kaihong 开鸿

容器核心技术

讲师： 蒋卫峰

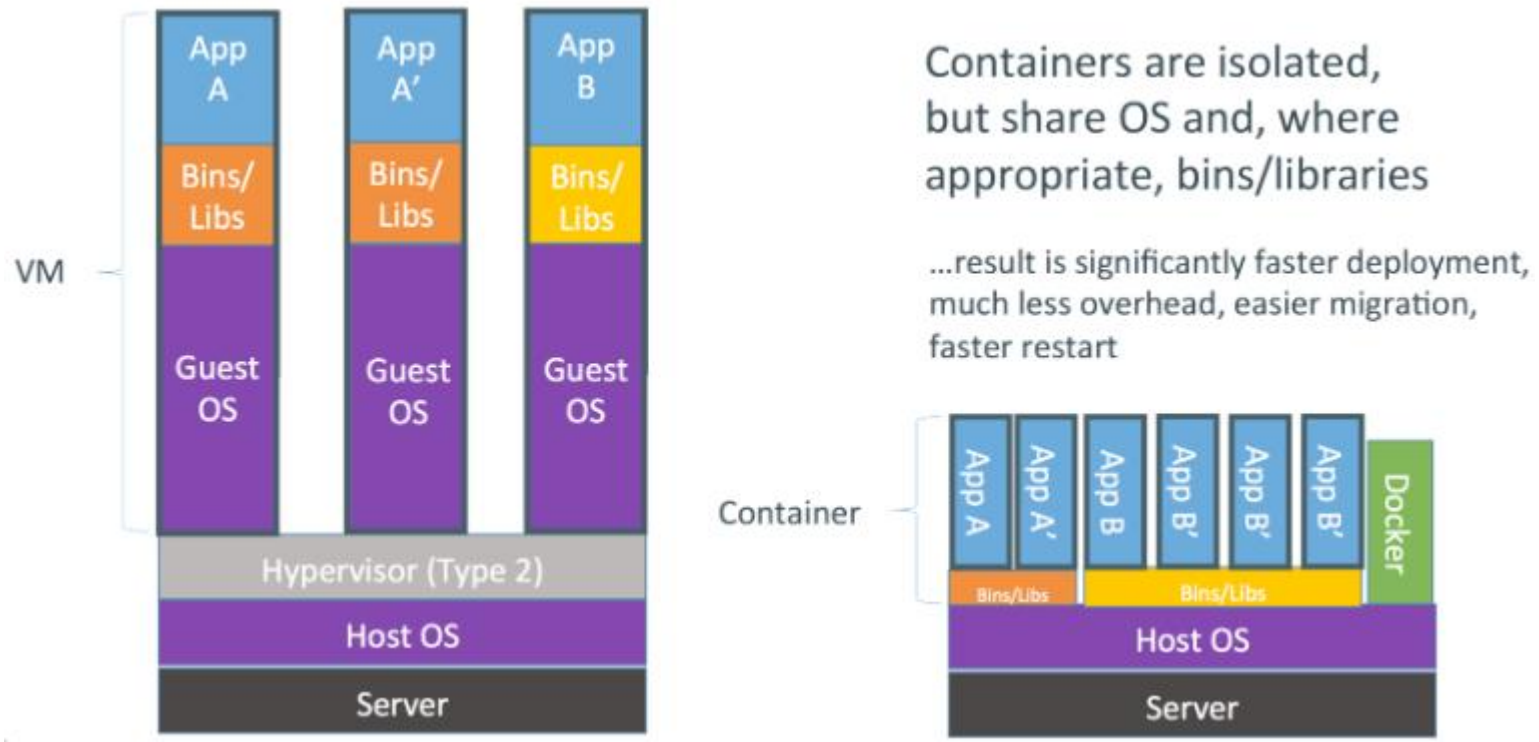
kaihong 开鸿

CONTENTS

01

容器相关知识

容器相关知识



容器借助 Linux 内核的 Namespaces、Apparmor、SELinux 情景模式（profile）、chroot 和 CGroup 等功能来提供类似于虚拟机的隔离环境。Linux 的安全模块可以确保正确地控制容器对宿主机和内核的访问，从而避免各种入侵活动。此外，在宿主机上可以运行不同的 Linux 发行版，只要它们运行在同样的 CPU 架构下。

docker历史

1979 — chroot
容器的概念始于 1979 年的 UNIX chroot，它是一个 UNIX 操作系统上的系统调用，用于将一个进程及其子进程的根目录改变到文件系统中的一个新位置，让这些进程只能访问到该目录。这个功能的想法是为每个进程提供独立的磁盘空间。其后在 1982年，它被加入到了 BSD 系统中。

2000 — FreeBSD Jails
FreeBSD Jails 是最早的容器技术之一，它由 R&D Associates 公司的 Derrick T. Woolworth 在 2000 年为 FreeBSD 引入。这是一个类似 chroot 的操作系统级的系统调用，但是为文件系统、用户、网络等的隔离增加了进程沙盒功能。因此，它可以为每个 jail 指定 IP 地址、可以对软件的安装和配置进行定制，等等。

2001 — Linux VServer
Linux VServer 是另外一种 jail 机制，它用于对计算机系统上的资源（如文件系统、CPU 处理时间、网络地址和内存等）进行安全地划分。每个所划分的分区叫做一个安全上下文（security context），在其中的虚拟系统叫做虚拟私有服务器（virtual private server，VPS）。

2004 — Solaris Containers
Solaris Containers 支持在 x86 和 SPARC 系统，首次出现在 2004 年 2 月发布的 Solaris 10 的 build 51 beta 上，其后完整发布在 2005 年的 Solaris 10 上。 Solaris Container 是由系统资源控制和通过 zones 提供的边界分离（boundary separation）所组合而成的。zones 是一个单一操作系统实例中的完全隔离的虚拟服务器。

2005 — OpenVZ
OpenVZ 类似于 Solaris Containers，它通过对 Linux 内核进行补丁来提供虚拟化、隔离、资源管理和状态检查（checkpointing）。每个 OpenVZ 容器都有一套隔离的文件系统、用户及用户组、进程树、网络、设备和 IPC 对象。

2006 — Process Containers
Process Containers 是由 Google 在 2006 年实现的，用于对一组进程进行限制、记账、隔离资源使用（CPU、内存、磁盘 I/O、网络等）。后来为了避免和 Linux 内核上下文中的“容器”一词混淆而改名为 Control Groups。它被合并到了 2.6.24 内核中。这表明 Google 很早就参与了容器技术的开发，以及它们是如何回馈到社区的。

docker历史

2007 — Control Groups
如上面所述，Control Groups （即 cgroups）是由 Google 实现的，并于 2007 年加到了 Linux 内核中。

2008 — LXC
LXC 的意思是 LinuX Containers，它是第一个最完善的 Linux 容器管理器的实现方案，是通过 cgroups 和 Linux 名字空间（namespace）实现的。LXC 存在于 liblxc 库中，提供了各种编程语言的 API 实现，包括 Python3、Python2、Lua、Go、Ruby 和 Haskell。与其它容器技术不同的是， LXC 可以工作在普通的 Linux 内核上，而不需要增加补丁。现在 LXC project 是由 Canonical 公司赞助并托管的。

2011 — Warden
Warden 是由 CloudFoundry 在 2011 年开发的，开始阶段是使用的 LXC，之后替换为他们自己的实现方案。不像 LXC，Warden 并不紧密耦合到 Linux 上，而是可以工作在任何可以提供隔离环境的操作系统上。它以后台守护进程的方式运行，为容器管理提供了 API。

2013 — LMCTFY
lmctfy 的意思是“让我为你包含（Let Me Contain That For You）”。这是一个 Google 容器技术的开源版本，提供 Linux 应用容器。Google 启动这个项目旨在提供性能可保证的、高资源利用率的、资源共享的、可超售的、接近零消耗的容器（参考自：lmctfy 演讲稿）。现在为 Kubernetes 所用的 cAdvisor 工具就是从 lmctfy 项目的成果开始的。lmctfy 首次发布于 2013 年10月，在 2015 年 Google 决定贡献核心的 lmctfy 概念，并抽象成 libcontainer，因此，lmctfy 现在已经没有活跃的开发了。

2013 — Docker
Docker 是到现在为止最流行和使用广泛的容器管理系统。它最初是一个叫做 dotCloud 的 PaaS 服务公司的内部项目，后来该公司改名为 Docker。类似 Warden，Docker 开始阶段使用的也是 LXC ，之后采用自己开发的 libcontainer 替代了它。不像其它的容器平台，Docker 引入了一整个管理容器的生态系统，这包括高效、分层的容器镜像模型、全局和本地的容器注册库、清晰的 REST API、命令行等等。稍后的阶段， Docker 推动实现了一个叫做 Docker Swarm 的容器集群管理方案。

docker历史

2014 — Rocket
Rocket 是由 CoreOS 所启动的项目，非常类似于 Docker，但是修复了一些 Docker 中发现的问题。CoreOS 说他们的目的是提供一个比 Docker 更严格的安全性和产品需求。更重要的是，它是在一个更加开放的标准 App Container 规范上实现的。在 Rocket 之外，CoreOS 也开发了其它几个可以用于 Docker 和 Kubernetes的容器相关的产品，如：CoreOS 操作系统、etcd 和 flannel。

2016 — Windows Containers
微软 2015 年也在 Windows Server 上为基于 Windows 的应用添加了容器支持，它称之为 Windows Containers。它与 Windows Server 2016 一同发布。通过该实现， Docker 可以原生地在 Windows 上运行 Docker 容器，而不需要启动一个虚拟机来运行 Docker（ Windows 上早期运行 Docker 需要使用 Linux 虚拟机）。

容器的现状
　　在业内部署软件应用从虚拟机逐渐移到了容器。其主要的原因是容器相比于虚拟机而言更加灵活和低消耗。Google 已经使用容器技术好多年了，它在Borg 和Omega 容器集群管理平台上可以成规模地运行 Google应用。更重要的是，Google为容器领域贡献了cgroups的实现和参与了libcontainer 项目。Google也在过去这些年借助容器在性能、资源利用和整体效率方面取得了巨大收益。最近，一直没有操作系统级的虚拟化技术的微软，也在Window Server上迅速采取动作实现了对容器的原生支持。
　　Docker 、Rocket 以及其它的容器平台并不能以一个单一主机运行在产品环境中，原因是这样面临着单点故障。当一组容器运行在一个单一宿主机时，如果宿主机失效，所有运行在该宿主机上的容器也会失效。要避免这个问题，应该使用容器宿主机集群。Google 借助其在Borg中取得的经验，开发了一个叫做Kubernetes的开源容器集群管理系统。Docker也启动了一个叫做Docker Swarm的解决方案。
　　微服务（Microservices ）是另一个突破性技术，在软件架构上可以将容器用于部署。微服务并不是一个新东西，只是一个相比标准的Web服务超快的轻量级Web服务。这是通过将功能单元（也许是一个单一服务或API方法）打包到一个服务中，并内嵌其到一个轻量级Web服务器软件中实现的。

02

Namespace内核实现

Linux Namespace简介

Linux Namespace 是 Linux 提供的一种内核级别环境隔离的方法。这种隔离机制和 chroot 很类似，chroot 是把某个目录修改为根目录，从而无法访问外部的内容。Linux Namesapce 在此基础之上，提供了对 UTS、IPC、Mount、PID、Network、User 等的隔离机制，如下所示。

分类	系统调用参数	相关内核版本
Mount Namespaces	CLONE_NEWNS	Linux 2.4.19
UTS Namespaces	CLONE_NEWUTS	Linux 2.6.19
IPC Namespaces	CLONE_NEWIPC	Linux 2.6.19
PID Namespaces	CLONE_NEWPID	Linux 2.6.19
Network Namespaces	CLONE_NEWNET	始于Linux 2.6.24 完成于 Linux 2.6.29
User Namespaces	CLONE_NEWUSER	始于 Linux 2.6.23 完成于 Linux 3.8)

```
//struct nsproxy用于汇集指向特定于子系统的命名空间包装器的指针
<nsproxy.h>
struct nsproxy {
    atomic_t count; //被使用计数
    struct uts_namespace *uts_ns; //使用指针，方便多个进程共享同一空间
    struct ipc_namespace *ipc_ns;
    struct mnt_namespace *mnt_ns;
    struct pid_namespace *pid_ns;
    struct user_namespace *user_ns;
    struct net *net_ns;
};
```

1 UTS CLONE_NEWUTS 主机名与域名
2 IPC CLONE_NEWIPC 信号量、共享内存
3 Mount CLONE_NEWNS 挂载点、文件系统
4 PID CLONE_NEWPID 进程编号
5 User CLONE_NEWUSER 用户和用户组
6 Network CLONE_NEWNET 网络设备

Linux Namespace-应用场景

kaihong 开鸿

一:主机名隔离空间的使用 (以下使用同一台机器,两个终端,以1,2区别，使用ushare --uts)

```
2]# ushare --uts //开启命名空间
2]# hostname bbb //设置主机名
2]# bash
2]# exit
1]# echo $HOSTNAME
1]# ubuntu                    //终端2改变了,但是终端1未影响，还是原来的主机名
```

二: 文件系统mount 与用户user隔离 (unshare –mount)

```
———— 终端1————-
1]# mount -l // 查看可用的mount 文件
1]# cd /var/tmp
1]# lftp 192.168.1.254
lftp 192.168.1.254:-> ls
lftp 192.168.1.254:/> cd ios/
lftp 192.168.1.254:/iso>get RHEL7-extras.iso
lftp 192.168.1.254:/iso> bye
tmp]# ls
tmp]# unshare --mount
tmp]# mount -t iso9660 -o loop,ro RHEL7-extras.iso /mnt/
tmp]# mount -l | grep iso
———— 终端2————-
2]# mount -l | grep iso 终端2 看不见终端1的mount信息
```

Linux Namespace-应用场景

三:IPC & PID 隔离 （unshare –pid –ipc –fork –mount –proc /bin/bash）

——— 终端1———-
1]# ps -ef
1]# systemctl stop docker
1]# pstree -p
systemd(1)
1]# unshare --pid --ipc --fork --mount-proc /bin/bash
1]#pstree -p　　//发现2346进程
——— 终端2———-
2]# kill -9 2346 //在终端2上随便找一个进程杀死，由于已隔离，无法杀死

Linux Namespace-应用场景

四:IPC 隔离
核心代码
```
//[...]
int child_pid = clone(child_main, child_stack+STACK_SIZE,
        CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD, NULL);
//[...]
```
产生一个子进程
——— 终端2———-
shell中使用ipcmk -Q命令建立一个message queue。
```
root@local:~# ipcmk -Q
Message queue id: 32769
root@local:~# ipcs -q
------ Message Queues --------
key        msqid   owner   perms   used-bytes   messages
0x4cf5e29f 32769   root    644     0            0
```
——— 终端1———-调用上述核心代码和新建的子进程中调用的shell中执行ipcs -q查看message queue
```
root@local:~# gcc -Wall ipc.c -o ipc.o && ./ipc.o
程序开始:
在子进程中!
root@NewNamespace:~# ipcs -q
------ Message Queues --------
key   msqid   owner   perms   used-bytes   messages
root@NewNamespace:~# exit
exit
```

Linux Namespace-应用场景

五:User namespace

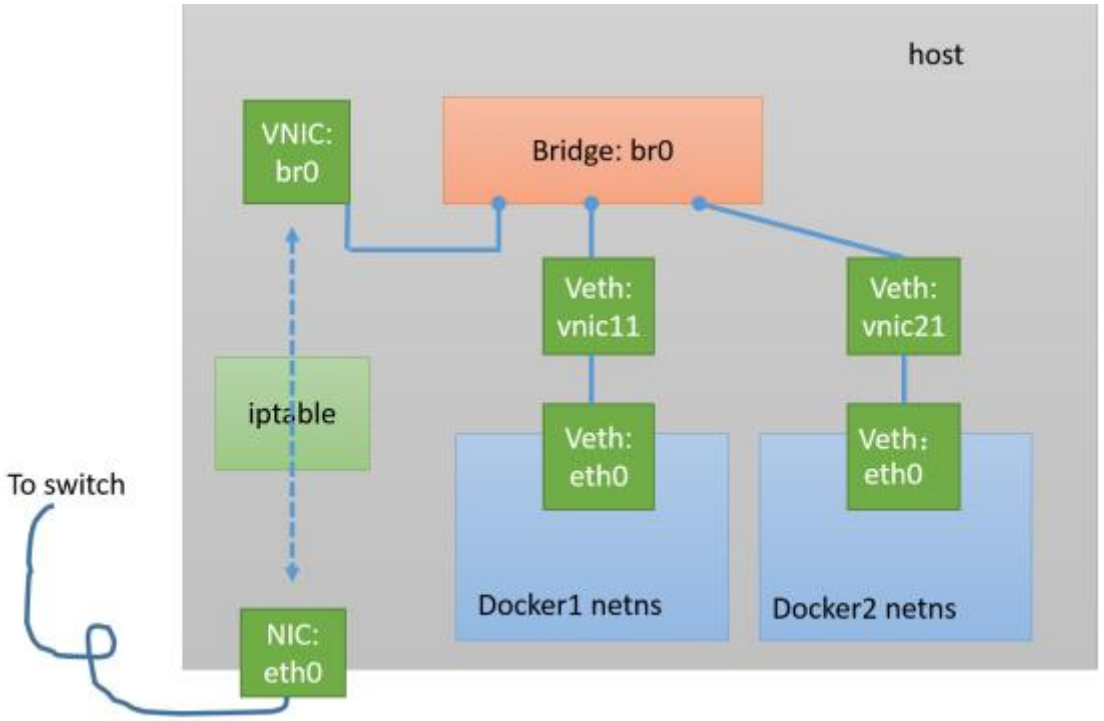
主要隔离了安全相关的标识符（identifiers）和属性（attributes），包括用户ID、用户组ID、root目录、key（指密钥）以及特殊权限。说得通俗一点，一个普通用户的进程经过clone()建立的新进程在新user namespace中能够拥有不一样的用户和用户组。这意味着一个进程在容器外属于一个没有特权的普通用户，可是他建立的容器进程却属于拥有全部权限的超级用户，这个技术为容器提供了极大的自由。

```
int child_main(void* args) {
        printf("在子进程中!\n");
        cap_t caps;
        printf("eUID = %ld;  eGID = %ld;  ",
                        (long) geteuid(), (long) getegid());
        caps = cap_get_proc();
        printf("capabilities: %s\n", cap_to_text(caps, NULL));
        execv(child_args[0], child_args);
        return 1;
}
//[...]
int child_pid = clone(child_main, child_stack+STACK_SIZE, CLONE_NEWUSER | SIGCHLD, NULL);
//[...]
```

先查看主机当前用户的uid和guid

```
———— 终端操作————-
sun@ubuntu$  id -u
sun@ubuntu$ 1000
sun@ubuntu$  id -g
sun@ubuntu$ 1000
sun@ubuntu$ gcc userns.c -Wall -lcap -o userns.o && ./userns.o
程序开始:
在子进程中!
eUID = 65534;  eGID = 65534;  capabilities: = cap_chown,cap_dac_override,[...]37+ep  <<--此处省略部分输出，已拥有所有权限
nobody@ubuntu$
```

Linux Namespace-应用场景



Docker使用linux的bridge和veth pair虚拟网络设备，以及network namespace对网络进行虚拟化。

Linux Bridge（网桥）是用纯软件实现的虚拟交换机，有着和物理交换机相同的功能，例如二层交换，MAC地址学习等。因此我们可以把tun/tap，veth pair等设备绑定到网桥上，就像是把设备连接到物理交换机上一样。此外它和veth pair、tun/tap一样，也是一种虚拟网络设备，具有虚拟设备的所有特性，例如配置IP，MAC地址等。

veth pair是成对出现的一种虚拟网络设备接口，一端连着网络协议栈，一端彼此相连。由于它的这个特性，常常被用于构建虚拟网络拓扑。例如连接两个不同的网络命名空间(netns)，连接docker容器，连接网桥(Bridge)等。

netns 可以创建一个完全隔离的新网络环境，这个环境包括一个独立的网卡空间，路由表，ARP表，ip地址表，iptables等。总之，与网络有关的组件都是独立的。

Linux Namespace API

kaihong 开鸿

一 通过clone()创建新进程的同时创建namespace是最常见做法

调用方式如下:

int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);

通过flags来控制使用多少功能。一共有二十多种CLONE_*的flag（标志位）参数用来控制clone进程的方方面面（如是否与父进程共享虚拟内存等等），下面外面逐一讲解clone函数传入的参数。

child_func: 传入子进程运行的程序主函数。

child_stack:传入子进程使用的栈空间

flags:表示使用哪些CLONE_*标志位

args:则可用于传入用户参数

CLONE_NEWIPC、CLONE_NEWNS、CLONE_NEWNET、CLONE_NEWPID、CLONE_NEWUSER和CLONE_NEWUTS

通常需要指定flags为以上六个常数的一个或多个，通过|（位或）操作来实现

Linux Namespace API


```
//[...]int child_main(void* arg)
 {
    printf("在子进程中!\n");
    sethostname("Changed Namespace", 12);
    execv(child_args[0], child_args);
    return 1;
}

int main()
 {
//[...]
int child_pid = clone(child_main, child_stack+STACK_SIZE,    CLONE_NEWUTS | SIGCHLD, NULL);
//[...]
}
```

Linux Namespace API

kaihong 开鸿

二 int setns(int fd, int nstype);

函数作用：setns() 将调用的进程与特定类型 namespace 的一个实例分离，并将该进程与该类型 namespace 的另一个实例重新关联。
fd ： 表示要加入的 namespace 的文件描述符，可以通过打开其中一个符号链接来获取，也可以通过打开 bind mount 到其中一个链接的文件来获取。
nstype：表示让调用者可以去检查 fd 指向的 namespace 类型，值可以设置为前文提到的常量 CLONE_NEW*，填 0 表示不检查。如果调用者已经明确知道自己要加入了 namespace 类型，或者不关心 namespace 类型，就可以使用该参数来自动校验。

```
main(int argc, char *argv[])
{
    int fd;

    if (argc < 3) {
        fprintf(stderr, "%s /proc/PID/ns/FILE cmd [arg...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);   /* 获取想要加入的 namespace 的文件描述符 */
    if (fd == -1)
        errExit("open");

    if (setns(fd, 0) == -1)         /* 加入该 namespace */
        errExit("setns");

    execvp(argv[2], &argv[2]);      /* 在加入的 namespace 中执行相应的命令 */
    errExit("execvp");
}
```

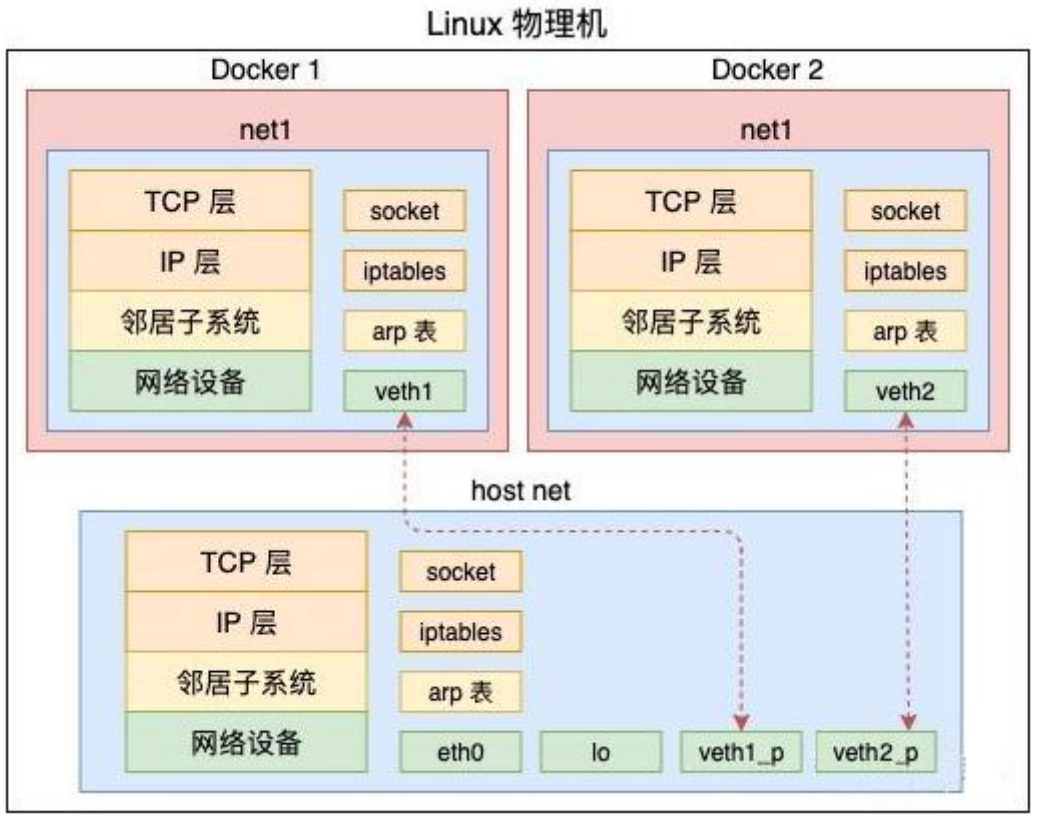
三 通过unshare()在原先进程上进行namespace隔离

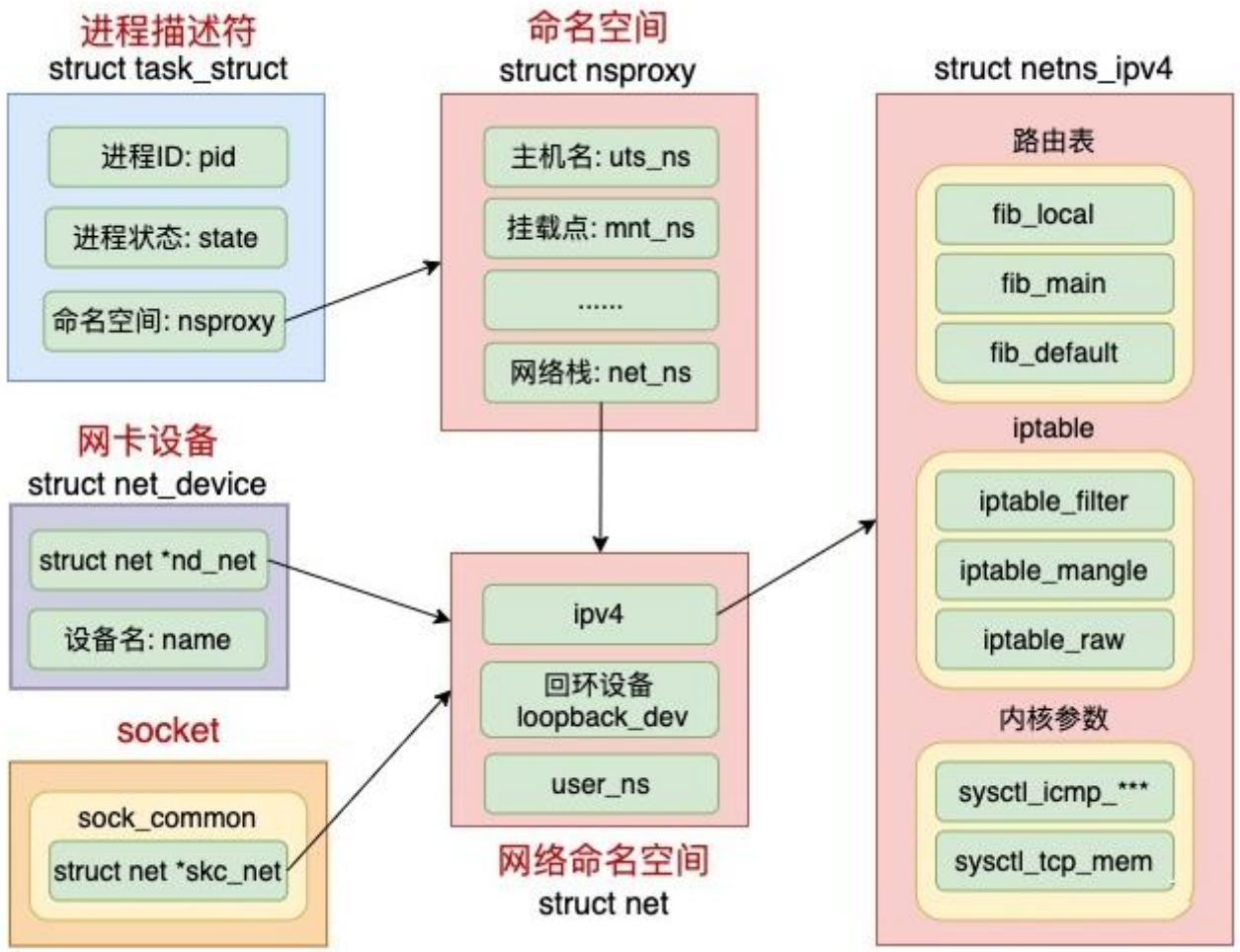
int unshare(int flags);

调用unshare()的主要作用就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。Linux中自带的unshare命令，就是通过unshare()系统调用实现的

```
if(0 == fork())
{
    unshare(CLONE_NEWXXXX); //返回 0为子进程代码，这个时候调用unshare命令，跳出原来父进程的命令空间
}
```

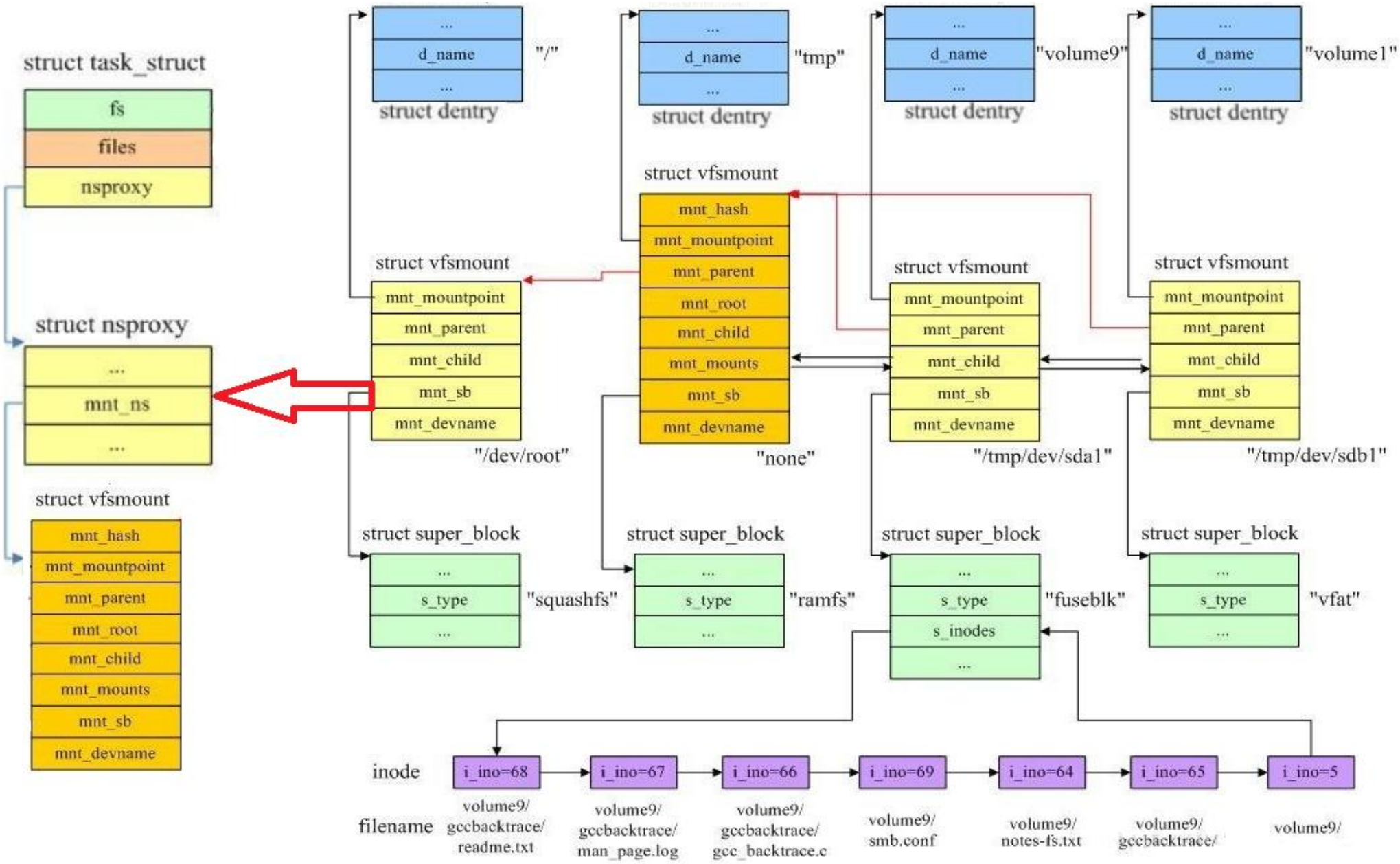
Linux Namespace 网络命名空间数据结构图





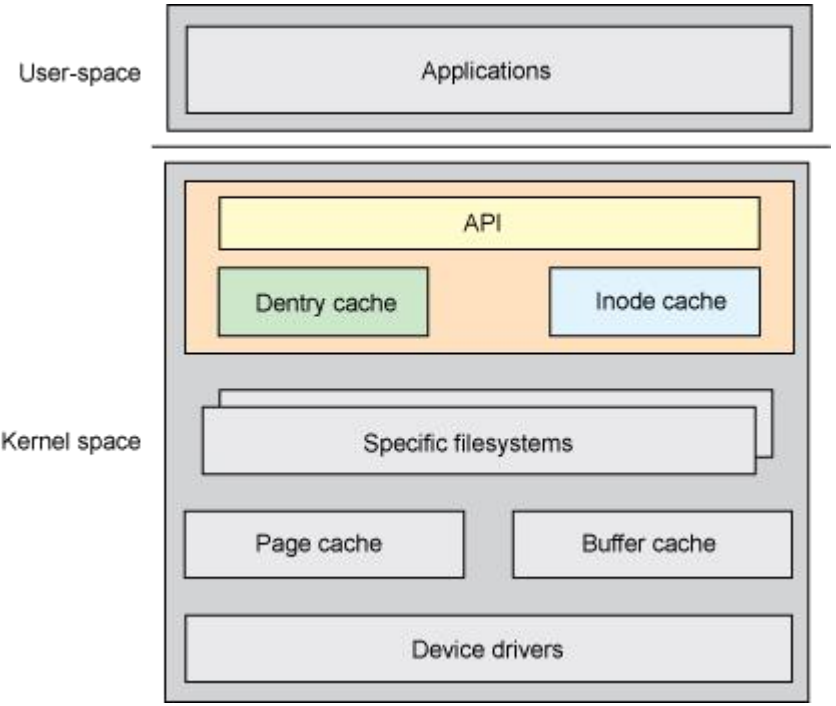
Linux Namespace 挂载点命名空间数据结构图

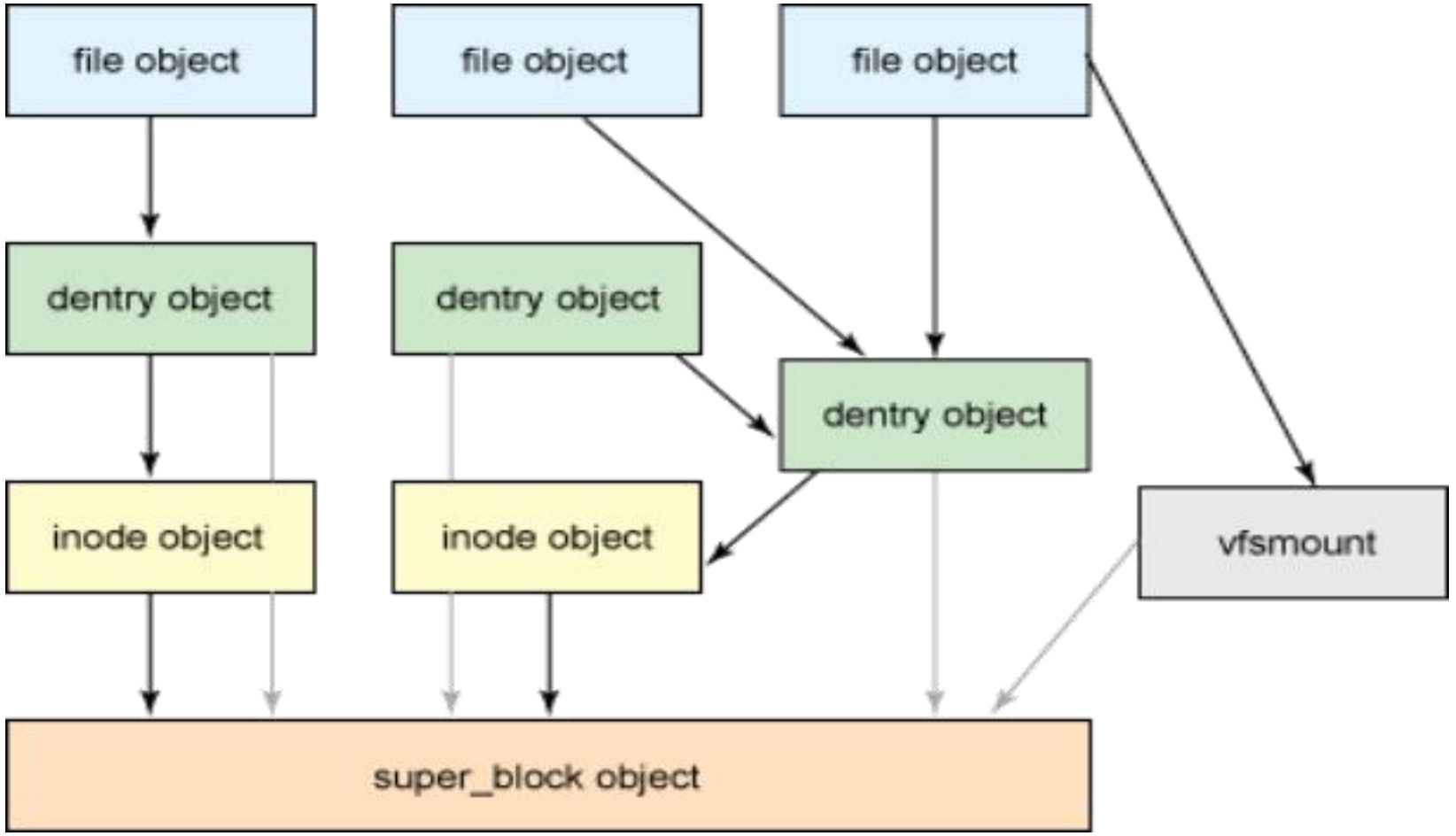
kaihong 开鸿



Linux Namespace 挂载点命名空间-VFS层次和对象

kaihong 开鸿

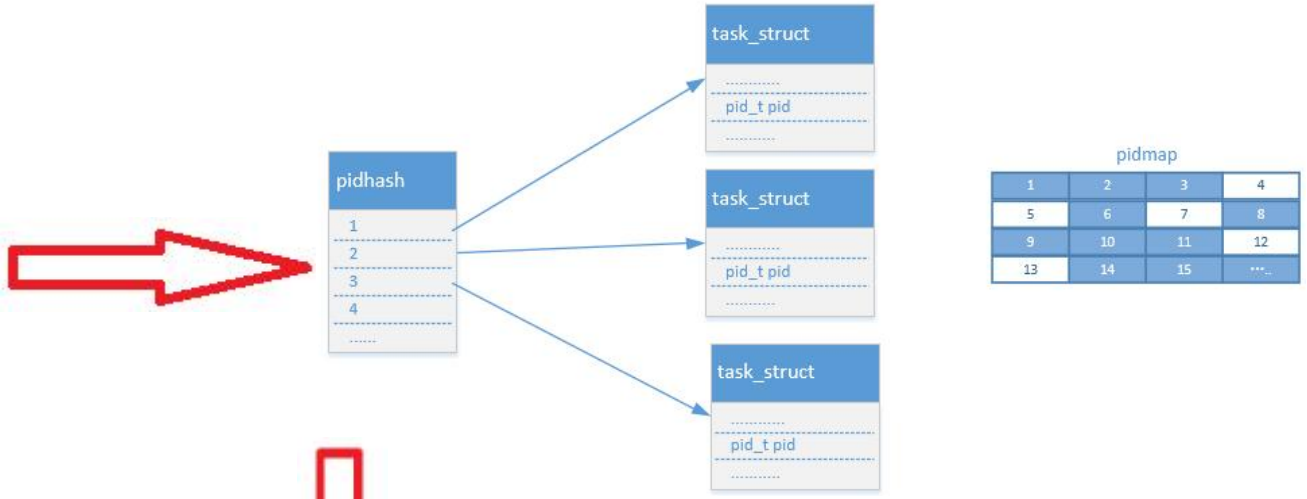


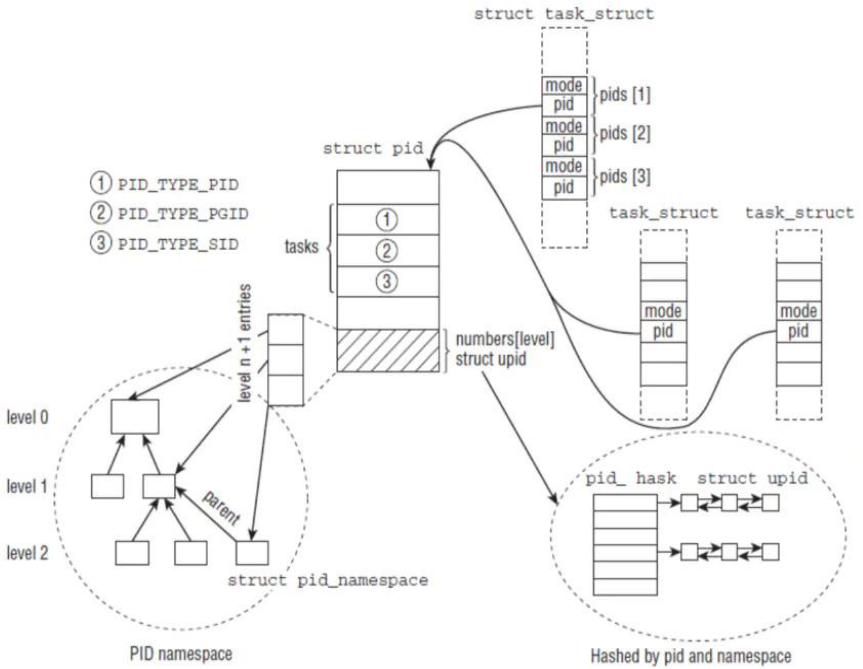


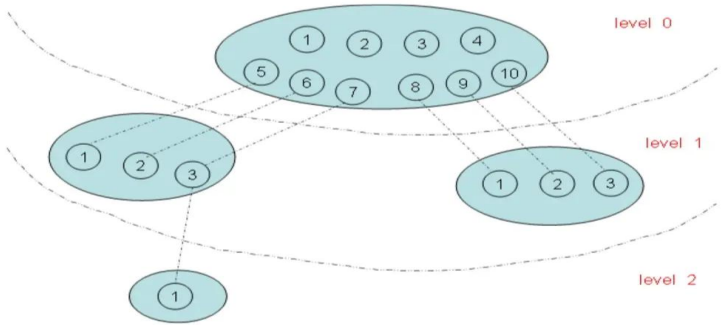
Linux Namespace PID命名空间数据结构图

kaihong 开鸿

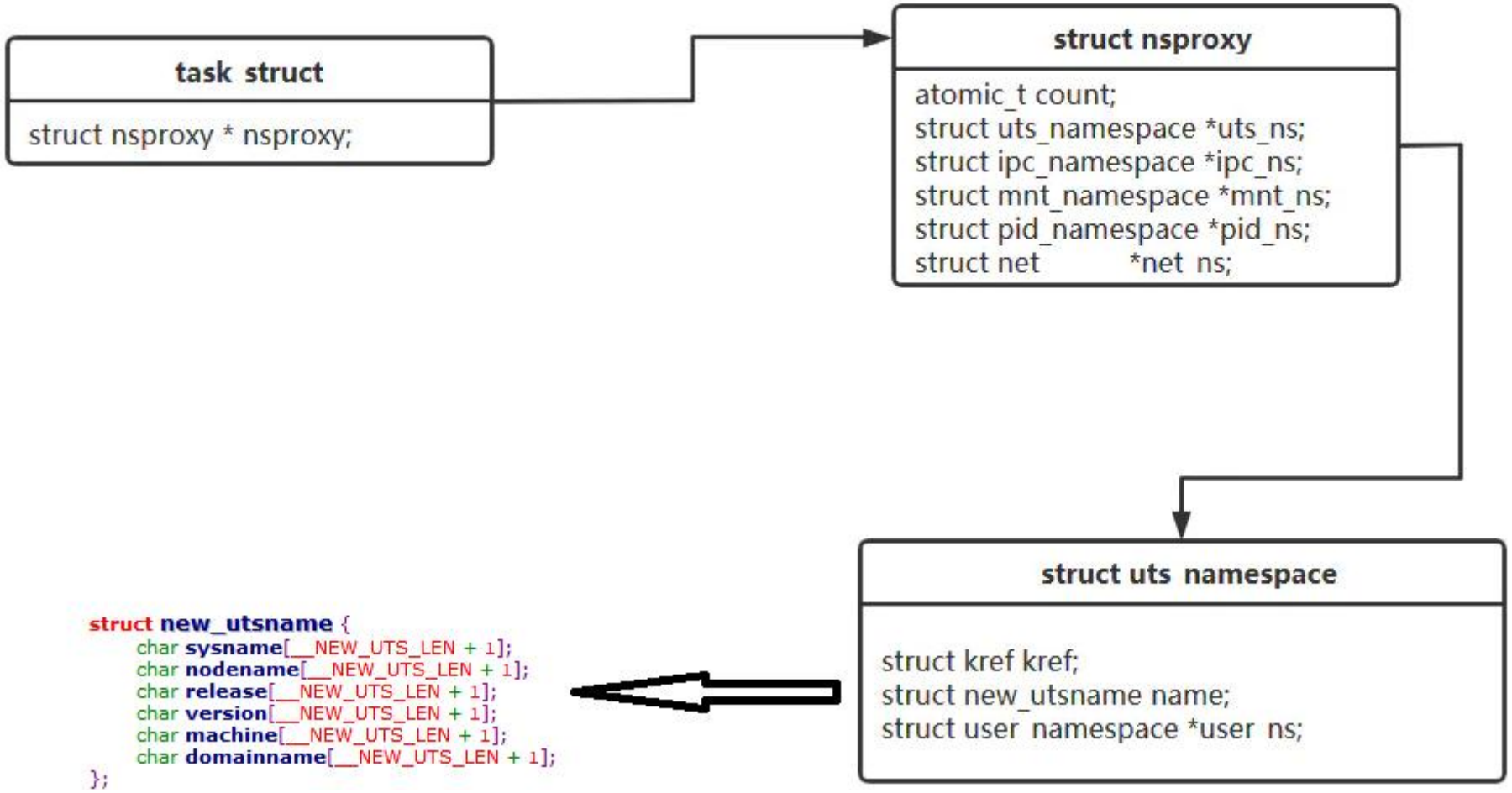
```
1  struct task_struct
2  {
3      .....
4      pid_t pid;
5      .....
6  }
```





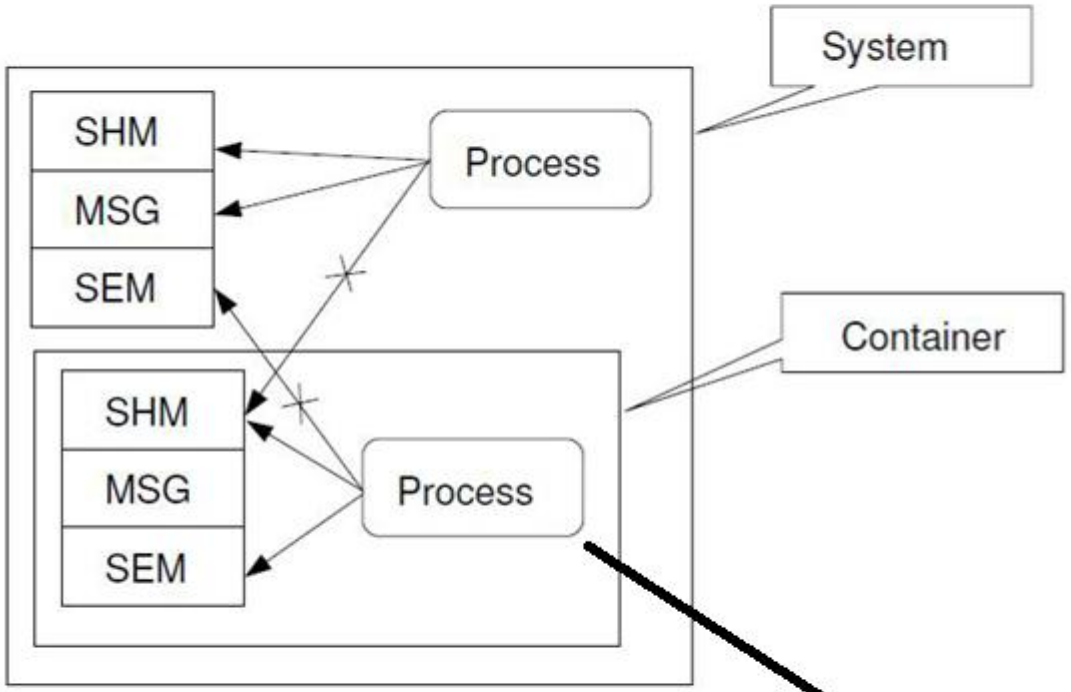


Linux Namespace UTS命名空间数据结构图



Linux Namespace IPC命名空间数据结构图

kaihong 开鸿




```
struct ipc_namespace {
    atomic_t count;
    struct ipc_ids  ids[3];

    int     sem_ctls[4];
    int     used_sems;

    int     msg_ctlmax;
    int     msg_ctlmnb;
    int     msg_ctlmni;
    atomic_t msg_bytes;
    atomic_t msg_hdrs;
    int     auto_msgmni;

    size_t  shm_ctlmax;
    size_t  shm_ctlall;
    int     shm_ctlmni;
    int     shm_tot;

    struct notifier_block ipcns_nb;

    /* The kern_mount of the mqueuefs sb.  We take a ref on it */
    struct vfsmount *mq_mnt;

    /* # queues in this ns, protected by mq_lock */
    unsigned int mq_queues_count;

    /* next fields are set through sysctl */
    unsigned int mq_queues_max;   /* initialized to DFLT_QUEUESMAX */
    unsigned int mq_msg_max;      /* initialized to DFLT_MSGMAX */
    unsigned int mq_msgsize_max;  /* initialized to DFLT_MSGSIZEMAX */

} ? end ipc_namespace ? ;
```

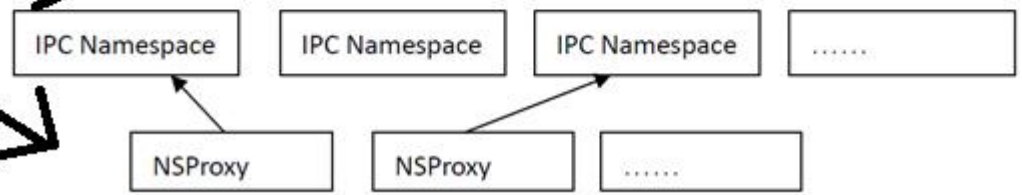
MSG，SHM，SEM结构

msg管理结构

SHM管理结构

SEM管理结构

mqueue管理结构



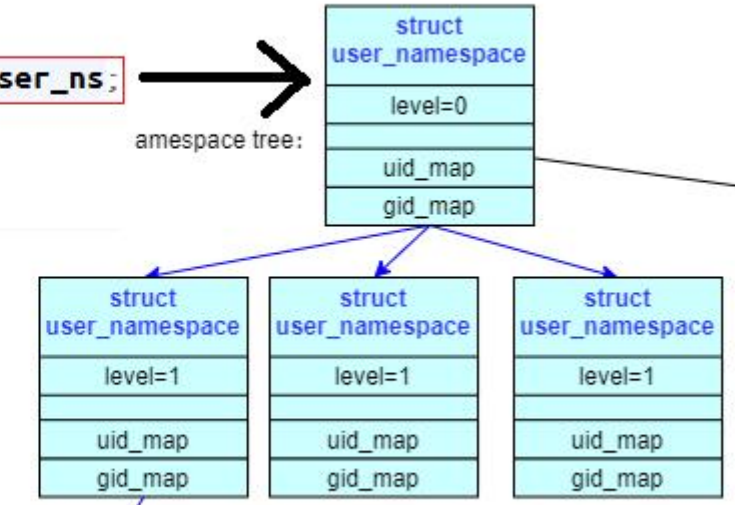
Linux Namespace User命名空间数据结构图

kaihong 开鸿

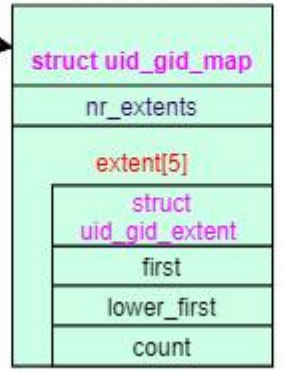
```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;
        struct user_namespace *user_ns;
        struct ucounts *ucounts;
        struct ns_common ns;
} __randomize_layout;
```

进程描述符
struct task_struct

amespace tree:



方式1 BASE_EXTENTS：
最多存储5个条目，存储在map->extent[]数组中。

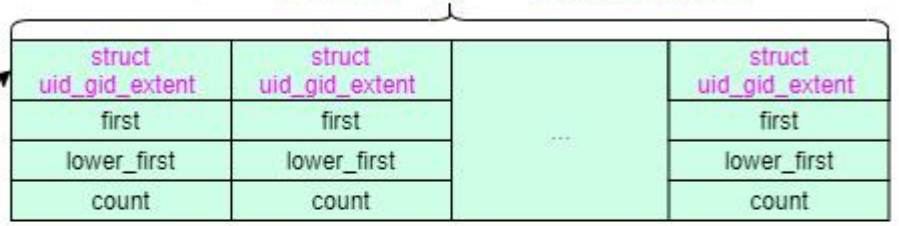


struct uid_gid_extent保存的是uid映射关系，成员说明如下：

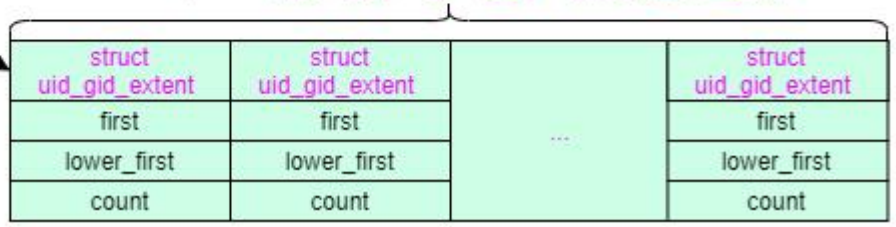
first：存储的是本user ns的uid。
lower_first：存储的是全局uid。
(在/proc/$$/uid_map配置时，本字段填的是父ns uid，内核最终转换成全局uid)
count：存储的是映射的个数。

方式2 MAX_EXTENTS：
最多存储340个条目，存储在map->forward[]和map->reverse[]中，需要额外的分配空间。

340个extent成员，使用first(子ns uid)作为关键字进行排序



340个extent成员，使用low_first(全局 uid)作为关键字进行排序



03

Cgroup内核实现

Linux Cgroup

Linux CGroup全称Linux Control Group， 是Linux内核的一个功能，用来限制，控制与分离一个进程组群的资源（如CPU、内存、磁盘输入输出等）。这个项目最早是由Google的工程师在2006年发起（主要是Paul Menage和Rohit Seth），最早的名称为进程容器（process containers）。在2007年时，因为在Linux内核中，容器（container）这个名词太过广泛，为避免混乱，被重命名为cgroup，并且被合并到2.6.24版的内核中去；

Cgroups提供了以下功能：

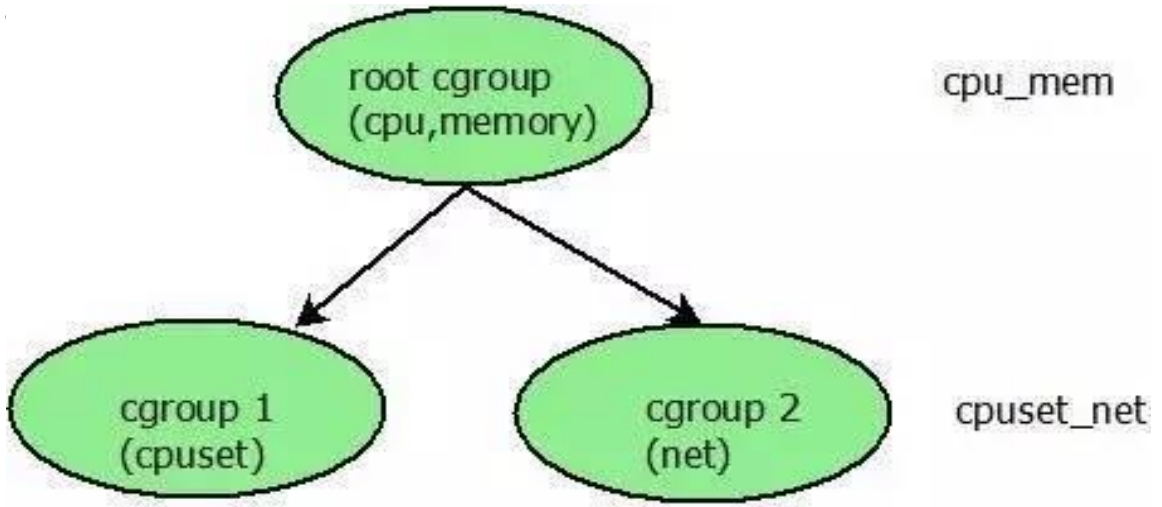
- 限制进程组可以使用的资源（Resource limiting ）：比如memory子系统可以为进程组设定一个memory使用上限，进程组使用的内存达到限额再申请内存，就会出发OOM（out of memory）；
- 进程组的优先级控制（Prioritization ）：比如可以使用cpu子系统为某个进程组分配cpu share；
- 记录进程组使用的资源量（Accounting ）：比如使用cpuacct子系统记录某个进程组使用的cpu时间；
- 进程组控制（Control）：比如使用freezer子系统可以将进程组挂起和恢复；

Cgroup子系统：

- blkio -- 这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等等）。
- cpu -- 这个子系统使用调度程序提供对 CPU 的 cgroup 任务访问。
- cpuacct -- 这个子系统自动生成 cgroup 中任务所使用的 CPU 报告。
- cpuset -- 这个子系统为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。
- devices -- 这个子系统可允许或者拒绝 cgroup 中的任务访问设备。
- freezer -- 这个子系统挂起或者恢复 cgroup 中的任务。
- memory -- 这个子系统设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。
- net_cls -- 这个子系统使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（tc）识别从具体 cgroup 中生成的数据包。

Cgroup专业术语：

- task：任务就是系统的一个进程
- control group：控制族群就是按照某种标准划分的进程。Cgroups 中的资源控制都是以控制族群为单位实现。一个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。一个进程组的进程可以使用 cgroups 以控制族群为单位分配的资源，同时受到 cgroups 以控制族群为单位设定的限制；
- 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控制族群是父节点控制族群的孩子，继承父控制族群的特定的属性；
- subsystem：一个子系统就是一个资源控制器，比如 cpu 子系统就是控制 cpu 时间分配的一个控制器。子系统必须附加（attach）到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制



Linux Cgroup场景-绑定进程到指定cpu核数

kaihong 开鸿

➢ 一 创建策略

```
#创建设置small组策略 4核
cgcreate -g cpuset:small
cgset -r cpuset.cpus=0-3 small
cgset -r cpuset.mems=0 small

#创建设置large组策略 8核
cgcreate -g cpuset:large
cgset -r cpuset.cpus=0-7 large
cgset -r cpuset.mems=0 large
```

二 编写一个消耗cpu的脚本

```
vi t1.sh
#/bin/bash
x=0
while [ True ];do
    x=$x+1
done;
```

三限制程序运行在固定cpu核数上

```
cgexec -g cpuset:small sh t1.sh &
```

Linux Cgroup场景-限制内存使用量

kaihong 开鸿

- 创建控制群组g2

#cgcreate -g memory:g2

- 查看默认内存是没有限制的

cgget -r memory.limit_in_bytes g2

- 限制内存只有1GB

cgset -r memory.limit_in_bytes=1073741824 g2

- 执行/tmp/highmemory.sh，进程号是21127

#vi /tmp/highmem.sh

```
#/bin/bash
x="a"
while [ True ];do
    x=$x$x
done;
```

- 将highmemory.sh进程加入g2的控制

将highmemory.sh进程加入g2的控制

#cgclassify -g memory:g2 21127

Linux Cgroup场景-限制磁盘读取速度

kaihong 开鸿

- 创建策略

要控制/dev/sdb的磁盘，通过下述命令查到磁盘驱动号8,16

```
#ls -l /dev/sdb
#创建控制组，设定8,16磁盘有1MB的读取速度限制
#cgcreate -g blkio:g1
#cgset -r blkio.throttle.read_bps_device='8:16 10485760' g1
```

- 启动读取测试命令，拿到pid 14468

```
#dd if=/dev/sdb of=/dev/null
```

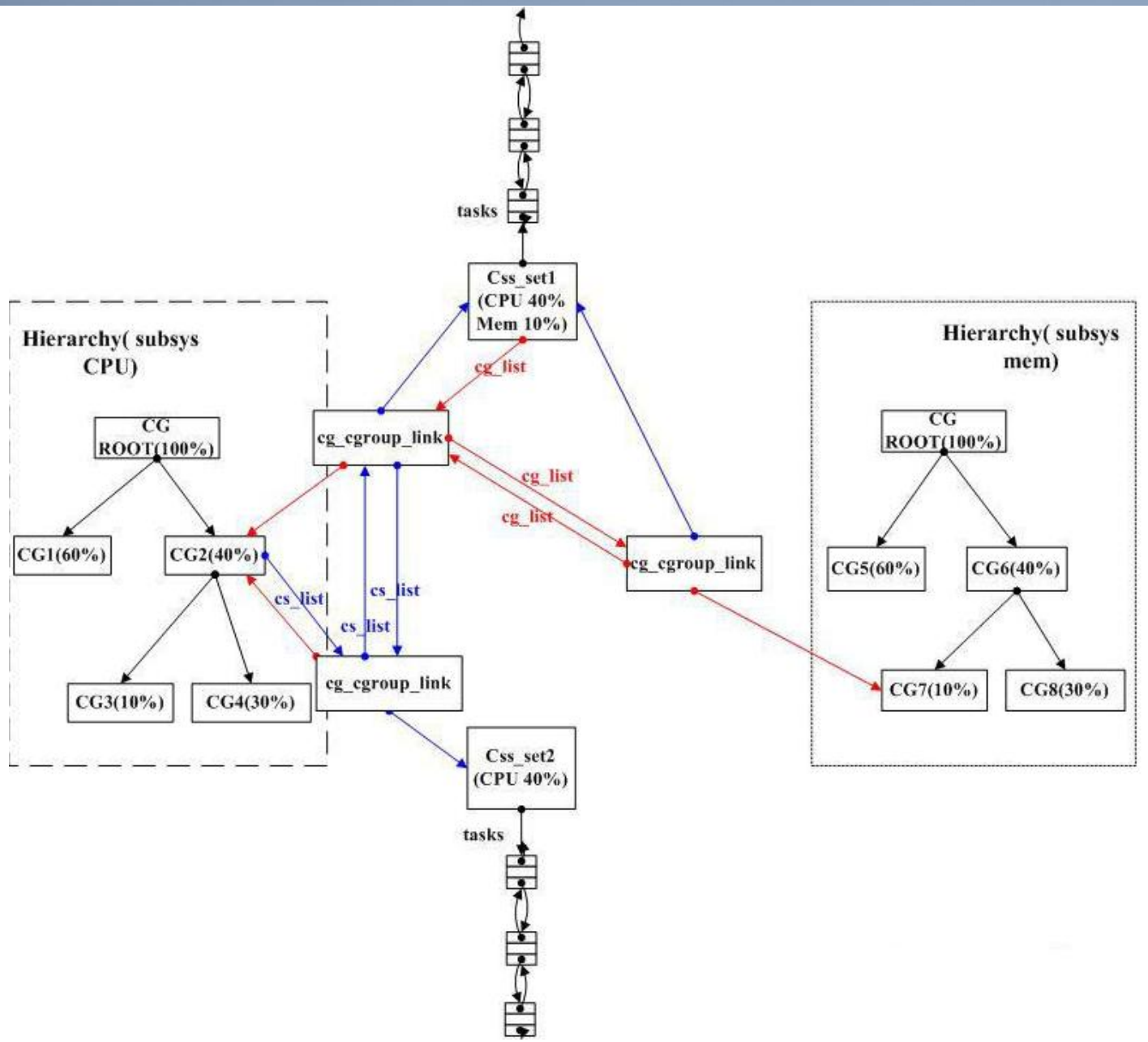
- 通过命令可以看到对磁盘读写速度的消耗

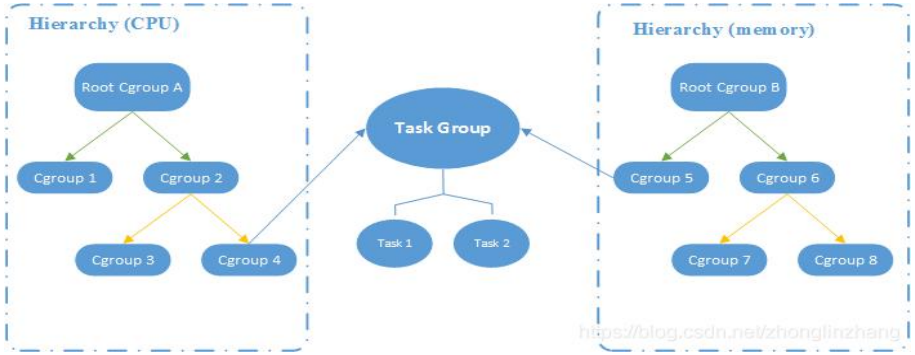
```
#iotop
```

- 将进程加入g1控制组后，读取速度被限制

```
#cgclassify -g blkio:g1 14468
```

Linux Cgroup原理

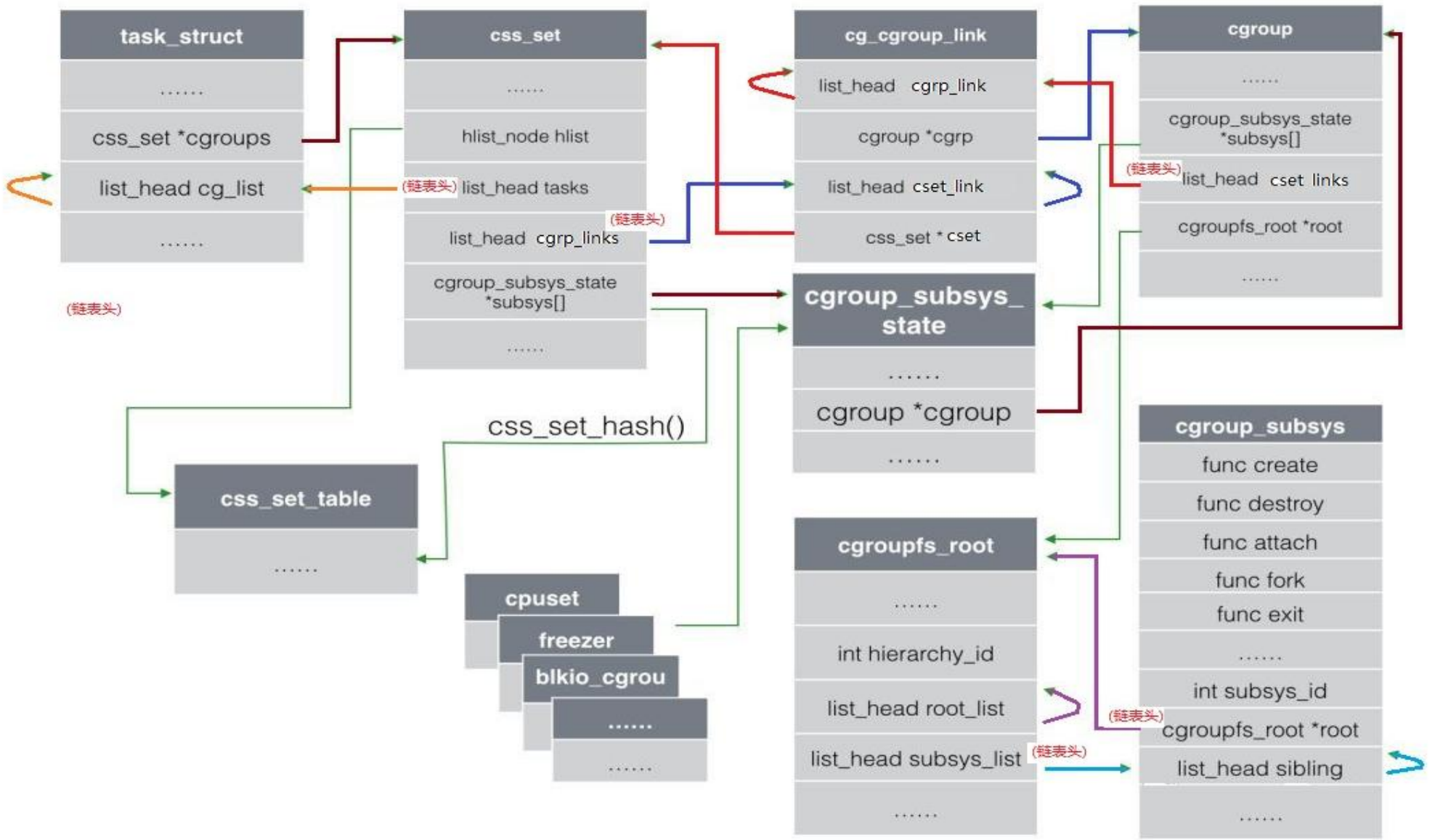




从实现角度来看，cgroup实现了一个通用的进程分组框架，不同资源的具体管理工作由各cgroup子系统来实现，当需要多个限制策略比如同时针对cpu和内存进行限制，则同时关联多个cgroup子系统即可。

- 每次在系统中创建新层级时，该系统中的所有任务都是那个层级的默认 cgroup的初始成员（我们称之为 root cgroup，此 cgroup 在创建层级时自动创建，后面在该层级中创建的 cgroup 都是此 cgroup 的后代）；
- 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，但是这些 cgroup 必须在不同的层级；
- 系统中的进程（任务）创建子进程（任务）时，该子任务自动成为其父进程所在 cgroup 的成员。然后可根据需要将该子任务移动到不同的 cgroup 中，但开始时它总是继承其父任务的 cgroup。

Linux Cgroup原理-数据结构



kaihong 开鸿

THANKS

深圳开鸿数字产业发展有限公司 | kaihongdigi.com

深开鸿公众号