚拟硬件、内核以及用户空间打包在新虚拟机当中，虚拟机利用Hypervisor运行在物理设备之上。每台虚拟机都能够获得唯一的操作系统和负载(应用程序)。简言之，虚拟机先需要虚拟一个物理环境，然后构建一个完整的操作系统，再搭建一层Runtime，供应用程序运行。

容器与虚拟机（3）

- 对于容器环境来说：
  - 不需要安装主机操作系统，直接将容器层安装在主机操作系统(通常是Linux变种)之上。
  - 容器可以看成一个装好了一组特定应用的虚拟机，它直接利用了宿主机的内核，抽象层比虚拟机更少，更加轻量化，启动速度极快。
  - 相比虚拟机，实例规模更小，单个物理机能够承载更多的容器，容器拥有更高的资源使用效率。
  - 容器易于迁移，但是只能被迁移到具有兼容操作系统内核的其他服务器当中，这样就会给迁移选择带来限制。
  - 虽然每个容器化应用都会共享相同的操作系统，但每套容器都拥有自己的隔离化用户空间。

容器与虚拟机（4）

- 以 Docker 为代表的容器技术一度被认为是虚拟化技术的替代品，然而这两种技术之间并不是不可调和的，这取决于具体的需求。
  - 如果只是希望将应用运行的实例进行隔离，容器对于管理应用运行环境、启动应用实例以及控制资源开销方面是一个极为高效的工具。容器之间共享同一个系统内核，内存的使用效率会得到提升。
    - Docker这一类的容器，其设计原则就是为了解决这种应用环境的修改以及应用部署的问题，并且这十分符合 DevOps理念
  - 如果寻找最好的环境隔离方案，邻居租户对系统的影响在虚拟化的方案下将不是一个问题。虚拟化层为用户提供一个包含虚拟化后的CPU、内存和IO设备等完整的虚拟机：包括内核在内的一个完整的系统镜像。
    - 尽管现在很多容器都在专注于提高其隔离能力，但是虚拟机的隔离还是要优于容器，并且现在针对虚拟服务器的管理的生态系统也很完善。

容器与虚拟机（5）

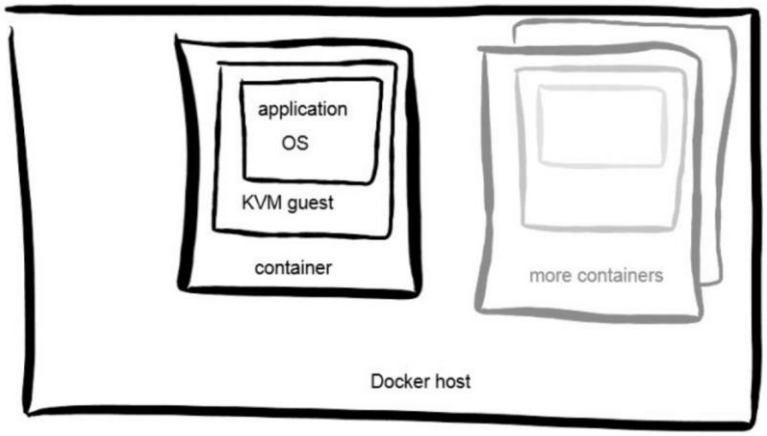
容器	虚拟机
启动应用	启动Guest OS和应用程序
部署的是容器镜像：镜像很小，一般是M级别	部署的是虚拟机镜像：镜像很大，一般是G级别
性能方面，与 VM 相比，容器表现更加出色，并且几乎可以秒启动。	启动操作系统以及初始化托管应用会花费几分钟的时间
容器具有Host OS的root权限	Guest OS运行在非根模式
可用性根据业务本身保证	可用性技术已成熟，如快照、克隆、动态迁移、异地双活
编排平台以k8s为代表，发展迅猛	以CloudStack、OpenStack和vCenter为代表，以在生产环境中使用

容器和虚拟机之间的主要区别在于虚拟化层的位置和操作系统资源的使用方式。

容器与虚拟机混合使用

一个容器中运行一个虚拟机

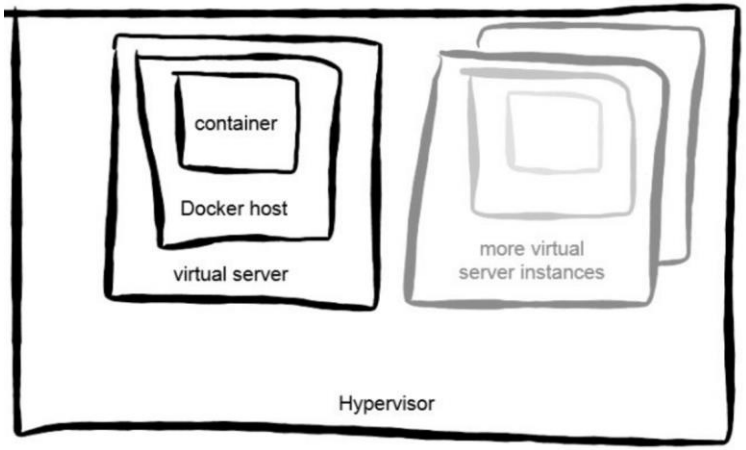
- Docker可以利用KVM镜像在最好的隔离性情况下满足DevOps的需求。
- 需要在启动容器时启动整个操作系统实例，意味着较长的启动时间以及低效的内存使用，只能通过KSM来提升内存利用率。
- 效果和效率虽不理想，但却是一个好的开始。



一个虚拟机中运行一个容器

- 虚拟机并不是由Docker控制，而是由虚拟化
- 一旦系统实例启动，Docker则可以运行容器
而内存的使用率需要KSM来提升。

生。



容器的优点

- 敏捷环境：容器技术最大的优点是创建容器实例比创建虚拟机示例快得多，容器轻量级的脚本可以从性能和大小方面减少开销。
- 提高生产力：容器通过移除跨服务依赖和冲突提高了开发者的生产力。每个容器都可以看作是一个不同的微服务，可以独立升级。
- 安全：容器之间的进程是相互隔离的，其中的基础设施亦是如此。这样其中一个容器的升级或者变化不会影响其他容器。

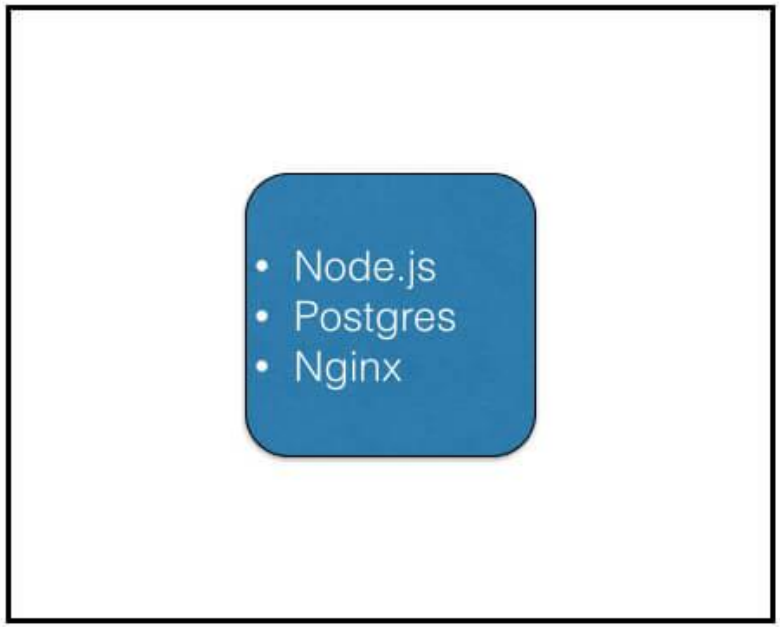
容器的缺点

- **不能适用所有场景**：一些应用程序并不适合运行在容器当中。
- **难以解决依赖关系**：容器运行在物理操作系统之上，相互之间共享底层的操作系统内核、库文件以及二进制文件，这可能会限其可移植性。
- **较差的隔离性**：共享同一个系统内核以及其他组件，在开始运行之前就已经获得了统一的底层授权，攻击可能进入到底层操作系统或者其它容器
- **复杂性增加**：随着容器及应用数量的增加，管理如此之多的容器是一个极具挑战性的任务，可以使用 Kubernetes 和 Mesos 等工具管理。
- **原生 Linux 支持**：大多数容器技术，比如 Docker，基于 Linux 容器，相比于在原生 Linux 中运行容器，在 Microsoft 中运行容器略显笨拙。

容器的分类（1）

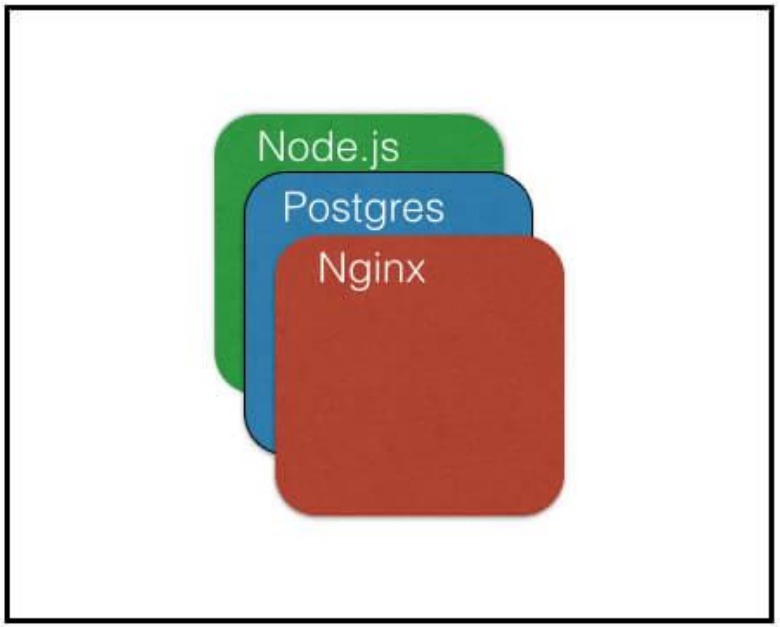
- 如维基百科中所述，“操作系统层虚拟化是一种计算机虚拟化技术，这种技术将操作系统内核虚拟化，可以允许多个独立用户空间的存在，而不是只有一个。这些实例会被称为容器、虚拟引擎、虚拟专用服务器等。从运行在容器中的程序来看，这些实例就如同真正的计算机。”
- 如上所述，容器共享宿主机的内核，但是提供用户空间隔离。在容器中安装、配置以及运行应用程序。相似的是，分配给容器的资源仅对自己可见。就好比是，任何虚拟机不能获取其他虚拟机的资源。
- 当需要配置大量具有相同配置的操作系统时，操作系统容器就会非常有用。因此，容器有助于创建模板，用于创建与另一个操作系统类似风格的容器。
- 要创建操作系统容器，我们可以利用容器技术，如 LXC，OpenVZ，Linux VServer，BSD Jails 和 Solaris 区域。

容器的分类（2）



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones



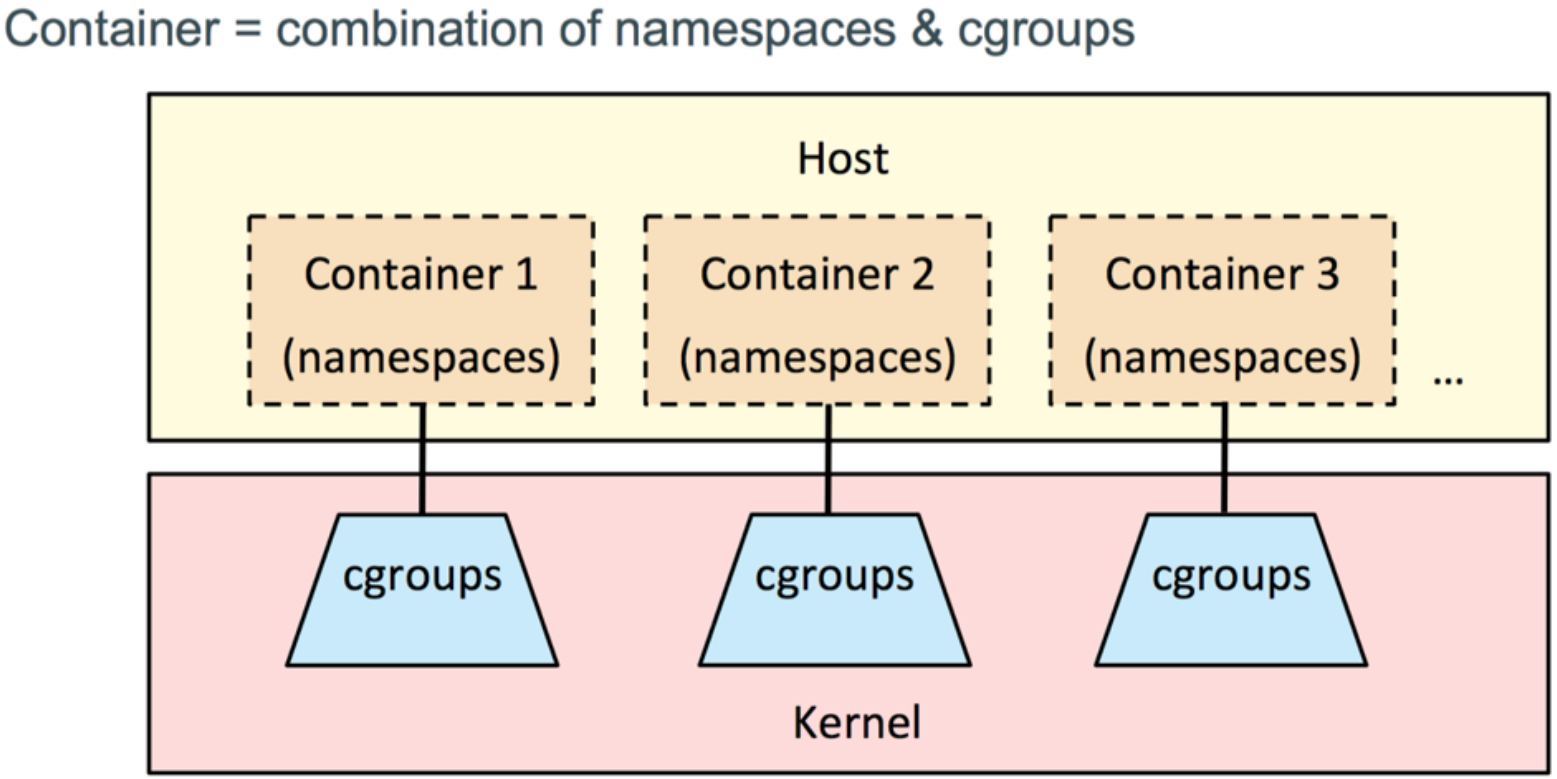
App containers

- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket

内容概要

- 容器与虚拟机
- **容器技术**
- 容器安全

容器框图


Container = combination of namespaces & cgroups
Host
Container 1
(namespaces)
Container 2
(namespaces)
Container 3
(namespaces)
...
cgroups
cgroups
cgroups
Kernel

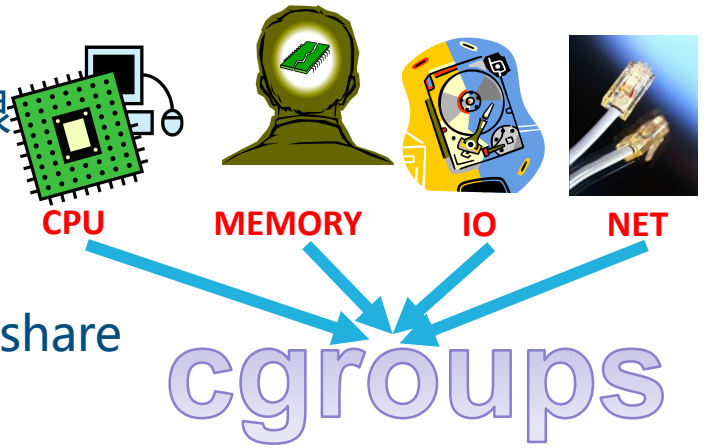
Cgroups

Cgroups

- **Control groups（cgroups）：**是Linux内核提供的一种可以限制、记录、隔离进程组（process groups）所使用物理资源（如cpu, memory, IO等）的机制。
  - Cgroups最早是由Google的工程师在2006年发起，其最早名称为进程容器
  - Cgroups目标是为资源管理提供一个统一的框架，既整合现有的cpuset等子系统，也为未来开发新的子系统提供接口
  - 适用于多种应用场景，从单个进程的资源控制，到操作系统层次的虚拟化
  - Cgroups是容器实现虚拟化所使用的资源管理手段，没有cgroups就没有容器

Cgroups提供功能

- 限制进程组可以使用的资源数量
  - Memory子系统可以为进程组设定内存使用上限
- 进程组的优先级控制
  - 可以使用cpu子系统为某个进程组分配特定cpu share
- 记录进程组使用的资源数量
  - 可以使用cpuacct子系统记录某个进程组使用的cpu时间
- 进程组隔离
  - 使用ns子系统使不同的进程组使用不同的namespace，以达到隔离的目的
- 进程组控制
  - 使用freezer子系统可以将进程组挂起和恢复



Cgroups中的概念

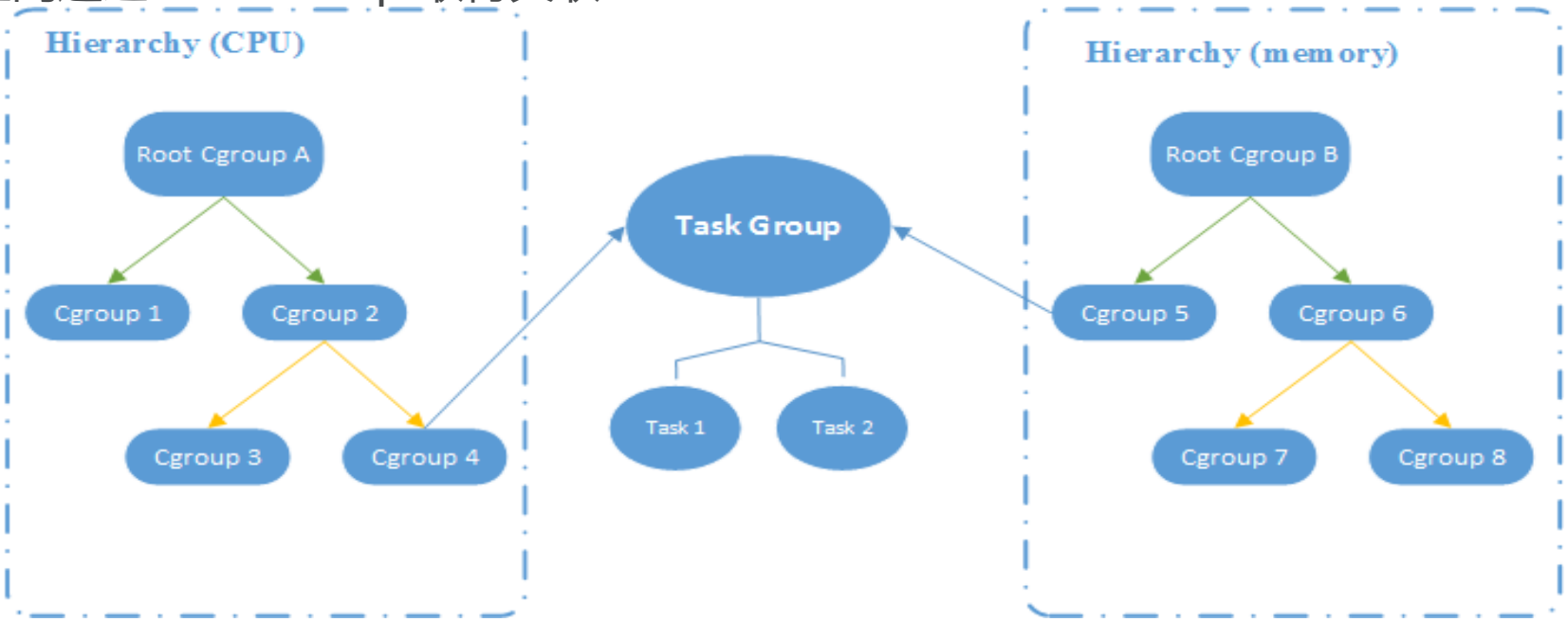
- 任务（task）
  - 在cgroups中，任务就是系统的一个进程。
- 控制族群（control group，cgroup）
  - 控制族群就是一组按照某种标准划分的进程。是Cgroups中的资源分配、控制和限制的单位。
  - 进程可以加入到某个控制族群，也从一个进程组迁移到另一个控制族群。
- 层级（hierarchy）
  - 控制族群可以组织成hierarchical的形式，既一颗控制族群树。
  - 控制族群树上的子节点是父节点的孩子，继承父控制族群的特定属性
- 子系统（subsytem）
  - 一个子系统是一个资源控制器：如cpu子系统是控制cpu时间分配的控制器
  - 一个子系统必须附加（attach）到一个层级上才能起作用
  - 被附加子系统的层级，其上的所有控制族群都受到这个子系统的控制

Cgroups简介

- 每次在系统中构建新层级时，系统中的所有任务都是那个层级的的默认cgroup (root cgroup，即树根) 的初始化成员
  - cgroup在创建层级时自动创建，后续在该层创建的cgroup都是root cgroup的子树
  - 一个子系统最多只能附加到一个层级
  - 一个层级可以附加多个子系统
  - 一个任务可以是多个cgroup的成员，但这些cgroup须在不同的层级
  - 系统中的进程在创建子进程时，该子进程自动成为其父进程所在的cgroup成员

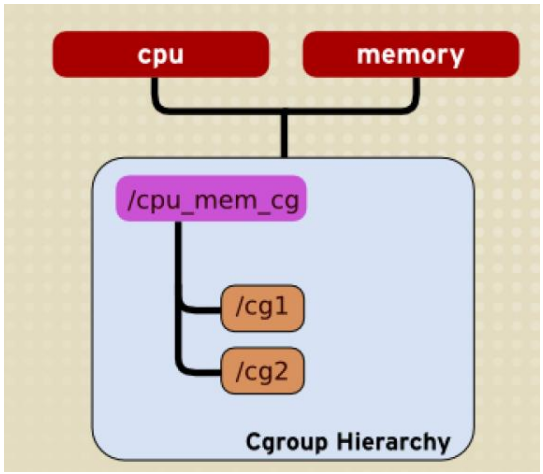
Cgroups简介

- 每次在系统中构建新层级时，系统中的所有任务都是那个层级的的默认cgroup (root cgroup，即树根) 的初始化成员
  - cgroup在创建层级时自动创建，后续在该层创建的cgroup都是root cgroup的子树
  - 一个子系统最多只能附加到一个层级
  - 一个层级可以附加多个子系统
  - 一个任务可以是多个cgroup的成员，但这些cgroup须在不同的层级
  - 系统中的进程在创建子进程时，该子进程自动成为其父进程所在的cgroup成员
- 如图所示cgroup层级关系，CPU 和 Memory 两个子系统有自己独立的层级系统，之间通过 Task Group 取得关联

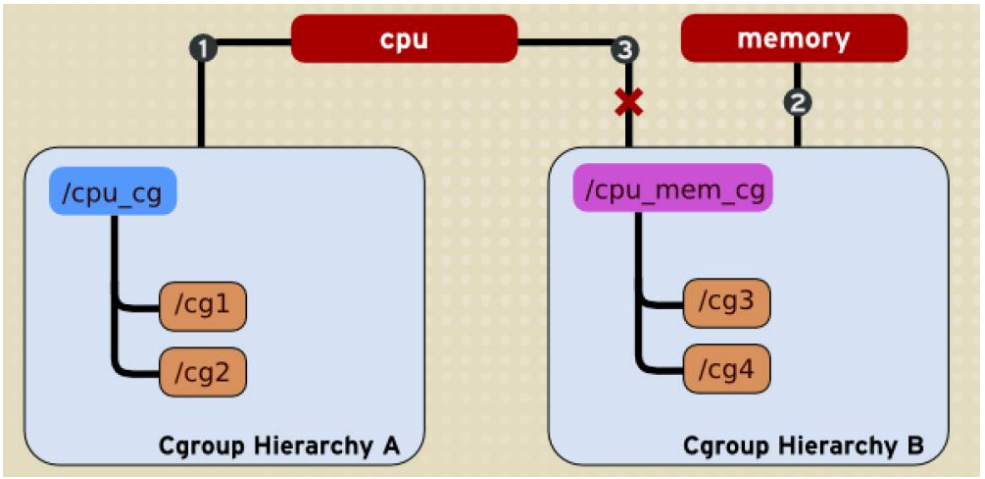


层级、子系统、控制群组和任务之间的关系

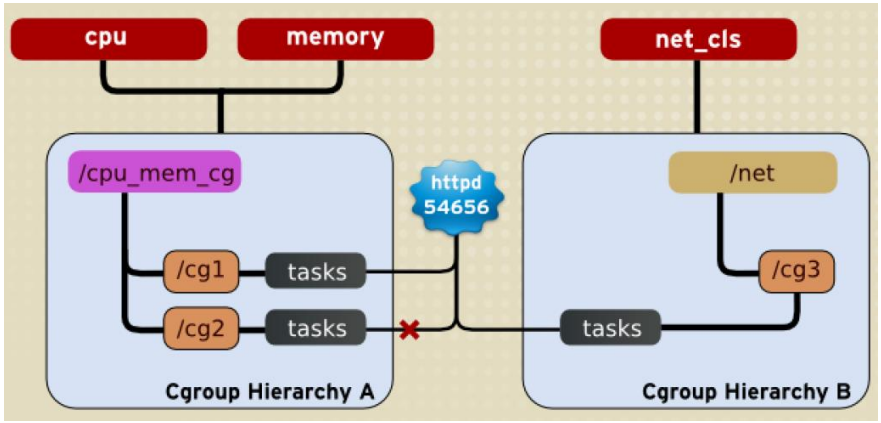
一个层级可以附加多个子系统



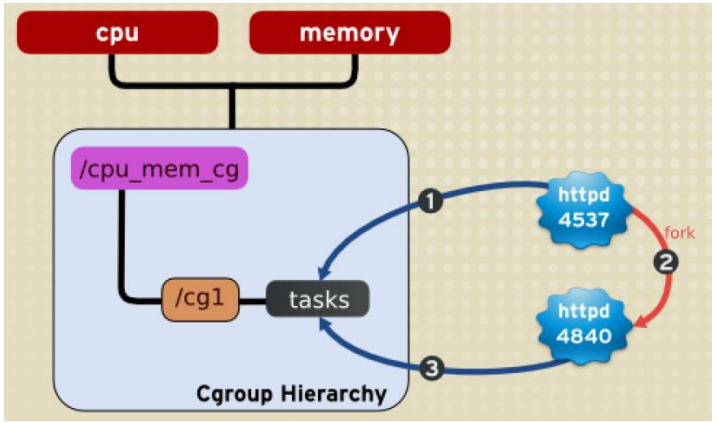
一个子系统只能附加到一个层级



一个任务不能属于同一层级的不同cgroup



子任务继承父任务cgroup信息



Cgroups子系统（1）

- cpu 子系统
  - 使用调度程序提供对cgroup中任务CPU使用率的限制，即控制cgroup中所有进程可以使用的 cpu时间片
- cpuset 子系统
  - 为 cgroups 中的任务分配单独的 cpu 核心和内存节点
- cpuacct 子系统
  - 自动统计 cgroup中任务的 cpu 使用报告
- memory 子系统
  - 设定 cgroup 中任务的内存使用限制，并自动生成任务所使用的内存资源报告

Cgroups子系统（2）

- blkio子系统
  - 为块设备设定输入/输出限制，如，物理设备（磁盘，固态硬盘，USB等），控制并监控 cgroup 中的任务对块设备的 I/O 访问
- devices 子系统
  - 控制任务能够访问的设备资源，允许或者拒绝cgroup中的进程访问设备。
- freezer 子系统
  - 挂起或者恢复 cgroup中的进程
- ns 子系统
  - 使 cgroup中的任务使用不同的 namespace
- net_cls 子系统
  - 标记 cgroup中进程的网络数据包，通过 traffic control对数据包进行控制

CPU子系统（1）

- Cgroup通过cpu.shares的控制文件分配CPU时间片
  - 对其写入整数值可以控制该 cgroup获得的时间片，通过Linux CFS调度器实现
  - CFS在真实的硬件上模拟了完全理想的多任务处理器“。在“完全理想的多任务处理器”下，每个进程（进程组）能同时获得CPU的执行时间。
    - 红黑树的键值是进程所谓的虚拟运行时间
    - CFS调度器都会选择红黑树的最左边的叶子节点作为下一个将获得cpu的进程
    - 每次tick中断，CFS调度器更新进程的虚拟运行时间，然后调整当前进程在红黑树中的位置
  - CFS对tick中断的处理在task_tick_fair中进行
    - 在组调度的情况下：从当前进程的se开始，沿着task_group树从下到上对se调用entity_tick，即更新各个 se的虚拟运行时间
    - 在非组调度的情况下：只会对当前进程se做处理。 处理完tick中断后，若有必要则进行调度
    - CFS调度是通过pick_next_task_fair 函数选择下一个运行的进程的

```
for_each_sched_entity(se) {
    cfs_rq = cfs_rq_of(se);
    entity_tick(cfs_rq, se, queued);
}
```

```
#define for_each_sched_entity(se) \
    for (; se; se = se->parent)
#define for_each_sched_entity(se) \
    for (; se; se = NULL)
```

CPU子系统（2）

- CFS定义了task_group管理调度

```
struct task_group {
    struct cgroup_subsys_state css;  //进程或cgroup获取它对应的task_group
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct sched_entity **se;      //指向该task_group在每个cpu的调度实体
    struct cfs_rq **cfs_rq;     //指向该task_group在每个cpu上所拥有的一个可调度的进程队列
    unsigned long shares;
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity **rt_se;
    struct rt_rq **rt_rq;
    struct rt_bandwidth rt_bandwidth;
#endif
    struct rcu_head rcu;
    struct list_head list;
    struct task_group *parent;
    struct list_head siblings;
    struct list_head children;
};
```

将task_group 连成一颗树

CPU子系统（3）

- cpu子系统本身结构体cpu_cgroup_subsys是对抽象cgroup_subsys的实现

```
struct cgroup_subsys cpu_cgroup_subsys = {
    .name    = "cpu",
    .create  = cpu_cgroup_create,
    .destroy= cpu_cgroup_destroy,
    .can_attach     = cpu_cgroup_can_attach,
    .attach  = cpu_cgroup_attach,
    .populate       = cpu_cgroup_populate,
    .subsys_id = cpu_cgroup_subsys_id,
    .early_init     = 1,
};
```

- Shares文件对应的cftype结构
  - 当对cgroup目录下文件进行操作时，该结构体中定义的函数指针指向的函数就会被调用

```
#ifdef CONFIG_FAIR_GROUP_SCHED
    static struct cftype files[] ={
    .name = "shares",
    .read_u64 = cpu_shares_read_u64,
    .write_u64 = cpu_shares_write_u64, },
#endif
```

//读取task_group中存储的shares就行了

//shares值最终就是通过调度实体的load 值来起作用

Cpuset子系统（1）

- Cpuset 子系统定义了一个叫 cpuset 的数据结构来管理 cgroup 中的任务能够使用的 cpu 和内存节点
  - Cpuset 子系统的实现是通过在内核代码加入一些 hook 代码

```
struct cpuset {
    struct cgroup_subsys_state css;     //用于task或cgroup获取cpuset结构

    unsigned long flags;
    cpumask_var_t cpus_allowed;         //定义了该cpuset包含的cpu节点
    nodemask_t mems_allowed;            //定义了该cpuset包含的内存节点
    struct cpuset *parent;              //用于维持 cpuset 的树状结构

    struct fmeter fmeter;    /* memory_pressure filter */
  /* partition number for rebuild_sched_domains() */
    int pn;                             //指定了 cpuset 的调度域的分区号
    int relax_domain_level;             //表示进行 cpu 负载均衡寻找空闲 cpu 的策略
    struct list_head stack_list;        //用于遍历 cpuset 的层次结构
};
```

  - 进程 task_struct 结构体也有一个用以存储进程的 cpus_allowed 信息的cpus_allowed 成员;一个用于存储进 程的 mems_allowed 信息的mems_allowed 成员，。

Cpuset子系统（2）

- 在内核初始化代码（ start_kernel 函数）中插入了cpuset 的初始化代码 cpuset_init
  - 将 top_cpuset 能使用的 cpu 和内存节点设置成所有 节点
  - 注册 cpuset 文件系统，这个是为了兼容性，在具体实现时，对 cpuset 文件系统的操作都被重定向了 cgroup 文件系统
- cpuset 子系统还在 do_basic_setup 函数（在 kernel_init 中被调用）中插入了对 cpuset_init_smp 的调用代码，用于 smp 相关的初始化工作。
  - 将 top_cpuset 的 cpu 和 memory 节点设置成所有 online 的节点，之前初始化时因不知哪些 online 节点只是简单设成所有，在 smp 初始化后将其设成所有 online 节点 。
  - 加入了两个 hook 函数， cpuset_track_online_cpus 和 cpuset_track_online_nodes，这个两个函数将在 cpu 和 memory 热插拔时被调用。
- 那cpuset如何对进程的调度起作用？这个跟task_struct中cpu_allowed字段有关
  - Task_struct的cpu_allowed和进程所属的cpuset的cpus_allowed保持一致
  - 在进程被fork出来的时候，进程继承了父进程的cpuset和cpus_allowed字段
  - fork后会被调用wake_up_new_task（除非指定CLONE_STOPPED标记），为新进程选择 cpu。选到 cpu 后，会与cpu_allowed比对，从而保证了新进程只能在 cpu_allowed 中的 cpu 上运行

Cpuset子系统（3）

- Linux中分配物理页框的函数有6个，get_free_pages，get_free_page, get_zeroed_page, alloc_pages, alloc_page, get_dma_pages
  - 以上函数最终都调用 alloc_pages→ alloc_pages_nodemask→ get_page_from_freelist→ cpuset_zone_allowed_softwall。从zone list中分配page时，判断当前节点是否属于mems_allowed。
- cpuset子系统最重要的两个控制文件
  - cpus文件指定进程可以使用的cpu节点，mems文件指定进程可以使用的memory节点。
  - 这两个文件的读写都是通过cpuset_common_file_read和cpuset_write_resmask实现的，通过private属性区分。cpuset_write_resmask根据文件类型分别调用 update_cpumask 和 update_nodemask更新cpu或memory的节点信息

```
static struct cftype files[] ={
    .name = "cpus",
    .read = cpuset_common_file_read,
    .write_string = cpuset_write_resmask,
    .max_write_len = (100U + 6 * NR_CPUS),
    .private = FILE_CPULIST,
},
```

```
static struct cftype files[] ={
    .name = "mems",
    .read = cpuset_common_file_read,
    .write_string = cpuset_write_resmask,
    .max_write_len = (100U + 6 * MAX_NUMNODES),
    .private = FILE_MEMLIST,
},
```

Memory子系统（1）

- memory子系统是通过linux的resource counter机制实现的
  - resource counter是内核为子系统提供的一种资源管理机制，用于记录资源的数据结构和相关函数

```
struct res_counter {
    unsigned long long usage;      //用于记录当前已使用的资源
    unsigned long long max_usage;  //用于记录使用过的最大资源量
    unsigned long long limit;      //用于设 置资源的使用上限
    unsigned long long soft_limit; //进程组使用的资源可以超过这个限制
    unsigned long long failcnt;/*  //用于记录资源分配失败的次数
     spinlock_t lock;
    struct res_counter *parent;    //指向父节点，这个变量用于处理层次性的资源管理
};
```

  - **void res_counter_init**用于初始化一个 res_counter
  - **int res_counter_charge**当资源被分配时，资源要被记录到res_counter。
    - 从当前res_counter开始，从下往上逐层增加资源的使用量。如果超过使用上限就增加失败次数；否则增加使用量的值。如果这个值已超过历史最大值，则更新最大值
  - **void res_counter_uncharge**当资源被归还到系统的时候，要在相应的res_counter减轻相应的使用量。此过程与**int res_counter_charge类似。**

Memory子系统（2）

- memory子系统定义了mem_cgroup结构体来管理cgroup相关的内存使用

```
struct mem_cgroup {
   struct cgroup_subsys_state css;      //便于task或 cgroup获取mem_cgroup
   struct res_counter res;              //用于管理memory资源
   struct res_counter memsw;            //用于管理memory+swap资源
   struct mem_cgroup_lru_info info;
   spinlock_t reclaim_param_lock;
   int prev_priority;
   int last_scanned_child;
   bool use_hierarchy;
   atomic_t oom_lock;      //用来标记资源控制和记录时是否是层次性的
   atomic_t refcnt;
   unsigned int swappiness;
   int oom_kill_disable;          //是否使用oom-killer
   bool memsw_is_minimum;      //如果memsw_is_minimum为true，则res.limit=memsw.limit
   struct mutex thresholds_lock;
   struct mem_cgroup_thresholds thresholds;
   struct mem_cgroup_thresholds memsw_thresholds;
   struct list_head oom_notify;
   unsigned long  move_charge_at_immigrate;
   struct mem_cgroup_stat_cpu *stat;
};
```

Memory子系统（3）

- memory子系统还定义了page_cgroup结构体，可看作是mem_map的扩展

```
struct page_cgroup {
   unsigned long flags;
   struct mem_cgroup *mem_cgroup;
   struct page *page;
   struct list_head lru;    /* per cgroup_LRU list */
};
```

- 每个page_cgroup和一个page关联，其mem_cgroup成员将page与特定mem_cgroup关联。
  - 每个进程有一个 mm_struct用于管理其内存，知道其所属进程，进而知道其所属mem_cgroup
  - 每个page通过page_cgroup知道其所属mem_cgroup

Memory子系统（4）

- 内存资源的管理的实现
  - page fault发生时，有两种情况需要分配新的页框：进程请求调页和COW
    - ① 子系统则在 do_fault、do_anonymous_page、do_wp_page植入 mem_cgroup_newpage_charge 来进行charge操作
    - ② memory子系统在do_swap_page 加入 mem_cgroup_try_charge_swapin处理页面换入时的charge
    - ③ 当页加入page cache时，mem_cgroup_cache_charge被植入add_to_page_cache_locked处理charge
    - ④ mem_cgroup_prepare_migration是用于处理内存迁移中的charge操作

    Charge函数最终调用mem_cgroup_try_charge →→ res_counter_charge处理memory和memory+swap
  - 除了charge操作，memory子系统还需要处理相应的uncharge操作
    - ① mem_cgroup_uncharge_page用于当匿名页完全unmaped
    - ② mem_cgroup_uncharge_cache_page用于page cache从radix-tree删除
    - ③ mem_cgroup_uncharge_swapcache用于swap cache从radix-tree删除
    - ④ mem_cgroup_uncharge_swap用于swap_entry的引用数减到0
    - ⑤ mem_cgroup_end_migration用于内存迁移结束时相关的uncharge操作

    Uncharge函数最终调用do_uncharge→→res_counter_uncharge对memory和memory+swap uncharge

Memory子系统（5）

- memory子系统也实现了一个cgroup_subsys

```
struct cgroup_subsys mem_cgroup_subsys = {
    .name = "memory",
    .subsys_id = mem_cgroup_subsys_id,
    .create = mem_cgroup_create,
    .pre_destroy = mem_cgroup_pre_destroy,
    .destroy = mem_cgroup_destroy,
    .populate = mem_cgroup_populate,
    .can_attach = mem_cgroup_can_attach,
    .cancel_attach =
    mem_cgroup_cancel_attach,
    .attach = mem_cgroup_move_task,
    .early_init = 0,
    .use_id = 1,
};
```

- memory子系统有两个配置文件memsw.limit_in_bytes

```
static struct cftype files[] ={
  .name = "memsw.limit_in_bytes",
  .private = MEMFILE_PRIVATE(_MEMSWAP, RES_LIMIT),
  .write_string = mem_cgroup_write,
  .read_u64 = mem_cgroup_read,
},
```

这个文件用于设定**memory+swap**上限值

```
static struct cftype files[] ={
  .name = "limit_in_bytes",
  .private = MEMFILE_PRIVATE(_MEM, RES_LIMIT),
  .write_string = mem_cgroup_write,
  .read_u64 = mem_cgroup_read,
},
```

这个文件用于设定**memory**上限值

Blkio子系统（1）

- blkio 子系统定义 blkio_cgroup结构来存储每个 cgroup 的 block IO 信息

```
struct blkio_cgroup {
   struct cgroup_subsys_state css;
   unsigned int weight; //存储的是blkio.weight的值，用于控制此cgroup中进程可占用的IO时间
   spinlock_t lock;
   struct hlist_head blkg_list;
   struct list_head policy_list; /* list of
   blkio_policy_node */
};
```

- blkio通过 CFQ IO调度器实现。IO 调度器的作用是处理进程的 IO 请求，然后将其交由相应的块设备处理。IO 调度器一般采用所谓的电梯算法
  - CFQ 现在是 Linux 内核默认的 IO 调度算法（电梯算法）
  - blkio通过 CFQ 组调度实现，这点跟 cpu 子系统是通过 CFS 组调度实现类似。
  - 每个进程有自己的 IO 请求队列，各个进程的队列之间则按时间片轮转来处 理，以保证每个进程都能公平的获得 IO 带宽。
  - CFQ 很复杂，代码很多

Blkio子系统（2）

- 在控制文件中写值可限制访问或带宽，从中读值可提供 I/O 操作信息Blkio子系统重要的控制文件是weight。
  - 向blkio.weight写入值（100到1000 ）可指定此cgroup块 I/O访问的相对比例（加权）
  - 这个跟 cpu 子系统里面的 cpu.shares 文件类似，控制进程占有的 IO 时间

```
static struct cftype files[] ={
    .name = "weight",
    .read_u64 = blkiocg_weight_read,
    .write_u64 = blkiocg_weight_write,
}
```

  - blkiocg_weight_read 和 blkiocg_weight_write 分别实现对 weight 字段读写操作。

Freezer子系统（1）

- freezer 子系统有一个控制文件：freezer.state，
  - 将 FROZEN 写入该文件，可以将 cgroup 中的进程挂起，
  - 将 THAWED 写入该文件，可以将已挂起的进程恢复。
  - FREEZING表示该 cgroup 中有些进程现在不能被 frozen。当这些不能被 frozen 的进程从该 cgroup 中消失的时候，FREEZING 会变成 FROZEN

```
struct freezer {
    struct cgroup_subsys_state css;
    enum freezer_state state;     //存储cgroup当前的状态
    spinlock_t lock; /* protects _writes_ to state */
};
```

  - 从freezer.state文件读取是 freezer_read 实现的，从 freezer 结构体从读出状态。
    - 如果是 FREEZING 状态，则需要更新状态：对该 cgroup 所有的进程迭代了一遍，分别统计进程数和已经 frozen 的进程数，根据统计结果改变状态。

```
static struct cftype files[] = {
    {.name = "state", .read_seq_string = freezer_read, .write_string =
    freezer_write, },
};
```

Freezer子系统（2）

- freezer.state写入由 freezer_write 来处理，从写入值获取目标状态，根据 goal_state 分别调用不同的实现函数：
  - 首先将当前状态设成 *CGROUP_FREEZING*，然后对 cgroup 中的进程进行迭代，迭代中对进程进行 freeze 操作，如果不成功则进行进一步的判断：如果是进程已经 frozen 了，那也直接进行下一次迭代，如果不是，则进行计数。 根据统计结果，如果所有进程都 frozen，则返回 0，否则返回-EBUSY 即有进程未被 frozen。

Freezer子系统（3）

- freezer 子系统结构体的定义：

```
struct cgroup_subsys freezer_subsys = {
    .name    = "freezer",
    .create  = freezer_create,
    .destroy= freezer_destroy,
    .populate       = freezer_populate,
    .subsys_id = freezer_subsys_id,
    .can_attach     = freezer_can_attach,
    .attach  = NULL,
    .fork    = freezer_fork,
    .exit    = NULL,
};
```

- can_attach 是在一个进程加入到一个 cgroup 之前调用的， 检查是否可以attach。
  - freezer_can_attach 中对 cgroup 当前的状态做了检查，如果是 frozen 就返回错误，即不能将一个进程加入到一个 已经frozen 的 cgroup中。

Devices子系统（1）

- devices子系统有 三个控制文件：
  - devices.allow用于指定cgroup中的进程可以访问的设备
    - 格式 type、major、minor 和 access
    - type指定设备类型；major和minor指定设备的主次设备号； access 则指定相应的权限（RWM），其中M指允许进程生成还不存在的设备文件
  - devices.deny用于指定cgroup中的进程不能访问的 设备
  - devices.list用于报告cgroup中的进程访问的设备

```
struct dev_cgroup {
   struct cgroup_subsys_state css;
   struct list_head whitelist;          //用来管理资源
};
```

- cgroup_subsystem_state 的结构来管理资源
- whitelist指向了该 cgroup 中的进程可以访问的 devices whilelist

Devices子系统（2）

- devices 子系统定义 了一个叫 dev_whitelist_item 的结构来管理可以访问的 device

```
struct dev_whitelist_item {
    u32 major, minor;
    short type;
    short access;
    struct list_head list; //用于将该结构体连到相应的dev_cgroup中whitelist指向的链表
    struct rcu_head rcu;
};
```

  - 对应于 devices.allow 中的一个条目
- 如此，devices 子系统则可管理 cgroup中进程可以访问的 devices
  - 一个进程能否访问对应的 devices
    - int devcgroup_inode_permission(struct inode *inode, int mask)
  - 一个进程要访问一个设备 就必须通过 mknod 建立相应的设备文件。
    - 提供了int devcgroup_inode_mknod(int mode, dev_t dev)

Devices子系统（3）

- devices 同样实现了一个 cgroup_subsys

```
struct cgroup_subsys devices_subsys = {
    .name = "devices",
    .can_attach = devcgroup_can_attach,
    .create = devcgroup_create,
    .destroy = devcgroup_destroy,
    .populate = devcgroup_populate,
    .subsys_id = devices_subsys_id,
};
```

- devices 相应的三个控制文件
  - devcgroup_access_write 通过调用 devcgroup_update_access 根据文件类型：allow则将 item 加入 whitelist；deny则将 item 从 whitelist 中删去

```
static struct cftype dev_cgroup_files[] = {
    {.name = "allow", .write_string   = devcgroup_access_write, .private = DEVCG_ALLOW,},
    {.name = "deny", .write_string = devcgroup_access_write, .private = DEVCG_DENY,},
    {.name = "list", .read_seq_string = devcgroup_seq_read, .private = DEVCG_LIST,},
};
```

Ns子系统

- ns 子系统是一个比较特殊的子系统：
  - ns 子系统没有自己的控制文件
  - ns 子系统没有属于自己的状态信息

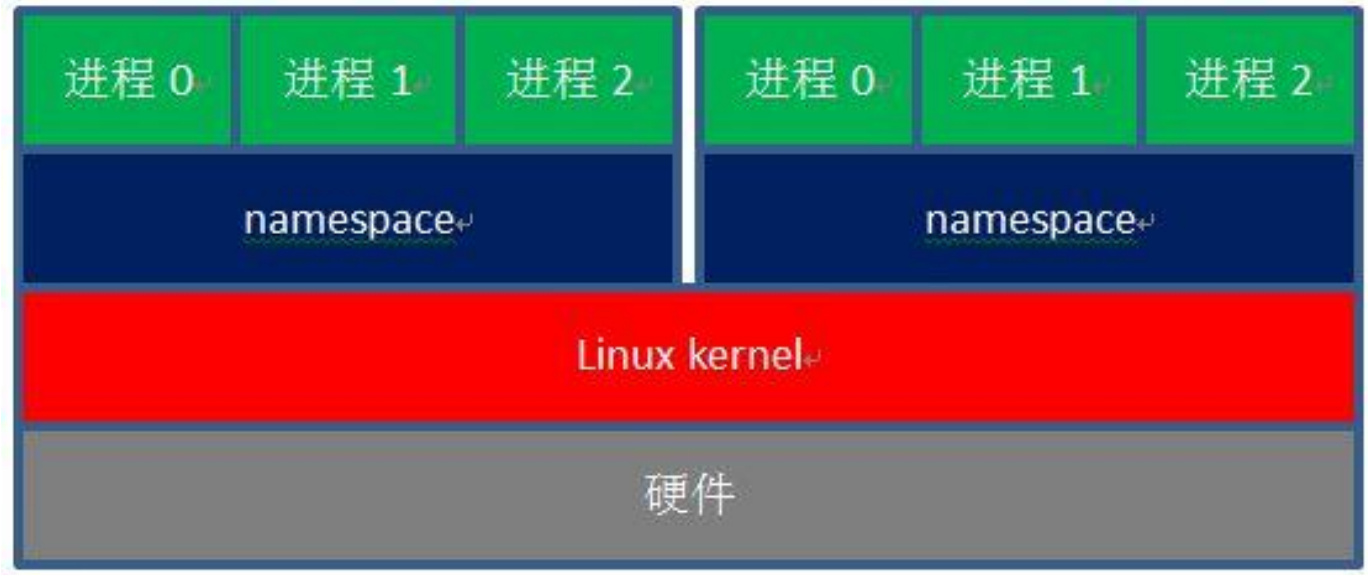
```
struct ns_cgroup {
   struct cgroup_subsys_state css;
};
```

- ns子系统实现比较简单
  - 只提供ns_cgroup_clone, 在unshare_nsproxy_namespaces和copy_process中被调用。
  - 只是在当前的 cgroup 下创建了一个子 cgroup，该子 cgroup 完全 clone 了当前 cgroup 的信息，然后将当前的进程移到新建立的 cgroup 中
  - 当前进程（即父进程）和子进程的命名空间不同时，调用 ns_cgroup_clone。实际上是提供了一种同命名空间的进程聚类机制。具有相同命名空间的进程会在相同 cgroup 中。

Namespace

Namespace介绍（1）

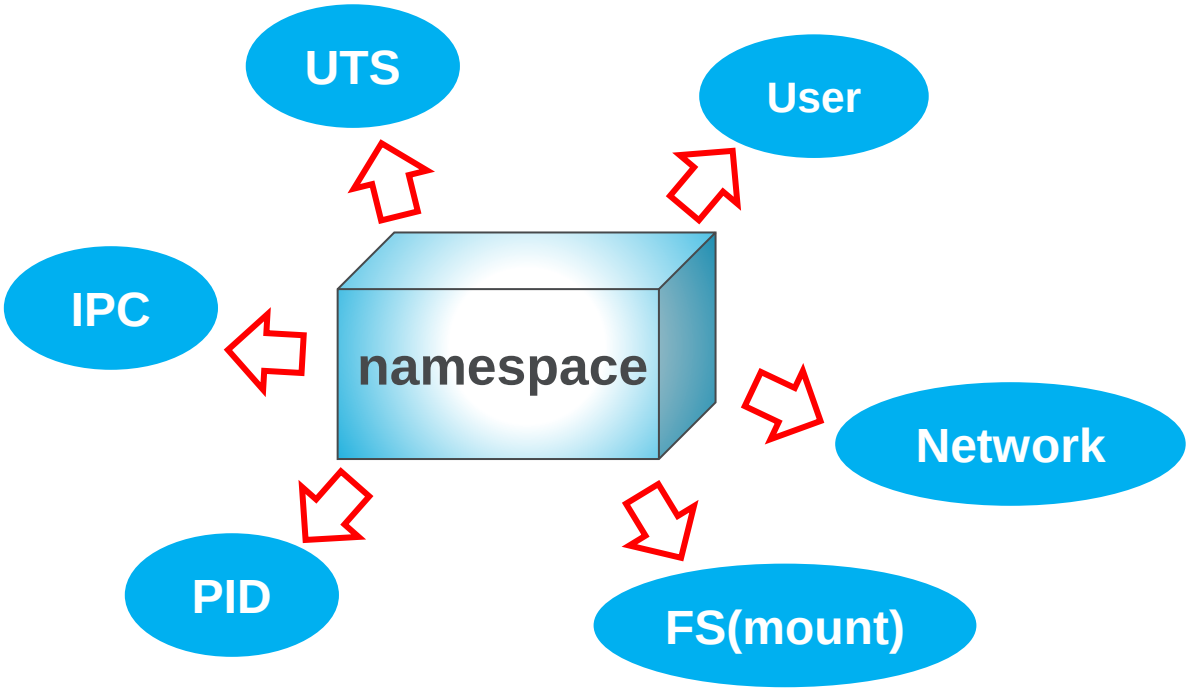
- namespace的主要目的是为了实现隔离环境的独立视图，同一个namespace下的进程可以感知彼此的变化，而对外界的进程一无所知。
  - 这样就可以让容器中的进程产生错觉，仿佛自己置身于一个独立的系统环境中，以此达到独立和隔离的目的。



Namespace介绍（2）

- Linux Namespace 是Linux提供的一种内核级别的环境隔离方法，在容器中用于进程组隔离。
  - Namespace是Linux chroot的扩展，提供了对UTS、IPC、Mount、PID、Network、User等的隔离。
    - Chroot可使得内部的文件系统无法访问外部的内容
  - 不同的进程组使用不同的namespace
    - 不同的进程组有各自的主机名，网络，文件系统挂载空间
  - 每个namespace里面的资源对其它namespace透明
  - namespace展现了系统的不同视图，并可以组织为层次结构
  - 创建namespace时，只需要指定相应的flag即可

Namespace介绍（3）



Namespace介绍（4）

- UTS namespace
  - 主机命名空间，独立出主机名和网络信息服务
- FS (mount) namespace
  - 挂载命名空间，进程运行时可以将挂载点与系统分离，使用这个功能时，可以达到 chroot 的功能，而在安全性方面比 chroot 更高
- PID namespace
  - 进程命名空间，空间内的PID 是独立分配，命名空间内的 PID 映射到命名空间外时会使用另外一个 PID
- IPC namespace
  - 进程间通信(IPC)的命名空间，可以将 SystemV 的 IPC 和 POSIX 的消息队列独立出来

Namespace介绍（5）

- Network namespace
  - 网络命名空间，用于隔离网络资源（/proc/net、IP 地址、网卡、路由等）。后台进程可以运行在不同命名空间内的相同端口上
- User namespace
  - 用户命名空间，用户 ID 和组 ID 在命名空间内外是不一样，在不同命名空间内可以存在相同的 ID
  - 将本地的虚拟user-id映射到真实的user-id

分类	Mount namespaces	UTS Namespaces	IPC Namespaces	PID Namespaces	Network Namespaces	User namespaces
系统调用参数	CLONE_NEWNS	CLONE_NEWUTS	CLONE_NEWIPC	CLONE_NEWPID	CLONE_NEWNET	CLONE_NEWUSER
内核版本	Linux 2.4.19	Linux 2.6.19	Linux 2.6.19	Linux 2.6.24	Linux 2.6.29	Linux 3.8
隔离内容	挂载点（文件系统）	主机名与域名	信号量、消息队列和共享内存	进程编号	网络设备、网络栈、端口等	用户和用户组

Namespace—UTS隔离

```
int container_main(void* arg)
{
    printf("Container - inside the container!\n");
    sethostname("container",10); /* 设置hostname */
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}
```

```
int main()
{
    printf("Parent - start a container!\n");
    int container_pid = clone(container_main,
container_stack+STACK_SIZE,
            CLONE_NEWUTS | SIGCHLD, NULL); /*启用
CLONE_NEWUTS Namespace隔离 */
    waitpid(container_pid, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

```
zm@ubuntu:~$ sudo ./uts
Parent - start a container!
Container - inside the container!
root@container:~# hostname
container
root@container:~# uname -n
container
```

Docker中，每个镜像基本都以自己所提供的服务命名了自己的hostname而没有对宿主机产生任何影响。

Namespace—IPC隔离（1）

- 容器中进程间通信采用的方法包括：信号量、消息队列和共享内存。容器内部进程间通信对宿主机来说，实际上是具有相同namespace中的进程间通信，因此需要一个唯一的标识符来进行区别。
  - ✓ 申请IPC资源就申请了这样一个全局唯一的32位ID，所以IPC namespace中实际上包含了系统IPC标识符以及实现POSIX消息队列的文件系统。

创建一个IPC的Queue

```
zm@ubuntu:~$ ipcmk -Q
Message queue id: 0
```

查看IPC Queue:

```
zm@ubuntu:~$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xd0d56eb2 0          zm         644        0            0
```

Namespace—IPC隔离（2）

运行没有CLONE_NEWIPC的程序

```
zm@ubuntu:~$ sudo ./uts
Parent - start a container!
Container - inside the container!

root@container:~# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xd0d56eb2 0          zm         644        0            0
```

运行CLONE_NEWIPC的程序

```
root@ubuntu:~$ sudo ./ipc
Parent - start a container!
Container - inside the container!

root@container:~# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

```
int container_pid = clone(container_main,
container_stack+STACK_SIZE, CLONE_NEWUTS |
CLONE_NEWIPC | SIGCHLD, NULL);
```

Docker本身通过socket或tcp进行通信。

Namespace—PID隔离（3）

- PID namespace隔离非常实用，它对进程PID重新标号，即两个不同namespace下的进程可以有同一个PID。每个PID namespace都有自己的计数程序。
  - 内核为所有的PID namespace维护了一个树状结构，最顶层的是系统初始时创建的，称之为root namespace。新创建的PID namespace称之为child namespace。
  - 所属的父节点可以看到子节点中的进程，并可以通过信号等方式对子节点中的进程产生影响。反之，则不可。
    - 每个PID namespace中第一个进程是“PID 1“，都会像传统Linux中的init进程一样拥有特权，起特殊作用。
    - 在PID namespace中，通过kill或ptrace不会影响父节点或兄弟节点的进程
    - 如果在新的PID namespace中重新挂载了/proc文件系统，只会显示同属一PID namespace中的其它进程。

Namespace—PID隔离（4）

```
int container_main(void* arg)
{
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container",10);
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main()
{
    printf("Parent [%5d] - start a container!\n", getpid());

    int container_pid = clone(container_main, container_stack+STACK_SIZE,
            CLONE_NEWUTS | CLONE_NEWPID | SIGCHLD, NULL);
    waitpid(container_pid, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

```
zm@ubuntu:~$ sudo ./pid
Parent [3474] - start a container!
Container [1] - inside the container!
root@container:~# echo $$
1
```

Namespace—PID隔离（5）

- 在容器里的shell输入ps、top等命令，仍会看到所有的进程。
  - 说明并没有进行完全隔离
  - Ps和top命令是通过读取/proc文件系统而获取当前的进程列表。然而，父子进程看到的/proc是一样的。
- 与其它namespace不同的是，为实现一个稳定、安全的容器，PID namespace还需要进行一些额外的工作才能确保其中的进程运行顺利。
  - 在容器中，启动的第一个进程需要实现类似init的功能，维护所有后续启动进程的运行状态。
    - Docker启动时，第一个进程是dockerinit，实现了进程监控和资源回收。

Namespace—PID隔离（6）

- 因init进程的特殊性，内核赋予了它特权—信号屏蔽。如果init中没有处理某个信号的代码逻辑，那么与init会屏蔽发送给它的该信号。这个功能的主要作用是防止init进程被误杀。
  - 父节点的进程如果发送发送SIGKILL或SIGSTOP，子节点的init会强制执行，即父节点中的进程有权终止子节点中的进程。如果并非这两个信号，则会被忽略。
  - 一旦init进程被销毁，同一PID namespace中的进程也会随之接收到SIGKILL信号。
- 如果只想看到PID namespace自身包含的进程，需要重新挂载/proc
  - mount -t proc proc /proc
- Unshare和sentus系统调用在PID namespace中的特殊性
  - 调用unshare()的进程并不进入新的PID namespace，而随后创建的子进程才会进入新的namespace，这个子进程成为新namespace中的init进程。setns()也类似。
  - 因为调用getpid()函数得到的PID是根据调用者所在的PID namespace而决定返回哪个PID，进入新的PID namespace会导致PID产生变化。

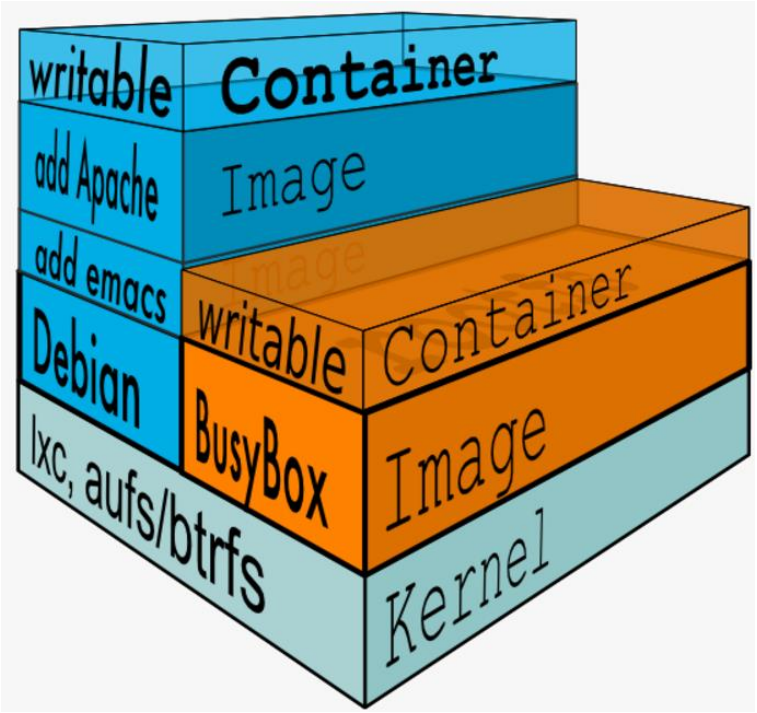
Namespace—mount隔离（1）

- Mount namespace通过隔离文件系统挂载点对文件系统的隔离提供支持，使不同容器使用不同的目录内容
  - 它是历史上第一个Linux namespace，所以它的标识位比较特殊，就是CLONE_NEWNS。
  - 隔离后，不同mount namespace中的文件结构发生变化也互不影响。
  - 在创建mount namespace时，会把当前的文件结构复制给新的namespace。新namespace中的所有mount操作都只影响自身的文件系统，而对外界不会产生任何影响
    - 通过/proc/[pid]/mounts查看到所有挂载在当前namespace中的文件系统
    - 通过/proc/[pid]/mountstats看到mount namespace中文件设备的统计信息，包括挂载文件的名字、文件系统类型、挂载位置等

Namespace—mount隔离（2）

- 首先需要一个rootfs(Root File System)
  - bootfs包含了bootloader和linux内核。用户是不能对这层作任何修改的。在内核启动之后，bootfs实际上会unmount掉。
  - rootfs则包含了一般系统上的常见目录结构，类似于/dev, /proc, /bin等等以及一些基本的文件和命令。
  - 可手动制作，也可通过工具制作

- 当前容器的文件系统是分层的，它的rootfs在mount之后会转为只读模式。 image是只读的，container部分则是可写的
- 用户想要修改底层只读层上的文件，这个文件就会被先拷贝到上层，修改后驻留在上层，并屏蔽原有的下层文件。



Namespace—mount隔离（3）

- 传统的mount namespace严格地实现了隔离，但是某些情况可能并不适用
  - 当父节点中的进程挂载了一张CD-ROM，这时子节点拷贝的目录结构无法自动挂载上这张CD-ROM
- 挂载传播（mount propagation）解决了这个问题
  - 挂载传播定义了挂载对象（mount object）之间的关系，系统用这些关系决定挂载对象中的挂载事件如何传播到其它挂载对象
    - 所谓传播事件，是指由一个挂载对象的状态变化导致的其它挂载对象的挂载与解除挂载动作的事件。
    - 共享关系（share relationship）：如果两个挂载对象具有共享关系，那么一个挂载对象中的挂载事件会传播到另一个挂载对象，反之亦然。
    - 从属关系（slave relationship）：如果两个挂载对象形成从属关系，那么一个挂载对象中的挂载事件会传播到另一个挂载对象，但是反过来不行；在这种关系中，从属对象是事件的接收者。

Namespace—mount隔离（4）

- 挂载状态
  - 共享挂载（shared）
    - 传播事件的挂载对象称为共享挂载
  - 从属挂载（slave）
    - 接收传播事件的挂载对象称为从属挂载
  - 共享/从属挂载（shared and slave）
  - 私有挂载（private）
    - 既不传播也不接收传播事件的挂载对象称为私有挂载
  - 不可绑定挂载（unbindable）
    - 与私有挂载相似，但是不允许执行绑定挂载，即创建mount namespace时这些文件对象不可被复制。

Namespace—mount隔离（5）

- 设置为共享挂载的命令
  - mount --make-shared <mount-object>
  - 从共享挂载克隆的挂载对象也是共享的挂载；它们相互传播挂载事件。
- 设置为从属挂载的命令
  - mount --make-slave <shared-mount-object>
  - 来自从属挂载克隆的挂载对象也是从属挂载，从属于原来主挂载对象。
- 将一个从属挂载对象设置为共享/从属挂载
  - 1) mount –make-shared <slave-mount-object>
  - 2) 将其移动到一个共享挂载对象下
- 把修改过的挂载对象重新标记为私有的，可以执行如下命令。
  - mount --make-private <mount-object>
- 将挂载对象标记为不可绑定
  - mount --make-unbindable <mount-object>

Namespace—mount隔离（6）

```
int container_main(void* arg)
{
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container",10);

    system("mount -t proc proc /proc");
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}
```

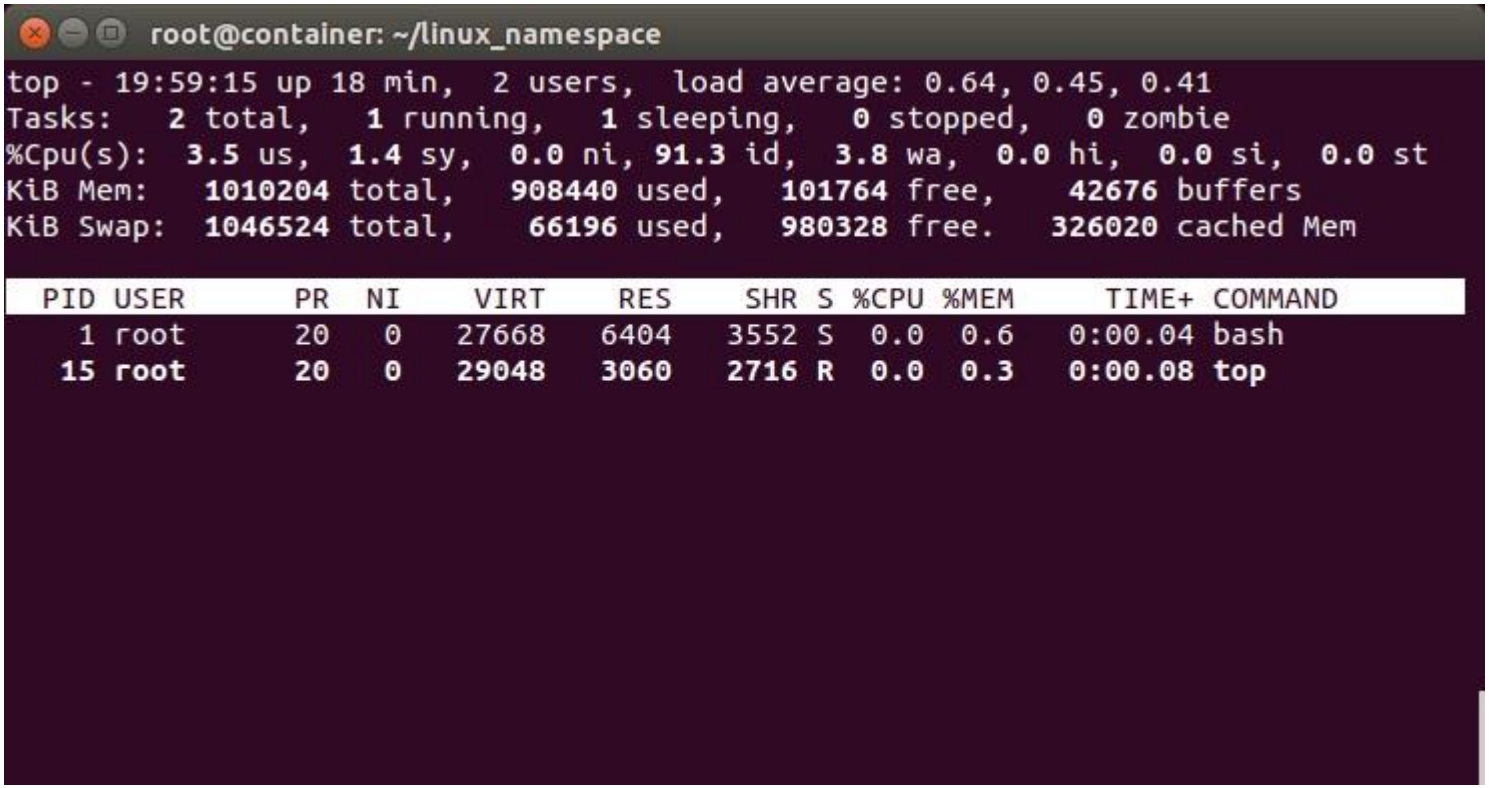
```
int main()
{
    printf("Parent [%5d] - start a container!\n", getpid());

    int container_pid = clone(container_main, container_stack+STACK_SIZE,
            CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | SIGCHLD, NULL);
    waitpid(container_pid, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

运行结果如下

```
zm@ubuntu:~$ sudo ./pid.mnt
Parent [ 3502] - start a container!
Container [    1] - inside the container!
root@container:~# ps -elf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 -  6917 wait   19:55 pts/2    00:00:00 /bin/bash
0 R root        14     1  0  80   0 -  5671 -      19:56 pts/2    00:00:00 ps -elf
```

Namespace—mount隔离（7）

top运行结果如下



Namespace—USER隔离（1）

- User namespace主要隔离了安全相关的标识符（identifiers）和属性（attributes），包括用户ID、用户组ID、root目录、key（指密钥）以及特殊权限。
  - 一个普通用户的进程通过clone()创建的新进程在新user namespace中可以拥有不同的用户和用户组。
    - 一个在容器外的非特权用户进程，可以创建有超级用户权限的容器内进程。
    - User namespace是目前的六个namespace中最后一个支持的
    - 目前Docker也还不支持user namespace，但是预留了相应接口。然而，Docker用到了在user namespace中提及的Capabilities机制。

  Linux把原来和超级用户相关的高级权限划分成为不同的单元，称为Capability。这样就可以独立地对特定Capability进行使能或禁止。

Namespace—USER隔离（2）

- 容器内外UID, GID映射
  - **/proc/<pid>/uid_map　　/proc/<pid>/gid_map**
  - 格式：ID-inside-ns ID-outside-ns length
    - ID-inside-ns表示新建的user namespace中对应的user/group ID
    - ID-outside-ns表示namespace外部映射的user/group ID。
    - length表示映射范围，通常为1，表示只映射一个，否则，则按顺序一一映射。
  - 把真实的uid=1000映射成容器内的uid=0

```
$ cat /proc/2465/uid_map
    0       1000          1
```

  - 这两个文件只允许拥有该user namespace中CAP_SETUID权限的进程写入一次，不允许修改。
  - 写入的进程必须是该user namespace的父namespace或者子namespace。

Namespace—USER隔离（3）

```
int container_main(void* arg)
{
    printf("Container [%5d] - inside the container!\n", getpid());

    printf("Container: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
           (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    ……
    printf("Container [%5d] - setup hostname!\n", getpid());

    sethostname("container",10);

    mount("proc", "/proc", "proc", 0, NULL);

    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}
```

```
void set_uid_map(pid_t pid, int inside_id, int outside_id, int length)
{
    char path[256];
    sprintf(path, "/proc/%d/uid_map", getpid());
    FILE* uid_map = fopen(path, "w");
    fprintf(uid_map, "%d %d %d", inside_id, outside_id, length);
    fclose(uid_map);
}
```

```
int main()
{
    const int gid=getgid(), uid=getuid();

    printf("Parent: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
           (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    pipe(pipefd);

    printf("Parent [%5d] - start a container!\n", getpid());

    int container_pid = clone(container_main, container_stack+STACK_SIZE,
            CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUSER | SIGCHLD, NULL);


    printf("Parent [%5d] - Container [%5d]!\n", getpid(), container_pid);

    set_uid_map(container_pid, 0, uid, 1);
    set_gid_map(container_pid, 0, gid, 1);

    printf("Parent [%5d] - user/group mapping done!\n", getpid());

    ……
    printf("Parent - container stopped!\n");
    return 0;
}
```

Namespace—USER隔离（4）

启动user隔离前

```
zm@ubuntu:~$ id
uid=1000(zm) gid=1000(zm) groups=1000(zm)
```

启动user隔离程序

```
zm@ubuntu:~$ ./user
Parent: eUID = 1000;  eGID = 1000, UID=1000, GID=1000
Parent [ 3262] - start a container!
Parent [ 3262] - Container [ 3263]!
Parent [ 3262] - user/group mapping done!
Container [  1] - inside the container!
Container: eUID = 0;  eGID = 0, UID=0, GID=0
Container [  1] - setup hostname!
```

在容器内查看id

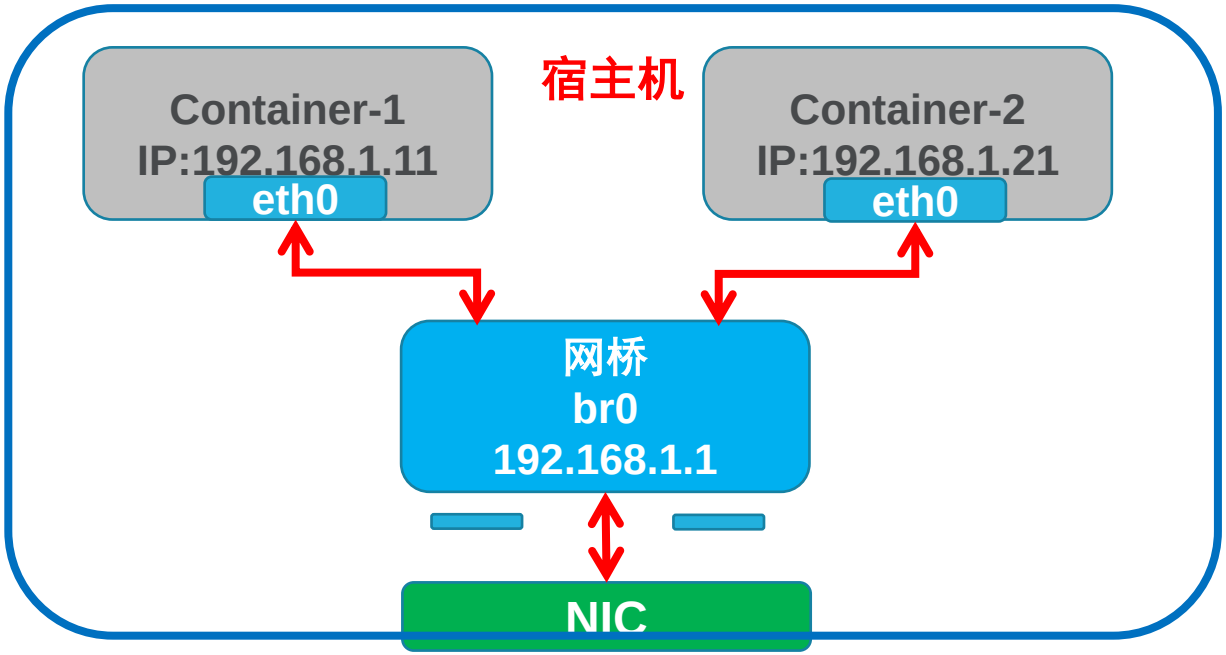
```
root@container:~# id

uid=0(root) gid=0(root) groups=0(root)
```

Namespace—Network隔离（1）

- 当在新建的namespace中启动一个“Apache”进程时，因主机上已经运行了一个“Apache”进程，出现“80端口已被占用”的错误。怎么办？
  - Network namespace主要提供关于网络资源的隔离，包括网络设备、IPv4和IPv6协议栈、IP路由表、防火墙、/proc/net目录、/sys/class/net目录、端口（socket）等
  - 一个物理的网络设备最多存在在一个network namespace中。一般情况下，物理网络设备分配在最初的root namespace中。
  - 如果有多块物理网卡，可以把其中一块或多块分配给新创建的network namespace。需要注意的是，当新创建的network namespace被释放时，在这个namespace中的物理网卡会返回到root namespace而非创建该进程的父进程所在的network namespace。

Namespace—Network隔离（2）

- 为实现网络隔离和通信，经典做法是创建一个veth pair，一端放置在新的namespace中，通常命名为eth0，一端放在原namespace中连接物理网络设备，再经网桥把所有设备连接进行路由转发
  - veth pair（虚拟网络设备对：有两端，类似管道，如果数据从一端传入另一端也能接收到，反之亦然）



Namespace—Network隔离（3）

- 代码中建立和测试network namespace较为复杂，可以通过命令行工具ip创建network namespace
  - ip netns add test_ns //创建一个命名为test_ns的network namespace
  - 通过ip netns exec命令可以在新建的network namespace下运行网络管理命令
    - ip netns exec test_ns ip link list //展示了新建namespace下可见的网络链接
    - ip netns exec test_ns ip link set dev eth0 up //启动eth0
  - 要实现与外部namespace进行通信还需要再建一个网络设备对
    - ip link add veth1 type veth peer name veth0 //创建网络设备对，发送到veth1的包veth0也能接收到，反之亦然。
    - ip link set veth0 netns test_ns //将网络设备对的veth0端分配给新建的namespace
    - ip netns exec test_ns ifconfig veth0 10.1.1.1/24 up //配置namespace中veth0的IP地址
    - ifconfig veth0 10.1.1.2/24 up //配置宿主机（原namespace）一端的IP并启用

此时连接外网是不可能的，可以通过建立网桥或者NAT映射来决定这个问题

Namespace操作（4）

- Namespace主要通过三个系统调用完成空间的隔离
  - 使用clone()来创建一个独立namespace的进程是最常见做法：创建一个新的进程，通过参数得到隔离的效果
  - Unshare：使某进程脱离某个namespace
  - Setns：把某进程加入到某个namespace

查看namespace文件（5）

- 可以在/proc/[pid]/ns文件下看到指向namespace号的文件
  - $ ls -l /proc/1024/ns

```
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user->user:[4026531837]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]
```

  - 如果两个进程指向的namespace编号相同，就说明他们在同一个namespace下，否则则在不同namespace里面。

设置Namespace（6）

- 通过setns()加入一个已经存在的namespace
  - 通过setns()系统调用，可以使进程从原先的namespace加入准备好的新namespace：int setns (int fd, int nstype)。
    - 参数fd表示我们要加入的namespace的文件描述符
    - 参数nstype让调用者可以去检查fd指向的namespace类型是否符合要求

  如：

```
fd = open(argv[1], O_RDONLY);   /* 获取namespace文件描述符 */
setns(fd, 0);                   /* 加入新的namespace */
execvp(argv[2], &argv[2]);      /* 执行程序 */
```

  ./ns_set_pgm ~/ns-file /bin/bash

  - 至此，可以在新的命名空间中执行shell命令了

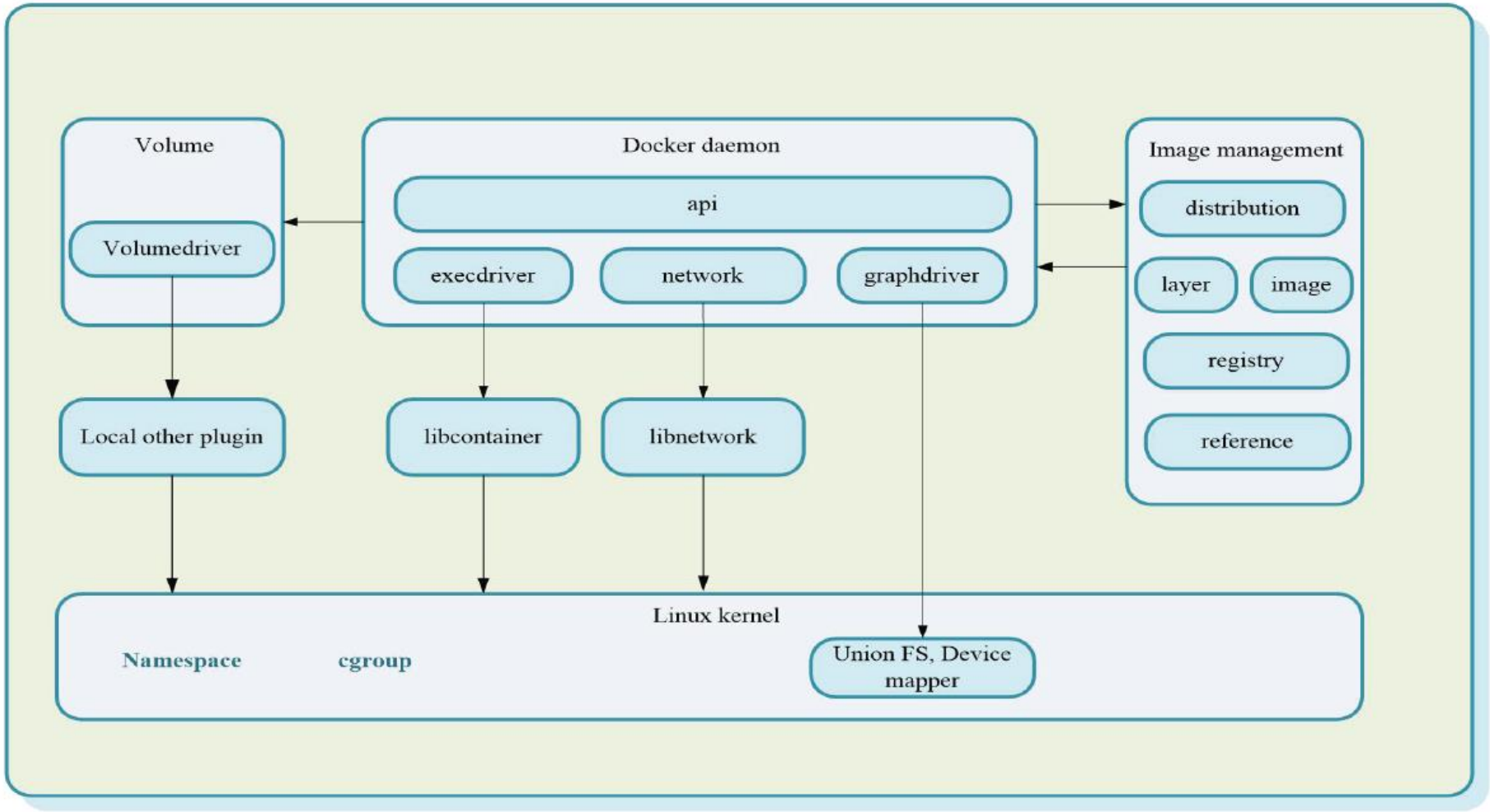
设置Namespace（7）

- 通过unshare()在原先进程上进行namespace隔离
  - 系统调用unshare()跟clone()功能类似，不同的是，unshare()运行在原先的进程上，不需要启动一个新进程
  - 调用unshare()的主要作用是不启动一个新进程就可以起到隔离的效果，相当于跳出原先的namespace进行操作。这样，你就可以在原进程进行一些需要隔离的操作。

[第15次课]容器技术与安全

- 容器与虚拟机
- 容器技术
- 容器安全

Docker框架图


Volume
Volumedriver
Docker daemon
api
execdriver
network
graphdriver
Image management
distribution
layer
image
registry
reference
Local other plugin
libcontainer
libnetwork
Linux kernel
Namespace
cgroup
Union FS, Device mapper

Docker安全问题

- 容器的磁盘资源限制问题
- 容器DoS攻击与流量限制问题
- 容器逃逸问题
- 容器超级权限开放问题

磁盘资源限制

- 容器本质上是一个进程，通过镜像层叠的方式来构建容易的文件系统。其本质上还是在宿主机文件系统的某一个目录下存储这些信息。所有容器的rootfs最终存储在宿主机上
- 存在宿主机磁盘容量被耗尽的情况，其它容器无法进行文件存储。需要多容器的磁盘使用量进行限制

容器DoS攻击与流量限制

- 当前在公网上的DoS攻击具有较成熟的防护产品，对传统网络有较好的防御效果，但容器中，攻击数据包可以不需要通过物理网卡就可以攻击同一个宿主机下的其它容器。传统DoS防御措施对容器之间的DoS攻击没有太大效果
- 默认的容器网络是网桥模式，所有容器都连接在网桥上。容器内网卡发出的数据包都会发往宿主机上的对应网卡，再由物理网卡转发。实际上所有容器共用一个卡，如果同宿主机中某一个容器抢占了大部分带宽，将会影响其它容器的使用。

容器逃逸

- 当容器从jail逃出，所有的容器以及宿主机都会受到威胁。
- 2014年通过open_by_handle_atd调用，可以暴力扫描宿主机文件IT获取宿主机敏感文件，以此达到逃逸，其逃逸过程如下：
  - 程序首先打开从苏书记文件系统挂载到容器内的某一文件，以获取宿主机的文件描述符引用，作为open_by_handle_at的第一个参数。之后的所有扫描都在宿主机的文件系统上进行，而非容器自身的rootfs
  - 构造根目录的file_handle类型的数据，根目录的inode编号为2，所以在f_handle中指定节点编号
  - 打开file_handle所描述的目录，遍历此目录下的所有文件。比较目录的名字和所要查找文件的目录，找到则记录其inode号。如此重复，直到查找成功
  - 将输入的最后文件file_handle描述符输出到oh参数中，所要查找的文件的file_handle结构已经构造完成
  - 通过以上步骤获取了目标文件的文件描述符，则可以进行任何操作

超级权限

- Docker引入超级权限，在docker run时加上--privileged可使容器获得超级权限
  - 如果制定了privileged参数，则调用setPrivileged操作
    - 通过capabilities.GetAllCapabilities()获取超级用户所有能力赋给容器
    - 通过GetHostDeviceNodes()扫描宿主机所有设备文件挂载到容器
  - 能力的主体是进程，通过限制进程的能力限制用户的权限
    - Effecitve：表示进程当前有效能力位图
    - Permitted：表示进程可以使用的能力位图
    - Inheritable：当前进程的子进程可以继承的能力位图
    - Bounding限制可以加入到inheritable集合的能力
  - GetHostDeviceNodes调用getDeviceNodes(“/dev”)，读取/dev目录下的所有文件，返回给调用者
    - 加了—privileged参数时，超级用户的权限全部复制给当前容器（四个位图都是1fffffffff），并且所有的设备文件都挂载在容器内

安全机制-容器 daemon安全

- 容器daemon安全
  - Docker向外界服务有四种通信形式，默认是以Unix套接字的方式与客户端通信，相对于TCP较安全
    - 只有进入daemon宿主机所在的及其并且有权访问daemon的域套接字才能与daemon建立通信
  - 如果以TCP方式提供服务，可以访问到daemon所在主机的用户都可以成为潜在的攻击者，并且数据可能被截获或修改
  - Docker提供了TLS传输层安全协议
    - 通过—tlsverify进行安全传输检验，通过—tlscacert、--tlskey、--tlscert三个参数来配置，客户端可以对服务端验证
- 容器之间的网络安全
  - Docker daemon指定– icc标志的时候，可以禁止容器与容器之间的通信，主要设定iptables规则实现

安全机制-镜像安全

- Docker目前提供registry访问权限控制以保证镜像的安全，并且提供了镜像数据签名功能，以防止镜像被篡改或损坏
  - Docker registry访问控制
    - 授权文件official.json，docker daemon第一次启动时会通过公网载入。包含两部分：
    - 第一部分针对特定registry目录的授权信息，包括公共镜像目录以及用户自己创建镜像的目录。
    - 第二部分是授权文件的数字签名信息，包括证书信息以及证书与文件内容组合而成的签名。
  - 镜像校验和：用来保证镜像的完整性，防止镜像破坏
    - Docker registry下的每个镜像都对应拥有自己的manifest文件以及该文件的签名，包括镜像所在的命名空间、镜像对应的标签、镜像校验方法以及校验和等
    - 获取镜像tag所对应的manifest，提取镜像ID
    - 提取镜像校验和和字段，得到镜像的实例
    - 得到完整校验和并于manifest文件中的进行对比验证

安全机制-内核安全

- 内核为容器主要提供了两种技术cgroups和namespaces，分别对容器进行资源限制和资源隔离，使容器感觉像是在一台独立主机中运行
  - ◆Cgroups资源限制
    - 容器本质是是进程，cgroups应用在容器是为了限制宿主机不同容器的资源使用量，避免单个容器好近宿主机资源而导致其它容器异常
  - ◆Namespaces资源隔离
    - 为了使容器处于独立环境中，Docker使用namespaces技术隔离容器，使容器与容器之间，容器与宿主机之间相互隔离
    - 容器目前只对uts、ipc、pid、network、mount这5种namespace进行完全隔离，user-namespace并为全部支持。除了上述资源之外，还有很多系统资源未被隔离，如/proc、/sys信息、SELinux、time、syslog和/dev设备等均未被隔离。

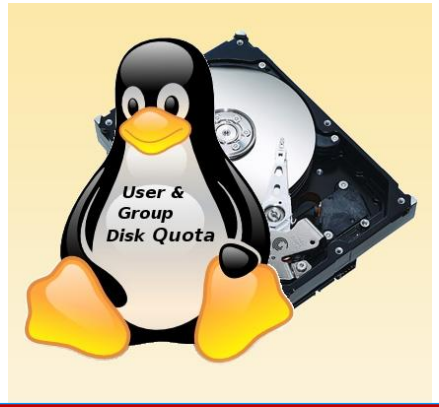
  当前内核安全只是满足了可用，离真正安全还有距离

安全机制-容器capability限制

- 容器run参数中提供了容器能力配置接口，可用在创建容器室在容器默认能力基础上对容器的能力进行增加和减少
  - 增添能力docker run --cap-add=[]；
    - docker run --cap-drop ALL --cap-add SYS_TIME ntpd /bin/sh
  - 消减能力docker run --cap-drop=[]
    - docker run --cap-drop SETUID --cap-drop SETGID ntpd /bin/sh
  - 所谓的能力就是代表用户所能执行的系统调用操作，以此来切割超级用户权限。如果被剥夺了这些能力，root也无法执行系统调用
  - 对于普通用户，有时需要使用root的权限，但是为了安全不能把普通用户提升为root用户，即可采用增添能力的方式。
  - 容器默认拥有CHOWN、DAC_OVERRIDE、FSETID、SETUID、SETFCAP、SYS_CHROOT等14个能力

安全机制-OS机制

- Docker通过一些额外的工具来加强自身的安全：
  - SELinux限制进程访问资源
  - 使用quota等技术限制容器磁盘使用量
  - 使用Traffic Controller技术对容器的流量进行控制



需注意点是：没有计算机系统是绝对安全的
安全机制只是为了提高攻击者的代价。

- SELinux是内核实现的强制访问控制(MAC)，SELinux实现了一个MAC系统，为每个进程提供标签(进程的域)，为文件设置标签(类型)
  - 标签由user、type、role、和level四部分组成
  - User:由权限构成的集合，非Linux用户。系统登录时为Linux用户匹配一个SELinux用户。默认会被映射到SELinux的unconfined_u用户。用户还可以自定义新用户
  - Type：是访问控制的基础，描述进程所能访问呢的资源类型
    - 常用文件资源的类型有blk_file, che_file, dir, fd, fifo_file, file, filesystem, lnk_file和sock_file等
    - 容器文件一般表示为svirt_sandbox_file_t或svirt_lxc_file_t，容器域一般用svirt_lxc_net_t标示
  - Role：一些类型的集合，是用户和类型的过渡
    - 一个用户可以有多个角色，一个角色可以使用不同类型
  - Level：定义具体的权限，可有两种选择，MLS (多层健全)或MCS (多级分类安全)
    - MLS从高到低将权限分为topsecret、secret、condifidental和unclassified四个级别，采用BLP模型，对每个实体进行分类。高级别实体不能写低级别实体，低级别实体不能读高级别实体
    - MCS通过敏感度和分类表示，从s0~s15共16个敏感级。分类进行数据划分，对文件的同一类型或同一域的数据打上标签。访问数据时必须具有足够的敏感度和正确的分类

- SELinux的3中模式，通过setenforce设置：setenforce 1/0/-1
  - Enforcing(1)：策略被强制执行，根据策略来拒绝或通过某操作
  - Permissive(0)：策略并不会执行，本该在Enforcing下应该拒绝的操作，此时只会触发安全事件日志记录、不会遭受拒绝
  - Disabled(-1)：SELinux被关闭
- SELinux 的3种访问控制方式
  - Type Enforcement：类型强制，是主要访问控制机制
  - Role-Based Access Control (RBAC)：基于SELinux用户
  - Multi-Level Security (MLS)：多层级安全
- SELinux是对现有的用户、用户组进行文件读写和执行的安全增强，主客体级别明确后，通过策略按指定的规则执行，规则有4部分组成
  - 源类型：进程类型—域；目标类型：被进程访问的客体的类型—类型；客体类别：允许访问的课题类别；权限：源类型可以对目标类型所做的操作。
    - Allow sshd_t console_device_t : chr_file {ioctl write getattr lock append open};
    - 源类型sshd_t允许访问console_device_t类型的客体，可执行的权限是ioctl、getatr、lock等操作

SELinux

- SELinux把所有的进程和文件打上标签，进程之间相互隔离，SELinux策略控制进程（容器）如何访问资源
- SELinux策略是全局的，不是针对具体用户而定，攻击者难以突破
- 减少提权风险，攻陷某个进程后，只能获取其对应的权限
  - 比如Apache的hpptd被攻陷，则攻击者只能访问httpd所能访问的文件
  - SELinux不是对容器现有安全体制进行替换，而是增添一道防线
- 在容器（Docker）中使用SELinux
  - Docker启动时，执行docker –d –selinux-enabled=true，Docker daemon则启用SELinux
  - Docker run的时候可以指定容器的user、role、type、level等标签
  - 通过security-opt参数这是容器进程的用户、角色、域、级别等信息
    - 如果启用了SELinux、但是docker run时不指定标签信息，运行会失败
    - Docker对SELinux的3种访问控制方式都可支持
  - Docker创建的每一个容器的进程域是相同的，但是进程的MCS是不一样的，能够访问的目标文件也不一样，从而达到容器之间的隔离以及容器与容器文件之间的隔离

Qutoa

- 目前cgroups没有对磁盘资源进行限制，采用quota技术可以对磁盘限额
  - Quota基于用户和文件系统，而基于基础或者目录的磁盘限额比较繁琐
- 容器磁盘限额的方法
  - 为每个容器创建一个用户，所有用户共用宿主机磁盘，磁盘限额只对普通用户有效
  - 选择支持可以对某目录进行限额的文件系统
    - 比如XFS可以对用户、用户组、目录、项目等形式对磁盘使用量进行限制
  - 创建虚拟文件系统，此文件系统只供一个容器使用
- 构建虚拟文件系统，作为rootfs最顶层的layer
  - 创建虚拟文件系统，将容器的rootfs构建与虚拟文件系统之上
    - Linux支持从磁盘文件创建爱你一个虚拟文件系统，将其设置为容器所用的文件系统，从而达到限额的目的
    - 创建4GB文件：sudo dd if=/dev/zero of=/usr/disk-quota.ext3 count=4096 bs=1MB
    - 创建文件系统：mkfs –t ext3 –F /usr/disk-quota.ext3
    - 挂载到目录：mount –o loop, rw, usrquota, grpquota /usr/disk-quota.ext3 /path/to/iage/top/lever

Traffic Controller

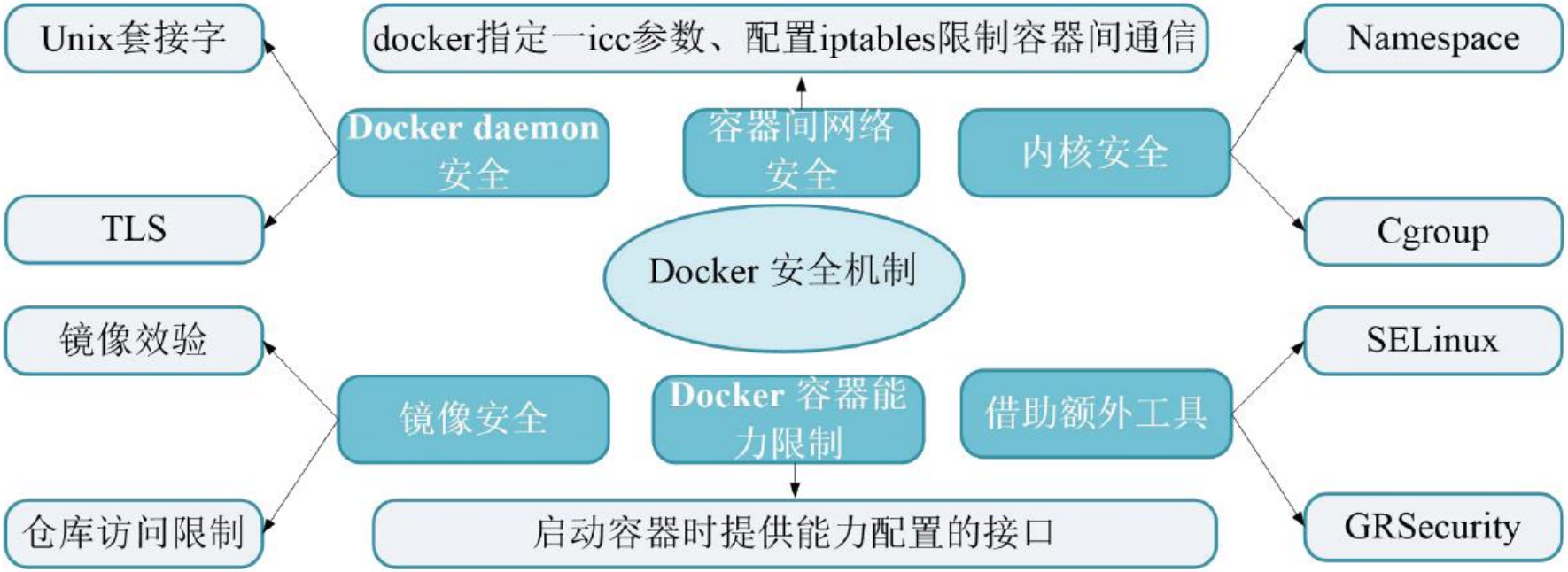
- Cgroup和namespace可以对容器的资源进行限制，但是在网络带宽方面却没有限制，尤其是构建容器云，存在多租户共同使用宿主机资源。无限制的大流量访问会破坏容器的实时交互能力

- Traffic Controller是Linux的流量控制模块，通过对数据包建立队列，并定义数据包发送规则，而实现对流量的限制、调度等控制操作。队列分为两种
  - 无类队列：对进入网卡的数据进行统一对待，能够接受数据包并对网卡流整形，但不能对数据包进行细致分类，其手段主要是排序、限速以及丢包
  - 分类队列：过滤器对进入网卡的数据根据不同需求进行分类，把数据包发送到相应队列中排队。每个子类可以使用自己的过滤器对数据进一步分类，直到不需再分类
  - Traffic Controller流量控制方式有4种
    - SHAPING：传输速率被控制在某个值以下，限制阈值可以小于有效带宽
    - SCHEDULING：根据数据包传输的优先级，对不同的传输流按照优先级分配
    - POLICING：用于处理接收到的数据
    - DROPPING：如果流量超过设置的阈值则丢弃数据包，对内外流量皆有效

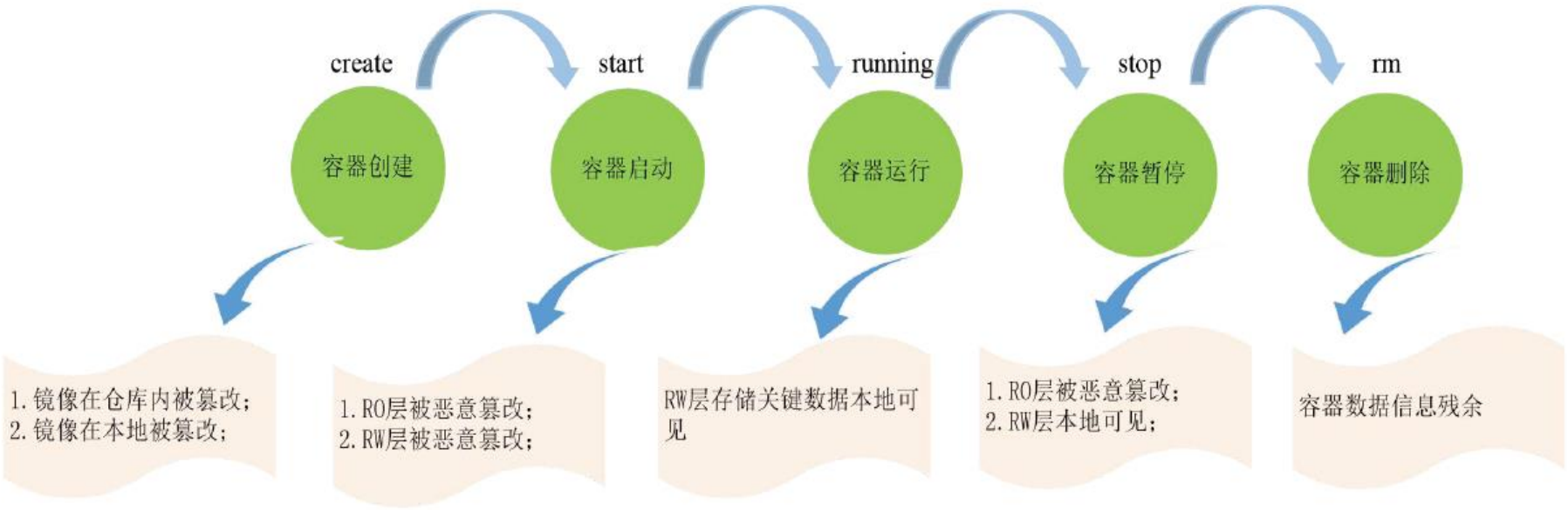
Traffic Controller

- 无类队列的使用
  - TBF是一种常用的无类队列，只是对数据包进行SHAPING，并不作SCHEDULING
    - tc disc add dev eth1 root handle 1:0 tbf rate 128kbit burst 1000 latency 50ms
- 分类队列的使用
  - 对不同的数据类型进行区分对待，CBQ是一种常用的分类队列。分类队列有过滤器和类。过滤器将数据包划分到不同的类，可递归处理类
    - 物理网卡100Mbit/s，有三个服务ftp、snmp和http
    - 首先建立根队列；然后再次队列上建立三个类；在三个队列上建立三个类；为根队列建立三个过滤器
- 容器会通过veth pari技术重建一对虚拟网卡，如果要对容器的流量进行监控，将Traffic Controller中的dev指定为veth*。
  - 在容器创建时添加此规则
  - 如果不需要容器之间在三四层通信，指定icc参数可以禁止容器间的直接通信
  - 采用Traffic Controller对容器网卡容量进行限制，防止同宿主机容器之间的DoS攻击

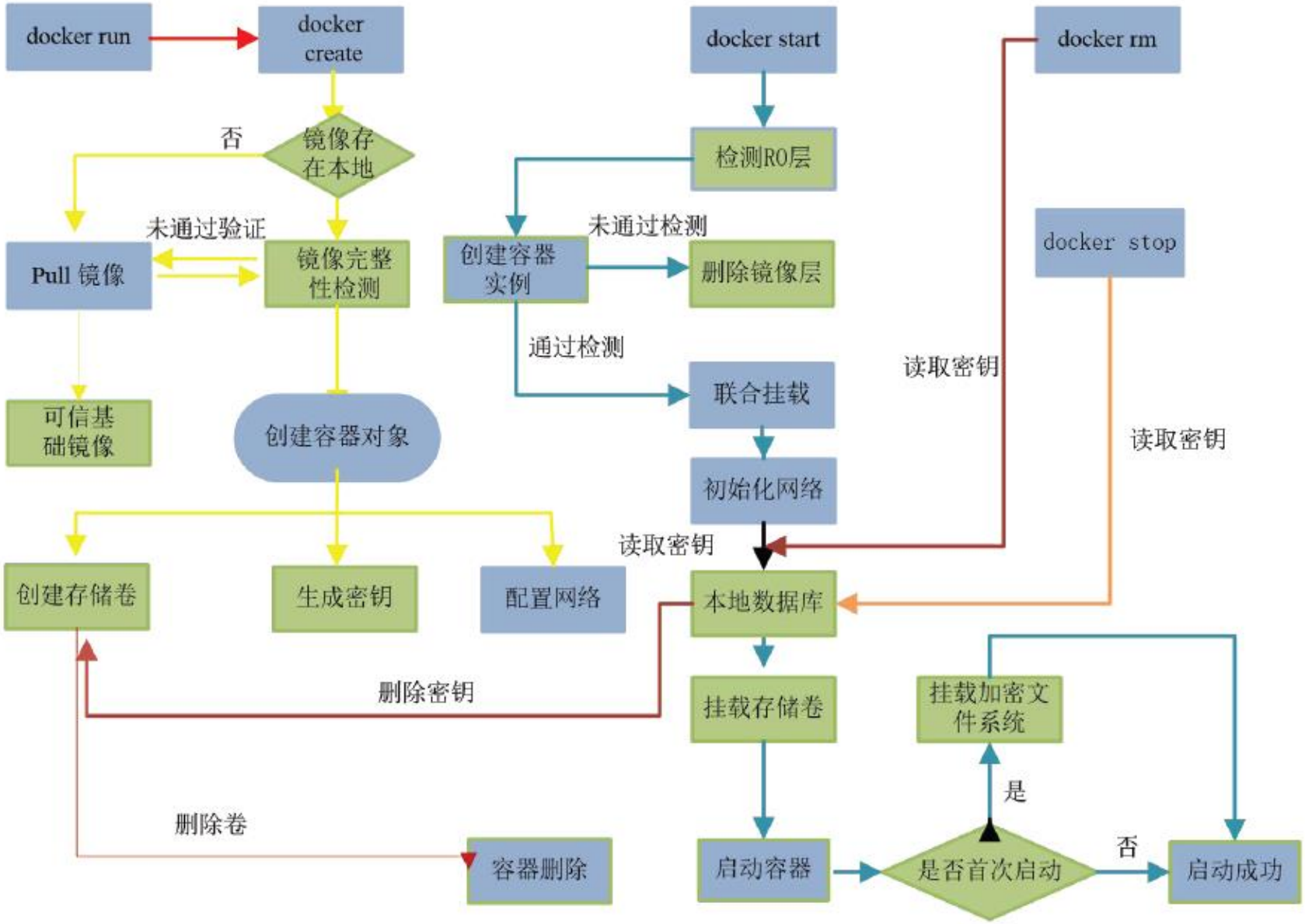
Docker官方安全方案



容器全生命周期安全（1）


create
start
running
stop
rm
容器创建
容器启动
容器运行
容器暂停
容器删除
1. 镜像在仓库内被篡改；
2. 镜像在本地被篡改；
1. RO层被恶意篡改；
2. RW层被恶意篡改；
RW层存储关键数据本地可见
1. RO层被恶意篡改；
2. RW层本地可见；
容器数据信息残余

容器全生命周期安全（2）


docker run
docker create
否
镜像存在本地
未通过验证
Pull 镜像
镜像完整性检测
可信基础镜像
创建容器对象
创建存储卷
生成密钥
配置网络
docker start
检测RO层
未通过检测
创建容器实例
删除镜像层
通过检测
联合挂载
初始化网络
读取密钥
本地数据库
删除密钥
挂载存储卷
删除卷
容器删除
启动容器
是否首次启动
是
否
挂载加密文件系统
启动成功
docker rm
读取密钥
docker stop
读取密钥

容器安全进阶

- 云原生安全
  - 微隔离
  - 安全服务功能链
- 机密容器

Q & A

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

T: 86 010-88889999 F: 86 010-88886666
E:daiyongming@zhongguokeyueyuan.com
地址：北京市海淀区闵庄路甲89号 100195