**1.namespace：**

  Linux Namespaces机制提供一种资源隔离方案。PID,IPC,Network等系统资源不再是全局性的，而是属于特定的Namespace。每个Namespace里面的资源对其他Namespace都是不可见的，要创建新的Namespace，只需要在调用clone时指定相应的flag。LinuxNamespaces机制为实现基于容器的虚拟化技术提供了很好的基础，容器正是利用这一特性实现了资源的隔离。不同container内的进程属于不同的Namespace，彼此透明，互不干扰。

  Linux很早就实现了一个系统调用chroot，该系统调用能够为进程提供一个限制的文件系统。chroot提供了一种简单的隔离模式：chroot内部的文件系统无法访问外部的内容。Linux Namespace在此基础上，提供了对UTS、IPC、mount、PID、network的隔离机制。

  UTS: 包含了运行内核的名称，版本，底层体系结构的信息

  IPC: 包含了所有与进程间通信有关的信息

  PID: 就是进程ID

  mount: 包含了文件系统的视图

  NET: 网络访问

**1.1 task\_struct中的结构：**  
struct task\_struct {   
...   
struct nsproxy \*nsproxy;   
...   
};

<——————将给定进程关联到所属的各个命名空间——————>  
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;   
struct net \*net\_ns;   
};   
  
**1.2 创建命名空间的方式**

  1.2.1. clone创建新进程时，可以设置选项，使新进程与父进程共享命名空间，还是新进程创建一个独立的命名空间.

  1.2.2. unshare系统调用，可以将进程的某些部分从父进程分离，其中也包括命名空间。

**1.3 PID**

PID命名空间按层次组织，在创建一个新的pid namespace，该命名空间中所有的pid都对父命名空间可见，但是子命名空间

看不到父命名空间的pid，因此进程在不同的pid namespace中具有不同的pid，只要能看到该进程的namespace都有一个PID。

  对于所有的进程来说，都有两种ID：一个是全局的ID（包含PID、TGID、PGRP、SID），保存在task\_struct->pid中；

另一个是局部的ID，即属于某个特定的命名空间的ID,对应task\_struct->pids数组，可以通过task\_struct->pids[pid\_type]->pid

来找到对应的pid结构。pid\_type：PIDTYPE\_PID，PIDTYPE\_PGID，PIDTYPE\_SID，PIDTYPE\_MAX

struct **pid**   
{   
atomic\_t count; //计数  
unsigned int level; //对应多少namespace  
struct hlist\_head tasks[PIDTYPE\_MAX]; //指回task\_struct  
struct rcu\_head rcu; //rcu是将所有struct pid组织起来的辅助结构  
struct upid numbers[1]; //numbers成员中存储的是struct upid结构，该结构是pid与pid\_namespace相关联的结构。  
}

**upid**  
struct**upid** {   
int nr;   
struct pid\_namespace \*ns;   
struct hlist\_node pid\_chain;   
};

所有的upid都保存在一个散列表中，通过upid->pid\_chain组织。

static struct hlist\_head \*pidhash;  
};  
**pid\_namespace**  
struct **pid\_namespace** {   
struct kref kref;   
struct pidmap pidmap[PIDMAP\_ENTRIES]; //保存该namespace中pid的分配情况  
int last\_pid; //保存上一个分配的pid  
struct task\_struct \*child\_reaper; //每个namespace都有一个进程来扮演Linux中init进程的角色,child\_reaper指向这个进程  
struct kmem\_cache \*pid\_cachep;   
unsigned int level; //表示该namespace在整个命名空间的层次  
struct pid\_namespace \*parent; //父namespace  
struct vfsmount \*proc\_mnt;   
struct bsd\_acct\_struct \*bacct;   
};

**2.cgroups：**限制被namespaces隔离起来的资源，为资源设置权重，计算使用量，操控任务启停。

**特点：**cgroups通过伪文件系统方式实现

             组织管理操作单元细粒度到线程级别，用户也可以创建销cgroup实现资源再分配

            资源管理的功能都已子系统方式实现，接口统一

            子任务创建之初与副任务同出一个cgroups

**作用：**资源限制：对任务使用的资源总额进行限制

             优先级分配：通过分配CPU时间片，IO带宽等来控制任务优先级

             资源统计：CPU使用时长，内存用量等

             任务控制：任务挂起，恢复等

**相互关系：**  cgroups具有层级结构，每个层级通过绑定对应的子系统进行资源控制，cgoups层级可以包含0或1个子节点。子节点继承

父节点挂载的子系统。

  1.一个子系统能附加到多个层级，前提是目标层级只有唯一一个子系统

  2.一个层级可以附加多个子系统

  3.一个任务可以是多个cgroup的成员，但是这些cgroup必须在不同的层级。

  4.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。

**Cgroups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。

     ns -- 名称空间子系统。

**cgroups数据结构：**

task\_struct中与cgroups有关的：

    struct css\_set \*cgroups;

    struct list\_head cg\_list;

其中cgroups指针指向了一个css\_set结构，而css\_set存储了与进程相关的cgroups信息。cg\_list是一个list\_head结构，用于将连到同一个

css\_set的进程组织成一个链表。

**struct** css\_set {

    atomic\_t refcount; //该css\_set的引用数

    struct hlist\_node hlist; //用于把所有css\_set组织成一个hash表，这样内核可以快速查找特定的css\_set

    struct list\_head tasks; //指向所有连到此css\_set的进程连成的链表

    struct list\_head cg\_links; //指向一个由struct cg\_cgroup\_link连成的链表

    struct cgroup\_subsys\_state \*subsys[CGROUP\_SUBSYS\_COUNT]; //subsys是一个数组，存储一组指向cgroup\_subsys\_state的指针。

一个cgroup\_subsys\_state就是进程与一个特定子系统相关的信息。通过这个指针数组，进程就可以获得相应的cgroups控制信息了

};

**struct** cgroup\_subsys\_state {

    struct cgroup \*cgroup;

    atomic\_t refcnt;

    unsigned long flags;

    struct css\_id \*id;

};

    cgroup指针指向了一个cgroup结构，也就是进程属于的cgroup。进程受到子系统的控制，实际上是通过加入到特定的cgroup实现的，因为cgroup

在特定的层级上，而子系统又是附加到层级上的。通过以上三个结构，进程就可以和cgroup连接起来了：task\_struct->css\_set->cgroup\_subsys\_state->cgroup。

    cgroup和css\_set是一个多对多的关系，一个进程对应一个css\_set，一个css\_set就存储了一组进程（有可能被几个进程共享）跟各个子系统相关的信息，

进程可以同时属于几个cgroup，只要这些cgroup不在同一个层级。一个cgroup中可以有多个进程，而且这些进程的css\_set不一定都相同，因为有些进程

可能还加入了其他cgroup。

struct cg\_cgroup\_link {

    struct list\_head cgrp\_link\_list;

    struct cgroup \*cgrp;

    struct list\_head cg\_link\_list;

    struct css\_set \*cg;

}；

   cg\_cgroup\_link作为一个中间结构将 cgroup和css\_set联系起来，cgrp\_link\_list和cg\_link\_list分别指向cgroup和css\_set所在的链表。每个进程都会指向一个css\_set，与这个css\_set关联的所有进程都会链入到css\_set->tasks链表，cgroup通过中间结构cg\_cgroup\_link来寻找所有与之关联的所有css\_set，从而可以得到与cgroup关联的所有进程。

  mount -t cgroup 查看当前系统所有根层级，进入到跟层级目录下，mkdir [名称]就可以创建一个cgroup，新创建的cgroup下的tasks文件为空的，表示当前cgroup无进程，而根级目录下的tasks文件内包含当前系统所有进程。