01-Cgroup详解

linux cgroup 简介

Linux cgroups 的全称是 Linux Control Groups，它是 Linux 内核的特性，主要作用是**限制、记录和隔离进程组（process groups）使用的物理资源（cpu、memory、IO 等）**。

2006 的时候，Google 的一些工程师（主要是 Paul Menage 和 Rohit Seth）启动了这个项目，最初的名字叫 **process containers**。因为 **container** 在内核中名字有歧义，2007 的时候改名为 **control groups**，并合并到 2008 年发布的 2.6.24 内核版本。

最初 cgroups 的版本被称为 v1，这个版本的 cgroups 设计并不友好，理解起来非常困难。后续的开发工作由 Tejun Heo 接管，他重新设计并重写了 cgroups，新版本被称为 v2，并首次出现在 kernel 4.5 版本。

cgroups 从设计之初使命就很明确，为进程提供资源控制，它主要的功能包括：

- **资源限制**：限制进程使用的资源上限，比如最大内存、文件系统缓存使用限制
- **优先级控制**：不同的组可以有不同的优先级，比如 CPU 使用和磁盘 IO 吞吐
- **审计**：计算 group 的资源使用情况，可以用来计费
- **控制**：挂起一组进程，或者重启一组进程

目前 cgroups 已经成为很多技术的基础，比如 LXC、**Docker**、systemd等。

NOTE：资源限制是这篇文章的重点，也是 docker 等容器技术的基础，接下来也将重点介绍下Cgroup的资源限制。

cgroups 核心概念

前面说过，cgroups 是用来对进程进行资源管理的，因此 cgroup 需要考虑如何抽象这两种概念：进程和资源，同时如何组织自己的结构。cgroups 中有几个非常重要的概念：

- **task**：任务，对应于系统中运行的一个实体，一般是指进程
- **subsystem**：子系统，具体的资源控制器（resource class 或者 resource controller），控制某个特定的资源使用。比如 CPU 子系统可以控制 CPU 时间，memory 子系统可以控制内存使用量
- **cgroup**：控制组，一组任务和子系统的关联关系，表示对这些任务进行怎样的资源管理策略
- **hierarchy**：层级树，一系列 cgroup 组成的树形结构。每个节点都是一个 cgroup，cgroup 可以有多个子节点，子节点默认会继承父节点的属性。系统中可以有多个 hierarchy

虽然 cgroup 支持 hierarchy，允许不同的子资源挂到不同的目录，但是多个树之间有各种限制，增加了理解和维护的复杂性。在实际使用中，所有的子资源都会统一放到某个路径下（比如 ubuntu16.04 的 `/sys/fs/cgroup/`），因此本文并不详细介绍多个树的情况，感兴趣的可以参考 RedHat 的这篇文档。

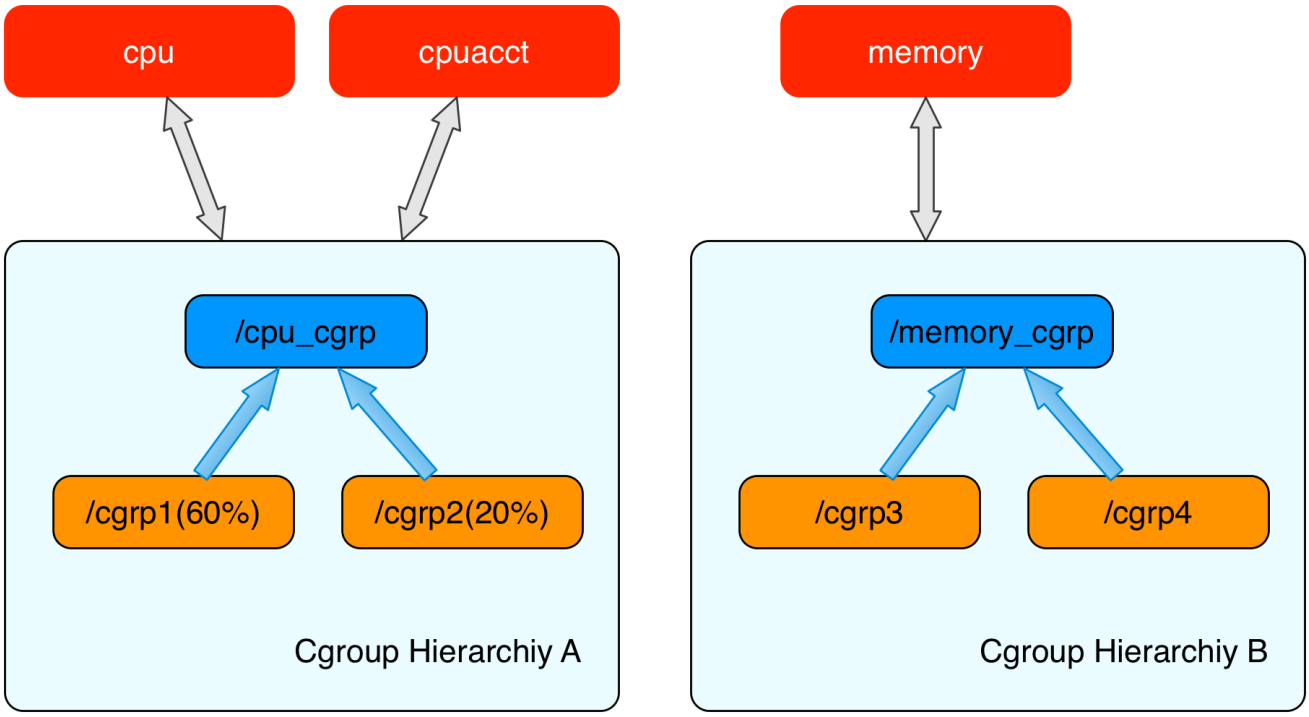
子资源系统（Resource Classes or SubSystem）

目前有下面这些资源子系统：

- Block IO（`blkio`）：限制块设备（磁盘、SSD、USB 等）的 IO 速率
- CPU Set(`cpuset`)：限制任务能运行在哪些 CPU 核上
- CPU Accounting(`cpuacct`)：生成 cgroup 中任务使用 CPU 的报告
- CPU (`CPU`)：限制调度器分配的 CPU 时间
- Devices (`devices`)：允许或者拒绝 cgroup 中任务对设备的访问
- Freezer (`freezer`)：挂起或者重启 cgroup 中的任务
- Memory (`memory`)：限制 cgroup 中任务使用内存的量，并生成任务当前内存的使用情况报告
- Network Classifier(`net_cls`)：为 cgroup 中的报文设置上特定的 classid 标志，这样 tc 等工具就能根据标记对网络进行配置
- Network Priority (`net_prio`)：对每个网络接口设置报文的优先级
- perf_event：识别任务的 cgroup 成员，可以用来做性能分析

cgroups 层级结构（Hierarchy）

内核使用 cgroup 结构体来表示一个 control group 对某一个或者某几个 cgroups 子系统的资源限制。cgroup 结构体可以组织成一颗树的形式，每一棵 cgroup 结构体组成的树称之为一个 cgroups 层级结构。cgroups层级结构可以 attach 一个或者几个 cgroups 子系统，当前层级结构可以对其 attach 的 cgroups 子系统进行资源的限制。每一个 cgroups 子系统只能被 attach 到一个 cpu 层级结构中。



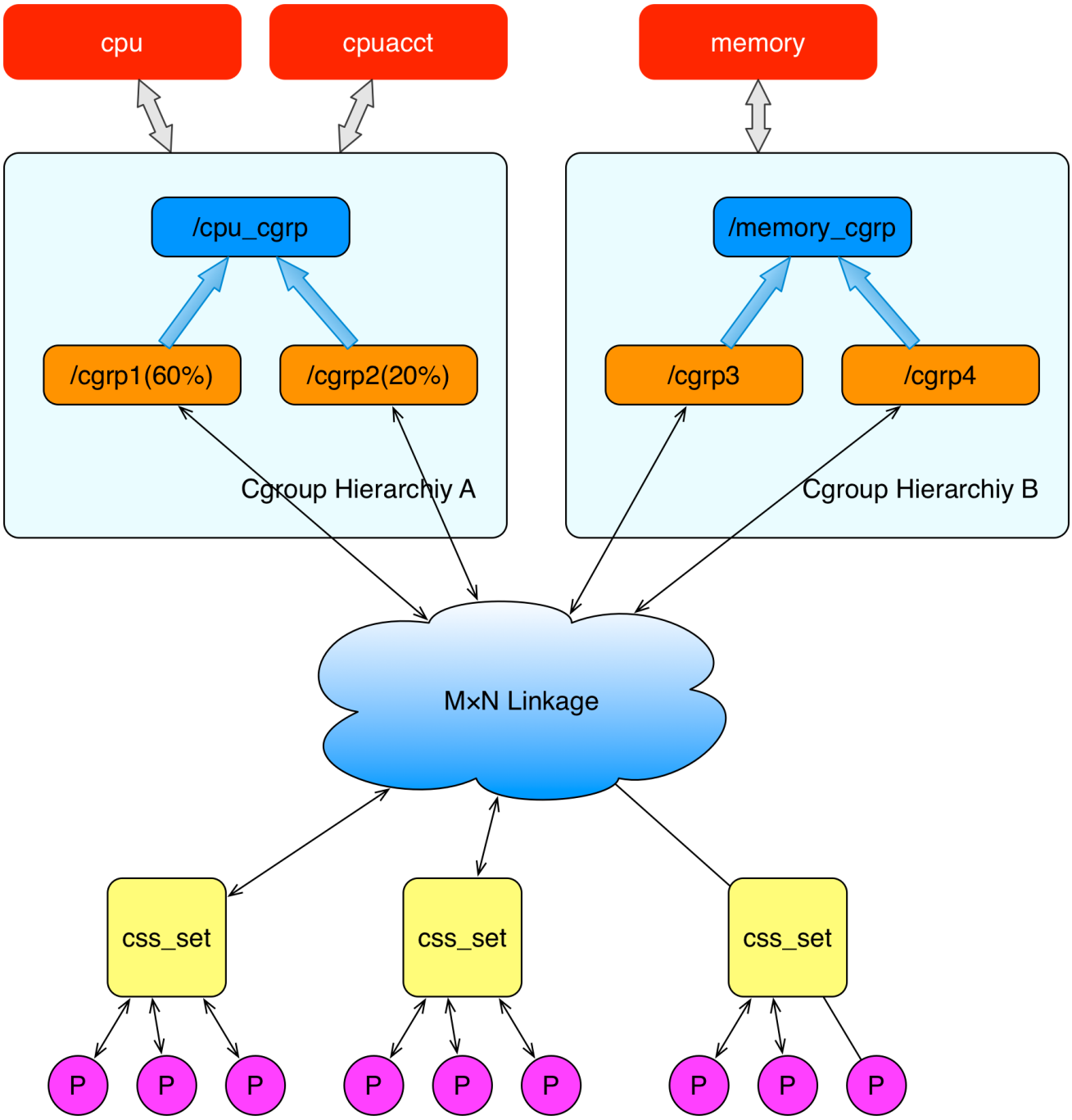
比如上图表示两个cgroups层级结构，每一个层级结构中是一颗树形结构，树的每一个节点是一个 cgroup 结构体（比如cpu_cgrp, memory_cgrp)。第一个 cgroups 层级结构 attach 了 cpu 子系统和 cpuacct 子系统， 当前 cgroups 层级结构中的 cgroup 结构体就可以对 cpu 的资源进行限制，并且对进程的 cpu 使用情况进行统计。 第二个 cgroups 层级结构 attach 了 memory 子系统，当前 cgroups 层级结构中的 cgroup 结构体就可以对 memory 的资源进行限制。

在每一个 cgroups 层级结构中，每一个节点（cgroup 结构体）可以设置对资源不同的限制权重。比如上图中 cgrp1 组中的进程可以使用60%的 cpu 时间片，而 cgrp2 组中的进程可以使用20%的 cpu 时间片。

cgroups与进程

上面提到了内核使用 cgroups 子系统对系统的资源进行限制，也提到了 cgroups 子系统需要 attach 到 cgroups 层级结构中来对进程进行资源控制，本小节重点关注一下内核是如何把进程与 cgroups 层级结构联系起来的。

在创建了 cgroups 层级结构中的节点（cgroup 结构体）之后，可以把进程加入到某一个节点的控制任务列表中，一个节点的控制列表中的所有进程都会受到当前节点的资源限制。同时某一个进程也可以被加入到不同的 cgroups 层级结构的节点中，因为不同的 cgroups 层级结构可以负责不同的系统资源。所以说进程和 cgroup 结构体是一个多对多的关系。



上面这个图从整体结构上描述了进程与 cgroups 之间的关系。最下面的P代表一个进程。每一个进程的描述符中有一个指针指向了一个辅助数据结构css_se（ cgroups subsystem set ）。 指向某一个css_set的进程会被加入到当前css_set的进程链表中。一个进程只能隶属于一个css_set，一个css_set可以包含多个进程，隶属于同一css_set的进程受到同一个css_set所关联的资源限制。

上图中的”M×N Linkage”说明的是css_set通过辅助数据结构可以与 cgroups 节点进行多对多的关联。但是 cgroups 的实现不允许css_set同时关联同一个cgroups层级结构下多个节点。 这是因为 cgroups 对同一种资源不允许有多个限制配置。

一个css_set关联多个 cgroups 层级结构的节点时，表明需要对当前css_set下的进程进行多种资源的控制。而一个 cgroups 节点关联多个css_set时，表明多个css_set下的进程列表受到同一份资源的相同限制。

cgroups文件系统

Linux 使用了多种数据结构在内核中实现了 cgroups 的配置，关联了进程和 cgroups 节点，那么 Linux 又是如何让用户态的进程使用到 cgroups 的功能呢？ Linux内核有一个很强大的模块叫 VFS (Virtual File System)。 VFS 能够把具体文件系统的细节隐藏起来，给用户态进程提供一个统一的文件系统 API 接口。 cgroups 也是通过 VFS 把功能暴露给用户态的，cgroups 与 VFS 之间的衔接部分称之为 cgroups 文件系统。下面先介绍一下 VFS 的基础知识，然后再介绍下 cgroups 文件系统的实现。

VFS

VFS 是一个内核抽象层，能够隐藏具体文件系统的实现细节，从而给用户态进程提供一套统一的 API 接口。VFS 使用了一种通用文件系统的设计，具体的文件系统只要实现了 VFS 的设计接口，就能够注册到 VFS 中，从而使内核可以读写这种文件系统。 这很像面向对象设计中的抽象类与子类之间的关系，抽象类负责对外接口的设计，子类负责具体的实现。其实，VFS本身就是用 c 语言实现的一套面向对象的接口。

通用文件模型

VFS 通用文件模型中包含以下四种元数据结构：

1. 超级块对象(superblock object)，用于存放已经注册的文件系统的信息。比如ext2，ext3等这些基础的磁盘文件系统，还有用于读写socket的socket文件系统，以及当前的用于读写cgroups配置信息的 cgroups 文件系统等。
2. 索引节点对象(inode object)，用于存放具体文件的信息。对于一般的磁盘文件系统而言，inode 节点中一般会存放文件在硬盘中的存储块等信息；对于socket文件系统，inode会存放socket的相关属性，而对于cgroups这样的特殊文件系统，inode会存放与 cgroup 节点相关的属性信息。这里面比较重要的一个部分是一个叫做 inode_operations 的结构体，这个结构体定义了在具体文件系统中创建文件，删除文件等的具体实现。
3. 文件对象(file object)，一个文件对象表示进程内打开的一个文件，文件对象是存放在进程的文件描述符表里面的。同样这个文件中比较重要的部分是一个叫 file_operations 的结构体，这个结构体描述了具体的文件系统的读写实现。当进程在某一个文件描述符上调用读写操作时，实际调用的是 file_operations 中定义的方法。 对于普通的磁盘文件系统，file_operations 中定义的就是普通的块设备读写操作；对于socket文件系统，file_operations 中定义的就是 socket 对应的 send/recv 等操作；而对于cgroups这样的特殊文件系统，file_operations 中定义的就是操作 cgroup 结构体等具体的实现。
4. 目录项对象(dentry object)，在每个文件系统中，内核在查找某一个路径中的文件时，会为内核路径上的每一个分量都生成一个目录项对象，通过目录项对象能够找到对应的 inode 对象，目录项对象一般会被缓存，从而提高内核查找速度。

使用 cgroups

cgroup 内核功能比较有趣的地方是它没有提供任何的系统调用接口，而是对 linux vfs 的一个实现，因此可以用类似文件系统的方式进行操作。

使用 cgroups 的方式有几种：

- 使用 cgroups 提供的虚拟文件系统，直接通过创建、读写和删除目录、文件来控制 cgroups
- 使用命令行工具，比如 libcgroup 包提供的 cgcreate、cgexec、cgclassify 命令
- 使用 `rules engine daemon` 提供的配置文件
- 当然，systemd、lxc、docker 这些封装了 cgroups 的软件也能让你通过它们定义的接口控制 cgroups 的内容

直接操作 cgroup 文件系统

查看 cgroups 挂载信息

在 ubuntu 18.04 的机器上，cgroups 已经挂载到文件系统上了，可以通过 `mount` 命令查看：

```
$ mount -t cgroup
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
```

如果没有的话，也可以通过以下命令来把想要的 subsystem mount 到系统中：

```
$ mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
```

上述命令表示把 cpu、cpuset、memory 三个子资源 mount 到 `/cgroup/cpu_and_mem` 目录下。

每个 cgroup 目录下面都会有描述该 cgroup 的文件，除了每个 cgroup 独特的资源控制文件，还有一些通用的文件：

- `tasks`：当前 cgroup 包含的任务（task）pid 列表，把某个进程的 pid 添加到这个文件中就等于把进程移到该 cgroup
- `cgroup.procs`：当前 cgroup 中包含的 thread group 列表，使用逻辑和 `tasks` 相同
- `notify_on_release`：0 或者 1，是否在 cgroup 销毁的时候执行 notify。如果为 1，那么当这个 cgroup 最后一个任务离开时（退出或者迁移到其他 cgroup），并且最后一个子 cgroup 被删除时，系统会执行 `release_agent` 中指定的命令
- `release_agent`：需要执行的命令

创建 cgroup

创建 cgroup，可以直接用 `mkdir` 在对应的子资源中创建一个目录：

```
$ mkdir /sys/fs/cgroup/cpu/mycgroup
$ ll /sys/fs/cgroup/cpu/mycgroup
total 0
-rw-r--r-- 1 root root 0 Dec 13 08:02 cgroup.clone_children
-rw-r--r-- 1 root root 0 Dec 13 08:02 cgroup.procs
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.stat
-rw-r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_all
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_sys
-r--r--r-- 1 root root 0 Dec 13 08:02 cpuacct.usage_user
-rw-r--r-- 1 root root 0 Dec 13 08:02 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 Dec 13 08:02 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 Dec 13 08:02 cpu.shares
-r--r--r-- 1 root root 0 Dec 13 08:02 cpu.stat
-rw-r--r-- 1 root root 0 Dec 13 08:02 notify_on_release
-rw-r--r-- 1 root root 0 Dec 13 08:02 tasks
```

上面命令在 cpu 子资源中创建了 `mycgroup`，创建 cgroup 之后，目录中会自动创建需要的文件。我们后面会详细讲解这些文件的含义，目前只需要知道它们能够控制对应子资源就行。

删除 cgroup

删除子资源，就是删除对应的目录：

```
rmdir /sys/fs/cgroup/cpu/mycgroup/
```

删除之后，如果 tasks 文件中有进程，它们会自动迁移到父 cgroup 中。

设置 cgroup 参数

设置 group 的参数就是直接往特定的文件中写入特定格式的内容，比如要限制 cgroup 能够使用的 CPU 核数：

```
echo 0-1 > /sys/fs/cgroup/cpuset/mycgroup/cpuset.cpus
```

把进程加入到 cgroup

要把某个已经运行的进程加入到 cgroup，可以直接往需要的 cgroup tasks 文件中写入进程的 PID：

```
echo 2358 > /sys/fs/cgroup/memory/mycgroup/tasks
```

在 cgroup 中运行进程

如果想直接把进程运行在某个 cgroup，但是运行前还不知道进程的 Pid 应该怎么办呢？

我们可以利用 cgroup 的继承方式来实现，因为子进程会继承父进程的 cgroup，因此我们可以把当前 shell 加入到要想的 cgroup：

```
echo $$ > /sys/fs/cgroup/cpu/mycgroup/tasks
```

上面的方案有个缺陷，运行完之后原来的 shell 还在 cgroup 中。如果希望进程运行完不影响当前使用的 shell，可以另起一个临时的 shell：

```
sh -c "echo \$$ > /sys/fs/cgroup/memory/mycgroup/tasks & & stress -m 1"
```

把进程移动到 cgroup

如果想要把进程移动到另外一个 cgroup，只要使用 `echo` 把进程 PID 写入到 cgroup tasks 文件中即可，原来 cgroup tasks 文件会自动删除该进程。

cgroup-tools

`cgroup-tools` 软件包提供了一系列命令可以操作和管理 cgroup，ubuntu 系统中可以通过下面的命令安装：

```
sudo apt-get install -y cgroup-tools
```

列出 cgroup mount 信息

最简单的，`lssubsys` 可以查看系统中存在的 subsystems：

```
lssubsys -am
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
blkio /sys/fs/cgroup/blkio
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls,net_prio /sys/fs/cgroup/net_cls,net_prio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
pids /sys/fs/cgroup/pids
rdma /sys/fs/cgroup/rdma
```

创建 cgroup

`cgcreate` 可以用来为用户创建指定的 cgroups：

```
sudo cgcreate -a cizixs -t cizixs -g cpu,memory:test1
ls cpu/test1
cgroup.clone_children  cpuacct.stat   cpuacct.usage_all     cpuacct.usage_percpu_sys   cpuacct.usage_sys   cpu.
cfs_period_us  cpu.shares  notify_on_release
cgroup.procs           cpuacct.usage  cpuacct.usage_percpu  cpuacct.usage_percpu_user  cpuacct.usage_user  cpu.
cfs_quota_us   cpu.stat    tasks
```

上面的命令表示在 `/sys/fs/cgroup/cpu` 和 `/sys/fs/cgroup/memory` 目录下面分别创建 `test1` 目录，也就是为 cpu 和 memory 子资源创建对应的 cgroup。

- 选项 `-t` 指定 `tasks` 文件的用户和组，也就是指定哪些人可以把任务添加到 cgroup 中，默认是从父 cgroup 继承
- `-a` 指定除了 `tasks` 之外所有文件（资源控制文件）的用户和组，也就是哪些人可以管理资源参数
- `-g` 指定要添加的 cgroup，冒号前是逗号分割的子资源类型，冒号后面是 cgroup 的路径（这个路径会添加到对应资源 mount 到的目录后面）。也就是说在特定目录下面添加指定的子资源

删除 cgroup

知道怎么创建，也要知道怎么删除。不然系统中保留着太多用不到的 cgroup 浪费系统资源，也会让管理很麻烦。

`cgdelete` 可以删除对应的 cgroups，它和 `cgcreate` 命令类似，可以用 `-g` 指定要删除的 cgroup：

```
cgroup sudo cgdelete -g cpu,memory:test1
```

`cgdelete` 也提供了 `-r` 参数可以递归地删除某个 cgroup 以及它所有的子 cgroup。

如果被删除的 cgroup 中有任务，这些任务会自动移到父 cgroup 中。

设置 cgroup 的参数

`cgset` 命令可以设置某个子资源的参数，比如如果要限制某个 cgroup 中任务能使用的 CPU 核数：

```
cgset -r cpuset.cpus=0-1 /mycgroup
```

`-r` 后面跟着参数的键值对，每个子资源能够配置的键值对都有自己的规定，我们会在后面详细解释。

`cgset` 还能够把一个 cgroup 的参数拷贝到另外一个 cgroup 中：

```
cgset --copy-from group1/ group2/
```

NOTE: cgset 如果设置没有成功也不会报错，请一定要注意。

在某个 cgroup 中运行进程

`cgexec` 执行某个程序，并把程序添加到对应的 cgroups 中：

```
cgroup cgexec -g memory,cpu:cizixs bash
```

cgroups 是可以有层级结构的，因此可以直接创建具有层级关系的 cgroup，然后运行在该 cgroup 中：

```
cgcreate -g memory,cpu:groupname/foo
cgexec -g memory,cpu:groupname/foo bash
```

把已经运行的进程移动到某个 cgroup

要把某个已经存在的程序（能够知道它的 pid）移到某个 cgroup，可以使用 `cgclassify` 命令:

比如把当前 bash shell 移入到特定的 cgroup 中

```
cgclassify -g memory,cpu:/mycgroup $$
```

$$ 表示当前进程的 pid 号，上面命令可以方便地测试一些耗费内存或者 CPU 的进程，如果 /mycgroup 对 CPU 和 memory 做了限制。

这个命令也可以同时移动多个进程，它们 pid 之间用空格隔开：

```
cgclassify -g cpu,memory:group1 1701 1138
```

cgroup 子资源参数详解

每个 subssytem 负责系统的一部分资源管理，又分别提供多个参数可以控制，每个参数对应一个文件，往文件中写入特定格式的内容就能控制该资源。

blkio：限制设备 IO 访问

限制磁盘 IO 有两种方式：权重（weight）和上限（limit）。权重是给不同的应用（或者 cgroup）一个权重值，各个应用按照百分比来使用 IO 资源；上限是直接写死应用读写速率的最大值。

设置 cgroup 访问设备的权重：

设置的权重并不能保证什么，当只有某个应用在读写磁盘时，不管它权重多少，都能使用磁盘。只有当多个应用同时读写磁盘时，才会根据权重为应用分配读写的速率。

- `blkio.weight`：设置 cgroup 读写设备的权重，取值范围在 100-1000
- `blkio.weight_device`：设置 cgroup 使用某个设备的权重。当访问该设备时，它会使用当前值，覆盖 `blkio.weight` 的值。内容的格式为 `major:minor weight`，前面是设备的 major 和 minor 编号，用来唯一表示一个设备，后面是 100-1000 之间的整数值。设备号的分配可以参考：https://www.kernel.org/doc/html/v4.11/admin-guide/devices.html

设置 cgroup 访问设备的限制：

除了设置权重之外，还能设置 cgroup 磁盘的使用上限，保证 cgroup 中的进程读写磁盘的速率不会超过某个值。

- `blkio.throttle.read_bps_device`：最多每秒钟从设备读取多少字节
- `blkio.throttle.read_iops_device`：最多每秒钟从设备中执行多少次读操作
- `blkio.throttle.write_bps_device`：最多每秒钟可以往设备写入多少字节
- `blkio.throttle.write_iops_device`：最多每秒钟可以往设备执行多少次写操作

读写字节数的限制格式一样 `major:minor bytes_per_second`，前面两个数字代表某个设备，后面跟着一个整数，代表每秒读写的字节数，单位为比特，如果需要其他单位（KB、MB等）需要自行转换。比如要限制 `/dev/sda` 读速率上线为 10 Mbps，可以运行：

```
echo "8:0 10485760" >
/sys/fs/cgroup/blkio/mygroup/blkio.throttle.read_bps_device
```

iops 代表 IO per second，是每秒钟执行读写的次数，格式为 `major:minor operations_per_second`。比如，要限制每秒只能写 10 次，可以运行：

```
echo "8:0 10" >
/sys/fs/cgroup/blkio/mygroup/blkio.throttle.write_iops_device
```

除了限制磁盘使用之外，blkio 还提供了 throttle 规则下磁盘使用的统计数据。

- `blkio.throttle.io_serviced`：cgroup 中进程读写磁盘的次数，文件中内容格式为 `major:minor operation number`，表示对磁盘进行某种操作（read、write、sync、async、total）的次数
- `blkio.throttle.io_service_bytes`：和上面类似，不过这里保存的是操作传输的字节数
- `blkio.reset_stats`：重置统计数据，往该文件中写入一个整数值即可
- `blkio.time`：统计 cgroup 对各个设备的访问时间，格式为 `major:minor milliseconds`
- `blkio.io_serviced`：CFQ 调度器下，cgroup 对设备的各种操作次数，和 `blkio.throttle.io_serviced` 刚好相反，所有不是 throttle 下的请求
- `blkio.io_services_bytes`：CFQ 调度器下，cgroup 对各种设备的操作字节数
- `blkio.sectors`：cgroup 中传输的扇区次数，格式为 `major:minor sector_count`
- `blkio.queued`：cgroup IO 请求进队列的次数，格式为 `number operation`
- `blkio.dequeue`：cgroup 的 IO 请求被设备出队列的次数，格式为 `major:minor number`
- `blkio.avg_queue_size`：
- `blkio.merged`：cgroup 把 BIOS 请求合并到 IO 操作请求的次数，格式为 `number operation`
- `blkio.io_wait_time`：cgroup 等待队列服务的时间
- `blkio.io_service_time`：CFQ 调度器下，cgroup 处理请求的时间（从请求开始调度，到 IO 操作完成）

cpu：限制进程组 CPU 使用

CPU 子资源可以管理 cgroup 中任务使用 CPU 的行为，任务使用 CPU 资源有两种调度方式：完全公平调度（CFS，Completely Fair Scheduler）和 实时调度（RT，Real-Time Scheduler）。前者可以根据权重为任务分配响应的 CPU 时间片，后者能够限制使用 CPU 的核数。

CFS 调优参数：

CFS 调度下，每个 cgroup 都会分配一个权重，但是这个权重并不能保证任务使用 CPU 的具体数据。如果只有一个进程在运行（理论上，现实中机器上不太可能只有一个进程），不管它所在 cgroup 对应的 CPU 权重是多少，都能使用所有的 CPU 资源；在 CPU 资源紧张的情况，内核会根据 cgroup 的权重按照比例分配个给任务各自使用 CPU 的时间片。

CFS 调度模式下，也可以给 cgroup 分配一个使用上限，限制任务能使用 CPU 的核数。

设置 CPU 数字的单位都是微秒（microsecond），用 us 表示。

- `cpu.cfs_quota_us`：每个周期 cgroup 中所有任务能使用的 CPU 时间，默认为 `-1`，表示不限制 CPU 使用。需要配合 `cpu.cfs_period_us` 一起使用，一般设置为 `100000`（docker 中设置的值）
- `cpu.cfs_period_us`：每个周期中 cgroup 任务可以使用的时间周期，如果想要限制 cgroup 任务每秒钟使用 0.5 秒 CPU，可以在 `cpu.cfs_quota_us` 为 `100000` 的情况下把它设置为 `50000`。如果它的值比 `cfs_quota_us` 大，表明进程可以使用多个核 CPU，比如 `200000` 表示进程能够使用 `2.0` 核
- `cpu.stat`：CPU 使用的统计数据，`nr_periods` 表示已经过去的时间周期；`nr_throttled` 表示 cgroup 中任务被限制使用 CPU 的次数（因为超过了规定的上限）；`throttled_time` 表示被限制的总时间
- `cpu.shares`：cgroup 使用 CPU 时间的权重值。如果两个 cgroup 的权重都设置为 100，那么它们里面的任务同时运行时，使用 CPU 的时间应该是一样的；如果把其中一个权重改为 200，那么它能使用的 CPU 时间将是对方的两倍。

RT 调度模式下的参数：

RT 调度模式下和 CFS 中上限设置类似，区别是它只是限制实时任务的 CPU。

- `cpu.rt_period_us`：设置一个周期时间，表示多久 cgroup 能够重新分配 CPU 资源
- `cpu.rt_runtime_us`：设置运行时间，表示在周期时间内 cgroup 中任务能访问 CPU 的时间。这个限制是针对单个 CPU 核数的，如果是多核，需要乘以对应的核数

cpuacct： 任务使用 CPU 情况统计

cpuacct 不做任何资源限制，它的功能是资源统计，自动地统计 cgroup 中任务对 CPU 资源的使用情况，统计数据也包括子 cgroup 中的任务。

- `cpuacct.usage`：该 cgroup 中所有任务（包括子 cgroup 中的任务，下同）总共使用 CPU 的时间，单位是纳秒（ns）。往文件中写入 `0` 可以重置统计数据
- `cpuacct.stat`：该 cgroup 中所有任务使用 CPU 的user 和 system 时间，也就是用户态 CPU 时间和内核态 CPU 时间
- `cpuacct.usage_percpu`：该 cgroup 中所有任务使用各个 CPU 核数的时间，单位为纳秒（ns）

cpuset: cpu 绑定

除了限制 CPU 的使用量，cgroup 还能把任务绑定到特定的 CPU，让它们只运行在这些 CPU 上，这就是 `cpuset` 子资源的功能。除了 CPU 之外，还能绑定内存节点（memory node）。

NOTE：在把任务加入到 cpuset 的 task 文件之前，用户必须设置 `cpuset.cpus` 和 `cpuset.mems` 参数。

- `cpuset.cpus`：设置 cgroup 中任务能使用的 CPU，格式为逗号（`,`）隔开的列表，减号（`-`）可以表示范围。比如，`0-2,7` 表示 CPU 第 0，1，2，和 7 核。
- `cpuset.mems`：设置 cgroup 中任务能使用的内存节点，和 `cpuset.cpus` 格式一样

上面两个是最常用的参数，`cpuset` 中有很多其他参数，需要对 CPU 调度机制有深入的了解，很少用到，而且我也不懂，所以就不写了，具体可以参考参考文档中 RedHat 网站。

memory：限制内存使用

memory 子资源系统能限制 cgroup 中任务对内存的使用，也能生成它们使用数据的报告。

控制内存使用：

- `memory.limit_in_bytes`：cgroup 能使用的内存上限值，默认为字节；也可以添加 `k/K`、`m/M` 和 `g/G` 单位后缀。往文件中写入 `-1` 来移除设置的上限，表示不对内存做限制
- `memory.memsw.limit_in_bytes`：cgroup 能使用的内存加 swap 上限，用法和上面一样。写入 `-1` 来移除上限
- `memory.failcnt`：任务使用内存量达到 `limit_in_bytes` 上限的次数
- `memory.memsw.failcnt`：任务使用内存加 swap 量达到 `memsw.limit_in_bytes` 上限的次数
- `memory.soft_limit_in_bytes`：设置内存软上限。如果内存充足， cgroup 中的任务可以用到 `memory.limit_in_bytes` 设定的内存上限；当时当内存资源不足时，内核会让任务使用的内存不超过 `soft_limit_in_bytes` 中的值。文件内容的格式和 `limit_in_bytes` 一样
- `memory.swappiness`：设置内核 swap out 进程内存（而不是从 page cache 中回收页） 的倾向。默认值为 60，低于 60 表示降低倾向，高于 60 表示增加倾向；如果值高于 100，表示允许内核 swap out 进程地址空间的页。如果值为 0 表示倾向很低，而不是禁止该行为。

OOM 操作：

OOM 是 out of memory 的缩写，可以翻译成内存用光。cgroup 可以控制内存用完之后应该怎么处理进程，默认情况下，用光内存的进程会被杀死。

`memory.oom_control`：是否启动 OOM killer，如果启动（值为 0，是默认情况）超过内存限制的进程会被杀死；如果不启动（值为 1），使用超过限定内存的进程不会被杀死，而是被暂停，直到它释放了内存能够被继续使用。

统计内存使用情况：

- `memory.stat`

  ：汇报内存的使用情况，里面的数据包括：

  - `cache`：页缓存（page cache）字节数，包括 tmpfs（shmem）
  - `rss`：匿名和 swap cache 字节数，不包括 tmpfs
  - `mapped_file`：内存映射（memory-mapped）的文件大小，包括 tmpfs，单位是字节
  - `pgpgin`: paged into 内存的页数
  - `pgpgout`：paged out 内存的页数
  - `swap`：使用的 swap 字节数
  - `active_anon`：活跃的 LRU 列表中匿名和 swap 缓存的字节数，包括 tmpfs
  - `inactive_anon`：不活跃的 LRU 列表中匿名和 swap 缓存的字节数，包括 tmpfs
  - `active_file`：活跃 LRU 列表中文件支持的（file-backed）的内存字节数
  - `inactive_file`：不活跃列表中文件支持的（file-backed）的内存字节数
  - `unevictable`：不可以回收的内存字节数
- `memory.usage_in_bytes`：cgroup 中进程当前使用的总内存字节数
- `memory.memsw.usage_in_bytes`：cgroup 中进程当前使用的总内存加上总 swap 字节数
- `memory.max_usage_in_bytes`：cgroup 中进程使用的最大内存字节数
- `memory.memsw.max_usage_in_bytes`：cgroup 中进程使用的最大内存加 swap 字节数

net_cls：为网络报文分类

`net_cls` 子资源能够给网络报文打上一个标记（classid），这样内核的 tc（traffic control）模块就能根据这个标记做流量控制。

`net_cls.classid`：包含一个整数值。从文件中读取是的十进制，写入的时候需要是十六进制。比如，`0x100001` 写入到文件中，读取的将是 `1048577`，`ip` 命令操作的形式为 `10:1`。

这个值的格式为 `0xAAAABBBB`，一共 32 位，分成前后两个部分，前置的 0 可以忽略，因此 `0x10001` 和 `0x00010001` 一样，表示为 `1:1`。

net_prio：网络报文优先级

`net_prio`（Network Priority）子资源能够动态设置 cgroup 中应用在网络接口的优先级。网络优先级是报文的一个属性值，`tc`可以设置网络的优先级，socket 也可以通过 `SO_PRIORITY` 选项设置它（但是很少应用会这么做）。

- `net_prio.prioidx`：只读文件，里面包含了一个整数值，内核用来标识这个 cgroup
- `net_prio.ifpriomap`：网络接口的优先级，里面可以包含很多行，用来为从网络接口中发出去的报文设置优先级。每行的格式为 `network_interface priority`，比如 `echo "eth0 5" > /sys/fs/cgroup/net_prio/mycgroup/net_prio.ifpriomap`

devices：设备黑白名单

device 子资源可以允许或者阻止 cgroup 中的任务访问某个设备，也就是黑名单和白名单的作用。

- `devices.allow`

  ：cgroup 中的任务能够访问的设备列表，格式为

  ```
  type major:minor access
  ```

  ，

  - `type` 表示类型,可以为 a(all), c(char), b(block)
  - `major:minor` 代表设备编号，两个标号都可以用`*` 代替表示所有，比如 `*:*` 代表所有的设备
  - `accss` 表示访问方式，可以为 `r`(read),`w`(write), `m`(mknod) 的组合
- `devices.deny`：cgroup 中任务不能访问的设备，和上面的格式相同
- `devices.list`：列出 cgroup 中设备的黑名单和白名单

freezer

`freezer` 子资源比较特殊，它并不和任何系统资源相关，而是能暂停和恢复 cgroup 中的任务。

- `freezer.state`

  ：这个文件值存在于非根 cgroup 中（因为所有的任务默认都在根 cgroup 中，停止所有的任务显然是错误的行为），里面的值表示 cgroup 中进程的状态：

  - `FROZEN`：cgroup 中任务都被挂起（暂停）
  - `FREEZING`：cgroup 中任务正在被挂起的过程中
  - `THAWED`：cgroup 中的任务已经正常恢复

要想挂起某个进程，需要先把它移动到某个有 freezer 的 cgroup 中，然后 Freeze 这个 cgroup。

NOTE:如果某个 cgroup 处于挂起状态，不能往里面添加任务。用户可以写入 `FROZEN` 和 `THAWED` 来控制进程挂起和恢复，`FREEZING` 不受用户控制。

总结

cgroup 提供了强大的功能，能够让我们控制应用的资源使用情况，也能统计资源使用数据，是容器技术的基础。但是 cgroup 整个系统也很复杂，甚至显得有些混乱，目前 cgroup 整个在被重写，新的版本被称为 cgroup V2，而之前的版本也就被称为了 V1。

cgroup 本身并不提供对网络资源的使用控制，只能添加简单的标记和优先级，具体的控制需要借助 linux 的 TC 模块来实现。

参考资料

- An introduction to cgroups and cgroupspy
- LXC, Cgroups and Advanced Linux Container Technology Lecture
- redhat doc:Subsystems and Tunable Parameters
- Docker背后的内核知识——cgroups资源限制
- Docker资源管理探秘：Docker背后的内核Cgroups机制
- Linux 内核 cgroups 简介
- 王喆锋： Linux Cgroups 详解
- Linux Cgroups V2 设计
- Understanding the new control groups API
- Resource Management: Linux Kernel Namespaces and cgroups – Rami Rosen
- https://tech.meituan.com/2015/03/31/cgroups.html