安全上下文

我们有时候在运行一个容器的时候，可能需要使用 sysctl 命令来修改内核参数，比如 net 、vm 、kernel 等参数，但是 systcl 需要容器拥有超级权限，才可以使用，在 Docker 容器启动的时候我们可以加上 --privileged 参数来使用特权模式。那么在 Kubernetes 中应该如何来使用呢？

这个时候我们就需要使用到 Kubernetes 中的 Security Context，也就是常说的安全上下文，主要是来限制容器非法操作宿主节点的系统级别的内容，使得节点的系统或者节点上其他容器组受到影响。Kubernetes 提供了三种配置安全上下文级别的方法：

- Container-level Security Context：仅应用到指定的容器
- Pod-level Security Context：应用到 Pod 内所有容器以及 Volume
- Pod Security Policies（PSP，废弃）：应用到集群内部所有 Pod 以及 Volume

我们可以用如下几种方式来设置 Security Context：

- 访问权限控制：根据用户 ID（UID）和组 ID（GID）来限制对资源（比如：文件）的访问权限
- Security Enhanced Linux（SELinux）：为对象分配 SELinux 标签
- 以 privileged（特权）模式运行
- Linux Capabilities：给某个特定的进程超级权限，而不用给 root 用户所有的 privileged 权限
- AppArmor：使用程序文件来限制单个程序的权限
- Seccomp：过滤容器中进程的系统调用（system call）
- AllowPrivilegeEscalation（允许特权扩大）：此项配置是一个布尔值，定义了一个进程是否可以比其父进程获得更多的特权，直接效果是，容器的进程上是否被设置 no_new_privs 标记。当出现如下情况时，AllowPrivilegeEscalation 的值始终为 true：
  - 容器以 privileged 模式运行
  - 容器拥有 CAP_SYS_ADMIN 的 Linux Capability

为 Pod 设置 Security Context

我们只需要在 Pod 定义的资源清单文件中添加 securityContext 字段，就可以为 Pod 指定安全上下文相关的设定，通过该字段指定的内容将会对当前 Pod 中的所有容器生效。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-pod-demo
spec:
  volumes:
    - name: sec-ctx-vol
      emptyDir: {}
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
    - name: sec-ctx-demo
      image: busybox
      command: ["sh", "-c", "sleep 60m"]
```

```
    volumeMounts:
      - name: sec-ctx-vol
        mountPath: /pod/demo
    securityContext:
      allowPrivilegeEscalation: false
```

在当前资源清单文件中我们在 Pod 下面添加了 securityContext 字段，其中：

- runAsUser 字段指定了该 Pod 中所有容器的进程都以 UID 1000 的身份运行
- runAsGroup 字段指定了该 Pod 中所有容器的进程都以 GID 3000 的身份运行
  - 如果省略该字段，容器进程的 GID 为 root(0)
  - 容器中创建的文件，其所有者为 userID 1000，groupID 3000
- fsGroup 字段指定了该 Pod 的 fsGroup 为 2000
  - 数据卷（对应挂载点 /pod/demo 的数据卷为 sec-ctx-demo） 的所有者以及在该数据卷下创建的任何文件，其 GID 都为 2000

下表是我们常用的一些 securityContext 字段设置内容介绍：

字段描述	字段名	字段说明
非Root	runAsNonRoot	如果为 true，则 kubernetes 在运行容器之前将执行检查，以确保容器进程不是以 root 用户（UID为0）运行，否则将不能启动容器；如果此字段不设置或者为 false，则不执行此检查。该字段也可以在容器的 securityContext 中设定，如果 Pod 和容器的 securityContext 中都设定了这个字段，则对该容器来说以容器中的设置为准。
用户	runAsUser	执行容器 entrypoint 进程的 UID。默认为镜像数据中定义的用户（也就是 Dockerfile 中通过 USER 指令指定）。该字段也可以在容器的 securityContext 中设定，如果 Pod 和容器的 securityContext 中都设定了这个字段，则对该容器来说以容器中的设置为准。
用户组	runAsGroup	执行容器 entrypoint 进程的 GID。默认为 Docker 引擎的 GID。该字段也可以在容器的 securityContext 中设定，如果 Pod 和容器的 securityContext 中都设定了这个字段，则对该容器来说以容器中的设置为准。
数据卷组	fsGroup	一个特殊的补充用户组，将被应用到 Pod 中所有容器。某些类型的数据卷允许 kubelet 修改 数据卷 的 ownership： 1. 修改后的 GID 取值来自于 fsGroup 2. setgid 标记位被设为 1（此时，数据卷中新创建的文件 owner 为 fsGroup） 3. 权限标记将与 rw-rw---- 执行或运算，如果该字段不设置，kubelete 将不会修改数据卷的 ownership 和 permission
补充用户组	supplementalGroups	该列表中的用户组将被作为容器的主 GID 的补充，添加到 Pod 中容器的 enrtypoint 进程。可以不设置。
SeLinux 选项	seLinuxOptions	此字段设定的 SELinux 上下文将被应用到 Pod 中所有容器。如果不指定，容器引擎将为每个容器分配一个随机的 SELinux 上下文。该字段也可以在容器的 securityContext 中设定，如果 Pod 和容器的 securityContext 中都设定了这个字段，则对该容器来说以容器中的设置为准。
sysctl 设置	sysctls	该列表中的所有 sysctl 将被应用到 Pod 中的容器。如果定义了容器引擎不支持的 sysctl，Pod 启动将会失败。

直接创建上面的 Pod 对象：

```
☸ ➜ kubectl apply -f security-context-pod-demo-1.yaml
☸ ➜ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
security-context-pod-demo   1/1     Running   0          6m45s
```

运行完成后，我们可以验证下容器中的进程运行的 ownership：

```
☸ ➜ kubectl exec security-context-pod-demo top
Mem: 3036300K used, 5002596K free, 357880K shrd, 49732K buff, 1650988K cached
CPU:  1.1% usr  1.4% sys  0.0% nic 97.2% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.08 0.19 0.14 4/1085 15
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 1000     S     3836  0.0   2  0.0 sh -c sleep 60m
   10     0 1000     R     3836  0.0   3  0.0 top
```

我们直接运行一个 top 进程，查看容器中的所有正在执行的进程，我们可以看到 USER ID 都为 1000（runAsUser 指定的），然后查看下挂载的数据卷的 ownership：

```
☸ ➜ kubectl exec security-context-pod-demo -- ls -la /pod
total 12
drwxr-xr-x    3 root     root          4096 Jan 14 07:51 .
drwxr-xr-x    1 root     root          4096 Jan 14 07:51 ..
drwxrwsrwx    2 root     2000          4096 Jan 14 07:51 demo
```

因为上面我们指定了 fsGroup=2000，所以声明挂载的数据卷 /pod/demo 的 GID 也变成了 2000，直接调用容器中的 id 命令：

```
☸ ➜ kubectl exec security-context-pod-demo -- id
uid=1000 gid=3000 groups=2000
```

我们可以看到 gid 为 3000，与 runAsGroup 字段所指定的一致，如果 runAsGroup 字段被省略，则 gid 取值为 0（即 root），此时容器中的进程将可以操作 root Group 的文件。

比如我们现在想要去删除容器中的 /tmp 目录就没有权限了，因为该目录的用户和组都是 root，而我们当前要去删除使用的进程的 ID 号就变成了 1000:3000，所以没有权限操作：

```
☸ ➜ kubectl exec security-context-pod-demo -- ls -la /tmp
total 8
drwxrwxrwt    2 root     root          4096 Jan  3 22:48 .
drwxr-xr-x    1 root     root          4096 Jan 14 07:51 ..
☸ ➜ kubectl exec security-context-pod-demo -- rm -rf /tmp
rm: can't remove '/tmp': Permission denied
command terminated with exit code 1
```

为容器设置 Security Context

除了在 Pod 中可以设置安全上下文之外，我们还可以单独为某个容器设置安全上下文，同样也是通过 securityContext 字段设置，当该字段的配置与 Pod 级别的 securityContext 配置相冲突时，容器级别的配置将覆盖 Pod 级别的配置。容器级别的 securityContext 不影响 Pod 中的数据卷，如下资源清单所示：

```yaml
# security-context-pod-demo-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-container-demo
spec:
```

```yaml
  securityContext:
    runAsUser: 1000
  containers:
    - name: sec-ctx-demo
      image: busybox
      command: ["sh", "-c", "sleep 60m"]
      securityContext:
        runAsUser: 2000
        allowPrivilegeEscalation: false
```

直接创建上面的 Pod 对象：

```
☸ ➜ kubectl apply -f security-context-pod-demo-2.yaml
☸ ➜ kubectl get pods
NAME                              READY   STATUS              RESTARTS   AGE
security-context-container-demo   1/1     Running             0          5s
```

同样我们直接执行容器中的 top 命令：

```
☸ ➜ kubectl exec security-context-container-demo -- top
Mem: 3027084K used, 5011812K free, 357880K shrd, 51040K buff, 1654628K cached
CPU:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.35 0.24 0.18 1/1083 15
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 2000     S     3836  0.0   1  0.0 sh -c sleep 60m
   10     0 2000     R     3836  0.0   2  0.0 top
```

容器的进程以 UID 2000 的身份运行，该取值由 spec.containers[*].securityContext.runAsUser 容器组中的字段定义。Pod 中定义的 spec.securityContext.runAsUser 取值 1000 被覆盖。

设置 Linux Capabilities

我们使用 docker/nerdctl run 的时候可以通过 --cap-add 和 --cap-drop 命令来给容器添加 Linux Capabilities。那么在 Kubernetes 下面如何来设置呢？要了解如何设置，首先我们还是需要了解下 Linux Capabilities 是什么？

Linux Capabilities

要了解 Linux Capabilities，这就得从 Linux 的权限控制发展来说明。在 Linux 2.2 版本之前，当内核对进程进行权限验证的时候，Linux 将进程划分为两类：特权进程（UID=0，也就是超级用户）和非特权进程（UID≠0），特权进程拥有所有的内核权限，而非特权进程则根据进程凭证（effective UID, effective GID, supplementary group 等）进行权限检查。

比如我们以常用的 passwd 命令为例，修改用户密码需要具有 root 权限，而普通用户是没有这个权限的。但是实际上普通用户又可以修改自己的密码，这是怎么回事呢？在 Linux 的权限控制机制中，有一类比较特殊的权限设置，比如 SUID(Set User ID on execution)，允许用户以可执行文件的 owner 的权限来运行可执行文件。因为程序文件 /bin/passwd 被设置了 SUID 标识，所以普通用户在执行 passwd 命令时，进程是以 passwd 的所有者，也就是 root 用户的身份运行，从而就可以修改密码了。

但是使用 SUID 却带来了新的安全隐患，当我们运行设置了 SUID 的命令时，通常只是需要很小一部分的特权，但是 SUID 却给了它 root 具有的全部权限，一旦 被设置了 SUID 的命令出现漏洞，是不是就很容易被利用了。

为此 Linux 引入了 Capabilities 机制来对 root 权限进行了更加细粒度的控制，实现按需进行授权，这样就大大减小了系统的安全隐患。

什么是 Capabilities

从内核 2.2 开始，Linux 将传统上与超级用户 root 关联的特权划分为不同的单元，称为 capabilites。Capabilites 每个单元都可以独立启用和禁用。这样当系统在作权限检查的时候就变成了：**在执行特权操作时，如果进程的有效身份不是 root，就去检查是否具有该特权操作所对应的 capabilites，并以此决定是否可以进行该特权操作**。比如如果我们要设置系统时间，就得具有 CAP_SYS_TIME 这个 capabilites。下面是从 capabilities man page 中摘取的 capabilites 列表：

capability 名称	描述
CAP_AUDIT_CONTROL	启用和禁用内核审计；改变审计过滤规则；检索审计状态和过滤规则
CAP_AUDIT_READ	允许通过 multicast netlink 套接字读取审计日志
CAP_AUDIT_WRITE	将记录写入内核审计日志
CAP_BLOCK_SUSPEND	使用可以阻止系统挂起的特性
CAP_CHOWN	修改文件所有者的权限
CAP_DAC_OVERRIDE	忽略文件的 DAC 访问限制
CAP_DAC_READ_SEARCH	忽略文件读及目录搜索的 DAC 访问限制
CAP_FOWNER	忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
CAP_FSETID	允许设置文件的 setuid 位
CAP_IPC_LOCK	允许锁定共享内存片段
CAP_IPC_OWNER	忽略 IPC 所有权检查
CAP_KILL	允许对不属于自己的进程发送信号

CAP_KILL	允许对不属于自己的进程发送信号
CAP_LEASE	允许修改文件锁的 FL_LEASE 标志
CAP_LINUX_IMMUTABLE	允许修改文件的 IMMUTABLE 和 APPEND 属性标志
CAP_MAC_ADMIN	允许 MAC 配置或状态更改
CAP_MAC_OVERRIDE	覆盖 MAC(Mandatory Access Control)
CAP_MKNOD	允许使用 mknod() 系统调用
CAP_NET_ADMIN	允许执行网络管理任务：接口、防火墙和路由等
CAP_NET_BIND_SERVICE	允许绑定到小于 1024 的端口
CAP_NET_BROADCAST	允许网络广播和多播访问
CAP_NET_RAW	允许使用原始套接字
CAP_SETGID	允许改变进程的 GID
CAP_SETFCAP	允许为文件设置任意的 capabilities
CAP_SETPCAP	允许向其它进程转移能力以及删除其它进程的任意能力(只限init进程)
CAP_SETUID	允许改变进程的 UID
CAP_SYS_ADMIN	允许执行系统管理任务，如加载或卸载文件系统、设置磁盘配额等
CAP_SYS_BOOT	允许重新启动系统
CAP_SYS_CHROOT	允许使用 chroot() 系统调用
CAP_SYS_MODULE	允许插入和删除内核模块
	允许提升优先级及设置其他进程的优

CAP_SYS_NICE	允许提升优先级及设置其他进程的优先级
CAP_SYS_PACCT	允许执行进程的 BSD 式审计
CAP_SYS_PTRACE	允许跟踪任何进程
CAP_SYS_RAWIO	允许直接访问 /devport、/dev/mem、/dev/kmem 及原始块设备
CAP_SYS_RESOURCE	忽略资源限制
CAP_SYS_TIME	允许改变系统时钟
CAP_SYS_TTY_CONFIG	允许配置 TTY 设备
CAP_SYSLOG	允许使用 syslog() 系统调用
CAP_WAKE_ALARM	允许触发一些能唤醒系统的东西(比如 CLOCK_BOOTTIME_ALARM 计时器)

Capabilities 的赋予和继承

Linux capabilities 分为进程 capabilities 和文件 capabilities。对于进程来说，capabilities 是细分到线程的，即每个线程可以有自己的 capabilities，对于文件来说，capabilities 保存在文件的扩展属性中。这里我们先分别介绍下线程（进程）的 capabilities 和文件的 capabilities。

线程的 capabilities

每一个线程，具有 5 个 capabilities 集合，每一个集合使用 64 位掩码来表示，显示为 16 进制格式，这 5 个 capabilities 集合分别是：

- Permitted
- Effective
- Inheritable
- Bounding
- Ambient

每个集合中都包含零个或多个 capabilities。这 5 个集合的具体含义如下：

- Permitted：定义了线程能够使用的 capabilities 的上限。线程可以通过系统调用 capset() 来从 Effective 或 Inheritable 集合中添加或删除 capability，前提是添加或删除的 capability 必须包含在 Permitted 集合中（其中 Bounding 集合也会有影响，具体参考下文）。 如果某个线程想向 Inheritable 集合中添加或删除 capability，首先它的 Effective 集合中得包含 CAP_SETPCAP 这个 capabiliy。
- Effective：内核检查线程是否可以进行特权操作时，检查的对象便是 Effective 集合。如之前所说，Permitted 集合定义了上限，线程可以删除 Effective 集合中的某 capability，随后在需要时，再从 Permitted 集合中恢

复该 capability，以此达到临时禁用 capability 的功能。

- Inheritable：当执行 exec() 系统调用时，能够被新的可执行文件继承的 capabilities，被包含在 Inheritable 集合中。这里需要说明一下，包含在该集合中的 capabilities 并不会自动继承给新的可执行文件，即不会添加到新线程的 Effective 集合中，它只会影响新线程的 Permitted 集合。
- Bounding：Bounding 集合是 Inheritable 集合的超集，如果某个 capability 不在 Bounding 集合中，即使它在 Permitted 集合中，该线程也不能将该 capability 添加到它的 Inheritable 集合中。

Bounding 集合的 capabilities 在执行 fork() 系统调用时会传递给子进程的 Bounding 集合，并且在执行 execve 系统调用后保持不变。

当线程运行时，不能向 Bounding 集合中添加 capabilities。
一旦某个 capability 被从 Bounding 集合中删除，便不能再添加回来。
将某个 capability 从 Bounding 集合中删除后，如果之前 Inherited 集合包含该 capability，将继续保留。但如果后续从 Inheritable 集合中删除了该 capability，便不能再添加回来。
Ambient
Linux 4.3 内核新增了一个 capabilities 集合叫 Ambient ，用来弥补 Inheritable 的不足。Ambient 具有如下特性：

Permitted 和 Inheritable 未设置的 capabilities，Ambient 也不能设置。
当 Permitted 和 Inheritable 关闭某权限（比如 CAP_SYS_BOOT）后，Ambient 也随之关闭对应权限。这样就确保了降低权限后子进程也会降低权限。
非特权用户如果在 Permitted 集合中有一个 capability，那么可以添加到 Ambient 集合中，这样它的子进程便可以在 Ambient、Permitted 和 Effective 集合中获取这个 capability。现在不知道为什么也没关系，后面会通过具体的公式来告诉你。
Ambient 的好处显而易见，举个例子，如果你将 CAP_NET_ADMIN 添加到当前进程的 Ambient 集合中，它便可以通过 fork() 和 execve() 调用 shell 脚本来执行网络管理任务，因为 CAP_NET_ADMIN 会自动继承下去。

如何使用 Capabilities

我们可以通过 getcap 和 setcap 两条命令来分别查看和设置程序文件的 capabilities 属性。比如当前我们是 test 这个用户，使用 getcap 命令查看 ping 命令目前具有的 capabilities：

```
[test@master ~]$ ll /bin/ping
-rwxr-xr-x. 1 root root 66176 Aug  4  2017 /bin/ping
[test@master ~]$ getcap /bin/ping
/bin/ping = cap_net_admin,cap_net_raw+p
```

我们可以看到具有 cap_net_admin 这个属性，所以我们现在可以执行 ping 命令：

```
[test@master ~]$ ping youdianzhishi.com
PING youdianzhishi.com (39.106.22.102): 56 data bytes
64 bytes from 39.106.22.102: icmp_seq=0 ttl=50 time=57.207 ms
64 bytes from 39.106.22.102: icmp_seq=1 ttl=50 time=58.131 ms
```

但是如果我们把命令的 capabilities 属性移除掉：

```
[test@master ~]$ sudo setcap cap_net_admin,cap_net_raw-p /bin/ping
[test@master ~]$ getcap /bin/ping
/bin/ping =
```

这个时候我们执行 ping 命令可以发现已经没有权限了：

```
[test@master ~]$ ping youdianzhishi.com
ping: socket: Operation not permitted
```

因为 ping 命令在执行时需要访问网络，所需的 capabilities 为 cap_net_admin 和 cap_net_raw，所以我们可以通过 setcap 命令可来添加它们：

```
[test@master ~]$ sudo setcap cap_net_admin,cap_net_raw+p /bin/ping
[test@master ~]$ getcap /bin/ping
/bin/ping = cap_net_admin,cap_net_raw+p
[test@master ~]$ ping youdianzhishi.com
PING youdianzhishi.com (39.106.22.102): 56 data bytes
64 bytes from 39.106.22.102: icmp_seq=0 ttl=50 time=57.207 ms
```

命令中的 p 表示 Permitted 集合，+ 号表示把指定的 capabilities 添加到这些集合中，- 号表示从集合中移除。

对于可执行文件的属性中有三个集合来保存三类 capabilities，它们分别是：

- Permitted：在进程执行时，Permitted 集合中的 capabilites 自动被加入到进程的 Permitted 集合中。
- Inheritable：Inheritable 集合中的 capabilites 会与进程的 Inheritable 集合执行与操作，以确定进程在执行 execve 函数后哪些 capabilites 被继承。
- Effective：Effective 只是一个 bit，如果设置为开启，那么在执行 execve 函数后，Permitted 集合中新增的 capabilities 会自动出现在进程的 Effective 集合中。

对于进程中有五种 capabilities 集合类型，相比文件的 capabilites，进程的 capabilities 多了两个集合，分别是 Bounding 和 Ambient。

- Bounding：Bounding 集合是 Inheritable 集合的超集，如果某个 capability 不在 Bounding 集合中，即使它在 Permitted 集合中，该线程也不能将该 capability 添加到它的 Inheritable 集合中。

Container Runtime Capabilities

我们说容器本质上就是一个进程，所以理论上容器就会和进程一样会有一些默认的开放权限，默认情况下 Docker/Containerd 会删除必须的 capabilities 之外的所有 capabilities，因为在容器中我们经常会以 root 用户来运行，使用 capabilities 现在后，容器中的使用的 root 用户权限就比我们平时在宿主机上使用的 root 用户权限要少很多了，这样即使出现了安全漏洞，也很难破坏或者获取宿主机的 root 权限，所以 Docker/Containerd 支持 Capabilities 对于容器的安全性来说是非常有必要的。

不过我们在运行容器的时候可以通过指定 --priviledged 参数来开启容器的超级权限，这个参数一定要慎用，因为他会获取系统 root 用户所有能力赋值给容器，并且会扫描宿主机的所有设备文件挂载到容器内部，所以是非常危险的操作。

但是如果你确实需要一些特殊的权限，我们可以通过 --cap-add 和 --cap-drop 这两个参数来动态调整，可以最大限度地保证容器的使用安全。下面表格中列出的 Capabilities 是 Docker 默认给容器添加的，我们可以通过 --cap-drop 去除其中一个或者多个：

Docker Capabilities	Linux Capabilities	Capability 描述
SETPCAP	CAP_SETPCAP	修改进程 capablilities

MKNOD	CAP_MKNOD	允许使用 mknod() 系统调用
AUDIT_WRITE	CAP_AUDIT_WRITE	将记录写入内核审计日志
CHOWN	CAP_CHOWN	修改文件所有者的权限
NET_RAW	CAP_NET_RAW	允许使用原始套接字
DAC_OVERRIDE	CAP_DAC_OVERRIDE	忽略文件的 DAC 访问限制
FOWNER	CAP_FOWNER	忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
FSETID	CAP_FSETID	允许设置文件的 setuid 位
KILL	CAP_KILL	允许对不属于自己的进程发送信号

SETGID	CAP_SETGID	允许改变进程的 GID
SETUID	CAP_SETUID	允许改变进程的 UID
NET_BIND_SERVICE	CAP_NET_BIND_SERVICE	允许绑定到小于 1024 的端口
SYS_CHROOT	CAP_SYS_CHROOT	允许使用 chroot() 系统调用
SETFCAP	CAP_SETFCAP	允许为文件设置任意的 capabilities

下面表格中列出的 `Capabilities` 是 Docker 默认删除的，我们可以通过 `--cap-add` 添加其中一个或者多个：

Docker capabilities	**Linux capabilities**	**Capability 描述**
SYS_MODULE	CAP_SYS_MODULE	允许插入和删除内核模块
SYS_RAWIO	CAP_SYS_RAWIO	允许直接访问 /devport、/dev/mem、/dev/kmem 及原始块设备
SYS_PACCT	CAP_SYS_PACCT	允许执行进程的 BSD 式审计
SYS_ADMIN	CAP_SYS_ADMIN	允许执行系统管理任务，如加载或卸载文件系统、设置磁盘配额等
SYS_NICE	CAP_SYS_NICE	允许提升优先级及设置其他进程的优先级
SYS_RESOURCE	CAP_SYS_RESOURCE	忽略资源限制

SYS_TIME	CAP_SYS_TIME	允许改变系统时钟
SYS_TTY_CONFIG	CAP_SYS_TTY_CONFIG	允许配置 TTY 设备
AUDIT_CONTROL	CAP_AUDIT_CONTROL	启用和禁用内核审计；改变审计过滤规则；检索审计状态和过滤规则
MAC_OVERRIDE	CAP_MAC_OVERRIDE	覆盖 MAC(Mandatory Access Control)
MAC_ADMIN	CAP_MAC_ADMIN	允许 MAC 配置或状态更改
NET_ADMIN	CAP_NET_ADMIN	允许执行网络管理任务
SYSLOG	CAP_SYSLOG	允许使用 syslog() 系统调用
DAC_READ_SEARCH	CAP_DAC_READ_SEARCH	忽略文件读及目录搜索的 DAC 访问限制
LINUX_IMMUTABLE	CAP_LINUX_IMMUTABLE	允许修改文件的 IMMUTABLE 和 APPEND 属性标志
NET_BROADCAST	CAP_NET_BROADCAST	允许网络广播和多播访问
IPC_LOCK	CAP_IPC_LOCK	允许锁定共享内存片段
IPC_OWNER	CAP_IPC_OWNER	忽略 IPC 所有权检查
SYS_PTRACE	CAP_SYS_PTRACE	允许跟踪任何进程
SYS_BOOT	CAP_SYS_BOOT	允许重新启动系统
LEASE	CAP_LEASE	允许修改文件锁的 FL_LEASE 标志
WAKE_ALARM	CAP_WAKE_ALARM	允许触发一些能唤醒系统的东西(比如 CLOCK_BOOTTIME_ALARM 计时器)
BLOCK_SUSPEND	CAP_BLOCK_SUSPEND	使用可以阻止系统挂起的特性

> `--cap-add` 和 `--cap-drop` 这两参数都支持 `ALL` 值，比如如果你想让某个容器拥有除了 `MKNOD` 之外的所有内核权限，那么可以执行命令：`☸ ➜ sudo docker run --cap-add=ALL --cap-drop=MKNOD ...` 。

比如现在我们需要修改网络接口数据，默认情况下是没有权限的，因为需要的 `NET_ADMIN` 这个 `Capabilities` 默认被移除了：

```
# docker 一样的方式
☸ ➜ nerdctl run -it --rm busybox /bin/sh
/ # ip link add dummy0 type dummy
ip: RTNETLINK answers: Operation not permitted
/ #
```

所以在不使用 `--privileged` 的情况下（不建议）我们可以使用 `--cap-add=NET_ADMIN` 将这个 `Capabilities` 添加回来：

```
☸ ➜ nerdctl run -it --rm --cap-add=NET_ADMIN busybox /bin/sh
/ # ip link add dummy0 type dummy
/ #
```

可以看到已经 OK 了。

Kubernetes 配置 Capabilities

上面我介绍了在 Docker/Containerd 容器下如何来配置 `Capabilities`，在 Kubernetes 中也可以很方便的来定义，我们只需要添加到 Pod 定义的 `spec.containers.securityContext.capabilities` 中即可，也可以进行 `add` 和 `drop` 配置，同样上面的示例，我们要给 busybox 容器添加 `NET_ADMIN` 这个 `Capabilities`，对应的 YAML 文件可以这样定义：

```yaml
# cpb-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpb-demo
spec:
  containers:
    - name: cpb
      image: busybox
      args:
        - sleep
        - "3600"
      securityContext:
        capabilities:
          add: # 添加
            - NET_ADMIN
          drop: # 删除
          - KILL
```

我们在 `securityContext` 下面添加了 `capabilities` 字段，其中添加了 `NET_ADMIN` 并且删除了 `KILL` 这个默认的容器 `Capabilities`，这样我们就可以在 Pod 中修改网络接口数据了：

```
☸ ➜ kubectl apply -f cpb-demo.yaml
☸ ➜ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
cpb-demo                   1/1     Running   0          2m9s
☸ ➜ kubectl exec -it cpb-demo -- /bin/sh
/ # ip link add dummy0 type dummy
/ #
```

在 Kubernetes 中通过 containers.securityContext.capabilities 进行配置容器的 Capabilities，当然最终还是通过容器运行时的 libcontainer 去借助 Linux kernel capabilities 实现的权限管理。