

## 58 | CGroup技术：内部创业公司应该独立核算成本

2019-08-09 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 18:40 大小 17.11M



我们前面说了，容器实现封闭的环境主要要靠两种技术，一种是“看起来是隔离”的技术 Namespace，另一种是用起来是隔离的技术 CGroup。

上一节我们讲了“看起来隔离”的技术 Namespace，这一节我们就来看一下“用起来隔离”的技术 CGroup。

CGroup 全称是 Control Group，顾名思义，它是用来做“控制”的。控制什么东西呢？当然是资源的使用了。那它都能控制哪些资源的使用呢？我们一起来看一看。

首先，cgroups 定义了下面的一系列子系统，每个子系统用于控制某一类资源。

cpu 子系统，主要限制进程的 cpu 使用率。

cpuacct 子系统，可以统计 cgroups 中的进程的 cpu 使用报告。

cpuset 子系统，可以为 cgroups 中的进程分配单独的 cpu 节点或者内存节点。

memory 子系统，可以限制进程的 memory 使用量。

blkio 子系统，可以限制进程的块设备 io。

devices 子系统，可以控制进程能够访问某些设备。


net\_cls 子系统，可以标记 cgroups 中进程的网络数据包，然后可以使用 tc 模块 ( traffic control ) 对数据包进行控制。

freezer 子系统，可以挂起或者恢复 cgroups 中的进程。

这么多子系统，你可能要说了，那我们不用都掌握吧？没错，这里面最常用的是对于 CPU 和内存的控制，所以下面我们详细来说它。

在容器这一章的第一节，我们讲了，Docker 有一些参数能够限制 CPU 和内存的使用，如果把它落地到 Cgroup 里面会如何限制呢？

为了验证 Docker 的参数与 Cgroup 的映射关系，我们运行一个命令特殊的 docker run 命令，这个命令比较长，里面的参数都会映射为 cgroup 的某项配置，然后我们运行 docker ps，可以看到，这个容器的 id 为 3dc0601189dd。

 复制代码

```
1 docker run -d --cpu-shares 513 --cpus 2 --cpuset-cpus 1,3 --memory 1024M --memory-swap :
2
3 # docker ps
4 CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
5 3dc0601189dd        testnginx:1        "/bin/sh -c 'nginx -..."   About a minute ago   Up
```

在 Linux 上，为了操作 Cgroup，有一个专门的 Cgroup 文件系统，我们运行 mount 命令可以查看。

 复制代码

```
1 # mount -t cgroup
2 cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,relatime)
3 cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
4 cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
```

```
5 cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
6 cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
7 cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpuacct)
8 cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
9 cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
10 cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugepages)
11 cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
12 cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
```

cgroup 文件系统多挂载到 /sys/fs/cgroup 下，通过上面的命令行，我们可以看到我们可以用 cgroup 控制哪些资源。


对于 CPU 的控制，我在这一章的第一节讲过，Docker 可以控制 cpu-shares、cpus 和 cpuset。

我们在 /sys/fs/cgroup/ 下面能看到下面的目录结构。

 复制代码

```
1 drwxr-xr-x 5 root root 0 May 30 17:00 blkio
2 lrwxrwxrwx 1 root root 11 May 30 17:00 cpu -> cpu,cpuacct
3 lrwxrwxrwx 1 root root 11 May 30 17:00 cpuacct -> cpu,cpuacct
4 drwxr-xr-x 5 root root 0 May 30 17:00 cpu,cpuacct
5 drwxr-xr-x 3 root root 0 May 30 17:00 cpuset
6 drwxr-xr-x 5 root root 0 May 30 17:00 devices
7 drwxr-xr-x 3 root root 0 May 30 17:00 freezer
8 drwxr-xr-x 3 root root 0 May 30 17:00 hugetlb
9 drwxr-xr-x 5 root root 0 May 30 17:00 memory
10 lrwxrwxrwx 1 root root 16 May 30 17:00 net_cls -> net_cls,net_prio
11 drwxr-xr-x 3 root root 0 May 30 17:00 net_cls,net_prio
12 lrwxrwxrwx 1 root root 16 May 30 17:00 net_prio -> net_cls,net_prio
13 drwxr-xr-x 3 root root 0 May 30 17:00 perf_event
14 drwxr-xr-x 5 root root 0 May 30 17:00 pids
15 drwxr-xr-x 5 root root 0 May 30 17:00 systemd
```


我们可以想象，CPU 的资源控制的配置文件，应该在 cpu,cpuacct 这个文件夹下面。

 复制代码

```
1 # ls
2 cgroup.clone_children  cpu.cfs_period_us  notify_on_release
3 cgroup.event_control  cpu.cfs_quota_us  release_agent
```


```
4 cgroup.procs          cpu.rt_period_us    system.slice
5 cgroup.sane_behavior  cpu.rt_runtime_us   tasks
6 cpuacct.stat          cpu.shares          user.slice
7 cpuacct.usage         cpu.stat
8 cpuacct.usage_percpu  docker
```

果真，这下面是对 cpu 的相关控制，里面还有一个路径叫 docker。我们进入这个路径。

 复制代码


```
1 ]# ls
2 cgroup.clone_children
3 cgroup.event_control
4 cgroup.procs
5 cpuacct.stat
6 cpuacct.usage
7 cpuacct.usage_percpu
8 cpu.cfs_period_us
9 cpu.cfs_quota_us
10 cpu.rt_period_us
11 cpu.rt_runtime_us
12 cpu.shares
13 cpu.stat
14 3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd
15 notify_on_release
16 tasks
```

这里面有个很长的 id，是我们创建的 docker 的 id。

 复制代码


```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# ls
2 cgroup.clone_children  cpuacct.usage_percpu  cpu.shares
3 cgroup.event_control  cpu.cfs_period_us     cpu.stat
4 cgroup.procs          cpu.cfs_quota_us      notify_on_release
5 cpuacct.stat          cpu.rt_period_us      tasks
6 cpuacct.usage         cpu.rt_runtime_us
```

在这里，我们能看到 cpu.shares，还有一个重要的文件 tasks。这里面是这个容器里所有进程的进程号，也即所有这些进程都被这些 cpu 策略控制。

 复制代码


```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat tasks
2 39487
3 39520
4 39526
5 39527
6 39528
7 39529
```

如果我们查看 `cpu.shares`，里面就是我们设置的 513。

 复制代码


```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat cpu.shares
2 513
```

另外，我们还配置了 `cpus`，这个值其实是由 `cpu.cfs_period_us` 和 `cpu.cfs_quota_us` 共同决定的。`cpu.cfs_period_us` 是运行周期，`cpu.cfs_quota_us` 是在周期内这些进程占用多少时间。我们设置了 `cpus` 为 2，代表的意思是，在周期 100000 毫秒的运行周期内，这些进程要占用 200000 毫秒的时间，也即需要两个 CPU 同时运行一个整整的周期。

 复制代码

```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat cpu.cfs_period_u
2 100000
3 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat cpu.cfs_quota_u:
4 200000
```


对于 `cpuset`，也即 `cpu` 绑核的参数，在另外一个文件夹里面 `/sys/fs/cgroup/cpuset`，这里面同样有一个 `docker` 文件夹，下面同样有 `docker id` 也即 `3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd` 文件夹，这里面的 `cpuset.cpus` 就是配置的绑定到 1、3 两个核。

 复制代码

```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat cpuset.cpus
2 1,3
```


这一章的第一节我们还讲了 Docker 可以限制内存的使用量，例如 memory、memory-swap、memory-swappiness。这些在哪里控制呢？

/sys/fs/cgroup/ 下面还有一个 memory 路径，控制策略就是在这里面定义的。

 复制代码

```
1 [root@deployer memory]# ls
2 cgroup.clone_children          memory.memsw.failcnt
3 cgroup.event_control          memory.memsw.limit_in_bytes
4 cgroup.procs                  memory.memsw.max_usage_in_bytes
5 cgroup.sane_behavior          memory.memsw.usage_in_bytes
6 docker                        memory.move_charge_at_immigrate
7 memory.failcnt                memory.numa_stat
8 memory.force_empty            memory.oom_control
9 memory.kmem.failcnt           memory.pressure_level
10 memory.kmem.limit_in_bytes    memory.soft_limit_in_bytes
11 memory.kmem.max_usage_in_bytes memory.stat
12 memory.kmem.slabinfo         memory.swappiness
13 memory.kmem.tcp.failcnt       memory.usage_in_bytes
14 memory.kmem.tcp.limit_in_bytes memory.use_hierarchy
15 memory.kmem.tcp.max_usage_in_bytes notify_on_release
16 memory.kmem.tcp.usage_in_bytes release_agent
17 memory.kmem.usage_in_bytes    system.slice
18 memory.limit_in_bytes         tasks
19 memory.max_usage_in_bytes     user.slice
```

这里面全是对于 memory 的控制参数，在这里面我们可看到了 docker，里面还有容器的 id 作为文件夹。

 复制代码


```
1 [docker]# ls
2 3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd
3 cgroup.clone_children
4 cgroup.event_control
5 cgroup.procs
6 memory.failcnt
7 memory.force_empty
8 memory.kmem.failcnt
9 memory.kmem.limit_in_bytes
10 memory.kmem.max_usage_in_bytes
11 memory.kmem.slabinfo
```

```

12 memory.kmem.tcp.failcnt
13 memory.kmem.tcp.limit_in_bytes
14 memory.kmem.tcp.max_usage_in_bytes
15 memory.kmem.tcp.usage_in_bytes
16 memory.kmem.usage_in_bytes
17 memory.limit_in_bytes
18 memory.max_usage_in_bytes
19 memory.memsw.failcnt
20 memory.memsw.limit_in_bytes
21 memory.memsw.max_usage_in_bytes
22 memory.memsw.usage_in_bytes
23 memory.move_charge_at_immigrate
24 memory.numa_stat
25 memory.oom_control
26 memory.pressure_level
27 memory.soft_limit_in_bytes
28 memory.stat
29 memory.swappiness
30 memory.usage_in_bytes
31 memory.use_hierarchy
32 notify_on_release
33 tasks
34
35 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# ls
36 cgroup.clone_children          memory.memsw.failcnt
37 cgroup.event_control           memory.memsw.limit_in_bytes
38 cgroup.procs                  memory.memsw.max_usage_in_bytes
39 memory.failcnt                memory.memsw.usage_in_bytes
40 memory.force_empty             memory.move_charge_at_immigrate
41 memory.kmem.failcnt            memory.numa_stat
42 memory.kmem.limit_in_bytes     memory.oom_control
43 memory.kmem.max_usage_in_bytes memory.pressure_level
44 memory.kmem.slabinfo           memory.soft_limit_in_bytes
45 memory.kmem.tcp.failcnt        memory.stat
46 memory.kmem.tcp.limit_in_bytes memory.swappiness
47 memory.kmem.tcp.max_usage_in_bytes memory.usage_in_bytes
48 memory.kmem.tcp.usage_in_bytes memory.use_hierarchy
49 memory.kmem.usage_in_bytes     notify_on_release
50 memory.limit_in_bytes          tasks
51 memory.max_usage_in_bytes

```

在 docker id 的文件夹下面，有一个 memory.limit\_in\_bytes，里面配置的就是 memory。

 复制代码


```

1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat memory.limit_in_
2 1073741824

```




还有 `memory.swappiness`，里面配置的就是 `memory-swappiness`。

 复制代码

```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat memory.swappiness:
2 7
```




还有就是 `memory.memsw.limit_in_bytes`，里面配置的是 `memory-swap`。

 复制代码

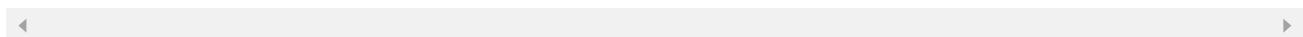
```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat memory.memsw.lir
2 1293942784
```



我们还可以看一下 `tasks` 文件的内容，`tasks` 里面是容器里面所有进程的进程号。

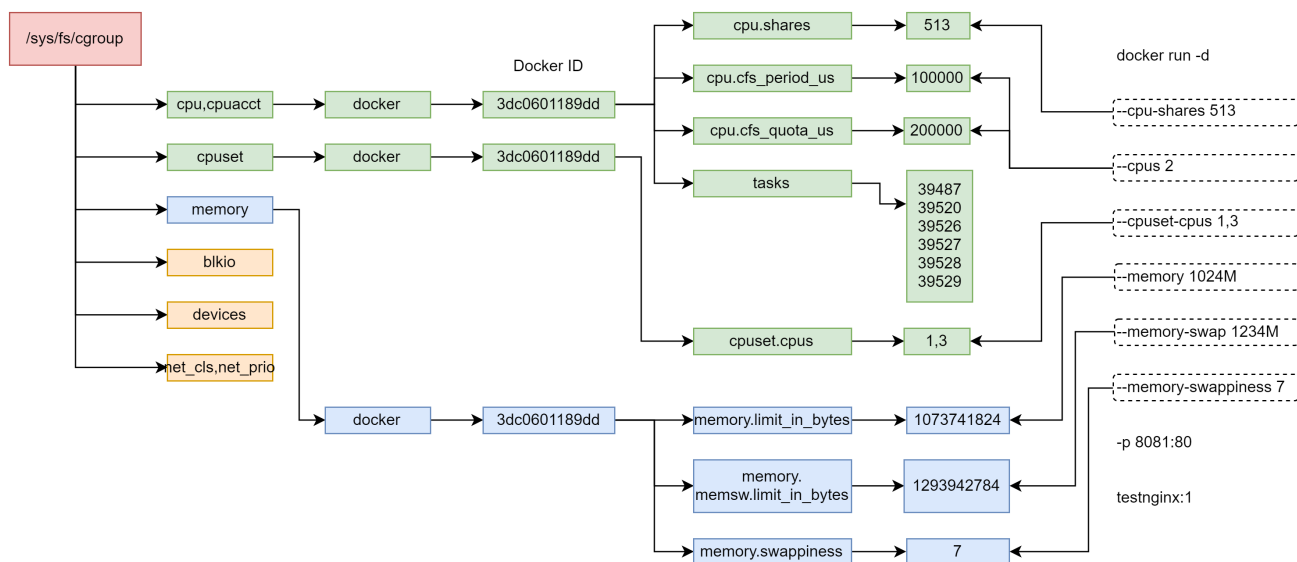
 复制代码

```
1 [3dc0601189dd218898f31f9526a6cfae83913763a4da59f95ec789c6e030ecfd]# cat tasks
2 39487
3 39520
4 39526
5 39527
6 39528
7 39529
```



至此，我们看到了 `cgroup` 对于 `Docker` 资源的控制，在用户态是如何表现的。我画了一张图总结一下。





在内核中，cgroup 是如何实现的呢？

首先，在系统初始化的时候，cgroup 也会进行初始化，在 start\_kernel 中，cgroup\_init\_early 和 cgroup\_init 都会进行初始化。

[复制代码](#)

```

1 asmlinkage __visible void __init start_kernel(void)
2 {
3     .....
4     cgroup_init_early();
5     .....
6     cgroup_init();
7     .....
8 }
  
```


在 cgroup\_init\_early 和 cgroup\_init 中，会有下面的循环。

[复制代码](#)

```


1 for_each_subsys(ss, i) {
2     ss->id = i;
3     ss->name = cgroup_subsys_name[i];
4     .....
5     cgroup_init_subsys(ss, true);
6 }
7
8 #define for_each_subsys(ss, ssid) \
9     for ((ssid) = 0; (ssid) < CGROUP_SUBSYS_COUNT && \
10         (((ss) = cgroup_subsys[ssid]) || true); (ssid)++)
  
```

for\_each\_subsys 会在 cgroup\_subsys 数组中进行循环。这个 cgroup\_subsys 数组是如何形成的呢？

 复制代码

```
1 #define SUBSYS(_x) [_x ## _cgrp_id] = &_amp;_x ## _cgrp_subsys,
2 struct cgroup_subsys *cgroup_subsys[] = {
3 #include <linux/cgroup_subsys.h>
4 };
5 #undef SUBSYS
```

SUBSYS 这个宏定义了这个 cgroup\_subsys 数组，数组中的项定义在 cgroup\_subsys.h 头文件中。例如，对于 CPU 和内存有下面的定义。

 复制代码

```
1 //cgroup_subsys.h
2
3 #if IS_ENABLED(CONFIG_CPUSETS)
4 SUBSYS(cpuaset)
5 #endif
6
7 #if IS_ENABLED(CONFIG_CGROUP_SCHED)
8 SUBSYS(cpu)
9 #endif
10
11 #if IS_ENABLED(CONFIG_CGROUP_CPUACCT)
12 SUBSYS(cpuacct)
13 #endif
14
15 #if IS_ENABLED(CONFIG_MEMCG)
16 SUBSYS(memory)
17 #endif
```

根据 SUBSYS 的定义，SUBSYS(cpu) 其实是 [cpu\_cgrp\_id] = &cpu\_cgrp\_subsys，而 SUBSYS(memory) 其实是 [memory\_cgrp\_id] = &memory\_cgrp\_subsys。


我们能够找到 cpu\_cgrp\_subsys 和 memory\_cgrp\_subsys 的定义。

```

1  cpuset_cgrp_subsys
2  struct cgroup_subsys cpuset_cgrp_subsys = {
3      .css_alloc      = cpuset_css_alloc,
4      .css_online     = cpuset_css_online,
5      .css_offline    = cpuset_css_offline,
6      .css_free       = cpuset_css_free,
7      .can_attach     = cpuset_can_attach,
8      .cancel_attach  = cpuset_cancel_attach,
9      .attach         = cpuset_attach,
10     .post_attach    = cpuset_post_attach,
11     .bind           = cpuset_bind,
12     .fork           = cpuset_fork,
13     .legacy_cftypes = files,
14     .early_init     = true,
15 };
16
17  cpu_cgrp_subsys
18  struct cgroup_subsys cpu_cgrp_subsys = {
19     .css_alloc      = cpu_cgroup_css_alloc,
20     .css_online     = cpu_cgroup_css_online,
21     .css_released   = cpu_cgroup_css_released,
22     .css_free       = cpu_cgroup_css_free,
23     .fork           = cpu_cgroup_fork,
24     .can_attach     = cpu_cgroup_can_attach,
25     .attach         = cpu_cgroup_attach,
26     .legacy_cftypes = cpu_files,
27     .early_init     = true,
28 };
29
30  memory_cgrp_subsys
31  struct cgroup_subsys memory_cgrp_subsys = {
32     .css_alloc = mem_cgroup_css_alloc,
33     .css_online = mem_cgroup_css_online,
34     .css_offline = mem_cgroup_css_offline,
35     .css_released = mem_cgroup_css_released,
36     .css_free = mem_cgroup_css_free,
37     .css_reset = mem_cgroup_css_reset,
38     .can_attach = mem_cgroup_can_attach,
39     .cancel_attach = mem_cgroup_cancel_attach,
40     .post_attach = mem_cgroup_move_task,
41     .bind = mem_cgroup_bind,
42     .dfl_cftypes = memory_files,
43     .legacy_cftypes = mem_cgroup_legacy_files,
44     .early_init = 0,
45 };

```

在 `for_each_subsys` 的循环里面，`cgroup_subsys[]` 数组中的每一个 `cgroup_subsys`，都会调用 `cgroup_init_subsys`，对于 `cgroup_subsys` 对于初始化。

 复制代码

```
1 static void __init cgroup_init_subsys(struct cgroup_subsys *ss, bool early)
2 {
3     struct cgroup_subsys_state *css;
4     .....
5     idr_init(&ss->css_idr);
6     INIT_LIST_HEAD(&ss->cfts);
7
8     /* Create the root cgroup state for this subsystem */
9     ss->root = &cgrp_dfl_root;
10    css = ss->css_alloc(cgroup_css(&cgrp_dfl_root.cgrp, ss));
11    .....
12    init_and_link_css(css, ss, &cgrp_dfl_root.cgrp);
13    .....
14    css->id = cgroup_idr_alloc(&ss->css_idr, css, 1, 2, GFP_KERNEL);
15    init_css_set.subsys[ss->id] = css;
16    .....
17    BUG_ON(online_css(css));
18    .....
19 }
```

`cgroup_init_subsys` 里面会做两件事情，一个是调用 `cgroup_subsys` 的 `css_alloc` 函数创建一个 `cgroup_subsys_state`；另外就是调用 `online_css`，也即调用 `cgroup_subsys` 的 `css_online` 函数，激活这个 `cgroup`。

对于 CPU 来讲，`css_alloc` 函数就是 `cpu_cgroup_css_alloc`。这里面会调用 `sched_create_group` 创建一个 `struct task_group`。在这个结构中，第一项就是 `cgroup_subsys_state`，也就是说，`task_group` 是 `cgroup_subsys_state` 的一个扩展，最终返回的是指向 `cgroup_subsys_state` 结构的指针，可以通过强制类型转换变为 `task_group`。

 复制代码

```
1 struct task_group {
2     struct cgroup_subsys_state css;
3
4     #ifdef CONFIG_FAIR_GROUP_SCHED
5     /* schedulable entities of this group on each cpu */
6     struct sched_entity **se;
7     /* runqueue "owned" by this group on each cpu */
```


```

8      struct cfs_rq **cfs_rq;
9      unsigned long shares;
10
11 #ifdef CONFIG_SMP
12      atomic_long_t load_avg ____cacheline_aligned;
13 #endif
14 #endif
15
16      struct rcu_head rcu;
17      struct list_head list;
18
19      struct task_group *parent;
20      struct list_head siblings;
21      struct list_head children;
22
23      struct cfs_bandwidth cfs_bandwidth;
24 };

```

在 `task_group` 结构中，有一个成员是 `sched_entity`，前面我们讲进程调度的时候，遇到过它。它是调度的实体，也即这一个 `task_group` 也是一个调度实体。

接下来，`online_css` 会被调用。对于 CPU 来讲，`online_css` 调用的是 `cpu_cgroup_css_online`。它会调用 `sched_online_group->online_fair_sched_group`。

 复制代码


```

1 void online_fair_sched_group(struct task_group *tg)
2 {
3     struct sched_entity *se;
4     struct rq *rq;
5     int i;
6
7     for_each_possible_cpu(i) {
8         rq = cpu_rq(i);
9         se = tg->se[i];
10        update_rq_clock(rq);
11        attach_entity_cfs_rq(se);
12        sync_throttle(tg, i);
13    }
14 }

```


在这里面，对于每一个 CPU，取出每个 CPU 的运行队列 `rq`，也取出 `task_group` 的 `sched_entity`，然后通过 `attach_entity_cfs_rq` 将 `sched_entity` 添加到运行队列中。

对于内存来讲，css\_alloc 函数就是 mem\_cgroup\_css\_alloc。这里面会调用 mem\_cgroup\_alloc，创建一个 struct mem\_cgroup。在这个结构中，第一项就是 cgroup\_subsys\_state，也就是说，mem\_cgroup 是 cgroup\_subsys\_state 的一个扩展，最终返回的是指向 cgroup\_subsys\_state 结构的指针，我们可以通过强制类型转换变为 mem\_cgroup。

 复制代码

```
1 struct mem_cgroup {
2     struct cgroup_subsys_state css;
3
4     /* Private memcg ID. Used to ID objects that outlive the cgroup */
5     struct mem_cgroup_id id;
6
7     /* Accounted resources */
8     struct page_counter memory;
9     struct page_counter swap;
10
11     /* Legacy consumer-oriented counters */
12     struct page_counter memsw;
13     struct page_counter kmem;
14     struct page_counter tcpmem;
15
16     /* Normal memory consumption range */
17     unsigned long low;
18     unsigned long high;
19
20     /* Range enforcement for interrupt charges */
21     struct work_struct high_work;
22
23     unsigned long soft_limit;
24
25     .....
26     int      swappiness;
27     .....
28     /*
29      * percpu counter.
30      */
31     struct mem_cgroup_stat_cpu __percpu *stat;
32
33     int last_scanned_node;
34
35     /* List of events which userspace want to receive */
36     struct list_head event_list;
37     spinlock_t event_list_lock;
38
39     struct mem_cgroup_per_node *nodeinfo[0];
40     /* WARNING: nodeinfo must be the last member here */
41 };
```


在 `cgroup_init` 函数中，`cgroup` 的初始化还做了一件很重要的事情，它会调用 `cgroup_init_cftypes(NULL, cgroup1_base_files)`，来初始化对于 `cgroup` 文件类型 `cftype` 的操作函数，也就是将 `struct kernfs_ops *kf_ops` 设置为 `cgroup_kf_ops`。

 复制代码

```
1 struct cftype cgroup1_base_files[] = {
2     .....
3     {
4         .name = "tasks",
5         .seq_start = cgroup_pidlist_start,
6         .seq_next = cgroup_pidlist_next,
7         .seq_stop = cgroup_pidlist_stop,
8         .seq_show = cgroup_pidlist_show,
9         .private = CGROUP_FILE_TASKS,
10        .write = cgroup_tasks_write,
11    },
12 }
13
14 static struct kernfs_ops cgroup_kf_ops = {
15     .atomic_write_len      = PAGE_SIZE,
16     .open                  = cgroup_file_open,
17     .release               = cgroup_file_release,
18     .write                 = cgroup_file_write,
19     .seq_start             = cgroup_seqfile_start,
20     .seq_next              = cgroup_seqfile_next,
21     .seq_stop              = cgroup_seqfile_stop,
22     .seq_show              = cgroup_seqfile_show,
23 };
```

在 `cgroup` 初始化完毕之后，接下来就是创建一个 `cgroup` 的文件系统，用了配置和操作 `cgroup`。


`cgroup` 是一种特殊的文件系统。它的定义如下：

 复制代码

```
1 struct file_system_type cgroup_fs_type = {
2     .name = "cgroup",
3     .mount = cgroup_mount,
4     .kill_sb = cgroup_kill_sb,
5     .fs_flags = FS_USERNS_MOUNT,
```

```
6 };
```

当我们 mount 这个 cgroup 文件系统的时候，会调用 `cgroup_mount->cgroup1_mount`。

 复制代码

```
1 struct dentry *cgroup1_mount(struct file_system_type *fs_type, int flags,
2                               void *data, unsigned long magic,
3                               struct cgroup_namespace *ns)
4 {
5     struct super_block *pinned_sb = NULL;
6     struct cgroup_sb_opts opts;
7     struct cgroup_root *root;
8     struct cgroup_subsys *ss;
9     struct dentry *dentry;
10    int i, ret;
11    bool new_root = false;
12    .....
13    root = kzalloc(sizeof(*root), GFP_KERNEL);
14    new_root = true;
15
16    init_cgroup_root(root, &opts);
17
18    ret = cgroup_setup_root(root, opts.subsys_mask, PERCPU_REF_INIT_DEAD);
19    .....
20    dentry = cgroup_do_mount(&cgroup_fs_type, flags, root,
21                             CGROUP_SUPER_MAGIC, ns);
22    .....
23    return dentry;
24 }
```

cgroup 被组织成为树形结构，因而有 `cgroup_root`。`init_cgroup_root` 会初始化这个 `cgroup_root`。`cgroup_root` 是 cgroup 的根，它有一个成员 `kf_root`，是 cgroup 文件系统的根 `struct kernfs_root`。`kernfs_create_root` 就是用来创建这个 `kernfs_root` 结构的。

 复制代码

```
1 int cgroup_setup_root(struct cgroup_root *root, u16 ss_mask, int ref_flags)
2 {
3     LIST_HEAD(tmp_links);
4     struct cgroup *root_cgrp = &root->cgrp;
5     struct kernfs_syscall_ops *kf_sops;
```






```

19 {
20     struct kernfs_node *kn;
21     unsigned flags;
22     int rc;
23
24     flags = KERNFS_FILE;
25
26     kn = kernfs_new_node(parent, name, (mode & S_IALLUGO) | S_IFREG, flags);
27
28     kn->attr.ops = ops;
29     kn->attr.size = size;
30     kn->ns = ns;
31     kn->priv = priv;
32     .....
33     rc = kernfs_add_one(kn);
34     .....
35     return kn;
36 }

```

从 `cgroup_setup_root` 返回后，接下来，在 `cgroup1_mount` 中，要做的一件事情是 `cgroup_do_mount`，调用 `kernfs_mount` 真的去 mount 这个文件系统，返回一个普通的文件系统都认识的 `dentary`。这种特殊的文件系统对应的文件操作函数为 `kernfs_file_fops`。

 复制代码

```

1 const struct file_operations kernfs_file_fops = {
2     .read          = kernfs_fop_read,
3     .write         = kernfs_fop_write,
4     .llseek       = generic_file_llseek,
5     .mmap         = kernfs_fop_mmap,
6     .open         = kernfs_fop_open,
7     .release      = kernfs_fop_release,
8     .poll         = kernfs_fop_poll,
9     .fsync        = noop_fsync,
10 };

```


当我们要写入一个 `CGroup` 文件来设置参数的时候，根据文件系统的操作，`kernfs_fop_write` 会被调用，在这里面会调用 `kernfs_ops` 的 `write` 函数，根据上面的定义为 `cgroup_file_write`，在这里会调用 `cftype` 的 `write` 函数。对于 CPU 和内存的 `write` 函数，有以下不同的定义。

```

1 static struct cftype cpu_files[] = {
2     #ifdef CONFIG_FAIR_GROUP_SCHED
3     {
4         .name = "shares",
5         .read_u64 = cpu_shares_read_u64,
6         .write_u64 = cpu_shares_write_u64,
7     },
8     #endif
9     #ifdef CONFIG_CFS_BANDWIDTH
10    {
11        .name = "cfs_quota_us",
12        .read_s64 = cpu_cfs_quota_read_s64,
13        .write_s64 = cpu_cfs_quota_write_s64,
14    },
15    {
16        .name = "cfs_period_us",
17        .read_u64 = cpu_cfs_period_read_u64,
18        .write_u64 = cpu_cfs_period_write_u64,
19    },
20 }
21
22
23 static struct cftype mem_cgroup_legacy_files[] = {
24     {
25         .name = "usage_in_bytes",
26         .private = MEMFILE_PRIVATE(_MEM, RES_USAGE),
27         .read_u64 = mem_cgroup_read_u64,
28     },
29     {
30         .name = "max_usage_in_bytes",
31         .private = MEMFILE_PRIVATE(_MEM, RES_MAX_USAGE),
32         .write = mem_cgroup_reset,
33         .read_u64 = mem_cgroup_read_u64,
34     },
35     {
36         .name = "limit_in_bytes",
37         .private = MEMFILE_PRIVATE(_MEM, RES_LIMIT),
38         .write = mem_cgroup_write,
39         .read_u64 = mem_cgroup_read_u64,
40     },
41     {
42         .name = "soft_limit_in_bytes",
43         .private = MEMFILE_PRIVATE(_MEM, RES_SOFT_LIMIT),
44         .write = mem_cgroup_write,
45         .read_u64 = mem_cgroup_read_u64,
46     },
47 }

```

如果设置的是 `cpu.shares`，则调用 `cpu_shares_write_u64`。在这里面，`task_group` 的 `shares` 变量更新了，并且更新了 CPU 队列上的调度实体。

 复制代码

```
1 int sched_group_set_shares(struct task_group *tg, unsigned long shares)
2 {
3     int i;
4
5     shares = clamp(shares, scale_load(MIN_SHARES), scale_load(MAX_SHARES));
6
7     tg->shares = shares;
8     for_each_possible_cpu(i) {
9         struct rq *rq = cpu_rq(i);
10        struct sched_entity *se = tg->se[i];
11        struct rq_flags rf;
12
13        update_rq_clock(rq);
14        for_each_sched_entity(se) {
15            update_load_avg(se, UPDATE_TG);
16            update_cfs_shares(se);
17        }
18    }
19    .....
20 }
```

但是这个时候别忘了，我们还没有将 CPU 的文件夹下面的 `tasks` 文件写入进程号呢。写入一个进程号到 `tasks` 文件里面，按照 `cgroup1_base_files` 里面的定义，我们应该调用 `cgroup_tasks_write`。

接下来的调用链为：`cgroup_tasks_write`->`__cgroup_procs_write`->`cgroup_attach_task`->`cgroup_migrate`->`cgroup_migrate_execute`。将这个进程和一个 `cgroup` 关联起来，也即将这个进程迁移到这个 `cgroup` 下面。

 复制代码


```
1 static int cgroup_migrate_execute(struct cgroup_mgctx *mgctx)
2 {
3     struct cgroup_taskset *tset = &mgctx->tset;
4     struct cgroup_subsys *ss;
5     struct task_struct *task, *tmp_task;
6     struct css_set *cset, *tmp_cset;
7     .....
8     if (tset->nr_tasks) {
```

```

9         do_each_subsys_mask(ss, ssid, mgctx->ss_mask) {
10             if (ss->attach) {
11                 tset->ssid = ssid;
12                 ss->attach(tset);
13             }
14         } while_each_subsys_mask();
15     }
16     .....
17 }

```

每一个 cgroup 子系统会调用相应的 attach 函数。而 CPU 调用的是 `cpu_cgroup_attach-> sched_move_task-> sched_change_group`。

 复制代码

```

1 static void sched_change_group(struct task_struct *tsk, int type)
2 {
3     struct task_group *tg;
4
5     tg = container_of(task_css_check(tsk, cpu_cgrp_id, true),
6                       struct task_group, css);
7     tg = autogroup_task_group(tsk, tg);
8     tsk->sched_task_group = tg;
9
10 #ifdef CONFIG_FAIR_GROUP_SCHED
11     if (tsk->sched_class->task_change_group)
12         tsk->sched_class->task_change_group(tsk, type);
13     else
14 #endif
15         set_task_rq(tsk, task_cpu(tsk));
16 }

```

在 `sched_change_group` 中设置这个进程以这个 `task_group` 的方式参与调度，从而使得上面的 `cpu.shares` 起作用。

对于内存来讲，写入内存的限制使用函数 `mem_cgroup_write->mem_cgroup_resize_limit` 来设置 `struct mem_cgroup` 的 `memory.limit` 成员。

在进程执行过程中，申请内存的时候，我们会调用 `handle_pte_fault->do_anonymous_page()->mem_cgroup_try_charge()`。

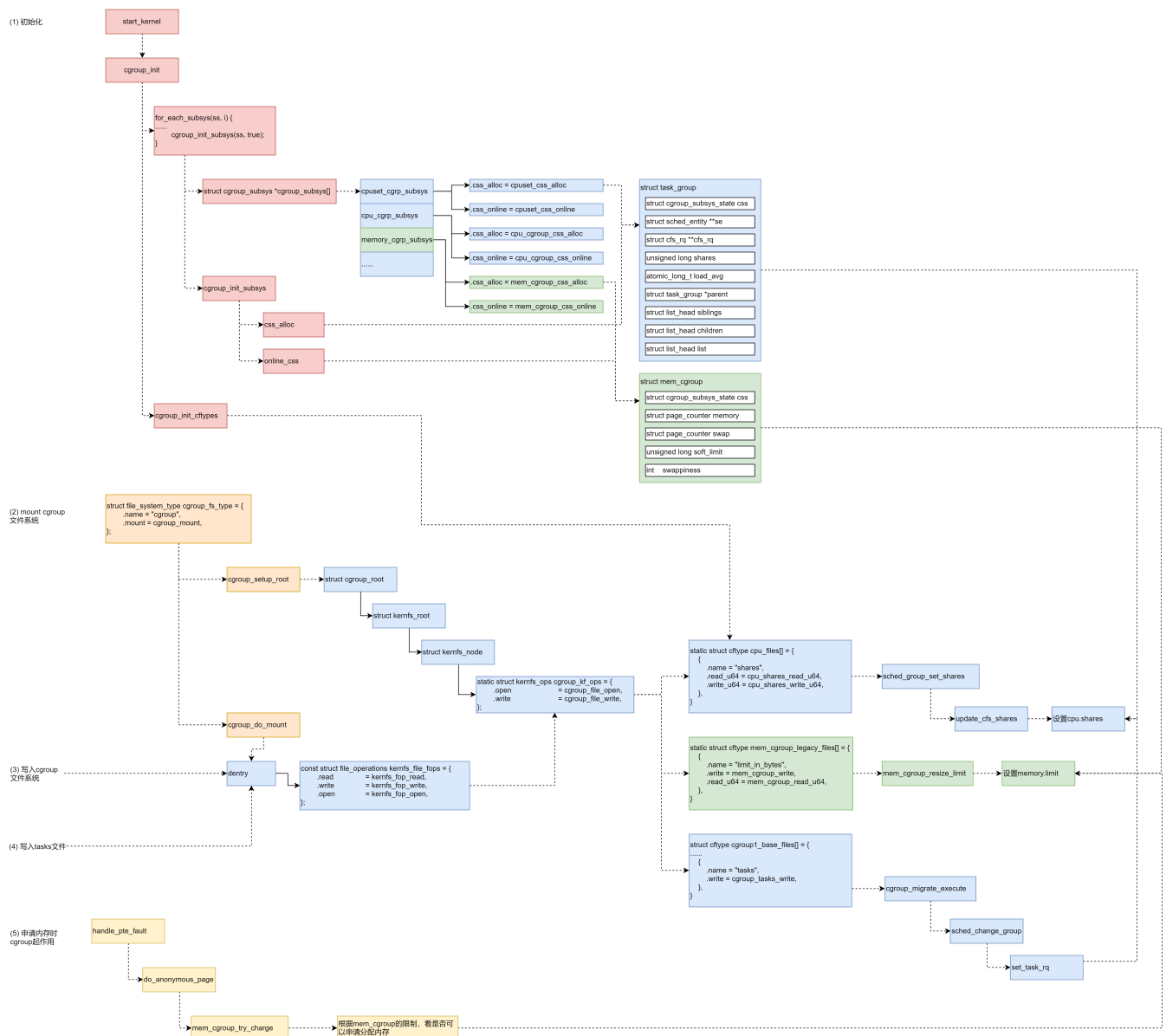
```
1 int mem_cgroup_try_charge(struct page *page, struct mm_struct *mm,
2                           gfp_t gfp_mask, struct mem_cgroup **memcgp,
3                           bool compound)
4 {
5     struct mem_cgroup *memcg = NULL;
6     .....
7     if (!memcg)
8         memcg = get_mem_cgroup_from_mm(mm);
9
10    ret = try_charge(memcg, gfp_mask, nr_pages);
11    .....
12 }
```

在 `mem_cgroup_try_charge` 中，先是调用 `get_mem_cgroup_from_mm` 获得这个进程对应的 `mem_cgroup` 结构，然后在 `try_charge` 中，根据 `mem_cgroup` 的限制，看是否可以申请分配内存。

至此，cgroup 对于内存的限制才真正起作用。

## 总结时刻

内核中 cgroup 的工作机制，我们在这里总结一下。



第一步，系统初始化的时候，初始化 cgroup 的各个子系统的操作函数，分配各个子系统的的数据结构。

第二步，mount cgroup 文件系统，创建文件系统的树形结构，以及操作函数。

第三步，写入 cgroup 文件，设置 cpu 或者 memory 的相关参数，这个时候文件系统的操作函数会调用到 cgroup 子系统的操作函数，从而将参数设置到 cgroup 子系统的的数据结构中。

第四步，写入 tasks 文件，将进程交给某个 cgroup 进行管理，因为 tasks 文件也是一个 cgroup 文件，统一会调用文件系统的操作函数进而调用 cgroup 子系统的操作函数，将 cgroup 子系统的的数据结构和进程关联起来。

第五步，对于 cpu 来讲，会修改 scheduled entity，放入相应的队列里面去，从而下次调度的时候就起作用了。对于内存的 cgroup 设定，只有在申请内存的时候才起作用。

## 课堂练习

这里我们用 cgroup 限制了 CPU 和内存，如何限制网络呢？给你一个提示 tc，请你研究一下。

欢迎留言和我分享你的疑惑和见解，也欢迎收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

 极客时间

# 趣谈 Linux 操作系统

## 像故事一样的操作系统入门课

刘超

网易杭州研究院  
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 57 | Namespace技术：内部创业公司应该独立运营

下一篇 59 | 数据中心操作系统：上市敲钟

## 精选留言 (4)

 写留言



行者

2019-08-10



老师，麻烦讲下华为鸿蒙系统，它和linux区别与联系是什么？



👍 1



**Zain Lau**

2019-08-09

二十天闭关冲北邮

展开 ▾



👍 1



**许童童**

2019-08-09

跟着老师一起精进。

展开 ▾



**安排**

2019-08-09

Cgroup文件系统是只存在内存中吗？每一次设置之后在掉电后是不是就消失了？

