openEuler
20.03 LTS
容器用户指南

发布日期 2020-03-26

目 录

法律声明......ix

前言......x

1 iSula 容器引擎......12

1.1 概述......12
1.2 安装与部署......13
1.2.1 安装方法......13
1.2.2 升级方法......13
1.2.3 部署配置......14
1.2.3.1 部署方式......14
1.2.3.2 存储说明......20
1.2.3.3 约束限制......21
1.2.3.4 DAEMON 多端口的绑定......22
1.2.3.5 配置 TLS 认证与开启远程访问......23
1.2.3.6 配置 devicemapper 存储驱动......26
1.2.4 卸载......29
1.3 使用方法......29
1.3.1 容器管理......29
1.3.1.1 创建容器......29
1.3.1.2 启动容器......33
1.3.1.3 运行容器......33
1.3.1.4 停止容器......38
1.3.1.5 强制停止容器......39
1.3.1.6 删除容器......40
1.3.1.7 接入容器......40
1.3.1.8 重命名容器......41
1.3.1.9 在容器中执行新命令......42
1.3.1.10 查询单个容器信息......44
1.3.1.11 查询所有容器信息......47
1.3.1.12 重启容器......48
1.3.1.13 等待容器退出......49

1.3.1.14 查看容器中进程信息 49
1.3.1.15 查看容器使用的资源 50
1.3.1.16 获取容器日志 51
1.3.1.17 容器与主机之间数据拷贝 52
1.3.1.18 暂停容器 53
1.3.1.19 恢复容器 53
1.3.1.20 从服务端实时获取事件消息 54
1.3.2 支持 CNI 网络 54
1.3.2.1 描述 54
1.3.2.2 接口 55
1.3.2.2.1 CNI 网络配置说明 55
1.3.2.2.2 加入 CNI 网络列表 56
1.3.2.2.3 退出 CNI 网络列表 56
1.3.2.3 使用限制 56
1.3.3 容器资源管理 56
1.3.3.1 资源的共享 56
1.3.3.2 限制运行时的 CPU 资源 58
1.3.3.3 限制运行时的内存 59
1.3.3.4 限制运行时的 IO 资源 60
1.3.3.5 限制容器 rootfs 存储空间 61
1.3.3.6 限制容器内文件句柄数 64
1.3.3.7 限制容器内可以创建的进程/线程数 65
1.3.3.8 配置容器内的 ulimit 值 66
1.3.4 特权容器 68
1.3.4.1 场景说明 68
1.3.4.2 使用限制 68
1.3.4.3 使用指导 70
1.3.5 CRI 接口 70
1.3.5.1 描述 70
1.3.5.2 接口 70
1.3.5.2.1 Runtime 服务 85
1.3.5.2.2 Image 服务 96
1.3.5.3 约束 100
1.3.6 镜像管理 101
1.3.6.1 docker 镜像管理 101
1.3.6.1.1 登录到镜像仓库 101
1.3.6.1.2 从镜像仓库退出登录 101
1.3.6.1.3 从镜像仓库拉取镜像 102
1.3.6.1.4 删除镜像 102

1.3.6.1.5 加载镜像 102
1.3.6.1.6 列出镜像 103
1.3.6.1.7 检视镜像 103
1.3.6.1.8 双向认证 104
1.3.6.2 embedded 镜像管理 105
1.3.6.2.1 加载镜像 105
1.3.6.2.2 列出镜像 106
1.3.6.2.3 检视镜像 106
1.3.6.2.4 删除镜像 106
1.3.7 容器健康状态检查 107
1.3.7.1 场景说明 107
1.3.7.2 配置方法 107
1.3.7.3 检查规则 107
1.3.7.4 使用限制 108
1.3.8 查询信息 108
1.3.8.1 查询服务版本信息 108
1.3.8.2 查询系统级信息 109
1.3.9 安全特性 110
1.3.9.1 seccomp 安全配置场景 110
1.3.9.1.1 场景说明 110
1.3.9.1.2 使用限制 110
1.3.9.1.3 使用指导 110
1.3.9.2 capabilities 安全配置场景 113
1.3.9.2.1 场景说明 113
1.3.9.2.2 使用限制 113
1.3.9.2.3 使用指导 113
1.3.9.3 SELinux 安全配置场景 114
1.3.9.3.1 场景说明 114
1.3.9.3.2 使用限制 114
1.3.9.3.3 使用指导 114
1.3.10 支持 OCI hooks 115
1.3.10.1 描述 115
1.3.10.2 接口 116
1.3.10.3 使用限制 116
1.4 附录 116
1.4.1 命令行参数说明 116
1.4.2 CNI 配置参数 118
2 系统容器 124
2.1 概述 124

2.2 安装指导 124
2.3 使用指南 125
2.3.1 简介 125
2.3.2 指定 rootfs 创建容器......125
2.3.3 通过 systemd 启动容器......126
2.3.4 容器内 reboot/shutdown......127
2.3.5 cgroup 路径可配置 128
2.3.6 namespace 化内核参数可写 130
2.3.7 共享内存通道 132
2.3.8 动态加载内核模块 133
2.3.9 环境变量持久化 134
2.3.10 最大句柄数限制 135
2.3.11 安全性和隔离性 136
2.3.11.1 user namespace 多对多......136
2.3.11.2 用户权限控制 137
2.3.11.3 proc 文件系统隔离（lxcfs） 140
2.3.12 容器资源动态管理（syscontainer-tools） 142
2.3.12.1 设备管理......143
2.3.12.2 网卡管理......146
2.3.12.3 路由管理......148
2.3.12.4 挂卷管理......150
2.4 附录 152
2.4.1 命令行接口列表 152
3 安全容器......154
3.1 概述 154
3.2 安装部署 156
3.2.1 安装方法......156
3.2.2 部署配置......157
3.2.2.1 docker-engine 容器引擎的配置 157
3.2.2.2 iSula 容器引擎的配置......157
3.2.2.3 安全容器全局配置文件 configuration.toml......158
3.3 使用方法......158
3.3.1 管理安全容器的生命周期......158
3.3.1.1 启动安全容器 158
3.3.1.2 停止安全容器 159
3.3.1.3 删除安全容器 159
3.3.1.4 在容器中执行一条新的命令 160
3.3.2 为安全容器配置资源......160
3.3.2.1 资源的共享......160

3.3.2.2 限制 CPU 资源......160
3.3.2.3 限制内存资源......163
3.3.2.4 限制 Blkio 资源......164
3.3.2.5 限制文件描述符资源......166
3.3.3 为安全容器配置网络......166
3.3.4 监控安全容器......171
3.4 附录......173
3.4.1 configuration.toml 配置说明......173
3.4.2 接口列表......175
4 Docker 容器......180
4.1 概述......180
4.2 安装部署......180
4.2.1 安装配置介绍及注意事项......180
4.2.1.1 注意事项......180
4.2.1.2 基本安装配置......181
4.2.1.2.1 配置 daemon 参数......181
4.2.1.2.2 daemon 运行目录配置......181
4.2.1.2.3 daemon 自带网络配置......181
4.2.1.2.4 daemon umask 配置......182
4.2.1.2.5 daemon 启动时间......182
4.2.1.2.6 关联组件 journald......182
4.2.1.2.7 关联组件 firewalld......183
4.2.1.2.8 关联组件 iptables......183
4.2.1.2.9 关联组件 audit......183
4.2.1.2.10 安全配置 seccomp......184
4.2.1.2.11 禁止修改 docker daemon 的私有目录......184
4.2.1.2.12 普通用户大量部署容器场景下的配置注意事项......184
4.2.1.3 存储驱动配置......184
4.2.1.3.1 配置 overlay2 存储驱动......185
4.2.1.3.2 配置 devicemapper 存储驱动......187
4.2.1.4 强制退出 docker 相关后台进程的影响......188
4.2.1.4.1 信号量残留......188
4.2.1.4.2 网卡残留......189
4.2.1.4.3 重启容器失败......189
4.2.1.4.4 服务无法正常重启......189
4.2.1.5 系统掉电影响......189
4.3 容器管理......190
4.3.1 创建容器......190
4.3.2 创建容器使用 hook-spec......196

4.3.3 创建容器配置健康检查......199
4.3.4 停止与删除容器......202
4.3.5 容器信息查询......203
4.3.6 修改操作......203
4.4 镜像管理......205
4.4.1 创建镜像......205
4.4.2 查看镜像......206
4.4.3 删除镜像......206
4.5 命令行参考......206
4.5.1 容器引擎......206
4.5.2 容器管理......209
4.5.2.1 attach......212
4.5.2.2 commit......212
4.5.2.3 cp......213
4.5.2.4 create......213
4.5.2.5 diff......217
4.5.2.6 exec......218
4.5.2.7 export......218
4.5.2.8 inspect......218
4.5.2.9 logs......219
4.5.2.10 pause/unpause......220
4.5.2.11 port......221
4.5.2.12 ps......221
4.5.2.13 rename......222
4.5.2.14 restart......222
4.5.2.15 rm......222
4.5.2.16 run......223
4.5.2.17 start......223
4.5.2.18 stats......224
4.5.2.19 stop......224
4.5.2.20 top......224
4.5.2.21 update......225
4.5.2.22 wait......226
4.5.3 镜像管理......226
4.5.3.1 build......227
4.5.3.2 history......229
4.5.3.3 images......230
4.5.3.4 import......230
4.5.3.5 load......231
4.5.3.6 login......231
4.5.3.7 logout......231

4.5.3.8 pull 231
4.5.3.9 push......232
4.5.3.10 rmi......232
4.5.3.11 save 233
4.5.3.12 search......233
4.5.3.13 tag 234
4.5.4 统计信息......234
4.5.4.1 events......234
4.5.4.2 info......235
4.5.4.3 version 235

法律声明

版权所有 © 2020 华为技术有限公司。

您对“本文档”的复制、使用、修改及分发受知识共享(Creative Commons)署名—相同方式共享 4.0 国际公共许可协议(以下简称“CC BY-SA 4.0”)的约束。为了方便用户理解，您可以通过访问 https://creativecommons.org/licenses/by-sa/4.0/ 了解 CC BY-SA 4.0 的概要 (但不是替代)。CC BY-SA 4.0 的完整协议内容您可以访问如下网址获取：https://creativecommons.org/licenses/by-sa/4.0/legalcode。

商标声明

openEuler 为华为技术有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

免责声明

本文档仅作为使用指导，除非适用法强制规定或者双方有明确书面约定, 华为技术有限公司对本文档中的所有陈述、信息和建议不做任何明示或默示的声明或保证，包括但不限于不侵权，时效性或满足特定目的的担保。

前言

概述

openEuler 软件包中提供容器运行的基础平台 iSula。

iSula 为华为容器技术方案品牌，其原意是一种非常强大的蚂蚁，学术上称为“子弹蚁”，因为被它咬一口，犹如被子弹打到那般疼痛。在居住于中南美洲亚马逊丛林的巴西原住民眼里，iSula 是世界上非常强大的昆虫之一。华为容器技术方案品牌因其含义取名。

iSula 基础容器平台同时提供 Docker engine 与轻量化容器引擎 iSulad，用户可根据需要自主选择。

同时根据不同使用场景，提供多种容器形态，包括：

- 适合大部分通用场景的普通容器
- 适合强隔离与多租户场景的安全容器
- 适合使用 systemd 管理容器内业务场景的系统容器

本文档提供容器引擎的安装和使用方法以及各个容器形态的部署使用方法。

读者对象

本文档主要适用于使用 openEuler 并需要安装容器的用户。用户需要具备以下经验和技能：

- 熟悉 Linux 基本操作
- 对容器有一定了解

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
须知	用于传递设备或环境安全警示信息。如不避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。

符号	说明
	“须知”不涉及人身伤害。
说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

1 iSula 容器引擎

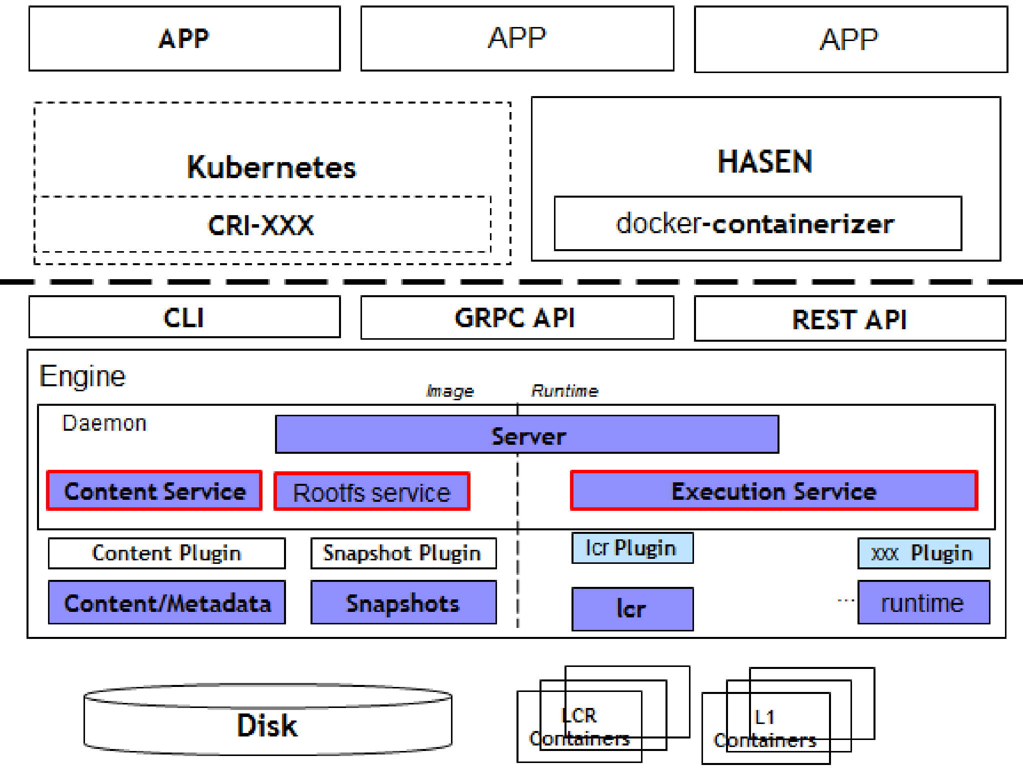
1.1 概述
1.2 安装与部署
1.3 使用方法
1.4 附录

1.1 概述

iSula 通用容器引擎相比 docker，是一种新的容器解决方案，提供统一的架构设计来满足 CT 和 IT 领域的不同需求。相比 Golang 编写的 Docker，轻量级容器使用 C/C++实现，具有轻、灵、巧、快的特点，不受硬件规格和架构的限制，底噪开销更小，可应用领域更为广泛。

容器统一架构如图 1-1 所示。

图1-1 容器统一架构



1.2 安装与部署

1.2.1 安装方法

iSulad 可以通过 yum 或 rpm 命令两种方式安装，由于 yum 会自动安装依赖，而 rpm 命令需要手动安装所有依赖，所以推荐使用 yum 安装。

这里给出两种安装方式的操作方法。

- （推荐）使用 yum 安装 iSulad，参考命令如下：

```
$ sudo yum install -y iSulad
```

- 使用 rpm 安装 iSulad，需要下载 iSulad 及其所有依赖库的 RPM 包，然后手动安装。安装单个 iSulad 的 RPM 包（依赖包安装方式相同），参考命令如下：

```
$ sudo rpm -ihv iSulad-xx.xx.xx-YYYYmmdd.HHMMSS.gitxxxxxxxx.aarch64.rpm
```

1.2.2 升级方法

- 若为相同大版本之间的升级，例如从 2.x.x 版本升级到 2.x.x 版本，请执行如下命令：

```
$ sudo yum update -y iSulad
```

- 若为不同大版本之间的升级，例如从 1.x.x 版本升级到 2.x.x 版本，请先保存当前的配置文件/etc/isulad/daemon.json，并卸载已安装的 iSulad 软件包，然后安装待升级的 iSulad 软件包，随后恢复配置文件。

📖 **说明**

- 可通过 **sudo rpm -qa |grep iSulad** 或 **isula version** 命令确认当前 iSulad 的版本号。
- 相同大版本之间，如果希望手动升级，请下载 iSulad 及其所有依赖库的 RPM 包进行升级，参考命令如下：

  ```
  $ sudo rpm -Uhv iSulad-xx.xx.xx-YYYYmmdd.HHMMSS.gitxxxxxxxx.aarch64.rpm
  ```

  若升级失败，可通过--force 选项进行强制升级，参考命令如下：

  ```
  $ sudo rpm -Uhv --force iSulad-xx.xx.xx-
  YYYYmmdd.HHMMSS.gitxxxxxxxx.aarch64.rpm
  ```

1.2.3 部署配置

1.2.3.1 部署方式

轻量级容器引擎（iSulad）服务端 daemon 为 isulad，isulad 可以通过配置文件进行配置，也可以通过命令行的方式进行配置，例如：isulad --xxx，优先级从高到低是：命令行方式>配置文件>代码中默认配置。

📖 **说明**

如果采用 systemd 管理 iSulad 进程，修改/etc/sysconfig/iSulad 文件中的 OPTIONS 字段，等同于命令行方式进行配置。

- **命令行方式**

  在启动服务的时候，直接通过命令行进行配置。其配置选项可通过以下命令查阅：

```
$ isulad --help
lightweight container runtime daemon

Usage:  isulad [global options]

GLOBAL OPTIONS:

     --authorization-plugin          Use authorization plugin
     --cgroup-parent                Set parent cgroup for all containers
     --cni-bin-dir                  The full path of the directory in which to
search for CNI plugin binaries. Default: /opt/cni/bin
     --cni-conf-dir                 The full path of the directory in which to
search for CNI config files. Default: /etc/cni/net.d
     --default-ulimit               Default ulimits for containers (default [])
  -e, --engine                      Select backend engine
  -g, --graph                       Root directory of the iSulad runtime
  -G, --group                       Group for the unix socket(default is isulad)
     --help                         Show help
     --hook-spec                    Default hook spec file applied to all
containers
  -H, --host                        The socket name used to create gRPC server
     --image-layer-check             Check layer intergrity when needed
     --image-opt-timeout             Max timeout(default 5m) for image
operation
     --insecure-registry             Disable TLS verification for the given
registry
     --insecure-skip-verify-enforce    Force to skip the insecure
verify(default false)
```

```
     --log-driver                  Set daemon log driver, such as: file
  -l, --log-level                  Set log level, the levels can be: FATAL
ALERT CRIT ERROR WARN NOTICE INFO DEBUG TRACE
     --log-opt                     Set daemon log driver options, such as: log-
path=/tmp/logs/ to set directory where to store daemon logs
     --native.umask                Default file mode creation mask (umask) for
containers
     --network-plugin              Set network plugin, default is null,
suppport null and cni
  -p, --pidfile                    Save pid into this file
     --pod-sandbox-image           The image whose network/ipc namespaces
containers in each pod will use. (default "rnd-
dockerhub.huawei.com/library/pause-${machine}:3.0")
     --registry-mirrors            Registry to be prepended when pulling
unqualified images, can be specified multiple times
     --start-timeout               timeout duration for waiting on a container
to start before it is killed
  -S, --state                      Root directory for execution state files
     --storage-driver              Storage driver to use(default overlay2)
  -s, --storage-opt                Storage driver options
     --tls                         Use TLS; implied by --tlsverify
     --tlscacert                   Trust certs signed only by this CA (default
"/root/.iSulad/ca.pem")
     --tlscert                     Path to TLS certificate file (default
"/root/.iSulad/cert.pem")
     --tlskey                      Path to TLS key file (default
"/root/.iSulad/key.pem")
     --tlsverify                   Use TLS and verify the remote
     --use-decrypted-key           Use decrypted private key by
default(default true)
  -V, --version                    Print the version
   --websocket-server-listening-port   CRI websocket streaming service
listening port (default 10350)
```

示例： 启动 isulad，并将日志级别调整成 DEBUG

```
$ isulad -l DEBUG
```

- **配置文件方式**

  isulad 配置文件为/etc/isulad/daemon.json，各配置字段说明如下：

配置参数	配置文件示例	参数解释	备注
-e, --engine	"engine": "lcr"	iSulad 的运行时，默认是 lcr	无
-G, --group	"group": "isulad"	socket 所属组	无
--hook-spec	"hook-spec": "/etc/default/isulad/hooks/default.json"	针对所有容器的默认钩子配置文件	无
-H, --host	"hosts": "unix:///var/run/isulad.sock"	通信方式	除本地 socket 外，还支持 tcp://ip:port 方式，port 范围（0-65535，排

配置参数	配置文件示例	参数解释	备注
			除被占用端口）
--log-driver	"log-driver": "file"	日志驱动配置	无
-l, --log-level	"log-level": "ERROR"	设置日志输出级别	无

--log-opt	"log-opts": {
	"log-file-mode": "0600",
	"log-path": "/var/lib/isulad",
	"max-file": "1",
	"max-size": "30KB"
	}
		日志相关的配置	可以指定 max-file，max-size，log-path。max-file 指日志文件个数；max-size 指日志触发防爆的阈值，若 max-file 为 1，max-size 失效；log-path 指定日志文件存储路径；log-file-mode 用于设置日志文件的读写权限，格式要求必须为八进制格式，如 0666。

--start-timeout	"start-timeout": "2m"	启动容器的耗时	无
--runtime	"default-runtime": "lcr"	创建容器时的 runtime 运行时，默认是 lcr	当命令行和配置文件均未指定时，默认为 lcr，runtime 的三种指定方式优先级：命令行>配置文件>默认 lcr，当前支持 lcr、kata-runtime。

无	"runtimes": {
	"kata-runtime": {
	"path": "/usr/bin/kata-runtime",
	"runtime-args": [
	"--kata-config",
	"/usr/share/defaults/kata-containers/configuration.toml"
	]
	}
	}
		启动容器时，通过此字段指定多 runtimes 配置，在此集合中的元素均为有效的启动容器的 runtime 运行时。	容器的 runtime 白名单，在此集合中的自定义 runtime 才是有效的。示例为以 kata-runtime 为例的配置。

-p, --pidfile	"pidfile": "/var/run/isulad.pid"	保存 pid 的文件	当启动一个容器引擎的时候不需要配置，当需要启动两个以上的容器引擎时才需要配置。
-g, --graph	"graph": "/var/lib/isulad"	iSulad 运行时的根目录	
-S, --state	"state": "/var/run/isulad"	执行文件的根目	

配置参数	配置文件示例	参数解释	备注
		录	
--storage-driver	"storage-driver": "overlay2"	镜像存储驱动，默认为 overlay2	当前只支持 overlay2

-s, --storage-opt	"storage-opts": ["overlay2.override_kernel_check=true"]	镜像存储驱动配置选项	可使用的选项为：
			`overlay2.override_kernel_check=true # 忽略内核版本检查`
			`overlay2.size=${size} # 设置 rootfs quota 限额为${size}大小`
			`overlay2.basesize=${size} #等价于 overlay2.size`

--image-opt-timeout	"image-opt-timeout": "5m"	镜像操作超时时间，默认为 5m	值为-1 表示不限制超时。
--registry-mirrors	"registry-mirrors": ["docker.io"]	镜像仓库地址	无
--insecure-registry	"insecure-registries": [ ]	不使用 TLS 校验的镜像仓库	无

--native.umask	"native.umask": "secure"	容器 umask 策略，默认"secure"，normal 为不安全配置	设置容器 umask 值。
			支持配置空字符（使用默认值 0027）、"normal"、"secure"：
			`normal # 启动的容器umask 值为 0022`
			`secure # 启动的容器umask 值为 0027（默认值）`

--pod-sandbox-image	"pod-sandbox-image": "rnd-dockerhub.huawei.com/library/pause-aarch64:3.0"	pod 默认使用镜像，默认为"rnd-dockerhub.huawei.com/library/pause-${machine}:3.0"	无
--network-plugin	"network-plugin": ""	指定网络插件，默认为空字符，表示无网络配置，创建的 sandbox 只有 loop 网卡。	支持 cni 和空字符，其他非法值会导致 isulad 启动失败。
--cni-bin-dir	"cni-bin-dir": ""	指定 cni 插件依赖的二进制的存储位置	默认为/opt/cni/bin

配置参数	配置文件示例	参数解释	备注
--cni-conf-dir	"cni-conf-dir": ""	指定 cni 网络配置文件的存储位置	默认为/etc/cni/net.d
--image-layer-check=false	"image-layer-check": false	开启镜像层完整性检查功能，设置为 true；关闭该功能，设置为 false。默认为关闭。	isulad 启动时会检查镜像层的完整性，如果镜像层被破坏，则相关的镜像不可用。isulad 进行镜像完整性校验时，无法校验内容为空的文件和目录，以及链接文件。因此若镜像因掉电导致上述类型文件丢失，isulad 的镜像数据完整性校验可能无法识别。isulad 版本变更时需要检查是否支持该参数，如果不支持，需要从配置文件中删除。
--insecure-skip-verify-enforce=false	"insecure-skip-verify-enforce": false	Bool 类型，是否强制跳过证书的主机名/域名验证，默认为 false。当设置为 true 时，为不安全配置，会跳过证书的主机名/域名验证	默认为 false（不跳过），注意：因 isulad 使用的 yajl json 解析库限制，若在/etc/isulad/daemon.json 配置文件中配置非 Bool 类型的其他符合 json 格式的值时，isulad 将使用默认值 false。
--use-decrypted-key=true	"use-decrypted-key": true	Bool 类型，指定是否使用不加密的私钥。指定为 true，表示使用不加密的私钥；指定为 false，表示使用的为加密后的私钥，即需要进行双向认证。	默认配置为 true(使用不加密的私钥)，注意：因 isulad 使用的 yajl json 解析库限制，若在/etc/isulad/daemon.json 配置文件中配置非 Bool 类型的其他符合 json 格式的值时，isulad 将使用默认值 true。
--tls	"tls":false	Bool 类型，是否使用 TLS	默认值为 false， 仅用于-H tcp://IP:PORT 方式
--tlsverify	"tlsverify":false	Bool 类型，是否使用 TLS，并验证远程访问	仅用于-H tcp://IP:PORT 方式
--tlscacert	"tls-config": {	TLS 证书相关	仅用于-H tcp://IP:PORT

配置参数	配置文件示例	参数解释	备注
--tlscert --tlskey	"CAFile": "/root/.iSulad/ca.pem", "CertFile": "/root/.iSulad/server-cert.pem", "KeyFile":"/root/.iSulad/server-key.pem" }	的配置	方式
--authorization-plugin	"authorization-plugin": "authz-broker"	用户权限认证插件	当前只支持 authz-broker
--cgroup-parent	"cgroup-parent": "lxc/mycgroup"	字符串类型，容器默认 cgroup 父路径	指定 daemon 端容器默认的 cgroup 父路径，如果客户端指定了--cgroup-parent，以客户端参数为准。 注意：如果启了一个容器 A，然后启一个容器 B，容器 B 的 cgroup 父路径指定为容器 A 的 cgroup 路径，在删除容器的时候需要先删除容器 B 再删除容器 A，否则会导致 cgroup 资源残留。

--default-ulimits	"default-ulimits": {
	"nofile": {
	"Name": "nofile",
	"Hard": 6400,
	"Soft": 3200
	}
	}
		ulimit 指定限制的类型，soft 值及 hard 值	指定限制的资源类型，如“nofile”。两个字段名字必须相同，即都为 nofile，否则会报错。Hard 指定的值需要大于等于 Soft'。如果 Hard 字段或者 Soft 字段未设置，则默认该字段默认为 0。

--websocket-server-listening-port	"websocket-server-listening-port": 10350	设置 CRI websocket 流式服务侦听端口，默认端口号 10350	指定 CRI websocket 流式服务侦听端，如果客户端指定了 --websocket-server-listening-port，以客户端参数为准。端口范围 1024-49151

示例：

```
$ cat /etc/isulad/daemon.json
{
    "group": "isulad",
    "default-runtime": "lcr",
    "graph": "/var/lib/isulad",
    "state": "/var/run/isulad",
    "engine": "lcr",
    "log-level": "ERROR",
    "pidfile": "/var/run/isulad.pid",
    "log-opts": {
        "log-file-mode": "0600",
        "log-path": "/var/lib/isulad",
        "max-file": "1",
        "max-size": "30KB"
    },
    "log-driver": "stdout",
    "hook-spec": "/etc/default/isulad/hooks/default.json",
    "start-timeout": "2m",
    "storage-driver": "overlay2",
    "storage-opts": [
        "overlay2.override kernel check=true"
    ],
    "registry-mirrors": [
        "docker.io"
    ],
    "insecure-registries": [
        "rnd-dockerhub.huawei.com"
    ],
    "pod-sandbox-image": "",
    "image-opt-timeout": "5m",
    "native.umask": "secure",
    "network-plugin": "",
    "cni-bin-dir": "",
    "cni-conf-dir": "",
    "image-layer-check": false,
    "use-decrypted-key": true,
    "insecure-skip-verify-enforce": false
}
```

须知

默认配置文件/etc/isulad/daemon.json 仅供参考，请根据实际需要进行配置

1.2.3.2 存储说明

文件名	文件路径	内容
*	/etc/default/isulad/	存放 isulad 的 OCI 配置文件和钩子模板文件，文件夹下的配置文件权限设置为 0640，sysmonitor 检查脚本权限为 0550

文件名	文件路径	内容
*	/etc/isulad/	isulad 的默认配置文件和 seccomp 的默认配置文件↵
isulad.sock	/var/run/	管道通信文件，客户端和 isulad 的通信使用的 socket 文件
isulad.pid	/var/run/	存放 isulad 的 PID，同时也是一个文件锁防止启动多个 isulad 实例
*	/run/lxc/	文件锁文件，isula 运行过程创建的文件
*	/var/run/isulad/	实时通讯缓存文件，isulad 运行过程创建的文件
*	/var/run/isula/	实时通讯缓存文件，isula 运行过程创建的文件
*	/var/lib/lcr/	LCR 组件临时目录
*	/var/lib/isulad/	isulad 运行的根目录，存放创建的容器配置、日志的默认路径、数据库文件、mount 点等 /var/lib/isulad/mnt/ ：容器 rootfs 的 mount 点 /var/lib/isulad/engines/lcr/ ：存放 lcr 容器配置目录，每个容器一个目录（以容器名命名）

1.2.3.3 约束限制

- 高并发场景（并发启动 200 容器）下，glibc 的内存管理机制会导致内存空洞以及虚拟内存较大（例如 10GB）的问题。该问题是高并发场景下 glibc 内存管理机制的限制，而不是内存泄露，不会导致内存消耗无限增大。可以通过设置 MALLOC_ARENA_MAX 环境变量来减少虚拟内存的问题，而且可以增大减少物理内存的概率。但是这个环境变量会导致 iSulad 的并发性能下降，需要用户根据实际情况做配置。

```
参考实践情况，平衡性能和内存，可以设置 MALLOC_ARENA_MAX 为 4。（在 arm64 服务器上面对 iSulad
的性能影响在 10%以内）

配置方法：
1. 手动启动 iSulad 的场景，可以直接 export MALLOC_ARENA_MAX=4，然后再启动 iSulad 即可。
2. systemd 管理 iSulad 的场景，可以修改/etc/sysconfig/iSulad，增加一条
MALLOC_ARENA_MAX=4 即可。
```

- 为 daemon 指定各种运行目录时的注意事项

  以--root 为例，当使用/new/path/作为 daemon 新的 Root Dir 时，如果/new/path/下已经存在文件，且目录或文件名与 isulad 需要使用的目录或文件名冲突（例如：engines、mnt 等目录）时，isulad 可能会更新原有目录或文件的属性，包括属主、权限等为自己的属主和权限。

  所以，用户需要明白重新指定各种运行目录和文件，会对冲突目录、文件属性的影响。建议用户指定的新目录或文件为 isulad 专用，避免冲突导致的文件属性变化以及带来的安全问题。

- 日志文件管理：

须知

日志功能对接： iSulad 由 systemd 管理，日志也由 systemd 管理，然后传输给 rsyslogd。rsyslog 默认会对写日志速度有限制，可以通过修改/etc/rsyslog.conf 文件，增加"$imjournalRatelimitInterval 0"配置项，然后重启 rsyslogd 的服务即可。

- 命令行参数解析限制

  使用 iSulad 命令行接口时，其参数解析方式与 docker 略有不同，对于命令行中带参数的 flag，不管使用长 flag 还是短 flag，只会将该 flag 后第一个空格或与 flag 直接相连接的'='后的字符串作为 flag 的参数，具体如下：

  a. 使用短 flag 时，与“-”连接的字符串中的每个字符都被当作短 flag（当有=号时，=号后的字符串当成=号前的短 flag 的参数）。

     isula run -du=root busybox 等价于 isula run -du root busybox 或 isula run -d -u=root busybox 或 isula run -d -u root busybox ，当使用 isula run -du:root 时，由于-:不是有效的短 flag，因此会报错。前述的命令行也等价于 isula run -ud root busybox，但不推荐这种使用方式，可能带来语义困扰。

  b. 使用长 flag 时，与“--”连接的字符串作为一个整体当成长 flag，若包含=号，则=号前的字符串为长 flag，=号后的为参数。

     ```
     isula run --user=root busybox
     ```

     等价于

     ```
     isula run --user root busybox
     ```

- 启动一个 isulad 容器，不能够以非 root 用户进行 isula run -i/-t/-ti 以及 isula attach/exec 操作。
- iSulad 对接 OCI 容器时，仅支持 kata-runtime 启动 OCI 容器。

1.2.3.4 DAEMON 多端口的绑定

描述

daemon 端可以绑定多个 unix socket 或者 tcp 端口，并在这些端口上侦听，客户端可以通过这些端口和 daemon 端进行交互。

接口

用户可以在/etc/isulad/daemon.json 文件的 hosts 字段配置一个或者多个端口。当然用户也可以不指定 hosts。

```
{
    "hosts": [
        "unix:///var/run/isulad.sock",
        "tcp://localhost:5678",
        "tcp://127.0.0.1:6789"
    ]
}
```

用户也可以在/etc/sysconfig/iSulad 中通过-H 或者--host 配置端口。用户同样可以不指定 hosts。

```
OPTIONS='-H unix:///var/run/isulad.sock --host tcp://127.0.0.1:6789'
```

如果用户在 daemon.json 文件及 iSulad 中均未指定 hosts，则 daemon 在启动之后将默认侦听 unix:///var/run/isulad.sock。

限制

- 用户不可以在/etc/isulad/daemon.json 和/etc/sysconfig/iSuald 两个文件中同时指定 hosts，如果这样做将会出现错误，isulad 无法正常启动；

  ```
  unable to configure the isulad with file /etc/isulad/daemon.json: the following directives are specified both as a flag and in the configuration file: hosts: (from flag: [unix:///var/run/isulad.sock tcp://127.0.0.1:6789], from file: [unix:///var/run/isulad.sock tcp://localhost:5678 tcp://127.0.0.1:6789])
  ```

- 若指定的 host 是 unix socket，则必须是合法的 unix socket，需要以"unix://"开头，后跟合法的 socket 绝对路径；
- 若指定的 host 是 tcp 端口，则必须是合法的 tcp 端口，需要以"tcp://"开头，后跟合法的 IP 地址和端口，IP 地址可以为 localhost；
- 可以指定至多 10 个有效的端口，超过 10 个则会出现错误，isulad 无法正常启动。

1.2.3.5 配置 TLS 认证与开启远程访问

描述

iSulad 采用 C/S 模式进行设计，在默认情况，iSulad 守护进程 isulad 只侦听本地/var/run/isulad.sock，因此只能在本地通过客户端 isula 执行相关命令操作容器。为了能使 isula 可以远程访问容器，isulad 守护进程需要通过 tcp:ip 的方式侦听远程访问的端口。然而，仅通过简单配置 tcp ip:port 进行侦听，这样会导致所有的 ip 都可以通过调用 isula -H tcp://<remote server ip>:port 与 isulad 通信，容易导致安全问题，因此推荐使用较安全版本的 TLS(**Transport Layer Security - 安全传输层协议**）方式进行远程访问。

生成 TLS 证书

- 明文私钥和证书生成方法示例

  ```
  #!/bin/bash
  set -e
  ```

```
echo -n "Enter pass phrase:"
read password
echo -n "Enter public network ip:"
read publicip
echo -n "Enter host:"
read HOST

echo " => Using hostname: $publicip, You MUST connect to iSulad using this
host!"

mkdir -p $HOME/.iSulad
cd $HOME/.iSulad
rm -rf $HOME/.iSulad/*

echo " => Generating CA key"
openssl genrsa -passout pass:$password -aes256 -out ca-key.pem 4096
echo " => Generating CA certificate"
openssl req -passin pass:$password -new -x509 -days 365 -key ca-key.pem -sha256
-out ca.pem -subj
"/C=CN/ST=zhejiang/L=hangzhou/O=Huawei/OU=iSulad/CN=iSulad@huawei.com"
echo " => Generating server key"
openssl genrsa -passout pass:$password -out server-key.pem 4096
echo " => Generating server CSR"
openssl req -passin pass:$password -subj /CN=$HOST -sha256 -new -key server-
key.pem -out server.csr
echo subjectAltName = DNS:$HOST,IP:$publicip,IP:127.0.0.1 >> extfile.cnf
echo extendedKeyUsage = serverAuth >> extfile.cnf
echo " => Signing server CSR with CA"
openssl x509 -req -passin pass:$password -days 365 -sha256 -in server.csr -CA
ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile
extfile.cnf
echo " => Generating client key"
openssl genrsa -passout pass:$password -out key.pem 4096
echo " => Generating client CSR"
openssl req -passin pass:$password -subj '/CN=client' -new -key key.pem -out
client.csr
echo " => Creating extended key usage"
echo extendedKeyUsage = clientAuth > extfile-client.cnf
echo " => Signing client CSR with CA"
openssl x509 -req -passin pass:$password -days 365 -sha256 -in client.csr -CA
ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem -extfile extfile-
client.cnf
rm -v client.csr server.csr extfile.cnf extfile-client.cnf
chmod -v 0400 ca-key.pem key.pem server-key.pem
chmod -v 0444 ca.pem server-cert.pem cert.pem
```

- 加密私钥和证书请求文件生成方法示例

```
#!/bin/bash

echo -n "Enter public network ip:"
read publicip
echo -n "Enter pass phrase:"
read password

# remove certificates from previous execution.
rm -f *.pem *.srl *.csr *.cnf
```

```
# generate CA private and public keys
echo 01 > ca.srl
openssl genrsa -aes256 -out ca-key.pem -passout pass:$password 2048
openssl req -subj
'/C=CN/ST=zhejiang/L=hangzhou/O=Huawei/OU=iSulad/CN=iSulad@huawei.com' -new -
x509 -days $DAYS -passin pass:$password -key ca-key.pem -out ca.pem

# create a server key and certificate signing request (CSR)
openssl genrsa -aes256 -out server-key.pem -passout pass:$PASS 2048
openssl req -new -key server-key.pem -out server.csr -passin pass:$password -
subj '/CN=iSulad'

echo subjectAltName = DNS:iSulad,IP:${publicip},IP:127.0.0.1 > extfile.cnf
echo extendedKeyUsage = serverAuth >> extfile.cnf
# sign the server key with our CA
openssl x509 -req -days $DAYS -passin pass:$password -in server.csr -CA ca.pem
-CAkey ca-key.pem -out server-cert.pem -extfile extfile.cnf

# create a client key and certificate signing request (CSR)
openssl genrsa -aes256 -out key.pem -passout pass:$password 2048
openssl req -subj '/CN=client' -new -key key.pem -out client.csr -passin
pass:$password

# create an extensions config file and sign
echo extendedKeyUsage = clientAuth > extfile.cnf
openssl x509 -req -days 365 -passin pass:$password -in client.csr -CA ca.pem -
CAkey ca-key.pem -out cert.pem -extfile extfile.cnf

# remove the passphrase from the client and server key
openssl rsa -in server-key.pem -out server-key.pem -passin pass:$password
openssl rsa -in key.pem -out key.pem -passin pass:$password

# remove generated files that are no longer required
rm -f ca-key.pem ca.srl client.csr extfile.cnf server.csr
```

接口

```
{
    "tls": true,
    "tls-verify": true,
    "tls-config": {
         "CAFile": "/root/.iSulad/ca.pem",
         "CertFile": "/root/.iSulad/server-cert.pem",
         "KeyFile":"/root/.iSulad/server-key.pem"
    }
}
```

限制

服务端支持的模式如下：

- 模式 1（验证客户端）：tlsverify, tlscacert, tlscert, tlskey。
- 模式 2（不验证客户端）：tls, tlscert, tlskey。

客户端支持的模式如下:

- 模式 1(使用客户端证书进行身份验证，并根据给定的 CA 验证服务器)：tlsverify, tlscacert, tlscert, tlskey。
- 模式 2(验证服务器)：tlsverify, tlscacert。

如果需要采用双向认证方式进行通讯，则服务端采用模式 1，客户端采用模式 1；

如果需要采用单向认证方式进行通讯，则服务端采用模式 2，客户端采用模式 2。

须知

- 采用 RPM 安装方式时，服务端配置可通过/etc/isulad/daemon.json 以及/etc/sysconfig/iSulad 配置修改
- 相比非认证或者单向认证方式，双向认证具备更高的安全性，推荐使用双向认证的方式进行通讯
- GRPC 开源组件日志不由 iSulad 进行接管，如果需要查看 GRPC 相关日志，请按需设置 GRPC_VERBOSITY 和 GRPC_TRACE 环境变量

示例

服务端：

```
isulad -H=tcp://0.0.0.0:2376 --tlsverify --tlscacert ~/.iSulad/ca.pem --tlscert
~/.iSulad/server-cert.pem --tlskey ~/.iSulad/server-key.pem
```

客户端：

```
isula version -H=tcp://$HOSTIP:2376 --tlsverify --tlscacert ~/.iSulad/ca.pem --
tlscert ~/.iSulad/cert.pem --tlskey ~/.iSulad/key.pem
```

1.2.3.6 配置 devicemapper 存储驱动

使用 devicemapper 存储驱动需要先配置一个 thinpool 设备，而配置 thinpool 需要一个独立的块设备，且该设备需要有足够的空闲空间用于创建 thinpool，请用户根据实际需求确定。这里假设独立块设备为/dev/xvdf，具体的配置方法如下：

一、配置 thinpool

1. 停止 isulad 服务。

```
# systemctl stop isulad
```

2. 基于块设备创建一个 lvm 卷。

```
# pvcreate /dev/xvdf
```

3. 使用刚才创建的物理卷创建一个卷组。

```
# vgcreate isula /dev/xvdf
Volume group "isula" successfully created:
```

4. 创建名为 thinpool 和 thinpoolmeta 的两个逻辑卷。

```
# lvcreate --wipesignatures y -n thinpool isula -l 95%VG
Logical volume "thinpool" created.
# lvcreate --wipesignatures y -n thinpoolmeta isula -l 1%VG
Logical volume "thinpoolmeta" created.
```

5. 将新创建的两个逻辑卷转换成 thinpool 以及 thinpool 所使用的 metadata，这样就完成了 thinpool 配置。

```
# lvconvert -y --zero n -c 512K --thinpool isula/thinpool --poolmetadata
isula/thinpoolmeta

WARNING: Converting logical volume isula/thinpool and isula/thinpoolmeta to
thin pool's data and metadata volumes with metadata wiping.
THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)
Converted isula/thinpool to thin pool.
```

二、修改 isulad 配置文件

6. 如果环境之前运行过 isulad，请先备份之前的数据。

```
# mkdir /var/lib/isulad.bk
# mv /var/lib/isulad/* /var/lib/isulad.bk
```

7. 修改配置文件

   这里提供了两种配置方式，用户可根据实际情况的选择合适的方式。

   – 编辑/etc/isulad/daemon.json，配置 storage-driver 字段值为 devicemapper，并配置 storage-opts 字段的相关参数，支持参数请参见参数说明。配置参考如下所示：

```
{
    "storage-driver": "devicemapper"
    "storage-opts": [
        "dm.thinpooldev=/dev/mapper/isula-thinpool",
        "dm.fs=ext4",
          "dm.min_free_space=10%"
    ]
}
```

   – 或者也可以通过编辑/etc/sysconfig/iSulad，在 isulad 启动参数里显式指定，支持参数请参见参数说明。配置参考如下所示：

```
OPTIONS="--storage-driver=devicemapper --storage-opt
dm.thinpooldev=/dev/mapper/isula-thinpool --storage-opt dm.fs=ext4 --
storage-opt dm.min_free_space=10%"
```

8. 启动 isulad，使配置生效。

```
# systemctl start isulad
```

参数说明

storage-opts 支持的参数请参见表 1-1。

表1-1 starage-opts 字段参数说明

参数	是否必选	含义

参数	是否必选	含义
dm.fs	是	用于指定容器使用的文件系统类型。当前必须配置为 ext4，即 dm.fs=ext4
dm.basesize	否	用于指定单个容器的最大存储空间大小，单位为 k/m/g/t/p，也可以使用大写字母，例如 dm.basesize=50G。该参数只在首次初始化时有效。
dm.mkfsarg	否	用于在创建基础设备时指定额外的 mkfs 参数。例如“dm.mkfsarg=-O ^has_journal”
dm.mountopt	否	用于在挂载容器时指定额外的 mount 参数。例如 dm.mountopt=nodiscard
dm.thinpooldev	否	用于指定容器/镜像存储时使用的 thinpool 设备。
dm.min_free_space	否	用于指定最小的预留空间，用百分比表示。例如 dm.min_free_space=10%，表示当剩余存储空间只剩 10%左右时，创建容器等和存储相关操作就会失败。

注意事项

- 配置 devicemapper 时，如果系统上没有足够的空间给 thinpool 做自动扩容，请禁止自动扩容功能。

  禁止自动扩容的方法是把/etc/lvm/profile/isula-thinpool.profile 中 thin_pool_autoextend_threshold 和 thin_pool_autoextend_percent 两个值都改成 100，如下所示：

  ```
  activation {
    thin pool autoextend threshold=100
    thin pool autoextend percent=100
  }
  ```

- 使用 devicemapper 时，容器文件系统必须配置为 ext4，需要在 isulad 的配置参数中加上--storage-opt dm.fs=ext4。
- 当 graphdriver 为 devicemapper 时，如果 metadata 文件损坏且不可恢复，需要人工介入恢复。禁止直接操作或篡改 daemon 存储 devicemapper 的元数据。
- 使用 devicemapper lvm 时，异常掉电导致的 devicemapper thinpool 损坏，无法保证 thinpool 损坏后可以修复，也不能保证数据的完整性，需重建 thinpool。

iSula 开启了 user namespace 特性，切换 devicemapper 存储池时的注意事项

- 一般启动容器时，deviceset-metadata 文件为：/var/lib/isulad/devicemapper/metadata/deviceset-metadata。
- 使用了 user namespace 场景下，deviceset-metadata 文件使用的是：/var/lib/isulad/{userNSUID.GID}/devicemapper/metadata/deviceset-metadata。

- 使用 devicemapper 存储驱动，容器在 user namespace 场景和普通场景之间切换时，需要将对应 deviceset-metadata 文件中的 BaseDeviceUUID 内容清空；针对 thinpool 扩容或者重建的场景下，也同样的需要将对应 deviceset-metadata 文件中的 BaseDeviceUUID 内容清空，否则 isulad 服务会重启失败。

1.2.4 卸载

卸载 iSulad 的操作步骤如下：

1. 卸载 iSulad 及其依赖软件包
   – 若使用 yum 方式安装，卸载的参考命令如下：
   ```
   $ sudo yum remove iSulad
   ```
   – 若使用 rpm 方式安装，需卸载 iSulad 及其依赖包，卸载单个 RPM 包的参考命令如下：
   ```
   sudo rpm -e iSulad-xx.xx.xx-YYYYmmdd.HHMMSS.gitxxxxxxxx.aarch64.rpm
   ```
2. 镜像、容器、volumes 以及相关配置文件不会自动删除，需要手动删除。参考命令如下：
   ```
   $ sudo rm -rf /var/lib/iSulad
   ```

1.3 使用方法

1.3.1 容器管理

1.3.1.1 创建容器

描述

isula create 命令用于创建一个新的容器。容器引擎会使用指定的容器镜像创建容器读写层，或者使用指定的本地 rootfs 作为容器的运行环境。创建完成后，会将容器的 ID 输出到标准输出，后续可以使用 isula start 命令启动该容器。新创建的容器状态为 **inited** 状态

用法

```
isula create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

参数

create 命令支持参数参考下表。

表1-2 create 命令参数列表

命令	参数	说明
create	--annotation	设置容器的 annotations。例如支持 native.umask 选项： `--annotation native.umask=normal`

		`# 启动的容器umask值为0022`
		`--annotation native.umask=secure`
		`# 启动的容器umask值为0027`
		注意如果没有配置该参数，则使用isulad 中的 umask 配置。

	--cap-drop	删除 Linux 权限功能
	--cgroup-parent	指定容器 cgroup 父路径
	--cpuset-cpus	允许执行的 CPU（e.g. 0-3，0，1）
	--cpu-shares	CPU 份额（相对权重）
	--cpu-quota	限制 CPU CFS（完全公平调度器）的配额
	--device=[]	为容器添加一个主机设备
	--dns	添加 DNS 服务器
	--dns-opt	添加 DNS 选项
	--dns-search	设定容器的搜索域
	-e, --env	设置环境变量
	--env-file	通过文件配置环境变量
	--entrypoint	启动容器时要运行的入口点
	--external-rootfs=PATH	指定一个不由 iSulad 管理的 rootfs(可以为文件夹或块设备)给容器
	--files-limit	调整容器内能够打开的文件句柄数（-1 表示不限制）
	--group-add=[]	指定额外的用户组添加到容器
	--help	打印帮助信息
	--health-cmd	在容器内执行的命令
	--health-exit-on-unhealthy	检测到容器非健康时是否杀死容器
	--health-interval	相邻两次命令执行的间隔时间
	--health-retries	健康检查失败最大的重试次数
	--health-start-period	容器初始化时间
	--health-timeout	单次检查命令执行的时间上限
	--hook-spec	钩子配置文件
	-H, --host	指定要连接的 iSulad socket 文件路

		径
	-h, --hostname	容器主机名称
	-i, --interactive	即使没有连接到容器的标准输入，也要保持容器的标准输入打开
	--hugetlb-limit=[]	大页文件限制，例如：--hugetlb-limit 2MB:32MB
	--log-opt=[]	日志驱动程序选项，默认禁用记录容器串口日志功能，可以通过"--log-opt disable-log=false"来开启。
	-l，--label	为容器设置标签
	--lablel-file	通过文件设置容器标签
	-m, --memory	内存限制
	--memory-reservation	设置容器内存限制，默认与--memory 一致。可认为--memory 是硬限制，--memory-reservation 是软限制；当使用内存超过预设值时，会动态调整（系统回收内存时尝试将使用内存降低到预设值以下），但不确保一定不超过预设值。一般可以和--memory 一起使用，数值小于--memory 的预设值，最小设置为4MB。
	--memory-swap	正整数，内存 + 交换空间，-1 表示不限制
	--memory-swappiness	正整数，swappiness 参数值可设置范围在 0 到 100 之间。 此参数值越低，就会让 Linux 系统尽量少用swap 分区，多用内存；参数值越高就是反过来，使内核更多的去使用swap 空间，默认值为-1，表示使用系统默认值。
	--mount	挂载主机目录到容器中
	--no-healthcheck	禁用健康检查配置
	--name=NAME	容器名
	--net=none	容器连接到网络
	--pids-limit	调整容器内能够执行的进程数（-1 表示不限制）
	--privileged	给予容器扩展的特权

<table>
<tr><td></td><td>-R, --runtime</td><td>容器运行时，参数支持"lcr"，忽略大小写，因此"LCR"和"lcr"是等价的</td></tr>
<tr><td></td><td>--read-only</td><td>设置容器的根文件系统为只读</td></tr>
<tr><td></td><td>--restart</td><td>当容器退出时重启策略
系统容器支持--restart on-reboot</td></tr>
<tr><td></td><td>--storage-opt</td><td>配置容器的存储驱动选项</td></tr>
<tr><td></td><td>-t, --tty</td><td>分配伪终端</td></tr>
<tr><td></td><td>--ulimit</td><td>为容器设置 ulimit 限制</td></tr>
<tr><td></td><td>-u, --user</td><td>用户名或 UID，格式[<name|uid>][:<group|gid>]</td></tr>
<tr><td></td><td>-v, --volume=[]</td><td>挂载一个卷</td></tr>
</table>

约束限制

- 使用--user 或--group-add 参数，在容器启动阶段校验 user 或 group 时，容器如果使用的是 OCI 镜像，是从镜像的真实 rootfs 的 etc/passwd 和 etc/group 文件中校验，如果使用的是 rootfs 文件夹或块设备作为容器的 rootfs，则校验的是 host 中的 etc/passwd 和 etc/group 文件；查找时使用的 rootfs 会忽略-v 和--mount 等挂载参数，意味着使用这些参数尝试覆盖 etc/passwd 和 etc/group 两个文件时，在查找阶段不生效，只在容器真正启动时生效。生成的配置保存在"iSulad 根目录/engine/容器 ID/start_generate_config.json"，文件格式如下：

```
{
    "uid": 0,
    "gid": 8,
    "additionalGids": [
        1234,
        8
    ]
}
```

示例

创建一个新容器

```
$ isula create busybox
fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
$ isula ps -a
STATUS PID IMAGE   COMMAND EXIT CODE RESTART COUNT STARTAT FINISHAT RUNTIME ID
NAMES                                                  inited -   busybox "sh"   0
0           -      -       lcr     fd7376591a9c fd7376591a9c4521...
```

1.3.1.2 启动容器

描述

isula start 命令用于启动一个或多个容器。

用法

```
isula start [OPTIONS] CONTAINER [CONTAINER...]
```

参数

start 命令支持参数参考下表。

表1-3 start 命令参数列表

命令	参数	说明
start	-H, --host	指定要连接的 iSulad socket 文件路径
	-R, --runtime	容器运行时，参数支持"lcr"，忽略大小写，因此"LCR"和"lcr"是等价的

示例

启动一个新容器

```
$ isula start fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
```

1.3.1.3 运行容器

描述

isula run 命令命令用于创建一个新的容器。会使用指定的容器镜像创建容器读写层，并且为运行指定的命令做好准备。创建完成后，使用指定的命令启动该容器。run 命令相当于 create 然后 start 容器。

用法

```
isula run [OPTIONS] ROOTFS|IMAGE [COMMAND] [ARG...]
```

参数

run 命令支持参数参考下表。

表1-4 run 命令参数列表

命令	参数	说明

run	--annotation	设置容器的 annotations。例如支持 native.umask 选项：
		`--annotation native.umask=normal # 启动的容器 umask 值为 0022`
		`--annotation native.umask=secure # 启动的容器 umask 值为 0027`
		注意如果没有配置该参数，则使用 isulad 中的 umask 配置。

	--cap-add	添加 Linux 功能
	--cap-drop	删除 Linux 功能
	--cgroup-parent	指定容器 cgroup 父路径
	--cpuset-cpus	允许执行的 CPU（e.g. 0-3，0，1）
	--cpu-shares	CPU 份额（相对权重）
	--cpu-quota	限制 CPU CFS（完全公平调度器）的配额
	-d, --detach	后台运行容器并打印容器 ID
	--device=[]	为容器添加一个主机设备
	--dns	添加 DNS 服务器
	--dns-opt	添加 DNS 选项
	--dns-search	设定容器的搜索域
	-e, --env	设置环境变量
	--env-file	通过文件配置环境变量
	--entrypoint	启动容器时要运行的入口点
	--external-rootfs=PATH	指定一个不由 iSulad 管理的 rootfs(可以为文件夹或块设备)给容器
	--files-limit	调整容器内能够打开的文件句柄数（-1 表示不限制）
	--group-add=[]	指定额外的用户组添加到容器
	--help	打印帮助信息
	--health-cmd	在容器内执行的命令
	--health-exit-on-unhealthy	检测到容器非健康时是否杀死容器

	--health-interval	相邻两次命令执行的间隔时间
	--health-retries	健康检查失败最大的重试次数
	--health-start-period	容器初始化时间
	--health-timeout	单次检查命令执行的时间上限
	--hook-spec	钩子配置文件
	-H, --host	指定要连接的 iSulad socket 文件路径
	-h, --hostname	容器主机名称
	--hugetlb-limit=[]	大页文件限制，例如：--hugetlb-limit 2MB:32MB
	-i, --interactive	即使没有连接到容器的标准输入，也要保持容器的标准输入打开
	--log-opt=[]	日志驱动程序选项，默认禁用记录容器串口日志功能，可以通过"--log-opt disable-log=false"来开启。
	-m, --memory	内存限制
	--memory-reservation	设置容器内存限制，默认与--memory 一致。可认为--memory 是硬限制，--memory-reservation 是软限制；当使用内存超过预设值时，会动态调整（系统回收内存时尝试将使用内存降低到预设值以下），但不确保一定不超过预设值。一般可以和--memory 一起使用，数值小于--memory 的预设值，最小设置为 4MB。
	--memory-swap	正整数，内存 + 交换空间，-1 表示不限制
	--memory-swappiness	正整数，swappiness 参数值可设置范围在 0 到 100 之间。 此参数值越低，就会让 Linux 系统尽量少用 swap 分区，多用内存；参数值越高就是反过来，使内核更多的去使用 swap 空间，默认值为-1，表示使用系统默认值。
	--mount	挂载主机目录到容器中
	--no-healthcheck	禁用健康检查配置
	--name=NAME	容器名

<table>
<tr><td></td><td>--net=none</td><td>容器连接到网络</td></tr>
<tr><td></td><td>--pids-limit</td><td>调整容器内能够执行的进程数（-1表示不限制）</td></tr>
<tr><td></td><td>--privileged</td><td>给予容器扩展的特权</td></tr>
<tr><td></td><td>-R, --runtime</td><td>容器运行时，参数支持"lcr"，忽略大小写，因此"LCR"和"lcr"是等价的</td></tr>
<tr><td></td><td>--read-only</td><td>设置容器的根文件系统为只读</td></tr>
<tr><td></td><td>--restart</td><td>当容器退出时重启策略
系统容器支持--restart on-reboot</td></tr>
<tr><td></td><td>--rm</td><td>当容器退出时自动清理容器</td></tr>
<tr><td></td><td>--storage-opt</td><td>配置容器的存储驱动选项</td></tr>
<tr><td></td><td>-t, --tty</td><td>分配伪终端</td></tr>
<tr><td></td><td>--ulimit</td><td>为容器设置 ulimit 限制</td></tr>
<tr><td></td><td>-u, --user</td><td>用户名或 UID，格式 [<name|uid>][:<group|gid>]</td></tr>
<tr><td></td><td>-v, --volume=[]</td><td>挂载一个卷</td></tr>
</table>

约束限制

- 容器父进程进程退出时，则对应的容器也自动退出。
- 创建普通容器时父进程不能为 init，因为普通容器的权限不够，导致容器可以创建成功，但是 attach 进去的时候会卡住。
- 运行容器时，不指定--net，默认 hostname 为 **localhost**。
- 使用--files-limit 参数传入一个很小的值，如 1 时，启动容器时，iSulad 创建 cgroup 子组后先设置 files.limit 值，再将容器进程的 PID 写入该子组的 cgroup.procs 文件，此时容器进程已经打开超过 1 个句柄，因而写入报错导致启动失败启动容器会失败。
- --mount 参数和--volume 参数同时存在时，如果目的路径有冲突，则--mount 会在--volume 之后挂载(即将--volume 中的挂载点覆盖掉)。

  备注：轻量级容器的参数中 type 支持 bind 或 squashfs，当 type=squashfs 时，src 是镜像的路径；原生 docker 的参数 type 支持 bind、volume、tmpfs。
- restart 重启策略不支持 unless-stopped。
- 以下三种情况与 docker 返回值不一致，docker 返回 127,轻量级容器返回 125

  --device 参数指定主机设备为不存在的设备

  --hook-spec 参数指定不存在的 hook json 文件

  --entrypoint 参数指定不存在的入口参数

- 使用--volume 参数时，由于容器启动时会对/dev/ptmx 设备进行删除重建，因此请勿将/dev 目录挂载至容器/dev 目录，应使用--device 对/dev 下的设备在容器中进行挂载
- 禁止使用 echo 的方式向 run 命令的 stdin 输入数据，会导致客户端卡死。应该直接将 echo 的值作为命令行参数传给容器

```
[root@localhost ~]# echo ls | isula run -i busybox /bin/sh


^C
[root@localhost ~]#
```

上述命令出现客户端卡死现象，这是由于上述命令相当于往 stdin 输入 ls，随后 EOF 被读取，客户端不再发送数据，等待服务端退出，但是服务端无法区分客户端是否需要继续发送数据，因而服务端卡在 read 数据上，最终导致双方均卡死。

正确的执行方式为：

```
[root@localhost ~]# isula run -i busybox ls
bin
dev
etc
home
proc
root
sys
tmp
usr
var
[root@localhost ~]#
```

- 使用 host 的根目录（/）作为容器的文件系统，那么在挂载路径时，如果有如下情况

表1-5 挂载情况

Host 路径（source）	容器路径（dest）
/home/test1	/mnt/
/home/test2	/mnt/abc

须知

第一种情况，先挂载/home/test1，然后挂载/home/test2，这种情况会导致/home/test1 的内容覆盖掉原来/mnt 下面的内容，这样可能导致/mnt 下面不存在 abc 目录，这样会导致挂载/home/test2 到/mnt/abc 失败。

第二种情况，先挂载/home/test2，然后挂载/home/test1。这种情况，第二次的挂载会把/mnt 的内容替换为/home/test1 的内容，这样第一次挂载的/home/test2 到/mnt/abc 的内容就看不到了。

因此，不支持第一种使用方式；第二种使用用户需要了解这种数据无法访问的风险

须知

- 高并发场景（并发启动 200 容器）下，glibc 的内存管理机制会导致内存空洞以及虚拟内存较大（例如 10GB）的问题。该问题是高并发场景下 glibc 内存管理机制的限制，而不是内存泄露，不会导致内存消耗无限增大。可以通过设置 MALLOC_ARENA_MAX 环境变量来减少虚拟内存的问题，而且可以增大减少物理内存的概率。但是这个环境变量会导致 iSulad 的并发性能下降，需要用户根据实际情况做配置。

参考实践情况，平衡性能和内存，可以设置 MALLOC_ARENA_MAX 为 4。（在 arm64 服务器上面对 iSulad 的性能影响在 10%以内）

配置方法：
1. 手动启动 iSulad 的场景，可以直接 export MALLOC_ARENA_MAX=4，然后再启动 iSulad 即可。
2. systemd 管理 iSulad 的场景，可以修改/etc/sysconfig/iSulad，增加一条 MALLOC_ARENA_MAX=4 即可。

示例

运行一个新容器

```
$ isula run -itd busybox
9c2c13b6c35f132f49fb7ffad24f9e673a07b7fe9918f97c0591f0d7014c713b
```

1.3.1.4 停止容器

描述

isula stop 命令用于停止一个或多个运行中的容器。首先向容器中的首进程会发送 **SIGTERM** 信号，在指定时间（默认为 10s）内容器未停止时，会发送 **SIGKILL**。

用法

```
isula stop [OPTIONS] CONTAINER [CONTAINER...]
```

参数

stop 命令支持参数参考下表。

表1-6 stop 命令参数列表

命令	参数	说明
stop	-f, --force	强制停止正在运行的容器
	-H, --host	指定要连接的 iSulad socket 文件路径
	-t, --time	先优雅停止，超过这个时间，则强行终止

约束限制

- 指定 t 参数且 t<0 时，请确保自己容器的应用会处理 stop 信号。

  Stop 的原理：Stop 会首先给容器发送 Stop 信号（SIGTERM），然后等待一定的时间（这个时间就是用户输入的 t），过了指定时间如果容器还仍处于运行状态，那么就发送 kill 信号（SIGKILL）使容器强制退出。

- 输入参数 t 的含义：

  t<0 ：表示一直等待，不管多久都等待程序优雅退出，既然用户这么输入了，表示对自己的应用比较放心，认为自己的程序有 合理的 stop 信号的处理机制。

  t=0 ： 表示不等，立即发送 kill -9 到容器。

  t>0 ： 表示等一定的时间，如果容器还未退出，就发送 kill -9 到容器。

  所以如果用户使用 t<0 （比如 t=-1），请确保自己容器的应用会正确处理 SIGTERM. 如果容器忽略了该信号，会导致 isula stop 一直卡住。

示例

停止一个容器

```
$ isula stop fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
```

1.3.1.5 强制停止容器

描述

isula kill 命令用于强制停止一个或多个运行中的容器。

用法

```
isula kill [OPTIONS] CONTAINER [CONTAINER...]
```

参数

kill 命令支持参数参考下表。

表1-7 kill 命令参数列表

命令	参数	说明
kill	-H, --host	指定要连接的 iSulad socket 文件路径
	-s, --signal	发送给容器的信号

示例

杀掉一个容器

```
$ isula kill fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
```

1.3.1.6 删除容器

描述

isula rm 命令用于删除一个或多个容器。

用法

```
isula rm [OPTIONS] CONTAINER [CONTAINER...]
```

参数

rm 命令支持参数参考下表。

表1-8 rm 命令参数列表

命令	参数	说明
rm	-f, --force	强制移除正在运行的容器
	-H, --host	指定要连接的 iSulad socket 文件路径
	-v, --volume	移除挂载在容器上的卷（备注：目前 iSulad 尚不使用此功能）

约束限制

- 在 IO 正常情况，空环境（只有 1 个容器）删除一个 running 容器的时间为 T1，200 个容器的环境（容器无大量 IO 操作，host IO 正常）删除一个 running 容器所需时间为 T2。T2 的规格为：T2 = max { T1 * 3， 5 } 秒钟。

示例

删除一个停止状态的容器

```
$ isula rm fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
```

1.3.1.7 接入容器

描述

isula attach 命令用于将当前终端的标准输入、标准输出和标准错误连接到正在运行的容器。仅支持 runtime 类型为 lcr 的容器。

用法

```
isula attach [OPTIONS] CONTAINER
```

参数

attach 命令支持参数参考下表。

表1-9 attach 命令参数列表

命令	参数	说明
attach	--help	打印帮助信息
	-H, --host	指定要连接的 iSulad socket 文件路径
	-D, --debug	开启 debug 模式

约束限制

- 原生 docker attach 容器会直接进入容器，而 isulad attach 容器后需要敲一个回车才进入。

示例

接入一个运行状态的容器

```
$ isula attach fd7376591a9c3d8ee9a14f5d2c2e5255b02cc44cddaabca82170efd4497510e1
/ #
/ #
```

1.3.1.8 重命名容器

描述

isula rename 命令用于重命名容器。

用法

```
isula rename [OPTIONS] OLD_NAME NEW_NAME
```

参数

rename 命令支持参数参考下表。

表1-10 rename 命令参数列表

命令	参数	说明
rename	-H, --host	重命名容器

示例

重命名一个容器

```
$ isula rename my_container my_new_container
```

1.3.1.9 在容器中执行新命令

描述

isula exec 命令用于正在运行的容器中运行一个新命令。新执行的命令将在容器的默认目录中运行。如果基础镜像指定了自定义目录，则将使用该目录。

用法

```
isula exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

参数

exec 命令支持参数参考下表。

表1-11 exec 命令参数列表

命令	参数	说明
exec	-d, --detach	后台运行命令
	-e, --env	设置环境变量（备注：目前 iSulad 尚不使用此功能）
	-H, --host	指定要连接的 iSulad socket 文件路径
	-i, --interactive	没有连接，也要保持标准输入打开（备注：目前 iSulad 尚不使用此功能）
	-t, --tty	分配伪终端（备注：目前 iSulad 尚不使用此功能）
	-u, --user	指定用户登录容器执行命令

约束限制

- isula exec 不指定任何参数时，会默认使用-it 参数， 表示分配一个伪终端，以交互式的方式进入容器
- 当使用 isula exec 执行脚本，在脚本中执行后台进程时，需使用 nohup 标志忽略 SIGHUP 信号。

使用 isula exec 运行脚本,在脚本中运行后台进程需使用 nohup 标志。否则内核会在 exec 执行的进程（session 首进程）退出时，向后台执行的进程发送 SIGHUP 信号，导致后台进程退出，出现僵尸进程。

- isula exec 进入容器进程后，不能执行后台程序，否则会出现卡死现象。

  isula exec 执行后台进程的方式如下：

  a. 使用 isula exec 进入容器终端，isula exec container_name bash
  b. 进入容器后，执行 script &
  c. 执行 exit，导致终端卡死

```
isula exec 进入容器后，执行后台程序卡住的原因为 isula exec 进入容器运行后台 while1 程序，当bash 退出时，while1 程序并不会退出，变为孤儿进程由 1 号
进程接管，while1 程序是由容器的初始 bash 进程 fork &exec 执行的，while1 进程复制了 bash 进程的文件句柄，导致 bash 退出时，句柄并未完全关闭，导致
console 进程收不到句柄关闭事件，epoll_wait 卡住，进程不退出。
```

- isula exec 不能用后台方式执行，否则可能会出现卡死现象。

  isula exec 后台执行的方式如下：

  使用 **isula exec 脚本 &** 的方式后台执行 exec，如：isula exec container_name script & ，isula exec 后台执行，执行的脚本中不断 cat 某一文件，正常时在当前终端有输出，如果在当前终端执行回车操作，客户端会因读 IO 失败而退出读 stdout 的动作，使终端不再输出，服务端由于进程仍然在 cat 文件，会继续往 fifo 的 buffer 里写入数据，当缓存写满时，容器内进程会卡死在 write 动作上。

- 轻量级容器使用 exec 执行带有管道操作的命令时，建议使用/bin/bash -c 方式执行该命令。

  典型应用场景：

  使用 isula exec container_name -it ls /test | grep "xx" | wc -l,用于统计 test 目录下 xx 的文件个数，因 exec 执行的为"ls /test",其输出通过管道进行 grep、wc 处理。exec 执行的为"ls /test"的输出会换行，再针对该输出进行处理时，结果有误。

  原因：使用 exec 执行 ls /test,输出带有换行，针对该输出进行“| grep "xx" | wc -l “，处理结果为 2（两行）

```
[root@localhost ~]# isula exec  -it container ls /test
xx    xx10  xx12  xx14  xx3   xx5   xx7   xx9
xx1   xx11  xx13  xx2   xx4   xx6   xx8
[root@localhost ~]#
```

  建议处理方式：使用 run/exec 执行带有管道操作的命令时，使用/bin/bash -c 执行命令，在容器中执行管道操作。

```
[root@localhost ~]# isula exec  -it container  /bin/sh -c "ls /test | grep "xx"
| wc -l"
15
[root@localhost ~]#
```

- 禁止使用 echo 的方式向 exec 命令的 stdin 输入数据，会导致客户端卡死。应该直接将 echo 的值作为命令行参数传给容器

```
[root@localhost ~]# echo ls | isula exec 38 /bin/sh


^C
[root@localhost ~]#
```

上述命令可能出现客户端卡死现象，这是由于上述命令相当于往 stdin 输入 ls，随后 EOF 被读取，客户端不再发送数据，等待服务端退出，但是服务端无法区分客户端是否需要继续发送数据，因而服务端卡在 read 数据上，最终导致双方均卡死。

正确的执行方式为：

```
[root@localhost ~]# isula exec 38 ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

示例

在运行中的容器中，执行 echo 命令

```
$ isula exec c75284634bee echo "hello,world"
hello,world
```

1.3.1.10 查询单个容器信息

描述

isula inspect 提供了容器的详细信息。

用法

```
isula inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]
```

参数

inspect 命令支持参数参考下表。

表1-12 inspect 命令参数列表

命令	参数	说明
inspect	-H, --host	指定要连接的 iSulad socket 文件路径
	-f, --format	使用模板格式化输出
	-t, --time	超时时间的秒数，若在该时间内 inspect 查询容器信息未执行成功，则停止等待并立即报错，默认为 120 秒，当配置小于等于 0 的值，表示不启用 timeout 机制 inspect 查询容器信息会一直等待，直到获取容器信息成功后返回。

约束限制

- 轻量级容器不支持 format 为“{{.State}}”的格式化输出，支持“{{json .State}}”的 json 格式化输出。当 inspect 镜像时，不支持-f 参数。

示例

查询容器信息

```
$ isula inspect c75284634bee
[
    {
        "Id": "c75284634beeede3ab86c828790b439d16b6ed8a537550456b1f94eb852c1c0a",
        "Created": "2019-08-01T22:48:13.993304927-04:00",
        "Path": "sh",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "Pid": 21164,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2019-08-02T06:09:25.535049168-04:00",
            "FinishedAt": "2019-08-02T04:28:09.479766839-04:00",
            "Health": {
                "Status": "",
                "FailingStreak": 0,
                "Log": []
            }
        },
        "Image": "busybox",
        "ResolvConfPath": "",
        "HostnamePath": "",
        "HostsPath": "",
        "LogPath": "none",
        "Name": "c75284634beeede3ab86c828790b439d16b6ed8a537550456b1f94eb852c1c0a",
        "RestartCount": 0,
        "HostConfig": {
            "Binds": [],
            "NetworkMode": "",
            "GroupAdd": [],
            "IpcMode": "",
            "PidMode": "",
            "Privileged": false,
            "SystemContainer": false,
            "NsChangeFiles": [],
            "UserRemap": "",
            "ShmSize": 67108864,
            "AutoRemove": false,
            "AutoRemoveBak": false,
            "ReadonlyRootfs": false,
            "UTSMode": "",
            "UsernsMode": "",
            "Sysctls": {},
            "Runtime": "lcr",
            "RestartPolicy": {
                "Name": "no",
                "MaximumRetryCount": 0
            },
```

```
        "CapAdd": [],
        "CapDrop": [],
        "Dns": [],
        "DnsOptions": [],
        "DnsSearch": [],
        "ExtraHosts": [],
        "HookSpec": "",
        "CPUShares": 0,
        "Memory": 0,
        "OomScoreAdj": 0,
        "BlkioWeight": 0,
        "BlkioWeightDevice": [],
        "CPUPeriod": 0,
        "CPUQuota": 0,
        "CPURealtimePeriod": 0,
        "CPURealtimeRuntime": 0,
        "CpusetCpus": "",
        "CpusetMems": "",
        "SecurityOpt": [],
        "StorageOpt": {},
        "KernelMemory": 0,
        "MemoryReservation": 0,
        "MemorySwap": 0,
        "OomKillDisable": false,
        "PidsLimit": 0,
        "FilesLimit": 0,
        "Ulimits": [],
        "Hugetlbs": [],
        "HostChannel": {
            "PathOnHost": "",
            "PathInContainer": "",
            "Permissions": "",
            "Size": 0
        },
        "EnvTargetFile": "",
        "ExternalRootfs": ""
    },
    "Mounts": [],
    "Config": {
        "Hostname": "localhost",
        "User": "",
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
            "TERM=xterm",
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Tty": true,
        "Cmd": [
            "sh"
        ],
        "Entrypoint": [],
        "Labels": {},
        "Annotations": {
            "log.console.file": "none",
            "log.console.filerotate": "7",
```

```
            "log.console.filesize": "1MB",
            "rootfs.mount": "/var/lib/isulad/mnt/rootfs",
            "native.umask": "secure"
        },
        "HealthCheck": {
            "Test": [],
            "Interval": 0,
            "Timeout": 0,
            "StartPeriod": 0,
            "Retries": 0,
            "ExitOnUnhealthy": false
        }
    },
    "NetworkSettings": {
        "IPAddress": ""
    }
  }
]
```

1.3.1.11 查询所有容器信息

描述

isula ps 用于查询所有容器的信息。

用法

```
isula ps [OPTIONS]
```

参数

ps 命令支持参数参考下表。

表1-13 ps 命令参数列表

命令	参数	说明
ps	-a, --all	显示所有的容器
	-H, --host	指定要连接的 iSulad socket 文件路径
	-q, --quiet	只显示容器名字
	-f, --filter	增加筛选过滤条件
	--format	按照模板声明的方式输出数据
	--no-trunc	不对容器 ID 进行截断打印

示例

查询所有容器信息

```
$ isula ps -a

ID          IMAGE                              STATUS  PID    COMMAND EXIT CODE
RESTART COUNT STARTAT       FINISHAT    RUNTIME NAMES
e84660aa059c rnd-dockerhub.huawei.com/official/busybox running 304765 "sh"    0
0           13 minutes ago -         lcr
e84660aa059cafb0a77a4002e65cc9186949132b8e57b7f4d76aa22f28fde016
$ isula ps -a --format "table {{.ID}} {{.Image}}" --no-trunc
ID                                                           IMAGE
e84660aa059cafb0a77a4002e65cc9186949132b8e57b7f4d76aa22f28fde016 rnd-
dockerhub.huawei.com/official/busybox
```

1.3.1.12 重启容器

描述

isula restart 用于重启一个或者多个容器。

用法

```
isula restart [OPTIONS] CONTAINER [CONTAINER...]
```

参数

restart 命令支持参数参考下表。

表1-14 restart 命令参数列表

命令	参数	说明
restart	-H, --host	指定要连接的 iSulad socket 文件路径
	-t, --time	先优雅停止，超过这个时间，则强行终止

约束限制

- 指定 t 参数且 t<0 时，请确保自己容器的应用会处理 stop 信号。

  restart 会首先调用 stop 停止容器。stop 会首先给容器发送 stop 信号（SIGTERM），然后等待一定的时间（这个时间就是用户输入的 t），过了一定时间如果容器仍处于运行状态，那么就发送 kill 信号（SIGKILL）使容器强制退出。

- 输入参数 t 的含义：

  t<0 ：表示一直等待，不管多久都等待程序优雅退出，既然用户这么输入了，表示对自己的应用比较放心，认为自己的程序有合理的 stop 信号的处理机制。

t=0 ： 表示不等，立即发送 kill -9 到容器。

t>0 ： 表示等一定的时间，如果容器还未退出，就发送 kill -9 到容器。

所以如果用户使用 t<0（比如 t=-1），请确保自己容器的应用会正确处理 SIGTERM. 如果容器忽略了该信号，会导致 isula restart 一直卡住。

示例

重启单个容器

```
$ isula restart c75284634beeede3ab86c828790b439d16b6ed8a537550456b1f94eb852c1c0a
 c75284634beeede3ab86c828790b439d16b6ed8a537550456b1f94eb852c1c0a
```

1.3.1.13 等待容器退出

描述

isula wait 用于等待一个或者多个容器退出。仅支持 runtime 类型为 lcr 的容器。

用法

```
isula wait [OPTIONS] CONTAINER [CONTAINER...]
```

参数

wait 命令支持参数参考下表。

表1-15 wait 命令参数列表

命令	参数	说明
wait	-H, --host	指定要连接的 iSulad socket 文件路径
	/	阻塞，直到容器停止，然后打印退出代码

示例

等待单个容器退出

```
$ isula wait c75284634beeede3ab86c828790b439d16b6ed8a537550456b1f94eb852c1c0a
 137
```

1.3.1.14 查看容器中进程信息

描述

isula top 用于查看容器中的进程信息。仅支持 runtime 类型为 lcr 的容器。

用法

```
isula top [OPTIONS] container [ps options]
```

参数

top 命令支持参数参考下表。

表1-16 top 命令参数列表

命令	参数	说明
top	-H, --host	指定要连接的 iSulad socket 文件路径
	/	查询运行容器的进程信息

示例

查询容器中进程信息

```
$ isula top 21fac8bb9ea8e0be4313c8acea765c8b4798b7d06e043bbab99fc20efa72629c
UID       PID  PPID  C STIME TTY         TIME CMD
root    22166 22163  0 23:04 pts/1    00:00:00 sh
```

1.3.1.15 查看容器使用的资源

描述

isula stats 用于实时显示资源使用的统计信息。仅支持 runtime 类型为 lcr 的容器。

用法

```
isula stats [OPTIONS] [CONTAINER...]
```

参数

stats 命令支持参数参考下表。

表1-17 stats 命令参数列表

命令	参数	说明
stats	-H, --host	指定要连接的 iSulad socket 文件路径
	-a, --all	显示所有容器（默认只显示运行中的容器）
	--no-stream	非流式方式的 stats，只打印第一次结果

示例

显示资源使用的统计信息

```
$ isula stats --no-stream
21fac8bb9ea8e0be4313c8acea765c8b4798b7d06e043bbab99fc20efa72629c
CONTAINER       CPU %     MEM USAGE / LIMIT       MEM %     BLOCK I / O
PIDS
21fac8bb9ea8     0.00      56.00 KiB / 7.45 GiB      0.00      0.00 B / 0.00 B
1
```

1.3.1.16 获取容器日志

描述

isula logs 用于获取容器的日志。仅支持 runtime 类型为 lcr 的容器。

用法

```
isula logs [OPTIONS] [CONTAINER...]
```

参数

logs 命令支持参数参考下表。

表1-18 logs 命令参数列表

命令	参数	说明
logs	-H, --host	指定要连接的 iSulad socket 文件路径
	-f, --follow	跟踪日志输出
	--tail	显示日志行数

约束限制

- 容器串口 logs 日志记录功能，默认为开启状态，需要关闭可以通过 isula create --log-opt disable-log=true 或 isula run --log-opt disable-log=true 关闭。

示例

获取容器日志

```
$ isula logs 6a144695f5dae81e22700a8a78fac28b19f8bf40e8827568b3329c7d4f742406
hello, world
hello, world
hello, world
```

1.3.1.17 容器与主机之间数据拷贝

描述

isula cp 用于容器与主机之间的数据拷贝，仅支持 runtime 类型为 lcr 的容器。

用法

```
isula cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH
isula cp [OPTIONS] SRC_PATH CONTAINER:DEST_PATH
```

参数

cp 命令支持参数参考下表。

表1-19 cp 命令参数列表

命令	参数	说明
cp	-H, --host	指定要连接的 iSulad socket 文件路径

约束限制

- iSulad 在执行拷贝时，不会挂载/etc/hostname， /etc/resolv.conf，/etc/hosts 三个文件，也不会对--volume 和--mount 参数传入的参数挂载到 host，所以对这些文件的拷贝使用的是镜像中的原始文件，而不是真实容器中的文件。

```
[root@localhost tmp]# isula cp b330e9be717a:/etc/hostname /tmp/hostname
[root@localhost tmp]# cat /tmp/hostname
[root@localhost tmp]#
```

- iSulad 在解压文件时，不会对文件系统中即将被覆盖的文件或文件夹做类型判断，而是直接覆盖，所以在拷贝时，如果源为文件夹，同名的文件会被强制覆盖为文件夹；如果源为文件，同名的文件夹会被强制覆盖为文件。

```
[root@localhost tmp]# rm -rf /tmp/test_file_to_dir && mkdir
/tmp/test_file_to_dir
[root@localhost tmp]# isula exec b330e9be717a /bin/sh -c "rm -rf
/tmp/test_file_to_dir && touch /tmp/test_file_to_dir"
[root@localhost tmp]# isula cp b330e9be717a:/tmp/test_file_to_dir /tmp
[root@localhost tmp]# ls -al /tmp | grep test_file_to_dir
-rw-r-----    1 root     root            0 Apr 26 09:59 test_file_to_dir
```

- iSulad 在 cp 拷贝过程中，会将容器 freeze 住，在拷贝完成后，恢复容器运行。

示例

将主机/test/host 目录拷贝到容器 21fac8bb9ea8 的/test 目录下。

```
isula cp /test/host 21fac8bb9ea8:/test
```

将容器 21fac8bb9ea8 的/www 目录拷贝到主机的/tmp 目录中。

```
isula cp 21fac8bb9ea8:/www /tmp/
```

1.3.1.18 暂停容器

描述

isula pause 用于暂停容器中所有的进程，仅支持 runtime 类型为 lcr 的容器。

用法

```
isula pause CONTAINER [CONTAINER...]
```

参数

命令	参数	说明
pause	-H, --host	指定要连接的 iSulad socket 文件路径

约束限制

- 只有状态为 running 的容器可以被执行 pause 操作
- 当容器被 pause 后，无法执行其他生命周期管理操作（如 restart/exec/attach/kill/stop/rm 等）
- 当带有健康检查配置的容器被 pause 后，容器状态最终变为 unhealthy 状态

示例

暂停一个正在运行的容器，命令示例如下：

```
$ isula pause 8fe25506fb5883b74c2457f453a960d1ae27a24ee45cdd78fb7426d2022a8bac
 8fe25506fb5883b74c2457f453a960d1ae27a24ee45cdd78fb7426d2022a8bac
```

1.3.1.19 恢复容器

描述

isula unpause 用于恢复容器中所有的进程，为 isula pause 的逆过程，仅支持 runtime 类型为 lcr 的容器。

用法

```
isula unpause CONTAINER [CONTAINER...]
```

参数

命令	参数	说明
pause	-H, --host	指定要连接的 iSulad socket 文件路

		径

约束限制

- 只有状态为 paused 的容器可以被执行 unpause 操作

示例

恢复一个被暂停的容器，命令示例如下：

```
$ isula unpause 8fe25506fb5883b74c2457f453a960d1ae27a24ee45cdd78fb7426d2022a8bac
 8fe25506fb5883b74c2457f453a960d1ae27a24ee45cdd78fb7426d2022a8bac
```

1.3.1.20 从服务端实时获取事件消息

描述

isula events 用于从服务端实时获取容器镜像生命周期、运行等事件消息，仅支持 runtime 类型为 lcr 的容器。

用法

```
isula events [OPTIONS]
```

参数

命令	参数	说明
events	-H, --host	指定要连接的 iSulad socket 文件路径
	-n, --name	获取指定容器的事件消息
	-S, --since	获取指定时间以来的事件消息

示例

从服务端实时获取事件消息，命令示例如下：

```
$ isula events
```

1.3.2 支持 CNI 网络

1.3.2.1 描述

实现 CRI 接口对接 CNI 网络的能力，包括 CNI 网络配置文件的解析、CNI 网络的加入和退出。Pod 需要支持网络时，例如通过 canal 等容器网络插件提供网络能力，那么需要 CRI 接口能够和 canal 实现对接，并且调用 canal 的接口，为 Pod 提供网络能力。

1.3.2.2 接口

CNI 对用户可见的接口，主要涉及 CNI 网络配置和 Pod 配置中 CNI 网络相关的项。

- CNI 网络配置相关的接口，主要是 isulad 指定 CNI 网络配置文件所在路径、CNI 网络插件二进制文件所在的路径以及使用的网络模式。详情请参见表 1-20。
- Pod 配置中 CNI 网络相关的项，主要是设置 Pod 加入的附加 CNI 网络列表，默认情况 Pod 只会加入到 default CNI 网络平面中，可以通过该配置把 Pod 加入到多个 CNI 网络平面中。

表1-20 CNI 网络配置接口

功能	命令行	配置文件	说明
设置 CNI 网络插件二进制文件所在路径	--cni-bin-dir	"cni-bin-dir": "",	默认为/opt/cni/bin
设置 CNI 网络配置文件所在路径	--cni-conf-dir	"cni-conf-dir": "",	系统会遍历目录下面所有后缀名为".conf"、".conflist"和".json"的文件。默认为/etc/cni/net.d
指定网络模式	--network-plugin	"network-plugin": "",	指定网络插件，默认为空字符，表示无网络配置，创建的 sandbox 只有 loop 网卡。支持 cni 和空字符，其他非法值会导致 isulad 启动失败。

附加 CNI 网络配置方式：

在 Pod 的配置文件的"annotations"中，增加一项"network.alpha.kubernetes.io/network": "网络平面配置";

网络平面配置为 json 格式，包含两项：

- name：指定 CNI 网络平面的名字
- interface：指定网络接口的名字

附件 CNI 网络配置方式示例如下：

```
"annotations" : {
      "network.alpha.kubernetes.io/network": "{\"name\": \"mynet\", \"interface\": \"eth1\"}"
 }
```

1.3.2.2.1 CNI 网络配置说明

CNI 网络配置包含两种类型，文件格式都为 json：

- 单网络平面配置，以.conf 和.json 为后缀的文件：具体的配置项请参见附录的表 1-28。
- 多网络平面配置，以.conflist 为后缀的文件：具体的配置项请参见附录的表 1-30。

1.3.2.2.2 加入 CNI 网络列表

如果 iSulad 配置了--network-plugin=cni，而且设置了 default 网络平面配置，那么在启动 Pod 的时候，会自动把 Pod 加入到 default 网络平面。如果在 Pod 的配置中配置了附加网络配置，那么启动 Pod 的时候也会把 Pod 加入到这些附加网络平面中。

Pod 配置中和网络相关的还有 port_mappings 项，用于设置 Pod 的端口映射关系。配置方式如下：

```
"port mappings":[
    {
       "protocol": 1,
       "container port": 80,
       "host port": 8080
     }
]
```

- protocal：表示映射使用的协议，支持 tcp（用 0 标识）、udp（用 1 标识）；
- container_port：表示容器映射出去的 port；
- host_port：表示映射到主机的 port。

1.3.2.2.3 退出 CNI 网络列表

StopPodSandbox 的时候，会调用退出 CNI 网络的接口，清理网络相关的资源。

说明

1. 在调用 RemovePodSandbox 接口之前，至少要调用一次 StopPodSandbox 接口
2. StopPodSandbox 调用 CNI 接口失败，可能导致的网络资源残留。

1.3.2.3 使用限制

- cniVersion 的版本，当前只支持 0.3.0 和 0.3.1。由于后期可能需要支持 0.1.0 和 0.2.0，错误日志打印时，保留了 0.1.0 和 0.2.0 的提示信息。
- name：必须是小写字符、数字、'-'以及'.'组成； '.'和'-'不能作为首字符和尾字符；而且长度不超过 200 个字符。
- 配置文件个数不超过 200 个，单个配置文件大小不超过 1MB。
- 扩展之后的参数，需要根据实际网络需求来配置，不需要使用的可选参数可以不写入到 netconf.json 文件中。

1.3.3 容器资源管理

1.3.3.1 资源的共享

描述

容器间或者容器与 host 之间可以共享 namespace 信息，包括 pid, net, ipc, uts。

用法

isula create/run 时使用 namespace 相关的参数共享资源，具体参数见下方参数列表。

参数

create/run 时可以指定下列参数。

参数项	参数说明	取值范围	是否必选
--pid	指定要共享的 pid namespace	[none, host, container:<containerID>]，none 表示不共享，host 表示与 host 共用 namespace，container:<containerID>表示与容器 containerID 共享同一个 namespace	否
--net	指定要共享的 net namespace	[none, host, container:<containerID>]，none 表示不共享，host 表示与 host 共用 namespace，container:<containerID>表示与容器 containerID 共享同一个 namespace	否
--ipc	指定要共享的 ipc namespace	[none, host, container:<containerID>]，none 表示不共享，host 表示与 host 共用 namespace，container:<containerID>表示与容器 containerID 共享同一个 namespace	否
--uts	指定要共享的 uts namespace	[none, host, container:<containerID>]，none 表示不共享，host 表示与 host 共用 namespace，container:<containerID>表示与容器 containerID 共享同一个 namespace	否

示例

如果两个容器需要共享同一个 pid namespace，在运行容器时，直接加上--pid container:<containerID> 即可，如：

```
isula run -tid --name test pid busybox sh
isula run -tid --name test --pid container:test_pid busybox sh
```

1.3.3.2 限制运行时的 CPU 资源

描述

可以通过参数限制容器的各项 cpu 资源值。

用法

isula create/run 时使用 cpu 相关的参数限制容器的各项 cpu 资源值，具体参数及取值见下方参数列表。

参数

create/run 时可以指定下列参数。

参数项	参数说明	取值范围	是否必选
--cpu-period	限制容器中 cpu cfs（完全公平调度）周期	64 位整数(int64)	否
--cpu-quota	限制容器中 cpu cfs(完全公平调度)的配额	64 位整数(int64)	否
--cpu-shares	限制容器中 cpu 相对权重	64 位整数(int64)	否
--cpuset-cpus	限制容器中使用 cpu 节点	字符串。值为要设置的 cpu 编号，有效范围为主机上的 cpu 数量，例如可以设置 0-3 或者 0,1.	否
--cpuset-mems	限制容器中 cpuset 使用的 mem 节点	字符串。值为要设置的 cpu 编号，有效范围为主机上的 cpu 数量，例如可以设置 0-3 或者 0,1.	否

示例

如果需要限制容器只是用特定的 cpu，在运行容器时，直接加上--cpuset-cpus number 即可，如：

```
isula run -tid --cpuset-cpus 0,2-3 busybox sh
```

说明

是否设置成功，请参见“查询单个容器信息”章节。

1.3.3.3 限制运行时的内存

描述

可以通过参数限制容器的各项内存值上限。

用法

isula create/run 时使用内存相关的参数限制容器的各项内存使用上限，具体参数及取值见下方参数列表。

参数

create/run 时可以指定下列参数。

参数项	参数说明	取值范围	是否必选
--memory	限制容器中内存使用上限	64 位整数(int64)。值为非负数，0 表示不设置（不限制）；单位可以为空(byte)，KB，MB，GB，TB，PB.	否
--memory-reservation	限制容器中内存的软上限	64 位整数(int64)。值为非负数，0 表示不设置（不限制）；单位可以为空(byte)，KB，MB，GB，TB，PB.	否
--memory-swap	限制容器中交换内存的上限	64 位整数(int64)。值为-1 或非负数，-1 表示不限制，0 表示不设置（不限制）；单位可以为空(byte)，KB，MB，GB，TB，PB.	否

参数项	参数说明	取值范围	是否必选
--kernel-memory	限制容器中内核内存的上限	64 位整数(int64)。值为非负数，0 表示不设置（不限制）；单位可以为空(byte)，KB，MB，GB，TB，PB.	否

示例

如果需要限制容器内内存的上限，在运行容器时，直接加上--memory <number>[<unit>]即可，如：

```
isula run -tid --memory 1G busybox sh
```

1.3.3.4 限制运行时的 IO 资源

描述

可以通过参数限制容器中设备读写速度。

用法

isula create/run 时使用--device-read-bps/--device-write-bps <device-path>:<number>[<unit>]来限制容器中设备的读写速度。

参数

create/run 时指定--device-read/write-bps 参数。

参数项	参数说明	取值范围	是否必选
--device-read-bps/--device-write-bps	限制容器中设备的读速度/写速度	64 位整数(int64)。值为正整数，可以为 0，0 表示不设置（不限制）；单位可以为空(byte)，KB，MB，GB，TB，PB.	否

示例

如果需要限制容器内设备的读写速度，在运行容器时，直接加上--device-write-bps/--device-read-bps <device-path>:<number>[<unit>]即可，例如，限制容器 busybox 内设备/dev/sda 的读速度为 1MB 每秒，则命令如下：

```
isula run -tid --device-write /dev/sda:1mb busybox sh
```

限制写速度的命令如下：

```
isula run -tid read-bps /dev/sda:1mb busybox sh
```

1.3.3.5 限制容器 rootfs 存储空间

描述

在 ext4 上使用 overlay2 时，可以设置单个容器的文件系统限额，比如设置 A 容器的限额为 5G，B 容器为 10G。

该特性通过 ext4 文件系统的 project quota 功能来实现，在内核支持的前提下，通过系统调用 SYS_IOCTL 设置某个目录的 project ID，再通过系统调用 SYS_QUOTACTL 设置相应的 project ID 的 hard limit 和 solft limit 值达到限额的目的。

用法

1. 环境准备

   文件系统支持 Project ID 和 Project Quota 属性，4.19 版本内核已经支持，外围包 e2fsprogs 版本不低于 1.43.4-2。

2. 在容器挂载 overlayfs 之前，需要对不同容器的 upper 目录和 work 目录设置不同的 project id，同时设置继承选项，在容器挂载 overlayfs 之后不允许再修改 project id 和继承属性。
3. 配额的设置需要在容器外以特权用户进行。
4. daemon 中增加如下配置

   ```
   -s overlay2 --storage-opt overlay2.override_kernel_check=true
   ```

5. daemon 支持以下选项，用于为容器设置默认的限制，

   --storage-opt overlay2.basesize=128M 指定默认限制的大小，若 isula run 时也指定了--storeage-opt size 选项，则以 run 时指定来生效，若 daemon 跟 isula run 时都不指定大小，则表示不限制。

6. 需要开启文件系统 Project ID 和 Project Quota 属性。
   - 新格式化文件系统并 mount

     ```
     # mkfs.ext4 -O quota,project /dev/sdb
     # mount -o prjquota /dev/sdb /var/lib/isulad
     ```

参数

create/run 时指定--storage-opt 参数。

参数项	参数说明	取值范围	是否必选
--storage-opt size=${rootfsSize}	限制容器 rootfs 存储空间。	rootfsSize 解析出的大小为 int64 范围内以字节表示的正数，默认单位为 B，也可指定为 ([kKmMgGtTpP])?[iI]?[bB]?$	否

示例

在 isula run/create 命令行上通过已有参数“--storage-opt size=<value>”来设置限额。其中 value 是一个正数，单位可以是[kKmMgGtTpP]?[iI]?[bB]?，在不带单位的时候默认单位是字节。

```
$ [root@localhost ~]# isula run -ti --storage-opt size=10M busybox
/ # df -h
Filesystem                Size      Used Available Use% Mounted on
overlay                  10.0M     48.0K     10.0M   0% /
none                     64.0M         0     64.0M   0% /dev
none                     10.0M         0     10.0M   0% /sys/fs/cgroup
tmpfs                    64.0M         0     64.0M   0% /dev
shm                      64.0M         0     64.0M   0% /dev/shm
/dev/mapper/vg--data-ext41
                          9.8G     51.5M      9.2G   1% /etc/hostname
/dev/mapper/vg--data-ext41
                          9.8G     51.5M      9.2G   1% /etc/resolv.conf
/dev/mapper/vg--data-ext41
                          9.8G     51.5M      9.2G   1% /etc/hosts
tmpfs                     3.9G         0      3.9G   0% /proc/acpi
tmpfs                    64.0M         0     64.0M   0% /proc/kcore
tmpfs                    64.0M         0     64.0M   0% /proc/keys
tmpfs                    64.0M         0     64.0M   0% /proc/timer_list
tmpfs                    64.0M         0     64.0M   0% /proc/sched_debug
tmpfs                     3.9G         0      3.9G   0% /proc/scsi
tmpfs                    64.0M         0     64.0M   0% /proc/fdthreshold
tmpfs                    64.0M         0     64.0M   0% /proc/fdenable
tmpfs                     3.9G         0      3.9G   0% /sys/firmware
/ #
/ # dd if=/dev/zero of=/home/img bs=1M count=12 && sync
dm-4: write failed, project block limit reached.
10+0 records in
9+0 records out
10432512 bytes (9.9MB) copied, 0.011782 seconds, 844.4MB/s
/ # df -h | grep overlay
overlay                  10.0M     10.0M         0 100% /
/ #
```

约束

1. 限额只针对 rw 层。

   overlay2 的限额是针对容器的 rw 层的，镜像的大小不计算在内。

2. 内核支持并使能。

   内核必须支持 ext4 的 project quota 功能，并在 mkfs 的时候要加上-O quota,project，挂载的时候要加上-o prjquota。任何一个不满足，在使用--storage-opt size=<value>时都将报错。

   ```
   $ [root@localhost ~]# isula run -it --storage-opt size=10Mb busybox df -h
   Error response from daemon: Failed to prepare rootfs with error: time="2019-04-09T05:13:52-04:00" level=fatal msg="error creating read-
   write layer with ID
   "a4c0e55e82c55e4ee4b0f4ee07f80cc2261cf31b2c2dfd628fa1fb00db97270f": --storage-
   ```

```
opt is supported only for overlay over
xfs or ext4 with 'pquota' mount option"
```

3. 限制额度的说明。
   a. 限制的额度大于 isulad 的 root 所在分区的 size 时，在容器内用 df 看到的文件系统的额度是 isulad 的 root 所在分区的 size，而不是设置的限额。
   b. --storage-opt size=0 代表不限制，且设置值不能小于 4096。size 的精度为 1 个字节，如果指定精度含小数个字节，小数部分被忽略，如指定 size=0.1 实际等同于 size=0 不限制。（受计算机存储浮点数精度的限制，即 0.9999999999999999999999999999 与 1 是等价的，具体的 9 的个数不同计算机可能存在差异，故设置 4095.9999999999999999999999999 与 4096 等价，其它情况类似），注意 isula inspect 显示原始命令行指定形式，如果含小数字节，需自行忽略小数部分。
   c. 限制的额度过小时，比如--storage-opt size=4k，可能会导致容器无法启动，因为启动容器本身需要创建一些文件。
   d. 上一次启动 isulad 时，isulad 的 root 所在分区挂载时加了-o prjquota 选项，这次启动时不加，那么上一次启动中创建的带 quota 的容器的设置值不生效。
   e. daemon 端配额--storage-opt overlay2.basesize，其取值范围与--storage-opt size 相同。
4. 指定 storage-opt 为 4k 时，轻量级容器启动与 docker 有差异

   使用选项 storage-opt size=4k 和镜像 rnd-dockerhub.huawei.com/official/ubuntu-arm64:latest 运行容器。

   docker 启动失败。

```
[root@localhost ~]# docker run -itd --storage-opt size=4k rnd-
dockerhub.huawei.com/official/ubuntu-arm64:latest
docker: Error response from daemon: symlink /proc/mounts
/var/lib/docker/overlay2/e6e12701db1a488636c881b44109a807e187b8db51a50015db34a1
31294fcf70-init/merged/etc/mtab: disk quota exceeded.
See 'docker run --help'.
```

   轻量级容器不报错，正常启动

```
[root@localhost ~]# isula run -itd --storage-opt size=4k rnd-
dockerhub.huawei.com/official/ubuntu-arm64:latest
636480b1fc2cf8ac895f46e77d86439fe2b359a1ff78486ae81c18d089bbd728
[root@localhost ~]# isula ps
STATUS  PID   IMAGE                                            COMMAND   EXIT CODE
RESTART COUNT STARTAT      FINISHAT RUNTIME ID          NAMES
running 17609 rnd-dockerhub.huawei.com/official/ubuntu-arm64:latest /bin/bash 0
0           2 seconds ago -        lcr    636480b1fc2c
636480b1fc2cf8ac895f46e77d86439fe2b359a1ff78486ae81c18d089bbd728
```

   在启动容器的过程中，如果需要在容器的 rootfs 路径下创建文件，若镜像本身占用的大小超过 4k，且此时的 quota 设置为 4k，则创建文件必定失败。

   docker 在启动容器的过程中，会比 isulad 创建更多的挂载点，用于挂载 host 上的某些路径到容器中，如/proc/mounts, /dev/shm 等，如果镜像内本身不存在这些文件，则会创建，根据上述原因该操作会导致文件创建失败，因而容器启动失败。

   轻量级容器在启动容器过程中，使用默认配置时，挂载点较少，如/proc，或/sys 等路径不存在时，才会创建。用例中的镜像 rnd-dockerhub.huawei.com/official/ubuntu-arm64:latest 本身含有/proc， /sys 等，因此整

个启动容器的过程中，都不会有新文件或路径生成，故轻量级容器启动过程不会报错。为验证这一过程，当把镜像替换为 rnd-dockerhub.huawei.com/official/busybox-aarch64:latest 时，由于该镜像内无/proc 存在，轻量级容器启动一样会报错。

```
[root@localhost ~]# isula run -itd --storage-opt size=4k rnd-
dockerhub.huawei.com/official/busybox-aarch64:latest
8e893ab483310350b8caa3b29eca7cd3c94eae55b48bfc82b350b30b17a0aaf4
Error response from daemon: Start container error: runtime error:
8e893ab483310350b8caa3b29eca7cd3c94eae55b48bfc82b350b30b17a0aaf4:tools/lxc star
t.c:main:404 starting container process caused "Failed to setup lxc,
please check the config file."
```

5. 其他说明。

   使用限额功能的 isulad 切换数据盘时，需要保证被切换的数据盘使用`prjquota`选项挂载，且/var/lib/isulad/storage/overlay2 目录的挂载方式与/var/lib/isulad 相同。

说明

切换数据盘时需要保证/var/lib/isulad/storage/overlay2 的挂载点被卸载。

1.3.3.6 限制容器内文件句柄数

描述

可以通过参数限制容器中可以打开的文件句柄数。

用法

isula create/run 时使用--files-limit 来限制容器中可以打开的文件句柄数。

参数

create/run 时指定--files-limit 参数。

参数项	参数说明	取值范围	是否必选
--files-limit	限制容器中可以打开的文件句柄数。	64 位整数(int64)。可以为 0、负，但不能超过 2 的 63 次方减 1，0、负表示不做限制（max）。 由于创建容器的过程中会临时打开一些句柄，所以此值不能设置的太小，不然容器可能不受 files limit 的限制（如果设置的数小于当前已经打开的句柄数，会导致	否

参数项	参数说明	取值范围	是否必选
		cgroup 文件写不进去），建议大于30。	

示例

在运行容器时，直接加上--files-limit n 即可，如：

```
isula run -ti --files-limit 1024 busybox bash
```

约束

1. 使用--files-limit 参数传入一个很小的值，如 1，可能导致容器启动失败。

```
[root@localhost ~]# isula run -itd --files-limit 1 rnd-
dockerhub.huawei.com/official/busybox-aarch64
004858d9f9ef429b624f3d20f8ba12acfbc8a15bb121c4036de4e5745932eff4
Error response from daemon: Start container error: Container is not
running:004858d9f9ef429b624f3d20f8ba12acfbc8a15bb121c4036de4e5745932eff4
```

而 docker 会启动成功，其 files.limit cgroup 值为 max。

```
[root@localhost ~]# docker run -itd --files-limit 1 rnd-
dockerhub.huawei.com/official/busybox-aarch64
ef9694bf4d8e803a1c7de5c17f5d829db409e41a530a245edc2e5367708dbbab
[root@localhost ~]# docker exec -it ef96 cat /sys/fs/cgroup/files/files.limit
max
```

根因是 lxc 和 runc 启动过程的原理不一样，lxc 创建 cgroup 子组后先设置 files.limit 值，再将容器进程的 PID 写入该子组的 cgroup.procs 文件，此时该进程已经打开超过 1 个句柄，因而写入报错导致启动失败。runc 创建 cgroup 子组后先将容器进程的 PID 写入该子组的 cgroup.procs 文件，再设置 files.limit 值，此时由于该子组内的进程已经打开超过 1 个句柄，因而写入 files.limit 不会生效，内核也不会报错，容器启动成功。

1.3.3.7 限制容器内可以创建的进程/线程数

描述

可以通过参数限制容器中能够创建的进程/线程数。

用法

在容器 create/run 时，使用参数--pids-limit 来限制容器中可以创建的进程/线程数。

参数

create/run 时指定--pids-limit 参数。

参数项	参数说明	取值范围	是否必选

参数项	参数说明	取值范围	是否必选
--pids-limit	限制容器中可以打开的文件句柄数。	64 位整数(int64)。可以为 0、负，但不能超过 2 的 63 次方减 1，0、负表示不做限制（max）。	否

示例

在运行容器时，直接加上--pids-limit n 即可，如：

```
isula run -ti --pids-limit 1024 busybox bash
```

约束

由于创建容器的过程中会临时创建一些进程，所以此值不能设置的太小，不然容器可能起不来，建议大于 10。

1.3.3.8 配置容器内的 ulimit 值

描述

可以通过参数控制执行程序的资源。

用法

在容器 create/run 时配置--ulimit 参数，或通过 daemon 端配置，控制容器中执行程序的资源。

参数

通过两种方法配置 ulimit

1. isula create/run 时使用--ulimit <type>=<soft>[:<hard>]来控制 shell 执行程序的资源。

参数项	参数说明	取值范围	是否必选
--ulimit	限制 shell 执行程序的资源	soft/hard 是 64 位整数(int64)。soft 取值 <= hard 取值，如果仅仅指定了 soft 的取值，则 hard=soft。对于某些类型的资源并不支持负数，详见下表	否

2. 通过 daemon 端参数或配置文件

   详见 1.2.3.1 部署方式中的--default-ulimits 相关选项。

   --ulimit 可以对以下类型的资源进行限制。

类型	说明	取值范围
core	limits the core file size (KB)	64 位整数(INT64)，无单位。可以为 0、负、其中-1 表示 UNLIMITED，即不做限制，其余的负数会被强制转换为一个大的正整数。
cpu	max CPU time (MIN)	
data	max data size (KB)	
fsize	maximum filesize (KB)	
locks	max number of file locks the user can hold	
memlock	max locked-in-memory address space (KB)	
msgqueue	max memory used by POSIX message queues (bytes)	
nice	nice priority	
nproc	max number of processes	
rss	max resident set size (KB)	
rtprio	max realtime priority	
rttime	realtime timeout	
sigpending	max number of pending signals	
stack	max stack size (KB)	
nofile	max number of open file descriptors	64 位整数(int64)，无单位。不可以为负，负数被强转为大数，设置时会出现 Operation not permitted

示例

在容器的创建或者运行时，加上--ulimit <type>=<soft>[:<hard>]即可，如：

```
isula create/run -tid --ulimit nofile=1024:2048 busybox sh
```

约束

不能在 daemon.json 和/etc/sysconfig/iSulad 文件（或 isulad 命令行）中同时配置 ulimit 限制，否则 isulad 启动会报错。

1.3.4 特权容器

1.3.4.1 场景说明

iSulad 默认启动的是普通容器，普通容器适合启动普通进程，其权限非常受限，仅具备/etc/default/isulad/config.json 中 capabilities 所定义的默认权限。当需要特权操作时（比如操作/sys 下的设备），需要特权容器完成这些操作，使用该特性，容器内的 root 将拥有宿主机的 root 权限， 否则，容器内的 root 在只是宿主机的普通用户权限。

1.3.4.2 使用限制

特权容器为容器提供了所有功能，还解除了设备 cgroup 控制器强制执行的所有限制，具备以下特性：

- Secomp 不 block 任何系统调用
- /sys、/proc 路径可写
- 容器内能访问主机上所有设备
- 系统的权能将全部打开

普通容器默认权能为：

Capability Key	Capability Description
SETPCAP	修改进程权能
MKNOD	允许使用 mknod()系统调用创建特殊文件
AUDIT_WRITE	向内核审计日志写记录
CHOWN	对文件的 UIDs 和 GIDs 做任意的修改(参考 chown(2))
NET_RAW	使用 RAW 和 PACKET sockets；为透明代理绑定任何地址
DAC_OVERRIDE	忽略文件的 DAC 访问限制
FOWNER	忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
FSETID	允许设置文件的 setuid 位
KILL	允许对不属于自己的进程发送信号
SETGID	允许改变进程的组 ID
SETUID	允许改变进程的用户 ID
NET_BIND_SERVICE	允许绑定到小于 1024 的端口
SYS_CHROOT	允许使用 chroot()系统调用
SETFCAP	允许向其他进程转移能力以及删除其他进程的能力

当容器为特权模式时，将添加以下权能

Capability Key	Capability Description
SYS_MODULE	加载和卸载内核模块
SYS_RAWIO	允许直接访问/devport,/dev/mem,/dev/kmem 及原始块设备
SYS_PACCT	允许执行进程的 BSD 式审计
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	允许提升优先级及设置其他进程的优先级
SYS_RESOURCE	忽略资源限制
SYS_TIME	允许改变系统时钟
SYS_TTY_CONFIG	允许配置 TTY 设备
AUDIT_CONTROL	启用和禁用内核审计；修改审计过滤器规则；提取审计状态和过滤规则
MAC_ADMIN	覆盖强制访问控制（Mandatory Access Control (MAC)），为 Smack Linux 安全模块(Linux Security Module (LSM)）而实现
MAC_OVERRIDE	允许 MAC 配置或状态改变。为 Smack LSM 而实现
NET_ADMIN	允许执行网络管理任务
SYSLOG	执行特权 syslog(2) 操作
DAC_READ_SEARCH	忽略文件读及目录搜索的 DAC 访问限制
LINUX_IMMUTABLE	允许修改文件的 IMMUTABLE 和 APPEND 属性标志
NET_BROADCAST	允许网络广播和多播访问
IPC_LOCK	允许锁定共享内存片段
IPC_OWNER	忽略 IPC 所有权检查
SYS_PTRACE	允许跟踪任何进程
SYS_BOOT	允许重新启动系统
LEASE	允许修改文件锁的 FL_LEASE 标志
WAKE_ALARM	触发将唤醒系统的功能，如设置 CLOCK_REALTIME_ALARM 和

Capability Key	Capability Description
	CLOCK_BOOTTIME_ALARM 定时器
BLOCK_SUSPEND	可以阻塞系统挂起的特性

1.3.4.3 使用指导

iSulad 使用--privileged 给容器添加特权模式，在非必要情况下，不要给容器添加特权，遵循最小特权原则，减少存在的安全风险。

```
isula run --rm -it --privileged busybox
```

1.3.5 CRI 接口

1.3.5.1 描述

CRI API 接口是由 kubernetes 推出的容器运行时接口，CRI 定义了容器和镜像的服务接口。ISulad 使用 CRI 接口，实现和 kubernetes 的对接。

因为容器运行时与镜像的生命周期是彼此隔离的，因此需要定义两个服务。该接口使用 Protocol Buffer 定义，基于 gRPC。

当前实现 CRI 版本为 v1alpha1 版本，官方 API 描述文件如下：

https://github.com/kubernetes/kubernetes/blob/release-1.14/pkg/kubelet/apis/cri/runtime/v1alpha2/api.proto，

ISulad 使用的为 pass 使用的 1.14 版本 API 描述文件，与官方 API 略有出入，以本文档描述的接口为准。

说明

CRI 接口 websocket 流式服务，服务端侦听地址为 127.0.0.1，端口为 10350，端口可通过命令行--websocket-server-listening-port 参数接口或者 daemon.json 配置文件进行配置。

1.3.5.2 接口

各接口中可能用到的参数清单如下，部分参数暂不支持配置，已在配置中标出。

接口参数列表

- **DNSConfig**

  配置 sandbox 的 DNS 服务器和搜索域

参数成员	描述
repeated string servers	集群的 DNS 服务器列表
repeated string searches	集群的 DNS 搜索域列表
repeated string options	DNS 可选项列表，参考 https://linux.die.net/man/5/resolv.conf

- **Protocol**

  协议的 enum 值列表

参数成员	描述
TCP = 0	TCP 协议
UDP = 1	UDP 协议

- **PortMapping**

  指定 sandbox 的端口映射配置

参数成员	描述
Protocol protocol	端口映射使用的协议
int32 container_port	容器内的端口号
int32 host_port	主机上的端口号
string host_ip	主机 IP 地址

- **MountPropagation**

  挂载传播属性的 enum 列表

参数成员	描述
PROPAGATION_PRIVATE = 0	无挂载传播属性，即 linux 中的 private
PROPAGATION_HOST_TO_CONTAINER = 1	挂载属性能从 host 传播到容器中，即 linux 中的 rslave
PROPAGATION_BIDIRECTIONAL = 2	挂载属性能在 host 和容器中双向传播，即 linux 中的 rshared

- **Mount**

  Mount 指定 host 上的一个挂载卷挂载到容器中（只支持文件和文件夹)

参数成员	描述
string container_path	容器中的路径
string host_path	主机上的路径
bool readonly	是否配置在容器中是只读的， 默认值： false
bool selinux_relabel	是否设置 SELinux 标签（不支持配置）
MountPropagation propagation	挂载传播属性配置（取值 **0/1/2**，分别对应

参数成员	描述
	private/rslave/rshared 传播属性） **默认值：0**

- **NamespaceOption**

参数成员	描述
bool host_network	是否使用 host 的网络命名空间
bool host_pid	是否使用 host 的 PID 命名空间
bool host_ipc	是否使用 host 的 IPC 命名空间

- **Capability**

  包含待添加与待删除的权能信息

参数成员	描述
repeated string add_capabilities	待新增的权能
repeated string drop_capabilities	待删除的权能

- **Int64Value**

  int64 类型的封装

参数成员	描述
int64 value	实际的 int64 值

- **UInt64Value**

  uint64 类型的封装

参数成员	描述
uint64 value	实际的 uint64 值

- **LinuxSandboxSecurityContext**

  配置 sandbox 的 linux 安全选项。

  注意，这些安全选项不会应用到 sandbox 中的容器中，也可能不适用于没有任何 running 进程的 sandbox

参数成员	描述
NamespaceOption namespace_options	配置 sandbox 的命名空间选项

参数成员	描述
SELinuxOption selinux_options	配置 SELinux 选项（不支持）
Int64Value run_as_user	配置 sandbox 中进程的 uid
bool readonly_rootfs	配置 sandbox 的根文件系统是否只读
repeated int64 supplemental_groups	配置除主 GID 之外的 sandbox 的 1 号进程用户组信息
bool privileged	配置 sandbox 是否为特权容器

string seccomp_profile_path	seccomp 配置文件路径，有效值为：
	// unconfined：不配置 seccomp
	// localhost/<配置文件的全路径>： 安装在系统上的配置文件路径
	// <配置文件的全路径>： 配置文件全路径
	// 默认不配置，即 unconfined。

- **LinuxPodSandboxConfig**

  设定和 Linux 主机及容器相关的一些配置

参数成员	描述
string cgroup_parent	sandbox 的 cgroup 父路径，runtime 可根据实际情况使用 cgroupfs 或 systemd 的语法。（不支持配置）
LinuxSandboxSecurityContext security_context	sandbox 的安全属性
map<string, string> sysctls	sandbox 的 linux sysctls 配置

- **PodSandboxMetadata**

  Sandbox 元数据包含构建 sandbox 名称的所有信息，鼓励容器运行时在用户界面中公开这些元数据以获得更好的用户体验，例如，运行时可以根据元数据生成 sandbox 的唯一命名。

参数成员	描述
string name	sandbox 的名称
string uid	sandbox 的 UID
string namespace	sandbox 的命名空间
uint32 attempt	尝试创建 sandbox 的次数，默认为 0

- **PodSandboxConfig**

  包含创建 sandbox 的所有必选和可选配置信息

参数成员	描述
PodSandboxMetadata metadata	sandbox 的元数据，这项信息唯一标识一个 sandbox，runtime 必须利用此信息确保操作正确，runtime 也可以根据此信息来改善用户体验，例如构建可读的 sandbox 名称。
string hostname	sandbox 的 hostname
string log_directory	配置 sandbox 内的容器的日志文件所存储的文件夹
DNSConfig dns_config	sandbox 的 DNS 配置
repeated PortMapping port_mappings	sandbox 的端口映射
map<string, string> labels	可用于标识单个或一系列 sandbox 的键值对
map<string, string> annotations	存储任意信息的键值对，这些值是不可更改的，且能够利用 PodSandboxStatus 接口查询
LinuxPodSandboxConfig linux	与 linux 主机相关的可选项

- **PodSandboxNetworkStatus**

  描述 sandbox 的网络状态

参数成员	描述
string ip	sandbox 的 ip 地址
string name	sandbox 内的网络接口名
string network	附加网络的名称

- **Namespace**

  命名空间选项

参数成员	描述
NamespaceOption options	Linux 命名空间选项

- **LinuxPodSandboxStatus**

  描述 Linux sandbox 的状态

参数成员	描述

参数成员	描述
Namespace **namespaces**	sandbox 命名空间

- **PodSandboxState**

  sandbox 状态值的 enum 数据

参数成员	描述
SANDBOX_READY = 0	sandbox 处于 ready 状态
SANDBOX_NOTREADY = 1	sandbox 处于非 ready 状态

- **PodSandboxStatus**

  描述 Podsandbox 的状态信息

参数成员	描述
string id	sandbox 的 ID
PodSandboxMetadata metadata	sandbox 的元数据
PodSandboxState state	sandbox 的状态值
int64 created_at	sandbox 的创建时间戳，单位纳秒
repeated PodSandboxNetworkStatus networks	sandbox 的多平面网络状态
LinuxPodSandboxStatus linux	Linux 规范的 sandbox 状态
map<string, string> labels	可用于标识单个或一系列 sandbox 的键值对
map<string, string> annotations	存储任意信息的键值对，这些值是不可被 runtime 更改的

- **PodSandboxStateValue**

  对 PodSandboxState 的封装

参数成员	描述
PodSandboxState state	sandbox 的状态值

- **PodSandboxFilter**

  用于列出 sandbox 时添加过滤条件，多个条件取交集显示

参数成员	描述

参数成员	描述
string id	sandbox 的 ID
PodSandboxStateValue state	sandbox 的状态
map<string, string> label_selector	sandbox 的 labels，label 只支持完全匹配，不支持正则匹配

- **PodSandbox**

  包含最小化描述一个 sandbox 的数据

参数成员	描述
string id	sandbox 的 ID
PodSandboxMetadata metadata	sandbox 的元数据
PodSandboxState state	sandbox 的状态值
int64 created_at	sandbox 的创建时间戳，单位纳秒
map<string, string> labels	可用于标识单个或一系列 sandbox 的键值对
map<string, string> annotations	存储任意信息的键值对，这些值是不可被 runtime 更改的

- **KeyValue**

  键值对的封装

参数成员	描述
string key	键
string value	值

- **SELinuxOption**

  应用于容器的 SELinux 标签

参数成员	描述
string user	用户
string role	角色
string type	类型
string level	级别

- **ContainerMetadata**

  Container 元数据包含构建 container 名称的所有信息，鼓励容器运行时在用户界面中公开这些元数据以获得更好的用户体验，例如，运行时可以根据元数据生成 container 的唯一命名。

参数成员	描述
string name	container 的名称
uint32 attempt	尝试创建 container 的次数，默认为 0

- **ContainerState**

  容器状态值的 enum 列表

参数成员	描述
CONTAINER_CREATED = 0	container 创建完成
CONTAINER_RUNNING = 1	container 处于运行状态
CONTAINER_EXITED = 2	container 处于退出状态
CONTAINER_UNKNOWN = 3	未知的容器状态

- **ContainerStateValue**

  封装 ContainerState 的数据结构

参数成员	描述
ContainerState **state**	容器状态值

- **ContainerFilter**

  用于列出 container 时添加过滤条件，多个条件取交集显示

参数成员	描述
string id	container 的 ID
PodSandboxStateValue state	container 的状态
string pod_sandbox_id	sandbox 的 ID
map<string, string> label_selector	container 的 labels，label 只支持完全匹配，不支持正则匹配

- **LinuxContainerSecurityContext**

  指定应用于容器的安全配置

参数成员	描述
Capability capabilities	新增或去除的权能
bool privileged	指定容器是否未特权模式， **默认值：false**
NamespaceOption namespace_options	指定容器的 namespace 选项
SELinuxOption selinux_options	SELinux context(可选配置项) **暂不支持**
Int64Value run_as_user	运行容器进程的 UID。 一次只能指定 run_as_user 与 run_as_username 其中之一，run_as_username 优先生效
string run_as_username	运行容器进程的用户名。 如果指定，用户必须存在于容器映像中（即在映像内的/etc/passwd 中），并由运行时在那里解析；否则，运行时必须出错
bool readonly_rootfs	设置容器中根文件系统是否为只读 **默认值由 config.json 配置**
repeated int64 supplemental_groups	容器运行的除主 GID 外首进程组的列表
string apparmor_profile	容器的 AppArmor 配置文件 **暂不支持**
string seccomp_profile_path	容器的 seccomp 配置文件路径
bool no_new_privs	是否在容器上设置 no_new_privs 的标志

- **LinuxContainerResources**

  指定 Linux 容器资源的特定配置

参数成员	描述
int64 cpu_period	CPU CFS（完全公平调度程序）周期。 **默认值：0**
int64 cpu_quota	CPU CFS（完全公平调度程序）配额。 **默认值：0**
int64 cpu_shares	所占 CPU 份额（相对于其他容器的相对权重）。 **默认值：0**
int64 memory_limit_in_bytes	内存限制（字节）。 **默认值：0**
int64 oom_score_adj	OOMScoreAdj 用于调整 oom-killer。 **默认值：0**
string cpuset_cpus	指定容器使用的 CPU 核心。 **默认值：“”**
string cpuset_mems	指定容器使用的内存节点。 **默认值：“”**

- **Image**

  Image 信息描述一个镜像的基本数据。

参数成员	描述
string id	镜像 ID
repeated string repo_tags	镜像 tag 名称 repo_tags
repeated string repo_digests	镜像 digest 信息
uint64 size	镜像大小
Int64Value uid	镜像默认用户 UID
string username	镜像默认用户名称

- **ImageSpec**

  表示镜像的内部数据结构，当前，ImageSpec 只封装容器镜像名称

参数成员	描述
string image	容器镜像名

- **StorageIdentifier**

  唯一定义 storage 的标识

参数成员	描述
string uuid	设备的 UUID

- **FilesystemUsage**

参数成员	描述
int64 timestamp	收集文件系统信息时的时间戳
StorageIdentifier storage_id	存储镜像的文件系统 UUID
UInt64Value used_bytes	存储镜像元数据的大小
UInt64Value inodes_used	存储镜像元数据的 inodes 个数

- **AuthConfig**

参数成员	描述
string username	下载镜像使用的用户名

string password	下载镜像使用的密码
string auth	下载镜像时使用的认证信息，base64 编码
string server_address	下载镜像的服务器地址，暂不支持
string identity_token	用于与镜像仓库鉴权的令牌信息，暂不支持
string registry_token	用于与镜像仓库交互的令牌信息，暂不支持

- **Container**

  用于描述容器信息，例如 ID，状态等。

参数成员	描述
string id	container 的 ID
string pod_sandbox_id	该容器所属的 sandbox 的 ID
ContainerMetadata metadata	container 的元数据
ImageSpec image	镜像规格
string image_ref	代表容器使用的镜像，对大多数 runtime 来产，这是一个 image ID 值
ContainerState state	container 的状态
int64 created_at	container 的创建时间戳，单位为纳秒
map<string, string> labels	可用于标识单个或一系列 container 的键值对
map<string, string> annotations	存储任意信息的键值对，这些值是不可被 runtime 更改的

- **ContainerStatus**

  用于描述容器状态信息

参数成员	描述
string id	container 的 ID
ContainerMetadata metadata	container 的元数据
ContainerState state	container 的状态
int64 created_at	container 的创建时间戳，单位为纳秒
int64 started_at	container 启动时的时间戳，单位为纳秒
int64 finished_at	container 退出时的时间戳，单位为纳秒
int32 exit_code	容器退出码

参数成员	描述
ImageSpec image	镜像规格
string image_ref	代表容器使用的镜像，对大多数 runtime 来产，这是一个 image ID 值
string reason	简要描述为什么容器处于当前状态
string message	易于人工阅读的信息，用于表述容器处于当前状态的原因
map<string, string> labels	可用于标识单个或一系列 container 的键值对
map<string, string> annotations	存储任意信息的键值对，这些值是不可被 runtime 更改的
repeated Mount mounts	容器的挂载点信息
string log_path	容器日志文件路径，该文件位于 PodSandboxConfig 中配置的 log_directory 文件夹下

- **ContainerStatsFilter**

  用于列出 container stats 时添加过滤条件，多个条件取交集显示

参数成员	描述
string id	container 的 ID
string pod_sandbox_id	sandbox 的 ID
map<string, string> label_selector	container 的 labels，label 只支持完全匹配，不支持正则匹配

- **ContainerStats**

  用于列出 container stats 时添加过滤条件，多个条件取交集显示

参数成员	描述
ContainerAttributes attributes	容器的信息
CpuUsage cpu	CPU 使用情况
MemoryUsage memory	内存使用情况
FilesystemUsage writable_layer	可写层使用情况

- **ContainerAttributes**

  列出 container 的基本信息

参数成员	描述
string id	容器的 ID
ContainerMetadata metadata	容器的 metadata
map<string,string> labels	可用于标识单个或一系列 container 的键值对
map<string,string> annotations	存储任意信息的键值对，这些值是不可被 runtime 更改的

- **CpuUsage**

  列出 container 的 CPU 使用信息

参数成员	描述
int64 timestamp	时间戳
UInt64Value usage_core_nano_seconds	CPU 的使用值，单位/纳秒

- **MemoryUsage**

  列出 container 的内存使用信息

参数成员	描述
int64 timestamp	时间戳
UInt64Value working_set_bytes	内存的使用值

- **FilesystemUsage**

  列出 container 的读写层信息

参数成员	描述
int64 timestamp	时间戳
StorageIdentifier storage_id	可写层目录
UInt64Value used_bytes	镜像在可写层的占用字节
UInt64Value inodes_used	镜像在可写层的占用 inode 数

- **Device**

  指定待挂载至容器的主机卷

参数成员	描述

string container_path	容器内的挂载路径
string host_path	主机上的挂载路径
string permissions	设备的 Cgroup 权限，（r 允许容器从指定的设备读取; w 允许容器从指定的设备写入; m 允许容器创建尚不存在的设备文件)

- **LinuxContainerConfig**

  包含特定于 Linux 平台的配置

参数成员	**描述**
LinuxContainerResources resources	容器的资源规范
LinuxContainerSecurityContext security_context	容器的 Linux 容器安全配置

- **ContainerConfig**

  包含用于创建容器的所有必需和可选字段

参数成员	**描述**
ContainerMetadata metadata	容器的元数据。 此信息将唯一标识容器，运行时应利用此信息来确保正确操作。 运行时也可以使用此信息来提升 UX（用户体检设计），例如通过构造可读名称。**(必选)**
ImageSpec image	容器使用的镜像 **(必选)**
repeated string command	待执行的命令 **默认值： "/bin/sh"**
repeated string args	待执行命令的参数
string working_dir	命令执行的当前工作路径
repeated KeyValue envs	在容器中配置的环境变量
repeated Mount mounts	待在容器中挂载的挂载点信息
repeated Device devices	待在容器中映射的设备信息
map<string, string> labels	可用于索引和选择单个资源的键值对。
map<string, string> annotations	可用于存储和检索任意元数据的非结构化键值映射。
string log_path	相对于 PodSandboxConfig.LogDirectory 的路径，用于存储容器主机上的日志（STDOUT 和 STDERR）。
bool stdin	是否打开容器的 stdin

bool stdin_once	当某次连接 stdin 的数据流断开时，是否立即断开其他与 stdin 连接的数据流（**暂不支持**）
bool tty	是否使用伪终端连接容器的 stdio
LinuxContainerConfig linux	linux 系统上容器的特定配置信息

- **NetworkConfig**

  Runtime 的网络配置

参数成员	描述
string pod_cidr	Pod IP 地址使用的 CIDR

- **RuntimeConfig**

  Runtime 的网络配置

参数成员	描述
NetworkConfig network_config	Runtime 的网络配置

- **RuntimeCondition**

  描述 runtime 的状态信息

参数成员	描述
string type	Runtime 状态的类型
bool status	Runtime 状态
string reason	简要描述 runtime 状态变化的原因
string message	具备可阅读性的信息表明 runtime 状态变化的原因

- **RuntimeStatus**

  Runtime 的状态

参数成员	描述
repeated RuntimeCondition conditions	描述当前 runtime 状态的列表

1.3.5.2.1 Runtime 服务

Runtime 服务中包含操作 pod 和容器的接口，以及查询 runtime 自身配置和状态信息的接口。

1.3.5.2.1.1 RunPodSandbox

接口原型

```
rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
```

接口描述

创建和启动一个 pod sandbox，若运行成功，sandbox 处于 ready 状态。

注意事项

1. 启动 sandbox 的默认镜像为 rnd-dockerhub.huawei.com/library/pause-${machine}:3.0， 其中${machine}为架构，在 x86_64 上，machine 的值为 amd64，在 arm64 上，machine 的值为 aarch64，当前 rnd-dockerhub 仓库上只有 amd64 和 aarch64 镜像可供下载，若机器上无此镜像，请确保机器能从 rnd-dockerhub 下载，若要使用其它镜像，请参考 iSulad 部署配置中的 pod-sandbox-image 指定镜像。
2. 由于容器命名以 PodSandboxMetadata 中的字段为来源，且以下划线"_"为分割字符，因此限制 metadata 中的数据不能包含下划线，否则会出现 sandbox 运行成功，但无法使用 ListPodSandbox 接口查询的现象。

参数

参数成员	描述
PodSandboxConfig config	sandbox 的配置
string runtime_handler	指定创建 sandbox 的 runtime 运行时，当前支持 lcr、kata-runtime 运行时类型。

返回值

返回值	描述
string pod_sandbox_id	成功，返回 response 数据

1.3.5.2.1.2 StopPodSandbox

接口原型

```
rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse) {}
```

接口描述

停止 pod sandbox，停止 sandbox 容器，回收分配给 sandbox 的网络资源（比如 IP 地址）。如果有任何 running 的容器属于该 sandbox，则必须被强制停止。

参数

参数成员	描述
string pod_sandbox_id	sandbox 的 id

返回值

返回值	描述
无	无

1.3.5.2.1.3 RemovePodSandbox

接口原型

```
rpc RemovePodSandbox(RemovePodSandboxRequest) returns (RemovePodSandboxResponse) {}
```

接口描述

删除 sandbox，如果有任何 running 的容器属于该 sandbox，则必须被强制停止和删除，如果 sandbox 已经被删除，不能返回错误。

注意事项

1. 删除 sandbox 时，不会删除 sandbox 的网络资源，在删除 pod 前必须先调用 StopPodSandbox 才能清理网络资源，调用者应当保证在删除 sandbox 之前至少调用一次 StopPodSandbox。

参数

参数成员	描述
string pod_sandbox_id	sandbox 的 id

返回值

返回值	描述
无	无

1.3.5.2.1.4 PodSandboxStatus

接口原型

```
rpc PodSandboxStatus(PodSandboxStatusRequest) returns (PodSandboxStatusResponse) {}
```

接口描述

查询 sandbox 的状态，如果 sandbox 不存在，返回错误。

参数

参数成员	描述
string pod_sandbox_id	sandbox 的 id
bool verbose	标识是否显示 sandbox 的一些额外信息。（暂不支持配置）

返回值

返回值	描述
PodSandboxStatus status	sandbox 的状态信息
map<string, string> info	sandbox 的额外信息，key 是任意 string，value 是 json 格式的字符串，这些信息可以是任意调试内容。当 verbose 为 true 时 info 不能为空。（暂不支持配置）

1.3.5.2.1.5 ListPodSandbox

接口原型

```
rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse) {}
```

接口描述

返回 sandbox 信息的列表，支持条件过滤。

参数

参数成员	描述
PodSandboxFilter filter	条件过滤参数

返回值

返回值	描述
repeated PodSandbox items	sandbox 信息的列表

1.3.5.2.1.6 CreateContainer

```
grpc::Status CreateContainer(grpc::ServerContext *context, const
runtime::CreateContainerRequest *request, runtime::CreateContainerResponse *reply)
{}
```

接口描述

在 PodSandbox 内创建一个容器。

注意事项

- 请求 CreateContainerRequest 中的 sandbox_config 与传递给 RunPodSandboxRequest 以创建 PodSandbox 的配置相同。 它再次传递，只是为了方便参考。PodSandboxConfig 是不可变的，在 pod 的整个生命周期内保持不变。
- 由于容器命名以 ContainerMetadata 中的字段为来源，且以下划线"_"为分割字符，因此限制 metadata 中的数据不能包含下划线，否则会出现 sandbox 运行成功，但无法使用 ListContainers 接口查询的现象。
- CreateContainerRequest 中无 runtime_handler 字段，创建 container 时的 runtime 类型和其对应的 sandbox 的 runtime 相同。

参数

参数成员	描述
string pod_sandbox_id	待在其中创建容器的 PodSandbox 的 ID。
ContainerConfig config	容器的配置信息
PodSandboxConfig sandbox_config	PodSandbox 的配置信息

补充

可用于存储和检索任意元数据的非结构化键值映射。有一些字段由于 cri 接口没有提供特定的参数，可通过该字段将参数传入

- 自定义

自定义 key:value	描述↲

cgroup.pids.max:int64_t	用于限制容器内的进/线程数（set -1 for unlimited）

返回值

返回值	描述
string container_id	创建完成的容器 ID

1.3.5.2.1.7 StartContainer

接口原型

```
rpc StartContainer(StartContainerRequest) returns (StartContainerResponse) {}
```

接口描述

启动一个容器。

参数

参数成员	描述
string container_id	容器 id

返回值

返回值	描述
无	无

1.3.5.2.1.8 StopContainer

接口原型

```
rpc StopContainer(StopContainerRequest) returns (StopContainerResponse) {}
```

接口描述

停止一个 running 的容器，支持配置优雅停止时间 timeout，如果容器已经停止，不能返回错误。

参数

参数成员	描述
string container_id	容器 id
int64 timeout	强制停止容器前的等待时间，默认值为 0，即强制停止容器。

返回值

无

1.3.5.2.1.9 RemoveContainer

接口原型

```
rpc RemoveContainer(RemoveContainerRequest) returns (RemoveContainerResponse) {}
```

接口描述

删除一个容器，如果容器正在运行，必须强制停止，如果容器已经被删除，不能返回错误。

参数

参数成员	描述
string container_id	容器 id

返回值

无

1.3.5.2.1.10 ListContainers

接口原型

```
rpc ListContainers(ListContainersRequest) returns (ListContainersResponse) {}
```

接口描述

返回 container 信息的列表，支持条件过滤。

参数

参数成员	描述

ContainerFilter filter	条件过滤参数

返回值

返回值	描述
repeated Container containers	容器信息的列表

1.3.5.2.1.11 ContainerStatus

接口原型

```
rpc ContainerStatus(ContainerStatusRequest) returns (ContainerStatusResponse) {}
```

接口描述

返回容器状态信息，如果容器不存在，则返回错误。

参数

参数成员	描述
string container_id	容器 id
bool verbose	标识是否显示 sandbox 的一些额外信息。（暂不支持配置）

返回值

返回值	描述
ContainerStatus status	容器的状态信息
map<string, string> info	sandbox 的额外信息，key 是任意 string，value 是 json 格式的字符串，这些信息可以是任意调试内容。当 verbose 为 true 时 info 不能为空。（暂不支持配置）

1.3.5.2.1.12 UpdateContainerResources

接口原型

```
rpc UpdateContainerResources(UpdateContainerResourcesRequest) returns
(UpdateContainerResourcesResponse) {}
```

接口描述

该接口用于更新容器资源配置。

注意事项

- 该接口仅用于更新容器的资源配置，不能用于更新 Pod 的资源配置。
- 当前不支持更新容器 oom_score_adj 配置。

参数

参数成员	描述
string container_id	容器 id
LinuxContainerResources linux	linux 资源配置信息

返回值

无

1.3.5.2.1.13 ExecSync

接口原型

```
rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
```

接口描述

以同步的方式在容器中执行命令，采用的 gRPC 通讯方式。

注意事项

执行执行一条单独的命令，不能打开终端与容器交互。

参数

参数成员	描述
string container_id	容器 ID
repeated string cmd	待执行命令
int64 timeout	停止命令的超时时间（秒）。 默认值：0（无超时限制）。 **暂不支持**

返回值

返回值	描述
bytes stdout	捕获命令标准输出
bytes stderr	捕获命令标准错误输出
int32 exit_code	退出代码命令完成。 默认值：0（成功）。

1.3.5.2.1.14 Exec

接口原型

```
rpc Exec(ExecRequest) returns (ExecResponse) {}
```

接口描述

在容器中执行命令，采用的 gRPC 通讯方式从 CRI 服务端获取 url，再通过获得的 url 与 websocket 服务端建立长连接，实现与容器的交互。

注意事项

执行执行一条单独的命令，也能打开终端与容器交互。stdin/stdout/stderr 之一必须是真的。如果 tty 为真，stderr 必须是假的。 不支持多路复用， 在这种情况下, stdout 和 stderr 的输出将合并为单流。

参数

参数成员	描述
string container_id	容器 ID
repeated string cmd	待执行的命令
bool tty	是否在 TTY 中执行命令
bool stdin	是否流式标准输入
bool stdout	是否流式标准输出
bool stderr	是否流式输出标准错误

返回值

返回值	描述
string url	exec 流服务器的完全限定 URL

1.3.5.2.1.15 Attach

接口原型

```
rpc Attach(AttachRequest) returns (AttachResponse) {}
```

接口描述

接管容器的 1 号进程，采用的 gRPC 通讯方式从 CRI 服务端获取 url，再通过获得的 url 与 websocket 服务端建立长连接，实现与容器的交互。仅支持 runtime 类型为 lcr 的容器。

参数

参数成员	描述
string container_id	容器 ID
bool tty	是否在 TTY 中执行命令
bool stdin	是否流式标准输入
bool stdout	是否流式标准输出
bool stderr	是否流式输出标准错误

返回值

返回值	描述
string url	attach 流服务器的完全限定 URL

1.3.5.2.1.16 ContainerStats

接口原型

```
rpc ContainerStats(ContainerStatsRequest) returns (ContainerStatsResponse) {}
```

接口描述

返回单个容器占用资源信息，仅支持 runtime 类型为 lcr 的容器。

参数

参数成员	描述
string container_id	容器 id

返回值

返回值	描述
ContainerStats stats	容器信息。注：disk 和 inodes 只支持 oci 格式镜像起的容器查询

1.3.5.2.1.17 ListContainerStats

接口原型

```
rpc ListContainerStats(ListContainerStatsRequest) returns
(ListContainerStatsResponse) {}
```

接口描述

返回多个容器占用资源信息，支持条件过滤

参数

参数成员	描述
ContainerStatsFilter filter	条件过滤参数

返回值

返回值	描述
repeated ContainerStats stats	容器信息的列表。注：disk 和 inodes 只支持 oci 格式镜像启动的容器查询

1.3.5.2.1.18 UpdateRuntimeConfig

接口原型

```
rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns
(UpdateRuntimeConfigResponse);
```

接口描述

提供标准的 CRI 接口，目的为了更新网络插件的 Pod CIDR，当前 CNI 网络插件无需更新 Pod CIDR，因此该接口只会记录访问日志。

注意事项

接口操作不会对系统管理信息修改，只是记录一条日志。

参数

参数成员	描述
RuntimeConfig runtime_config	包含 Runtime 要配置的信息

返回值

无

1.3.5.2.1.19 Status

接口原型

```
rpc Status(StatusRequest) returns (StatusResponse) {};
```

接口描述

获取 runtime 和 pod 的网络状态，在获取网络状态时，会触发网络配置的刷新。仅支持 runtime 类型为 lcr 的容器。

注意事项

如果网络配置刷新失败，不会影响原有配置；只有刷新成功时，才会覆盖原有配置。

参数

参数成员	描述
bool verbose	是否显示关于 Runtime 额外的信息（暂不支持）

返回值

返回值	描述
RuntimeStatus status	Runtime 的状态
map<string, string> info	Runtime 额外的信息，info 的 key 为任意值，value 为 json 格式，可包含任何 debug 信息；只有 Verbose 为 true 是才应该被赋值

1.3.5.2.2 Image 服务

提供了从镜像仓库拉取、查看、和移除镜像的 gRPC API。

1.3.5.2.2.1 ListImages

接口原型

```
rpc ListImages(ListImagesRequest) returns (ListImagesResponse) {}
```

接口描述

列出当前已存在的镜像信息。

注意事项

为统一接口，对于 embedded 格式镜像，可以通过 cri images 查询到。但是因 embedded 镜像不是标准 OCI 镜像，因此查询得到的结果有以下限制：

- 因 embedded 镜像无镜像 ID，显示的镜像 ID 为镜像的 config digest。
- 因 embedded 镜像本身无 digest 仅有 config 的 digest，且格式不符合 OCI 镜像规范，因此无法显示 digest。

参数

参数成员	描述
ImageSpec filter	筛选的镜像名称

返回值

返回值	描述
repeated Image images	镜像信息列表

1.3.5.2.2.2 ImageStatus

接口原型

```
rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResponse) {}
```

接口描述

查询指定镜像信息。

注意事项

1. 查询指定镜像信息，若镜像不存在，则返回 ImageStatusResponse，其中 Image 设置为 nil。
2. 为统一接口，对于 embedded 格式镜像，因不符合 OCI 格式镜像，缺少字段，无法通过本接口进行查询。

参数

参数成员	描述
ImageSpec image	镜像名称
bool verbose	查询额外信息，暂不支持，无额外信息返回

返回值

返回值	描述
Image image	镜像信息
map<string, string> info	镜像额外信息，暂不支持，无额外信息返回

1.3.5.2.2.3 PullImage

接口原型

```
rpc PullImage(PullImageRequest) returns (PullImageResponse) {}
```

接口描述

下载镜像。

注意事项

当前支持下载 public 镜像，使用用户名、密码、auth 信息下载私有镜像，不支持 authconfig 中的 server_address、identity_token、registry_token 字段。

参数

参数成员	描述
ImageSpec image	要下载的镜像名称
AuthConfig auth	下载私有镜像时的验证信息
PodSandboxConfig sandbox_config	在 Pod 上下文中下载镜像（暂不支持）

返回值

返回值	描述
string image_ref	返回已下载镜像信息

1.3.5.2.2.4 RemoveImage

接口原型

```
rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResponse) {}
```

接口描述

删除指定镜像。

注意事项

为统一接口，对于 embedded 格式镜像，因不符合 OCI 格式镜像，缺少字段，无法通过本接口使用 image id 进行删除。

参数

参数成员	描述
ImageSpec image	要删除的镜像名称或者 ID

返回值

无

1.3.5.2.2.5 ImageFsInfo

接口原型

```
rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResponse) {}
```

接口描述

查询存储镜像的文件系统信息。

注意事项

查询到的为镜像元数据下的文件系统信息。

参数

无

返回值

返回值	描述

repeated FilesystemUsage image_filesystems	镜像存储文件系统信息

1.3.5.3 约束

1. 如果创建 sandbox 时，PodSandboxConfig 参数中配置了 log_directory，则所有属于该 sandbox 的 container 在创建时必须在 ContainerConfig 中指定 log_path，否则可能导致容器无法使用 CRI 接口启动，甚至无法使用 CRI 接口删除。

   容器的真实 LOGPATH=log_directory/log_path，如果 log_path 不配置，那么最终的 LOGPATH 会变为 LOGPATH=log_directory。

   - 如果该路径不存在，isulad 在启动容器时会创建一个软链接，指向最终的容器日志真实路径，此时 log_directory 变成一个软链接，此时有两种情况：
     i. 第一种情况，如果该 sandbox 里其它容器也没配置 log_path，在启动其它容器时，log_directory 会被删除，然后重新指向新启动容器的 log_path，导致之前启动的容器日志指向后面启动容器的日志。
     ii. 第二种情况，如果该 sandbox 里其它容器配置了 log_path，则该容器的 LOGPATH=log_directory/log_path，由于 log_directory 实际是个软链接，使用 log_directory/log_path 为软链接指向容器真实日志路径时，创建会失败。
   - 如果该路径存在，isulad 在启动容器时首先会尝试删除该路径（非递归），如果该路径是个文件夹，且里面有内容，删除会失败，从而导致创建软链接失败，容器启动失败，删除该容器时，也会出现同样的现象，导致删除失败。

2. 如果创建 sandbox 时，PodSandboxConfig 参数中配置了 log_directory，且 container 创建时在 ContainerConfig 中指定 log_path，那么最终的 LOGPATH=log_directory/log_path，isulad 不会递归的创建 LOGPATH，因而用户必须保证 dirname(LOGPATH)存在，即最终的日志文件的上一级路径存在。

3. 如果创建 sandbox 时，PodSandboxConfig 参数中配置了 log_directory，如果有两个或多个 container 创建时在 ContainerConfig 中指定了同一个 log_path，或者不同的 sandbox 内的容器最终指向的 LOGPATH 是同一路径，若容器启动成功，则后启动的容器日志路径会覆盖掉之前启动的容器日志路径。

4. 如果远程镜像仓库中镜像内容发生变化，调用 CRI Pull image 接口重新下载该镜像时，若本地原来存储有原镜像，则原镜像的镜像名称、TAG 会变更为“none”

   举例如下：

   本地已存储镜像：

```
IMAGE                                          TAG                   IMAGE ID          SIZE
rnd-dockerhub.huawei.com/pproxyisulad/test   latest                99e59f495ffaa
753kB
```

   远程仓库中 rnd-dockerhub.huawei.com/pproxyisulad/test:latest 镜像更新后，重新下载后：

```
IMAGE                                          TAG                   IMAGE ID          SIZE
<none>                                         <none>                99e59f495ffaa
753kB
rnd-dockerhub.huawei.com/pproxyisulad/test   latest                d8233ab899d41
1.42MB
```

使用 isula images 命令行查询，REF 显示为"-"：

```
REF                                              IMAGE ID            CREATED
SIZE
rnd-dockerhub.huawei.com/pproxyisulad/test:latest d8233ab899d41        2019-
02-14 19:19:37  1.42MB
-                                                99e59f495ffaa        2016-05-04
02:26:41  753kB
```

1.3.6 镜像管理

1.3.6.1 docker 镜像管理

1.3.6.1.1 登录到镜像仓库

描述

isula login 命令用于登录到镜像仓库。登录成功后可以使用 isula pull 命令从该镜像仓库拉取镜像。如果镜像仓库不需要密码，则拉取镜像前不需要执行该命令。

用法

```
isula login [OPTIONS] SERVER
```

参数

login 命令支持参数请参见“附录 > 表 1-21”。

示例

```
$ isula login -u abc my.csp-edge.com:5000

Login Succeeded
```

1.3.6.1.2 从镜像仓库退出登录

描述

isula logout 命令用于从镜像仓库退出登录。退出登录成功后再执行 isula pull 命令从该镜像仓库拉取镜像会因为未认证而拉取失败。

用法

```
isula logout SERVER
```

参数

logout 命令支持参数请参见“附录 > 表 1-22”。

示例

```
$ isula logout my.csp-edge.com:5000
Logout Succeeded
```

1.3.6.1.3 从镜像仓库拉取镜像

描述

从镜像仓库拉取镜像到本地。

用法

```
isula pull [OPTIONS] NAME[:TAG|@DIGEST]
```

参数

pull 命令支持参数请参见“附录 >表 1-23”。

示例

```
$ isula pull localhost:5000/official/busybox
Image "localhost:5000/official/busybox" pulling
Image
"localhost:5000/official/busybox@sha256:bf510723d2cd2d4e3f5ce7e93bf1e52c8fd76831995
ac3bd3f90ecc866643aff" pulled
```

1.3.6.1.4 删除镜像

描述

删除一个或多个镜像。

用法

```
isula rmi [OPTIONS] IMAGE [IMAGE...]
```

参数

rmi 命令支持参数请参见“附录 >表 1-24”。

示例

```
$ isula rmi rnd-dockerhub.huawei.com/official/busybox
Image "rnd-dockerhub.huawei.com/official/busybox" removed
```

1.3.6.1.5 加载镜像

描述

从一个 tar 包加载镜像。该 tar 包必须是使用 docker save 命令导出的 tar 包或格式一致的 tar 包。

用法

```
isula load [OPTIONS]
```

参数

load 命令支持参数请参见“附录 >表 1-25”。

示例

```
$ isula load -i busybox.tar
Load image from "/root/busybox.tar" success
```

1.3.6.1.6 列出镜像

描述

列出当前环境中所有镜像。

用法

```
isula images
```

参数

images 命令支持参数请参见“附录 >表 1-26”。

示例

```
$ isula images
REF                                              IMAGE ID           CREATED             SIZE
rnd-dockerhub.huawei.com/official/busybox:latest e4db68de4ff2       2019-06-15
08:19:54  1.376 MB
```

1.3.6.1.7 检视镜像

描述

返回该镜像的配置信息。可以使用-f 参数过滤出需要的信息。

用法

```
isula inspect [options] CONTAINER|IMAGE [CONTAINER|IMAGE...]
```

参数

inspect 命令支持参数请参见“附录 >表 1-27”。

示例

```
$ isula inspect -f "{{json .image.id}}" rnd-dockerhub.huawei.com/official/busybox
"e4db68de4ff27c2adfea0c54bbb73a61a42f5b667c326de4d7d5b19ab71c6a3b"
```

1.3.6.1.8 双向认证

描述

开启该功能后 isulad 和镜像仓库之间的通信采用 https 通信，isulad 和镜像仓库都会验证对方的合法性。

用法

要支持该功能，需要镜像仓库支持该功能，同时 isulad 也需要做相应的配置：

1. 修改 isulad 的配置(默认路径/etc/isulad/daemon.json)，将配置里的 use-decrypted-key 项配置为 false。
2. 需要将相关的证书放置到/etc/isulad/certs.d 目录下对应的镜像仓库命名的文件夹下，证书具体的生成方法见 docker 的官方链接：
   - https://docs.docker.com/engine/security/certificates/
   - https://docs.docker.com/engine/security/https/
3. 执行 systemctl restart isulad 重启 isulad。

参数

可以在/etc/isulad/daemon.json 中配置参数，也可以在启动 isulad 时携带参数：

```
isulad --use-decrypted-key=false
```

示例

配置 use-decrypted-key 参数为 false

```
$ cat /etc/isulad/daemon.json
{
   "group": "isulad",
   "graph": "/var/lib/isulad",
   "state": "/var/run/isulad",
   "engine": "lcr",
   "log-level": "ERROR",
   "pidfile": "/var/run/isulad.pid",
   "log-opts": {
      "log-file-mode": "0600",
      "log-path": "/var/lib/isulad",
      "max-file": "1",
      "max-size": "30KB"
   },
   "log-driver": "stdout",
   "hook-spec": "/etc/default/isulad/hooks/default.json",
   "start-timeout": "2m",
   "storage-driver": "overlay2",
   "storage-opts": [
      "overlay2.override kernel check=true"
   ],
   "registry-mirrors": [
      "docker.io"
```

```
    ],
    "insecure-registries": [
        "rnd-dockerhub.huawei.com"
    ],
    "pod-sandbox-image": "",
    "image-opt-timeout": "5m",
    "native.umask": "secure",
    "network-plugin": "",
    "cni-bin-dir": "",
    "cni-conf-dir": "",
    "image-layer-check": false,
    "use-decrypted-key": false,
    "insecure-skip-verify-enforce": false
}
```

将证书放到对应的目录下

```
$ pwd
/etc/isulad/certs.d/my.csp-edge.com:5000
$ ls
ca.crt  tls.cert  tls.key
```

重启 isulad

```
$ systemctl restart isulad
```

执行 pull 命令从仓库下载镜像

```
$ isula pull my.csp-edge.com:5000/busybox
Image "my.csp-edge.com:5000/busybox" pulling
Image "my.csp-
edge.com:5000/busybox@sha256:f1bdc62115dbfe8f54e52e19795ee34b4473babdeb9bc4f83045d8
5c7b2ad5c0" pulled
```

1.3.6.2 embedded 镜像管理

1.3.6.2.1 加载镜像

描述

根据 embedded 镜像的 manifest 加载镜像。注意--type 的值必须填写 embedded。

用法

```
isula load [OPTIONS] --input=FILE --type=TYPE
```

参数

load 命令支持参数请参见“附录 >表 1-25”。

示例

```
$ isula load -i test.manifest --type embedded
Load image from "/root/work/bugfix/tmp/ci testcase data/embedded/img/test.manifest"
success
```

1.3.6.2.2 列出镜像

描述

列出当前环境中所有镜像。

用法

```
isula images [OPTIONS]
```

参数

images 命令支持参数请参见“附录 >表 1-26”。

示例

```
$ isula images
REF                          IMAGE ID           CREATED             SIZE
test:v1                       9319da1f5233       2018-03-01 10:55:44  1.273 MB
```

1.3.6.2.3 检视镜像

描述

返回该镜像的配置信息。可以使用-f 参数过滤出需要的信息。

用法

```
isula inspect [options] CONTAINER|IMAGE [CONTAINER|IMAGE...]
```

参数

inspect 命令支持参数请参见“附录 >表 1-27”。

示例

```
$ isula inspect -f "{{json .created}}" test:v1
"2018-03-01T15:55:44.322987811Z"
```

1.3.6.2.4 删除镜像

描述

删除一个或多个镜像。

用法

```
isula rmi [OPTIONS] IMAGE [IMAGE...]
```

参数

rmi 命令支持参数请参见“附录 > 表 1-24”。

示例

```
$ isula rmi test:v1
Image "test:v1" removed
```

1.3.7 容器健康状态检查

1.3.7.1 场景说明

在实际的生产环境中，开发者提供的应用程序或者平台提供的服务难免存在 bug，因此，一套管理系统对运行的应用程序进行周期性的健康检查和修复就是不可或缺的。容器健康检查机制便添加了用户定义的对容器进行健康检查的功能。在容器创建时配置--health-cmd 选项，在容器内部周期性地执行命令，通过命令的返回值来监测容器的健康状态。

1.3.7.2 配置方法

在容器启动时的配置：

```
isula run -itd --health-cmd "echo iSulad >> /tmp/health check file || exit 1" --
health-interval 5m --health-timeout 3s --health-exit-on-unhealthy  busybox bash
```

可配置的选项：

- --health-cmd，必选，在容器内执行的命令。返回值为 0 表示成功，非 0 表示失败。
- --health-interval，默认 30s，最大为 int64 上限（纳秒），自定义配置最小值 1s，相邻两次命令执行的间隔时间（注：入参 0s 时视为 default）。
- --health-timeout，默认 30s，最大为 int64 上限（纳秒），自定义配置最小值 1s，单次检查命令执行的时间上限，超时则任务命令执行失败（注：入参 0s 时视为 default），仅支持 runtime 类型为 lcr 的容器。
- --health-start-period，默认 0s，最大为 int64 上限（纳秒），自定义配置最小值 1s，容器初始化时间。
- --health-retries，默认 3，最大为 int32 上限，健康检查失败最大的重试次数。
- --health-exit-on-unhealthy，默认 false，检测到容器非健康时是否杀死容器。

1.3.7.3 检查规则

1. 容器启动后，容器状态中显示 health:starting。
2. 经过 start-period 时间后开始，以 interval 为间隔周期性在容器中执行 CMD。即：当一次命令执行完毕后，经过 interval 时间，执行下一次命令。
3. 若 CMD 命令在 timeout 限制的时间内执行完毕，并且返回值为 0，则视为一次检查成功。否则视为一次检查失败。检查成功后，容器状态变为 health:healthy。
4. 若 CMD 命令连续 retries 次检查失败，则容器状态变为 health:unhealthy。失败后容器也会继续进行健康检查。
5. 容器状态为 health:unhealthy 时，任意一次检查成功会使得容器状态变为 health:healthy。
6. 设置--exit-on-unhealthy 的情况下，如果容器因为非被杀死退出（退出返回值 137）后，健康检查只有容器在重新启动后才会继续生效。

7. CMD 执行完毕或超时时，docker daemon 会将这次检查的起始时间、返回值和标准输出记录到容器的配置文件中。最多记录 5 条。此外，容器的配置文件中还存储着健康检查的相关参数。
8. 运行中的容器的健康检查状态也会被写入容器配置中。通过 isula inspect 可以看到。

```
"Health": {
    "Status": "healthy",
    "FailingStreak": 0,
    "Log": [
        {
            "Start": "2018-03-07T07:44:15.481414707-05:00",
            "End": "2018-03-07T07:44:15.556908311-05:00",
            "ExitCode": 0,
            "Output": ""
        },
        {
            "Start": "2018-03-07T07:44:18.557297462-05:00",
            "End": "2018-03-07T07:44:18.63035891-05:00",
            "ExitCode": 0,
            "Output": ""
        },
        ......
}
```

1.3.7.4 使用限制

- 容器内健康检查的状态信息最多保存 5 条。会保存最后得到的 5 条记录。
- 容器启动时若健康检查相关参数配置为 0，则按照默认值处理。
- 带有健康检查配置的容器启动后，若 iSulad daemon 退出，则健康检查不会执行。iSulad daemon 再次启动后，正在运行且带有健康检查配置的容器其健康状态会变为 starting。之后检查规则同上。
- 如果健康检查从第一次开始便一直失败，其状态保持与之前一致（starting）,直到达到指定失败次数（--health-retries）后变为 unhealthy，或者检查成功后变为 healthy。
- 对于 OCI 类型的 runtime 的容器，健康检查功能待完善。目前仅完整支持 lcr 类型的容器。

1.3.8 查询信息

1.3.8.1 查询服务版本信息

描述

isula version 命令用于查询 iSulad 服务的版本信息。

用法

```
isula version
```

实例

查询版本信息

```
isula version
```

如果 isulad 服务正常运行，则可以查看到客户端、服务端以及 OCI config 的版本等信息。

```
Client:
  Version:     1.0.31
  Git commit:   fa7f9902738e8b3d7f2eb22768b9a1372ddd1199
  Built:       2019-07-30T04:21:48.521198248-04:00

Server:
  Version:     1.0.31
  Git commit:   fa7f9902738e8b3d7f2eb22768b9a1372ddd1199
  Built:       2019-07-30T04:21:48.521198248-04:00

OCI config:
  Version:     1.0.0-rc5-dev
  Default file: /etc/default/isulad/config.json
```

若 isulad 服务未运行，则仅仅查询到客户端的信息，并提示无法连接到服务端。

```
Client:
  Version:     1.0.31
  Git commit:   fa7f9902738e8b3d7f2eb22768b9a1372ddd1199
  Built:       2019-07-30T04:21:48.521198248-04:00

Can not connect with server.Is the iSulad daemon running on the host?
```

因此，isula version 命令也常常用来检验 isulad 是否正常运行。

1.3.8.2 查询系统级信息

描述

isula info 命令用于对系统级信息，以及容器和镜像数目等信息的查询。

用法

```
isula info
```

示例

查询系统级信息，可以展示容器数目，镜像数目，内核版本、操作系统等信息

```
$ isula info
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 8
Server Version: 1.0.31
Logging Driver: json-file
```

```
Cgroup Driverr: cgroupfs
Hugetlb Pagesize: 2MB
Kernel Version: 4.19
Operating System: Fedora 29 (Twenty Nine)
OSType: Linux
Architecture: x86 64
CPUs: 8
Total Memory: 7 GB
Name: localhost.localdomain
iSulad Root Dir: /var/lib/isulad
```

1.3.9 安全特性

1.3.9.1 seccomp 安全配置场景

1.3.9.1.1 场景说明

seccomp（**secure computing mode**）是 linux kernel 从 2.6.23 版本开始引入的一种简洁的 sandboxing 机制。在一些特定场景下，用户需要在容器中执行一些“特权”操作，但又不想启动特权容器，用户经常会在 run 时添加--cap-add 来获得一些“小范围”的权限。对于安全要求比较严格的容器实例，上述的 CAP 粒度不一定能够满足安全需要，可使用一些办法精细化控制权限范围。

- 举例

  普通容器场景中，用户使用-v 将宿主机某目录（包含某普通用户无法执行的二进制），映射到容器中。

  在容器中，可以将二进制修改权限 chmod 4777 加入 S 标志位。这样在宿主机上，原先不能运行二进制的普通用户（或者运行此二进制受限），可以在 S 标志位的添加动作后，在运行此二进制的时候，获取到二进制自身的权限（比如 root），从而提权运行或者访问其他文件。

  这个场景，如果在严格安全要求下，需要使用 seccomp 裁剪 chmod、fchmod、fchmodat 系统调用。

1.3.9.1.2 使用限制

- seccomp 可能会影响性能，设置 seccomp 之前需要对场景进行评估，确定必要时加入 seccomp 配置。

1.3.9.1.3 使用指导

通过--security-opt 将配置文件传给要过滤系统调用的容器。

```
isula run -itd --security-opt seccomp=/path/to/seccomp/profile.json rnd-dockerhub.huawei.com/official/busybox
```

说明

1. 创建容器时通过--security-opt 将配置文件传给容器时，采用默认配置文件（/etc/isulad/seccomp_default.json）。
2. 创建容器时--security-opt 设置为 unconfined 时，对容器不过滤系统调用。
3. “/path/to/seccomp/profile.json”需要是绝对路径。

获取普通容器的默认 seccomp 配置

- 启动一个普通容器（或者是带--cap-add 的容器），并查看默认权限配置：

```
cat /etc/isulad/seccomp_default.json | python -m json.tool > profile.json
```

  可以看到"seccomp"字段中，有很多的"syscalls"，在此基础上，仅提取 syscalls 的部分，参考定制 seccomp 配置文件，进行定制化操作。

```
"defaultAction": "SCMP_ACT_ERRNO",
"syscalls": [
{
"action": "SCMP_ACT_ALLOW",
"name": "accept"
},
{
"action": "SCMP_ACT_ALLOW",
"name": "accept4"
},
{
"action": "SCMP_ACT_ALLOW",
"name": "access"
},
{
"action": "SCMP_ACT_ALLOW",
"name": "alarm"
},
{
"action": "SCMP_ACT_ALLOW",
"name": "bind"
},
]...
```

- 查看转换为 lxc 可识别的 seccomp 配置

```
cat
/var/lib/isulad/engines/lcr/74353e38021c29314188e29ba8c1830a4677ffe5c4decda77a1
e0853ec8197cd/seccomp
...
waitpid allow
write allow
writev allow
ptrace allow
personality allow [0,0,SCMP_CMP_EQ,0]
personality allow [0,8,SCMP_CMP_EQ,0]
personality allow [0,131072,SCMP_CMP_EQ,0]
personality allow [0,131080,SCMP_CMP_EQ,0]
personality allow [0,4294967295,SCMP_CMP_EQ,0]
...
```

定制 seccomp 配置文件

在启动容器的时候使用--security-opt 引入 seccomp 配置文件，容器实例会按照配置文件规则进行限制系统 API 的运行。首先获取普通容器的默认 seccomp，得到完整模板，然后按照本节定制配置文件，启动容器：

```
isula run --rm -it --security-opt seccomp:/path/to/seccomp/profile.json rnd-dockerhub.huawei.com/official/busybox
```

配置文件模板：

```
{
"defaultAction": "SCMP_ACT_ALLOW",
"syscalls": [
{
"name": "syscall-name",
"action": "SCMP_ACT_ERRNO",
"args": null
}
]
}
```

须知

- defaultAction、syscalls：对应的 action 的类型是一样的，但其值是不能一样的，目的就是让所有的 syscal 都有一个默认的 action，并且如果 syscalls 数组中有明确的定义，就以 syscalls 中的为准，由于 defaultAction、action 的值不一样，就能保证 action 不会有冲突。当前支持的 action 有：

  "SCMP_ACT_ERRNO"：禁止，并打印错误信息。

  "SCMP_ACT_ALLOW"：允许。

- syscalls： 数组，可以只有一个 syscall，也可以有多个，可以带 args，也可以不带。
- name：要过滤的 syscall。
- args：数组，里面的每个 object 的定义如下：

```
type Arg struct {
Index    uint    `json:"index"`     //参数的序号，如 open(fd, buf, len),fd 对应的就是 0，buf 为 1
Value    uint64  `json:"value"`     //跟参数进行比较的值
ValueTwo uint64  `json:"value_two"` //仅当 Op=MaskEqualTo 时起作用，用户传入值跟 Value 按位与操作后，跟 ValueTwo 进行比较，若相等则执行 action。
Op       Operator `json:"op"`
}
```

  args 中的 Op，其取值可以下页面的任意一种：

  "SCMP_CMP_NE": NotEqualTo

  "SCMP_CMP_LT": LessThan

  "SCMP_CMP_LE": LessThanOrEqualTo

  "SCMP_CMP_EQ": EqualTo

  "SCMP_CMP_GE": GreaterThanOrEqualTo

"SCMP_CMP_GT": GreaterThan
"SCMP_CMP_MASKED_EQ": MaskEqualTo

1.3.9.2 capabilities 安全配置场景

1.3.9.2.1 场景说明

capabilities 机制是 linux kernel 2.2 之后引入的安全特性，用更小的粒度控制超级管理员权限,可以避免使用 root 权限，将 root 用户的权限细分为不同的领域，可以分别启用或禁用。capabilities 详细信息可通过 Linux Programmer's Manual 进行查看（capabilities(7) - Linux man page）：

```
man capabilities
```

1.3.9.2.2 使用限制

- isulad 默认 Capabilities（白名单）配置如下，普通容器进程将默认携带：

```
"CAP_CHOWN",
"CAP_DAC_OVERRIDE",
"CAP_FSETID",
"CAP_FOWNER",
"CAP_MKNOD",
"CAP_NET_RAW",
"CAP_SETGID",
"CAP_SETUID",
"CAP_SETFCAP",
"CAP_SETPCAP",
"CAP_NET_BIND_SERVICE",
"CAP_SYS_CHROOT",
"CAP_KILL",
"CAP_AUDIT_WRITE"
```

- 默认的权能配置，包含了 CAP_SETUID 和 CAP_FSETID，如 host 和容器共享目录，容器可对共享目录的二进制文件进行文件权限设置，host 上的普通用户可能使用该特性进行提权攻击。CAP_AUDIT_WRITE，容器可以对 host 写入，存在一定的风险，如果使用场景不需要，推荐在启动容器的时候使用--cap-drop 将其删除。
- 增加 Capabilities 意味着容器进程具备更大的能力，同时也会开放更多的系统调用接口。

1.3.9.2.3 使用指导

iSulad 使用--cap-add/--cap-drop 给容器增加/删去特定的权限，在非必要情况下，不要给容器增加额外的权限，推荐将容器默认但非必要的权限也去掉。

```
isula run --rm -it --cap-add all --cap-drop SYS_ADMIN rnd-dockerhub.huawei.com/official/busybox
```

1.3.9.3 SELinux 安全配置场景

1.3.9.3.1 场景说明

SELinux(Security-Enhanced Linux)是一个 linux 内核的安全模块，提供了访问控制安全策略机制，iSulad 将采用 MCS（多级分类安全）实现对容器内进程打上标签限制容器访问资源的方式，减少提权攻击的风险，防止造成更为重要的危害。

1.3.9.3.2 使用限制

- 确保宿主机已使能 SELinux，且 daemon 端已打开 SELinux 使能开发（/etc/isulad/daemon.json 中“selinux-enabled”字段为 true，或者命令行参数添加--selinux-enabled）
- 确保宿主机上已配置合适的 selinux 策略，推荐使用 container-selinux 进行配置
- 引入 SELinux 会影响性能，设置 SELinux 之前需要对场景进行评估，确定必要时打开 daemon 端 SELinux 开关并设置容器 SELinux 配置
- 对挂载卷进行标签配置时，源目录不允许为/、/usr、/etc、/tmp、/home、/run、/var、/root 以及/usr 的子目录。

说明

- 目前 iSulad 不支持对容器的文件系统打标签，确保容器文件系统及配置目录打上容器可访问标签，需使用 chcon 命令对其打上标签。
- 若 iSulad 启用 SELinux 访问控制，建议 daemon 启动前对/var/lib/isulad 目录打上标签，容器容器创建时目录下生产的文件及文件夹将默认继承其标签，例如：

```
chcon -R system_u:object_r:container_file_t:s0 /var/lib/isulad
```

1.3.9.3.3 使用指导

- daemon 端使能 selinux：

```
isulad --selinux-enabled
```

- 启动容器时配置 selinux 标签安全上下文

  --security-opt="label=user:USER" 配置安全上下文用户

  --security-opt="label=role:ROLE" 配置安全上下文角色

  --security-opt="label=type:TYPE" 配置安全上下文类型

  --security-opt="label=level:LEVEL" 配置安全上下文域

  --security-opt="label=disable" 容器禁用 SELinux 配置

```
$ isula run -itd --security-opt label=type:container t --security-opt
label=level:s0:c1,c2 rnd-dockerhub.huawei.com/official/centos
9be82878a67e36c826b67f5c7261c881ff926a352f92998b654bc8e1c6eec370
```

- 为挂载卷打 selinux 标签('z'为共享模式)

```
$ isula run -itd -v /test:/test:z rnd-dockerhub.huawei.com/official/centos
9be82878a67e36c826b67f5c7261c881ff926a352f92998b654bc8e1c6eec370

$ls -Z /test
system_u:object_r:container_file_t:s0 file
```

1.3.10 支持 OCI hooks

1.3.10.1 描述

支持在容器的生命周期中，运行 OCI 标准 hooks。包括三种类型的 hooks：

- prestart hook：在执行 isula start 命令之后，而在容器的 1 号进程启动之前，被执行。
- poststart hook：在容器 1 号进程启动之后，而在 isula start 命令返回之前，被执行。
- poststop hook：在容器被停止之后，但是在停止命令返回之前，被执行。

OCI hooks 的配置格式规范如下：

- path：格式是字符串，必须项，必须为绝对路径，指定的文件必须有可执行权限。
- args：格式是字符串数组，可选项，语义和 execv 的 args 一致。
- env：格式是字符串数组，可选项，语义和环境变量一致，内容为键值对，如："PATH=/usr/bin"。
- timeout：格式是整数，可选项，必须大于 0，表示钩子执行的超时时间。如果钩子进程运行时间超过配置的时间，那么钩子进程会被杀死。

hook 的配置为 json 格式，一般存放在 json 结尾的文件中，示例如下：

```
{
       "prestart": [
           {
              "path": "/usr/bin/echo",
              "args": ["arg1", "arg2"],
              "env":  [ "key1=value1"],
              "timeout": 30
           },
           {
              "path": "/usr/bin/ls",
              "args": ["/tmp"]
           }
       ],
       "poststart": [
           {
              "path": "/usr/bin/ls",
              "args": ["/tmp"],
              "timeout": 5
           }
       ],
       "poststop": [
           {
              "path": "/tmp/cleanup.sh",
              "args": ["cleanup.sh", "-f"]
           }
       ]
}
```

1.3.10.2 接口

isulad 和 isula 都提供了 hook 的接口，isulad 提供了默认的 hook 配置，会作用于所有容器；而 isula 提供的 hook 接口，只会作用于当前创建的容器。

isulad 提供的默认 OCI hooks 配置：

- 通过/etc/isulad/daemon.json 配置文件的 hook-spec 配置项设置 hook 配置的文件路径："hook-spec": "/etc/default/isulad/hooks/default.json"。
- 通过 isulad --hook-spec 参数设置 hook 配置的文件路径。

isula 提供的 OCI hooks 配置：

- isula create --hook-spec：指定 hook 配置的 json 文件的路径。
- isula run --hook-spec：指定 hook 配置的 json 文件的路径。

run 的配置其实也是在 create 阶段生效了。

1.3.10.3 使用限制

- hook-spec 指定的路径必须是绝对路径。
- hook-spec 指定的文件必须存在。
- hook-spec 指定的路径对应的必须是普通文本文件，格式为 json。
- hook-spec 指定的文件大小不能超过 10MB。
- hooks 配置的 path 字段必须为绝对路径。
- hooks 配置的 path 字段指定文件必须存在。
- hooks 配置的 path 字段指定文件必须有可执行权限。
- hooks 配置的 path 字段指定文件的 owner 必须是 root。
- hooks 配置的 path 字段指定文件必须只有 root 有写权限。
- hooks 配置的 timeout 必须大于 0。

1.4 附录

1.4.1 命令行参数说明

表1-21 login 命令参数列表

命令	参数	说明
login	-H, --host	指定要连接的 iSulad socket 文件路径
	-p, --password	登录镜像仓库的密码
	--password-stdin	从标准输入获取仓库的密码

	-u, --username	登录镜像仓库的用户名

表1-22 logout 命令参数列表

命令	参数	说明
logout	-H, --host	指定要连接的 iSulad socket 文件路径

表1-23 pull 命令参数列表

命令	参数	说明
pull	-H, --host	指定要连接的 iSulad socket 文件路径

表1-24 rmi 命令参数列表

命令	参数	说明
rmi	-H, --host	指定要连接的 iSulad socket 文件路径
	-f, --force	强制移除镜像

表1-25 load 命令参数列表

命令	参数	说明
load	-H, --host （仅 isula 支持）	指定要连接的 iSulad socket 文件路径
	-i, --input	指定从哪里导入镜像。如果是 docker 类型，则为镜像压缩包路径，如果是 embedded 类型，则为镜像 manifest 路径。
	--tag	不使用默认的镜像名称，而是使用 TAG 指定的名称，type 为 docker 类型时支持该参数
	-t, --type	镜像类型，取值为 embedded 或 docker（默认值）

表1-26 images 命令参数列表

命令	参数	说明
images	-H, --host	指定要连接的 iSulad socket 文件路径
	-q, --quit	只显示镜像名字

表1-27 inspect 命令参数列表

命令	参数	说明
inspect	-H, --host	指定要连接的 iSulad socket 文件路径
	-f, --format	使用模板格式化输出
	-t, --time	超时时间的秒数，若在该时间内 inspect 查询容器信息未执行成功，则停止等待并立即报错，默认为 120 秒，当配置小于等于 0 的值，表示不启用 timeout 机制 inspect 查询容器信息会一直等待，直到获取容器信息成功后返回。

1.4.2 CNI 配置参数

表1-28 CNI 单网络配置参数

参数	类型	是否可选	说明
cniVersion	string	必选	CNI 版本号，当前只支持 0.3.0，0.3.1。
name	string	必选	网络名称，由用户自定义，需保证唯一。
type	string	必选	网络类型。目前支持的网络类型： underlay_ipvlan overlay_l2 underlay_l2 vpc-router dpdk-direct

参数	类型	是否可选	说明
			phy-direct
ipmasp	bool	可选	设置 IP masquerade
ipam	结构体	可选	详细定义参考 IPAM 参数定义
ipam.type	string	可选	IPAM 类型，目前支持的类型： （1）underlay_l2、overlay_l2、vpc-router 组网默认值 distributed_l2，且只支持 distributed_l2。 （2）underlay_ipvlan 组网，默认 distributed_l2。**CCN 场景只支持 null、fixed；CCE 和 FST 5G core 场景只支持 null、distributed_l2。** （3）phy-direct、dpdk-direct 组网，默认 l2，可选 null、distributed_l2。**FST 5G core 场景只支持 null、distributed_l2。** 说明： 超出选择范围（比如 host-local），Canal 会自动设置为默认，不会返回错误。 null：不使用 canal 管理 ip。 fixed：固定 ip，CCN 场景使用。 l2：目前没有场景使用。 distributed_l2：使用分布式小子网管理 ip。
ipam.subnet	string	可选	子网信息。Canal 支持的 subnet mask 范围为[8,29]，并且要求 IP 地址不能为 Multicast 地址（如 224.0.0.0/4），保留地址（240.0.0.0/4），本地 link 地址（169.254.0.0/16）以及本地 loop 地址（127.0.0.0/8）。
ipam.gateway	string	可选	网关 IP
ipam.range-start	string	可选	可用的起始 IP 地址
ipam.range-end	string	可选	可用的结束 IP 地址
ipam.routes	结构体	可	subnet 列表，每个元素都是一个 route 字

参数	类型	是否可选	说明
		选	典。参考 route 定义.
ipam.routes.dst	string	可选	表示目的网络
ipam.routes.gw	string	可选	表示网关地址
dns	结构体	可选	包含一些 DNS 的特殊值。
dns.nameservers	[]string	可选	nameservers
dns.domain	string	可选	domain
dns.search	[]string	可选	search
dns.options	[]string	可选	选项
multi_entry	int	可选	表示一个 vnic 需要的 ip 数量,范围 0~16。对于物理直通，单个网卡最多可申请 128 个 IP。
backup_mode	bool	可选	表示主备模式，仅用于 phy-direct 和 dpdk-direct 组网。
vlanID	int	可选	0~4095,允许 PaaS 直接指定。
vlan_inside	bool	可选	true 表示 vlan 功能由 Node 内部实现， false 表示 vlan 在外部实现。
vxlanID	int	可选	0~16777215,允许 PaaS 直接指定。
vxlan_inside	bool	可选	true 表示 vlan 功能由 Node 内部实现， false 表示 vlan 在外部实现。
action	string	可选	该参数只能和特殊 containerID “000000000000”一起使用。 Create 表示创建网络。 Delete 表示删除网络。
args	map[string]interface{}	可选	主要描述键值对类型。表 1-29

参数	类型	是否可选	说明
runtimeConfig	结构体	可选	无
capabilities	结构体	可选	无

表1-29 CNI args 参数表

参数	类型	是否可选	说明
K8S_POD_NAME	string	可选	申请固定 IP（runtimeConfig.ican_caps.fixed_ip 为 true）时需要设置 K8S_POD_NAME
K8S_POD_NAMESPACE	string	可选	申请固定 IP（runtimeConfig.ican_caps.fixed_ip 为 true）时需要设置 K8S_POD_NAMESPACE
SECURE_CONTAINER	string	可选	安全容器标志
multi_port	int	可选	默认值为 1，取值范围 1-8。只支持 phy-direct 和 dpdk-direct 两种类型网络，指定直通网卡数量
phy-direct	string	可选	用于在创建硬直通容器网络时指定接入的网卡
dpdk-direct	string	可选	用于在创建 dpdk 直通容器网络时指定接入的网卡
tenant_id	string	可选	租户的 ID。 **只支持 vpc-router 类型网络。**
vpc_id	string	可选	VPC 的 ID。 **只支持 vpc-router 类型网络。**
secret_name	string	可选	表示 k8s apiserver 中保存有 ak sk 的对象名。 **只支持 vpc-router 类型网络** 参考配置 VPC-Router 逻辑网络

参数	类型	是否可选	说明
IP	string	可选	用户指定 ip 地址，格式“192.168.0.10”
K8S_POD_NETWORK_ARGS	string	可选	指定 ip 地址，格式“192.168.0.10”。若 args 中 IP 和 K8S_POD_NETWORK_ARGS 都不为空，以 K8S_POD_NETWORK_ARGS 为准。
INSTANCE_NAME	string	可选	INSTANCE ID。 参考支持容器固定 IP
dist_gateway_disable	bool	可选	true 表示不创建 gateway，false 表示创建 gateway。
phynet	string 或 []string	可选	所需加入的的物理平面信息，为预先定义好的物理网络名称，与 SNC 体系中的呼应，输入两个平面名时，支持主备平面。例如："phy_net1" 或 ["phy_net2","phy_net3"]

endpoint_policies	struct	可选	"endpoint_policies": [
			{
			"Type": "",
			"ExceptionList": [
			""
			],
			"NeedEncap": true,
			"DestinationPrefix": ""
			}
			]

port_map	struct	可选	NAT 类型网络中，支持容器端口发布至主机端口。 "port_map": [{ "local_port": number, "host_port": number, "protocol": [string…] }...]

表1-30 CNI 多网络配置参数

参数	类型	是否可选	说明
cniVersion	string	必选	CNI 版本号，当前只支持 0.3.0，0.3.1。
name	string	必选	网络名称，由用户自定义，需保证唯一。
plugins	struct	必选	具体配置请参见表1-28。

2 系统容器

2.1 概述
2.2 安装指导
2.3 使用指南
2.4 附录

2.1 概述

系统容器主要应对在重计算、高性能、大并发的场景下，重型应用和业务云化的问题。相比较虚拟机技术，系统容器可直接继承物理机特性，同时具备性能更优良，较少 overhead 的优点。从系统资源分配来看，系统容器在有限资源上相比虚拟机可分配更多计算单元，降低成本，通过系统容器可以构建产品的差异化竞争力，提供计算密度更高，价格更便宜，性能更优良的的计算单元实例。

2.2 安装指导

步骤 1 首先需要安装 iSulad 容器引擎。

```
# yum install iSulad
```

步骤 2 安装系统容器依赖包。

```
# yum install isulad-tools authz isulad-lxcfs-toolkit lxcfs
```

步骤 3 查看 iSulad 是否已经启动。

```
# systemctl status isulad
```

步骤 4 开启 lxcfs 和 authz 服务。

```
# systemctl start lxcfs
# systemctl start authz
```

----结束

2.3 使用指南

2.3.1 简介

系统容器基于 iSula 容器引擎进行功能增强，提供系统容器相关功能。系统容器提供的容器管理功能和 iSula 容器引擎保持一致，其命令格式和功能与 iSula 容器引擎相同。

本文档仅描述系统容器提供的增强功能对应的使用方式，其它命令行操作请参考"1 iSula 容器引擎"章节。

系统容器功能仅涉及 isula create/run 命令行，后续未特别说明，各功能均使用此命令行。其命令行格式如下所示：

```
isula create/run [OPTIONS] [COMMAND] [ARG...]
```

其中：

- OPTIONS：命令参数，可以一个或者多个，可选参数请参见“1 iSula 容器引擎 > 1.4 附录 > 1.4.1 命令行参数说明”中对应内容。
- COMMAND：系统容器启动后执行的命令。
- ARG：系统容器启动后执行命令对应的参数。

2.3.2 指定 rootfs 创建容器

功能描述

系统容器不同于普通容器，普通容器需要指定一个容器镜像来启动，而系统容器通过参数 --external-rootfs 指定一个本地的根文件系统 rootfs（Root File System）来启动，rootfs 包含了容器运行时依赖的操作系统环境。

参数说明

命令	参数	参数指定值说明
isula create/run	--external-rootfs	• 字符串变量。 • 容器根文件系统对应的绝对路径，即 rootfs 的路径。

约束限制

- 参数--external-rootfs 指定的 rootfs 目录必须为绝对路径，不能为相对路径。
- 参数--external-rootfs 指定的 rootfs 目录必须为一个完整运行的操作系统环境，否则容器会启动失败。
- 容器删除时，不会删除--external-rootfs 指定的 rootfs 目录。
- 不支持在 x86 环境上运行基于 arm rootfs 的容器，也不支持在 arm 环境上运行基于 x86 rootfs 的容器。

- 同一份 rootfs，不建议启动多个容器实例，即同一份 rootfs 只供一个生命周期内的容器实例使用。

使用示例

假设本地 rootfs 的路径为/root/myrootfs，那么启动一个系统容器的命令如下：

```
# isula run  -tid --system-container --external-rootfs /root/myrootfs none init
```

说明

rootfs 为自定义的文件系统，请用户自行准备。例如容器镜像的 tar 包解压后，即为一个 rootfs。

2.3.3 通过 systemd 启动容器

功能描述

系统容器与普通容器最大的差异就在于容器启动的 init 进程，普通容器无法通过 systemd 启动系统服务，而系统容器具备这个能力，通过在启动容器时指定 --system-contianer 参数可以使能 systemd 服务。

参数说明

命令	参数	参数指定值说明
isula create/run	--system-container	• 布尔变量，取值为 true、false，未指定值时表示 true。 • 指定某个容器类型是否属于系统容器，必须使能。

约束限制

- systemd 服务需要调用一些特殊系统调用，包括 mount、umount2、unshare、reboot 以及 name_to_handle_at，所以在不开启特权容器标签的情况下，系统容器打开了调用上述接口的权限。
- 系统容器都是 init 启动，init 进程不响应表示正常退出的 SIGTERM 信号，stop 默认在 10s 之后才会强制杀死容器。如果需要快速结束，可以手动指定 stop 的超时时间。
- --system-container 必须配合--external-rootfs 参数一起使用。
- 系统容器内支持运行各类服务，服务的启停通过 systemctl 来管理，服务之间可能会出现相互依赖关系导致异常情况下某些服务进程出现 D/Z 状态，使得容器无法正常退出。
- 系统容器内的某些服务进程可能会影响其它操作结果，例如容器内若运行了 NetworkManager 服务，可能会影响向容器添加网卡的行为（网卡添加成功然后被 NetworkManger 停掉），导致不可预期的结果。
- 系统容器和主机暂时无法实现 udev 事件隔离，所以 fstab 配置也暂不支持。
- systemd 服务可能和 libcgroup 提供的 cgconfig 服务在功能上出现冲突，建议在容器内去掉 libcgroup 相关的包或者配置 cgconfig 服务的 Delegate 值为 no。

使用示例

- 指定--system-container 和--external-rootfs 参数启动系统容器。

```
[root@localhost ~]# isula run -tid -n systest01 --system-container --external-
rootfs /root/myrootfs none init
```

- 执行以上命令后容器成功运行，通过 exec 进容器查看进程信息，可看到 systemd 服务已启动。

```
[root@localhost ~]# isula exec -it systest01 bash
[root@localhost /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  2 06:49 ?        00:00:00 init
root        14     1  2 06:49 ?        00:00:00 /usr/lib/systemd/systemd-journal
root        16     1  0 06:49 ?        00:00:00 /usr/lib/systemd/systemd-network
dbus        23     1  0 06:49 ?        00:00:00 /usr/bin/dbus-daemon --system --
root        25     0  0 06:49 ?        00:00:00 bash
root        59    25  0 06:49 ?        00:00:00 ps -ef
```

- 容器内执行 systemctl 命令查看服务状态，可看到服务被 systemd 管理。

```
[root@localhost /]# systemctl status dbus
● dbus.service - D-Bus System Message Bus
   Loaded: loaded (/usr/lib/systemd/system/dbus.service; static; vendor preset:
disabled)
   Active: active (running) since Mon 2019-07-22 06:49:38 UTC; 2min 5
8s ago
     Docs: man:dbus-daemon(1)
 Main PID: 23 (dbus-daemon)
   CGroup: /system.slice/dbus.service
           └─23 /usr/bin/dbus-daemon --system --address=systemd: --nofork --
nopidf
ile --systemd-activation --syslog-only

Jul 22 06:49:38 localhost systemd[1]: Started D-Bus System Message Bus.
```

- 容器内通过 systemctl stop/start 服务，可看到服务被 systemd 管理。

```
[root@localhost /]# systemctl stop dbus
Warning: Stopping dbus.service, but it can still be activated by:
  dbus.socket
[root@localhost /]# systemctl start dbus
```

2.3.4 容器内 reboot/shutdown

功能描述

系统容器支持在容器内执行 reboot 和 shutdown 命令。执行 reboot 命令效果同重启容器一致；执行 shutdown 命令效果同停止容器一致。

参数说明

命令	参数	参数指定值说明
isula create/run	--restart	● 字符串变量。 ● 可取指定值：

命令	参数	参数指定值说明
		on-reboot：表示重启系统容器。

约束限制

- shutdown 功能，依赖于不同的 OS，以实际容器运行环境对应 OS 为准。
- 执行“shutdown -h now”命令关闭系统时，不能多次占用 console。例如“isula run -ti”命令打开一个 console，在另一个 host bash 中 isula attach 该容器，会打开另一个 console，此时执行 shutdown 会失败。

使用示例

- 容器启动时指定--restart on-reboot 参数，示例如下：

```
[root@localhost ~]# isula run -tid --restart on-reboot --system-container --
external-rootfs /root/myrootfs none init
106faae22a926e22c828a0f2b63cf5c46e5d5986ea8a5b26de81390d0ed9714f
```

- 进入容器执行 reboot 命令：

```
[root@localhost ~]# isula exec -it 10 bash
[root@localhost /]# reboot
```

查看容器是否重启：

```
[root@localhost ~]# isula exec -it 10 ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0  21588  9504 ?        Ss   12:11   0:00 init
root        14  0.1  0.0  27024  9376 ?        Ss   12:11   0:00 /usr/lib/system
root        17  0.0  0.0  18700  5876 ?        Ss   12:11   0:00 /usr/lib/system
dbus        22  0.0  0.0   9048  3624 ?        Ss   12:11   0:00 /usr/bin/dbus-d
root        26  0.0  0.0   8092  3012 ?        Rs+  12:13   0:00 ps aux
```

- 进入容器执行 shutdown 命令：

```
[root@localhost ~]# isula exec -it 10 bash
[root@localhost /]# shutdown -h now
[root@localhost /]# [root@localhost ~]#
```

检查容器是否停止：

```
[root@localhost ~]# isula exec -it 10 bash
Error response from daemon: Exec container error;Container is not
running:106faae22a926e22c828a0f2b63cf5c46e5d5986ea8a5b26de81390d0ed9714f
```

2.3.5 cgroup 路径可配置

功能描述

系统容器提供在宿主机上进行容器资源隔离和预留的能力。通过--cgroup-parent 参数，可以将容器使用的 cgroup 目录指定到某个特定目录下，从而达到灵活分配宿主机资源的目的。例如可以设置容器 a、b、c 的 cgroup 父路径为/lxc/cgroup1，容器 d、e、f 的 cgroup 父路径为/lxc/cgroup2，这样通过 cgroup 路径将容器分为两个 group，实现容器 cgroup 组层面的资源隔离。

参数说明

命令	参数	参数指定值说明
isula create/run	--cgroup-parent	• 字符串变量。 • 指定容器 cgroup 父路径。

除了通过命令行指定单个系统容器对应的 cgroup 父路径外，还可通过修改 iSulad 容器引擎启动配置文件，指定所有容器的 cgroup 路径。

配置文件路径	配置项	配置项说明
/etc/isulad/daemon.json	--cgroup-parent	• 字符串变量。 • 指定容器默认 cgroup 父路径。 • 配置示例："cgroup-parent": "/lxc/mycgroup"

约束限制

- 如果 daemon 端和客户端都设置了 cgroup parent 参数，最终以客户端指定的--cgroup-parent 生效。
- 如果已启动容器 A，然后启动容器 B，容器 B 的 cgroup 父路径指定为容器 A 的 cgroup 路径，在删除容器的时候需要先删除容器 B 再删除容器 A，否则会导致 cgroup 资源残留。

使用示例

启动系统容器，指定--cgroup-parent 参数：

```
[root@localhost ~]# isula run -tid --cgroup-parent /lxc/cgroup123 --system-container --external-rootfs /root/myrootfs none init
115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
```

查看容器 init 进程的 cgroup 信息：

```
[root@localhost ~]# isula inspect -f "{{json .State.Pid}}" 11
22167
[root@localhost ~]# cat /proc/22167/cgroup
13:blkio:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
12:perf_event:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
11:cpuset:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
10:pids:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
9:rdma:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
8:devices:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac460
```

```
9f332e
7:hugetlb:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac460
9f332e
6:memory:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609
f332e
5:net cls,net prio:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4
b9ecac4609f332e
4:cpu,cpuacct:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9eca
c4609f332e
3:files:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f
332e
2:freezer:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac460
9f332e
1:name=systemd:/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ec
ac4609f332e/init.scope
0::/lxc/cgroup123/115878a4dfc7c5b8c62ef8a4b44f216485422be9a28f447a4b9ecac4609f332e
```

可以看到容器的 cgroup 父路径被设置为/sys/fs/cgroup/<controller>/lxc/cgroup123

同时，对于所有容器 cgroup 父路径的设置可以配置一下容器 daemon 文件，例如：

```
{
        "cgroup-parent": "/lxc/cgroup123",
}
```

然后重启容器引擎，配置生效。

2.3.6 namespace 化内核参数可写

功能描述

对于运行在容器内的业务，如数据库，大数据，包括普通应用，有对部分内核参数进行设置和调整的需求，以满足最佳的业务运行性能和可靠性。内核参数要么不允许修改，要么全部允许修改（特权容器）：

在不允许用户在容器内修改时，只提供了--sysctl 外部接口，而且容器内不能灵活修改参数值。

在允许用户在容器内修改时，部分内核参数是全局有效的，某个容器修改后，会影响主机上所有的程序，安全性降低。

系统容器提供--ns-change-opt 参数，可以指定 namespace 化的内核参数在容器内动态设置，当前仅支持 net、ipc。

参数说明

命令	参数	参数指定值说明
isula create/run	--ns-change-opt	• 字符串变量。 • 仅支持配置 net、ipc： net：支持/proc/sys/net 目录下所有 namespace 化参数。

命令	参数	参数指定值说明
		ipc：支持的 namespace 化参数列表如下： /proc/sys/kernel/msgmax /proc/sys/kernel/msgmnb /proc/sys/kernel/msgmni /proc/sys/kernel/sem /proc/sys/kernel/shmall /proc/sys/kernel/shmmax /proc/sys/kernel/shmmni /proc/sys/kernel/shm_rmid_forced /proc/sys/fs/mqueue/msg_default /proc/sys/fs/mqueue/msg_max /proc/sys/fs/mqueue/msgsize_default /proc/sys/fs/mqueue/msgsize_max /proc/sys/fs/mqueue/queues_max ● 支持通知指定多个 namespce 配置，多个配置间通过逗号隔开，例如：--ns-change-opt=net,ipc。

约束限制

- 如果容器启动同时指定了--privileged（特权容器）和--ns-change-opt，则--ns-change-opt 不生效。

使用示例

启动容器， 指定--ns-change-opt=net：

```
[root@localhost ~]# isula run -tid --ns-change-opt net --system-container --
external-rootfs /root/myrootfs none init
4bf44a42b4a14fdaf127616c90defa64b4b532b18efd15b62a71cbf99ebc12d2
[root@localhost ~]# isula exec -it 4b mount | grep /proc/sys
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sys/net type proc (rw,nosuid,nodev,noexec,relatime)
```

可以看到容器内/proc/sys/net 挂载点为 rw，说明 net 相关的 namespace 化的内核参数具有读写权限。

再启动一个容器，指定--ns-change-opt=ipc：

```
[root@localhost ~]# isula run -tid --ns-change-opt ipc --system-container --
external-rootfs /root/myrootfs none init
c62e5e5686d390500dab2fa76b6c44f5f8da383a4cbbeac12cfada1b07d6c47f
[root@localhost ~]# isula exec -it c6 mount | grep /proc/sys
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
```

```
proc on /proc/sys/kernel/shmmax type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/shmmni type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/shmall type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/shm rmid forced type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/msgmax type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/msgmni type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/msgmnb type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/kernel/sem type proc (rw,nosuid,nodev,noexec,relatime)
proc on /proc/sys/fs/mqueue type proc (rw,nosuid,nodev,noexec,relatime)
```

可以看到容器内 ipc 相关的内核参数挂载点为 rw，说明 ipc 相关的 namespace 化的内核参数具有读写权限。

2.3.7 共享内存通道

功能描述

系统容器提供容器与主机进程通过共享内存进行通信的功能，通过在容器创建时配置--host-channel 参数，可以在容器与主机之间共享同一 tmpfs，从而达到主机与容器间通信的功能。

参数说明

命令	参数	参数指定值说明
isula create/run	--host-channel	• 字符串变量，格式为： `<host path>:<container path>:<rw/ro>:<size limit>` • 参数说明如下： <host path>：将在宿主机上挂载 tmpfs 的路径，必须是绝对路径。 <container path>：将在容器内挂载 tmpfs 的路径，必须是绝对路径。 <rw/ro>：在容器内挂载的文件系统的使用权限，只能配置为 rw（可读写）或 ro（只读），默认为 rw。 <size limit>：挂载的 tmpfs 能够使用的最大限制，最小支持 1 物理页（4KB），最大支持系统总物理内存的 1/2 。默认为 64MB。

约束限制

- 宿主机上挂载的 tmpfs 的生命周期为从容器启动到容器删除，容器删除并解除对空间的占用后会移除这片空间。
- 容器删除时会将宿主机上挂载 tmpfs 的路径删除，所以不允许使用宿主机上已存在的目录。
- 为了宿主机上非 root 用户运行的进程能够与容器内进行通信，宿主机上 tmpfs 挂载的权限为 1777。

使用示例

创建容器时指定--host-channel 参数：

```
[root@localhost ~]# isula run --rm -it --host-channel /testdir:/testdir:rw:32M --
system-container --external-rootfs /root/myrootfs none init
root@3b947668eb54:/# dd if=/dev/zero of=/testdir/test.file bs=1024 count=64K
dd: error writing '/testdir/test.file': No space left on device
32769+0 records in
32768+0 records out
33554432 bytes (34 MB, 32 MiB) copied, 0.0766899 s, 438 MB/s
```

说明

- 使用--host-channel 大小限制时，若在容器内创建共享文件，则会受到容器内的内存配额限制（在容器内存使用达到极限时可能会产生 oom）。
- 若用户在主机端创建共享文件，则不受容器内的内存配额限制。
- 若用户需要在容器内创建共享文件，且业务为内存密集型，可以通过设置容器内存配额为在原本基础上加上--host-channel 配置的大小来消除影响。

2.3.8 动态加载内核模块

功能描述

容器内业务可能依赖某些内核模块，可通过设置环境变量的方式，在系统容器启动前动态加载容器中业务需要的内核模块到宿主机，此特性需要配合 isulad-hooks 一起使用，具体使用可参看 2.3.12 容器资源动态管理（syscontainer-tools）章节。

参数说明

命令	参数	参数指定值说明
isula create/run	-e KERNEL_MODULES=module_name1,module_name	• 字符串变量。 • 支持配置多个模块，模块名以逗号分隔。

约束限制

- 如果加载的内核模块是未经过验证的，或者跟宿主机已有模块冲突的场景，会导致宿主机出现不可预知问题，在做加载内核模块时需要谨慎操作。
- 动态加载内核模块通过将需要加载的内核模块传递给容器，此功能是依靠 isulad-tools 捕获到容器启动的环境变量实现，依赖 isulad-tools 的正确安装部署。
- 加载的内核模块需要手动进行删除。

使用示例

启动系统容器时，指定-e KERNEL_MODULES 参数，待系统容器启动后，可以看到 ip_vs 模块被成功加载到内核中。

```
[root@localhost ~]# lsmod | grep ip_vs
[root@localhost ~]# isula run -tid -e KERNEL_MODULES=ip_vs,ip_vs_wrr --hook-spec
/etc/isulad-tools/hookspec.json --system-container --external-rootfs /root/myrootfs
none init
ae18c4281d5755a1e153a7bff6b3b4881f36c8e528b9baba8a3278416a5d0980
[root@localhost ~]# lsmod | grep ip_vs
ip_vs_wrr            16384  0
ip_vs               176128  2 ip_vs_wrr
nf_conntrack        172032  7
xt_conntrack,nf_nat,nf_nat_ipv6,ipt_MASQUERADE,nf_nat_ipv4,nf_conntrack_netlink,ip_
vs
nf_defrag_ipv6        20480  2 nf_conntrack,ip_vs
libcrc32c            16384  3 nf_conntrack,nf_nat,ip_vs
```

说明

- 宿主机需要安装 isulad-tools。
- 需要指定--hooks-spec 为 isulad hooks。

2.3.9 环境变量持久化

功能描述

系统容器支持通过指定--env-target-file 接口参数将 env 变量持久化到容器 rootfs 目录下的配置文件中。

参数说明

命令	参数	参数指定值说明
isula create/run	--env-target-file	• 字符串变量。 • env 持久化文件必须在 rootfs 目录下，且配置为绝对路径。

约束限制

- --env-target-file 指定的目标文件如果存在的话，大小不能超过 10MB。
- --env-target-file 指定的参数为 rootfs 目录下的绝对路径。
- 如果--env 和目标文件里面的 env 出现冲突，以--env 指定值的参数为准。

使用示例

启动系统容器，指定 env 环境变量和--env-target-file 参数：

```
[root@localhost ~]# isula run -tid -e abc=123 --env-target-file /etc/environment --
system-container --external-rootfs /root/myrootfs none init
b75df997a64da74518deb9a01d345e8df13eca6bcc36d6fe40c3e90ea1ee088e
[root@localhost ~]# isula exec b7 cat /etc/environment
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
abc=123
```

可以看到容器的 env 变量（abc=123）已经持久化到/etc/environment 配置文件中。

2.3.10 最大句柄数限制

功能描述

系统容器支持对容器内使用文件句柄数进行限制，文件句柄包括普通文件句柄和网络套接字，启动容器时，可以通过指定--files-limit 参数限制容器内打开的最大句柄数。

参数说明

命令	参数	参数指定值说明
isula create/run	--files-limit	• 整数值，不能为负数。 • 指定为 0 表示不受限制，最大值限制由当前内核 files cgroup 决定。

约束限制

- 如果--files-limit 指定的值太小，可能会导致系统容器无法通过 exec 执行命令，报"open too many files"错误，所以 files limit 的值应该设置大一些。
- 文件句柄包括普通文件句柄和网络套接字。

使用示例

使用--files-limit 限制容器内打开文件句柄数需要内核支持 files cgroup，可以执行以下命令查看：

```
[root@localhost ~]# cat /proc/1/cgroup | grep files
10:files:/
```

结果显示 files，说明内核支持 files cgroup。

容器启动指定--files-limit 参数，并检查 files.limit 参数是否成功写入：

```
[root@localhost ~]# isula run  -tid --files-limit 1024 --system-container  --
external-rootfs /tmp/root-fs  empty init
01e82fcf97d4937aa1d96eb8067f9f23e4707b92de152328c3fc0ecb5f64e91d
[root@localhost ~]# isula exec -it 01e82fcf97d4 bash
[root@localhost ~]# cat /sys/fs/cgroup/files/files.limit
1024
```

可以看出，容器内文件句柄数被成功限制。

2.3.11 安全性和隔离性

2.3.11.1 user namespace 多对多

功能描述

user namespace 是将容器的 root 映射到主机的普通用户，使得容器中的进程和用户在容器里有特权，但是在主机上就是普通权限，防止容器中的进程逃逸到主机上，进行非法操作。更进一步，使用 user namespace 技术后，容器和主机使用不同的 uid 和 gid，保证容器内部的用户资源和主机资源进行隔离，例如文件描述符等。

系统容器支持通过--user-remap 接口参数将不同容器的 user namespace 映射到宿主机不同的 user namespace，实现容器 user namespace 隔离。

参数说明

命令	参数	参数指定值说明
isula create/run	--user-remap	参数格式为<uid>:<gid>:<offset>，参数说明如下： • uid、gid 为整数型，且必须大于等于 0。 • offset 为整数型，且必须大于 0，并且小于 65536。取值不能太小，否则容器无法启动。 • uid 加上 offset 的值必须小于等于 2^{32}-1，gid 加上 offset 的值必须小于等于 2^{32}-1，否则容器启动会报错。

约束限制

- 如果系统容器指定了--user-remap，那么 rootfs 目录必须能够被--user-remap 指定的 uid/gid 用户所访问，否则会导致容器 user namespace 无法访问 rootfs，容器启动失败。
- 容器内所有的 id 都应该能映射到主机 rootfs，某些目录/文件可能是从主机 mount 到容器，比如/dev/pts 目录下面的设备文件，如果 offset 值太小可能会导致 mount 失败。
- uid、gid 和 offset 的值由上层调度平台控制，容器引擎只做合法性检查。
- --user-remap 只适用于系统容器。
- --user-remap 和--privileged 不能共存，否则容器启动会报错。
- 如果 uid 或 gid 指定为 0，则--user-remap 参数不生效。

使用指导

说明

指定--user-remap 参数前，请先将 rootfs 下所有目录和文件的 uid 和 gid 做整体偏移，偏移量为--user-remap 指定 uid 和 gid 的偏移量。

例如将 dev 目录的 uid 和 gid 整体 uid 和 gid 偏移 100000 的参考命令为：

chown 100000:100000 dev

系统容器启动指定--user-remap 参数：

```
[root@localhost ~]# isula run -tid --user-remap 100000:100000:65535 --system-container --external-rootfs /home/root-fs none /sbin/init
eb9605b3b56dfae9e0b696a729d5e1805af900af6ce24428fde63f3b0a443f4a
```

分别在宿主机和容器内查看/sbin/init 进程信息：

```
[root@localhost ~]# isula exec eb ps aux | grep /sbin/init
root       1  0.6  0.0  21624  9624 ?       Ss   15:47   0:00 /sbin/init
[root@localhost ~]# ps aux | grep /sbin/init
100000    4861  0.5  0.0  21624  9624 ?       Ss   15:47   0:00 /sbin/init
root      4948  0.0  0.0 213032   808 pts/0    S+   15:48   0:00 grep --color=auto /sbin/init
```

可以看到/sbin/init 进程在容器内的 owner 是 root 用户，但是在宿主机的 owner 是 uid=100000 这个用户。

在容器内创建一个文件，然后在宿主机上查看文件的 owner：

```
[root@localhost ~]# isula exec -it eb bash
[root@localhost /]# echo test123 >> /test123
[root@localhost /]# exit
exit
[root@localhost ~]# ll /home/root-fs/test123
-rw-------. 1 100000 100000 8 Aug  2 15:52 /home/root-fs/test123
```

可以看到，在容器内生成了一个文件，它的 owner 是 root，但是在宿主机上看到的 owner 是 id=100000 这个用户。

2.3.11.2 用户权限控制

功能描述

容器引擎支持通过 TLS 认证方式来认证用户的身份，并依此控制用户的权限，当前容器引擎可以对接 authz 插件实现权限控制。

接口说明

通过配置 iSulad 容器引擎启动参数来指定权限控制插件，daemon 配置文件默认为/etc/isulad/daemon.json。

配置参数	示例	说明
--authorization-plugin	"authorization-plugin": "authz-broker"	用户权限认证插件，当前只支持 authz-broker。

约束限制

- authz 需要配置用户权限策略，策略文件默认为/var/lib/authz-broker/policy.json，该配置文件支持动态修改，修改完即时生效，不需要重启插件服务。

- 由于容器引擎为 root 用户启动，放开一般用户使用的一些命令可能会导致该用户不当获得过大权限，需谨慎配置。目前 container_attach、container_create 和 container_exec_create 动作可能会有风险。
- 对于某些复合操作，比如 isula exec、isula attach 等命令依赖 isula inspect 是否有权限，如果用户没有 inspect 权限会直接报错。
- 采用 SSL/TLS 加密通道在增加安全性的同时也会带来性能损耗，如增加延时，消耗较多的 CPU 资源，除了数据传输外，加解密需要更大吞吐量，因此在并发场景下，相比非 TLS 通信，其并发量有一定程度上的下降。经实测，在 ARM 服务器（Cortex-A72 64 核）接近空载情况下，采用 TLS 并发起容器，其最大并发量在 200~250 范围内。
- 服务端指定--tlsverify 时，认证文件默认配置路径为/etc/isulad。且默认文件名分别为 ca.pem、cert.pem、key.pem。

使用示例

步骤 1 确认宿主机安装了 authz 插件，如果需要安装，安装并启动 authz 插件服务命令如下：

```
[root@localhost ~]# yum install authz
[root@localhost ~]# systemctl start authz
```

步骤 2 要启动该功能，首先需要配置容器引擎和用户的 TLS 证书。可以使用 OPENSSL 来生成需要的证书，具体步骤如下：

```
#SERVERSIDE

# Generate CA key
openssl genrsa -aes256 -passout "pass:$PASSWORD" -out "ca-key.pem" 4096
# Generate CA
openssl req -new -x509 -days $VALIDITY -key "ca-key.pem" -sha256 -out "ca.pem" -
passin "pass:$PASSWORD" -subj
"/C=$COUNTRY/ST=$STATE/L=$CITY/O=$ORGANIZATION/OU=$ORGANIZATIONAL_UNIT/CN=$COMMON_N
AME/emailAddress=$EMAIL"
# Generate Server key
openssl genrsa -out "server-key.pem" 4096

# Generate Server Certs.
openssl req -subj "/CN=$COMMON_NAME" -sha256 -new -key "server-key.pem" -out
server.csr

echo "subjectAltName = DNS:localhost,IP:127.0.0.1" > extfile.cnf
echo "extendedKeyUsage = serverAuth" >> extfile.cnf

openssl x509 -req -days $VALIDITY -sha256 -in server.csr -passin "pass:$PASSWORD" -
CA "ca.pem" -CAkey "ca-key.pem" -CAcreateserial -out "server-cert.pem" -extfile
extfile.cnf

#CLIENTSIDE


openssl genrsa -out "key.pem" 4096
openssl req -subj "/CN=$CLIENT_NAME" -new -key "key.pem" -out client.csr
echo "extendedKeyUsage = clientAuth" > extfile.cnf
openssl x509 -req -days $VALIDITY -sha256 -in client.csr -passin "pass:$PASSWORD" -
```

```
CA "ca.pem" -CAkey "ca-key.pem" -CAcreateserial -out "cert.pem" -extfile
extfile.cnf
```

若要直接使用以上过程作为脚本，需替换各变量为配置数值。生成 CA 时使用的参数若为空则写为“"”。PASSWORD、COMMON_NAME、CLIENT_NAME、VALIDITY 为必选项。

步骤 3 容器引擎启动时添加 TLS 相关参数和认证插件相关参数，并保证认证插件的运行。此外，为了使用 TLS 认证，容器引擎必须使用 TCP 侦听的方式启动，不能使用传统的 unix socket 的方式启动。容器 demon 端配置如下：

```
{
    "tls": true,
    "tls-verify": true,
    "tls-config": {
          "CAFile": "/root/.iSulad/ca.pem",
          "CertFile": "/root/.iSulad/server-cert.pem",
          "KeyFile":"/root/.iSulad/server-key.pem"
    },
    "authorization-plugin": "authz-broker"
}
```

步骤 4 然后需要配置策略，对于基本授权流程，所有策略都位于一个配置文件下/var/lib/authz-broker/policy.json。该配置文件支持动态修改，更改时不需要重新启动插件，只需要向 authz 进程发送 SIGHUP 信号。文件格式是每行一个策略 JSON 对象。每行只有一个匹配。具体的策略配置示例如下：

- 所有用户都可以运行所有 iSulad 命令：{"name":"policy_0","users":[""],"actions":[""]}。
- Alice 可以运行所有 iSulad 命令：{"name":"policy_1","users":["alice"],"actions":[""]}。
- 空用户都可以运行所有 iSulad 命令：{"name":"policy_2","users":[""],"actions":[""]}。
- Alice 和 Bob 可以创建新的容器：{"name":"policy_3","users":["alice","bob"],"actions":["container_create"]}。
- service_account 可以读取日志并运行 docker top：{"name":"policy_4","users":["service_account"],"actions":["container_logs","container_top"]}。
- Alice 可以执行任何 container 操作：{"name":"policy_5","users":["alice"],"actions":["container"]}。
- Alice 可以执行任何 container 操作，但请求的种类只能是 get：{"name":"policy_5","users":["alice"],"actions":["container"], "readonly":true }。

说明

- 配置中匹配 action 支持正则表达式。
- users 不支持正则表达式。
- users 不能有重复用户，即同一用户不能被多条规则匹配。

步骤 5 配置并更新完之后，客户端配置 TLS 参数连接容器引擎，即是以受限的权限访问。

```
[root@localhost ~]#  isula version --tlsverify --tlscacert=/root/.iSulad/ca.pem --tlscert=/root/.iSulad/cert.pem --tlskey=/root/.iSulad/key.pem  -H=tcp://127.0.0.1:2375
```

如果想默认配置 TLS 认证进行客户端连接，可以将文件移动到~/.iSulad，并设置 ISULAD_HOST 和 ISULAD_TLS_VERIFY 变量（而不是每次调用时传递 -H=tcp://$HOST:2375 和--tlsverify）。

```
[root@localhost ~]# mkdir -pv ~/.iSulad
[root@localhost ~]# cp -v {ca,cert,key}.pem ~/.iSulad
[root@localhost ~]# export ISULAD_HOST=localhost:2375 ISULAD_TLS_VERIFY=1
[root@localhost ~]# isula version
```

----结束

2.3.11.3 proc 文件系统隔离（lxcfs）

场景描述

容器虚拟化带来轻量高效，快速部署的同时，也因其隔离性不够彻底，给用户带来一定程度的使用不便。由于 Linux 内核 namespace 本身还不够完善，因此容器在隔离性方面也存在一些缺陷。例如，在容器内部 proc 文件系统中可以看到宿主机上的 proc 信息（如 meminfo， cpuinfo，stat， uptime 等）。利用 lxcfs 工具可以将容器内的看到宿主机/proc 文件系统的内容，替换成本容器实例的相关/proc 内容，以便容器内业务获取正确的资源数值。

接口说明

系统容器对外提供两个工具包：一个是 lxcfs 软件，另外一个是配合 lxcfs 一起使用的 lxcfs-toolkit 工具。其中 lxcfs 作为宿主机 daemon 进程常驻，lxcfs-toolkit 通过 hook 机制将宿主机的 lxcfs 文件系统绑定挂载到容器。

lxcfs-toolkit 命令行格式如下：

```
lxcfs-toolkit [OPTIONS] COMMAND [COMMAND_OPTIONS]
```

命令	功能说明	参数
remount	将 lxcfs 重新 mount 到容器中	--all：对所有的容器执行 remout lxcfs 操作 --container-id：remount lxcfs 到特定的容器 ID
umount	将 lxcfs 从容器中 umount 掉	--all：对所有的容器执行 umout lxcfs 操作 --container-id：对特定容器执行 umount lxcfs 操作
check-lxcfs	检查 lxcfs 服务是否运行	无
prestart	在 lxcfs 服务启动前将 /var/lib/lxcfs 目录 mount 到	无

命令	功能说明	参数
	容器中	

约束限制

- 当前只支持 proc 文件系统下的 cpuinfo, meminfo, stat, diskstats, partitions，swaps 和 uptime 文件，其他的文件和其他内核 API 文件系统（比如 sysfs）未做隔离 。
- 安装 rpm 包后会在/var/lib/isulad/hooks/hookspec.json 生成样例 json 文件，用户如果需要增加日志功能，需要在定制时加入--log 配置。
- diskstats 只能显示支持 cfq 调度的磁盘信息，无法显示分区信息。容器内设备会被显示为/dev 目录下的名字。若不存在则为空。此外，容器根目录所在设备会被显示为 sda。
- 挂载 lxcfs 时必须使用 slave 参数。若使用 shared 参数，可能会导致容器内挂载点泄露到主机，影响主机运行 。
- lxcfs 支持服务优雅降级使用，若 lxcfs 服务 crash 或者不可用，容器内查看到的 cpuinfo, meminfo, stat, diskstats, partitions, swaps 和 uptime 均为 host 信息，容器其它业务功能不受影响。
- lxcfs 底层依赖 fuse 内核模块以及 libfuse 库，因此需要内核支持 fuse。
- lxcfs 当前仅支持容器内运行 64 位的 app，如果容器内运行 32 位的 app 可能会导致 app 读取到的 cpuinfo 信息不符合预期。
- lxcfs 只是对容器 cgroup 进行资源视图模拟，对于容器内的系统调用（例如 sysconf）获取到的仍然是主机的信息，lxcfs 无法做到内核隔离。
- lxcfs 使用隔离后的 cpuinfo 显示的 cpu 信息具有如下特征：
  - processor：从 0 开始依次递增。
  - physical id：从 0 开始依次递增。
  - sibliing：固定为 1。
  - core id：固定为 0。
  - cpu cores：固定为 1。

使用示例

步骤 1 首先需要安装 lxcfs 和 lxcfs-toolkit 这两个包，并启动 lxcfs 服务。

```
[root@localhost ~]# yum install lxcfs lxcfs-toolkit
[root@localhost ~]# systemctl start lxcfs
```

步骤 2 容器启动完成之后查看容器内是否存在 lxcfs 挂载点。

```
[root@localhost ~]# isula run -tid -v /var/lib/lxc:/var/lib/lxc --hook-spec
/var/lib/isulad/hooks/hookspec.json --system-container --external-rootfs
/home/root-fs none init
a8acea9fea1337d9fd8270f41c1a3de5bceb77966e03751346576716eefa9782
[root@localhost ~]# isula exec a8 mount | grep lxcfs
lxcfs on /var/lib/lxc/lxcfs type fuse.lxcfs
(rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other)
```

```
lxcfs on /proc/cpuinfo type fuse.lxcfs
(rw,nosuid,nodev,relatime,user id=0,group id=0,allow other)
lxcfs on /proc/diskstats type fuse.lxcfs
(rw,nosuid,nodev,relatime,user id=0,group id=0,allow other)
lxcfs on /proc/meminfo type fuse.lxcfs
(rw,nosuid,nodev,relatime,user id=0,group id=0,allow other)
lxcfs on /proc/partitions type fuse.lxcfs
(rw,nosuid,nodev,relatime,user id=0,group id=0,allow other)
lxcfs on /proc/stat type fuse.lxcfs
(rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other)
lxcfs on /proc/swaps type fuse.lxcfs
(rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other)
lxcfs on /proc/uptime type fuse.lxcfs
(rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other)
```

步骤 3 执行 update 命令更新容器的 cpu 和 mem 资源配置，然后查看容器资源。根据如下回显可知，容器资源视图显示的是容器真实资源数据而不是宿主机的数据。

```
[root@localhost ~]# isula update --cpuset-cpus 0-1 --memory 1G a8
a8
[root@localhost ~]# isula exec a8 cat /proc/cpuinfo
processor      : 0
BogoMIPS       : 100.00
cpu MHz        : 2400.000
Features       : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part       : 0xd08
CPU revision    : 2

processor      : 1
BogoMIPS       : 100.00
cpu MHz        : 2400.000
Features       : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part       : 0xd08
CPU revision    : 2

[root@localhost ~]# isula exec a8 free -m
           total       used       free     shared  buff/cache   available
Mem:         1024         17        997          7          8        1006
Swap:        4095          0       4095
```

----结束

2.3.12 容器资源动态管理（syscontainer-tools）

普通容器无法支持对容器内的资源进行管理，例如添加一个块设备到容器、插入一块物理/虚拟网卡到容器。系统容器场景下，通过 syscontainer-tools 工具可以实现动态为容器挂载/卸载块设备，网络设备，路由和卷等资源。

要使用此功能，需要安装 syscontainer-tools 工具：

```
[root@localhost ~]# yum install syscontainer-tools
```

2.3.12.1 设备管理

功能描述

isulad-tools 支持将宿主机上的块设备（比如磁盘、LVM）或字符设备（比如 GPU、binner、fuse）添加到容器中。在容器中使用该设备，例如可以对磁盘进行 fdisk 格式化，写入 fs 等操作。在容器不需要设备时，isulad-tools 可以将设备从容器中删除，归还宿主机。

命令格式

```
isulad-tools [COMMADN][OPTIONS] <container_id> [ARG...]
```

其中：

COMMAND：设备管理相关的命令。

OPTIONS：设备管理命令支持的选项。

container_id：容器 id。

ARG：命令对应的参数。

参数说明

命令	功能说明	选项说明	参数说明
add-device	将宿主机块设备/字符设备添加到容器中。	支持的选项如下： • --blkio-weight-device：设置块设备 IO 权重（相对权重，10-100 之间）。 • --device-read-bps：设置块设备读取速率限制（byte/秒）。 • --device-read-iops：设置块设备读取速率限制（IO/秒）。 • --device-write-bps：设置块设备写入速率限制（byte/秒）。 • --device-write-iops：设置块设备写入速率限制（IO/秒）。 • --follow-partition：如果块设备是基础块设备（主 SCSI 块磁盘），加入此参数可以添加主磁	参数格式为：hostdevice[:containerdevice][:permission] [hostdevice[:containerdevice][:permission] ...] 其中： hostdevice：设备在主机上的路径。 containerdevice：设备在容器中的路径。 permission：容器内对设备的操作权限。

命令	功能说明	选项说明	参数说明
		盘下的所有分区。 • --force：如果容器中已有块设备/字符设备，使用此参数覆盖旧的块设备/字符设备文件。 • --update-config-only：只更新配置文件不实际做添加磁盘动作。	
remove-device	将块设备/字符设备从容器中删除，还原至宿主机。	支持的选项如下： --follow-partition：如果块设备是基础块设备（主SCSI 块磁盘），加入此参数可以删除容器中主磁盘下的所有分区，还原至宿主机。	参数格式为： hostdevice[:containerdevice] [hostdevice[:containerdevice] ...] 其中： hostdevice：设备在主机上的路径。 containerdevice：设备在容器中的路径。
list-device	列出容器中所有的块设备/字符设备。	支持的选项如下： • --pretty：按照 json 格式输出。 • --sub-partition：如果某磁盘为主磁盘，加入此flag，在显示主磁盘的同时，也显示主磁盘的子分区。	无
update-device	更新磁盘Qos。	支持的选项如下： • --device-read-bps：设置块设备读取速率限制（byte/秒），建议设置值大于等于 1024。 • --device-read-iops：设置块设备读取速率限制（IO/秒）。 • --device-write-bps：设置块设备写入速率限制（byte/秒），建议设置值大于等于 1024。 • --device-write-iops：设置块设备写入速率限制（IO/秒）。	无

约束限制

- 添加/删除设备的时机可以是容器实例非运行状态，完成操作后启动容器，容器内会有体现；也可以在容器运行时（running）动态添加。
- 不能在容器内和 host 上并发进行 fdisk 对磁盘的格式化写入，会影响容器磁盘使用。
- add-device 将磁盘添加到容器的特定目录时，如果容器内的父目录为多级目录（比如/dev/a/b/c/d/e...）且目录层级不存在，则 isulad-tools 会自动在容器内创建对应目录；当删除时，不会将创建的父目录删除。如果用户下一次 add-device 到该父目录，则会提示已经存在无法添加成功。
- add-device 添加磁盘、更新磁盘参数时，配置磁盘 Qos；当配置磁盘 Qos 的 read/write bps、read/write IOPS 值时，不建议配置值过小，当设置过小时，会造成磁盘表现为不可读（实际原因是速度过慢），最终影响业务功能。
- 使用--blkio-weight-device 来限制指定块设备的权重，如果当前块设备仅支持 BFQ 模式，可能会报错，提示用户检查当前 OS 环境是否支持 BFQ 块设备权重值设置。

使用示例

- 启动一个系统容器，指定 hook spec 为 isulad hook 执行配置脚本

```
[root@localhost ~]# isula run -tid --hook-spec /etc/isulad-tools/hookspec.json
--system-container --external-rootfs /root/root-fs none init
eed1096c8c7a0eca6d92b1b3bc3dd59a2a2adf4ce44f18f5372408ced88f8350
```

- 添加一个块设备到容器

```
[root@localhost ~]# isulad-tools add-device ee /dev/sdb:/dev/sdb123
Add device (/dev/sdb) to container(ee,/dev/sdb123) done.
[root@localhost ~]# isula exec ee fdisk -l /dev/sdb123
Disk /dev/sdb123: 50 GiB, 53687091200 bytes, 104857600 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xda58a448

Device       Boot Start      End   Sectors Size Id Type
/dev/sdb123p1      2048 104857599 104855552  50G  5 Extended
/dev/sdb123p5      4096 104857599 104853504  50G 83 Linux
```

- 更新设备信息

```
[root@localhost ~]# isulad-tools update-device --device-read-bps /dev/sdb:10m
ee
Update read bps for device (/dev/sdb,10485760) done.
```

- 删除设备

```
[root@localhost ~]# isulad-tools remove-device ee /dev/sdb:/dev/sdb123
Remove device (/dev/sdb) from container(ee,/dev/sdb123) done.
Remove read bps for device (/dev/sdb) done.
```

2.3.12.2 网卡管理

功能描述

isulad-tools 支持将宿主机上的物理网卡或虚拟网卡插入到容器，在不使用网卡的时候从容器中删除归还给宿主机，并且可以动态修改网卡配置。插入物理网卡即把宿主机上一块网卡直接添加到容器中，插入虚拟网卡则需要先创建一对 veth pair，之后将一端插入到容器中。

命令格式

```
isulad-tools [COMMADN][OPTIONS] <container_id>
```

其中：

COMMAND：网卡管理相关的命令。

OPTIONS：网卡管理命令支持的选项。

container_id：容器 id。

参数说明

命令	功能说明	选项说明
add-nic	给容器创建一个网卡。	支持的选项如下： • --type：设置网卡类型，当前只支持 eth/veth。 • --name：设置网卡名称，格式为 [host:]<container>，host 不写是随机名字。 • --ip：设置网卡 IP 地址。 • --mac：设置网卡 mac 地址。 • --bridge：设置网卡绑定的网桥。 • --mtu：设置网卡的 mtu 值，默认 1500。 • --update-config-only：如果此 flag 设置了，只更新配置文件，不会实际做添加网卡的动作。 • --qlen：配置 qlen 值，默认为 1000。
remove-nic	从容器中将网卡删除，还原至宿主机。	支持的选项如下： • --type：设置网卡的类型。 • --name：设置网卡的名称，格式为 [host:]<container>。
list-nic	列出容器中所有的网卡。	支持的选项如下：

命令	功能说明	选项说明
		• --pretty：按照 json 格式输出。 • --filter：按照过滤格式输出，比如--filter '{"ip":"192.168.3.4/24", "Mtu":1500}'。
update-nic	更改容器内指定网卡的配置参数。	支持的选项如下： • --name：容器内网卡名（必须项）。 • --ip：设置网卡 IP 地址。 • --mac：设置网卡 mac 地址。 • --bridge：设置网卡绑定的网桥。 • --mtu：设置网卡的 mtu 值。 • --update-config-only：如果此 flag 设置了，只更新配置文件，不会实际做更新网卡的动作。 • --qlen：配置 qlen 值。

约束限制

- 支持添加物理网卡（eth）和虚拟网卡（veth）两种类型。
- 在添加网卡时可以同时对网卡进行配置，参数包括--ip/--mac/--bridge/--mtu/--qlen。
- 支持最多添加 8 个物理网卡到容器。
- 使用 isulad-tools add-nic 向容器添加 eth 网卡后，如果不加 hook，在容器退出前必须手工将 nic 删除，否则在 host 上的 eth 网卡的名字会被更改成容器内的名字。
- 对于物理网卡（1822 vf 网卡除外），add-nic 必须使用原 mac 地址，update-nic 禁止修改 mac 地址，容器内也不允许修改 mac 地址。
- 使用 isulad-tools add-nic 时，设置 mtu 值，设置范围跟具体的网卡型号有关。
- 使用 isulad-tools 向容器添加网卡和路由时，建议先执行 add-nic 添加网卡，然后执行 add-route 添加路由；使用 isulad-tools 从容器删除网卡和路由时，建议先执行 remove-route 删除路由，然后执行 remove-nic 删除网卡。
- 使用 isulad-tools 添加网卡时，一块网卡只能添加到一个容器中。

使用示例

- 启动一个系统容器，指定 hook spec 为 isulad hook 执行配置脚本：

```
[root@localhost ~]# isula run -tid --hook-spec /etc/isulad-tools/hookspec.json
--system-container --external-rootfs /root/root-fs none init
2aaca5c1af7c872798dac1a468528a2ccbaf20b39b73fc0201636936a3c32aa8
```

- 添加一个虚拟网卡到容器

```
[root@localhost ~]# isulad-tools add-nic --type "veth" --name abc2:bcd2 --ip
172.17.28.5/24 --mac 00:ff:48:13:xx:xx --bridge docker0 2aaca5c1af7c
Add network interface to container 2aaca5c1af7c (bcd2,abc2) done
```

- 添加一个物理网卡到容器

```
[root@localhost ~]# isulad-tools add-nic --type "eth" --name eth3:eth1 --ip
172.17.28.6/24  --mtu 1300  --qlen 2100 2aaca5c1af7c
Add network interface to container 2aaca5c1af7c (eth3,eth1) done
```

说明

添加虚拟网卡或物理网卡时，请确保网卡处于空闲状态，添加正在使用的网卡会导致系统网络断开。

2.3.12.3 路由管理

功能描述

isulad-tools 工具可以对系统容器进行动态添加/删除路由表。

命令格式

```
isulad-tools [COMMADN][OPTIONS] <container_id> [ARG...]
```

其中：

COMMAND：路由管理相关的命令。

OPTIONS：路由管理命令支持的选项。

container_id：容器 id。

ARG：命令对应的参数。

接口说明

命令	功能说明	选项说明	参数说明
add-route	将网络路由规则添加到容器中。	支持的选项如下： --update-config-only：添加此参数，只更新配置文件，不做实际的更新路由表的动作。	参数格式：[{rule1},{rule2}] rule 样例： '[{"dest":"default", "gw":"192.168.10.1"},{"dest":"192.168.0.0/16","dev":"eth0","src":"192.168.1.2"}]' • dest：目标网络，如果为空则是默认网关。 • src：路由源 IP。 • gw：路由网关。 • dev：网络设备。
remove-route	从容器中删除路由。	支持的选项如下： --update-config-only：设置此参数，只更新配置	参数格式：[{rule1},{rule2}] rule 样例：

命令	功能说明	选项说明	参数说明
		文件，不做实际从容器中删除路由的动作。	'[{"dest":"default", "gw":"192.168.10.1"},{"dest":"192.168.0.0/16","dev":"eth0","src":"192.168.1.2"}]' ● dest：目标网络，如果为空则是默认网关。 ● src：路由源 IP。 ● gw：路由网关。 ● dev：网络设备。
list-route	列出容器中所有的路由规则。	支持的选项如下： ● --pretty：按照 json 格式输出。 ● --filter：按照过滤格式输出，比如--filter '{"ip":"192.168.3.4/24", "Mtu":1500}'。	无

约束限制

- 使用 isulad-tools 向容器添加网卡和路由时，建议先执行 add-nic 添加网卡，然后执行 add-route 添加路由；使用 isulad-tools 从容器删除网卡和路由时，建议先执行 remove-route 删除路由，然后执行 remove-nic 删除网卡。
- 向容器内添加路由规则时，需确保所添加的路由规则与容器内现有的路由规则不会产生冲突。

使用示例

- 启动一个系统容器，指定 hook spec 为 isulad hook 执行配置脚本：

```
[root@localhost ~]# isula run -tid --hook-spec /etc/isulad-tools/hookspec.json
--system-container --external-rootfs /root/root-fs none init
0d2d68b45aa0c1b8eaf890c06ab2d008eb8c5d91e78b1f8fe4d37b86fd2c190b
```

- isulad-tools 向系统容器添加一块物理网卡：

```
[root@localhost ~]# isulad-tools add-nic --type "eth" --name enp4s0:eth123 --ip
172.17.28.6/24  --mtu 1300  --qlen 2100 0d2d68b45aa0
Add network interface (enp4s0) to container (0d2d68b45aa0,eth123) done
```

- isulad-tools 添加一条路由规则到系统容器，注意格式需按照'[{"dest":"default", "gw":"192.168.10.1"},{"dest":"192.168.0.0/16","dev":"eth0","src":"192.168.1.2"}]'来配置。如果 dest 为空会自动填成 default。

```
[root@localhost ~]# isulad-tools add-route 0d2d68b45aa0
'[{"dest":"172.17.28.0/32", "gw":"172.17.28.5","dev":"eth123"}]'
Add route to container 0d2d68b45aa0, route:
{dest:172.17.28.0/32,src:,gw:172.17.28.5,dev:eth123} done
```

- 查看容器内是否新增一条路由规则：

```
[root@localhost ~]# isula exec -it 0d2d68b45aa0 route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.17.28.0     172.17.28.5      255.255.255.255 UGH   0      0        0 eth123
172.17.28.0     0.0.0.0         255.255.255.0   U     0      0       0 eth123
```

2.3.12.4 挂卷管理

功能描述

普通容器仅支持在创建时指定--volume 参数将宿主机的目录/卷挂载到容器实现资源共享，但是无法在容器运行时将挂载到容器中的目录/卷卸载掉，也不支持将宿主机的目录/卷挂载到容器。系统容器可以通过 isulad-tools 工具实现动态将宿主机的目录/卷挂载到容器，以及将容器中的目录/卷进行卸载。

命令格式

```
isulad-tools [COMMADN][OPTIONS] <container_id> [ARG...]
```

其中：

COMMAND：路由管理相关的命令。

OPTIONS：路由管理命令支持的选项。

container_id：容器 id。

ARG：命令对应的参数。

接口说明

表2-1

命令	功能说明	选项说明	参数说明
add-path	将宿主机文件/目录添加到容器中。	无	参数格式为： hostpath:containerpath:permission [hostpath:containerpath:permission ...] 其中： hostdevice：卷在主机上的路径。 containerdevice：卷在容器中的路径。 permission：容器内对挂载路径的操作权限。
remove-path	将容器中的目录/文件删除，还原到宿	无	参数格式为： hostpath:containerpath

命令	功能说明	选项说明	参数说明
	主机中。		[hostpath:containerpath ...] 其中： hostdevice：卷在主机上的路径。 containerdevice：卷在容器中的路径。
list-path	列出容器中所有的path 目录。	支持的选项如下： --pretty：按照json 格式输出。	无

约束限制

- 挂载目录（add-path）的时候必须要指定绝对路径。
- 挂载目录（add-path）会在主机上生成/.sharedpath 挂载点。
- 最多可以向单个容器中添加 128 个 volume，超过 128 后无法添加成功。
- add-path 不能将主机目录覆盖容器中的根目录目录（/），否则会造成功能影响。

使用示例

- 启动一个系统容器，指定 hook spec 为 isulad hook 执行配置脚本：

```
[root@localhost ~]# isula run -tid --hook-spec /etc/isulad-tools/hookspec.json
--system-container --external-rootfs /root/root-fs none init
e45970a522d1ea0e9cfe382c2b868d92e7b6a55be1dd239947dda1ee55f3c7f7
```

- isulad-tools 将宿主机某个目录挂载到容器，实现资源共享：

```
[root@localhost ~]# isulad-tools add-path e45970a522d1
/home/test123:/home/test123
Add path (/home/test123) to container(e45970a522d1,/home/test123) done.
```

- 宿主机目录/home/test123 创建一个文件，然后在容器内查看文件是否可以访问：

```
[root@localhost ~]# echo "hello world" > /home/test123/helloworld
[root@localhost ~]# isula exec e45970a522d1 bash
[root@localhost /]# cat /home/test123/helloworld
hello world
```

- isulad-tools 将挂载目录从容器内删除：

```
[root@localhost ~]# isulad-tools remove-path e45970a522d1
/home/test123:/home/test123
Remove path (/home/test123) from container(e45970a522d1,/home/test123) done
[root@localhost ~]# isula exec e45970a522d1 bash
[root@localhost /]# ls /home/test123/helloworld
ls: cannot access '/home/test123/helloworld': No such file or directory
```

2.4 附录

2.4.1 命令行接口列表

此处仅列出系统容器与普通容器的差异命令，其他命令用户可以查阅 iSulad 容器引擎相关章节，或者执行 isula XXX --help 进行查询。

命令	参数	参数指定值说明
isula create/run	--external-rootfs	• 字符串变量。 • 宿主机某个绝对路径。 • 运行系统容器时，必须使用此参数指定特定虚拟机的 rootfs。
	--system-container	• 布尔变量。 • 指定某个容器是否属于系统容器，如果是系统容器场景，必须使能。
	--add-host	• 字符串变量。 • 格式为：<hostname>:<ip>，指定容器的 hosts 配置，可以指定多个参数。
	--dns，--dns-option，--dns-search	• 字符串变量。 • 可以指定多个，指定容器的 dns 配置。
	--ns-change-opt	• 字符串变量。 • 容器 namespace 化内核参数可修改选项，参数只能为 net 或 ipc，如果指定多个，用逗号隔开，例如--ns-change-opt=net,ipc。
	--oom-kill-disable	• 布尔变量。 • 表示是否打开 oom-kill-disable 功能。
	--shm-size	• 字符串变量。 • 设置/dev/shm 大小，默认 64MB。 支持单位 B(b)、K(k)、M(m)、G(g)、T(t)、P(p)。
	--sysctl	• 字符串变量。 • 指定容器内核参数值，格式为 key=value，可传入多个，sysctl 白名单如下： kernel.msgmax，kernel.msgmnb，kernel.msgmni，kernel.sem，kernel.shmall，kernel.shmmax，kernel.shmmni， kernel.shm_rmid_forced，kernel.pid_max，net.，fs.mqueue。 **说明** 容器内 kernel.pid_max 参数需要内核支持 pid_max namespace 化，否则会报错。

命令	参数	参数指定值说明
		容器内 sysctl 白名单参数值限制与物理机对应的内核参数限制保持一致（包括参数类型、参数取值范围等）。
	--env-target-file	• 字符串变量。 • 指定 env 持久化文件路径（路径必须为绝对路径，且文件必须在 rootfs 目录下），文件如果存在不能超过 10MB，如果--env 和文件里面的 env 出现冲突，--env 指定值生效。 • 绝对路径的根目录/为 rootfs 根目录,，即要指定文件路径为容器内/etc/environment，只用指定 env-target-file=/etc/environment，而不是 env-target-file=/path/of/root-fs/etc/environemt。
	--cgroup-parent	• 字符串变量。 • 指定容器的 cgroup 父目录，cgroup 根路径为 /sys/fs/cgroup/<controller>。
	--host-channel	• 字符串变量。 • 指定宿主机和容器共享内存空间（tmpfs），格式为： <host path>:<container path>:<rw/ro>:<size limit>
	--files-limit	• 字符串变量。 • 整数值，指定容器内文件句柄数最大值。
	--user-remap	• 字符串变量。 • 参数格式为：<uid>:<gid>:<offset>

3 安全容器

3.1 概述
3.2 安装部署
3.3 使用方法
3.4 附录

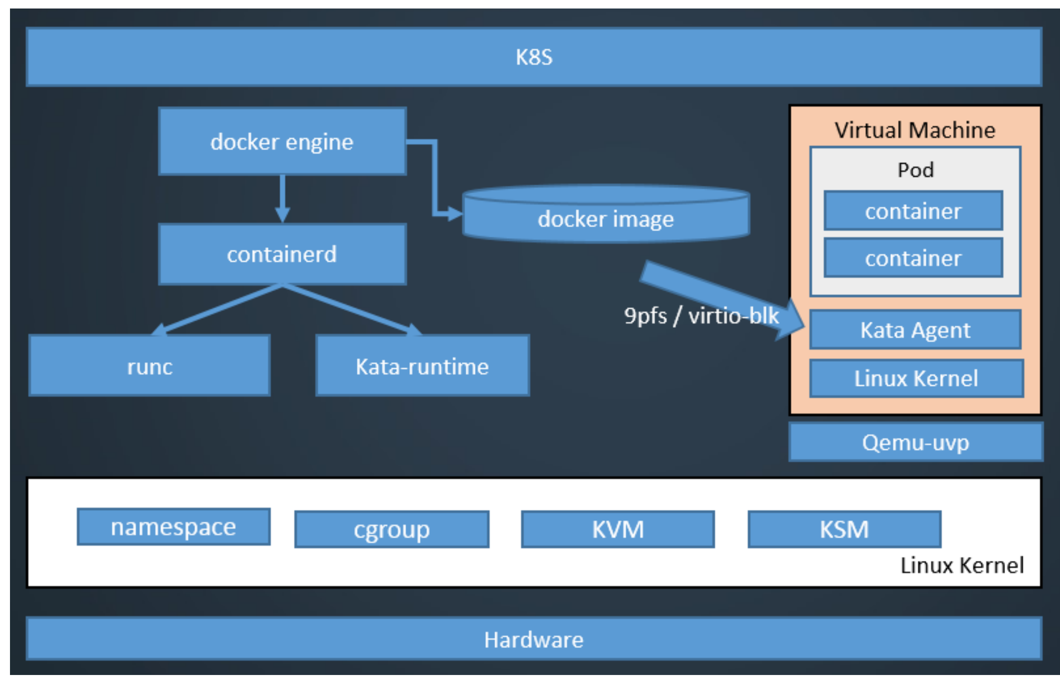
3.1 概述

安全容器是虚拟化技术和容器技术的有机结合，相比普通 linux 容器，安全容器具有更好的隔离性。

普通 linux 容器利用 namespace 进行进程间运行环境的隔离，并使用 cgroup 进行资源限制；因此普通 linux 容器本质上还是共用同一个内核，单个容器有意或无意影响到内核都会影响到整台宿主机上的容器。

安全容器是使用虚拟化层进行容器间的隔离，同一个主机上不同的容器间运行互相不受影响。

图3-1 安全容器架构



安全容器与 Kubernetes 中的 Pod 概念紧密联系，Kubernetes 为容器调度管理平台的开源生态标准，它定义了一组容器操作相关接口（Container Runtime Interface 简称 CRI）。

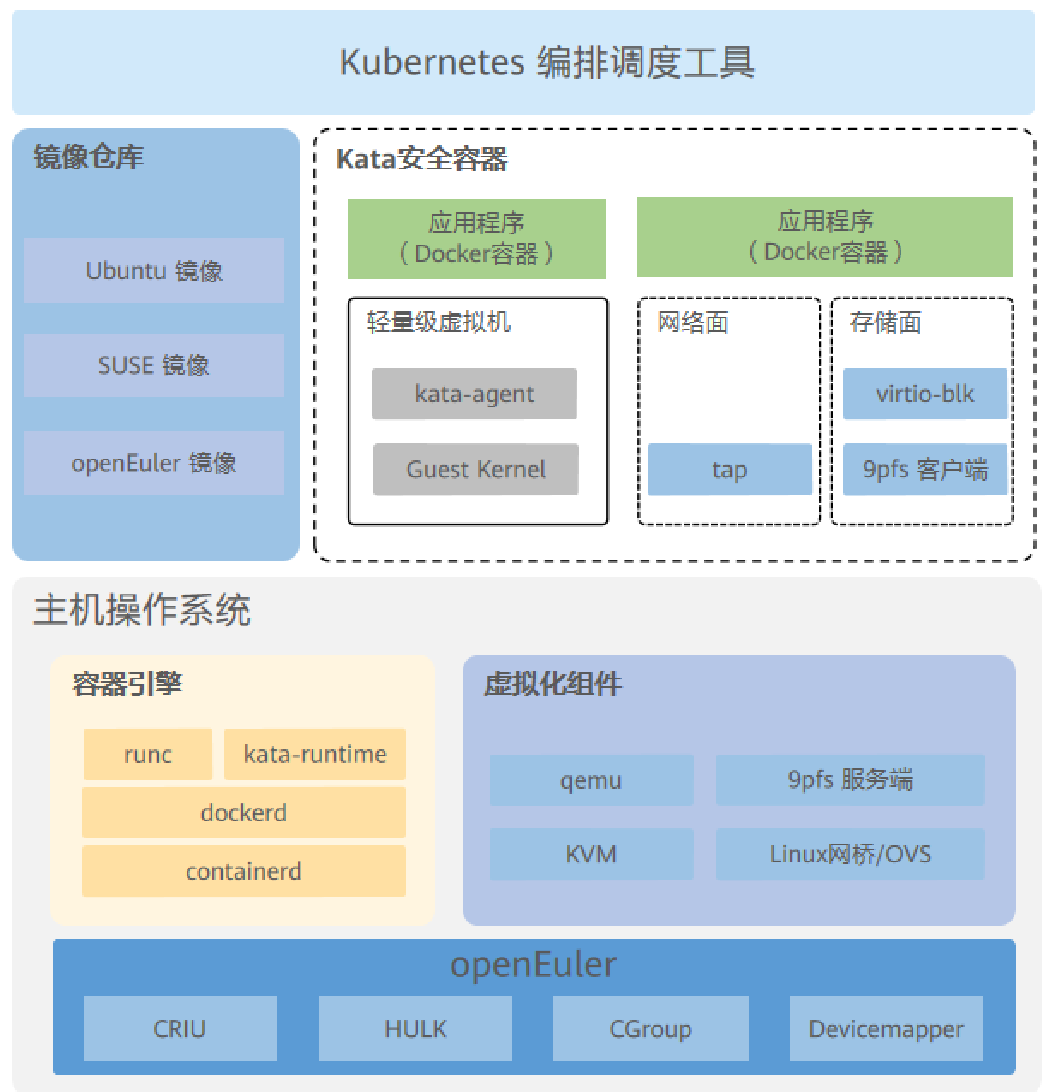
在 CRI 标准中，Pod 为完成一组服务需要的一组容器集合，是编排调度的最小单元，通常共享 IPC 和网络 namespace；一个 Pod 必然包含一个占位容器（pause 容器）以及一个或多个业务容器，其中 pause 容器与的生命周期相同。

其中安全容器中的一个轻量级虚拟机对应为一个 Pod，在此虚拟机中启动的第一个容器为 pause 容器，以后依次启动的容器为业务容器。

安全容器同时提供启动单个容器与启动 Pod 的功能。

安全容器与周边组件的关系如图 3-2 所示。

图3-2 安全容器与周边组件的关系



3.2 安装部署

3.2.1 安装方法

前提条件

- 为了获取更好的性能体验，安全容器需要运行在裸金属服务器上，**暂不支持安全容器运行在虚拟机内**。
- 安全容器运行依赖以下组件，请确保环境上已安装所需版本的依赖组件。以下组件来自配套的 openEuler 版本。如果使用 iSula 容器引擎，请参考 iSula 容器引擎的 1.2.1 安装方法章节安装 iSulad。
  - docker-engine

– qemu

安装操作

安全容器发布组件集成在同一个 kata-containers-<*version*>.rpm 包中，使用 rpm 命令可以直接安装对应的软件，其中 version 为。

```
rpm -ivh kata-containers-<version>.rpm
```

3.2.2 部署配置

3.2.2.1 docker-engine 容器引擎的配置

为了让 docker-engine 容器引擎支持新的容器运行时 kata-runtime，需要通过以下步骤对 docker-engine 容器引擎进行配置：

1. 请保证环境上所有的软件包（docker-engine、kata-containers）都已经安装完毕。
2. 停止 docker-engine。

   ```
   systemctl stop docker
   ```

3. 修改 docker-engine 的配置文件/etc/docker/daemon.json，并新增如下配置：

   ```
   {
     "runtimes": {
       "kata-runtime": {
         "path": "/usr/bin/kata-runtime",
         "runtimeArgs": [
             "--kata-config",
             "/usr/share/defaults/kata-containers/configuration.toml"
           ]
       }
     }
   }
   ```

4. 重新启动 docker-engine。

   ```
   systemctl start docker
   ```

3.2.2.2 iSula 容器引擎的配置

与 docker-engine 容器引擎类似，为了让 iSula 容器引擎支持新的容器运行时 kata-runtime，需要通过以下步骤对 iSula 容器引擎进行配置：

1. 请保证环境上所有的软件包（iSulad、kata-containers）都已经安装完毕。
2. 停止 isulad。

   ```
   systemctl stop isulad
   ```

3. 修改 iSula 容器引擎的配置文件/etc/isulad/daemon.json，并新增如下配置：

   ```
   {
     "runtimes": {
       "kata-runtime": {
         "path": "/usr/bin/kata-runtime",
         "runtime-args": [
             "--kata-config",
             "/usr/share/defaults/kata-containers/configuration.toml"
           ]
   ```

```
    }
  }
}
```

4. 重新启动 isulad。

```
systemctl start isulad
```

3.2.2.3 安全容器全局配置文件 configuration.toml

安全容器提供全局配置文件 configuration.toml 进行配置开关，用户也可以定制安全容器配置文件路径与配置选项。

在 docker-engine 的 runtimeArges 字段可以利用--kata-config 指定私有文件，默认的配置文件路径为/usr/share/defaults/kata-containers/configuration.toml。

常用配置文件字段如下，详细的配置文件选项参考 3.4.1 configuration.toml 配置说明。

1. hypervisor.qemu
   - path ：指定虚拟化 qemu 执行路径。
   - kernel ：指定 guest kernel 执行路径。
   - initrd ：指定 guest initrd 执行路径。
   - machin_type ：指定模拟芯片类型，其中 arm 为 virt，x86 架构为 pc。
   - kernel_params ：指定 guest 内核运行参数。
2. proxy.kata
   - path ：指定 kata-proxy 运行路径。
   - enable_debug ：kata-proxy 进程 debug 开关。
3. agent.kata
   - enable_blk_mount ：开启 block 设备 guest 挂载。
   - enable_debug ：kata-agent 进程 debug 开关。
4. runtime
   - enable_cpu_memory_hotplug：CPU 和内存热插拔开关。
   - enable_debug：kata-runtime 进程 debug 开关。

3.3 使用方法

本章介绍使用安全容器的方法。

3.3.1 管理安全容器的生命周期

3.3.1.1 启动安全容器

用户可以使用 docker-engine 或者 iSulad 作为安全容器的容器引擎，两者的调用方式类似，请用户自行选择一种方式启动安全容器。

启动安全容器的操作步骤如下：

1. 确保安全容器组件已经正确安装部署。

2. 准备容器镜像。假设容器镜像为 busybox，使用 docker-engine 和 iSula 容器引擎下载容器镜像的命令分别如下：

```
docker pull busybox
isula pull busybox
```

3. 启动一个安全容器。使用 docker-engine 和 iSula 容器引擎启动安全容器的命令分别如下：

```
docker run -tid --runtime kata-runtime --network none busybox <command>
isula run -tid --runtime kata-runtime --network none busybox <command>
```

说明

安全容器网络使用仅支持 CNI 网络，不支持 CNM 网络，不支持使用-p 和--expose 暴露容器端口，使用安全容器时需指定参数--net=none。

4. 启动一个 Pod
   a. 启动 pause 容器并根据回显获取 pod 的 sandbox-id。使用 docker-engine 和 iSula 容器引擎启动的命令分别如下：

```
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox <pause-image> <command>
isula run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.cri.container-type=sandbox <pause-image> <command>
```

   b. 创建业务容器并加入到这个 pod 中。使用 docker-engine 和 iSula 容器引擎创建的命令分别如下：

```
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=container --annotation
io.kubernetes.sandbox.id=<sandbox-id> busybox <command>
isula run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.cri.container-type=container --annotation
io.kubernetes.cri.sandbox-id=<sandbox-id> busybox <command>
```

   --annotation 用于容器类型的标注，这里的 docker-engine 和 isula 提供该字段，上游社区的开源 docker 引擎则不提供。

3.3.1.2 停止安全容器

- 停止一个安全容器。

```
docker stop <contaienr-id>
```

- 停止一个 Pod。

  Pod 停止需要注意顺序，pause 容器与 Pod 生命周期相同，因此先停止业务容器后再停止 pause 容器。

3.3.1.3 删除安全容器

删除前请确保容器已经停止：

```
docker rm <container-id>
```

如果需要强制删除一个正在运行的容器，可以使用**-f** 强制删除：

```
docker rm -f <container-id>
```

3.3.1.4 在容器中执行一条新的命令

由于 pause 容器仅作为占位容器，如果启动一个 Pod 时，请在业务容器内执行新的命令，pause 容器并没有相应的指令；如果只启动一个容器时，则可以直接执行新增命令：

```
docker exec -ti <container-id> <command>
```

说明

1. 如遇到 docker exec -ti 进入容器的同时，另一终端执行 docker restart 或者 docker stop 命令造成 exec 界面卡住的情况，可使用 Ctrl+P+Q 退出 docker exec 操作界面。
2. 如果使用-d 参数则命令在后台执行，不会打印错误信息，其退出码也不能作为命令执行是否正确的判断依据。

3.3.2 为安全容器配置资源

安全容器运行于虚拟化隔离的轻量级虚拟机内，因此资源的配置应分为两部分：对轻量级虚拟机的资源配置，即 Host 资源配置；对虚拟机内容器的配置，即 Guest 容器资源配置。以下资源配置均分为这两部分。

3.3.2.1 资源的共享

由于安全容器运行于虚拟化隔离的轻量虚拟机内，故无法访问 Host 上某些 namespace 下的资源，因此启动时不支持--net host，--ipc host，--pid host，--uts host。

当启动一个 Pod 时，同一个 Pod 中的所有容器默认共享同一个 net namespace 和 ipc namespace。如果同一个 Pod 中的容器需要共享 pid namespace，则可以通过 Kubernetes 进行配置，Kubernetes 1.11 版本该值为默认关闭。

3.3.2.2 限制 CPU 资源

1. 配置轻量级虚拟机 CPU 运行资源

   对轻量级虚拟机的 CPU 资源配置即虚拟机运行的 vcpu 配置，安全容器使用--annotation com.github.containers.virtcontainers.sandbox_cpu 配置轻量级虚拟机运行 CPU 资源，该参数仅可配置在 pause 容器上：

```
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --annotation
com.github.containers.virtcontainers.sandbox cpu=<cpu-nums> <pause-image>
<command>
```

   举例：

```
# 启动一个 pause 容器
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --annotation
com.github.containers.virtcontainers.sandbox_cpu=4 busybox sleep 999999
be3255a3f66a35508efe419bc52eccd3b000032b9d8c9c62df611d5bdc115954

# 进入容器查看 CPU 信息，查看 CPU 个数是否与
com.github.containers.virtcontainers.sandbox_cpu 配置的 CPU 个数相等
docker exec be32 lscpu
Architecture:       aarch64
Byte Order:         Little Endian
CPU(s):             4
```

```
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):          4
```

说明

CPU 个数可以设置的最大值为当前 OS 上可供运行的 CPU 值（除去隔离核），最小值为 0.5 个 CPU。

2. 配置容器 CPU 运行资源

配置容器 CPU 运行资源与开源 docker 容器配置 CPU 运行资源的方式相同，可以通过 docker run 命令中 CPU 资源限制相关的参数进行配置：

参数	含义
--cpu-shares	设置容器能使用的 CPU 时间比例。
--cpus	设置容器可以使用的 CPU 个数。
--cpu-period	设置容器进程的调度周期。
--cpu-quota	设置每个容器进程调度周期内能够使用的 CPU 时间。
--cpuset-cpus	设置容器进程可以使用的 CPU 列表。 **说明** 安全容器使用 --cpuset-cpus 参数绑定 CPU 时，CPU 的编号不能超过安全容器对应的轻量级虚机中 CPU 的个数减 1（轻量级虚机中 CPU 的编号从 0 开始）。
--cpuset-mems	设定该容器进程可以访问的内存节点。 **说明** 安全容器不支持多 NUMA 架构和配置，使用 NUMA memory 的--cpuset-mems 参数只能配置为 0。

3. 配置 CPU 热插拔功能

说明

安全容器 CPU 热插拔功能需要虚拟化组件 qemu 支持 CPU 热插拔。

kata-runtime 配置文件 config.toml 中 **enable_cpu_memory_hotplug** 选项负责开启和禁用 CPU 和内存热插拔。默认取值为 false，表示禁用 CPU 和内存热插拔功能；取值为 true，表示开启 CPU 和内存热插拔功能。

kata-runtime 中复用了**--cpus** 选项实现了 CPU 热插拔的功能，通过统计 Pod 中所有容器的**--cpus** 选项的和，然后确定需要热插多少个 CPU 到轻量级虚机中。

举例：

```
# 启动一个 pause 容器，轻量级虚机默认分配了 1 个 vcpu
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox busybox sleep 999999
77b40fb72f63b11dd3fcab2f6dabfc7768295fced042af8c7ad9c0286b17d24f
```

```
# 查看启动完 pause 容器后轻量级虚机中 CPU 个数
docker exec 77b40fb72f6 lscpu
Architecture:        x86 64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              1
On-line CPU(s) list:  0
Thread(s) per core:   1
Core(s) per socket:   1
Socket(s):           1

# 在同一个 Pod 中启动新的容器并通过--cpus 设置容器需要的 CPU 数量为 4
docker run -tid --runtime kata-runtime --network none --cpus 4 --annotation
io.kubernetes.docker.type=container --annotation
io.kubernetes.sandbox.id=77b40fb72f63b11dd3fcab2f6dabfc7768295fced042af8c7ad9c0
286b17d24f busybox sleep 999999
7234d666851d43cbdc41da356bf62488b89cd826361bb71d585a049b6cedafd3

# 查看当前轻量级虚机中 CPU 的个数
docker exec 7234d6668 lscpu
Architecture:        x86 64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list:  0-3
Thread(s) per core:   1
Core(s) per socket:   1
Socket(s):           4

# 删除热插了 CPU 的容器后，查看轻量级虚机中 CPU 的个数
docker rm -f 7234d666851d
7234d666851d

docker exec 77b40fb72f6  lscpu
Architecture:        x86 64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              1
On-line CPU(s) list:  0
Thread(s) per core:   1
Core(s) per socket:   1
Socket(s):           1
```

说明

由于 pause 容器只是一个占位容器没有工作负载，所以轻量级虚机启动时默认分配的 1 个 CPU 可以被其它容器共享，因此上面例子中启动的新容器只需要再热插 3 个 CPU 到轻量级虚机中即可。

– 当停止热插了 CPU 的容器后，启动容器时热插进去的 CPU 也会被拔出。

3.3.2.3 限制内存资源

1. 配置轻量级虚拟机 MEM 运行资源

   对轻量级虚拟机的 MEM 资源配置即虚拟机运行的内存进行配置，安全容器使用--annotation com.github.containers.virtcontainers.sandbox_mem 配置轻量级虚拟机运行 MEM 资源，该参数仅可配置在 pause 容器上：

```
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --annotation
com.github.containers.virtcontainers.sandbox mem=<memory-size> <pause-image>
<command>
```

   举例：

```
# 启动一个pause容器，通过--annotation
com.github.containers.virtcontainers.sandbox_mem=4G为轻量级虚机分配4G内存
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --annotation
com.github.containers.virtcontainers.sandbox mem=4G busybox sleep 999999
1532c3e59e7a45cd6b419aa1db07dd0069b0cdd93097f8944177a25e457e4297

# 查看轻量级虚机中内存信息，查看内存大小是否与
com.github.containers.virtcontainers.sandbox_mem配置的内存大小相等
docker exec 1532c3e free -m
              total        used        free      shared  buff/cache   available
Mem:           3950          20        3874          41          55        3858
Swap:             0           0           0
```

说明

- 如果没有通过--annotation com.github.containers.virtcontainers.sandbox_mem 显示地设置轻量级虚机的内存大小，则轻量级虚机默认使用的内存大小为 1GB。
- 安全容器一个 Pod 的最小内存规格是 1GB，支持的最大内存规格是 256GB。如果用户分配的内存规格超过 256GB，可能会出现未定义的错误，安全容器暂不支持超过 256GB 的大内存场景。

2. 配置容器 MEM 运行资源

   配置容器 MEM 运行资源与开源 docker 容器配置 MEM 运行资源的方式相同，可以通过 docker run 命令中 MEM 资源限制相关的参数进行配置：

参数	含义
-m/--memory	设置容器进程可以使用的内存大小。 说明 ● 当内存热插拔开关关闭时，-m 的取值要小于等于轻量级虚机启动时分配的内存大小。

3. 配置 MEM 热插功能

   同配置 CPU 热插拔功能一样，MEM 的热插功能也是由 kata-runtime 配置文件 config.toml 中 **enable_cpu_memory_hotplug** 选项配置，用法参见 3。

说明

内存资源当前只支持热插，不支持内存热拔。

kata-runtime 中复用了**-m** 选项实现了 MEM 热插的功能，通过统计 Pod 中所有容器的**-m** 选项的和，然后确定需要热插多少内存到轻量级虚机中，例如，

举例：

```
# 启动一个pause容器，轻量级虚机默认分配了1GB内存
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox busybox sleep 999999
99b78508ada3fa7dcbac457bb0f6e3784e64e7f7131809344c5496957931119f

# 查看启动完pause容器后轻量级虚机中的内存大小
docker exec 99b78508ada free -m
           total       used       free     shared  buff/cache   available
Mem:         983         18        914         36          50         908
Swap:          0          0          0

# 在同一个Pod中启动新的容器并通过-m设置容器需要的内存大小为4G
docker run -tid --runtime kata-runtime --network none -m 4G --annotation
io.kubernetes.docker.type=container --annotation
io.kubernetes.sandbox.id=99b78508ada3fa7dcbac457bb0f6e3784e64e7f7131809344c5496
957931119f busybox sleep 999999
c49461745a712b2ef3127fdf43b2cbb034b7614e6060b13db12b7a5ff3c830c8

# 查看当前轻量级虚机中内存的大小
docker exec c49461745 free -m
           total       used       free     shared  buff/cache   available
Mem:        4055         69       3928         36          57        3891
Swap:          0          0          0

# 删除热插了CPU的容器后，查看轻量级虚机中内存的大小
docker rm -f c49461745
c49461745

# 因为热插的内存暂不支持热拔功能，所以轻量级虚机中在删除热插内存容器之后还是拥有4GB的内存
docker exec 99b78508ada free -m
           total       used       free     shared  buff/cache   available
Mem:        4055         69       3934         36          52        3894
Swap:          0          0          0
```

说明

由于 pause 容器只是一个占位容器没有工作负载，所以轻量级虚机启动时分配的内存可以被其它容器共享使用，因此上面例子中启动的新容器只需要再热插 3GB 的内存到轻量级虚机中即可。

3.3.2.4 限制 Blkio 资源

1. 配置轻量级虚拟机 Blkio 运行资源

   对轻量级虚拟机的 BlkIio 资源配置，安全容器使用--annotation com.github.containers.virtcontainers.blkio_cgroup 配置轻量级虚拟机使用的块设备的 blkio 资源，该参数仅可配置在 pause 容器上：

```
docker run -tid --runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --annotation
com.github.containers.virtcontainers.blkio_cgroup=<blkio json格式字符串> <pause-
image> <command>
```

   其中--annotation com.github.containers.virtcontainers.blkio_cgroup 的取值要符合下面 BlkioCgroup 结构体的定义：

```
// BlkioCgroup for Linux cgroup 'blkio' data exchange
type BlkioCgroup struct {
    // Items specifies per cgroup values
    Items []BlockIOCgroupItem `json:"blkiocgroup,omitempty"`
}

type BlockIOCgroupItem struct {
    // Path represent path of blkio device
    Path string `json:"path,omitempty"`
    // Limits specifies the blkio type and value
    Limits []IOLimit `json:"limits,omitempty"`
}

type IOLimit struct {
    // Type specifies IO type
    Type string `json:"type,omitempty"`
    // Value specifies rate or weight value
    Value uint64 `json:"value,omitempty"`
}
```

IOLimit 结构体中 Type 字段取值列表为：

```
// BlkioThrottleReadBps is the key to fetch throttle read bps
BlkioThrottleReadBps = "throttle read bps"

// BlkioThrottleWriteBps is the key to fetch throttle write bps
BlkioThrottleWriteBps = "throttle write bps"

// BlkioThrottleReadIOPS is the key to fetch throttle read iops
BlkioThrottleReadIOPS = "throttle read iops"

// BlkioThrottleWriteIOPS is the key to fetch throttle write iops
BlkioThrottleWriteIOPS = "throttle write iops"

// BlkioWeight is the key to fetch blkio_weight
BlkioWeight = "blkio_weight"

// BlkioLeafWeight is the key to fetch blkio_leaf_weight
BlkioLeafWeight = "blkio_leaf_weight"
```

举例：

```
docker run -tid --runtime kata-runtime --network none --annotation
com.github.containers.virtcontainers.blkio cgroup='{"blkiocgroup":[{"path":"/de
v/sda","limits":[{"type":"throttle read bps","value":400},{"type":"throttle wri
te bps","value":400},{"type":"throttle read iops","value":700},{"type":"throttl
e write iops","value":699}]},{"limits":[{"type":"blkio weight","value":78}]}]}'
busybox sleep 999999
```

上面命令表示对启动的安全容器所使用的/dev/sda 磁盘进行 blkio 限流，分别将 throttle_read_bps 限速为 400bps，throttle_write_bps 限速为 400bps，throttle_read_iops 限速为 700 次/秒，throttle_write_iops 限速为 699 次/秒，以及所在 blkio cgroup 组的权重值设置为 78。

3.3.2.5 限制文件描述符资源

为了避免在容器中打开大量 9p 共享目录中的文件导致主机上文件描述符资源耗尽，使得安全容器无法正常提供服务，安全容器支持自定义配置安全容器 qemu 进程最多可以打开的文件描述符数量限制。

安全容器通过复用 docker run 命令中的**--files-limit** 选项来设置安全容器 qemu 进程最多可以打开文件描述符，该参数仅可配置在 pause 容器上，使用方法如下所示：

```
docker run -tid --runtime kata-runtime --network none --annotation
io.kubernetes.docker.type=podsandbox --files-limit <max-open-files> <pause-image>
bash
```

说明

- 如果**--files-limit** 选项的取值小于安全容器默认设置的最小值 1024 且不为 0 时，安全容器 qemu 进程最多可以打开的文件描述符数量会被设置为最小值 1024。
- 如果**--files-limit** 选项的取值为 0 时，安全容器 qemu 进程最多可以打开的文件描述符数量为系统可以打开文件描述符的最大值/proc/sys/fs/file-max 除以 400 后得到的默认值。
- 如果启动安全容器时没有显示指定**--files-limit** 可以打开的文件描述符的上限，安全容器 qemu 进程可以打开的文件描述符数量的上限和系统默认值保持一致。

3.3.3 为安全容器配置网络

tap 设备网络支持

安全容器技术是基于 Qemu VM 实现的，对于物理机系统来说，安全容器就相当于是一个 VM，所以安全容器可以在 Neutron 网络中将 VM 通过 TAP 技术接入外部网络。我们这里不需要关心 TAP 设备的创建和网桥对接等问题，只需要将指定的 TAP 设备（host 已经存在）热插进 pause 容器的 VM，并更新网卡信息即可。

相关命令行如下：

1. **为已经启动的容器添加一个 tap 网卡（interface）**

```
$ cat ./test-iface.json | kata-runtime kata-network add-iface 6ec7a98 -
```

其中：6ec7a98 是容器 ID 的截断,test-infs.json 是描述网卡信息的文件，举例如下：

```
{
    "device": "tap-test",
    "name": "eth-test",
    "IPAddresses": [
        {
            "address": "172.16.0.3",
            "mask": "16"
        }
    ],
    "hwAddr":"02:42:20:6f:a3:69",
    "mtu": 1500,
    "vhostUserSocket":"/usr/local/var/run/openvswitch/vhost-user1",
    "queues":5
}
```

上述 json 文件中各个字段的含义说明如下：

字段	是否可选	说明

字段	是否可选	说明
device	必选	设置网卡的主机端名字。支持字母、数字、下划线“_”、“-” 以及“.”字符，必须以字母开头，且长度不超过 15。需要确保同一个宿主机上 device 不能重复。
name	必选	设置网卡的容器内名称。支持字母、数字、下划线“_”、“-” 以及“.”字符，必须以字母开头，且长度不超过 15。需要确保同一个 Sandbox 内 name 不能重复。
IPAddresses	可选	设置网卡的 IP 地址。暂时支持一张网卡配置一个 IP，如果不配置 IP，则不会在容器内部配置 IP。
hwAddr	必选	设置网卡的 mac 地址值。
mtu	必选	设置网卡的 mtu 值。允许范围为[46, 9600]之间
vhostUserSocket	可选	设置 dpdk 轮询 socket 路径。路径最大长度 128 字节，命名规则支持数字、字母、“-”。必须以字母开头。
queues	可选	设置网卡多队列的队列数目。如果不配置，默认为 0。

kata-runtime kata-network add-iface 添加网卡命令执行返回结果说明：

- 命令执行成功：从**命令的标准输出**返回 json 格式插入网卡的信息，json 格式内容和传入的网卡信息相同。

  例如：

  ```
  $ kata-runtime kata-network add-iface <container-id> net.json
  {"device":"tap test","name":"eth-
  test","IPAddresses":[{"Family":2,"Address":"173.85.100.1","Mask":"24"}],"m
  tu":1500,"hwAddr":"02:42:20:6e:03:01","pciAddr":"01.0/00"}
  ```

- 命令执行失败：从**命令的标准输出**返回字符串 null。

  例如：

  ```
  $ kata-runtime kata-network add-iface <container-id> netbad.json
  2>/dev/null
  null
  ```

说明

当网卡添加成功时，如果为其指定了 IP 地址，则 kata 会为新添加的网卡添加一条 destination 为同网段地址的默认路由，如上例中添加网卡之后容器内有如下路由被添加：

```
[root@6ec7a98 /]# ip route
172.16.0.0/16 dev eth-test proto kernel scope link src 172.16.0.3
```

2. **列出已经添加的网卡**

```
$ kata-runtime kata-network list-ifaces 6ec7a98
[{"name":"eth-
test","mac":"02:42:20:6f:a3:69","ip":["172.16.0.3/16"],"mtu":1500}]
```

可以查询到我们刚才添加的网卡信息。

kata-runtime kata-network list-ifaces 列出已添加网卡命令执行返回结果说明：

- 命令执行成功：从**命令的标准输出**返回 json 格式的 Pod 中所有插入网卡的信息。

  如果 Pod 中插入了多个网卡设备，返回的是一个 json 数组格式的网卡信息

  ```
  $ kata-runtime kata-network list-ifaces <container-id>
  [{"name":"container eth","mac":"02:42:20:6e:a2:59","ip":["172.17.25.23/8"]
  ,"mtu":1500},{"name":"container eth 2","mac":"02:90:50:6b:a2:29","ip":["19
  2.168.0.34/24"],"mtu":1500}]
  ```

  如果 Pod 中没有插入任何网卡设备，从**命令的标准输出**返回字符串 null。

  ```
  $ kata-runtime kata-network list-ifaces <container-id>
  null
  ```

- 命令执行失败：从**命令的标准输出**返回字符串 null，从**命令的标准错误输出**返回错误描述信息。

  例如：

  ```
  $ kata-runtime kata-network list-ifaces <container-id>
  null
  ```

3. **为指定网卡添加一条路由**

```
$ cat ./test-route.json | kata-runtime kata-network add-route 6ec7a98 -
[{"dest":"default","gateway":"172.16.0.1","device":"eth-test"}]
```

kata-runtime kata-network add-route 为指定网卡添加一条路由命令执行返回结果说明：

- 命令执行成功：从**命令的标准输出**返回 json 格式的添加的路由信息。

  例如：

  ```
  $ kata-runtime kata-network add-route <container-id> route.json
  [{"dest":"177.17.0.0/24","gateway":"177.17.25.1","device":"netport test 1"
  }]
  ```

- 命令执行失败：从**命令的标准输出**返回字符串 null，从**命令的标准错误输出**返回错误描述信息。

  例如：

  ```
  $ kata-runtime kata-network add-route <container-id> routebad.json
  2>/dev/null
  null
  ```

字段说明如下：

- dest：设置路由对应的网段。格式为<ip>/<mask>，<ip>必选。分如下 3 种情况：
  i. 配置<ip>/<mask>；
  ii. 只配置<ip>，则默认掩码为 32；
  iii. 若配置了"dest":"default"，默认无 dest，需传入 gateway。
- gateway：设置路由的下一跳网关。设置"dest":"default"时，gateway 必选；其他情况可选。

– device：必选。设置路由对应的网卡名称，最长支持 15 字符。

说明

如果为容器内的回环设备 lo 添加路由时，路由配置文件中的"device"字段对应的设备名称为"lo"。

4. **删除指定路由**

```
$ cat ./test-route.json | kata-runtime kata-network del-route 6ec7a98 -
```

test-route.json 字段与添加路由输入 json 文件的字段相同。

kata-runtime kata-network del-route 删除指定路由命令执行返回结果说明：

– 命令执行成功：从**命令的标准输出**返回 json 格式的添加的路由信息。

例如：

```
$ kata-runtime kata-network del-route <container-id> route.json
[{"dest":"177.17.0.0/24","gateway":"177.17.25.1","device":"netport test 1"
}]
```

– 命令执行失败：从**命令的标准输出**返回字符串 null，从**命令的标准错误输出**返回错误描述信息。

例如：

```
$ kata-runtime kata-network del-route <container-id> routebad.json
2>/dev/null
null
```

说明

- 输入字段中 dest 为必选，device/gateway 均为可选。kata 根据不同字段进行模糊匹配，删除对应的路由规则。例如指定了 dest 为某个 IP，则所有该 IP 的规则都会被删除。
- 如果删除的是容器内回环设备 lo 的路由时，路由配置文件中的"device"字段对应的设备名称为"lo"。

5. **删除已经添加的网卡**

```
$ cat ./test-iface.json | kata-runtime kata-network del-iface 6ec7a98 -
```

说明

删除网卡时，仅根据网卡容器内名称（name 字段）来删除。即便填写其他字段，kata 也不会使用。

kata-runtime kata-network del-iface 删除网卡命令执行返回结果说明：

– 命令执行成功：从**命令标准输出**返回 null 字符串。

例如：

```
$ kata-runtime kata-network del-iface <container-id> net.json
null
```

– 命令执行失败：从**命令标准输出**返回删除失败网卡 json 格式的信息，从**命令的标准错误输出**返回错误描述信息。

例如：

```
$ kata-runtime kata-network del-iface <container-id> net.json
{"device":"tapname fun 012","name":"netport test 1","IPAddresses":[{"Famil
y":0,"Address":"177.17.0.1","Mask":"8"}],"mtu":1500,"hwAddr":"02:42:20:6e:
a2:59","linkType":"tap"}
```

以上为常用场景和命令行示例，具体命令行接口见附录 3.4.2 接口列表。

kata IPVS 子系统

安全容器提供添加 ipvs 命令的接口，支持对容器设置 ipvs 规则。功能包含对虚拟服务的添加/编辑/删除、对真实服务器的添加/编辑/删除、查询 ipvs 服务信息、设置连接超时、清理系统连接缓存，并支持对规则的批量导入。

1. 为容器添加一个虚拟服务地址

```
kata-runtime kata-ipvs ipvsadm --parameters "--add-service --tcp-service
172.17.0.7:80 --scheduler rr --persistent 3000" <container-id>
```

2. 修改容器虚拟服务参数

```
kata-runtime kata-ipvs ipvsadm --parameters "--edit-service --tcp-service
172.17.0.7:80 --scheduler rr --persistent 5000" <container-id>
```

3. 删除容器虚拟服务地址

```
kata-runtime kata-ipvs ipvsadm --parameters "--delete-service --tcp-service
172.17.0.7:80" <container-id>
```

4. 为虚拟服务地址添加一个真实服务器

```
kata-runtime kata-ipvs ipvsadm --parameters "--add-server --tcp-service
172.17.0.7:80 --real-server 172.17.0.4:80 --weight 100" <container-id>
```

5. 修改容器真实服务器参数

```
kata-runtime kata-ipvs ipvsadm --parameters "--edit-server --tcp-service
172.17.0.7:80 --real-server 172.17.0.4:80 --weight 200" <container-id>
```

6. 删除容器真实服务器

```
kata-runtime kata-ipvs ipvsadm --parameters "--delete-server --tcp-service
172.17.0.7:80 --real-server 172.17.0.4:80" <container-id>
```

7. 查询服务信息

```
kata-runtime kata-ipvs ipvsadm --parameters "--list" <container-id>
```

8. 逐条导入耗时太久，可将规则写入文件后，批量导入

```
kata-runtime kata-ipvs ipvsadm --restore - < <规则文件路径> <container-id>
```

说明

单条添加时默认使用 NAT 模式，批量导入时添加真实服务器需手动添加-m 参数使用 NAT 模式。

规则文件内容示例：

-A -t 10.10.11.12:100 -s rr -p 3000

-a -t 10.10.11.12:100 -r 172.16.0.1:80 -m

-a -t 10.10.11.12:100 -r 172.16.0.1:81 -m

-a -t 10.10.11.12:100 -r 172.16.0.1:82 -m

9. 清理系统连接缓存

```
kata-runtime kata-ipvs cleanup --parameters "--orig-dst 172.17.0.4 --protonum
tcp" <container-id>
```

10. 为 tcp/tcpfin/udp 连接设置超时

```
kata-runtime kata-ipvs ipvsadm --parameters "--set 100 100 200" <container-id>
```

说明

1. 每个容器支持 iptables 规则数量最大为 20000 条（5k service，3 个 server/service），add-service 和 add-server 都算作规则。
2. 批量导入前需清空已有规则。
3. 不存在并发测试场景。
4. 以上为常用命令示例，具体命令行接口见附录 3.4.2 接口列表。

3.3.4 监控安全容器

描述

kata events 命令用于显示指定容器状态。包括但不限于容器内存、CPU、Pid、 Blkio、大页内存、网络等信息。

用法

```
kata-runtime events [command options] <container-id>
```

参数

- -- interval value：设置查询周期。如果不使用该参数，默认查询周期为 5 秒。
- --stats： 显示容器信息并退出查询。

前置条件

要查询的容器状态必须为 running，否则报错：Container ID (<container_id>) does not exist。

该命令只支持查询监控一个容器的状态

示例

- 每隔三秒显示容器状态。

```
$ kata-runtime events --interval 3s 5779b2366f47
{
   "data": {
      "blkio": {},
      "cpu": {
         "throttling": {},
         "usage": {
            "kernel": 130000000,
            "percpu": [
               214098440
            ],
            "total": 214098440,
            "user": 10000000
         }
      },
      "hugetlb": {},
      "intel rdt": {},
      "interfaces": [
```

```
            {
                "name": "lo",
                "rx_bytes": 0,
                "rx_dropped": 0,
                "rx_errors": 0,
                "rx_packets": 0,
                "tx_bytes": 0,
                "tx_dropped": 0,
                "tx_errors": 0,
                "tx_packets": 0
            }
        ],
        "memory": {
            "cache": 827392,
            "kernel": {
                "failcnt": 0,
                "limit": 9223372036854771712,
                "max": 421888,
                "usage": 221184
            },
            "kernelTCP": {
                "failcnt": 0,
                "limit": 0
            },
            "raw": {
                "active_anon": 49152,
                "active_file": 40960,
                "cache": 827392,
                "dirty": 0,
                "hierarchical_memory_limit": 9223372036854771712,
                "hierarchical_memsw_limit": 9223372036854771712,
                "inactive_anon": 0,
                "inactive_file": 839680,
                "mapped_file": 540672,
                "pgfault": 6765,
                "pgmajfault": 0,
                "pgpgin": 12012,
                "pgpgout": 11803,
                "rss": 4096,
                "rss_huge": 0,
                "shmem": 32768,
                "swap": 0,
                "total_active_anon": 49152,
                "total_active_file": 40960,
                "total_cache": 827392,
                "total_dirty": 0,
                "total_inactive_anon": 0,
                "total_inactive_file": 839680,
                "total_mapped_file": 540672,
                "total_pgfault": 6765,
                "total_pgmajfault": 0,
                "total_pgpgin": 12012,
                "total_pgpgout": 11803,
                "total_rss": 4096,
                "total_rss_huge": 0,
```

```
                "total_shmem": 32768,
                "total_swap": 0,
                "total_unevictable": 0,
                "total_writeback": 0,
                "unevictable": 0,
                "writeback": 0
            },
            "swap": {
                "failcnt": 0,
                "limit": 9223372036854771712,
                "max": 34201600,
                "usage": 1204224
            },
            "usage": {
                "failcnt": 0,
                "limit": 9223372036854771712,
                "max": 34201600,
                "usage": 1204224
            }
        },
        "pids": {
            "current": 1
        },
        "tcp": {},
        "tcp6": {},
        "udp": {},
        "udp6": {}
    },
    "id": "5779b2366f47cd1468ebb1ba7c52cbdde3c7d3a5f2af3eefadc8356700fc860b",
    "type": "stats"
}
```

- 显示容器状态并立即返回

```
kata-runtime events --stats <container_id>
```

该命令返回内容的格式与上一条相同，区别为只显示一次信息后便退出。

3.4 附录

3.4.1 configuration.toml 配置说明

说明

configuration.toml 配置文件中各个字段的取值以 kata-containers-<version>.rpm 包中的 configuration.toml 文件为准，不支持用户对配置文件中的字段任意取值。

```
[hypervisor.qemu]
path : 指定虚拟化 qemu 执行路径
kernel : 指定 guest kernel 执行路径
initrd : 指定 guest initrd 执行路径
image : 指定 guest image 执行路径（不适用）
machine_type : 指定模拟芯片类型，ARM 架构为 virt，x86 架构为 pc
kernel_params : 指定 guest 内核运行参数
firmware : 指定固件路径，设空则使用默认固件
machine_accelerators : 指定加速器
```

```
default_vcpus : 指定每个 SB/VM 的默认 vCPU 数量
default_maxvcpus : 指定每个 SB/VM 的默认最大 vCPU 数量
default_root_ports : 指定每个 SB/VM 的默认 Root Ports 数量
default_bridges : 指定每个 SB/VM 的默认 bridges 数量
default_memory : 指定每个 SB/VM 的默认内存大小，默认为 1024 MiB
memory_slots : 指定每个 SB/VM 的内存插槽数量，默认为 10
memory_offset : 指定内存偏移量，默认为 0
disable_block_device_use : 禁止将块设备用于容器的 rootfs
shared_fs : 指定共享文件系统类型，默认为 virtio-9p
virtio_fs_daemon : 指定 vhost-user-fs 守护进程路径
virtio_fs_cache_size : 指定 DAX 缓存的默认大小
virtio_fs_cache : 指定缓存模式
block_device_driver : 指定块设备驱动
block_device_cache_set : 指定块设备是否设置缓存相关选项，默认 false
block_device_cache_direct : 指定是否使能 O_DIRECT，默认 false
block_device_cache_noflush : 指定是否忽略设备刷新请求，默认 false
enable_iothreads : 使能 iothreads
enable_mem_prealloc : 使能 VM RAM 预分配，默认 false
enable_hugepages : 使能大页，默认 false
enable_swap : 使能 swap，默认 false
enable_debug : 使能 qemu debug，默认 false
disable_nesting_checks : 关闭嵌套检查
msize_9p = 8192 : 指定每个 9p 包传输的字节数
use_vsock : 使用 vsocks 与 agent 直接通信（前提支持 vsocks），默认 false
hotplug_vfio_on_root_bus : 使能 vfio 设备在 root bus 热插拔，默认 false
disable_vhost_net : 关闭 vhost_net，默认 false
entropy_source : 指定默认熵源
guest_hook_path : 指定 guest hook 二进制路径

[factory]
enable_template : 使能 VM 模板，默认 false
template_path : 指定模板路径
vm_cache_number : 指定 VMCache 的缓存数量，默认 0
vm_cache_endpoint : 指定 VMCache 使用的 Unix socket 的地址，默认/var/run/kata-
containers/cache.sock

[proxy.kata]
path : 指定 kata-proxy 运行路径
enable_debug : 使能 proxy debug，默认 false

[shim.kata]
path : 指定 kata-shim 运行路径
enable_debug : 使能 shim debug，默认 false
enable_tracing : 使能 shim opentracing

[agent.kata]
enable_debug : 使能 agent debug，默认 false
enable_tracing : 使能 agent tracing
trace_mode : 指定 trace 模式
trace_type : 指定 trace 类型
enable_blk_mount : 开启 block 设备 guest 挂载

[netmon]
enable_netmon : 使能网络监控，默认 false
path : 指定 kata-netmon 运行路径
```

```
enable_debug : 使能 netmon debug，默认 false

[runtime]
enable_debug : 使能 runtime debug，默认 false
enable_cpu_memory_hotplug : 使能 cpu 和内存热插拔，默认 false
internetworking_model : 指定 VM 和容器网络互联模式
disable_guest_seccomp : 关闭在 guest 应用 seccemp 安全机制，默认 true
enable_tracing : 使能 runtime opentracing，默认 false
disable_new_netns : 不为 shim 和 hypervisor 进程创建网络命名空间，默认 false
experimental : 开启实验特性，不支持用户自定义配置
```

3.4.2 接口列表

表3-1 kata-runtime 网络相关的命令行接口

命令	子命令	文件示例	字段	含义	备注

kata-network
说明
• kata-network 命令需成组使用。不经过 kata-runtime kata-network 添加的网络设备，无法使用 kata-runtime kata-network 删除或者列出。反之亦然。
• kata-runtime kata-network 通过文件或 stdin 传入配置参数。
	add-iface
	说明
	• 一个 interface 只能添加到 1 个容器中。
	• 执行结果以返回值为准（非零返回值）。
		{
		"device":"tap1",
		"name":"eth1",
		"IPAddresses":[{"address":"172.17.1.10","mask":"24"}],
		"mtu":1300,
		"hwAddr":"02:42:20:6f:a2:80"
		"vhostUserSocket":"/usr/local/var/run/openvswitch/vhost-user1"
		}
			device	设置网卡的主机端名称	必选。支持字母、数字、下划线“_”、“-”以及“.”字符，必须以字母开头，且长度不超过 15。需要确保同一个宿主机上 device 不能重复。

			name	设置网卡的容器内名称	必选。支持字母、数字、下划线“_”、“-”以及“.”字符，必须以字母开头，且长度不超过 15。需要确保同一个 Sandbox 内 name 不能重复。
			IPAddresses	设置网卡的 IP 地址	可选。 暂时支持一张网卡配置一个 IP，如果不配置 IP，则不会在容器内部配置 IP。
			mtu	设置网卡的 mtu 值	必选。 有效范围 46~9600。

命令	子命令	文件示例	字段	含义	备注
			hwAddr	设置网卡的mac 值	必选。
			vhostUserSocket	设置 dpdk 轮循 socket 路径	可选。 路径最大长度128 字节，命名规则支持数字、字母、“-”。必须以字母开头。

	del-iface	{
		"name":"eth1"
		}
			无	删除容器内的一个网卡	**说明**
					删除网卡时，仅根据网卡容器内名称（name 字段）来删除。即便填写其他字段，kata 也不会使用。

	list-ifaces	无	无	查询容器内的网卡列表	无

	add-route	{
		"dest":"172.17.10.10/24",
		"gateway":"",
		"device":"eth1"
		}
			dest	设置路由对应的网段	格式为<ip>/<mask>，<ip>必选。
					分三种情况：
					1. 配置<ip>/<mask>；
					2. 只配置<ip>，则默认掩码为32；
					3. 配置"dest":"default"，默认无 dest，需传入 gateway。

			gateway	设置路由的下一跳网关	设置"dest":"default"时，gateway 必选；其他情况可选。
			device	设置路由对应的网卡名称	必选。 最长支持 15 字符。

命令	子命令	文件示例	字段	含义	备注

	del-route	{
		"dest":"172.17.10.10/24"
		}
			无	删除容器的路由规则	dest 为必选，device/gateway 均为可选。
					说明
					kata 根据不同字段进行模糊匹配，删除对应的路由规则。

	list-routes	无	无	查询容器内的路由列表	无

表3-2 kata-ipvs 命令行接口

<table>
<tr><th>命令</th><th>子命令</th><th>字段</th><th>参数</th><th>子参数</th><th>含义</th><th>备注</th></tr>
<tr><td rowspan="5">kata-ipvs</td><td rowspan="5">ipvsadm</td><td rowspan="5">--parameters</td><td rowspan="3">-A, --add-service</td><td>-t, --tcp-service
-u, --udp-service</td><td>虚拟服务类型</td><td>必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。
举例：
kata-runtime kata-ipvs ipvsadm --parameters "--add-service --tcp-service 172.17.0.7:80 --scheduler rr --persistent 3000" <container-id></td></tr>
<tr><td>-s, --scheduler</td><td>负载均衡调度算法</td><td>必选项。取值范围：rr|wrr|lc|wlc|lblc|lblcr|dh|sh|sed|nq。</td></tr>
<tr><td>-p, --persistent</td><td>持续服务时间</td><td>必选项。取值范围[1, 2678400]，单位 s。</td></tr>
<tr><td rowspan="2">-E, --edit-service</td><td>-t, --tcp-service
-u, --udp-service</td><td>虚拟服务类型</td><td>必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。</td></tr>
<tr><td>-s, --scheduler</td><td>负载均衡调度算法</td><td>必选项。取值范围：rr|wrr|lc/wlc/lblc/lblcr/dh/sh/sed/nq。</td></tr>
</table>

命令	子命令	字段	参数	子参数	含义	备注
				-p, --persistent	持续服务时间	必选项。取值范围[1, 2678400]，单位 s。
			-D, --delete-service	-t, --tcp-service -u, --udp-service	虚拟服务类型	必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。
			-a, --add-server	-t, --tcp-service -u, --udp-service	虚拟服务类型	必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。 举例： `kata-runtime kata-ipvs ipvsadm --parameters "--add-server --tcp-service 172.17.0.7:80 --real-server 172.17.0.4:80 --weight 100" <container-id>`
				-r, --real-server	真实服务器地址	必选项。格式为“ip:port”，port 取值[1,65535]。
				-w, --weight	权重	可选项，取值[0,65535]。
			-e, --edit-server	-t, --tcp-service -u, --udp-service	虚拟服务类型	必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。
				-r, --real-server	真实服务器地址	必选项。格式为“ip:port”，port 取值[1,65535]。
				-w, --weight	权重	可选项，取值[0,65535]。
			-d, --delete-server	-t, --tcp-service -u, --udp-service	虚拟服务类型	必选项。--tcp-service、--udp-service，两个参数只能选择其一。格式为“ip:port”，port 取值[1,65535]。

<table>
<tr><th>命令</th><th>子命令</th><th>字段</th><th>参数</th><th>子参数</th><th>含义</th><th>备注</th></tr>
<tr><td rowspan="8"></td><td rowspan="6"></td><td rowspan="5"></td><td></td><td>-r, --real-server</td><td>真实服务器地址</td><td>必选项。格式为“ip:port”，port 取值[1,65535]。</td></tr>
<tr><td>-L, --list</td><td>-t, --tcp-service
-u, --udp-service</td><td>指定查询虚拟服务信息</td><td>可选项。
举例：
<code>kata-runtime kata-ipvs ipvsadm --parameters "--list --tcp-service ip:port" <container-id></code></td></tr>
<tr><td rowspan="3">--set</td><td>--tcp</td><td>tcp 超时</td><td>必选项，取值[0, 1296000]。
举例：
<code>kata-runtime kata-ipvs ipvsadm --parameters "--set 100 100 200" <container-id></code></td></tr>
<tr><td>--tcpfin</td><td>tcpfin 超时</td><td>必选项，取值[0, 1296000]。</td></tr>
<tr><td>--udp</td><td>udp 超时</td><td>必选项，取值[0, 1296000]。</td></tr>
<tr><td>--restore</td><td colspan="2">-</td><td>标准输入批量导入
可指定规则文件</td><td>举例：
<code>kata-runtime kata-ipvs ipvsadm --restore - < <规则文件路径> <container-id></code>
说明
单条添加时默认使用 NAT 模式，批量导入时添加真实服务器需手动添加-m 参数使用 NAT 模式。
规则文件内容示例：
-A -t 10.10.11.12:100 -s rr -p 3000
-a -t 10.10.11.12:100 -r 172.16.0.1:80 -m
-a -t 10.10.11.12:100 -r 172.16.0.1:81 -m
-a -t 10.10.11.12:100 -r 172.16.0.1:82 -m</td></tr>
<tr><td rowspan="2">cleanup</td><td rowspan="2">--parameters</td><td colspan="2">-d, --orig-dst</td><td>ip 信息</td><td>必选项。
举例：
<code>kata-runtime kata-ipvs cleanup --parameters "--orig-dst 172.17.0.4 --protonum tcp" <container-id></code></td></tr>
<tr><td colspan="2">-p, --protonum</td><td>协议类型</td><td>必选项，取值为 tcp|udp 。</td></tr>
</table>

4 Docker 容器

4.1 概述

4.2 安装部署

4.3 容器管理

4.4 镜像管理

4.5 命令行参考

4.1 概述

Docker 是一个开源的 Linux 容器引擎项目， 用以实现应用的快速打包、部署和交付。Docker 的英文本意是码头工人，码头工人的工作就是将商品打包到 container(集装箱)并且搬运 container、装载 container。 对应到 Linux 中，Docker 就是将 app 打包到 container，通过 container 实现 app 在各种平台上的部署、运行。Docker 通过 Linux Container 技术将 app 变成一个标准化的、可移植的、自管理的组件，从而实现应用的“一次构建，到处运行”。Docker 技术特点就是：应用快速发布、部署简单、管理方便，应用密度更高。

4.2 安装部署

4.2.1 安装配置介绍及注意事项

本章节主要介绍和开源容器 docker 安装相关的重要配置。

4.2.1.1 注意事项

- docker-engine rpm 包与 containerd rpm 包、runc rpm 包、podman rpm 包不能同时安装。因为 docker-engine rpm 包中已经包含 Docker 运行所需的所有组件，其中包括 containerd、runc、docker 二进制，且 containerd、runc 和 podman rpm 包也分别提供了对应的二进制，所以重复安装时会出现软件包冲突。

4.2.1.2 基本安装配置

4.2.1.2.1 配置 daemon 参数

可以通过在/etc/docker/daemon.json 文件中添加配置项自定义配置参数，相关配置项以及如何使用可以通过 dockerd --help 查看。配置示例如下：

```
cat /etc/docker/daemon.json
{
   "debug": true,
   "storage-driver": "overlay2",
   "storage-opts": ["overlay2.override_kernel_check=true"]
}
```

4.2.1.2.2 daemon 运行目录配置

用户需要明白重新指定各种运行目录和文件（包括--graph、--exec-root 等），可能会存在目录冲突，或文件属性变换，对应用的正常使用造成影响。

须知

用户指定的目录或文件应为 docker 专用，避免冲突导致的文件属性变化带来安全问题。

- 以--graph 为例，当我们使用/new/path/作为 daemon 新的 Root Dir 时，如果/new/path/下已经存在文件，且目录或文件名与 docker 需要使用的目录或文件名冲突（例如： containers、hooks、tmp 等目录）时，docker 可能会更新原有目录或文件的属性，包括属主、权限等为自己的属主和权限。

须知

从 docker-17.05 开始，--graph 参数被标记为 Deprecated，用新的参数--data-root 替代。

4.2.1.2.3 daemon 自带网络配置

- Docker daemon 使用--bip 参数指定 docker0 网桥的网段之后，如果在下一次重启的时候去掉--bip 参数，docker0 网桥会沿用上一次的--bip 配置，即使重启之前已经删除 docker0 网桥。原因是 docker 会保存网络配置并在下一次重启的时候默认恢复上一次配置。
- Docker network create 并发创建网络的时候，可以创建具有相同名字的两个网络。原因是 docker network 是通过 id 来区分的，name 只是个便于识别的别名而已，不保证唯一性。
- Docker 在桥接 bridge 网络模式下，Docker 容器是通过宿主机上的 NAT 模式，建立与宿主机之外世界的通信。Docker Daemon 在启动一个容器时，每在宿主机上映射一个端口都会启动一个 docker-proxy 进程来实现访问代理。建议用户在使用这种 userland-proxy 时，只映射必须的端口，减少 docker-proxy 进行端口映射所消耗的资源。

4.2.1.2.4 daemon umask 配置

容器主进程和 exec 进程的默认 umask 为 0022，为了满足安全性需求，避免容器受到攻击，修改 runc 的实现，将默认 umask 修改为 0027。修改后 others 群组将无法访问新建文件或目录。

docker 启动容器时的默认 umask 值为 0027，可以在 dockerd 启动时，使用--exec-opt native.umask=normal 参数将容器启动时的 umask 修改为 0022。

须知

如果 docker create/run 也配置了 native.umask 参数，则以 docker create/run 中的配置为准。

详细的配置见 docker create 和 docker run 章节的参数说明。

4.2.1.2.5 daemon 启动时间

Docker 服务由 systemd 管理，systemd 对各个服务的启动时间有限制，如果指定时间内 docker 服务未能成功启动，则可能由以下原因导致：

- 如果使用 devicemapper 且为第一次启动，docker daemon 需要对该设备做文件系统初始化操作，而该操作会进行大量磁盘 IO 操作，在磁盘性能不佳或存在大量 IO 竞争时，很可能会导致 docker daemon 启动超时。devicemapper 设备只需要初始化一次，后续 docker daemon 启动时不再需要重复初始化。
- 如果当前系统资源占用太高，导致系统卡顿，系统所有的操作都会变慢，也可能会出现 docker 服务启动超时的情况。
- daemon 重启过程中，需要遍历并读取 docker 工作目录下每一个容器的配置文件、容器 init 层和可写层的配置，如果当前系统存在过多容器（包含 created 和 exited 的容器），并且磁盘读写性能受限，也会出现 daemon 遍历文件过久导致 docker 服务启动超时的情况。

出现服务启动超时情况，建议对以下两种情况进行排查调整：

- 容器编排层定期清理不需要的容器，尤其是 exited 的容器。
- 结合解决方案的性能要求场景，调整编排层的清理周期和 docker 服务的启动时间。

4.2.1.2.6 关联组件 journald

重启 systemd-journald 后需要重启 docker daemon。journald 通过 pipe 获取 docker daemon 的日志，如果 journald 服务重启，会导致该 pipe 被关闭，docker 的日志写入操作便会触发 SIGPIPE 信号，该错误信号会导致 docker daemon crash。由于忽略该信号影响严重，可能导致后续 docker daemon 的日志无法记录，因此建议用户在重启 journald 服务或者 journald 异常后主动去重启 docker daemon，保证 docker 日志能够被正常记录，避免 daemon crash 导致的状态异常。

4.2.1.2.7 关联组件 firewalld

需要在重启或拉起 firewalld 之后重启 docker 服务，保证 docker 服务在 firewalld 之后启动。

- firewalld 服务启动会清空当前系统的 iptables 规则，所以在启动 docker daemon 过程中，重启 firewalld 可能会导致 docker 服务插入 iptables 规则失败，从而导致 docker 服务启动失败。
- docker 服务启动后重启 firewalld 服务，或者状态发生了变化（从启动到停止，或者从停止到启动），会导致 docker 的 iptables 规则被删除，创建带端口映射的容器失败。

4.2.1.2.8 关联组件 iptables

docker 使用--icc=false 选项时，可以限制容器之间互通，但若 os 自带某些规则，可以造成限制容器之间互通失效，例如：

```
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
...
0     0 ACCEPT    icmp --  *     *      0.0.0.0/0          0.0.0.0/0
...
0     0 DROP      all  --  docker0 docker0  0.0.0.0/0          0.0.0.0/0
...
```

在 Chain FORWARD 中，DROP 上面多出了一条 ACCEPT icmp 的规则，造成加了--icc=false 后，容器之间也能 ping 通，但容器之间如果使用 udp/tcp 协议，对端仍然是不可达的。

因此，在容器 os 中使用 docker，如果需要使用--icc=false 选项时，建议先在 host 上清理一下 iptables 相关的规则。

4.2.1.2.9 关联组件 audit

docker 支持配置 audit，但不是强制的。例如：

```
-w /var/lib/docker -k docker
-w /etc/docker -k docker
-w /usr/lib/systemd/system/docker.service -k docker
-w /usr/lib/systemd/system/docker.socket -k docker
-w /etc/sysconfig/docker -k docker
-w /usr/bin/docker-containerd -k docker
-w /usr/bin/docker-runc -k docker
-w /etc/docker/daemon.json -k docker
```

配置 docker 的 audit，好处在于可以记录更多信息便于审计，但从安全角度来看，它对防攻击并没有实质性的作用。另一方面，audit 配置会导致严重的效率问题，可能导致系统卡顿，生产环境中请谨慎使用。

下面以“-w /var/lib/docker -k docker”为例，演示 docker audit 的配置：

```
[root@localhost signal]# cat /etc/audit/rules.d/audit.rules | grep docker -w
/var/lib/docker/  -k docker
[root@localhost signal]# auditctl -R /etc/audit/rules.d/audit.rules | grep docker
[root@localhost signal]# auditctl -l | grep docker -w /var/lib/docker/ -p rwxa -k
docker
```

说明

-p [r|w|x|a] 和-w 一起使用，观察用户对这个目录的读、写、执行或者属性变化（如时间戳变化）。这样的话，在/var/lib/docker 目录下的任何文件、目录操作，都会打印日志到 audit.log 中，从而会有太多的日志往 audit.log 中记录，会严重地影响 auditd， 比如内存、cpu 占用等，进而影响 os 的运行。例如：每次执行"ls /var/lib/docker/containers"都会有类似如下日志记录到/var/log/audit/audit.log 中。

```
type=SYSCALL msg=audit(1517656451.457:8097): arch=c000003e syscall=257 success=yes
exit=3 a0=ffffffffffffff9c a1=1b955b0 a2=90800 a3=0 items=1 ppid=17821 pid=1925
auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts6 ses=4
comm="ls" exe="/usr/bin/ls" subj=unconfined u:unconfined r:unconfined t:s0-
s0:c0.c1023 key="docker"type=CWD msg=audit(1517656451.457:8097):
cwd="/root"type=PATH msg=audit(1517656451.457:8097): item=0
name="/var/lib/docker/containers" inode=1049112 dev=fd:00 mode=040700 ouid=0 ogid=0
rdev=00:00 obj=unconfined_u:object_r:container_var_lib_t:s0 objtype=NORMAL
```

4.2.1.2.10 安全配置 seccomp

在做容器网络性能测试时发现，Docker 相对于原生内核 namespace 性能有所下降，经分析开启 seccomp 后，系统调用（如：sendto）不会通过 system_call_fastpath 进行，而是调用 tracesys，这会带来性能大幅下降。因此，建议在有高性能要求的业务的容器场景下关闭 seccomp，示例如下：

```
docker run -itd --security-opt seccomp=unconfined busybox:latest
```

4.2.1.2.11 禁止修改 docker daemon 的私有目录

不允许对 Docker 用的根目录（默认/var/lib/docker）和运行时目录（默认/run/docker）以及其文件作任何修改，包括在该目录下删除文件，添加文件，对目录或者文件做软/硬链接，修改文件的属性/权限，修改文件的内容等，如果确实需要做修改，后果自负。

4.2.1.2.12 普通用户大量部署容器场景下的配置注意事项

普通用户在 OS 主机上能创建的进程数的上限，例如：可以在系统中创建配置文件“/etc/security/limits.d/20-nproc.conf”限制；类似的，普通用户在容器里也能创建的进程数的上限，由容器镜像中“/etc/security/limits.d/20-nproc.conf”文件对应的值决定，如下所示：

```
cat /etc/security/limits.conf
*      soft   nproc   4096
```

当普通用户大量部署容器，导致容器内进程过多资源不够出现报错时，需要把容器镜像“/etc/security/limits.d/20-nproc.conf”文件中如上所示的 4096 配置值加大。

可配置的最大值请参考内核的最大能力，如下：

```
[root@localhost ~]# sysctl -a | grep pid_max
kernel.pid_max = 32768
```

4.2.1.3 存储驱动配置

本发行版 docker 支持 overlay2 和 devicemapper 两种存储驱动。由于 overlay2 较 devicemapper 而言，拥有更好的性能，建议用户在生产环境中优先考虑。

4.2.1.3.1 配置 overlay2 存储驱动

配置方法

docker 默认为使用 overlay2 存储驱动，也可以通过如下两种方式显示指定。

- 编辑/etc/docker/daemon.json，通过 storage-driver 字段显示指定。

```
cat /etc/docker/daemon.json
{
    "storage-driver": "overlay2"
}
```

- 编辑/etc/sysconfig/docker-storage，通过 docker deamon 启动参数显示指定。

```
cat /etc/sysconfig/docker-storage
DOCKER_STORAGE_OPTIONS="--storage-driver=overlay2"
```

注意事项

- 部分容器生命周期管理的操作会报找不到相应的 rootfs 或者相关的可执行文件。
- 如果容器的健康检查配置的是执行容器内的可执行文件，也会报错，导致容器的健康检查失败。
- 如果将 overlay2 作为 graphdriver，在容器中第一次修改镜像中的文件时，若该文件的大小大于系统剩余的空间，修改将会失败。因为即使修改很小，也要把这个文件完整的拷贝到上层，剩余空间不足导致失败。
- overlay2 文件系统相比普通文件系统天然存在一些行为差异，归纳如下：
  - 内核版本

    overlay2 只兼容原生 4.0 以上内核，建议配合使用 ext4 文件系统。
  - Copy-UP 性能问题

    修改 lower 层文件会触发文件复制到 upper 层，其中数据块复制和 fsync 比较耗时。
  - rename 目录问题
    - 只有源路径和目标路径都在 merged 层时，才允许 rename 系统调用，否则 rename 系统调用会报错-EXDEV。
    - 内核 4.10 引入了 redirect dir 特性来修复 rename 问题，对应内核选项为 CONFIG_OVERLAY_FS_REDIRECT_DIR。

      在使用 overlay2 场景下，对文件系统目录进行重命名时，如果系统配置文件/sys/module/overlay/parameters/redirect_dir 中配置的特性开关为关闭状态，则会导致使用失败；如果用户要使用相关特性，需要用户手动设置/sys/module/overlay/parameters/redirect_dir 为“Y”。
  - Hard link break 问题
    - 当 lower 层目录中有多个硬链接，在 merged 层写入数据会触发 Copy-UP，导致硬链接断开。
    - 内核 4.13 引入了 index feature 来修复这个问题，对应内核选项为 CONFIG_OVERLAY_FS_INDEX。注意这个选项没有前向兼容性，不支持热升级。
  - st_dev 和 st_ino 变化

触发 Copy-UP 之后，用户只能看到 merged 层中的新文件，inode 会变化。虽然 attr 和 xattr 可以复制，但 st_dev 和 st_ino 具有唯一性，不可复制。这会导致 stat 和 ls 查看 到相应的变化。

- fd 变化

  Copy-UP 之前，以只读模式打开文件得到描述符 fd1，Copy-UP 之后，打开同名文件得到文件描述符 fd2， 二者实际指向不同的文件。向 fd2 写入的数据不会在 fd1 中体现。

异常场景

容器使用配置了 overlay2 存储驱动的过程中，可能出现挂载点被覆盖的异常情况。例如

异常场景-挂载点被覆盖

挂载关系：在问题容器的挂载点的下面，存在一个/var/lib/docker/overlay2 的挂载点：

```
[root@localhost ~]# mount -l | grep overlay
overlay on
/var/lib/docker/overlay2/844fd3bca8e616572935808061f009d106a8748dfd29a0a4025645457f
a21785/merged type overlay
(rw,relatime,seclabel,lowerdir=/var/lib/docker/overlay2/l/JL5PZQLNDCIBU3ZOG3LPPDBHI
J:/var/lib/docker/overlay2/l/ELRPYU4JJG4FDPRLZJCZZE4UO6,upperdir=/var/lib/docker/ov
erlay2/844fd3bca8e616572935808061f009d106a8748dfd29a0a4025645457fa21785/diff,workdi
r=/var/lib/docker/overlay2/844fd3bca8e616572935808061f009d106a8748dfd29a0a402564545
7fa21785/work)
/dev/mapper/dm-root on /var/lib/docker/overlay2 type ext4
(rw,relatime,seclabel,data=ordered)
```

执行部分 docker 命令会遇到错误，比如：

```
[root@localhost ~]# docker rm 1348136d32
docker rm: Error response from daemon: driver "overlay2" failed to remove root
filesystem for 1348136d32: error while removing
/var/lib/docker/overlay2/844fd3bca8e616572935808061f009d106a8748dfd29a0a4025645457f
a21785: invalid argument
```

此时，在主机侧可以发现对应容器的 rootfs 找不到，但这并不意味着 rootfs 丢失，只是被/var/lib/docker/overlay2 挂载点覆盖，业务仍然可以正常运行，不受影响。修复方案可以参考如下：

- 修复方案一

  a. 确定当前 docker 所使用 graphdriver：

  ```
  docker info | grep "Storage Driver"
  ```

  b. 查询当前的挂载点：

  ```
  Devicemapper: mount -l | grep devicemapper
  Overlay2: mount -l | grep overlay2
  ```

  输出格式为： A on B type C (D)

■ A：块设备名称或 overlay
■ B：挂载点
■ C：文件系统类型
■ D：挂载属性

c. 从下往上逐一 umount 这些挂载点 B。
d. 然后全部 docker restart 这些容器，或者删除所有容器。
e. 重启 docker。

```
systemctl restart docker
```

- 修复方案二
  a. 业务迁移
  b. 节点重启

4.2.1.3.2 配置 devicemapper 存储驱动

用户如果需要使用 devicemapper 存储驱动，可以通过如下两种方式显示指定。

- 编辑/etc/docker/daemon.json，通过 storage-driver 字段显示指定。

```
cat /etc/docker/daemon.json
{
    "storage-driver": "devicemapper"
}
```

- 编辑/etc/sysconfig/docker-storage，通过 docker deamon 启动参数显示指定。

```
cat /etc/sysconfig/docker-storage
DOCKER_STORAGE_OPTIONS="--storage-driver=devicemapper"
```

注意事项

- 使用 devicemapper 必须使用 devicemapper+direct-lvm 的方式，配置的方法可以参考 https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production 。
- 配置 devicemapper 时，如果系统上没有足够的空间给 thinpool 做自动扩容，请禁止自动扩容功能。
- 禁止把/etc/lvm/profile/docker-thinpool.profile 中如下两个值都改成 100。

```
activation {
  thin_pool_autoextend_threshold=80
  thin_pool_autoextend_percent=20
}
```

- 使用 devicemapper 时推荐加上--storage-opt dm.use_deferred_deletion=true --storage-opt dm.use_deferred_removal=true。
- 使用 devicemapper 时，容器文件系统推荐使用 ext4，需要在 docker daemon 的配置参数中加 上--storage-opt dm.fs=ext4。
- 当 graphdriver 为 devicemapper 时，如果 metadata 文件损坏且不可恢复，需要人工介入恢复。禁止直接操作或篡改 daemon 存储 devicemapper 的元数据。
- 使用 devicemapper lvm 时，异常掉电导致的 devicemapper thinpool 损坏，无法保证 thinpool 损坏后可以修复，也不能保证数据的完整性，需重建 thinpool。

docker daemon 开启了 user namespace 特性，切换 devicemapper 存储池时的注意事项

- 一般启动容器时，deviceset-metadata 文件为：/var/lib/docker/devicemapper/metadata/deviceset-metadata。
- 使用了 user namespace 场景下，deviceset-metadata 文件使用的是：/var/lib/docker/{userNSUID.GID}/devicemapper/metadata/deviceset-metadata。
- 使用 devicemapper 存储驱动，容器在 user namespace 场景和普通场景之间切换时，需要将对应 deviceset-metadata 文件中的 BaseDeviceUUID 内容清空；针对 thinpool 扩容或者重建的场景下，也同样的需要将对应 deviceset-metadata 文件中的 BaseDeviceUUID 内容清空，否则 docker 服务会重启失败。

4.2.1.4 强制退出 docker 相关后台进程的影响

docker 的调用链很长，使 docker 相关后台进程强制退出（例如使用 kill -9）可能会导致一些数据状态不一致，本章节列举一些强制退出进程可能导致的问题。

4.2.1.4.1 信号量残留

使用 devicemapper 作为 graphdriver 时，强制退出进程可能导致信号量残留。docker 在操作 dm 的过程中会创建信号量，如果在释放信号量前，daemon 被强制退出，可能导致该信号量无法释放，一次强制退出最多泄露一个信号量，泄露概率低。而 linux 系统有信号量上限限制，当信号量泄露次数达到上线值时将无法创建新的信号量，进而导致 docker daemon 启动失败。排查方法如下：

1. 首先查看系统上残留的信号量

```
$ ipcs
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0d4d3358 238977024  root       600        1
0x0d4d0ec9 270172161  root       600        1
0x0d4dc02e 281640962  root       600        1
```

2. 接着用 dmsetup 查看 devicemapper 创建的信号量，该信号量集合是上一步中查看到的系统信号量的子集

```
$ dmsetup udevcookies
```

3. 最后查看内核信号量设置上限，第四个值就是当前系统的信号量使用上限

```
$ cat /proc/sys/kernel/sem
250     32000   32      128
```

   如果步骤 1 中残留的信号量数量与步骤 3 中看到的信号量上限相等，则是达到上限，此时 docker daemon 无法正常启动。可以使用下述命令增加信号量使用上限值来让 docker 恢复启动

```
$ echo 250 32000  32  1024 > /proc/sys/kernel/sem
```

   也可以手动清理 devicemapper 残留的信号量（下面是清理一分钟以前申请的 dm 相关信号量）

```
$ dmsetup udevcomplete all 1
This operation will destroy all semaphores older than 1 minutes with keys that
have a prefix 3405 (0xd4d).
```

```
Do you really want to continue? [y/n]: y
0 semaphores with keys prefixed by 3405 (0xd4d) destroyed. 0 skipped.
```

4.2.1.4.2 网卡残留

使用 bridge 模式启动容器的过程中，强制退出 daemon 可能导致网卡残留。使用 bridge 网络模式，当 docker 创建容器时，会先在 host 上创建一对 veth，然后再把该网卡信息存到数据库中，如果在创建完成，存到 docker 的数据库之前，daemon 被强制退出，那么该网卡无法被 docker 关联，下次启动也无法删除（docker 本身会清理自己数据库中不用的网卡），从而造成网卡残留。

4.2.1.4.3 重启容器失败

容器 hook 耗时较长，且启动阶段遇到 containerd 被强制退出，再次执行容器 start 操作可能失败。容器启动阶段遇到 containerd 被强制退出，docker start 操作直接返回错误；containerd 被重新拉起后，上次启动可能仍处于 runc create 执行阶段（执行用户自定义 hook，可能耗时较长），此时再次下发 docker start 命令启动该容器，可能提示以下错误：

```
Error response from daemon: oci runtime error: container with id exists: xxxxxx
```

该错误是由 runc create 一个已经存在（创建中）的容器导致，等第一次 start 对应的 runc 操作结束后再次执行 docker start 便可以成功。

由于 hook 的执行不受 docker 控制，这种场景下尝试回收该容器有可能导致 containerd 进程启动卡死（执行未知 hook 程序），且问题的风险可控（短期影响当前容器的创建）：

- 问题出现后等待第一次操作结束可以再次成功启动该容器。
- 一般是在容器启动失败后创建新的容器，不复用已经失败的容器。

综上，该问题暂时作为场景约束。

4.2.1.4.4 服务无法正常重启

短时间内频繁重启 docker 服务导致该服务无法正常重启。docker 系统服务由 systemd 负责监控，如果 docker 服务在 10s 内重启次数超过 5 次，systemd 服务就会监控到该异常行为，因此会禁止 docker 服务启动。只有等到下一个 10s 周期开始后，docker 服务才能响应重启命令正常重启。

4.2.1.5 系统掉电影响

主机意外掉电或系统 panic 等场景下，由于 docker daemon 的状态无法及时刷新到磁盘，导致重启后 docker daemon 状态不正常，可能出现的问题有（包括但不限于）：

- 掉电前创建的容器，重启后 docker ps -a 看不到，该问题是因为该容器的状态文件没有刷新到磁盘，从而导致重启后 daemon 无法获取到该容器的状态（镜像、卷、网络等也可能会有类似问题）。
- 掉电前某个文件正处于写入状态，尚未完全写入，重启后 daemon 重新加载该文件发现文件格式不正常或内容不完整，导致重启加载出错。
- 针对掉电时会破坏 docker DB 的情况，在重启节点时会清理 data-root 下面的 db 文件。因此重启前创建的如下信息在重启后会被删除：
  - network，用 docker network 创建的资源会在重启后清除。

- volume，用 docker volume 创建的资源会在重启后删除。
- 构建缓存，构建缓存信息会在重启后删除。
- containerd 保存的元数据，由于启动容器会重建 containerd 元数据，重启节点会清理 containerd 中保存的元数据。

说明

用户若选择采用手动清理恢复环境的方式，可通过配置环境变量“DISABLE_CRASH_FILES_DELETE=true”屏蔽 daemon 掉电重启时 db 文件清理功能。

4.3 容器管理

4.3.1 创建容器

下载镜像

运行 docker 命令需要 root 权限，当你使用普通用户登录时，需要用 sudo 权限执行 docker 命令。

```
[root@localhost ~]# docker pull busybox
```

该命令行将在 docker 官方的镜像库中下载 busybox:latest（命令行中没指定 TAG，所以使用默认的 TAG 名 latest），镜像在下载过程中将检测所依赖的层本地是否存在，如果存在就跳过。从私有镜像库下载镜像时，请带上 registry 描述，例如：假如建立了一个私有镜像库，地址为 192.168.1.110:5000，里面有一些常用镜像。使用下面命令行从私有镜像库中下载镜像。

```
[root@localhost ~]# docker pull 192.168.1.110:5000/busybox
```

从私有镜像库中下载下来的 image 名字带有镜像库地址的信息名字比较长，可以用 docker tag 命令生成一个名字简单点的 image。

```
[root@localhost ~]# docker tag 192.168.1.110:5000/busybox busybox
```

可以通过 docker images 命令查看本地镜像列表。

运行一个简单的应用

```
[root@localhost ~]# docker run busybox /bin/echo "Hello world"
Hello world
```

该命令行使用 busybox:latest（命令行中没有指定 tag，所以使用默认的 tag 名 latest）镜像创建了一个容器，在容器内执行了 echo "Hello world"。使用下面命令行可以查看刚才创建的这个容器。

```
[root@localhost ~]# docker ps -l
CONTAINER ID       IMAGE             COMMAND                  CREATED             STATUS
PORTS             NAMES
d8c0a3315bc0       busybox "/bin/echo 'Hello wo…"   5 seconds ago      Exited (0) 3
seconds ago                   practical_franklin
```

创建一个交互式的容器

```
[root@localhost ~]# docker run -it busybox /bin/bash
root@bf22919af2cf:/# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp
usr  var
root@bf22919af2cf:/# pwd
/
```

-ti 选项分配一个伪终端给容器并可以使用 STDIN 进行交互，可以看到这时可以在容器内执行一些命令。这时的容器看起来完全是一个独立的 linux 虚拟机。使用 exit 命令退出容器。

后台运行容器

执行下面命令行，-d 指示这个容器在后台运行，--name=container1 指定容器的名字为 container1。

```
[root@localhost ~]# docker run -d --name=container1 busybox /bin/sh -c "while
true;do echo hello world;sleep 1;done"
7804d3e16d69b41aac5f9bf20d5f263e2da081b1de50044105b1e3f536b6db1c
```

命令行的执行结果是返回了这个容器的 ID，没有返回命令的执行结果 hello world，此时容器在后台运行，可以用 docker ps 命令查看正在运行的容器:

```
[root@localhost ~]# docker ps
CONTAINER ID       IMAGE             COMMAND                CREATED           STATUS
PORTS              NAMES
7804d3e16d69       busybox "/bin/sh -c 'while tr"   11 seconds ago     Up 10 seconds
container1
```

用 docker logs 查看容器运行的输出：

```
[root@localhost ~]# docker logs container1
hello world
hello world
hello world
...
```

容器网络连接

默认情况下，容器可以访问外部网络，而外部网络访问容器时需要通过端口映射，下面以在 docker 中运行私有镜像库服务 registry 为例。下面的命令行中-P 使 registry 镜像中开放的端口暴露给主机。

```
[root@localhost ~]# docker run --name=container registry -d -P registry
cb883f6216c2b08a8c439b3957fb396c847a99079448ca741cc90724de4e4731
```

container_registry 这个容器已经启动了，但是并不知道容器中的服务映射到主机的哪个端口，通过 docker port 查看端口映射。

```
[root@localhost ~]# docker port container_registry
5000/tcp -> 0.0.0.0:49155
```

从输出可以看出，容器内的 5000 端口映射到了主机的 49155 端口。通过主机 IP:49155 就可以访问 registry 服务了，在浏览器中输入 http://localhost:49155 就可以返回 registry 的版本信息。

在运行 registry 镜像的时候还可以直接指定端口映射如：

```
docker run --name=container_registry -d -p 5000:5000 registry
```

通过-p 5000:5000 指定容器的 5000 端口映射到主机的 5000 端口。

注意事项

- **启动容器不能单独加-a stdin**

  启动容器时，不能单独加-a stdin，必须要同时加上-a stdout 或者-a stderr，否则会导致终端即使在容器退出后也会卡住。

- **避免使用已有容器的长 id、短 id 作为新容器的 name**

  创建容器时，避免使用已有容器 A 的长 id 或短 id 作为新容器 B 的 name。若使用容器 A 的长 id 作为容器 B 的 name，当使用容器 B 的 name 作为指定容器进行操作时，docker 匹配到的是容器 A。若使用容器 A 的短 id 作为容器 B 的 name，当使用容器 A 的短 id 作为指定容器进行相关操作时，docker 匹配到的是容器 B。这是因为，docker 在匹配容器时，先精确匹配所有容器的长 id。若未匹配成功，再根据 container_name 进行精确匹配；若还未匹配成功，直接对容器 id 进行模糊匹配。

- **使用 sh/bash 等依赖标准输入输出的容器应该使用`-ti`参数，避免出现异常**

  正常情况：不用`-ti`参数启动 sh/bash 等进程容器，容器会马上退出。

  出现这种问题的原因在于，docker 会先创建一个匹配用于容器内业务的 stdin，在不设置-ti 等交互式参数时，docker 会在容器启动后关闭该 pipe，而业务容器进程 sh/bash 在检测到 stdin 被关闭后会直接退出。

  异常情况：如果在上述过程中的特定阶段（关闭该 pipe 之前）强制杀死 docker daemon，会导致该 pipe 的 daemon 端没有被及时关闭，这样即使不带`-ti`的 sh/bash 进程也不会退出，导致异常场景，这种容器就需要手动清理。

  Daemon 重启后会接管原有的容器 stream，而不带`-ti`参数的容器可能就无法处理（因为正常情况下这些容器不存在 stream 需要接管）；真实业务下几乎不存在这种使用方式(不带 `-ti`的 sh/bash 没有任何作用)，为了避免这类问题发生，限制交互类容器应该使用 `-ti`参数。

- **容器存储卷**

  启动容器时如果通过`-v`参数将主机上的文件挂载到容器中，在主机或容器中使用 vi 或 sed 命令修改文件可能会使文件 inode 发生改变，从而导致主机和容器内的文件不同步。容器中挂载文件时应该尽量避免使用这种文件挂载的方式（或不与 vi 和 sed 同时使用），也可以通过挂载文件上层目录来避免该问题。在 docker 挂载卷时“nocopy”选项可以避免将容器内挂载点目录下原有的文件拷贝到主机源目录下，但是这个选项只能在挂载匿名卷时使用，不能在 bind mount 的场景下使用。

- **避免使用可能会对 host 造成影响的选项**

  --privileged 选项会让容器获得所有权限，容器可以做挂载操作和修改/proc、/sys 等目录，可能会对 host 造成影响，普通容器需要避免使用该选项。

  共享 host 的 namespace，比如--pid host/--ipc host/--net host 等选项可以让容器跟 host 共享命名空间，同样会导致容器影响 host 的结果，需要避免使用。

- **kernel memory cgroup 不稳定，禁止使用**

  kernel memory cgroup 在小于 4.0 版本的 Linux 内核上仍属于实验阶段，运行起来不稳定，虽然 Docker 的 Warning 说是小于 4.0 就可以，但是我们评估认为，kmemcg 在高版本内核仍然不稳定，所以不管是低版本还是高版本，均禁止使用。

  当 docker run --kernel-memory 时，会产生如下告警：

```
WARNING: You specified a kernel memory limit on a kernel older than 4.0. Kernel
memory limits are experimental on older kernels, it won't work as expected as
expected and can cause your system to be unstable.
```

- **blkio-weight 参数在支持 blkio 精确控制的内核下不可用**

  --blkio-weight-device 可以实现容器内更为精确的 blkio 控制，该控制需要指定磁盘设备，可以通过 docker --blkio-weight-device 参数实现。同时在这种内核下 docker 不再提供--blkio-weight 方式限制容器 blkio，使用该参数创建容器将会报错:

```
docker: Error response from daemon: oci runtime error: container_linux.go:247:
starting container process caused "process_linux.go:398: container init caused
\"process_linux.go:369: setting cgroup config for ready process caused
\\\"blkio.weight not supported, use weight_device instead\\\"\""
```

- **使用--blkio-weight-device 需要磁盘支持 CFQ 调度策略**

  --blkio-weight-device 参数需要磁盘工作于完全公平队列调度（CFQ：Completely Fair Queuing）的策略时才能工作。

  通过查看磁盘 scheduler 文件（/sys/block/<磁盘>/queue/scheduler）可以获知磁盘支持的策略以及当前所采用的策略，如查看 sda：

```
# cat /sys/block/sda/queue/scheduler noop [deadline] cfq
```

  当前 sda 支持三种调度策略：noop, deadline, cfq，并且正在使用 deadline 策略。通过 echo 修改策略为 cfq：

```
# echo cfq > /sys/block/sda/queue/scheduler
```

- **容器基础镜像中 systemd 使用限制**

  通过基础镜像创建的容器在使用过程中，容器基础镜像中的 systemd 仅用于系统容器，普通容器不支持使用。

并发性能

- docker 内部的消息缓冲有一个上限，超过这个上限就会将消息丢弃，因此在并发执行命令时建议不要超过 1000 条命令，否则有可能会造成 docker 内部消息丢失，从而造成容器无法启动等严重问题。
- 并发创建容器并对容器执行 restart 时会偶现“oci runtime error: container init still running”报错，这是因为 containerd 对事件等待队列进行了性能优化，容器 stop 过程中执行 runc delete，尝试在 1s 内 kill 掉容器的 init 进程，如果 1s 内 init 进程还没有被 kill 掉的话 runc 会返回该错误。由于 containerd 的 GC（垃圾回收机制）每隔 10s 会回收之前 runc delete 的残留资源， 所以并不影响下次对容器的操作，一般出现上述报错的话等待 4~5s 之后再次启动容器即可。

安全特性解读

1. docker 默认的权能配置分析

   原生的 docker 默认配置如下，默认进程携带的 Cap 如下:

```
"CAP_CHOWN",
"CAP_DAC_OVERRIDE",
"CAP_FSETID",
"CAP_FOWNER",
"CAP_MKNOD",
"CAP_NET_RAW",
"CAP_SETGID",
"CAP_SETUID",
"CAP_SETFCAP",
"CAP_SETPCAP",
"CAP_NET_BIND_SERVICE",
"CAP_SYS_CHROOT",
"CAP_KILL",
"CAP_AUDIT_WRITE",
```

默认的 seccomp 配置是白名单，不在白名单的 syscall 默认会返回 SCMP_ACT_ERRNO，根据给 docker 不同的 Cap 开放不同的系统调用，不在上面的权限，默认 docker 都不会给到容器。

2. CAP_SYS_MODULE

   CAP_SYS_MODULE 这个 Cap 是让容器可以插入 ko，增加该 Cap 可以让容器逃逸，甚至破坏内核。因为容器最大的隔离是 Namespace，在 ko 中只要把他的 Namespace 指向 init_nsproxy 即可。

3. CAP_SYS_ADMIN

   sys_admin 权限给容器带来的能力有：

   - 文件系统（mount，umount，quotactl）
   - namespace 设置相关的（setns，unshare，clone new namespace）
   - driver ioctl
   - 对 pci 的控制，pciconfig_read, pciconfig_write, pciconfig_iobase
   - sethostname

4. CAP_NET_ADMIN

   容器中有访问网络接口的和 sniff 网络流量的权限，容器可以获取到所有容器包括 host 的网络流量，对网络隔离破坏极大。

5. CAP_DAC_READ_SEARCH

   该权限开放了，两个系统调用 open_by_handle_at，name_to_handle_at，如果 host 上没有 selinux 保护，容器中可通过暴力搜索 file_handle 结构的 inode 号，进而可以打开 host 上的任意文件，影响文件系统的隔离性。

6. CAP_SYS_RAWIO

   容器中可对 host 写入 io 端口，可造成 host 内核崩溃。

7. CAP_SYS_PTRACE

   容器中有 ptrace 权限，可对容器的进程进行 ptrace 调试。现 runc 已经修补该漏洞，但有些工具比如 nsenter 和 docker-enter 并没有改保护，容器中可对这些工具执行的进程进行调试，获取这些工具带入的资源信息（Namespace、fd 等），另外， ptrace 可以绕过 seccomp，极大增加内核攻击面。

8. Docker Cap 接口 --cap-add all

   --cap-add all 表示赋予容器所有的权能，包括本节提到的比较危险的权能，使得容器可以逃逸。

9. 不要禁用 docker 的 seccomp 特性

   默认的 docker 有一个 seccomp 的配置，配置中使用的是白名单，不在配置的 sys_call 会被 seccomp 禁掉，使用接口--security-opt 'seccomp:unconfined'可以禁止使用 seccomp 特性。如果禁用 seccomp 或使用自定义 seccomp 配置但过滤名单不全，都会增加容器对内核的攻击面。

10. 不要配置/sys 和/proc 目录可写

    /sys 和/proc 目录包含了 linux 维护内核参数、设备管理的接口，容器中配置该目录可写可能会导致容器逃逸。

11. Docker 开放 Cap --CAP_AUDIT_CONTROL

    容器可以通过控制系统 audit 系统，并且通过 AUDIT_TTY_GET/AUDIT_TTY_SET 等命令可以获取审计系统中记录的 tty 执行输入记录，包括 root 密码。

12. CAP_BLOCK_SUSPEND 和 CAP_WAKE_ALARM

    容器可拥有阻塞系统挂起(epoll)的能力。

13. CAP_IPC_LOCK

    容器拥有该权限后，可以突破 ulimit 中的 max locked memory 限制，任意 mlock 超大内存块，造成一定意义的 DoS 攻击。

14. CAP_SYS_LOG

    容器拥有该权限后，可以 dmesg 读取系统内核日志，突破内核 kaslr 防护。

15. CAP_SYS_NICE

    容器拥有该权限后，可以改变进程的调度策略和优先级，造成一定意义的 DoS 攻击。

16. CAP_SYS_RESOURCE

    容器可以绕过对其的一些资源限制，比如磁盘空间资源限制、keymaps 数量限制、pipe-size-max 限制等，造成一定意义的 DoS 攻击。

17. CAP_SYS_TIME

    容器可以改变 host 上的时间。

18. Docker 默认 Cap 风险分析

    Docker 默认的 Cap，包含了 CAP_SETUID 和 CAP_FSETID，如 host 和容器共享目录，容器可对共享目录的二进制文件进行+s 设置，host 上的普通用户可使用其进行提权 CAP_AUDIT_WRITE，容器可以对 host 写入，容器可以对 host 写入日志，host 需配置日志防爆措施。

19. Docker 和 host 共享 namespace 参数，比如--pid，--ipc, --uts

    该参数为容器和 host 共享 namespace 空间，容器和 host 的 namespace 隔离没有了，容器可对 host 进行攻击。比如，使用--pid 和 host 共享 pid namespace，容器中可以看到 host 上的进程 pid 号，可以随意杀死 host 的进程。

20. --device 把 host 的敏感目录或者设备，映射到容器中

    Docker 管理面有接口可以把 host 上的目录或者设备映射到容器中，比如--device，-v 等参数，不要把 host 上的敏感目录或者设备映射到容器中。

4.3.2 创建容器使用 hook-spec

原理及使用场景

docker 支持 hook 的扩展特性，hook 应用与底层 runc 的执行过程中，遵循 OCI 标准：https://github.com/opencontainers/runtime-spec/blob/master/config.md#hooks 。

hook 主要有三种类型：prestart，poststart，poststop。分别作用于容器内用户应用程序启动之前，容器应用程序启动之后，容器应用程序停止之后。

接口参考

当前为 docker run 和 create 命令增加了参数“--hook-spec”，后面接 spec 文件的绝对路径，可以指定容器启动时的需要添加的 hook，这些 hook 会自动附加在 docker 自己动态创建的 hook 后面（当前 docker 只有一个 libnetwork 的 prestart hook），随容器的启动/销毁过程执行用户指定的程序。

spec 的结构体定义为：

```
// Hook specifies a command that is run at a particular event in the lifecycle of a
container
type Hook struct{
            Path    string   `json:"path"`
            Args    []string `json:"args,omitempty"`
            Env     []string `json:"env,omitempty"`
            Timeout *int     `json:"timeout,omitempty"`
}
// Hooks for container setup and teardown
type  Hooks struct{
            // Prestart is a list of hooks to be run before the container process
is executed.
            // On Linux, they are run after the container namespaces are created.
            Prestart []Hook `json:"prestart,omitempty"`
            // Poststart is a list of hooks to be run after the container process
is started.
            Poststart []Hook `json:"poststart,omitempty"`
            // Poststop is a list of hooks to be run after the container process
exits.
            Poststop []Hook `json:"poststop,omitempty"`
}
```

- Spec 文件的 path、args、env 都是必填信息；
- Timeout 选填(建议配置)，参数类型为 int，不接受浮点数，范围为[1, 120]。
- Spec 内容应该是 json 格式的，格式不对会报错，示例参考前面。
- 使用的时候既可以`docker run --hook-spec /tmp/hookspec.json xxx`，也可以 `docker create --hook-spec /tmp/hookspec.json xxx && docker start xxx`。

为容器定制特有的 hook

以启动过程中添加一个网卡的过程来说明。下面是相应的 hook spec 文件内容：

```
{
    "prestart": [
```

```
        {
            "path": "/var/lib/docker/hooks/network-hook",
            "args": ["network-hook", "tap0", "myTap"],
            "env": [],
            "timeout": 5
        }
    ],
    "poststart":[],
    "poststop":[]
}
```

指定 prestart hook 增加一个网络 hook 的执行。路径是/var/lib/docker/hooks/network-hook，args 代表程序的参数，第一个参数一般是程序名字，第二个是程序接受的参数。对于 network-hook 这个 hook 程序，需要两个参数，第一个是主机上的网卡名字，第二个是在容器内的网卡重命名。

- 注意事项

  a. hook path 必须为 docker 的 graph 目录（--graph）下的 hooks 文件夹下，默认一般为 /var/lib/docker/hooks，可以通过 docker info 命令查看 root 路径。

  ```
  [root@localhost ~]# docker info
  ...
  Docker Root Dir: /var/lib/docker
  ...
  ```

  这个路径可能会跟随用户手动配置，以及 user namespace 的使用（daemon --userns-remap）而变化。 path 进行软链接解析后，必须以 Docker Root Dir/hooks 开头（如本例中使用 /var/lib/docker/hooks 开头），否则会直接报错。

  b. hooks path 必须指定绝对路径，因为这个是由 daemon 处理，相对路径对 daemon 无意义。同时绝对路径也更满足安全要求。

  c. hook 程序打印到 stderr 的输出会打印给客户端并对容器的声明周期产生影响（比如启动失败），而输出到 stdout 的打印信息会被直接忽略。

  d. 严禁在 hook 里反向调用 docker 的指令。

  e. 配置的 hook 执行文件必须要有可执行权限，否则 hook 执行会报错。

  f. 使用 hook 时，执行时间应尽量短。如果 hook 中的 prestart 时间过长（超过 2 分钟），则会导致容器启动超时失败，如果 hook 中的 poststop 时间过长（超过 2 分钟），也会导致容器异常。

  目前已知的异常如下：执行 docker stop 命令停止容器时，2 分钟超时执行清理时，由于 hook 还没执行结束，因此会等待 hook 执行结束（该过程持有锁），从而导致和该容器相关的操作都会卡住，需要等到 hook 执行结束才能恢复。另外，由于 docker stop 命令的 2 分钟超时处理是异步的过程，因此即使 docker stop 命令返回了成功，容器的状态也依然是 up 状态，需要等到 hook 执行完后状态才会修改为 exited。

- 使用建议

  a. 建议配置 hook 的 Timeout 超时时间阈值，超时时间最好在 5s 以内。

  b. 建议不要配置过多 hook，每个容器建议 prestart、poststart、poststop 这三个 hook 都只配置一个，过多 hook 会导致启动时间长。

c. 建议用户识别多个 hook 之间的依赖关系，如果存在依赖关系，在组合 hook 配置文件时要根据依赖关系灵活调整顺序，hook 的执行顺序是按照配置的 spec 文件上的先后顺序。

多个 hook-spec

当有多个 hook 配置文件，要运行多个 hook 时，用户必须自己手工将多个 hook 配置文件组合成一个配置文件，使用--hook-spec 参数指定此合并后的配置文件，方可生效所有的 hook；如果配置多个--hook-spec 参数，则只有最后一个生效。

配置举例：

hook1.json 内容如下：

```
# cat /var/lib/docker/hooks/hookspec.json
{
   "prestart": [
      {
         "path": "/var/lib/docker/hooks/lxcfs-hook",
         "args": ["lxcfs-hook", "--log", "/var/log/lxcfs-hook.log"],
         "env": []
      }
    ],
    "poststart":[],
    "poststop":[]
}
```

hook2.json 内容如下：

```
# cat /etc/isulad-tools/hookspec.json
{
     "prestart": [
       {
            "path": "/docker-root/hooks/docker-hooks",
            "args": ["docker-hooks", "--state", "prestart"],
            "env": []
       }
      ],
      "poststart":[],
      "poststop":[
        {
            "path": "/docker-root/hooks/docker-hooks",
            "args": ["docker-hooks", "--state", "poststop"],
            "env": []
        }
       ]
}
```

手工合并后的 json 内容如下：

```
{
      "prestart":[
        {
             "path": "/var/lib/docker/hooks/lxcfs-hook",
             "args": ["lxcfs-hook", "--log", "/var/log/lxcfs-hook.log"],
             "env": []
```

```
        },
        {
            "path": "/docker-root/hooks/docker-hooks",
            "args": ["docker-hooks", "--state", "prestart"],
            "env": []
        }
      ],
      "poststart":[],
      "poststop":[
         {
            "path": "/docker-root/hooks/docker-hooks",
            "args": ["docker-hooks", "--state", "poststop"],
            "env": []
         }
       ]
}
```

需要注意的是，docker daemon 会按照数组顺序依次读取 hook 配置文件中 prestart 等 action 中的 hook 二进制，进行执行动作。用户需要识别多个 hook 之间的依赖关系，如果有依赖关系，在组合 hook 配置文件时要根据依赖关系灵活调整顺序。

为所有容器定制默认的 hook

Docker daemon 同样可以接收--hook-spec 的参数，--hook-spec 的语义与 docker create/run 的--hook-spec 参数相同，这里不再复述。也可以在/etc/docker/daemon.json 里添加 hook 配置：

```
{
    "hook-spec": "/tmp/hookspec.json"
}
```

容器在运行时，会首先执行 daemon 定义的--hook-spec 中指定的 hooks，然后再执行每个容器单独定制的 hooks。

4.3.3 创建容器配置健康检查

Docker 提供了用户定义的对容器进行健康检查的功能。在 Dockerfile 中配置 HEALTHCHECK CMD 选项，或在容器创建时配置--health-cmd 选项，在容器内部周期性地执行命令，通过命令的返回值来监测容器的健康状态。

配置方法

- 在 Dockerfile 中添加配置，如：

  ```
  HEALTHCHECK --interval=5m --timeout=3s --health-exit-on-unhealthy=true \
    CMD curl -f http://localhost/ || exit 1
  ```

  可配置的选项：

  a. --interval=DURATION，默认 30s，相邻两次命令执行的间隔时间。另外，容器启动后，经过 interval 时间进行第一次检查。

  b. --timeout=DURATION，默认 30s，单次检查命令执行的时间上限，超时则任务命令执行失败。

  c. --start-period=DURATION，默认 0s，容器初始化时间。初始化期间也会执行健康检查，健康检查失败不会计入最大重试次数。但是，如果在初始化期间

运行状况检查成功，则认为容器已启动。之后所有连续的检查失败都将计入最大重试次数。

d. --retries=N，默认 3，健康检查失败最大的重试次数。

e. --health-exit-on-unhealthy=BOOLEAN，默认 false，检测到容器非健康时是否杀死容器

f. CMD，必选，在容器内执行的命令。返回值为 0 表示成功，非 0 表示失败。

在配置了 HEALTHCHECK 后创建镜像，HEALTHCHECK 相关配置会被写入镜像的配置中。通过 docker inspect 可以看到。如：

```
"Healthcheck": {
   "Test": [
      "CMD-SHELL",
      "/test.sh"
   ]
},
```

- 在容器创建时的配置：

```
docker run -itd --health-cmd "curl -f http://localhost/ || exit 1" --health-interval 5m --health-timeout 3s --health-exit-on-unhealthy centos bash
```

可配置的选项：

a. --health-cmd，必选，在容器内执行的命令。返回值为 0 表示成功，非 0 表示失败。

b. --health-interval，默认 30s，最大为 int64 上限（纳秒）相邻两次命令执行的间隔时间。

c. --health-timeout，默认 30s，最大为 int64 上限（纳秒），单次检查命令执行的时间上限，超时则任务命令执行失败。

d. --health-start-period，默认 0s，最大为 int64 上限（纳秒），容器初始化时间。

e. --health-retries，默认 3，最大为 int32 上限，健康检查失败最大的重试次数。

f. --health-exit-on-unhealthy，默认 false，检测到容器非健康时是否杀死容器。

容器启动后，HEALTHCHECK 相关配置会被写入容器的配置中。通过 docker inspect 可以看到。如：

```
"Healthcheck": {
   "Test": [
      "CMD-SHELL",
      "/test.sh"
   ]
},
```

检查规则

1. 容器启动后，容器状态中显示 health:starting。
2. 经过 start-period 时间后开始，以 interval 为间隔周期性在容器中执行 CMD。即：当一次命令执行完毕后，经过 interval 时间，执行下一次命令。
3. 若 CMD 命令在 timeout 限制的时间内执行完毕，并且返回值为 0，则视为一次检查成功，否则视为一次检查失败。检查成功后，容器状态变为 health:healthy。

4. 若 CMD 命令连续 retries 次检查失败，则容器状态变为 health:unhealthy。失败后容器也会继续进行健康检查。
5. 容器状态为 health:unhealthy 时，任意一次检查成功会使得容器状态变为 health:healthy。
6. 设置--health-exit-on-unhealthy 的情况下，如果容器因为非被杀死退出（退出返回值 137）后，健康检查只有容器在重新启动后才会继续生效。
7. CMD 执行完毕或超时时，docker daemon 会将这次检查的起始时间、返回值和标准输出记录到容器的配置文件中。最多记录最新的 5 条数据。此外，容器的配置文件中还存储着健康检查的相关参数。

通过 docker ps 可以看到容器状态。

```
[root@bac shm]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
7de2228674a2        testimg             "bash"              About an hour ago   Up About
an hour (unhealthy)                     cocky_davinci
```

运行中的容器的健康检查状态也会被写入容器配置中。通过 docker inspect 可以看到。

```
"Health": {
    "Status": "healthy",
    "FailingStreak": 0,
    "Log": [
        {
            "Start": "2018-03-07T07:44:15.481414707-05:00",
            "End": "2018-03-07T07:44:15.556908311-05:00",
            "ExitCode": 0,
            "Output": ""
        },
        {
            "Start": "2018-03-07T07:44:18.557297462-05:00",
            "End": "2018-03-07T07:44:18.63035891-05:00",
            "ExitCode": 0,
            "Output": ""
        },
        ......
}
```

说明

- 容器内健康检查的状态信息最多保存 5 条。会保存最后得到的 5 条记录。
- 容器内健康检查相关配置同时最多只能有一条生效。Dockerfile 中配置的靠后的条目会覆盖靠前的；容器创建时的配置会覆盖镜像中的。
- 在 Dockerfile 中可以通过 HEALTHCHECK NONE 来取消引用的镜像中的健康检查配置。在容器运行时可以通过配置--no-healthcheck 来取消镜像中的健康检查配置。不允许在启动时同时配置健康检查相关选项与--no-healthcheck 选项。
- 带有健康检查配置的容器启动后，若 docker daemon 退出，则健康检查不会执行，一直等待。docker daemon 再次启动后，容器健康状态会变为 starting。之后检查规则同上。
- 构建容器镜像时若健康检查相关参数配置为空，则按照默认值处理。
- 容器启动时若健康检查相关参数配置为 0，则按照默认值处理。

4.3.4 停止与删除容器

用 docker stop 停止名为 container1 的容器：

```
[root@localhost ~]# docker stop container1
```

也可以用 docker kill 来杀死容器达到停止容器的目的：

```
[root@localhost ~]# docker kill container1
```

当容器停止之后，可以使用 docker rm 删除容器：

```
[root@localhost ~]# docker rm container1
```

当然，使用 docker rm -f 强制删除容器也是可以的：

```
[root@localhost ~]# docker rm -f container1
```

注意事项

- 禁止使用 docker rm -f XXX 删除容器。如果使用强制删除，docker rm 会忽略过程中的错误，可能导致容器相关元数据残留。如果使用普通删除，如果删除过程出错，则会删除失败，不会导致元数据残留。
- 避免使用 docker kill 命令。docker kill 命令发送相关信号给容器内业务进程，依赖于容器内业务进程对信号的处理策略，可能导致业务进程的信号处理行为与指令的预期不符合的情况。
- docker stop 处于 restarting 状态的容器可能容器不会马上停止。如果一个容器使用了重启规则，当容器处于 restarting 状态时，docker stop 这个容器时有很低的概率会立即返回，容器仍然会在重启规则的作用下再次启动。
- 不能用 docker restart 重启加了--rm 参数的容器。加了--rm 参数的容器在退出时，容器会主动删除，如果重启一个加了--rm 的参数的容器， 可能会导致一些异常情况，比如启动容器时，同时加了--rm 与-ti 参数，对容器执行 restart 操作，可能会概率性卡住无法退出。

docker stop/restart 指定 t 参数且 t<0 时，请确保自己容器的应用会处理 stop 信号

Stop 的原理：（Restart 会调用 Stop 流程）

1. Stop 会首先给容器发送 Stop 信号（15）
2. 然后等待一定的时间（这个时间就是用户输入的 t）
3. 过了一定时间，如果容器还活着，那么就发送 kill 信号（9）使容器强制退出

输入参数 t（单位 s）的含义：

- t<0 ： 表示死等，不管多久都等待程序优雅退出，既然用户这么输入了，表示对自己的应用比较放心，认为自己的程序有合理的 stop 信号的处理机制
- t=0 ： 表示不等，立即发送 kill -9 到容器
- t>0 ： 表示等一定的时间，如果容器还未退出，就发送 kill -9 到容器

所以如果用户使用 t<0 (比如 t=-1)，请确保自己容器的应用会正确处理 signal 15，如果容器忽略了该信号，会导致 docker stop 一直卡住。

如果容器处于 Dead 状态，可能底层文件系统处于 busy 状态，需要手动删除

Docker 在执行容器删除时，先停止容器的相关进程，之后将容器状态更改为 Dead，最后执行容器 rootfs 的删除操作。当文件系统或者 device mapper 处于忙碌状态时，最后一步 rootfs 的删除会失败。docker ps -a 查看会发现容器处于 Dead 状态。Dead 状态的容器不能再次启动，需要等待文件系统不繁忙时，手动再次执行 docker rm 进行删除。

共享 pid namespace 容器，子容器处于 pause 状态会使得父容器 stop 卡住，并影响 docker run 命令执行

使用--pid 参数创建共享 pid namespace 的父子容器，在执行 docker stop 父容器时，如果子容器中有进程无法退出（比如处于 D 状态、pause 状态），会产生父容器 docker stop 命令等待的情况，需要手动恢复这些进程，才能正常执行命令。

遇到该问题的时候，请对 pause 状态的容器使用 docker inspect 命令查询 PidMode 对应的父容器是否为需要 docker stop 的容器。如果是该容器，请使用 docker unpause 将子容器解除 pause 状态，指令即可继续执行。

一般来说，导致该类问题的可能原因是容器对应的 pid namespace 由于进程残留导致无法被销毁。如果上述方法无法解决问题，可以通过借助 linux 工具，获取容器内残留进程，确定 pid namespace 中进程无法退出的原因，解决后容器就可以退出：

- 获取容器 pid namespace id

```
docker inspect --format={{.State.Pid}} CONTAINERID | awk '{print
"/proc/"$1"/ns/pid"}' |xargs readlink
```

- 获取该 namespace 下的线程

```
 ls -l /proc/*/task/*/ns/pid |grep -F PIDNAMESPACE ID |awk '{print $9}' |awk -F
\/ '{print $5}'
```

4.3.5 容器信息查询

在任何情况下，容器的状态都不应该以 docker 命令执行是否成功返回为判断标准。如想查看容器状态，建议使用：

```
docker inspect <NAME|ID>
```

4.3.6 修改操作

docker exec 进入容器启动多个进程的注意事项

docker exec 进入容器执行的第一个命令为 bash 命令时，当退出 exec 时，要保证在这次 exec 启动的进程都退出了，再执行 exit 退出，否则会导致 exit 退出时终端卡主的情况。如果要在 exit 退出时，exec 中启动的进程仍然在后台保持运行，要在启动进程时加上 nohup。

docker rename 和 docker stats <container_name>的使用冲突

如果使用 docker stats <container_name> 实时监控容器，当使用 docker rename 重命名容器之后，docker stats 中显示的名字将还是原来的名字，不是 rename 后的名字。

docker rename 操作 restarting 状态的容器可能会失败

对一个处于 restarting 状态的容器执行 rename 操作的时候，docker 会同步修改容器网络的相关配置。由于 restarting 状态的容器可能还未真正启动起来，网络是不存在的，导致 rename 操作报错 sandbox 不存在。建议 rename 只操作非 restarting 的稳定状态的容器。

docker cp

1. 使用 docker cp 向容器中拷贝文件时，docker ps 以及所有对这个容器的操作都将等待 docker cp 结束之后才能进行。
2. 容器以非 root 用户运行，当使用 docker cp 命令复制主机上的一个非 root 权限的文件到容器时，文件在容器中的权限角色会变成 root。docker cp 与 cp 命令不同，docker cp 会修改复制到容器中文件的 uid 和 gid 为 root。

docker login

执行 docker login 后，会将 usrer/passwd 经 aes（256 位）加密后保存在 /root/.docker/config.json，同时生成 *root*.docker/aeskey(权限 0600)，用来解密 /root/.docker/config.json 中的 usrer/passwd。目前不能定时更新 aeskey，只能由用户手动删除 aeskey 来更新。aeskey 更新后，不管是否重启过 docker daemon，都需要重新 login，才可以 push。例如：

```
root@hello:~/workspace/dockerfile# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: example Password:
Login Succeeded
root@hello:~/workspace/dockerfile# docker push example/empty
The push refers to a repository [docker.io/example/empty]
547b6288eb33: Layer already exists
latest: digest:
sha256:99d4fb4ce6c6f850f3b39f54f8eca0bbd9e92bd326761a61f106a10454b8900b size: 524
root@hello:~/workspace/dockerfile# rm /root/.docker/aeskey
root@hello:~/workspace/dockerfile# docker push example/empty
WARNING: Error loading config file:/root/.docker/config.json - illegal base64 data
at input byte 0
The push refers to a repository [docker.io/example/empty]
547b6288eb33: Layer already exists
errors:
denied: requested access to the resource is denied
unauthorized: authentication required
root@hello:~/workspace/dockerfile# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: example
Password:
Login Succeeded
root@hello:~/workspace/dockerfile# docker push example/empty
The push refers to a repository [docker.io/example/empty]
547b6288eb33: Layer already exists
latest: digest:
sha256:99d4fb4ce6c6f850f3b39f54f8eca0bbd9e92bd326761a61f106a10454b8900b size: 524
```

4.4 镜像管理

4.4.1 创建镜像

docker pull、docker build、docker commit、docker import、docker load 都可以创建一个新的镜像，关于这些命令的使用详见命令行参考镜像管理。

注意事项

1. 避免并发 docker load 和 docker rmi 操作。 如果同时满足如下两个条件，可能导致并发性问题：
   - 某个镜像存在于系统中。
   - 同时对该镜像进行 docker rmi 和 docker load 操作。

   所以使用时应该避免这种场景（注：所有的镜像创建操作如 tag，build，load 和 rmi 并发都有可能会导致类似的错误，应该尽量避免这类操作与 rmi 的并发）。
2. 如果 Docker 操作镜像时系统掉电，可能导致镜像损坏，需要手动恢复。

   由于 Docker 在操作镜像（pull/load/rmi/build/combine/commit/import 等）时,镜像数据的操作是异步的、镜像元数据是同步的。所以如果在镜像数据未全部刷到磁盘时掉电，可能导致镜像数据和元数据不一致。对用户的表现是镜像可以看到(有可能是 none 镜像)，但是无法启动容器，或者启动后的容器有异常。这种情况下应该先使用 docker rmi 删除该镜像，然后重新进行之前的操作，系统可以恢复。
3. 生产环境节点应避免存留超大数量镜像，请及时清理不使用的镜像。

   镜像数目过多会导致 docker image 等命令执行过慢，从而导致 docker build/docker commit 等相关命令执行失败，并可能导致内存堆积。在生产环境中，请及时清理不再使用的镜像和中间过程镜像。
4. 使用--no-parent 参数 build 镜像时，如果有多个 build 操作同时进行，并且 Dockerfile 里 FROM 的镜像相同，则可能会残留镜像，分为以下两种情况：
   - FROM 的镜像不是完整镜像，则有可能会残留 FROM 的镜像运行时生成的镜像。残留的镜像名类似 base_v1.0.0-app_v2.0.0，或者残留<none>镜像。
   - 如果 Dockerfile 里的前几条指令相同，则有可能会残留<none>镜像。

可能会产生 none 镜像场景

1. none 镜像是指没有 tag 的最顶层镜像，比如 ubuntu 的 imageID，只有一个 tag 是 ubuntu，如果这个 tag 没了，但是 imageID 还在，那么这个 imageID 就变成了 none 镜像。
2. Save 镜像的过程中因为要把镜像的数据导出来，所以对 image 进行保护，但是如果这个时候来一个删除操作，可能会 untag 成功，删除镜像 ID 失败，造成该镜像变成 none 镜像。
3. 执行 docker pull 时掉电，或者系统 panic，可能出现 none 镜像，为保证镜像完整性，此时可通过 docker rmi 删除镜像后重新拉取。
4. 执行 docker save 保存镜像时，如果指定的名字为镜像 ID，则 load 后的镜像也没有 tag，其镜像名为 none。

build 镜像的同时删除该镜像，有极低概率导致镜像 build 失败

目前的 build 镜像的过程是通过引用计数来保护的，当 build 完一个镜像后，紧接着就给该镜像的引用计数加 1（holdon 操作），一旦 holdon 操作成功，该镜像就不会被删除了，但是在 holdon 之前，有极低的概率，还是可以删除成功，导致 build 镜像失败。

4.4.2 查看镜像

查看本地镜像列表：

```
docker images
```

4.4.3 删除镜像

注意事项

禁止使用 docker rmi －f XXX 删除镜像。如果使用强制删除，docker rmi 会忽略过程中的错误，可能导致容器或者镜像关元数据残留。如果使用普通删除，如果删除过程出错，则会删除失败，不会导致元数据残留。

4.5 命令行参考

4.5.1 容器引擎

Docker daemon 是一个常驻后台的系统进程，docker 子命令执行前先要启动 docker daemon。

如果是通过 rpm 包或者系统包管理工具安装的，就可以使用 systemctl start docker 来启动 docker daemon。

docker 命令支持多个参数选项，对于参数选项有以下约定：

1. 单个字符的选项可以合并在一起，如：

   ```
   docker run -t -i busybox /bin/sh
   ```

   可以写成

   ```
   docker run -ti busybox /bin/sh
   ```

2. 在命令帮助中看到的如--icc=true 之类的 bool 命令选项，如果没有使用这个选项，则这个标志位的值就是在命令帮助中看到的默认值，如果使用了这个选项则这个标志位的值就是命令帮助中看的值的相反值，如果启动 docker daemon 没有加上使用--icc 选项，则默认设置了--icc=true,如果使用了--icc 选项则表示是--icc=false。
3. 在命令帮助中看到的--attach=[]之类的选项，表示这类的选项可以多次设置，如：

   ```
   docker run --attach=stdin --attach=stdout -i -t busybox /bin/sh
   ```

4. 在命令帮助中看到的-a, --attach=[]之类的选项，表示这种选项既可以用-a value 指定也可以用--attach=value 指定。如：

   ```
   docker run -a stdin --attach=stdout -i -t busybox /bin/sh
   ```

5. --name=” ”之类的选项需要的是一个字符串，只能指定一次，-c=0 之类的选项需要的是一个整数，只能指定一次。

表4-1 docker daemon 启动时指定参数详解

参数名称	说明
--api-cors-header	开放远程 API 调用的 CORS 头信息。这个接口开关对想进行二次开发的上层应用提供了支持。为 remote API 设置 CORS 头信息。
--authorization-plugin=[]	指定认证插件。
-b, --bridge=""	挂载已经存在的网桥设备到 Docker 容器里。注意，使用 **none** 可以停用容器里的网络。
--bip=""	使用 CIDR 地址来设定自动创建的网桥的 IP。注意，此参数和 -b 不能一起使用。
--cgroup-parent	为所有容器设定 cgroup 父目录。
--config-file=/etc/docker/daemon.json	启动 docker daemon 的配置文件。
--containerd	指定 containerd 的 socket 路径。
-D, --debug=false	开启 Debug 模式。
--default-gateway	容器 IPv4 地址的默认网关。
--default-gateway-v6	容器 IPv6 地址的默认网关。
--default-ulimit=[]	容器的默认 ulimit 值。
--disable-legacy-registry	不允许使用原版 registry。
--dns=[]	强制容器使用 DNS 服务器。 例如： --dns 8.8.x.x
--dns-opt=[]	指定使用 DNS 的选项。
--dns-search=[]	强制容器使用指定的 DNS 搜索域名。 例如： --dns-search example.com

--exec-opt=[]	设置运行时执行选项。
	例如支持 native.umask 选项：
	`# 启动的容器umask值为0022`
	`--exec-opt native.umask=normal`
	
	`# 启动的容器umask值为0027（默认值）`
	`--exec-opt  native.umask=secure`
	注意如果 docker create/run 也配置了 native.umask 参数则以 docker create/run 中的配置为准。

参数名称	说明
--exec-root=/var/run/docker	指定执行状态文件存放的根目录。
--fixed-cidr=""	设定子网固定 IP（ex: 10.20.0.0/16），这个子网 IP 必须属于网桥内的。
--fixed-cidr-v6	同上，使用与 IPv6。
-G, --group="docker"	在后台运行模式下，赋予指定的 Group 到相应的 unix socket 上。注意，当此参数 --group 赋予空字符串时，将去除组信息。
-g, --graph="/var/lib/docker"	配置 Docker 运行时根目录。

-H, --host=[]	在后台模式下指定 socket 绑定，可以绑定一个或多个 tcp://host:port, unix:///path/to/socket, fd://* 或 fd://socketfd。例如：
	$ dockerd -H tcp://0.0.0.0:2375
	或者
	$ export DOCKER_HOST="tcp://0.0.0.0:2375"

--insecure-registry=[]	指定非安全连接的仓库，docker 默认所有的连接都是 TLS 证书来保证安全的，如果仓库不支持 https 连接或者证书是 docker daemon 不清楚的证书颁发机构颁发的，则启动 daemon 的时候要指定如--insecure-registry=192.168.1.110:5000，使用私有仓库都要指定。
--image-layer-check=true	开启镜像层完整性检查功能，设置为 true；关闭该功能，设置为 false。如果没有该参数，默认为关闭。 docker 启动时会检查镜像层的完整性，如果镜像层被破坏，则相关的镜像不可用。docker 进行镜像完整性校验时，无法校验内容为空的文件和目录，以及链接文件。因此若镜像因掉电导致上述类型文件丢失，docker 的镜像数据完整性校验可能无法识别。docker 版本变更时需要检查是否支持该参数，如果不支持，需要从配置文件中删除。
--icc=true	启用容器间的通信。
--ip="0.0.0.0"	容器绑定端口时使用的默认 IP 地址。
--ip-forward=true	启动容器的 net.ipv4.ip_forward。
--ip-masq=true	使能 IP 伪装。
--iptables=true	启动 Docker 容器自定义的 iptable 规则。
-l, --log-level=info	设置日志级别。

参数名称	说明
--label=[]	设置 daemon 标签，以 key=value 形式设置。
--log-driver=json-file	设置容器日志的默认日志驱动。
--log-opt=map[]	设置日志驱动参数。
--mtu=0	设置容器网络的 MTU 值，如果没有这个参数，选用默认 route MTU，如果没有默认 route，就设置成常量值 1500。
-p, --pidfile="/var/run/docker.pid"	后台进程 PID 文件路径。
--raw-logs	带有全部时间戳并不带 ANSI 颜色方案的日志。
--registry-mirror=[]	指定 dockerd 优先使用的镜像仓库。
-s, --storage-driver=""	强制容器运行时使用指定的存储驱动
--selinux-enabled=false	启用 selinux 支持，3.10.0-862.14 及以上内核版本不支持--selinux-enabled=true。
--storage-opt=[]	配置存储驱动的参数，存储驱动为 devicemapper 的时候有效（e.g. dockerd --storage-opt dm.blocksize=512K）。
--tls=false	启动 TLS 认证开关。
--tlscacert="/root/.docker/ca.pem"	通过 CA 认证过的的 certificate 文件路径。
--tlscert="/root/.docker/cert.pem"	TLS 的 certificate 文件路径。
--tlskey="/root/.docker/key.pem"	TLS 的 key 文件路径。
--tlsverify=false	使用 TLS 并做后台进程与客户端通讯的验证。
--insecure-skip-verify-enforce	是否强制跳过证书的主机名/域名验证，默认为 false（不跳过）。
--use-decrypted-key=true	指定使用解密私钥。
--userland-proxy=true	容器 LO 设备使用 userland proxy。
--userns-remap	容器内使用 user 命名空间的用户映射表。 **说明** 当前版本不支持该参数。

4.5.2 容器管理

当前 docker 支持的子命令，按照功能划分为以下几组：

功能划分	命令	命令功能

功能划分	命令		命令功能
主机环境相关	version		查看 docker 版本信息
	info		查看 docker 系统和主机环境信息
容器相关	容器生命周期管理	create	由 image 创建一个容器
		run	由 image 创建一个容器并运行
		start	开始一个已停止运行的容器
		stop	停止一个运行中的容器
		restart	重启一个容器
		wait	等待一个容器停止，并打印出退出码
		rm	删除一个容器
	容器内进程管理	pause	暂停一个容器内的所有进程
		unpause	恢复一个容器内被暂停的所用进程
		top	查看容器内的进程
		exec	在容器内执行进程
	容器检视工具	ps	查看运行中的容器（不加任何选项）
		logs	显示一个容器的日志信息
		attach	连接到一个容器的标准输入输出
		inspect	返回容器的底层信息
		port	列出容器与主机的端口映射
		diff	返回容器相对于镜像中的 rootfs 所作

功能划分	命令		命令功能
			的改动
		cp	容器与主机之间复制文件
		export	将一个容器中的文件系统导出为一个 tar 包
		stats	实时查看容器的资源占用情况
images 相关	生成一个新 image	build	通过一个 Dockerfile 构建一个 image
		commit	基于容器的 rootfs 创建一个新的 image
		import	将 tar 包中的内容作为文件系统创建一个 image
		load	从一个 tar 包中加载一个 image
	与 image 仓库有关	login	登录一个 registry
		logout	登出一个 registry
		pull	从 registry 中拉取一个 image
		push	将一个 image 推送到 registry 中
		search	在 registry 中搜寻 image
	与 image 管理有关	images	显示系统中的 image
		history	显示一个 image 的变化历史
		rmi	删除 image
		tag	给 image 打标签
		save	将一个 image 保存到一个 tar 包中
其他	events		从 docker daemon

功能划分	命令	命令功能
		中获取实时事件
	rename	重命名容器

其中有些子命令还有一些参数选项如 docker run,通过 docker COMMAND --help 可以查看相应 COMMAND 命令的帮助，命令选项参考上文的命令选项约定。下面详细介绍每个命令的使用。

4.5.2.1 attach

用法：**docker attach [OPTIONS] CONTAINER**

功能：附加到一个运行着的容器

选项：

--no-stdin=false 不附加 STDIN

--sig-proxy=true 代理所有到容器内部的信号，不代理 SIGCHLD, SIGKILL, SIGSTOP

示例：

```
$ sudo docker attach attach test
root@2988b8658669:/# ls bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc
root  run  sbin  srv  sys  tmp  usr  var
```

4.5.2.2 commit

用法：**docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]**

功能：由一个容器创建一个新的 image

选项：

-a, --author="" 指定作者

-m, --message="" 提交的信息

-p, --pause=true 在提交过程中暂停容器

示例：

运行一个容器，然后将这个容器提交成一个新的 image

```
$ sudo docker commit test busybox:test
sha256:be4672959e8bd8a4291fbdd9e99be932912fe80b062fba3c9b16ee83720c33e1

$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox            latest              e02e811dd08f        2 years ago         1.09MB
```

4.5.2.3 cp

用法：docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH|-

docker cp [OPTIONS] SRC_PATH|- CONTAINER:DEST_PATH

功能：从指定的容器内的一个路径复制文件或文件夹到主机的指定路径中，或者把主机的文件或者文件夹拷贝到容器内。

注意：docker cp 不支持容器内/proc，/sys，/dev，/tmp 等虚拟文件系统以及用户在容器内自行挂载的文件系统内的文件拷贝。

选项：

-a, --archive　将拷贝到容器的文件属主设置为容器运行用户（--user）

-L, --follow-link　解析并跟踪文件的符号链接

示例：

复制 reigistry 容器中/test 目录到主机的/home/aaa 目录中

```
$ sudo docker cp registry:/test /home/aaa
```

4.5.2.4 create

用法：**docker create [OPTIONS] IMAGE [COMMAND] [ARG...]**

功能：使用 image 创建一个新的容器，并将返回一个容器的 ID，创建之后的容器用 docker start 命令启动，OPTIONS 用于创建容器时对容器进行配置，有些选项将覆盖 image 中对容器的配置，COMMAND 指定容器启动时执行的命令。

选项：

表4-2 参数说明

参数	参数含义
-a --attach=[]	使控制台 Attach 到容器内进程的 STDIN,STDOUT,STDERR
--name=""	指定容器的名字
--add-host=[host:ip]	在容器内的/etc/hosts 中添加一个 hostname 到 IP 地址的映射 e.g. --add-host=test:10.10.10.10

--annotation	设置容器的 annotations。例如支持 native.umask 选项：
	`--annotation native.umask=normal # 启动的容器 umask 值为 0022`
	`--annotation native.umask=secure # 启动的容器 umask 值为 0027`
	注意如果没有配置该参数，则使用 dockerd 中的 umask 配置。

--blkio-weight	blockio 的相对权重，从 10 到 1000

参数	参数含义
--blkio-weight-device=[]	blockio 权重（设置相对权重）
-c, --cpu-shares=0	容器获得主机 CPU 的相对权重，通过设置这个选项获得更高的优先级，默认所有的容器都是获得相同的 CPU 优先权。
--cap-add=[]	添加 Linux 权能
--cap-drop=[]	清除 Linux 权能
--cgroup-parent	为容器设置 cgroup 父目录
--cidfile=""	将容器的 ID 写到指定的文件中 e.g. --cidfile=/home/cidfile-test 将该容器的 ID 写入到 /home/cidfile-test 中
--cpu-period	设置 CFS（完全公平调度策略）进程的 CPU 周期。 默认值为 100ms；一般--cpu-period 参数和--cpu-quota 是配合使用的，比如--cpu-period=50000 --cpu-quota=25000，意味着如果有 1 个 CPU，该容器可以每 50ms 获取到 50% 的 CPU。 使用--cpus=0.5 也可达到同样的效果
--cpu-quota	设置 CFS(完全公平调度策略)进程的 CPU 配额，默认为 0，即没有限制
--cpuset-cpus	设置容器中进程允许运行的 CPU (0-3, 0,1)。默认没有限制
--cpuset-mems	设置容器中进程运行运行的内存内存节点 (0-3, 0,1)，只对 NUMA 系统起作用
--device=[]	将主机的设备添加到容器中 (e.g. --device=/dev/sdc:/dev/xvdc:rwm)
--dns=[]	强制容器使用指定的 dns 服务器（e.g. 创建容器时指定--dns=114.114.xxx.xxx，将在容器的/etc/resolv.conf 中写入 nameserver 114.114.xxx.xxx 并将覆盖原来的内容）
--dns-opt=[]	设置 DNS 选项
--dns-search=[]	强制容器使用指定的 dns 搜索域名
-e, --env=[]	设置容器的环境变量 --env=[KERNEL_MODULES=]: 在容器中插入指定模块。目前仅支持 Host 主机上有的模块，且容器删除后 Host 主机上模块仍然驻留，且容器需要同时指定--hook-spec 选项。以下都是参数的合法格式： KERNEL_MODULERS=

参数	参数含义
	KERNEL_MODULERS=a KERNEL_MODULERS=a,b KERNEL_MODULERS=a,b,
--entrypoint=""	覆盖 image 中原有的 entrypoint，entrypoint 设置容器启动时执行的命令
--env-file=[]	从一个文件中读取环境变量，多个环境变量在文件中按行分割（e.g. --env-file=/home/test/env,其中 env 文件中存放了多个环境变量）
--expose=[]	开放一个容器内部的端口，使用下文介绍的-P 选项将会使开放的端口映射到主机的一个端口。
--group-add=[]	指定容器添加到额外的组
-h, --hostname=""	设置容器主机名
--health-cmd	设置容器健康检查执行的命令
--health-interval	相邻两次命令执行的间隔时间，默认 30s
--health-timeout	单次检查命令执行的时间上限，超时则任务命令执行失败，默认 30s
--health-start-period	容器启动距离第一次执行健康检查开始的时间，默认 0s
--health-retries	健康检查失败最大的重试次数，默认 3
--health-exit-on-unhealthy	容器被检查为非健康后停止容器，默认 false
--host-channel=[]	设置一个通道供容器内进程与主机进行通信，格式：<host path>:<container path>:<rw/ro>:<size limit>
-i, --interactive=false	设置 STDIN 打开即使没有 attached
--ip	设置容器的 IPv4 地址
--ip6	设置容器的 IPv6 地址
--ipc	指定容器的 ipc 命名空间
--isolation	指定容器隔离策略
-l, --label=[]	设置容器的标签
--label-file=[]	从文件中获取标签
--link=[]	链接到其他容器，这个选项将在容器中添加一些被链接容器 IP 地址和端口的环境变量及在/etc/hosts 中添加一条映射（e.g. --link=name:alias）
--log-driver	设置容器的日志驱动
--log-opt=[]	设置日志驱动选项

参数	参数含义
-m, --memory=""	设置容器的内存限制，格式<number><optional unit>，其中 unit = b, k, m or g。该参数最小值为 4m。
--mac-address	设置容器的 mac 地址（e.g. 92:d0:c6:0a:xx:xx）
--memory-reservation	设置容器内存限制，默认与--memory 一致。可认为--memory 是硬限制，--memory-reservation 是软限制；当使用内存超过预设值时，会动态调整（系统回收内存时尝试将使用内存降低到预设值以下），但不确保一定不超过预设值。一般可以和--memory 一起使用，数值小于--memory 的预设值。
--memory-swap	设置普通内存和交换分区的使用总量，-1 为不做限制。如果不设置，则为--memory 值的 2 倍，即 SWAP 可再使用与--memory 相同的内存量。
--memory-swappiness=-1	设置容器使用交换内存的时机,以剩余内存百分比为度量（0-100）
--net="bridge"	设置容器的网络模式，当前 1.3.0 版本的 docker 有四个模式:bridge、host、none、container:<name|id>。默认使用的是 bridge。 • bridge：使用桥接模式在 docker daemon 启动时使用的网桥上创建一个网络栈。 • host:在容器内使用主机的网络栈 • none:不使用网络 • container:<name|id>：重复利用另外一个容器的网络栈
--no-healthcheck	设置容器不使用健康检查
--oom-kill-disable	禁用 OOM killer，建议如果不设置-m 参数，也不要设置此参数。
--oom-score-adj	调整容器的 oom 规则（-1000 到 1000）
-P, --publish-all=false	将容器开放的所有端口一一映射到主机的端口，通过主机的端口可以访问容器内部，通过下文介绍的 docker port 命令可以查看具体容器端口和主机端口具体的映射关系。
-p, --publish=[]	将容器内的一个端口映射到主机的一个端口，format: ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort | containerPort，如果没有指定 IP 代表侦听主机所有网卡的访问，如果没有指定 hostPort,表示自动分配主机的端口。
--pid	设置容器的 PID 命名空间
--privileged=false	给予容器额外的权限，当使用了--privileged 选项，容器将可以访问主机的所有设备。

参数	参数含义
--restart=""	设置容器退出时候的重启规则，当前 1.3.1 版本支持 3 个规则： • no：当容器停止时，不重启。 • on-failure：当容器退出码为非 0 时重启容器，这个规则可以附加最大重启次数，如 on-failure:5，最多重启 5 次。 • always：无论退出码是什么都退出。
--read-only	将容器的根文件系统以只读的形式挂载
--security-opt=[]	容器安全规则
--shm-size	/dev/shm 设备的大小，默认值是 64M
--stop-signal=SIGTERM	容器停止信号，默认为 SIGTERM
-t, --tty=false	分配一个伪终端
--tmpfs=[]	挂载 tmpfs 目录
-u, --user=""	指定用户名或者用户 ID
--ulimit=[]	ulimit 选项
--userns	指定容器的 user 命名空间
-v, --volume=[]	将主机的一个目录挂载到容器内部，或者在容器中创建一个新卷（e.g. -v /home/test:/home 将主机的/home/test 目录挂载到容器的/home 目录下，-v /tmp 在容器中的根目录下创建 tmp 文件夹，该文件夹可以被其他容器用--volumes-from 选项共享 ）。不支持将主机目录挂载到容器/proc 子目录，否则启动容器会报错。
--volume-driver	设置容器的数据卷驱动，可选。
--volumes-from=[]	将另外一个容器的卷挂载到本容器中，实现卷的共享（e.g. -volumes-from container_name 将 container_name 中的卷挂载到这个容器中 ）。-v 和--volumes-from=[]是两个非常重要的选项用于数据的备份和热迁移。
-w, --workdir=""	指定容器的工作目录，进入容器时的目录

示例：

创建了一个名为 busybox 的容器，创建之后的容器用 docker start 命令启动。

```
$ sudo docker create -ti --name=busybox busybox /bin/bash
```

4.5.2.5 diff

用法：**docker diff CONTAINER**

功能：检视容器的差异，相比于容器刚创建时做了哪些改变

选项：无

示例：

```
$ sudo docker diff registry
C /root
A /root/.bash history
A /test
```

4.5.2.6 exec

用法：**docker exec [OPTIONS] CONTAINER COMMAND [ARG...]**

功能：在一个运行的容器内执行命令

选项：

-d, --detach=false 在后台运行

-i, --interactive=false 保持容器的 STDIN 打开

-t, --tty=false 分配一个虚拟终端

--privileged 以特权模式执行命令

-u, --user 指定用户名或者 UID

示例：

```
$ sudo docker exec -ti exec_test ls
bin  etc  lib      media  opt  root  sbin  sys    tmp  var
dev  home  lib64  mnt   proc  run   srv   test  usr
```

4.5.2.7 export

用法：**docker export CONTAINER**

功能：将一个容器的文件系统内容以 tar 包导出到 STDOUT

选项：无

示例：

将名为 busybox 的容器的内容导出到 busybox.tar 包中：

```
$ sudo docker export busybox > busybox.tar
$ ls
busybox.tar
```

4.5.2.8 inspect

用法：**docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]**

功能：返回一个容器或者镜像的底层信息

选项：

-f, --format=""　　按照给定的格式输出信息

-s, --size　　若查询类型为容器，显示该容器的总体文件大小

--type　　返回指定类型的 JSON 格式

-t, --time=120　超时时间的秒数，若在该时间内 docker inspect 未执行成功，则停止等待并立即报错。默认为 120 秒。

示例：

1. 返回一个容器的信息

```
$ sudo docker inspect busybox_test
[
    {
        "Id": "9fbb8649d5a8b6ae106bb0ac7686c40b3cbd67ec2fd1ab03e0c419a70d755577",
        "Created": "2019-08-28T07:43:51.27745746Z",
        "Path": "bash",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 64177,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2019-08-28T07:43:53.021226383Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
......
```

2. 按照给定格式返回一个容器的指定信息，下面的例子返回 busybox_test 容器 IP 地址

```
$ sudo docker inspect -f {{.NetworkSettings.IPAddress}} busybox test
172.17.0.91
```

4.5.2.9 logs

用法：**docker logs [OPTIONS] CONTAINER**

功能：抓取容器内的日志信息，容器可以使运行状态的也可以是停止状态的

选项：

-f, --follow=false　　实时打印日志信息

-t, --timestamps=false　　显示日志的时间戳

--since　　显示指定时间之后的日志

--tail="all" 设置显示的行数，默认显示所有

示例：

1. 查看 jaegertracing 容器的日志信息，该容器上跑了一个 jaegertracing 服务

```
$ sudo docker logs jaegertracing
{"level":"info","ts":1566979103.3696961,"caller":"healthcheck/handler.go:99","m
sg":"Health Check server started","http-port":14269,"status":"unavailable"}
{"level":"info","ts":1566979103.3820567,"caller":"memory/factory.go:55","msg":"
Memory storage configuration","configuration":{"MaxTraces":0}}
{"level":"info","ts":1566979103.390773,"caller":"tchannel/builder.go:94","msg":
"Enabling service discovery","service":"jaeger-collector"}
{"level":"info","ts":1566979103.3908608,"caller":"peerlistmgr/peer list mgr.go:
111","msg":"Registering active peer","peer":"127.0.0.1:14267"}
{"level":"info","ts":1566979103.3922884,"caller":"all-in-
one/main.go:186","msg":"Starting agent"}
{"level":"info","ts":1566979103.4047635,"caller":"all-in-
one/main.go:226","msg":"Starting jaeger-collector TChannel server","port":14267}
{"level":"info","ts":1566979103.404901,"caller":"all-in-
one/main.go:236","msg":"Starting jaeger-collector HTTP server","http-
port":14268}
{"level":"info","ts":1566979103.4577134,"caller":"all-in-
one/main.go:256","msg":"Listening for Zipkin HTTP traffic","zipkin.http-
port":9411}
```

2. 加上-f 选项，实时打印 jaegertracing 容器的日志信息

```
$ sudo docker logs -f jaegertracing
{"level":"info","ts":1566979103.3696961,"caller":"healthcheck/handler.go:99","m
sg":"Health Check server started","http-port":14269,"status":"unavailable"}
{"level":"info","ts":1566979103.3820567,"caller":"memory/factory.go:55","msg":"
Memory storage configuration","configuration":{"MaxTraces":0}}
{"level":"info","ts":1566979103.390773,"caller":"tchannel/builder.go:94","msg":
"Enabling service discovery","service":"jaeger-collector"}
{"level":"info","ts":1566979103.3908608,"caller":"peerlistmgr/peer list mgr.go:
111","msg":"Registering active peer","peer":"127.0.0.1:14267"}
{"level":"info","ts":1566979103.3922884,"caller":"all-in-
one/main.go:186","msg":"Starting agent"}
```

4.5.2.10 pause/unpause

用法：**docker pause CONTAINER**

docker unpause CONTAINER

功能：这两个命令是配对使用的，docker pause 暂停容器内的所有进程，docker unpause 恢复暂停的进程

选项：无

示例：

本示例将演示一个跑了 docker registry（docker 镜像服务）服务的容器，当使用 docker pause 命令暂停这个容器的进程后，使用 curl 命令访问该 registry 服务将阻塞，使用 docker unpause 命令将恢复 registry 服务，可以用 curl 命令访问。

1. 启动一个 registry 容器

```
$ sudo docker run -d --name pause_test -p 5000:5000 registry
```

此时可以用 curl 命令访问这个服务，请求状态码会返回 200 OK。

```
$ sudo curl -v 127.0.0.1:5000
```

2. 暂停这个容器内的进程

```
$ sudo docker pause pause_test
```

此时用 curl 命令访问这个服务将阻塞，等待服务开启。

3. 恢复运行这个容器内的进程

```
$ sudo docker unpause pause_test
```

此时步骤 2 中的 curl 访问将恢复运行，请求状态码返回 200 OK。

4.5.2.11 port

用法：**docker port CONTAINER [PRIVATE_PORT[/PROTO]]**

功能：列出容器的端口映射，或者查找指定端口在主机的哪个端口

选项：无

示例：

1. 列出容器所有的端口映射

```
$ sudo docker port registry
5000/tcp -> 0.0.0.0.: 5000
```

2. 查找容器指定端口的映射

```
$ sudo docker port registry 5000
0.0.0.0.: 5000
```

4.5.2.12 ps

用法：**docker ps [OPTIONS]**

功能：根据不同的选项列出不同状态的容器，在不加任何选项的情况下，将列出正在运行的容器

选项：

-a, --all=false 显示所用的容器

-f, --filter=[] 筛选值，可用的筛选值有：exited=<int>容器的退出码 status=(restarting|running|paused|exited)容器的状态码（e.g. -f status=running，列出正在运行的容器）

-l, --latest=false 列出最近创建的一个容器

-n=-1 列出最近 n 此创建的容器

--no-trunc=false 将 64 位的容器 ID 全部显示出来，默认显示 12 位容器的 ID

-q, --quiet=false 显示容器的 ID

-s, --size=false   显示容器的大小

示例：

1. 列出正在运行的容器

```
$ sudo docker ps
```

2. 列出所有的容器

```
$ sudo docker ps -a
```

4.5.2.13 rename

用法：**docker rename OLD_NAME NEW_NAME**

功能：重命名容器

示例：

示例中，用 docker run 创建并启动一个容器，使用 docker rename 对容器重命名，并查看容器名是否改变。

```
$ sudo docker ps
CONTAINER ID       IMAGE              COMMAND            CREATED            STATUS
PORTS             NAMES
b15976967abb       busybox:latest       "bash"            3 seconds ago      Up 2
seconds                          festive morse
$ sudo docker rename pedantic euler new name
$ sudo docker ps
CONTAINER ID       IMAGE              COMMAND            CREATED            STATUS
PORTS             NAMES
b15976967abb       busybox:latest       "bash"            34 seconds ago     Up 33
seconds                         new_name
```

4.5.2.14 restart

用法：**docker restart [OPTIONS] CONTAINER [CONTAINER...]**

功能：重启一个运行中的容器

选项：

-t, --time=10   在杀掉容器之前等待容器停止的秒数，如果容器已停止，就重启。默认为 10 秒。

示例：

```
$ sudo docker restart busybox
```

说明

容器在 restart 过程中，如果容器内存在 D 状态或 Z 状态的进程，可能会导致容器重启失败，这需要进一步分析导致容器内进程 D 状态或 Z 状态的原因，待容器内进程 D 状态或 Z 状态解除后，再进行容器 restart 操作。

4.5.2.15 rm

用法：**docker rm [OPTIONS] CONTAINER [CONTAINER...]**

功能：删除一个或多个容器

选项：

-f, --force=false　　强制删除运行中的容器

-l, --link=false　　Remove the specified link and not the underlying container

-v, --volumes=false　　Remove the volumes associated with the container

示例：

1. 删除一个停止运行的容器

```
$ sudo docker rm test
```

2. 删除一个正在运行的容器

```
$ sudo docker rm -f rm_test
```

4.5.2.16 run

用法：**docker run [OPTIONS] IMAGE [COMMAND] [ARG...]**

功能：该命令将由指定的 image（如果指定的 IMAGE 不存在，则从官方镜像库中下载一个镜像）创建一个容器，并启动这个容器，并在容器中执行指定的命令。该命令集成了 docker create 命令、docker start 命令、docker exec 命令。

选项：（该命令的选项与 docker create 命令的选项一样，请参考 docker create 命令选项，仅仅多了以下两个选项）

--rm=false　　设置容器退出时自动删除容器

-v 挂载本地目录或匿名卷到容器内。注意：当将本地目录以带有 selinux 的安全标签的方式挂载到容器内的同时，尽量不要同时做该本地目录的增删操作，否则该安全标签可能不生效

--sig-proxy=true　　发往进程信号的代理，SIGCHLD, SIGSTOP, SIGKILL 不使用代理

示例：

使用 busybox 镜像运行一个容器，在容器启动后执行/bin/sh

```
$ sudo docker run -ti busybox /bin/sh
```

4.5.2.17 start

用法：**docker start [OPTIONS] CONTAINER [CONTAINER...]**

功能：启动一个或多个未运行容器

选项：

-a, --attach=false　　容器的标准输出和错误输出附加到 host 的 STDOUT 和 STDERR 上

-i, --interactive=false　　容器的标准输入附加到 host 的 STDIN 上

实例：

启动一个名为busybox的容器，添加-i -a选项附加标准输入输出，容器启动后直接进入容器内部，输入exit可以退出容器。

如果启动容器时不加-i -a选项，容器将在后台启动。

```
$ sudo docker start -i -a busybox
```

4.5.2.18 stats

用法：**docker stats [OPTIONS] [CONTAINER...]**

功能：持续监控并显示指定容器（若不指定，则默认全部容器）的资源占用情况

选项：

-a, --all　　显示所有容器（默认仅显示运行状态的容器）

--no-stream　　只显示第一次的结果，不持续监控

示例：

示例中，用docker run创建并启动一个容器，docker stats将输出容器的资源占用情况。

```
$ sudo docker stats
CONTAINER ID        NAME                    CPU %               MEM USAGE / LIMIT     MEM %
NET I/O             BLOCK I/O           PIDS
2e242bcdd682        jaeger                  0.00%               77.08MiB / 125.8GiB   0.06%
42B / 1.23kB        97.9MB / 0B         38
02a06be42b2c        relaxed chandrasekhar   0.01%               8.609MiB / 125.8GiB
0.01%               0B / 0B             0B / 0B             10
deb9e49fdef1        hardcore montalcini     0.01%               12.79MiB / 125.8GiB
0.01%               0B / 0B             0B / 0B             9
```

4.5.2.19 stop

用法：**docker stop [OPTIONS] CONTAINER [CONTAINER...]**

功能：通过向容器发送一个SIGTERM信号并在一定的时间后发送一个SIGKILL信号停止容器

选项：

-t, --time=10　　在杀掉容器之前等待容器退出的秒数，默认为10S

示例：

```
$ sudo docker stop -t=15 busybox
```

4.5.2.20 top

用法：**docker top CONTAINER [ps OPTIONS]**

功能：显示一个容器内运行的进程

选项：无

示例：

先运行了一个名为 top_test 的容器，并在其中执行了 top 指令

```
$ sudo docker top top test
UID              PID              PPID             C                STIME
TTY              TIME             CMD
root             70045            70028            0                15:52
pts/0            00:00:00         bash
```

显示的 PID 是容器内的进程在主机中的 PID 号。

4.5.2.21 update

用法：**docker update [OPTIONS] CONTAINER [CONTAINER...]**

功能：热变更一个或多个容器配置。

选项：

表4-3 参数说明

参数	参数含义
--accel=[]	设置容器加速器，可设置一个或多个
--blkio-weight	设置容器 blockio 的相对权重，从 10 到 1000
--cpu-shares	设置容器获得主机 CPU 的相对权重，通过设置这个选项获得更高的优先级，默认所有的容器都是获得相同的 CPU 优先权。
--cpu-period	设置 CFS（完全公平调度策略）进程的 CPU 周期。 默认值为 100ms；一般--cpu-period 参数和--cpu-quota 是配合使用的，比如--cpu-period=50000 --cpu-quota=25000，意味着如果有 1 个 CPU，该容器可以每 50ms 获取到 50% 的 CPU。
--cpu-quota	设置 CFS(完全公平调度策略)进程的 CPU 配额，默认为 0，即没有限制
--cpuset-cpus	设置容器中进程允许运行的 CPU (0-3, 0,1)。默认没有限制
--cpuset-mems	设置容器中进程运行运行的内存内存节点 (0-3, 0,1)，只对 NUMA 系统起作用
--kernel-memory=""	设置容器的 kernerl 内存限制，格式<number><optional unit>，其中 unit = b, k, m or g
-m, --memory=""	设置容器的内存限制，格式<number><optional unit>，其中 unit = b, k, m or g。该参数最小值为 4m。
--memory-reservation	设置容器内存限制，默认与--memory 一致。可认为--memory 是硬限制，--memory-reservation 是软限制；当使

参数	参数含义
	用内存超过预设值时，会动态调整（系统回收内存时尝试将使用内存降低到预设值以下），但不确保一定不超过预设值。一般可以和--memory 一起使用，数值小于--memory 的预设值。
--memory-swap	设置普通内存和交换分区的使用总量，-1 为不做限制。如果不设置，则为--memory 值的 2 倍，即 SWAP 可再使用与--memory 相同的内存量。
--restart=""	设置容器退出时候的重启规则，当前 1.3.1 版本支持 3 个规则： • no：当容器停止时，不重启。 • on-failure：当容器退出码为非 0 时重启容器，这个规则可以附加最大重启次数，如 on-failure:5，最多重启 5 次。 • always：无论退出码是什么都退出。
--help	打印 help 信息

示例：

变更一个容器名为 busybox 的 cpu 和 mem 配置，包括容器获得主机 CPU 的相对权重值为 512，容器中进程允许运行的 CPU 核心为 0,1,2,3，容器运行内存限制为 512m。

```
$ sudo docker update  --cpu-shares 512  --cpuset-cpus=0,3 --memory 512m ubuntu
```

4.5.2.22 wait

用法：**docker wait CONTAINER [CONTAINER...]**

功能：等待一个容器停止，并打印出容器的退出码

选项：无

示例：

先开启一个名为 busybox 的容器

```
$ sudo docker start -i -a busybox
```

执行 docker wait

```
$ sudo docker wait busybox
0
```

将阻塞等待 busybox 容器的退出，退出 busybox 容器后将看到打印退出码“0”。

4.5.3 镜像管理

4.5.3.1 build

用法：**docker build [OPTIONS] PATH | URL | -**

功能：使用指定路径中的 Dockerfile 生成构建一个新的 image

选项：常用选项参数如下，更多选项可以查看 docker help build

表4-4 参数说明

参数	参数含义
--force-rm=false	即使没有构建成功也删除构建过程中生成的容器
--no-cache=false	构建 image 的过程中不使用缓存
-q, --quiet=false	禁止构建过程中产生的冗余信息
--rm=true	构建成功后删除过程中生成的容器
-t, --tag=""	指定构建生成的 image 的 tag 名
--build-arg=[]	设置构建参数
--label=[]	镜像相关参数设置，各参数意义与 create 类似
--isolation	指定容器的隔离方法
--pull	构建时总是尝试获取最新版本镜像

Dockerfile 介绍：

Dockerfile 是一个镜像的表示，可以通过 Dockerfile 来描述构建镜像的步骤，并自动构建一个容器，所有的 Dockerfile 命令格式都是：**INSTRUCTION arguments**

FROM 命令

格式：FROM <image> 或 FROM <image>:<tag>

功能：该命令指定基本镜像，是所有 Dockerfile 文件的第一个命令，如果没有指定基本镜像的 tag，使用默认 tag 名 latest。

RUN 命令

格式：RUN <command> (the command is run in a shell - `/bin/sh -c`) 或者

RUN ["executable", "param1", "param2" ...] (exec form)

功能：RUN 命令会在上面 FROM 指定的镜像里执行指定的任何命令，然后提交(commit)结果，提交的镜像会在后面继续用到。RUN 命令等价于:

docker run image command

docker commit container_id

注释

使用#注释

MAINTAINER 命令

格式：MAINTAINER <name>

功能：命令用来指定维护者的姓名和联系方式

ENTRYPOINT 命令

格式：ENTRYPOINT cmd param1 param2 ... 或者 ENTRYPOINT ["cmd", "param1", "param2"...]

功能：设置在容器启动时执行命令

USER 命令

格式：USER name

功能：指定 memcached 的运行用户

EXPOSE 命令

格式：EXPOSE <port> [<port>...]

功能：开放镜像的一个或多个端口

ENV 命令

格式：ENV <key> <value>

功能：设置环境变量，设置了后，后续的 RUN 命令都可以使用

ADD 命令

格式：ADD <src> <dst>

功能：从 src 复制文件到 container 的 dest 路径，<src> 是相对被构建的源目录的相对路径，可以是文件或目录的路径，也可以是一个远程的文件 url，<dest> 是 container 中的绝对路径

VOLUME 命令

格式：VOLUME ["<mountpoint>"]

功能：创建一个挂载点用于共享目录

WORKDIR 命令

格式：workdir <path>

功能：配置 RUN, CMD, ENTRYPOINT 命令设置当前工作路径可以设置多次，如果是相对路径，则相对前一个 WORKDIR 命令

CMD 命令

格式：CMD ["executable","param1","param2"] (like an exec, preferred form)

CMD ["param1","param2"] (as default parameters to ENTRYPOINT)

CMD command param1 param2 (as a shell)

功能：一个 Dockerfile 里只能有一个 CMD，如果有多个，只有最后一个生效

ONBUILD 命令

格式：ONBUILD [其它指令]

功能：后面跟其它指令，比如 RUN、COPY 等，这些指令，在当前镜像构建时并不会被执行，只有当以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行

下面是 Dockerfile 的一个完整例子，该 Dockerfile 将构建一个安装了 sshd 服务的 image

```
FROM busybox
ENV  http proxy http://192.168.0.226:3128
ENV  https proxy https://192.168.0.226:3128
RUN apt-get update && apt-get install -y openssh-server
RUN mkdir -p /var/run/sshd
EXPOSE 22
ENTRYPOINT /usr/sbin/sshd -D
```

示例：

1. 以上文的 Dockerfile 构建一个 image

```
$ sudo docker build -t busybox:latest
```

2. 通过以下命令可以看到这个生成的 image：

```
docker images | grep busybox
```

4.5.3.2 history

用法：**docker history [OPTIONS] IMAGE**

功能：显示一个 image 的变化历史

选项：

-H, --human=true

--no-trunc=false　　不对输出进行删减

-q, --quiet=false　　只显示 ID

示例：

```
$ sudo docker history busybox:test
IMAGE             CREATED          CREATED BY         SIZE              COMMENT
be4672959e8b       15 minutes ago     bash               23B
21970dfada48       4 weeks ago                          128MB              Imported
from -
```

4.5.3.3 images

用法：**docker images [OPTIONS] [NAME]**

功能：列出存在的 image，不加选项时不显示中间的 image

选项：

-a, --all=false　　显示所有的镜像，

-f, --filter=[]　　指定一个过滤值(i.e. 'dangling=true')

--no-trunc=false　　不对输出进行删减

-q, --quiet=false　　只显示 ID

示例：

```
$ sudo docker images
REPOSITORY         TAG                IMAGE ID          CREATED           SIZE
busybox           latest             e02e811dd08f       2 years ago        1.09MB
```

4.5.3.4 import

用法：**docker import URL|- [REPOSITORY[:TAG]]**

功能：把包含了一个 rootfs 的 tar 包导入为镜像。与 docker export 相对应。

选项：无

示例：

从上文介绍的 docker export 命令时导出的 busybox.tar 用 docker import 命令生成一个新的 image

```
$ sudo docker import busybox.tar busybox:test
sha256:a79d8ae1240388fd3f6c49697733c8bac4d87283920defc51fb0fe4469e30a4f
$ sudo docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox            test                a79d8ae12403        2 seconds ago        1.3MB
```

4.5.3.5 load

用法：**docker load [OPTIONS]**

功能：把 docker save 出来的 tar 包重新加载一个镜像。与 docker save 相对应。

选项：

-i, --input=""

示例：

```
$ sudo docker load -i busybox.tar
Loaded image ID:
sha256:e02e811dd08fd49e7f6032625495118e63f597eb150403d02e3238af1df240ba
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
busybox            latest              e02e811dd08f        2 years ago          1.09MB
```

4.5.3.6 login

用法：**docker login [OPTIONS] [SERVER]**

功能：登录到一个镜像服务库，没有指定 server 时，默认登录到 https://index.docker.io/v1/

选项：

-e, --email="" Email

-p, --password="" 密码

-u, --username="" 用户名

示例：

```
$ sudo docker login
```

4.5.3.7 logout

用法：**docker logout [SERVER]**

功能：从一个镜像服务器中登出，没有指定 server 时，默认登出 https://index.docker.io/v1/

选项：无

示例：

```
$ sudo docker logout
```

4.5.3.8 pull

用法：**docker pull [OPTIONS] NAME[:TAG]**

功能：从一个镜像库（官方的或私有的）中拉取一个镜像

选项：

-a, --all-tags=false 下载一个镜像仓库的所有镜像（一个镜像仓库可以被打多个标签，比如一个 busybox 镜像库，可能有多个标签如 busybox:14.04,busybox:13.10,busybox:latest 等，使用-a 选项后，将所有标签的 busybox 镜像拉取下来）

示例：

1. 从官方镜像库中拉取 nginx 镜像

```
$ sudo docker pull nginx
Using default tag: latest
latest: Pulling from official/nginx
94ed0c431eb5: Pull complete
9406c100a1c3: Pull complete
aa74daafd50c: Pull complete
Digest: sha256:788fa27763db6d69ad3444e8ba72f947df9e7e163bad7c1f5614f8fd27a311c3
Status: Downloaded newer image for nginx:latest
```

   拉取镜像时会检测所依赖的层是否存在，如果存在就用本地的层。

2. 从私有镜像库中拉取镜像

   从私有镜像库中拉取 Fedora 镜像，比如所使用的私有镜像库的地址是 192.168.1.110:5000：

```
$ sudo docker pull 192.168.1.110:5000/fedora
```

4.5.3.9 push

用法：**docker push NAME[:TAG]**

功能：将一个 image 推送到镜像库中

选项：无

示例：

1. 将一个 image 推送到私有镜像库 192.168.1.110:5000 中
2. 将要推送的镜像打标签（docker tag 命令将在下文介绍），本例中要推送的镜像为 busybox:sshd

```
$ sudo docker tag ubuntu:sshd 192.168.1.110:5000/busybox:sshd
```

3. 将打好标签的镜像推送到私有镜像库中

```
$ sudo docker push 192.168.1.110:5000/busybox:sshd
```

   推送的时候会自动检测所依赖的层在镜像库中是否已存在，如果以存在，跳过该层。

4.5.3.10 rmi

用法：**docker rmi [OPTIONS] IMAGE [IMAGE...]**

功能：删除一个或多个镜像，如果一个镜像在镜像库中有多个标签，删除镜像的时候只是进行 untag 操作，当删除的是只有一个标签的镜像时，将依次删除所依赖的层。

选项：

-f, --force=false　　强制删除 image

--no-prune=false　　不删除没有标签的父镜像

示例：

```
$ sudo docker rmi 192.168.1.110:5000/busybox:sshd
```

4.5.3.11 save

用法：**docker save [OPTIONS] IMAGE [IMAGE...]**

功能：保存一个 image 到一个 tar 包，输出默认是到 STDOUT

选项：

-o, --output=""　　输出到文件中而不是 STDOUT

示例：

```
$ sudo docker save -o nginx.tar nginx:latest
$ ls
nginx.tar
```

4.5.3.12 search

用法：**docker search [OPTIONS] TERM**

功能：在镜像库中查找特定的镜像

选项：

--automated=false　　显示自动构建的 image

--no-trunc=false　　不对输出进行删减

-s, --stars=0　　只显示特定星级以上的 image

示例：

1. 在官方镜像库中搜寻 nginx

```
$ sudo docker search nginx
NAME                             DESCRIPTION                                      STARS
OFFICIAL          AUTOMATED
nginx                            Official build of Nginx.                           11873
[OK]
jwilder/nginx-proxy                Automated Nginx reverse proxy for docker con…
1645                                 [OK]
richarvey/nginx-php-fpm             Container running Nginx + PHP-FPM capable of…
739                                  [OK]
linuxserver/nginx                  An Nginx container, brought to you by LinuxS…
74
bitnami/nginx                     Bitnami nginx Docker Image                         70
[OK]
tiangolo/nginx-rtmp                Docker image with Nginx using the nginx-rtmp…
51                                   [OK]
```

2. 在私有镜像库中搜寻 busybox，在私有镜像库中搜寻时要加上私有镜像库的地址

```
$ sudo docker search 192.168.1.110:5000/busybox
```

4.5.3.13 tag

用法：**docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]**

功能：将一个镜像打标签到一个库中

选项：

-f, --force=false　　如果存在相同的 tag 名将强制替换原来的 image

示例：

```
$ sudo docker tag busybox:latest busybox:test
```

4.5.4 统计信息

4.5.4.1 events

用法：**docker events [OPTIONS]**

功能：从 docker daemon 中获取实时事件

选项：

--since=""　　显示指定时间戳之后的事件

--until=""　　显示直到指定之间戳的事件

示例：

该示例中，执行 docker events 后，用 docker run 创建并启动一个容器，docker events 将输出 create 事件和 start 事件。

```
$ sudo docker events
2019-08-28T16:23:09.338838795+08:00 container create
53450588a20800d8231aa1dc4439a734e16955387efb5f259c47737dba9e2b5e
(image=busybox:latest, name=eager_wu)
2019-08-28T16:23:09.339909205+08:00 container attach
53450588a20800d8231aa1dc4439a734e16955387efb5f259c47737dba9e2b5e
(image=busybox:latest, name=eager_wu)
2019-08-28T16:23:09.397717518+08:00 network connect
e2e20f52662f1ee2b01545da3b02e5ec7ff9c85adf688dce89a9eb73661dedaa
(container=53450588a20800d8231aa1dc4439a734e16955387efb5f259c47737dba9e2b5e,
name=bridge, type=bridge)
2019-08-28T16:23:09.922224724+08:00 container start
53450588a20800d8231aa1dc4439a734e16955387efb5f259c47737dba9e2b5e
(image=busybox:latest, name=eager_wu)
2019-08-28T16:23:09.924121158+08:00 container resize
53450588a20800d8231aa1dc4439a734e16955387efb5f259c47737dba9e2b5e (height=48,
image=busybox:latest, name=eager_wu, width=210)
```

4.5.4.2 info

用法：**docker info**

功能：显示 docker 系统级的相关信息，包括系统中的 Container 数量、Image 数量、Image 的存储驱动、容器的执行驱动、内核版本、主机操作系统版本等信息。

选项：无

示例：

```
$ sudo docker info
Containers: 4
 Running: 3
 Paused: 0
 Stopped: 1
Images: 45
Server Version: 18.09.0
Storage Driver: devicemapper
 Pool Name: docker-thinpool
 Pool Blocksize: 524.3kB
 Base Device Size: 10.74GB
 Backing Filesystem: ext4
 Udev Sync Supported: true
 Data Space Used: 11GB
 Data Space Total: 51GB
 Data Space Available: 39.99GB
 Metadata Space Used: 5.083MB
 Metadata Space Total: 532.7MB
 Metadata Space Available: 527.6MB
 Thin Pool Minimum Free Space: 5.1GB
 Deferred Removal Enabled: true
 Deferred Deletion Enabled: true
 Deferred Deleted Device Count: 0
......
```

4.5.4.3 version

用法：**docker version**

功能：显示 docker 的版本信息，包括 Client 版本、Server 版本、Go 版本、OS/Arch 等信息

选项：无

示例：

```
$ sudo docker version
Client:
 Version:         18.09.0
 EulerVersion:     18.09.0.48
 API version:      1.39
 Go version:       go1.11
 Git commit:       cbf6283
 Built:           Mon Apr  1 00:00:00 2019
```

```
 OS/Arch:          linux/arm64
 Experimental:      false

Server:
 Engine:
  Version:         18.09.0
  EulerVersion:     18.09.0.48
  API version:      1.39 (minimum version 1.12)
  Go version:       go1.11
  Git commit:       cbf6283
  Built:           Mon Apr  1 00:00:00 2019
  OS/Arch:         linux/arm64
  Experimental:     false
```