

57 | Namespace技术：内部创业公司应该独立运营

2019-08-07 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 12:34 大小 11.52M



上一节我们讲了 Docker 的基本原理，今天我们来看一下，“看起来隔离的”技术 namespace 在内核里面是如何工作的。

既然容器是一种类似公司内部创业的技术，我们可以设想一下，如果一个创新项目要独立运营，应该成立哪些看起来独立的组织和部门呢？

首先是**用户管理**，咱们这个小分队应该有自己独立的用户和组管理体系，公司里面并不是任何人都知道我们在做什么。

其次是**项目管理**，咱们应该有自己独立的项目管理体系，不能按照大公司的来。

然后是**档案管理**，咱们这个创新项目的资料一定要保密，要不然创意让人家偷走了可不好。

最后就是**合作部**，咱们这个小分队还是要和公司其他部门或者其他公司合作的，所以需要一个人外向的人来干这件事情。

对应到容器技术，为了隔离不同类型的资源，Linux 内核里面实现了以下几种不同类型的 namespace。

UTS，对应的宏为 CLONE_NEWUTS，表示不同的 namespace 可以配置不同的 hostname。


User，对应的宏为 CLONE_NEWUSER，表示不同的 namespace 可以配置不同的用户和组。

Mount，对应的宏为 CLONE_NEWNS，表示不同的 namespace 的文件系统挂载点是隔离的

PID，对应的宏为 CLONE_NEWPID，表示不同的 namespace 有完全独立的 pid，也即一个 namespace 的进程和另一个 namespace 的进程，pid 可以是一样的，但是代表不同的进程。


Network，对应的宏为 CLONE_NEWNET，表示不同的 namespace 有独立的网络协议栈。

还记得咱们启动的那个容器吗？

 复制代码

```
1 # docker ps
2 CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
3 f604f0e34bc2        testnginx:1        "/bin/sh -c 'nginx -..." 17 hours ago       Up
```

我们可以看这个容器对应的 entrypoint 的 pid。通过 docker inspect 命令，可以看到，进程号为 58212。

 复制代码

```
1 [root@deployer ~]# docker inspect f604f0e34bc2
2 [
3   {
4     "Id": "f604f0e34bc263bc32ba683d97a1db2a65de42ab052da16df3c7811ad07f0dc3",
5     "Created": "2019-07-15T17:43:44.158300531Z",
6     "Path": "/bin/sh",
```

```

7      "Args": [
8          "-c",
9          "nginx -g \"daemon off;\""
10     ],
11     "State": {
12         "Status": "running",
13         "Running": true,
14         "Pid": 58212,
15         "ExitCode": 0,
16         "StartedAt": "2019-07-15T17:43:44.651756682Z",
17         "FinishedAt": "0001-01-01T00:00:00Z"
18     },
19     .....
20     "Name": "/youthful_torvalds",
21     "RestartCount": 0,
22     "Driver": "overlay2",
23     "Platform": "linux",
24     "HostConfig": {
25         "NetworkMode": "default",
26         "PortBindings": {
27             "80/tcp": [
28                 {
29                     "HostIp": "",
30                     "HostPort": "8081"
31                 }
32             ]
33         },
34         .....
35     },
36     "Config": {
37         "Hostname": "f604f0e34bc2",
38         "ExposedPorts": {
39             "80/tcp": {}
40         },
41         "Image": "testnginx:1",
42         "Entrypoint": [
43             "/bin/sh",
44             "-c",
45             "nginx -g \"daemon off;\""
46         ],
47     },
48     "NetworkSettings": {
49         "Bridge": "",
50         "SandboxID": "7fd3eb469578903b66687090e512958658ae28d17bce1a7cee2da3148d1df",
51         "Ports": {
52             "80/tcp": [
53                 {
54                     "HostIp": "0.0.0.0",
55                     "HostPort": "8081"
56                 }
57             ]
58         },

```

```

59         "Gateway": "172.17.0.1",
60         "IPAddress": "172.17.0.3",
61         "IPPrefixLen": 16,
62         "MacAddress": "02:42:ac:11:00:03",
63         "Networks": {
64             "bridge": {
65                 "NetworkID": "c8eef1603afb399bf17af154be202fd1e543d3772cc83ef4a1ca3",
66                 "EndpointID": "8d9bb18ca57889112e758ede193d2cfb45cbf794c9d952819763",
67                 "Gateway": "172.17.0.1",
68                 "IPAddress": "172.17.0.3",
69                 "IPPrefixLen": 16,
70                 "MacAddress": "02:42:ac:11:00:03",
71             }
72         }
73     }
74 }
75 ]

```

如果我们用 `ps` 查看机器上的 `nginx` 进程，可以看到 `master` 和 `worker`，`worker` 的父进程是 `master`。

 复制代码

```

1 # ps -ef |grep nginx
2 root      58212 58195  0 01:43 ?        00:00:00 /bin/sh -c nginx -g "daemon off;"
3 root      58244 58212  0 01:43 ?        00:00:00 nginx: master process nginx -g daemon o
4 33         58250 58244  0 01:43 ?        00:00:00 nginx: worker process
5 33         58251 58244  0 01:43 ?        00:00:05 nginx: worker process
6 33         58252 58244  0 01:43 ?        00:00:05 nginx: worker process
7 33         58253 58244  0 01:43 ?        00:00:05 nginx: worker process

```

在 `/proc/pid/ns` 里面，我们能够看到这个进程所属于的 6 种 namespace。我们拿出两个进程来，应该可以看出来，它们属于同一个 namespace。

 复制代码

```


1 # ls -l /proc/58212/ns
2 lrwxrwxrwx 1 root root 0 Jul 16 19:19 ipc -> ipc:[4026532278]
3 lrwxrwxrwx 1 root root 0 Jul 16 19:19 mnt -> mnt:[4026532276]
4 lrwxrwxrwx 1 root root 0 Jul 16 01:43 net -> net:[4026532281]
5 lrwxrwxrwx 1 root root 0 Jul 16 19:19 pid -> pid:[4026532279]
6 lrwxrwxrwx 1 root root 0 Jul 16 19:19 user -> user:[4026531837]
7 lrwxrwxrwx 1 root root 0 Jul 16 19:19 uts -> uts:[4026532277]
8

```

```
9 # ls -l /proc/58253/ns
10 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 ipc -> ipc:[4026532278]
11 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 mnt -> mnt:[4026532276]
12 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 net -> net:[4026532281]
13 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 pid -> pid:[4026532279]
14 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 user -> user:[4026531837]
15 lrwxrwxrwx 1 33 tape 0 Jul 16 19:20 uts -> uts:[4026532277]
```

接下来，我们来看，如何操作 namespace。这里我们重点关注 pid 和 network。


操作 namespace 的常用指令 **nsenter**，可以用来运行一个进程，进入指定的 namespace。例如，通过下面的命令，我们可以运行 /bin/bash，并且进入 nginx 所在容器的 namespace。

 复制代码

```
1 # nsenter --target 58212 --mount --uts --ipc --net --pid -- env --ignore-environment --
2
3 root@f604f0e34bc2:/# ip addr
4 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen :
5     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
6     inet 127.0.0.1/8 scope host lo
7         valid_lft forever preferred_lft forever
8 23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group (
9     link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
10    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
11        valid_lft forever preferred_lft forever
```


另一个命令是 **unshare**，它会离开当前的 namespace，创建且加入新的 namespace，然后执行参数中指定的命令。

例如，运行下面这行命令之后，pid 和 net 都进入了新的 namespace。

 复制代码

```
1 unshare --mount --ipc --pid --net --mount-proc=/proc --fork /bin/bash
```

如果从 shell 上运行上面这行命令的话，好像没有什么变化，但是因为 pid 和 net 都进入了新的 namespace，所以我们查看进程列表和 ip 地址的时候应该会发现有所不同。


 复制代码

```
1 # ip addr
2 1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
3     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4
5 # ps aux
6 USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
7 root         1   0.0   0.0 115568  2136 pts/0    S    22:55   0:00 /bin/bash
8 root        13   0.0   0.0 155360  1872 pts/0    R+   22:55   0:00 ps aux
```

果真，我们看不到宿主机上的 IP 地址和网卡了，也看不到宿主机上的所有进程了。

另外，我们还可以通过函数操作 namespace。


第一个函数是**clone**，也就是创建一个新的进程，并把它放到新的 namespace 中。

 复制代码

```
1 int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

clone 函数我们原来介绍过。这里面有一个参数 flags，原来我们没有注意它。其实它可以设置为 CLONE_NEWUTS、CLONE_NEWUSER、CLONE_NEWNS、CLONE_NEWPID。CLONE_NEWNET 会将 clone 出来的新进程放到新的 namespace 中。

第二个函数是**setns**，用于将当前进程加入到已有的 namespace 中。


 复制代码

```
1 int setns(int fd, int nstype);
```

其中，fd 指向 /proc/[pid]/ns/ 目录里相应 namespace 对应的文件，表示要加入哪个 namespace。nstype 用来指定 namespace 的类型，可以设置为 CLONE_NEWUTS、

CLONE_NEWUSER、CLONE_NEWNS、CLONE_NEWPID 和 CLONE_NEWNET。

第三个函数是**unshare**，它可以使当前进程退出当前的 namespace，并加入到新创建的 namespace。


 复制代码

```
1 int unshare(int flags);
```

其中，flags 用于指定一个或者多个上面的 CLONE_NEWUTS、CLONE_NEWUSER、CLONE_NEWNS、CLONE_NEWPID 和 CLONE_NEWNET。

clone 和 unshare 的区别是，unshare 是使当前进程加入新的 namespace；clone 是创建一个新的子进程，然后让子进程加入新的 namespace，而当前进程保持不变。

这里我们尝试一下，通过 clone 函数来进入一个 namespace。

 复制代码

```
1 #define _GNU_SOURCE
2 #include <sys/wait.h>
3 #include <sys/utsname.h>
4 #include <sched.h>
5 #include <string.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #define STACK_SIZE (1024 * 1024)
10
11 static int childFunc(void *arg)
12 {
13     printf("In child process.\n");
14     execlp("bash", "bash", (char *) NULL);
15     return 0;
16 }
17
18 int main(int argc, char *argv[])
19 {
20     char *stack;
21     char *stackTop;
22     pid_t pid;
23
24     stack = malloc(STACK_SIZE);
```


```

25     if (stack == NULL)
26     {
27         perror("malloc");
28         exit(1);
29     }
30     stackTop = stack + STACK_SIZE;
31
32     pid = clone(childFunc, stackTop, CLONE_NEWNS|CLONE_NEWPID|CLONE_NEWNET|SIGCHLD, NUL
33     if (pid == -1)
34     {
35         perror("clone");
36         exit(1);
37     }
38     printf("clone() returned %ld\n", (long) pid);
39
40     sleep(1);
41
42     if (waitpid(pid, NULL, 0) == -1)
43     {
44         perror("waitpid");
45         exit(1);
46     }
47     printf("child has terminated\n");
48     exit(0);
49 }

```

在上面的代码中，我们调用 clone 的时候，给的参数是 CLONE_NEWNS|CLONE_NEWPID|CLONE_NEWNET，也就是说，我们会进入一个新的 pid、network，以及 mount 的 namespace。

如果我们编译运行它，可以得到下面的结果。

 复制代码

```

1 # echo $$
2 64267
3
4 # ps aux | grep bash | grep -v grep
5 root      64267  0.0  0.0 115572  2176 pts/0    Ss   16:53   0:00 -bash
6
7 # ./a.out
8 clone() returned 64360
9 In child process.
10
11 # echo $$
12 1

```



```

13
14 # ip addr
15 1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
16     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
17
18 # exit
19 exit
20 child has terminated
21
22 # echo $$
23 64267

```

通过 `echo $$`，我们可以得到当前 `bash` 的进程号。一旦运行了上面的程序，我们会进入一个新的 `pid` 的 `namespace`。

当我们再次 `echo`


的时候，就会发现当前 *bash* 的进程号变成了 `1`。上面的程序运行了一个新的 *bash*，它在一

的时候，就可以得到原来进程号。

`clone` 系统调用我们在[进程的创建](#)那一节解析过，当时我们没有看关于 `namespace` 的代码，现在我们就来看一看，`namespace` 在内核做了哪些事情。

在内核里面，`clone` 会调用 `_do_fork->copy_process->copy_namespaces`，也就是说，在创建子进程的时候，有一个机会可以复制和设置 `namespace`。

`namespace` 是在哪里定义的呢？在每一个进程的 `task_struct` 里面，有一个指向 `namespace` 结构体的指针 `nsproxy`。

 复制代码

```

1 struct task_struct {
2     .....
3     /* Namespaces: */
4     struct nsproxy          *nsproxy;
5     .....
6 }
7
8 /*
9  * A structure to contain pointers to all per-process

```


```

10  * namespaces - fs (mount), uts, network, sysvipc, etc.
11  *
12  * The pid namespace is an exception -- it's accessed using
13  * task_active_pid_ns. The pid namespace here is the
14  * namespace that children will use.
15  */
16  struct nsproxy {
17      atomic_t count;
18      struct uts_namespace *uts_ns;
19      struct ipc_namespace *ipc_ns;
20      struct mnt_namespace *mnt_ns;
21      struct pid_namespace *pid_ns_for_children;
22      struct net            *net_ns;
23      struct cgroup_namespace *cgroup_ns;
24  };

```

我们可以看到在 struct nsproxy 结构里面，有我们上面讲过的各种 namespace。

在系统初始化的时候，有一个默认的 init_nsproxy。


 复制代码

```

1  struct nsproxy init_nsproxy = {
2      .count                = ATOMIC_INIT(1),
3      .uts_ns               = &init_uts_ns,
4  #if defined(CONFIG_POSIX_MQUEUE) || defined(CONFIG_SYSVIPC)
5      .ipc_ns               = &init_ipc_ns,
6  #endif
7      .mnt_ns               = NULL,
8      .pid_ns_for_children  = &init_pid_ns,
9  #ifdef CONFIG_NET
10     .net_ns                = &init_net,
11 #endif
12 #ifdef CONFIG_CGROUPS
13     .cgroup_ns              = &init_cgroup_ns,
14 #endif
15 };

```

下面，我们来看 copy_namespaces 的实现。

 复制代码

```

1  /*
2   * called from clone. This now handles copy for nsproxy and all

```

```

3  * namespaces therein.
4  */
5  int copy_namespaces(unsigned long flags, struct task_struct *tsk)
6  {
7      struct nsproxy *old_ns = tsk->nsproxy;
8      struct user_namespace *user_ns = task_cred_xxx(tsk, user_ns);
9      struct nsproxy *new_ns;
10
11     if (likely(!(flags & (CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWIPC |
12                          CLONE_NEWPID | CLONE_NEWNET |
13                          CLONE_NEWCGROUP)))) {
14         get_nsproxy(old_ns);
15         return 0;
16     }
17
18     if (!ns_capable(user_ns, CAP_SYS_ADMIN))
19         return -EPERM;
20     .....
21     new_ns = create_new_namespaces(flags, tsk, user_ns, tsk->fs);
22
23     tsk->nsproxy = new_ns;
24     return 0;
25 }

```

如果 clone 的参数里面没有 CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWNET | CLONE_NEWCGROUP，就返回原来的 namespace，调用 get_nsproxy。

接着，我们调用 create_new_namespaces。

 复制代码

```

1  /*
2   * Create new nsproxy and all of its the associated namespaces.
3   * Return the newly created nsproxy. Do not attach this to the task,
4   * leave it to the caller to do proper locking and attach it to task.
5   */
6  static struct nsproxy *create_new_namespaces(unsigned long flags,
7      struct task_struct *tsk, struct user_namespace *user_ns,
8      struct fs_struct *new_fs)
9  {
10     struct nsproxy *new_nsp;
11
12     new_nsp = create_nsproxy();
13     .....
14     new_nsp->mnt_ns = copy_mnt_ns(flags, tsk->nsproxy->mnt_ns, user_ns, new_fs);

```


```

15 .....
16         new_nsp->uts_ns = copy_utsname(flags, user_ns, tsk->nsproxy->uts_ns);
17 .....
18         new_nsp->ipc_ns = copy_ipcs(flags, user_ns, tsk->nsproxy->ipc_ns);
19 .....
20         new_nsp->pid_ns_for_children =
21             copy_pid_ns(flags, user_ns, tsk->nsproxy->pid_ns_for_children);
22 .....
23         new_nsp->cgroup_ns = copy_cgroup_ns(flags, user_ns,
24                                             tsk->nsproxy->cgroup_ns);
25 .....
26         new_nsp->net_ns = copy_net_ns(flags, user_ns, tsk->nsproxy->net_ns);
27 .....
28         return new_nsp;
29 .....
30 }

```

在 `create_new_namespaces` 中，我们可以看到对于各种 namespace 的复制。

我们来看 `copy_pid_ns` 对于 pid namespace 的复制。

 复制代码


```

1 struct pid_namespace *copy_pid_ns(unsigned long flags,
2     struct user_namespace *user_ns, struct pid_namespace *old_ns)
3 {
4     if (!(flags & CLONE_NEWPID))
5         return get_pid_ns(old_ns);
6     if (task_active_pid_ns(current) != old_ns)
7         return ERR_PTR(-EINVAL);
8     return create_pid_namespace(user_ns, old_ns);
9 }

```

在 `copy_pid_ns` 中，如果没有设置 `CLONE_NEWPID`，则返回老的 pid namespace；如果设置了，就调用 `create_pid_namespace`，创建新的 pid namespace。

我们再来看 `copy_net_ns` 对于 network namespace 的复制。

 复制代码

```

1 struct net *copy_net_ns(unsigned long flags,
2     struct user_namespace *user_ns, struct net *old_net)

```

```

3 {
4     struct ucounts *ucounts;
5     struct net *net;
6     int rv;
7
8     if (!(flags & CLONE_NEWNET))
9         return get_net(old_net);
10
11     ucounts = inc_net_namespaces(user_ns);
12 .....
13     net = net_alloc();
14 .....
15     get_user_ns(user_ns);
16     net->ucounts = ucounts;
17     rv = setup_net(net, user_ns);
18 .....
19     return net;
20 }

```

在这里面，我们需要判断，如果 flags 中不包含 CLONE_NEWNET，也就是不会创建一个新的 network namespace，则返回 old_net；否则需要新建一个 network namespace。

然后，copy_net_ns 会调用 net = net_alloc()，分配一个新的 struct net 结构，然后调用 setup_net 对新分配的 net 结构进行初始化，之后调用 list_add_tail_rcu，将新建的 network namespace，添加到全局的 network namespace 列表 net_namespace_list 中。

我们来看一下 setup_net 的实现。

 复制代码

```

1 /*
2  * setup_net runs the initializers for the network namespace object.
3  */
4 static __net_init int setup_net(struct net *net, struct user_namespace *user_ns)
5 {
6     /* Must be called with net_mutex held */
7     const struct pernet_operations *ops, *saved_ops;
8     LIST_HEAD(net_exit_list);
9
10     atomic_set(&net->count, 1);
11     refcount_set(&net->passive, 1);
12     net->dev_base_seq = 1;
13     net->user_ns = user_ns;
14     idr_init(&net->netns_ids);

```

```

15     spin_lock_init(&net->nsid_lock);
16
17     list_for_each_entry(ops, &pernet_list, list) {
18         error = ops_init(ops, net);
19         .....
20     }
21     .....
22 }

```

在 `setup_net` 中，这里面有一个循环 `list_for_each_entry`，对于 `pernet_list` 的每一项 `struct pernet_operations`，运行 `ops_init`，也就是调用 `pernet_operations` 的 `init` 函数。

这个 `pernet_list` 是怎么来的呢？在网络设备初始化的时候，我们要调用 `net_dev_init` 函数，这里面有下面的代码。

 复制代码

```

1 register_pernet_device(&loopback_net_ops)
2
3 int register_pernet_device(struct pernet_operations *ops)
4 {
5     int error;
6     mutex_lock(&net_mutex);
7     error = register_pernet_operations(&pernet_list, ops);
8     if (!error && (first_device == &pernet_list))
9         first_device = &ops->list;
10    mutex_unlock(&net_mutex);
11    return error;
12 }
13
14 struct pernet_operations __net_initdata loopback_net_ops = {
15     .init = loopback_net_init,
16 };

```

`register_pernet_device` 函数注册了一个 `loopback_net_ops`，在这里面，把 `init` 函数设置为 `loopback_net_init`。

 复制代码

```

1 static __net_init int loopback_net_init(struct net *net)
2 {

```

```
3     struct net_device *dev;
4     dev = alloc_netdev(0, "lo", NET_NAME_UNKNOWN, loopback_setup);
5     .....
6     dev_net_set(dev, net);
7     err = register_netdev(dev);
8     .....
9     net->loopback_dev = dev;
10    return 0;
11    .....
12 }
```

在 `loopback_net_init` 函数中，我们会创建并且注册一个名字为"lo"的 `struct net_device`。注册完之后，在这个 namespace 里面就会出现一个这样的网络设备，称为 `loopback` 网络设备。

这就是为什么上面的实验中，创建出的新的 network namespace 里面有一个 lo 网络设备。

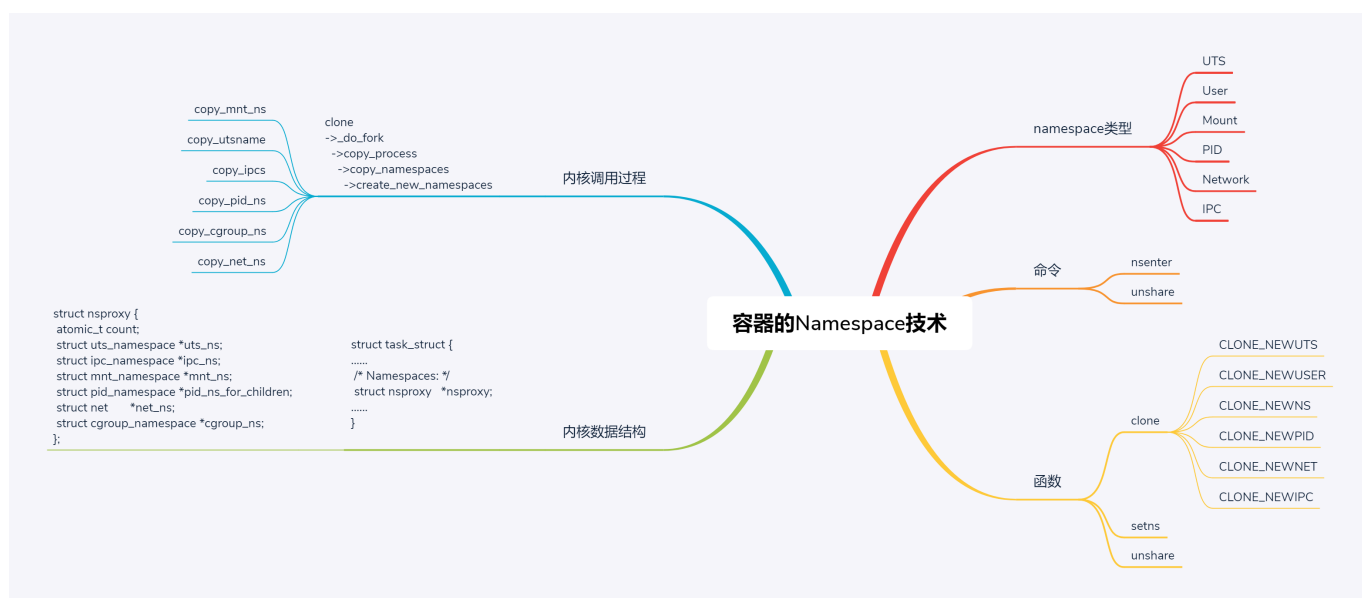
总结时刻

这一节我们讲了 namespace 相关的技术，有六种类型，分别是 UTS、User、Mount、Pid、Network 和 IPC。

还有两个常用的命令 `nsenter` 和 `unshare`，主要用于操作 Namespace，有三个常用的函数 `clone`、`setns` 和 `unshare`。

在内核里面，对于任何一个进程 `task_struct` 来讲，里面都会有一个成员 `struct nsproxy`，用于保存 namespace 相关信息，里面有 `struct uts_namespace`、`struct ipc_namespace`、`struct mnt_namespace`、`struct pid_namespace`、`struct net *net_ns` 和 `struct cgroup_namespace *cgroup_ns`。

创建 namespace 的时候，我们在内核中会调用 `copy_namespaces`，调用顺序依次是 `copy_mnt_ns`、`copy_utsname`、`copy_ipcs`、`copy_pid_ns`、`copy_cgroup_ns` 和 `copy_net_ns`，来复制 namespace。



课堂练习

网络的 Namespace 有一个非常好的命令 `ip netns`。请你研究一下这个命令，并且创建一个容器，用这个命令查看网络 namespace。

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。



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。

上一篇 56 | 容器：大公司为保持创新，鼓励内部创业

下一篇 58 | CGroup技术：内部创业公司应该独立核算成本

精选留言 (2)

写留言



许童童

2019-08-07

这一讲让我对namespace在Docker中起什么作用的理解更深入了。



安排

2019-08-07

越看越有意思，逐渐深入

展开 ▾

