LXD

LXD contributors

2022 年 04 月 07 日

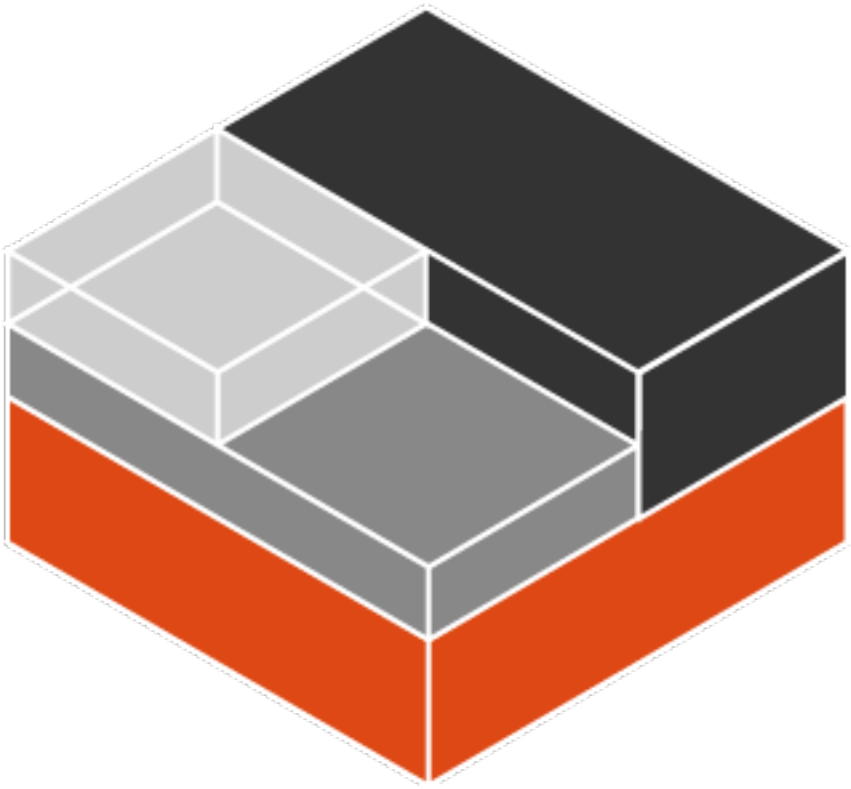
目次

第 1 章　セキュリティ 3

第 2 章　サポート 5

第 3 章　コントリビュート 7

3.1 LXD を使い始めるには 7
3.2 設定 27
3.3 イメージ 123
3.4 オペレーション 133
3.5 REST API 159
3.6 内部動作とデバッグ 230
3.7 外部リソース 244



LXD は、次世代のシステムコンテナおよび仮想マシンマネージャです。コンテナや仮想マシンの中で動作する完全な Linux システムに統一されたユーザーエクスペリエンスを提供します。

LXD はイメージベースで、多くの Linux ディストリビューションに対応しています。そして、非常にパワフルでありながら、非常にシンプルな REST API を中心に構築されています。

LXD とは何か、何ができるのか、より良いアイデアを得るためには、オンラインで試すことができます! また、ローカルで動作させたい場合は、LXD を使い始めるにはをご覧ください。

- リリースのアナウンス: https://linuxcontainers.org/ja/lxd/news/
- リリースの tarball: https://linuxcontainers.org/ja/lxd/downloads/
- ドキュメント: https://lxd-ja.readthedocs.io/ja/latest/

第 1 章

セキュリティ

LXD のインストールが安全であることを保証するために、以下の点を考慮してください。

- オペレーティングシステムを最新に保ち、利用可能なすべてのセキュリティパッチをインストールする。
- サポートされている LXD のバージョン（LTS リリースまたは月例機能リリース）のみを使用する。
- LXD デーモンとリモート API へのアクセスを制限すること。
- 必要とされない限り、特権コンテナを使わないこと。特権的なコンテナを使う場合は、適切なセキュリティ対策をしてください。詳細は LXC セキュリティページを参照してください。
- ネットワークインターフェイスを安全に設定してください。

詳細は**セキュリティ**をご覧ください。

重要: UNIX ソケットを介した LXD へのローカルアクセスは、常に LXD へのフルアクセスを許可します。これは、任意のインスタンス上のセキュリティ機能を変更できる能力に加えて、任意のインスタンスにファイルシステムパスやデバイスをアタッチする能力を含みます。

したがって、あなたのシステムへのルートアクセスを信頼できるユーザーにのみ、このようなアクセスを与えるべきです。

第 2 章

サポート

助けを求める方法については、**サポート**を参照してください。

第 3 章

コントリビュート

修正や新機能の提供は大歓迎です。詳しくは**コントリビュート**をご覧ください。

3.1 LXD を使い始めるには

このセクションのドキュメントに加えて、ウェブサイト上の LXD を使い始めるには を参照してください。

3.1.1 動作環境

LXD は Go 1.18 以上を必要とし、 golang のコンパイラのみでテストされています。(訳注: 以前は gccgo もサポートされていましたが golang のみになりました)

ビルドには最低 2GB の RAM を推奨します。

必要なカーネルバージョン

サポートされる最小のカーネルバージョンは 5.4 です。

LXD には以下の機能をサポートするカーネルが必要です。

- Namespaces (pid, net, uts, ipc と mount)
- Seccomp

以下のオプションの機能はさらなるカーネルオプションを必要とします。

- Namespaces (user と cgroup)
- AppArmor (mount mediation に対する Ubuntu パッチを含む)
- Control Groups (blkio, cpuset, devices, memory, pids と net_prio)

- CRIU (正確な詳細は CRIU のアップストリームを参照のこと)

さらに使用している LXC のバージョンで必要とされる他のカーネルの機能も必要です。

LXC

LXD は以下のビルドオプションでビルドされた LXC 4.0.0 以上を必要とします。

- apparmor (もし LXD の apparmor サポートを使用するのであれば)
- seccomp

Ubuntu を含む、さまざまなディストリビューションの最近のバージョンを動かすためには、LXCFS もインストールする必要があります。

QEMU

仮想マシンを利用するには QEMU 6.0 以降が必要です。

追加のライブラリー (と開発用のヘッダ)

LXD はデータベースとして `dqlite` を使用しています。ビルドしセットアップするためには `make deps` を実行してください。

LXD は他にもいくつかの (たいていはパッケージ化されている) C ライブラリーを使用しています。

- libacl1
- libcap2
- liblz4 (`dqlite` で使用)
- libuv1 (`dqlite` で使用)
- libsqlite3 >= 3.25.0 (`dqlite` で使用)

ライブラリーそのものとライブラリーの開発用ヘッダ (-dev パッケージ) の全てをインストールしたことを確認してください。

3.1.2 LXD のインストール

LXD をインストールする最も簡単な方法は提供されているパッケージのどれかをインストールすることですが、ソースから LXD をインストールすることもできます。

パッケージからの LXD のインストール

LXD のデーモンは Linux でしか動作しませんが、クライアントツール (`lxc`) はほとんどのプラットフォームで利用可能です。

OS	フォーマット	コマンド
Linux	Snap	snap install lxd
Windows	Chocolatey	choco install lxc
MacOS	Homebrew	brew install lxc

様々な Linux ディストリビューションや OS への LXD のインストールについては、私たちのウェブサイトに詳しい説明があります。

LXD のソースからのインストール

LXD の開発には liblxc の最新バージョン（4.0.0 以上が必要）を使用することをおすすめします。さらに LXD が動作するためには Golang 1.16 以上が必要です。Ubuntu では次のようにインストールできます:

```
sudo apt update
sudo apt install acl attr autoconf dnsmasq-base git golang libacl1-dev libcap-dev␣
↪liblxc1 liblxc-dev libsqlite3-dev libtool libudev-dev liblz4-dev libuv1-dev make pkg-
↪config rsync squashfs-tools tar tcl xz-utils ebtables
```

デフォルトのストレージバックエンドである "directory" に加えて、LXD ではいくつかのストレージバックエンドが使えます。これらのツールをインストールすると、initramfs への追加が行われ、ホストのブートが少しだけ遅くなるかもしれませんが、特定のバックエンドを使いたい場合には必要です:

```
sudo apt install lvm2 thin-provisioning-tools
sudo apt install btrfs-progs
```

テストスイートを実行するには、次のパッケージも必要です:

```
sudo apt install curl gettext jq sqlite3 uuid-runtime socat bind9-dnsutils
```

ソースからの最新版のビルド

この方法は LXD の最新版をビルドしたい開発者や Linux ディストリビューションで提供されない LXD の特定のリリースをビルドするためのものです。 Linux ディストリビューションへ統合するためのソースからのビルドはここでは説明しません。それは将来、別のドキュメントで取り扱うかもしれません。

```
git clone https://github.com/lxc/lxd
cd lxd
```

これで LXD の現在の開発ツリーをダウンロードしてソースツリー内に移動します。その後下記の手順にしたがって実際に LXD をビルド、インストールしてください。

ソースからのリリース版のビルド

LXD のリリース tarball は完全な依存ツリーと libraft と LXD のデータベースのセットアップに使用する libdqlite のローカルコピーをバンドルしています。

```
tar zxvf lxd-4.18.tar.gz
cd lxd-4.18
```

これでリリース tarball を解凍し、ソースツリー内に移動します。その後下記の手順にしたがって実際に LXD をビルド、インストールしてください。

ビルドの開始

実際のビルドは Makefile の 2 回の別々の実行により行われます。 1 つは `make deps` でこれは LXD に必要とされるライブラリーをビルドします。もう 1 つは `make` で LXD 自体をビルドします。 `make deps` の最後に `make` の実行に必要な環境変数を設定するための手順が表示されます。新しいバージョンの LXD がリリースされたらこれらの環境変数の設定は変わるかもしれませんので、 `make deps` の最後に表示された手順を使うようにしてください。下記の手順（例示のために表示します）はあなたがビルドする LXD のバージョンのものとは一致しないかもしれません。

ビルドには最低 2GB の RAM を搭載することを推奨します。

```
make deps
# `make deps` が出力した export のコマンド列を使って環境変数を設定してください。
# 例:
#  export CGO_CFLAGS="${CGO_CFLAGS} -I$(go env GOPATH)/deps/dqlite/include/ -I$(go env␣
↪GOPATH)/deps/raft/include/"
#  export CGO_LDFLAGS="${CGO_LDFLAGS} -L$(go env GOPATH)/deps/dqlite/.libs/ -L$(go env␣
↪GOPATH)/deps/raft/.libs/"
```

(次のページに続く)

(前のページからの続き)

```
#  export LD_LIBRARY_PATH="$(go env GOPATH)/deps/dqlite/.libs/:$(go env GOPATH)/deps/
↪raft/.libs/:${LD_LIBRARY_PATH}"
#  export CGO_LDFLAGS_ALLOW="(-Wl,-wrap,pthread_create)|(-Wl,-z,now)"
make
```

ソースからのビルド結果のインストール

ビルドが完了したら、ソースツリーを維持したまま、あなたのお使いのシェルのパスに `$(go env GOPATH)/bin` を追加し `LD_LIBRARY_PATH` 環境変数を `make deps` で表示された値に設定すれば、 LXD が利用できます。 `~/.bashrc` ファイルの場合は以下のようになります。

```
export PATH="${PATH}:$(go env GOPATH)/bin"
export LD_LIBRARY_PATH="$(go env GOPATH)/deps/dqlite/.libs/:$(go env GOPATH)/deps/raft/.
↪libs/:${LD_LIBRARY_PATH}"
```

これで `lxd` と `lxc` コマンドの実行ファイルが利用可能になり LXD をセットアップするのに使用できます。 `LD_LIBRARY_PATH` 環境変数のおかげで実行ファイルは `$(go env GOPATH)/deps` にビルドされた依存ライブラリーを自動的に見つけて使用します。

マシンセットアップ

LXD が非特権コンテナを作成できるように、root ユーザーに対する sub{u,g}id の設定が必要です:

```
echo "root:1000000:1000000000" | sudo tee -a /etc/subuid /etc/subgid
```

これでデーモンを実行できます（sudo グループに属する全員が LXD とやりとりできるように `--group sudo` を指定します。別に指定したいグループを作ることもできます）:

```
sudo -E PATH=${PATH} LD_LIBRARY_PATH=${LD_LIBRARY_PATH} $(go env GOPATH)/bin/lxd --group␣
↪sudo
```

注：*newuidmap/newgidmap* ツールがシステムに存在し、*/etc/subuid*、*/etc/subgid* が存在する場合は、*root* ユーザーに少なくとも *10M* の *uid/gid* の連続した範囲を許可するように設定する必要があります。

3.1.3 よく聞かれる質問 (FAQ)

LXD サーバをリモートからアクセス可能にするには？

デフォルトでは LXD サーバはローカルの unix ソケットのみをリッスンしているためネットワークからはアクセス可能ではありません。リッスンする追加のアドレスを指定することでネットワークから LXD を利用可能にできます。これは core.https_address 設定で実現できます。

現状のサーバ設定を確認するには、以下のコマンドを実行します。

```
lxc config show
```

リッスンするアドレスを設定するには、まず利用可能なアドレスを調べた上で、次にサーバで config set コマンドを実行します。

```
ip addr
lxc config set core.https_address 192.168.1.15
```

リモート *API* へのアクセス も参照してください。

https 経由で lxc remote add を実行したらパスワードを聞かれたがどうすればよいか？

デフォルトではセキュリティ上の理由から LXD はパスワードを設定していないため、 lxc remote add でリモートは追加できません。パスワードを設定するには LXD が実行中のホスト上で以下のコマンドを実行します。

```
lxc config set core.trust_password SECRET
```

これでリモートパスワードが設定されるので、 lxc remote add 実行時にこのパスワードを使用できます。

あるいはクライアント証明書を .config/lxc/client.crt からサーバにコピーして以下のコマンドで追加すれば、パスワードを設定しなくてもサーバにアクセスできます。

```
lxc config trust add client.crt
```

詳細は **リモート *API* 認証** を参照してください。

LXD のストレージを設定するには？

LXD は btrfs, ceph, directory, lvm と zfs ベースのストレージをサポートします。

まず、あなたが選択したファイルシステムに関連するツール（btrfs-progs, lvm2 あるいは zfsutils-linux）をマシーン上にインストールしてください。

（訳注：LXD をインストールしただけの）デフォルトの状態では LXD はネットワークやストレージが設定されていません。以下のコマンドにより基本の設定を実行できます。

```
lxd init
```

`lxd init` はディレクトリーベースのストレージと ZFS の両方をサポートします。それ以外のストレージを使いたい場合は `lxc storage` コマンドを使う必要があります。

```
lxc storage create default BACKEND [OPTIONS...]
lxc profile device add default root disk path=/ pool=default
```

BACKEND は `btrfs`, `ceph`, `dir`, `lvm`, `zfs` のいずれかです。

明示的に指定しない場合、 LXD は妥当なデフォルトサイズでループデバイスをベースにしたストレージをセットアップします。

本番環境ではパフォーマンスと信頼性の両方の理由でブロックデバイスをベースにしたストレージを使うべきです。

LXD を使ってコンテナをマイグレートするには？

ライブマイグレーションには CRIU と呼ばれるツールを両方のホストにインストールする必要があります。Ubuntu では以下のコマンドでインストールできます。

```
sudo apt install criu
```

次に以下のコマンドでコンテナを起動します。

```
lxc launch ubuntu SOME-NAME
sleep 5s # コンテナの起動が完了するのをを待ちます。
lxc move host1:SOME-NAME host2:SOME-NAME
```

これでコンテナがマイグレートされるはずです。マイグレーションはいまだ実験的な段階にあり環境によっては動かないかもしれないことに注意してください。動かない場合は lxc-devel メーリングリストに報告してください。そうすれば私たちが必要に応じて CRIU メーリングリストに報告します。

自分のホームディレクトリをコンテナ内にバインドマウントできますか？

はい。ディスクデバイスを使って以下のようにすればできます。

```
lxc config device add container-name home disk source=/home/${USER} path=/home/ubuntu
```

非特権コンテナの場合は、以下のいずれかも必要です。

- `lxc config device add` の実行に `shift=true` を指定する。これは `shiftfs` がサポートされているかに依存します（`lxc info` 参照）。
- raw.idmap エントリーを使用する（**ユーザー名前空間** *(user namespace)* **用の** *ID* **のマッピング** 参照）。
- マウントしたいホームディレクトリに再帰的な POSIX ACL を設定する。

上記のいずれかを実行すればコンテナ内のユーザーは read/write パーミッションに沿ってアクセス可能です。上記のいずれも設定しない場合、アクセスしようとすると UID/GID (65536:65536) のオーバーフローが発生し、全ユーザーで読み取り可能 (world readable) 以外のファイルへのアクセスは失敗します。

特権コンテナではコンテナ内の UID/GID が外部と同じなためこの問題はありません。しかしこれは特権コンテナのセキュリティの問題のほとんどの原因でもあります。

LXD コンテナ内で Docker を動かすには？

LXD のコンテナ内で Docker を動かすにはコンテナの `security.nesting` プロパティを `true` にする必要があります。

```
lxc config set <container> security.nesting true
```

LXD コンテナはカーネルモジュールをロードすることはできないので、お使いの Docker の設定によっては、ホストで追加のカーネルモジュールをロードする必要があるかもしれないことに注意してください。

コンテナが必要とするカーネルモジュールのカンマ区切りリストを以下のコマンドで指定すればホストでそれらのモジュールをロードできます。

```
lxc config set <container> linux.kernel_modules <modules>
```

コンテナ内に `/.dockerenv` ファイルを作成するとネストした環境内で実行しているために発生するエラーを Docker が無視するようにできるという報告もあります。

コンテナの起動に関する問題

もしコンテナが起動しない場合や、期待通りの動きをしない場合に最初にすべきことは、コンテナが生成したコンソールログを見ることです。これには `lxc console --show-log CONTAINERNAME` コマンドを使います。

次の例では、`systemd` が起動しない RHEL 7 システムを調べています。

```
# lxc console --show-log systemd
Console log:

Failed to insert module 'autofs4'
Failed to insert module 'unix'
Failed to mount sysfs at /sys: Operation not permitted
Failed to mount proc at /proc: Operation not permitted
[!!!!!!] Failed to mount API filesystems, freezing.
```

ここでのエラーは、`/sys` と `/proc` がマウントできないというエラーです。これは非特権コンテナでは正しい動きです。しかし、LXD は **可能であれば** 自動的にこれらのファイルシステムをマウントします。

コンテナの要件 では、コンテナには `/sbin/init` が存在するだけでなく、空の `/dev`、`/proc`、`/sys` フォルダーが存在していなければならないと定められています。もしこれらのフォルダーが存在しなければ、LXD はこれらをマウントできません。そして、systemd がこれらをマウントしようとします。非特権コンテナでは、systemd はこれを行う権限はなく、フリーズしてしまいます。

何かが変更される前に環境を見ることはできます。`raw.lxc` 設定パラメーターを使って、明示的にコンテナ内の init を変更できます。これは Linux カーネルコマンドラインに `init=/bin/bash` を設定するのと同じです。

```
lxc config set systemd raw.lxc 'lxc.init.cmd = /bin/bash'
```

次のようになります:

```
root@lxc-01:~# lxc config set systemd raw.lxc 'lxc.init.cmd = /bin/bash'
root@lxc-01:~# lxc start systemd
root@lxc-01:~# lxc console --show-log systemd

Console log:

[root@systemd /]#
root@lxc-01:~#
```

コンテナが起動しましたので、期待通りにコンテナ内で動いていないことを確認できます。

```
root@lxc-01:~# lxc exec systemd bash
[root@systemd ~]# ls
[root@systemd ~]# mount
mount: failed to read mtab: No such file or directory
[root@systemd ~]# cd /
[root@systemd /]# ls /proc/
sys
[root@systemd /]# exit
```

LXD は自動修復を試みますので、起動時に作成されたフォルダもあります。コンテナをシャットダウンして再起動すると問題は解決されます。しかし問題の根源は依然として存在しています。**テンプレートに必要なファイルが含まれていないという問題です**。

ネットワークの問題

大規模な**プロダクション環境**では、複数の VLAN を持ち、LXD クライアントを直接それらの VLAN に接続するのが一般的です。netplan と systemd-networkd を使っている場合、いくつかの最悪の問題を引き起こす可能性があるバグに遭遇するでしょう。

VLAN ベースのブリッジでは netplan で systemd-networkd が使えない

執筆時点（2019-03-05）では、netplan は VLAN にアタッチされたブリッジにランダムな MAC アドレスを割り当てられません。常に同じ MAC アドレスを選択するため、同じネットワークセグメントに複数のマシンが存在する場合、レイヤー 2 の問題が発生します。複数のブリッジを作成することも困難です。代わりに `network-manager` を使ってください。設定例は次のようになります。管理アドレスが 10.61.0.25 で、VLAN102 をクライアントのトラフィックに使います。

```
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    eth0:
      dhcp4: no
      accept-ra: no
      # This is the 'Management Address'
      addresses: [ 10.61.0.25/24 ]
      gateway4: 10.61.0.1
      nameservers:
        addresses: [ 1.1.1.1, 8.8.8.8 ]
    eth1:
```

(次のページに続く)

(前のページからの続き)

```
    dhcp4: no
    accept-ra: no
    # A bogus IP address is required to ensure the link state is up
    addresses: [ 10.254.254.25/32 ]

vlans:
  vlan102:
    accept-ra: no
    dhcp4: no
    id: 102
    link: eth1

bridges:
  br102:
    accept-ra: no
    dhcp4: no
    interfaces: [ "vlan102" ]
    # A bogus IP address is required to ensure the link state is up
    addresses: [ 10.254.102.25/32 ]
    parameters:
      stp: false
```

注意事項

- eth0 はデフォルトゲートウェイの指定がある管理インターフェースです
- vlan102 は eth1 を使います
- br102 は vlan102 を使います。そして **bogus な /32 の IP アドレスが割り当てられています**

他に重要なこととして、`stp: false` を設定することがあります。そうしなければ、ブリッジは最大で 10 秒間 `learning` 状態となります。これはほとんどの DHCP リクエストが投げられる期間よりも長いです。クロスコネクトされてループを引き起こす可能性はありませんので、このように設定しても安全です。

port security に気をつける

スイッチは MAC アドレスの変更を許さず、不正な MAC アドレスのトラフィックをドロップするか、ポートを完全に無効にするものが多いです。ホストから LXD インスタンスに ping できたとしても、**異なった** ホストから ping できない場合は、これが原因の可能性があります。この原因を突き止める方法は、アップリンク（この場合は eth1）で tcpdump を実行することです。すると、応答は送るが ACK を取得できない 'ARP Who has xx.xx.xx.xx tell yy.yy.yy.yy'、もしくは ICMP パケットが行き来しているものの、決して他のホストで受け取られないのが見えるでしょう。

不必要に特権コンテナを実行しない

特権コンテナはホスト全体に影響する処理を行うことができます。例えば、ネットワークカードをリセットするために、`/sys` 内のものを使えます。これは **ホスト全体** に対してリセットを行い、ネットワークの切断を引き起こします。ほぼすべてのことが非特権コンテナで実行できます。コンテナ内から NFS マウントしたいというような、通常とは異なる特権が必要なケースでは、バインドマウントを使う必要があるかもしれません。

3.1.4 セキュリティ

LXD のインストールが安全であることを保証するために、以下の点を考慮してください。

- オペレーティングシステムを最新に保ち、利用可能なすべてのセキュリティパッチをインストールする。
- サポートされている LXD のバージョン（LTS リリースまたは月例機能リリース）のみを使用する。
- LXD デーモンとリモート API へのアクセスを制限すること。
- 必要とされない限り、特権コンテナを使わないこと。特権的なコンテナを使う場合は、適切なセキュリティ対策をしてください。詳細は LXC セキュリティページを参照してください。
- ネットワークインターフェイスを安全に設定してください。

詳細な情報は以下のセクションを参照してください。

セキュリティ上の問題を発見した場合、その問題の報告方法については LXD のセキュリティポリシー (原文: LXD security policy) を参照してください。

サポートされているバージョン

サポートされていないバージョンの LXD は実運用環境では絶対に使用しないでください。

LXD には 2 種類のリリースがあります。

- 月次機能リリース
- LTS リリース

フィーチャーリリースでは、最新のものだけがサポートされ、通常はポイントリリースは行いません。次の月次リリースまでユーザーに待ってもらいます。

LTS リリースでは、定期的にバグフィックス・リリースを行います。これは、フィーチャー・リリースに含まれるバグフィックスを集積したもので、新機能は含まれません。

LXD デーモンへのアクセス

LXD は UNIX ソケットを介してローカルにアクセスできるデーモンで、設定されていれば TLS (Transport Layer Security) ソケットを介してリモートにアクセスすることもできます。ソケットにアクセスできる人は、ホストデバイスやファイルシステムをアタッチしたり、すべてのインスタンスのセキュリティ機能をいじったりするなど、LXD を完全に制御することができます。

したがって、デーモンへのアクセスを信頼できるユーザに制限するようにしてください。

LXD デーモンへのローカルアクセス

LXD デーモンは root で動作し、ローカル通信用の UNIX ソケットを提供します。LXD のアクセス制御は、グループメンバーシップに基づいて行われます。root ユーザーと `lxd` グループのすべてのメンバーがローカルデーモンと対話できます。

重要: UNIX ソケットを介した LXD へのローカルアクセスは、常に LXD へのフルアクセスを許可します。これは、任意のインスタンス上のセキュリティ機能を変更できる能力に加えて、任意のインスタンスにファイルシステムパスやデバイスをアタッチする能力を含みます。

したがって、あなたのシステムへのルートアクセスを信頼できるユーザーにのみ、このようなアクセスを与えるべきです。

リモート API へのアクセス

デフォルトでは、デーモンへのアクセスはローカルでのみ可能です。`core.https_address` という設定オプション（**サーバ設定**参照）を設定することで、同じ API を TLS ソケットでネットワーク上に公開することができます。リモートクライアントは、LXD に接続して、公開用にマークされたイメージにアクセスできます。

リモートクライアントが API にアクセスできるように、信頼できるクライアントとして認証する方法がいくつかあります。詳細は**リモート *API* 認証**を参照してください。

本番環境では、`core.https_address` に、（ホスト上の任意のアドレスではなく）サーバーが利用可能な単一のアドレスを設定する必要があります。さらに、許可されたホスト/サブネットからのみ LXD ポートへのアクセスを許可するファイアウォールルールを設定する必要があります。

コンテナのセキュリティ

LXD コンテナはセキュリティのために幅広い機能を使うことができます。

デフォルトでは、コンテナは **非特権** (*unprivileged*) であり、ユーザーネームスペース内で動作することを意味し、コンテナ内のユーザーの能力を、コンテナが所有するデバイスに対する制限された権限を持つホスト上の通常のユーザーに制限します。

コンテナ間のデータ共有が必要ない場合は、`security.idmap.isolated`（**インスタンスの設定**参照）を有効にすることで、各コンテナに対して重複しない uid/gid マップを使用し、他のコンテナに対する潜在的な DoS（サービス拒否）攻撃を防ぐことができます。

LXD はまた、**特権** (*privileged*) コンテナを実行することができます。そのようなコンテナの中でルートアクセスを持つユーザは、閉じ込められた状態から逃れる方法を見つけるだけでなく、ホストを DoS することができるでしょう。

コンテナのセキュリティと私たちが使っているカーネルの機能についてのより詳細な情報は LXC セキュリティページにあります。

ネットワークセキュリティ

ネットワークインターフェースは必ず安全に設定してください。どのような点を考慮すべきかは、使用するネットワークモードによって異なります。

ブリッジ型 NIC のセキュリティ

LXD のデフォルトのネットワークモードは、各インスタンスが接続する「管理された」プライベートネットワークのブリッジを提供することです。このモードでは、ホスト上に `lxdbr0` というインターフェースがあり、それがインスタンスのブリッジとして機能します。

ホストは、管理されたブリッジごとに `dnsmasq` のインスタンスを実行し、IP アドレスの割り当てと、権威 DNS および再帰 DNS サービスの提供を担当します。

DHCPv4 を使用しているインスタンスには、IPv4 アドレスが割り当てられ、インスタンス名の DNS レコードが作成されます。これにより、インスタンスが DHCP リクエストに偽のホスト名情報を提供して、DNS レコードを偽装することができなくなります。

`dnsmasq` サービスは、IPv6 のルータ広告機能も提供します。つまり、インスタンスは SLAAC を使って自分の IPv6 アドレスを自動設定するので、`dnsmasq` による割り当ては行われません。しかし、DHCPv4 を使用しているインスタンスは、SLAAC IPv6 アドレスに相当する AAAA の DNS レコードも取得します。これは、インスタンスが IPv6 アドレスを生成する際に、IPv6 プライバシー拡張を使用していないことを前提としています。

このデフォルト構成では、DNS 名を偽装することはできませんが、インスタンスはイーサネットブリッジに接続されており、希望するレイヤー 2 トラフィックを送信することができます。これは、信頼されていないインスタンスがブリッジ上で MAC または IP の偽装を効果的に行うことができることを意味します。

デフォルトの設定では、ブリッジに接続されたインスタンスがブリッジに (潜在的に悪意のある)IPv6 ルータ広告を送信することで、LXD ホストの IPv6 ルーティングテーブルを修正することも可能です。これは、`lxdbr0` インターフェイスが`/proc/sys/net/ipv6/conf/lxdbr0/accept_ra` を `2` に設定して作成されているためで、`forwarding` が有効であるにもかかわらず、LXD ホストがルーター広告を受け入れることを意味しています（詳細は/proc/sys/net/ipv4/* Variables を参照してください)。

しかし、LXD はいくつかのブリッジ型 NIC (Network interface controller) セキュリティ機能を提供しており、インスタンスがネットワーク上に送信することを許可されるトラフィックの種類を制御するために使用することができます。これらの NIC 設定は、インスタンスが使用しているプロファイルに追加する必要がありますが、以下のように個々のインスタンスに追加することもできます。

ブリッジ型 NIC には、以下のようなセキュリティ機能があります。

キー	タイプ	デフォルト	必須	説明
security.mac_filtering	boolean	false	no	インスタンスが他人の MAC アドレスを詐称することを防ぐ。
security.ipv4_filtering	boolean	false	no	インスタンスが他の IPv4 アドレスになりすますことを防ぎます（mac_filtering を有効にします）。
security.ipv6_filtering	boolean	false	no	インスタンスが他の IPv6 アドレスになりすますことを防ぎます（mac_filtering を有効にします）。

プロファイルで設定されたデフォルトのブリッジ型 NIC の設定は、インスタンスごとに以下の方法で上書きすることができます。

```
lxc config device override <instance> <NIC> security.mac_filtering=true
```

これらの機能を併用することで、ブリッジに接続されているインスタンスが MAC アドレスや IP アドレスを詐称することを防ぐことができます。これらのオプションは、ホスト上で利用可能なものに応じて、`xtables`（iptables、ip6tables、ebtables）または `nftables` を使用して実装されます。

これらのオプションは、ネストされたコンテナが異なる MAC アドレスを持つ親ネットワークを使用すること（ブリッジされた NIC や macvlan NIC を使用すること）を効果的に防止することができます。

IP フィルタリング機能は、スプーフィングされた IP を含む ARP および NDP アドバタイジングをブロックし、スプーフィングされたソースアドレスを含むすべてのパケットをブロックします。

`security.ipv4_filtering` または `security.ipv6_filtering` が有効で、インスタンスに IP アドレスが割り当てられない場合（`ipvX.address=none` またはブリッジで DHCP サービスが有効になっていないため）、そのプロトコルのすべての IP トラフィックがインスタンスからブロックされます。

`security.ipv6_filtering` が有効な場合、IPv6 のルータ広告がインスタンスからブロックされます。

`security.ipv4_filtering` または `security.ipv6_filtering` が有効な場合、ARP、IPv4 または IPv6 ではないイーサネットフレームはすべてドロップされます。これにより、スタックされた VLAN QinQ（802.1ad）フレームが IP フィルタリングをバイパスすることを防ぎます。

ルート化された NIC のセキュリティ

「ルーテッド」と呼ばれる別のネットワークモードがあります。このモードでは、コンテナとホストの間に veth ペアを提供します。このネットワークモードでは、LXD ホストがルータとして機能し、コンテナの IP 宛のトラフィックをコンテナの veth インターフェイスに誘導するスタティックルートがホストに追加されます。

デフォルトでは、コンテナからのルータ広告が LXD ホスト上の IPv6 ルーティングテーブルを変更するのを防ぐために、ホスト上に作成された veth インタフェースは、その `accept_ra` 設定が無効になっています。それに加え

て、コンテナが持っていることをホストが知らない IP に対するソースアドレスの偽装を防ぐために、ホスト上の rp_filter が 1 に設定されています。

3.1.5 コントリビュート

プロジェクトに貢献する前に、以下のガイドラインを確認してください。

プルリクエスト

このプロジェクトへの変更は、Github でプルリクエストとして提案してください。https://github.com/lxc/lxd

そのあと、提案はコードレビューを経て承認され、メインブランチにマージされます。

コミット構成

コミットを次のように分類する必要があります:

- API 拡張 (doc/api-extensions.md と shared/version.api.go を含む変更に対して api: Add XYZ extension)
- ドキュメント (doc/ 内のファイルに対して doc: Update XYZ)
- API 構造 (shared/api/ の変更に対して shared/api: Add XYZ)
- Go クライアントパッケージ (client/ の変更に対して client: Add XYZ)
- CLI (lxc/ の変更に対して lxc/<command>: Change XYZ)
- スクリプト (scripts/ の変更に対して scripts: Update bash completion for XYZ)
- LXD デーモン (lxd/ の変更に対して lxd/<package>: Add support for XYZ)
- テスト (tests/ の変更に対して tests: Add test for XYZ)

同様のパターンが LXD コードツリーの他のツールにも適用されます。そして複雑さによっては、さらに小さな単位に分けられるかもしれません。

CLI ツール (lxc/) 内の文字列を更新する際は、テンプレートを更新してコミットする必要があるでしょう:

- make i18n
- git commit -a -s -m "i18n: Update translation templates" po/

このようにすることで、コントリビューションに対するレビューが容易になり、stable ブランチへバックポートするプロセスが大幅に簡素化されます。

ライセンスと著作権

デフォルトで、このプロジェクトに対するいかなる貢献も Apache 2.0 ライセンスの下で行われます。

変更の著者は、そのコードに対する著作権を保持します (著作権の譲渡はありません)。

開発者の起源の証明

このプロジェクトへの貢献の追跡を改善するために、DCO 1.1 を使用しており、ブランチに入るすべての変更に対して「サインオフ」手順を使用しています。

サインオフとは、あなたがそのコミットを書いたことを証明する、そのコミットの説明の最後にある単純な行です。この行は、自分が書いたものであることを証明したり、オープンソースとして渡す権利があることを証明したりします。

> Developer Certificate of Origin Version 1.1
>
> Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 660 York Street, Suite 102, San Francisco, CA 94110 USA
>
> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.
>
> Developer's Certificate of Origin 1.1
>
> By making a contribution to this project, I certify that:
>
> (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
>
> (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
>
> (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
>
> (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

有効なサインオフラインの例は以下の通りです。

```
Signed-off-by: Random J Developer <random@developer.org>
```

実名と有効な電子メールアドレスを使用してください。残念ながら、ペンネームや匿名での投稿はできません。

また、それぞれのコミットには作者が個別に署名する必要があります。大きなセットの一部であってもです。`git commit -s` が役に立つでしょう。

Code of Conduct

コントリビュートする際には、行動規範を遵守しなければなりません。行動規範は、以下のサイトから入手できます。https://github.com/lxc/lxd/blob/master/CODE_OF_CONDUCT.md

開発を始める

開発環境をセットアップし LXD の新機能に取り組みを開始するには以下の手順に従ってください。

依存ライブラリーのビルド

依存ライブラリーをビルドするには *LXD* **のソースからのインストール** の手順に従ってください。

あなたの fork の remote を追加

依存ライブラリーをビルドし終わったら、 GitHub の fork を remote として追加しその fork にスイッチできます。

```
git remote add myfork git@github.com:<your_username>/lxd.git
git remote update
git checkout myfork/master
```

LXD のビルド

最後にレポジトリ内で `make` を実行すれば LXD のあなたの fork をビルドできます。

この時点であなたが最も行いたいであろうことはあなたの fork 上にあなたの変更のための新しいブランチを作ることです。

```
git checkout -b [name_of_your_new_branch]
git push myfork [name_of_your_new_branch]
```

LXD の新しいコントリビュータのための重要な注意事項

- 永続データは LXD_DIR ディレクトリに保管されます。これは lxd init で作成されます。 LXD_DIR のデフォルトは /var/lib/lxd か snap ユーザーは /var/snap/lxd/common/lxd です。
- 開発中はバージョン衝突を避けるため LXD のあなたの fork 用に LXD_DIR の値を変更すると良いでしょう。
- あなたのソースからコンパイルされる実行ファイルはデフォルトでは $(go env GOPATH)/bin に生成されます。
  - あなたの変更をテストするときはこれらの実行ファイル（インストール済みかもしれないグローバルの lxd ではなく）を明示的に起動する必要があります。
  - これらの実行ファイルを適切なオプションを指定してもっと便利に呼び出せるように ~/.bashrc にエイリアスを作るという選択も良いでしょう。
- 既存のインストール済み LXD のデーモンを実行するための systemd サービスが設定されている場合はバージョン衝突を避けるためにサービスを無効にすると良いでしょう。

3.1.6 サポート

以下のチャネルで情報を得たり、ユーザーサポートを依頼することができます。

サポートとコミュニティ

LXD コミュニティと交流するために以下のチャンネルが用意されています。

バグレポート

バグレポートや機能要求は以下の場所で受け付けています。https://github.com/lxc/lxd/issues/new

フォーラム

フォーラムは以下の場所にあります。https://discuss.linuxcontainers.org

メーリングリスト

開発者やユーザーの議論には LXC のメーリングリストを利用しています。メーリングリストは以下の場所にあります。https://lists.linuxcontainers.org

IRC

ライブの議論がお好みならば、irc.libera.chat の#lxc で私たちを見つけることができます。必要であれば Getting started with IRC を参照してください。

商用サポート

LXD の商用サポートは、Canonical Ltd を通じて受けることができます。

ドキュメント

公式ドキュメントは https://lxd-ja.readthedocs.io/ja/latest/ (原文は https://linuxcontainers.org/lxd/docs/master/) で入手できます。

その他の資料は、website、YouTube、フォーラムの Tutorials section にあります。

3.2 設定

LXD は次のコンポーネントの設定を保管しています。

3.2.1 コンテナ

イントロダクション

コンテナは LXD のデフォルトタイプであり、現時点では一番機能を持っており、LXD インスタンスの完全な実装です。

これは `liblxc` (LXC) を使って実装しています。

設定

設定オプションについては **インスタンスの設定** をご覧ください。

ライブマイグレーション

LXD では、CRIU を使ったコンテナのライブマイグレーションが使えます。コンテナのメモリ転送を最適化するために、`migration.incremental.memory` プロパティを `true` に設定することで、CRIU の pre-copy 機能を使うように LXD を設定できます。つまり、LXD は CRIU にコンテナの一連のメモリダンプを実行するように要求します。ダンプが終わると、LXD はメモリダンプを指定したリモートホストに送ります。理想的なシナリオでは、各メモリダンプを前のメモリダンプとの差分にまで減らし、それによりすでに同期されたメモリの割合を増やします。同期されたメモリの割合が `migration.incremental.memory.goal` で設定したしきい値と等しいか超えた場合、LXD は CRIU に最終的なメモリダンプを実行し、転送するように要求します。`migration.incremental.memory.iterations` で指定したメモリダンプの最大許容回数に達した後、まだしきい値に達していない場合は、LXD は最終的なメモリダンプを CRIU に要求し、コンテナを移行します。

3.2.2 インスタンスの設定

インスタンス

プロパティ

次のプロパティは、インスタンスに直接結びつくプロパティであり、プロファイルの一部ではありません:

- `name`
- `architecture`

`name` はインスタンス名であり、インスタンスのリネームでのみ変更できます。

有効なインスタンス名は次の条件を満たさなければなりません:

- 1 〜 63 文字
- ASCII テーブルの文字、数字、ダッシュのみから構成される
- 1 文字目は数字、ダッシュではない
- 最後の文字はダッシュではない

この要件は、インスタンス名が DNS レコードとして、ファイルシステム上で、色々なセキュリティプロファイル、そしてインスタンス自身のホスト名として適切に使えるように定められています。

Key/value 形式の設定

key/value 形式の設定は、名前空間構造を取っており、現在は次のような名前空間があります:

- `boot` (ブートに関連したオプション、タイミング、依存性、…)
- `cloud-init` (cloud-init の設定)
- `environment` (環境変数)
- `image` (作成時のイメージプロパティのコピー)
- `limits` (リソース制限)
- `nvidia` (NVIDIA と CUDA の設定)
- `raw` (生のインスタンス設定を上書きする)
- `security` (セキュリティポリシー)
- `user` (ユーザーの指定するプロパティを保持。検索可能)
- `volatile` (インスタンス固有の内部データを格納するために LXD が内部的に使用する設定)

現在設定できる項目は次のものです:

キー	型	デフォルト値	ライブアップデート	条件	説明
agent.nic_config	boolean	false	n/a	virtual-machine	デフ
boot.autostart	boolean	-	n/a	-	LXD
boot.autostart.delay	integer	0	n/a	-	イン
boot.autostart.priority	integer	0	n/a	-	イン
boot.host_shutdown_timeout	integer	30	yes	-	強制
boot.stop.priority	integer	0	n/a	-	イン
cloud-init.network-config	string	eth0 上の DHCP	no	-	Clou
cloud-init.user-data	string	#cloud-config	no	-	Clou
cloud-init.vendor-data	string	#cloud-config	no	-	Clou
cluster.evacuate	string	auto	n/a	-	イン
environment.*	string	-	yes (exec)	-	イン
limits.cpu	string	-	yes	-	イン
limits.cpu.allowance	string	100%	yes	container	どれ
limits.cpu.priority	integer	10 (maximum)	yes	container	同じ
limits.disk.priority	integer	5 (medium)	yes	-	負荷
limits.hugepages.64KB	string	-	yes	container	64 K
limits.hugepages.1MB	string	-	yes	container	1 MI

キー	型	デフォルト値	ライブアップデート	条件	説明
limits.hugepages.2MB	string	-	yes	container	2 MI
limits.hugepages.1GB	string	-	yes	container	1 GI
limits.kernel.*	string	-	no	container	イン
limits.memory	string	-	yes	-	ホス
limits.memory.enforce	string	hard	yes	container	hard
limits.memory.hugepages	boolean	false	no	virtual-machine	イン
limits.memory.swap	boolean	true	yes	container	この
limits.memory.swap.priority	integer	10 (maximum)	yes	container	高い
limits.network.priority	integer	0 (minimum)	yes	-	負荷
limits.processes	integer	- (max)	yes	container	イン
linux.kernel_modules	string	-	yes	container	イン
linux.sysctl.*	string	-	no	container	sysct
migration.incremental.memory	boolean	false	yes	container	イン
migration.incremental.memory.goal	integer	70	yes	container	イン
migration.incremental.memory.iterations	integer	10	yes	container	イン
migration.stateful	boolean	false	no	virtual-machine	ステ
nvidia.driver.capabilities	string	compute,utility	no	container	イン
nvidia.runtime	boolean	false	no	container	ホス
nvidia.require.cuda	string	-	no	container	必要
nvidia.require.driver	string	-	no	container	必要
raw.apparmor	blob	-	yes	-	生成
raw.idmap	blob	-	no	unprivileged conta iner	生（
raw.lxc	blob	-	no	container	生成
raw.qemu	blob	-	no	virtual-machine	生成
raw.seccomp	blob	-	no	container	生（
security.devlxd	boolean	true	no	-	イン
security.devlxd.images	boolean	false	no	container	devl
security.idmap.base	integer	-	no	unprivileged container	割り
security.idmap.isolated	boolean	false	no	unprivileged container	イン
security.idmap.size	integer	-	no	unprivileged container	使用
security.nesting	boolean	false	yes	container	イン
security.privileged	boolean	false	no	container	特権
security.protection.delete	boolean	false	yes	-	イン
security.protection.shift	boolean	false	yes	container	イン
security.agent.metrics	boolean	true	no	virtual-machine	状態
security.secureboot	boolean	true	no	virtual-machine	UEF
security.syscalls.allow	string	-	no	container	\n 区

キー	型	デフォルト値	ライブアップデート	条件	説明
security.syscalls.deny	string	-	no	container	\n 区
security.syscalls.deny_compat	boolean	false	no	container	x86_
security.syscalls.deny_default	boolean	true	no	container	デフ
security.syscalls.intercept.bpf	boolean	false	no	container	bpf
security.syscalls.intercept.bpf.devices	boolean	false	no	container	devi
security.syscalls.intercept.mknod	boolean	false	no	container	mkno
security.syscalls.intercept.mount	boolean	false	no	container	moun
security.syscalls.intercept.mount.allowed	string	-	yes	container	イン
security.syscalls.intercept.mount.fuse	string	-	yes	container	指定
security.syscalls.intercept.mount.shift	boolean	false	yes	container	moun
security.syscalls.intercept.sched_setscheduler	boolean	false	no	container	sche
security.syscalls.intercept.setxattr	boolean	false	no	container	setx
snapshots.schedule	string	-	no	-	Cron
snapshots.schedule.stopped	bool	false	no	-	停止
snapshots.pattern	string	snap%d	no	-	スナ
snapshots.expiry	string	-	no	-	スナ
user.*	string	-	n/a	-	自由

LXD は内部的に次の揮発性の設定を使います:

キー	型	デフォルト値	説明
volatile.apply_template	string	-	次の起動時にトリガーされるテンプレートフックの名前
volatile.base_image	string	-	インスタンスを作成したイメージのハッシュ（存在する場合）
volatile.cloud-init.instance-id	string	-	cloud-init に公開するインスタンス ID (UUID)
volatile.evacuate.origin	string	-	待避したインスタンスのオリジン（クラスタメンバー）
volatile.idmap.base	integer	-	インスタンスの主 idmap の範囲の最初の ID
volatile.idmap.current	string	-	インスタンスで現在使用中の idmap
volatile.idmap.next	string	-	次にインスタンスが起動する際に使う idmap
volatile.last_state.idmap	string	-	シリアライズ化したインスタンスの uid/gid マップ
volatile.last_state.power	string	-	最後にホストがシャットダウンした時点のインスタンスの状態
volatile.vsock_id	string	-	最後の起動時に使用されたインスタンスの vsock ID
volatile.uuid	string	-	インスタンスの UUID （全サーバとプロジェクト内でグローバルにユニーク）
volatile.<name>.apply_quota	string	-	次回のインスタンス起動時に適用されるディスククォータ
volatile.<name>.ceph_rbd	string	-	Ceph のディスクデバイスの RBD デバイスパス
volatile.<name>.host_name	string	-	ホスト上のネットワークデバイス名
volatile.<name>.hwaddr	string	-	ネットワークデバイスの MAC アドレス（ `hwaddr` プロパティがデバイスに設定されていない場合）
volatile.<name>.last_state.created	string	-	物理デバイスのネットワークデバイスが作られたかどうか ("true" または "false")
volatile.<name>.last_state.mtu	string	-	物理デバイスをインスタンスに移動したときに使われていたネットワークデバイスの元の MTU
volatile.<name>.last_state.hwaddr	string	-	物理デバイスをインスタンスに移動したときに使われていたネットワークデバイスの元の MAC
volatile.<name>.last_state.vf.id	string	-	SR-IOV の仮想ファンクション（VF）をインスタンスに移動したときに使われていた VF の ID
volatile.<name>.last_state.vf.hwaddr	string	-	SR-IOV の仮想ファンクション（VF）をインスタンスに移動したときに使われていた VF の MAC
volatile.<name>.last_state.vf.vlan	string	-	SR-IOV の仮想ファンクション（VF）をインスタンスに移動したときに使われていた VF の元の VLAN
volatile.<name>.last_state.vf.spoofcheck	string	-	SR-IOV の仮想ファンクション（VF）をインスタンスに移動したときに使われていた VF の元の spoof チェックの設定

加えて、次のユーザー設定がイメージで共通になっています（サポートを保証するものではありません）：

キー	型	デフォルト値	説明
user.meta-data	string	-	cloud-init メタデータ。設定は seed 値に追加されます

便宜的に型（type）を定義していますが、すべての値は文字列として保存されます。そして REST API を通して文字列として提供されます（後方互換性を損なうことなく任意の追加の値をサポートできます）。

これらの設定は lxc ツールで次のように設定できます:

```
lxc config set <instance> <key> <value>
```

揮発性（volatile）の設定はユーザーは設定できません。そして、インスタンスに対してのみ直接設定できます。

生（raw）の設定は、LXD が使うバックエンドの機能に直接アクセスできます。これを設定することは、自明ではない方法で LXD を破壊する可能性がありますので、可能な限り避ける必要があります。

CPU 制限

CPU 制限は cgroup コントローラの `cpuset` と `cpu` を組み合わせて実装しています。

`limits.cpu` は `cpuset` コントローラを使って、使う CPU を固定（ピンニング）します。使う CPU の組み合わせ（例: `1,2,3`）もしくは使う CPU の範囲（例: `0-3`）で指定できます。

代わりに CPU 数を指定した場合（例: `4`）、LXD は CPU の固定（ピンニング）がされていない全インスタンスのダイナミックな負荷分散を行い、マシン上の負荷を分散しようとします。インスタンスが起動したり停止するたびに、インスタンスはリバランスされます。これはシステムに CPU が足された場合も同様にリバランスされます。

単一の CPU に固定（ピンニング）するためには、CPU 数との区別をつけるために、範囲を指定する文法（例: `1-1`）を使う必要があります。

`limits.cpu.allowance` は、時間の制限を与えたときは CFS スケジューラのクォータを、パーセント指定をした場合は全体的な CPU シェアの仕組みを使います。

時間制限（例: `20ms/50ms`）はひとつの CPU 相当の時間に関連するので、ふたつの CPU の時間を制限するには、100ms/50ms のような指定を使うようにします。

パーセント指定を使う場合は、制限は負荷状態にある場合のみに適用されます。そして設定は、同じ CPU（もしくは CPU の組）を使う他のインスタンスとの比較で、インスタンスに対するスケジューラの優先度を計算するのに使われます。

`limits.cpu.priority` は、CPU の組を共有するいくつかのインスタンスに割り当てられた CPU の割合が同じ場合に、スケジューラの優先度スコアを計算するために使われます。

VM CPU トポロジー

LXD の仮想マシンはデフォルトでは vCPU を 1 つだけ割り当てて、それはホストの CPU のベンダーとタイプにマッチしたものとして表示されますがシングルコアでスレッドはありません。

`limits.cpu` を単一の整数に設定すると、複数の vCPU が割り当てられゲストにフルのコアとして公開します。これらの vCPU ホスト上の特定の物理コアにピン止めはされません。

`limits.cpu` に CPU ID (`lxc info --resources` で表示されます) の範囲やカンマ区切りリストを指定すると、vCPU はそれらの物理コアにピン止めされます。このシナリオでは LXD は CPU 設定が実際のハードウェアトポロジーと合っているか確認し、合っている場合はゲストのトポロジーに複製します。

例えばピン止めの設定が 8 スレッドを含む場合、スレッドの各ペアは同じコアから提供され、コア番号が 2 つの CPU にまたがる場合でも、 LXD はゲストに 2 つの CPU を提供し、各 CPU は 2 つのコアを持ち、各コアは 2 つのスレッドを持ちます。NUMA レイアウトも同様に複製され、このシナリオではゲストは十中八九各 CPU ソケットにつき 1 つ、合計 2 つの NUMA ノードを持つことになるでしょう。

複数の NUMA ノードの環境では、メモリも同様に NUMA ノードに分割され、それに応じてホストでピン止めされ、その後ゲストにも公開します。

これら全てによりゲスト内で非常に高いパフォーマンスのオペレーションが可能です。これはゲストのスケジューラーが NUMA ノード間でメモリを共有したりプロセスを移動する際にソケット、コア、スレッドについて適切に判断し、 NUMA トポロジーも考慮できるからです。

デバイス設定

LXD は、標準の POSIX システムが動作するのに必要な基本的なデバイスを常にインスタンスに提供します。これらはインスタンスやプロファイルの設定では見えず、上書きもできません。

このデバイスには次のようなデバイスが含まれます:

- `/dev/null` (キャラクターデバイス)
- `/dev/zero` (キャラクターデバイス)
- `/dev/full` (キャラクターデバイス)
- `/dev/console` (キャラクターデバイス)
- `/dev/tty` (キャラクターデバイス)
- `/dev/random` (キャラクターデバイス)
- `/dev/urandom` (キャラクターデバイス)
- `/dev/net/tun` (キャラクターデバイス)
- `/dev/fuse` (キャラクターデバイス)

- lo (ネットワークインターフェース)

これ以外に関しては、インスタンスの設定もしくはインスタンスで使われるいずれかのプロファイルで定義する必要があります。デフォルトのプロファイルには、インスタンス内で eth0 になるネットワークインターフェースが通常は含まれます。

インスタンスに追加でデバイスを追加する場合は、デバイスエントリーを直接インスタンスかプロファイルに追加できます。

デバイスはインスタンスの実行中に追加・削除できます。

各デバイスエントリーは一意な名前で識別されます。もし同じ名前が後続のプロファイルやインスタンス自身の設定で使われている場合、エントリー全体が新しい定義で上書きされます。

デバイスエントリーは次のようにインスタンスに追加するか:

```
lxc config device add <instance> <name> <type> [key=value]...
```

もしくは次のようにプロファイルに追加します:

```
lxc profile device add <profile> <name> <type> [key=value]...
```

デバイスタイプ

LXD では次のデバイスタイプが使えます:

ID (database)	Name	Condition	Description
0	*none*	-	継承ブロッカー
1	*nic*	-	ネットワークインターフェース
2	*disk*	-	インスタンス内のマウントポイント
3	*unix-char*	container	Unix キャラクターデバイス
4	*unix-block*	container	Unix ブロックデバイス
5	*usb*	-	USB デバイス
6	*gpu*	-	GPU デバイス
7	*infiniband*	container	インフィニバンドデバイス
8	*proxy*	container	プロキシデバイス
9	*unix-hotplug*	container	Unix ホットプラグデバイス
10	*tpm*	-	TPM デバイス
11	*pci*	VM	PCI デバイス

Type: none

サポートされるインスタンスタイプ: コンテナ, VM

none タイプのデバイスはプロパティを一切持たず、インスタンス内に何も作成しません。

プロファイルからのデバイスの継承を止めるためだけに存在します。

継承を止めるには、継承をスキップしたいデバイスと同じ名前の none タイプのデバイスを追加するだけです。デバイスは、もともと含まれているプロファイルの後にプロファイルに追加されるか、直接インスタンスに追加されます。

Type: nic

LXD では、様々な種類のネットワークデバイス（ネットワークインターフェースコントローラーや NIC と呼びます）が使えます:

インスタンスにネットワークデバイスを追加する際には、追加したいデバイスのタイプを選択するのに 2 つの方法があります。`nictype` プロパティを指定するか `network` プロパティを使うかです。

`network` プロパティを使って NIC を指定する

`network` プロパティを指定する場合、 NIC は既存の管理されたネットワークにリンクされ、 `nictype` はネットワークのタイプに応じて自動的に検出されます。

NIC の設定の一部は個々の NIC で変更可能ではなくネットワークから継承されます。

これらの詳細は下記の NIC 固有のセクションの "Managed" カラムに記載します。

利用可能な NIC

NIC ごとにどのプロパティが設定可能かの詳細については下記を参照してください。

次の NIC は `nictype` か `network` プロパティを使って選択できます。

- bridged: ホスト上に存在するブリッジを使います。ホストのブリッジとインスタンスを接続する仮想デバイスペアを作成します。
- macvlan: 既存のネットワークデバイスをベースに MAC が異なる新しいネットワークデバイスを作成します。
- sriov: SR-IOV が有効な物理ネットワークデバイスの仮想ファンクション（virtual function）をインスタンスに与えます。

次の NIC は `network` プロパティのみを使って選択できます。

- *ovn*: 既存の OVN ネットワークを使用し、インスタンスが接続する仮想デバイスペアを作成します。

次の NIC は nictype プロパティのみを使って選択できます。

- physical: ホストの物理デバイスを直接使います。対象のデバイスはホスト上では見えなくなり、インスタンス内に出現します。
- ipvlan: 既存のネットワークデバイスをベースに MAC アドレスは同じですが IP アドレスが異なる新しいネットワークデバイスを作成します。
- p2p: 仮想デバイスペアを作成し、片方をインスタンス内に置き、残りの片方をホスト上に残します。
- routed: 仮想デバイスペアを作成し、ホストからインスタンスに繋いで静的ルートをセットアップし ARP/NDP エントリーをプロキシします。これにより指定された親インタフェースのネットワークにインスタンスが参加できるようになります。

nic: bridged

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: nictype, network

ホストの既存のブリッジを使用し、ホストのブリッジをインスタンスに接続するための仮想デバイスのペアを作成します。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Managed	Description
parent	string	-	yes	yes	ホストデバイスの名前
network	string	-	yes	no	（parent の代わりに）デバイスをリンクする先の LXD ネットワーク
name	string	カーネルが割り当て	no	no	インスタンス内でのインタフェースの名前
mtu	integer	親の MTU	no	yes	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	no	新しいインタフェースの MAC アドレス
host_name	string	ランダムに割り当て	no	no	ホスト内でのインタフェースの名前
limits.ingress	string	-	no	no	入力トラフィックの I/O 制限値（さまざまな単位が使用可能、下記参照）
limits.egress	string	-	no	no	出力トラフィックの I/O 制限値（さまざまな単位が使用可能、下記参照）
limits.max	string	-	no	no	`limits.ingress` と `limits.egress` の両方を同じ値に変更する
ipv4.address	string	-	no	no	DHCP でインスタンスに割り当てる IPv4 アドレス（security.ipv4_filtering 設定時に全ての IPv4 トラフィックを制限するには none と設定可能）
ipv6.address	string	-	no	no	DHCP でインスタンスに割り当てる IPv6 アドレス（security.ipv6_filtering 設定時に全ての IPv6 トラフィックを制限するには none と設定可能）
ipv4.routes	string	-	no	no	ホスト上で NIC に追加する IPv4 静的ルートのカンマ区切りリスト
ipv6.routes	string	-	no	no	ホスト上で NIC に追加する IPv6 静的ルートのカンマ区切りリスト
ipv4.routes.external	string	-	no	no	NIC にルーティングしアップリンクのネットワーク (BGP) で公開する IPv4 静的ルートのカンマ区切りリスト
ipv6.routes.external	string	-	no	no	NIC にルーティングしアップリンクのネットワーク (BGP) で公開する IPv6 静的ルートのカンマ区切りリスト
security.mac_filtering	boolean	false	no	no	インスタンスが他の MAC アドレスになりすますのを防ぐ
security.ipv4_filtering	boolean	false	no	no	インスタンスが他の IPv4 アドレスになりすますのを防ぐ (これを設定すると mac_filtering も有効になります)
security.ipv6_filtering	boolean	false	no	no	インスタンスが他の IPv6 アドレスになりすますのを防ぐ (これを設定すると mac_filtering も有効になります)
maas.subnet.ipv4	string	-	no	yes	インスタンスを登録する MAAS IPv4 サブネット

nic: macvlan

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: `nictype`, `network`

既存のネットワークデバイスを元に新しいネットワークデバイスをセットアップしますが、異なる MAC アドレスを用います。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Managed	Description
parent	string	-	yes	yes	ホストデバイスの名前
network	string	-	yes	no	（parent の代わりに）デバイスをリンクする先の LXD ネットワーク
name	string	カーネルが割り当て	no	no	インスタンス内部でのインタフェース名
mtu	integer	親の MTU	no	yes	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	no	新しいインタフェースの MAC アドレス
vlan	integer	-	no	no	アタッチ先の VLAN ID
gvrp	boolean	false	no	no	GARP VLAN Registration Protocol を使って VLAN を登録する
maas.subnet.ipv4	string	-	no	yes	インスタンスを登録する MAAS IPv4 サブネット
maas.subnet.ipv6	string	-	no	yes	インスタンスを登録する MAAS IPv6 サブネット
boot.priority	integer	-	no	no	VM のブート優先度 (高いほうが先にブート)

nic: sriov

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: `nictype`, `network`

SR-IOV を有効にした物理ネットワークデバイスの仮想ファンクションをインスタンスに渡します。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Managed	Description
parent	string	-	yes	yes	ホストデバイスの名前
network	string	-	yes	no	（parent の代わりに）デバイスをリンクする先の LXD ネットワーク
name	string	カーネルが割り当て	no	no	インスタンス内部でのインタフェース名
mtu	integer	カーネルが割り当て	no	yes	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	no	新しいインタフェースの MAC アドレス
security.mac_filtering	boolean	false	no	no	インスタンスが他の MAC アドレスになりすますのを防ぐ
vlan	integer	-	no	no	アタッチ先の VLAN ID
maas.subnet.ipv4	string	-	no	yes	インスタンスを登録する MAAS IPv4 サブネット
maas.subnet.ipv6	string	-	no	yes	インスタンスを登録する MAAS IPv6 サブネット
boot.priority	integer	-	no	no	VM のブート優先度 (高いほうが先にブート)

nic: ovn

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: `network`

既存の OVN ネットワークを使用し、インスタンスが接続する仮想デバイスペアを作成します。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Managed	Description
network	string	-	yes	yes	デバイスの接続先の LXD ネットワーク
acceleration	string	none	no	no	ハードウェアオフローディングを有効にする。 `none` か `sriov` (下記の SR-IOV ハードウェアアクセラレーション参照)
name	string	カーネルが割り当て	no	no	インスタンス内部でのインタフェース名
host_name	string	ランダムに割り当て	no	no	ホスト内部でのインタフェース名
hwaddr	string	ランダムに割り当て	no	no	新しいインターフェースの MAC アドレス
ipv4.address	string	-	no	no	DHCP でインスタンスに割り当てる IPv4 アドレス
ipv6.address	string	-	no	no	DHCP でインスタンスに割り当てる IPv6 アドレス
ipv4.routes	string	-	no	no	ホスト上で nic に追加する IPv4 静的ルートのカンマ区切りリスト
ipv6.routes	string	-	no	no	ホスト上で nic に追加する IPv6 静的ルートのカンマ区切りリスト
ipv4.routes.external	string	-	no	no	NIC へのルートとアップリンクネットワークでの公開に使用する IPv4 静的ルートのカンマ区切りリスト
ipv6.routes.external	string	-	no	no	NIC へのルートとアップリンクネットワークでの公開に使用する IPv6 静的ルートのカンマ区切りリスト
boot.priority	integer	-	no	no	VM のブート優先度 (高いほうが先にブート)
security.acls	string	-	no	no	適用するネットワーク ACL のカンマ区切りリスト
security.acls.default.ingress.action	string	reject	no	no	どの ACL ルールにもマッチしない ingress トラフィックに使うアクション
security.acls.default.egress.action	string	reject	no	no	どの ACL ルールにもマッチしない egress トラフィックに使うアクション
security.acls.default.ingress.logged	boolean	false	no	no	どの ACL ルールにもマッチしない ingress トラフィックをログ出力するかどうか
security.acls.default.egress.logged	boolean	false	no	no	どの ACL ルールにもマッチしない egress トラフィックをログ出力するかどうか

SR-IOV ハードウェアアクセラレーション:

acceleration=sriov を使用するためには互換性のある SR-IOV switchdev が使用できる物理 NIC が LXD ホスト内に存在する必要があります。LXD は、物理 NIC (PF) が switchdev モードに設定されて OVN の統合 OVN ブリッジに接続されており、1 つ以上の仮想ファンクション (VF) がアクティブであることを想定しています。

これを実現するための前提となるセットアップの行程は以下の通りです。

PF と VF のセットアップ:

PF 上 (以下の例では 0000:09:00.0 の PCI アドレスで enp9s0f0np0 という名前) の VF をアクティベートしアンバインドします。次に switchdev モードと PF 上の hw-tc-offload を有効にします。最後に VF をリバインドします。

```
echo 4 > /sys/bus/pci/devices/0000:09:00.0/sriov_numvfs
for i in $(lspci -nnn | grep "Virtual Function" | cut -d' ' -f1); do echo 0000:$i > /sys/
↪bus/pci/drivers/mlx5_core/unbind; done
devlink dev eswitch set pci/0000:09:00.0 mode switchdev
ethtool -K enp9s0f0np0 hw-tc-offload on
for i in $(lspci -nnn | grep "Virtual Function" | cut -d' ' -f1); do echo 0000:$i > /sys/
↪bus/pci/drivers/mlx5_core/bind; done
```

OVS のセットアップ:

ハードウェアオフロードを有効にし、 PF NIC を統合ブリッジ (通常は br-int という名前) に追加します。

```
ovs-vsctl set open_vswitch . other_config:hw-offload=true
systemctl restart openvswitch-switch
ovs-vsctl add-port br-int enp9s0f0np0
ip link set enp9s0f0np0 up
```

nic: physical

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: nictype

物理デバイスそのものをパススルー。対象のデバイスはホストからは消失し、インスタンス内に出現します。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Description
parent	string	-	yes	ホストデバイスの名前
name	string	カーネルが割り当て	no	インスタンス内部でのインタフェース名
mtu	integer	親の MTU	no	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	新しいインタフェースの MAC アドレス
vlan	integer	-	no	アタッチ先の VLAN ID
gvrp	boolean	false	no	GARP VLAN Registration Protocol を使って VLAN を登録する
maas.subnet.ipv4	string	-	no	インスタンスを登録する MAAS IPv4 サブネット
maas.subnet.ipv6	string	-	no	インスタンスを登録する MAAS IPv6 サブネット
boot.priority	integer	-	no	VM のブート優先度 (高いほうが先にブート)

nic: ipvlan

サポートされるインスタンスタイプ: コンテナ

この NIC の指定に使えるプロパティ: `nictype`

既存のネットワークデバイスを元に新しいネットワークデバイスをセットアップしますが、異なる IP アドレスを用います。

LXD は現状 L2 と L3S モードで IPVLAN をサポートします。

このモードではゲートウェイは LXD により自動的に設定されますが、インスタンスが起動する前に `ipv4.address` と `ipv6.address` の設定の 1 つあるいは両方を使うことにより IP アドレスを手動で指定する必要があります。

DNS に関しては、ネームサーバは自動的には設定されないので、インスタンス内部で設定する必要があります。

ipvlan の nictype を使用するには以下の sysctl の設定が必要です。

IPv4 アドレスを使用する場合

```
net.ipv4.conf.<parent>.forwarding=1
```

IPv6 アドレスを使用する場合

```
net.ipv6.conf.<parent>.forwarding=1
net.ipv6.conf.<parent>.proxy_ndp=1
```

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Description
parent	string	-	yes	ホストデバイスの名前
name	string	カーネルが割り当て	no	インスタンス内部でのインタフェース名
mtu	integer	親の MTU	no	新しいインタフェースの MTU
mode	string	l3s	no	IPVLAN のモード (`l2` か `l3s` のいずれか)
hwaddr	string	ランダムに割り当て	no	新しいインタフェースの MAC アドレス
ipv4.address	string	-	no	インスタンスに追加する IPv4 静的アドレスのカンマ区切りリスト。 `l2` モードでは CIDR 形式か単一アドレス形式で指定可能（単一アドレスの場合はサブネットは /24）
ipv4.gateway	string	auto	no	`l3s` モードではデフォルト IPv4 ゲートウェイを自動的に追加するかどうか (auto か none を指定可能)。 `l2` モードではゲートウェイの IPv4 アドレスを指定。
ipv4.host_table	integer	-	no	（メインのルーティングテーブルに加えて） IPv4 の静的ルートを追加する先のルーティングテーブル ID
ipv6.address	string	-	no	インスタンスに追加する IPv6 静的アドレスのカンマ区切りリスト。 `l2` モードでは CIDR 形式か単一アドレス形式で指定可能（単一アドレスの場合はサブネットは /64）
ipv6.gateway	string	auto (l3s), - (l2)	no	`l3s` モードではデフォルト IPv6 ゲートウェイを自動的に追加するかどうか (auto か none を指定可能)。 `l2` モードではゲートウェイの IPv6 アドレスを指定。
ipv6.host_table	integer	-	no	（メインのルーティングテーブルに加えて） IPv6 の静的ルートを追加する先のルーティングテーブル ID
vlan	integer	-	no	アタッチ先の VLAN ID
gvrp	boolean	false	no	GARP VLAN Registration Protocol を使って VLAN を登録する

nic: p2p

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: `nictype`

仮想デバイスペアを作成し、片方はインスタンス内に配置し、もう片方はホストに残します。

デバイス設定プロパティは以下の通りです。

Key	Type	Default	Required	Description
name	string	カーネルが割り当て	no	インスタンス内部でのインタフェース名
mtu	integer	カーネルが割り当て	no	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	新しいインタフェースの MAC アドレス
host_name	string	ランダムに割り当て	no	ホスト内でのインタフェースの名前
limits.ingress	string	-	no	入力トラフィックの I/O 制限値（さまざまな単位が使用可能、下記参照）
limits.egress	string	-	no	出力トラフィックの I/O 制限値（さまざまな単位が使用可能、下記参照）
limits.max	string	-	no	`limits.ingress` と `limits.egress` の両方を同じ値に変更する
ipv4.routes	string	-	no	ホスト上で NIC に追加する IPv4 静的ルートのカンマ区切りリスト
ipv6.routes	string	-	no	ホスト上で NIC に追加する IPv6 静的ルートのカンマ区切りリスト
boot.priority	integer	-	no	VM のブート優先度 (高いほうが先にブート)

nic: routed

サポートされるインスタンスタイプ: コンテナ, VM

この NIC の指定に使えるプロパティ: `nictype`

この NIC タイプは運用上は IPVLAN に似ていて、ブリッジを作成することなくホストの MAC アドレスを共用してインスタンスが外部ネットワークに参加できるようにします。

しかしカーネルに IPVLAN サポートを必要としないこととホストとインスタンスが互いに通信できることが IPVLAN とは異なります。

さらにホスト上の netfilter のルールを尊重し、ホストのルーティングテーブルを使ってパケットをルーティングしますのでホストが複数のネットワークに接続している場合に役立ちます。

IP アドレスは `ipv4.address` と `ipv6.address` の設定のいずれかあるいは両方を使って、インスタンスが起動する前に手動で指定する必要があります。

コンテナでは veth ペアを使用し、VM では TAP デバイスを使用します。そしてホスト側の veth の上に下記のリンクローカルゲートウェイ IP アドレスを設定し、それらをインスタンス内のデフォルトゲートウェイに設定します。

169.254.0.1 fe80::1

コンテナではこれらはインスタンスの NIC インタフェースのデフォルトゲートウェイに自動的に設定されます。しかし VM では IP アドレスとデフォルトゲートウェイは手動か cloud-init のような仕組みを使って設定する必要があります。

またお使いのコンテナイメージがインタフェースに対して DHCP を使うように設定されている場合、上記の自動的に追加される設定は削除される可能性が高く、その後手動か cloud-init のような仕組みを使って設定する必要があることにもご注意ください。

次にインスタンスの IP アドレス全てをインスタンスの veth インタフェースに向ける静的ルートをホスト上に設定します。

この nic は `parent` のネットワークインタフェースのセットがあってもなくても利用できます。

`parent` ネットワークインタフェースのセットがある場合、インスタンスの IP の ARP/NDP のプロキシエントリーが親のインタフェースに追加され、インスタンスが親のインタフェースのネットワークにレイヤ 2 で参加できるようにします。

DNS に関してはネームサーバは自動的には設定されないので、インスタンス内で設定する必要があります。

次の sysctl の設定が必要です。

IPv4 アドレスを使用する場合は

```
net.ipv4.conf.<parent>.forwarding=1
```

IPv6 アドレスを使用する場合は

```
net.ipv6.conf.all.forwarding=1
net.ipv6.conf.<parent>.forwarding=1
net.ipv6.conf.all.proxy_ndp=1
net.ipv6.conf.<parent>.proxy_ndp=1
```

それぞれの NIC デバイスに複数の IP アドレスを追加できます。しかし複数の routed NIC インターフェースを使うほうが望ましいかもしれません。その場合はデフォルトゲートウェイの衝突を避けるため、後続のインターフェースで ipv4.gateway と ipv6.gateway の値を none に設定するべきです。さらにこれらの後続のインタフェースには ipv4.host_address と ipv6.host_address を用いて異なるホスト側のアドレスを設定することが有用かもしれません。

デバイス設定プロパティ

Key	Type	Default	Required	Description
parent	string	-	no	インスタンスが参加するホストデバイス名
name	string	カーネルが割り当て	no	インスタンス内でのインタフェース名
host_name	string	ランダムに割り当て	no	ホスト内でのインターフェース名
mtu	integer	親の MTU	no	新しいインタフェースの MTU
hwaddr	string	ランダムに割り当て	no	新しいインタフェースの MAC アドレス
limits.ingress	string	-	no	内向きトラフィックに対する bit/s での I/O 制限（さまざまな単位をサポート、下記参照）
limits.egress	string	-	no	外向きトラフィックに対する bit/s での I/O 制限（さまざまな単位をサポート、下記参照）
limits.max	string	-	no	limits.ingress と limits.egress の両方を指定するのと同じ
ipv4.routes	string	-	no	ホスト上で NIC に追加する IPv4 静的ルートのカンマ区切りリスト（L2 ARP/NDP プロキシを除く）
ipv4.address	string	-	no	インスタンスに追加する IPv4 静的アドレスのカンマ区切りリスト
ipv4.gateway	string	auto	no	自動的に IPv4 のデフォルトゲートウェイを追加するかどうか（ auto か none を指定可能）
ipv4.host_address	string	169.254.0.1	no	ホスト側の veth インターフェースに追加する IPv4 アドレス
ipv4.host_table	integer	-	no	（メインのルーティングテーブルに加えて） IPv4 の静的ルートを追加する先のルーティングテーブル ID
ipv4.neighbor_probe	boolean	true	no	IP アドレスが利用可能か知るために親のネットワークを調べるかどうか
ipv6.address	string	-	no	インスタンスに追加する IPv6 静的アドレスのカンマ区切りリスト
ipv6.routes	string	-	no	ホスト上で NIC に追加する IPv6 静的ルートのカンマ区切りリスト（L2 ARP/NDP プロキシを除く）
ipv6.gateway	string	auto	no	自動的に IPv6 のデフォルトゲートウェイを追加するかどうか（ auto か none を指定可能）
ipv6.host_address	string	fe80::1	no	ホスト側の veth インターフェースに追加する IPv6 アドレス
ipv6.host_table	integer	-	no	（メインのルーティングテーブルに加えて） IPv6 の静的ルートを追加する先のルーティングテーブル ID
ipv6.neighbor_probe	boolean	true	no	IP アドレスが利用可能か知るために親のネットワークを調べるかどうか
vlan	integer	-	no	アタッチ先の VLAN ID
gvrp	boolean	false	no	GARP VLAN Registration Protocol を使って VLAN を登録する

ブリッジ、ipvlan、macvlan を使った物理ネットワークへの接続

bridged、ipvlan、macvlan インターフェースタイプのいずれも、既存の物理ネットワークへ接続できます。

macvlan は、物理 NIC を効率的に分岐できます。つまり、物理 NIC からインスタンスで使える第 2 のインターフェースを取得できます。macvlan を使うことで、ブリッジデバイスと veth ペアの作成を減らせますし、通常はブリッジよりも良いパフォーマンスが得られます。

macvlan の欠点は、macvlan は外部との間で通信はできますが、自身の親デバイスとは通信できないことです。つまりインスタンスとホストが通信する必要がある場合は macvlan は使えません。

そのような場合は、ブリッジを選ぶのが良いでしょう。macvlan では使えない MAC フィルタリングと I/O 制限も使えます。

ipvlan は macvlan と同様ですが、フォークされたデバイスが静的に割り当てられた IP アドレスを持ち、ネットワーク上の親の MAC アドレスを受け継ぐ点が異なります。

SR-IOV

sriov インターフェースタイプで、SR-IOV が有効になったネットワークデバイスを使えます。このデバイスは、複数の仮想ファンクション（Virtual Functions: VFs）をネットワークデバイスの単一の物理ファンクション（Physical Function: PF）に関連付けます。PF は標準の PCIe ファンクションです。一方、VFs は非常に軽量な PCIe ファンクションで、データの移動に最適化されています。VFs は PF のプロパティを変更できないように、制限された設定機能のみを持っています。VFs は通常の PCIe デバイスとしてシステム上に現れるので、通常の物理デバイスと同様にインスタンスに与えることができます。sriov インターフェースタイプは、システム上の SR-IOV が有効になったネットワークデバイス名が、parent プロパティに設定されることを想定しています。すると LXD は、システム上で使用可能な VFs があるかどうかをチェックします。デフォルトでは、LXD は検索で最初に見つかった使われていない VF を割り当てます。有効になった VF が存在しないか、現時点で有効な VFs がすべて使われている場合は、サポートされている VF 数の最大値まで有効化し、最初の使用可能な VF をつかいます。もしすべての使用可能な VF が使われているか、カーネルもしくはカードが VF 数を増加させられない場合は、LXD はエラーを返します。

sriov ネットワークデバイスは次のように作成します:

```
lxc config device add <instance> <device-name> nic nictype=sriov parent=<sriov-enabled-
↪device>
```

特定の未使用な VF を使うように LXD に指示するには、host_name プロパティを追加し、有効な VF 名を設定します。

MAAS を使った統合管理

もし、LXD ホストが接続されている物理ネットワークを MAAS を使って管理している場合で、インスタンスを直接 MAAS が管理するネットワークに接続したい場合は、MAAS とやりとりをしてインスタンスをトラッキングするように LXD を設定できます。

そのためには、デーモンに対して、`maas.api.url` と `maas.api.key` を設定しなければなりません。そして、`maas.subnet.ipv4` と `maas.subnet.ipv6` の両方またはどちらかを、インスタンスもしくはプロファイルの `nic` エントリーに設定します。

これで、LXD はすべてのインスタンスを MAAS に登録し、適切な DHCP リースと DNS レコードがインスタンスに与えられます。

`ipv4.address` もしくは `ipv6.address` を設定した場合は、MAAS 上でも静的な割り当てとして登録されます。

Type: infiniband

サポートされるインスタンスタイプ: コンテナ

LXD では、InfiniBand デバイスに対する 2 種類の異なったネットワークタイプが使えます:

- `physical`: ホストの物理デバイスをパススルーで直接使います。対象のデバイスはホスト上では見えなくなり、インスタンス内に出現します
- `sriov`: SR-IOV が有効な物理ネットワークデバイスの仮想ファンクション（virtual function）をインスタンスに与えます

ネットワークインターフェースの種類が異なると追加のプロパティが異なります。現時点のリストは次の通りです:

Key	Type	Default	Required	Used by	Description
nictype	string	-	yes	all	デバイスタイプ。physical か sriov のいずれか
name	string	カーネルが割り当て	no	all	インスタンス内部でのインターフェース名
hwaddr	string	ランダムに割り当て	no	all	新しいインターフェースの MAC アドレス。 20 バイト全てを指定するか短い 8 バイト (この場合親デバイスの最後の 8 バイトだけを変更) のどちらかを設定可能
mtu	integer	親の MTU	no	all	新しいインターフェースの MTU
parent	string	-	yes	physical, sriov	ホスト上のデバイス、ブリッジの名前

physical な infiniband デバイスを作成するには次のように実行します:

```
lxc config device add <instance> <device-name> infiniband nictype=physical parent=
↪<device>
```

InfiniBand デバイスでの SR-IOV

InfiniBand デバイスは SR-IOV をサポートしますが、他の SR-IOV と違って、SR-IOV モードでの動的なデバイスの作成はできません。つまり、カーネルモジュール側で事前に仮想ファンクション（virtual functions）の数を設定する必要があるということです。

sriov の infiniband でデバイスを作るには次のように実行します:

```
lxc config device add <instance> <device-name> infiniband nictype=sriov parent=<sriov-
↪enabled-device>
```

Type: disk

サポートされるインスタンスタイプ: コンテナ, VM

ディスクエントリーは基本的にインスタンス内のマウントポイントです。ホスト上の既存ファイルやディレクトリのバインドマウントでも構いませんし、ソースがブロックデバイスであるなら、通常のマウントでも構いません。

LXD では以下の追加のソースタイプをサポートします。

- Ceph-rbd: 外部で管理されている既存の ceph RBD デバイスからマウントします。 LXD は ceph をインスタンスの内部のファイルシステムを管理するのに使用できます。ユーザーが事前に既存の ceph RBD を持っておりそれをインスタンスに使いたい場合はこのコマンドを使用できます。コマンド例

```
lxc config device add <instance> ceph-rbd1 disk source=ceph:<my_pool>/<my-volume> ceph.
↪user_name=<username> ceph.cluster_name=<username> path=/ceph
```

- Ceph-fs: 外部で管理されている既存の ceph FS からマウントします。 LXD は ceph をインスタンスの内部のファイルシステムを管理するのに使用できます。ユーザーが事前に既存の ceph ファイルシステムを持っておりそれをインスタンスに使いたい場合はこのコマンドを使用できます。コマンド例

```
lxc config device add <instance> ceph-fs1 disk source=cephfs:<my-fs>/<some-path> ceph.
↪user_name=<username> ceph.cluster_name=<username> path=/cephfs
```

- VM cloud-init: user.vendor-data, user.user-data と user.meta-data 設定キーから cloud-init 設定の ISO イメージを生成し VM にアタッチできるようにします。この ISO イメージは VM 内で動作する cloud-init が起動時にドライバを検出し設定を適用します。仮想マシンのインスタンスでのみ利用可能です。コマンド例

```
lxc config device add <instance> config disk source=cloud-init:config
```

現状では仮想マシンではルートディスク (path=/) と設定ドライブ (source=cloud-init:config) のみがサポートされます。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
limits.read	string	-	no	byte/s（さまざまな単位が使用可能、下記参照）もしくは iops（あとに "iops" と付けなければなりません）で指定する読み込みの I/O 制限値
limits.write	string	-	no	byte/s（さまざまな単位が使用可能、下記参照）もしくは iops（あとに "iops" と付けなければなりません）で指定する書き込みの I/O 制限値
limits.max	string	-	no	`limits.read` と `limits.write` の両方を同じ値に変更する
path	string	-	yes	ディスクをマウントするインスタンス内のパス
source	string	-	yes	ファイル・ディレクトリ、もしくはブロックデバイスのホスト上のパス
required	boolean	true	no	ソースが存在しないときに失敗とするかどうかを制御する
readonly	boolean	false	no	マウントを読み込み専用とするかどうかを制御する
size	string	-	no	byte（さまざまな単位が使用可能、下記参照す）で指定するディスクサイズ。rootfs（/）でのみサポートされます
size.state	string	-	no	上の size と同じですが仮想マシン内のランタイム状態を保存するために使われるファイルシステムボリュームに適用されます
recursive	boolean	false	no	ソースパスを再帰的にマウントするかどうか
pool	string	-	no	ディスクデバイスが属するストレージプール。LXD が管理するストレージボリュームにのみ適用されます
propagation	string	-	no	バインドマウントをインスタンスとホストでどのように共有するかを管理する（デフォルトである `private`, `shared`, `slave`, `unbindable`, `rshared`, `rslave`, `runbindable`, `rprivate` のいずれか。詳しくは Linux kernel の文書 shared subtree をご覧ください）
shift	boolean	false	no	ソースの uid/gid をインスタンスにマッチするように変換させるためにオーバーレイの shift を設定するか（コンテナのみ）
raw.mount.options	string	-	no	ファイルシステム固有のマウントオプション
ceph.user_name	string	admin	no	ソースが ceph か cephfs の場合に適切にマウントするためにユーザーが ceph user_name を指定しなければなりません
ceph.cluster_name	string	ceph	no	ソースが ceph か cephfs の場合に適切にマウントするためにユーザーが ceph cluster_name を指定しなければなりません
boot.priority	integer	-	no	VM のブート優先度 (高いほうが先にブート)

Type: unix-char

サポートされるインスタンスタイプ: コンテナ

UNIX キャラクターデバイスエントリーは、シンプルにインスタンスの /dev に、リクエストしたキャラクターデバイスを出現させます。そしてそれに対して読み書き操作を許可します。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
source	string	-	no	ホスト上でのパス
path	string	-	no	インスタンス内のパス（"source" と "path" のどちらかを設定しなければいけません）
major	int	device on host	no	デバイスのメジャー番号
minor	int	device on host	no	デバイスのマイナー番号
uid	int	0	no	インスタンス内のデバイス所有者の UID
gid	int	0	no	インスタンス内のデバイス所有者の GID
mode	int	0660	no	インスタンス内のデバイスのモード
required	boolean	true	no	このデバイスがインスタンスの起動に必要かどうか

Type: unix-block

サポートされるインスタンスタイプ: コンテナ

UNIX ブロックデバイスエントリーは、シンプルにインスタンスの /dev に、リクエストしたブロックデバイスを出現させます。そしてそれに対して読み書き操作を許可します。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
source	string	-	no	ホスト上のパス
path	string	-	no	インスタンス内のパス（"source" と "path" のどちらかを設定しなければいけません）
major	int	device on host	no	デバイスのメジャー番号
minor	int	device on host	no	デバイスのマイナー番号
uid	int	0	no	インスタンス内のデバイス所有者の UID
gid	int	0	no	インスタンス内のデバイス所有者の GID
mode	int	0660	no	インスタンス内のデバイスのモード
required	boolean	true	no	このデバイスがインスタンスの起動に必要かどうか

Type: usb

サポートされるインスタンスタイプ: コンテナ, VM

USB デバイスエントリーは、シンプルにリクエストのあった USB デバイスをインスタンスに出現させます。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
vendorid	string	-	no	USB デバイスのベンダー ID
productid	string	-	no	USB デバイスのプロダクト ID
uid	int	0	no	インスタンス内のデバイス所有者の UID
gid	int	0	no	インスタンス内のデバイス所有者の GID
mode	int	0660	no	インスタンス内のデバイスのモード
required	boolean	false	no	このデバイスがインスタンスの起動に必要かどうか（デフォルトは false で、すべてのデバイスがホットプラグ可能です）

Type: gpu

GPU デバイスエントリーは、シンプルにリクエストのあった GPU デバイスをインスタンスに出現させます。

注釈: コンテナデバイスは、同時に複数の GPU とマッチングさせることができます。しかし、仮想マシンの場合、デバイスは 1 つの GPU にしかマッチしません。

利用可能な GPU

以下の GPU が `gputype` プロパティを使って指定できます。

- *physical* GPU 全体をパススルーします。 `gputype` が指定されない場合これがデフォルトです。
- *mdev* 仮想 GPU を作成しインスタンスにパススルーします。
- *mig* MIG (Multi-Instance GPU) を作成しインスタンスにパススルーします。
- *sriov* SR-IOV を有効にした GPU の仮想ファンクション（virtual function）をインスタンスに与えます。

gpu: physical

サポートされるインスタンスタイプ: コンテナ, VM

GPU 全体をパススルーします。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
vendorid	string	-	no	GPU デバイスのベンダー ID
productid	string	-	no	GPU デバイスのプロダクト ID
id	string	-	no	GPU デバイスのカード ID
pci	string	-	no	GPU デバイスの PCI アドレス
uid	int	0	no	インスタンス（コンテナのみ）内のデバイス所有者の UID
gid	int	0	no	インスタンス（コンテナのみ）内のデバイス所有者の GID
mode	int	0660	no	インスタンス（コンテナのみ）内のデバイスのモード

gpu: mdev

サポートされるインスタンスタイプ: VM

仮想 GPU を作成しインスタンスにパススルーします。利用可能な mdev プロファイルの一覧は `lxc info --resources` を実行すると確認できます。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
vendorid	string	-	no	GPU デバイスのベンダー ID
productid	string	-	no	GPU デバイスのプロダクト ID
id	string	-	no	GPU デバイスのカード ID
pci	string	-	no	GPU デバイスの PCI アドレス
mdev	string	-	yes	使用する mdev プロファイル（例: i915-GVTg_V5_4）

gpu: mig

サポートされるインスタンスタイプ: コンテナ

MIG コンピュートインスタンスを作成しパススルーします。現状これは NVIDIA MIG を事前に作成しておく必要があります。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
vendorid	string	-	no	GPU デバイスのベンダー ID
productid	string	-	no	GPU デバイスのプロダクト ID
id	string	-	no	GPU デバイスのカード ID
pci	string	-	no	GPU デバイスの PCI アドレス
mig.ci	int	-	no	既存の MIG コンピュートインスタンス ID
mig.gi	int	-	no	既存の MIG GPU インスタンス ID
mig.uuid	string	-	no	既存の MIG デバイス UUID ("MIG-" 接頭辞は省略可)

注意: "mig.uuid" (Nvidia drivers 470+) か、 "mig.ci" と "mig.gi" (古い Nvidia ドライバー) の両方を設定する必要があります。

gpu: sriov

サポートされるインスタンスタイプ: VM

SR-IOV が有効な GPU の仮想ファンクション（virtual function）をインスタンスに与えます。

次に挙げるプロパティがあります:

Key	Type	Default	Required	Description
vendorid	string	-	no	GPU デバイスのベンダー ID
productid	string	-	no	GPU デバイスのプロダクト ID
id	string	-	no	GPU デバイスのカード ID
pci	string	-	no	GPU デバイスの PCI アドレス

Type: proxy

サポートされるインスタンスタイプ: コンテナ（nat と 非 nat モード）、 VM （nat モードのみ）

プロキシデバイスにより、ホストとインスタンス間のネットワーク接続を転送できます。このデバイスを使って、ホストのアドレスの一つに到達したトラフィックをインスタンス内のアドレスに転送したり、その逆を行ったりして、ホストを通してインスタンス内にアドレスを持てます。

利用できる接続タイプは次の通りです:

- tcp <-> tcp
- udp <-> udp
- unix <-> unix
- tcp <-> unix
- unix <-> tcp
- udp <-> tcp
- tcp <-> udp
- udp <-> unix
- unix <-> udp

プロキシデバイスは nat モードもサポートします。nat モードではパケットは別の接続を通してプロキシされるのではなく NAT を使ってフォワードされます。これはターゲットの送り先が PROXY プロトコル（非 nat モードでプロキシデバイスを使う場合はこれはクライアントアドレスを渡す唯一の方法です）をサポートする必要なく、クライアントのアドレスを維持できるという利点があります。

プロキシデバイスを nat=true に設定する際は、以下のようにターゲットのインスタンスが NIC デバイス上に静的 IP を持つよう LXD で設定する必要があります。

```
lxc config device set <instance> <nic> ipv4.address=<ipv4.address> ipv6.address=<ipv6.
↪address>
```

静的な IPv6 アドレスを設定するためには、親のマネージドネットワークは ipv6.dhcp.stateful を有効にする必要があります。

NAT モードでサポートされる接続のタイプは以下の通りです。

- tcp <-> tcp
- udp <-> udp

IPv6 アドレスを設定する場合は以下のような角括弧の記法を使います。

```
connect=tcp:[2001:db8::1]:80
```

connect のアドレスをワイルドカード (IPv4 では 0.0.0.0 、 IPv6 では [::] にします）に設定することで、インスタンスの IP アドレスを指定できます。

listen のアドレスも非 NAT モードではワイルドカードのアドレスが使用できます。しかし nat モードを使う際は LXD ホスト上の IP アドレスを指定する必要があります。

Key	Type	Default	Required	Description
listen	string	-	yes	バインドし、接続を待ち受けるアドレスとポート (<type>:<addr>:<port>[-<port>][,<port>])
connect	string	-	yes	接続するアドレスとポート (<type>:<addr>:<port>[-<port>][,<port>])
bind	string	host	no	ホスト/インスタンスのどちら側にバインドするか
uid	int	0	no	listen する Unix ソケットの所有者の UID
gid	int	0	no	listen する Unix ソケットの所有者の GID
mode	int	0644	no	listen する Unix ソケットのモード
nat	bool	false	no	NAT 経由でプロキシを最適化するかどうか（インスタンスの NIC が静的 IP を持つ必要あり）
proxy_protocol	bool	false	no	送信者情報を送信するのに HAProxy の PROXY プロトコルを使用するかどうか
security.uid	int	0	no	特権を落とす UID
security.gid	int	0	no	特権を落とす GID

```
lxc config device add <instance> <device-name> proxy listen=<type>:<addr>:<port>[-<port>
↪][,<port>] connect=<type>:<addr>:<port> bind=<host/instance>
```

Type: unix-hotplug

サポートされるインスタンスタイプ: コンテナ

Unix ホットプラグデバイスのエントリーは依頼された unix デバイスをインスタンスの /dev に出現させ、デバイスがホストシステムに存在する場合はデバイスへの読み書き操作を許可します。実装はホスト上で稼働する systemd-udev に依存します。

以下の設定があります。

Key	Type	Default	Required	Description
vendorid	string	-	no	unix デバイスのベンダー ID
productid	string	-	no	unix デバイスの製品 ID
uid	int	0	no	インスタンス内でのデバイスオーナーの UID
gid	int	0	no	インスタンス内でのデバイスオーナーの GID
mode	int	0660	no	インスタンス内でのデバイスのモード {
required	boolean	false	no	このデバイスがインスタンスを起動するのに必要かどうか。(デフォルトは false で全てのデバイスはホットプラグ可能です)

Type: tpm

サポートされるインスタンスタイプ: コンテナ, VM

TPM デバイスのエントリーは TPM エミュレーターへのアクセスを可能にします。

以下の設定があります。

Key	Type	Default	Required	Description
path	string	-	yes	インスタンス内でのパス（コンテナのみ）

Type: pci

サポートされるインスタンスタイプ: VM

PCI デバイスエントリーは生の PCI デバイスをホストから仮想マシンに渡すために使用されます。

以下の設定があります。

Key	Type	Default	Required	Description
address	string	-	yes	デバイスの PCI アドレス

ストレージとネットワーク制限の単位

バイト数とビット数を表す値は全ていくつかの有用な単位を使用し特定の制限がどういう値かをより理解しやすいようにできます。

10 進と 2 進 (kibi) の単位の両方がサポートされており、後者は主にストレージの制限に有用です。

現在サポートされているビットの単位の完全なリストは以下の通りです。

- bit (1)
- kbit (1000)
- Mbit (1000^2)
- Gbit (1000^3)
- Tbit (1000^4)
- Pbit (1000^5)
- Ebit (1000^6)
- Kibit (1024)
- Mibit (1024^2)
- Gibit (1024^3)
- Tibit (1024^4)
- Pibit (1024^5)
- Eibit (1024^6)

現在サポートされているバイトの単位の完全なリストは以下の通りです。

- B または bytes (1)

- kB (1000)
- MB (1000^2)
- GB (1000^3)
- TB (1000^4)
- PB (1000^5)
- EB (1000^6)
- KiB (1024)
- MiB (1024^2)
- GiB (1024^3)
- TiB (1024^4)
- PiB (1024^5)
- EiB (1024^6)

インスタンスタイプ

LXD ではシンプルなインスタンスタイプが使えます。これは、インスタンスの作成時に指定できる文字列で表されます。

3 つの指定方法があります:

- `<instance type>`
- `<cloud>:<instance type>`
- `c<CPU>-m<RAM in GB>`

例えば、次の 3 つは同じです:

- t2.micro
- aws:t2.micro
- c1-m1

コマンドラインでは、インスタンスタイプは次のように指定します:

```
lxc launch ubuntu:20.04 my-instance -t t2.micro
```

使えるクラウドとインスタンスタイプのリストは次をご覧ください:

https://github.com/dustinkirkland/instance-type

limits.hugepages.[size] を使った hugepage の制限

LXD では limits.hugepage.[size] キーを使ってコンテナが利用できる hugepage の数を制限できます。hugepage の制限は hugetlb cgroup コントローラーを使って行われます。これはつまりこれらの制限を適用するためにホストシステムが hugetlb コントローラーを legacy あるいは unified cgroup の階層に公開する必要があることを意味します。アーキテクチャーによって複数の hugepage のサイズを公開していることに注意してください。さらに、アーキテクチャーによっては他のアーキテクチャーとは異なる hugepage のサイズを公開しているかもしれません。

hugepage の制限は非特権コンテナ内で hugetlbfs ファイルシステムの mount システムコールをインターセプトするように LXD を設定しているときには特に有用です。LXD が hugetlbfs mount システムコールをインターセプトすると LXD は正しい uid と gid の値を mount オプションに指定して hugetblfs ファイルシステムをコンテナにマウントします。これにより非特権コンテナからも hugepage が利用可能となります。しかし、ホストで利用可能な hugepage をコンテナが使い切ってしまうのを防ぐため、limits.hugepages.[size] を使ってコンテナが利用可能な hugepage の数を制限することを推奨します。

limits.kernel.[limit name] を使ったリソース制限

LXD では、指定したインスタンスのリソース制限を設定するのに、 limits.kernel.* という名前空間のキーが使えます。LXD は limits.kernel.* のあとに指定されるキーのリソースについての妥当性の確認は一切行ないません。LXD は、使用中のカーネルで、指定したリソースがすべてが使えるのかどうかを知ることができません。LXD は単純に limits.kernel.* の後に指定されるリソースキーと値をカーネルに渡すだけです。カーネルが適切な確認を行います。これにより、ユーザーは使っているシステム上で使えるどんな制限でも指定できます。いくつか一般的に使える制限は次の通りです:

Key	Resource	Description
limits.kernel.as	RLIMIT_AS	プロセスの仮想メモリーの最大サイズ
limits.kernel.core	RLIMIT_CORE	プロセスのコアダンプファイルの最大サイズ
limits.kernel.cpu	RLIMIT_CPU	プロセスが使える CPU 時間の秒単位の制限
limits.kernel.data	RLIMIT_DATA	プロセスのデーターセグメントの最大サイズ
limits.kernel.fsize	RLIMIT_FSIZE	プロセスが作成できるファイルの最大サイズ
limits.kernel.locks	RLIMIT_LOCKS	プロセスが確立できるファイルロック数の制限
limits.kernel.memlock	RLIMIT_MEMLOCK	プロセスが RAM 上でロックできるメモリのバイト数の制限
limits.kernel.nice	RLIMIT_NICE	引き上げることができるプロセスの nice 値の最大値
limits.kernel.nofile	RLIMIT_NOFILE	プロセスがオープンできるファイルの最大値
limits.kernel.nproc	RLIMIT_NPROC	呼び出し元プロセスのユーザーが作れるプロセスの最大数
limits.kernel.rtprio	RLIMIT_RTPRIO	プロセスに対して設定できるリアルタイム優先度の最大値
limits.kernel.sigpending	RLIMIT_SIGPENDING	呼び出し元プロセスのユーザーがキューに入れられるシグナルの最大数

指定できる制限の完全なリストは `getrlimit(2)/setrlimit(2)` システムコールの man ページで確認できます。`limits.kernel.*` 名前空間内で制限を指定するには、`RLIMIT_` を付けずに、リソース名を小文字で指定します。例えば、`RLIMIT_NOFILE` は `nofile` と指定します。制限は、コロン区切りのふたつの数字もしくは `unlimited` という文字列で指定します（例: `limits.kernel.nofile=1000:2000`）。単一の値を使って、ソフトリミットとハードリミットを同じ値に設定できます（例: `limits.kernel.nofile=3000`）。明示的に設定されないリソースは、インスタンスを起動したプロセスから継承されます。この継承は LXD でなく、カーネルによって強制されます。

スナップショットの定期実行と設定

LXD は 1 分毎に最大 1 回作成可能なスナップショットの定期実行をサポートします。3 つの設定項目があります。

- `snapshots.schedule` には短縮された cron 書式: <**分**> <**時**> <**日**> <**月**> <**曜日**> を指定します。これが空 (デフォルト) の場合はスナップショットは作成されません。
- `snapshots.schedule.stopped` は自動的にスナップショットを作成する際にインスタンスを停止するかどうかを制御します。デフォルトは `false` です。
- `snapshots.pattern` は pongo2 のテンプレート文字列を指定し、 pongo2 のコンテキストには `creation_date` 変数を含みます。スナップショットの名前に禁止された文字が含まれないように日付をフォーマットする (例: `{{ creation_date|date:"2006-01-02_15-04-05" }}`) べきであることに注意してください。名前の衝突を防ぐ別の方法はプレースホルダ `%d` を使うことです。(プレースホルダを除いて) 同じ名前のスナップショットが既に存在する場合、既存の全てのスナップショットの名前を考慮に入れてプレースホルダの最大の番号を見つけます。新しい名前にはこの番号を 1 増やしたものになります。スナップショットが存在しない場合の開始番号は `0` になります。 `snapshots.pattern` のデフォルトの挙動は `snap%d` のフォーマット文字列と同じです。

pongo2 の文法を使ってスナップショット名にタイムスタンプを含める例:

```
lxc config set INSTANCE snapshots.pattern "{{ creation_date|date:'2006-01-02_15-04-05' }}
↪"
```

これにより作成日時 {date/time of creation} を秒の精度まで含んだスナップショット名になります。

3.2.3 ネットワーク設定

- **ブリッジ**: インスタンスを接続する L2 ブリッジを作成（ローカルの DHCP と DNS を提供可能）。これがデフォルトです。
- *macvlan*: インスタンスを親の macvlan インターフェースに接続する際に使用するプリセットの設定を提供。
- *sriov*: インスタンスを親の SR-IOV インターフェースに接続する際に使用するプリセットの設定を提供。
- *ovn*: OVN SDN (software defined network) システムを使って論理的なネットワークを作成。
- **物理**: OVN ネットワークを親のインターフェースに接続する際に使用するプリセットの設定を提供。

希望するタイプは以下のように --type 引数で指定できます。

```
lxc network create <name> --type=bridge [options...]
```

--type 引数が指定されない場合は、デフォルトの bridge が使用されます。

設定キーは現状では全てのネットワークタイプでサポートされている以下のネームスペースによって名前空間が分けられています。

- maas (MAAS ネットワーク識別)
- user (ユーザーのメタデータに対する自由形式の key/value)

ネットワーク: ブリッジ

LXD でのネットワークの設定タイプの 1 つとして、 LXD はネットワークブリッジの作成と管理をサポートしています。LXD のブリッジは下層のネイティブな Linux のブリッジと Open vSwitch を利用できます。

LXD のブリッジの作成と管理は lxc network コマンドで行えます。LXD で作成されたブリッジはデフォルトでは "managed" です。というのは LXD はさらにローカルの dnsmasq DHCP サーバをセットアップし、希望すれば (これがデフォルトです) ブリッジに対して NAT も行います。

ブリッジが LXD に管理されているときは、 bridge ネームスペースを使って設定値を変更できます。

さらに、 LXD は既存の Linux ブリッジを利用することも出来ます。この場合、ブリッジは lxc network で作成する必要はなく、インスタンスかプロファイルのデバイス設定で下記のように単に参照できます。

```
devices:
  eth0:
     name: eth0
     nictype: bridged
     parent: br0
     type: nic
```

ネットワークフォワード:

ブリッジのネットワークサポートは **ネットワークフォワード** 参照。

ネットワークの設定プロパティ:

LXD のネットワークの設定項目の完全なリストは以下の通りです。

ブリッジネットワークでは以下の設定キーネームスペースが現状サポートされています。

- `bridge` (L2 インタフェースの設定)
- `fan` (Ubuntu FAN overlay に特有な設定)
- `tunnel` (ホスト間のトンネリングの設定)
- `ipv4` (L3 IPv4 設定)
- `ipv6` (L3 IPv6 設定)
- `dns` (DNS サーバと名前解決の設定)
- `raw` (raw の設定のファイルの内容)

IP アドレスとサブネットは CIDR 形式 (`1.1.1.1/24` や `fd80:1234::1/64`) で指定することを想定しています。

例外としてトンネルのローカルとリモートのアドレスは単なるアドレス (`1.1.1.1` や `fd80:1234::1`) を指定します。

キー	型	条件	デフォルト	説明
bgp.peers.NAME.address	string	bgp server	-	ピアのアドレス (IPv4 か IPv6)
bgp.peers.NAME.asn	integer	bgp server	-	ピアの AS 番号
bgp.peers.NAME.password	string	bgp server	- (パスワード無し)	ピアのセッションパスワード（省
bgp.ipv4.nexthop	string	bgp server	ローカルアドレス	広告されたプリフィクスの next-h
bgp.ipv6.nexthop	string	bgp server	ローカルアドレス	広告されたプリフィクスの next-h
bridge.driver	string	-	native	ブリッジのドライバ ("native" か "
bridge.external_interfaces	string	-	-	ブリッジに含める未設定のネット
bridge.hwaddr	string	-	-	ブリッジの MAC アドレス

表 2 – 前のページから

キー	型	条件	デフォルト	説明
bridge.mode	string	-	standard	ブリッジの稼働モード ("standard"
bridge.mtu	integer	-	1500	ブリッジの MTU (tunnel か fan か
dns.domain	string	-	lxd	DHCP のクライアントに広告し D
dns.mode	string	-	managed	DNS の登録モード ("none" は DN
dns.search	string	-	-	完全なドメインサーチのカンマ区
dns.zone.forward	string	-	managed	正引き DNS レコード用の DNS ゾ
dns.zone.reverse.ipv4	string	-	managed	IPv4 逆引き DNS レコード用の D
dns.zone.reverse.ipv6	string	-	managed	IPv6 逆引き DNS レコード用の D
fan.overlay_subnet	string	ファンモード	240.0.0.0/8	FAN の overlay として使用するサ
fan.type	string	ファンモード	vxlan	FAN のトンネル・タイプ ("vxlan"
fan.underlay_subnet	string	ファンモード	自動（作成時のみ）	FAN の underlay として使用するサ
ipv4.address	string	標準モード	自動（作成時のみ）	ブリッジの IPv4 アドレス (CIDR
ipv4.dhcp	boolean	ipv4 アドレス	true	DHCP を使ってアドレスを割り当
ipv4.dhcp.expiry	string	ipv4 dhcp	1h	DHCP リースの有効期限
ipv4.dhcp.gateway	string	ipv4 dhcp	ipv4.address	サブネットのゲートウェイのアド
ipv4.dhcp.ranges	string	ipv4 dhcp	全てのアドレス	DHCP に使用する IPv4 の範囲 (開
ipv4.firewall	boolean	ipv4 アドレス	true	このネットワークに対するファイ
ipv4.nat.address	string	ipv4 アドレス	-	ブリッジからの送信時に使うソー
ipv4.nat	boolean	ipv4 アドレス	false	NAT にするかどうか（通常のブリ
ipv4.nat.order	string	ipv4 アドレス	before	必要な NAT のルールを既存のルー
ipv4.ovn.ranges	string	-	-	子供の OVN ネットワークルーター
ipv4.routes	string	ipv4 アドレス	-	ブリッジへルーティングする追加
ipv4.routing	boolean	ipv4 アドレス	true	ブリッジの内外にトラフィックを
ipv6.address	string	標準モード	自動（作成時のみ）	ブリッジの IPv6 アドレス (CIDR
ipv6.dhcp	boolean	ipv6 アドレス	true	DHCP 上で追加のネットワーク設
ipv6.dhcp.expiry	string	ipv6 dhcp	1h	DHCP リースの有効期限
ipv6.dhcp.ranges	string	ipv6 ステートフル dhcp	全てのアドレス	DHCP に使用する IPv6 の範囲 (開
ipv6.dhcp.stateful	boolean	ipv6 dhcp	false	DHCP を使ってアドレスを割り当
ipv6.firewall	boolean	ipv6 アドレス	true	このネットワークに対するファイ
ipv6.nat.address	string	ipv6 アドレス	-	ブリッジからの送信時に使うソー
ipv6.nat	boolean	ipv6 アドレス	false	NAT にするかどうか (未設定の場
ipv6.nat.order	string	ipv6 アドレス	before	必要な NAT のルールを既存のルー
ipv6.ovn.ranges	string	-	-	子供の OVN ネットワークルーター
ipv6.routes	string	ipv6 アドレス	-	ブリッジへルーティングする追加
ipv6.routing	boolean	ipv6 アドレス	true	ブリッジの内外にトラフィックを
maas.subnet.ipv4	string	ipv4 アドレス	-	インスタンスを登録する MAAS I
maas.subnet.ipv6	string	ipv6 アドレス	-	インスタンスを登録する MAAS I

表 2 – 前のページから

キー	型	条件	デフォルト	説明
raw.dnsmasq	string	-	-	設定に追加する dnsmasq の設定フ
tunnel.NAME.group	string	vxlan	239.0.0.1	vxlan のマルチキャスト設定 (loca
tunnel.NAME.id	integer	vxlan	0	vxlan トンネルに使用するトンネ
tunnel.NAME.interface	string	vxlan	-	トンネルに使用するホスト・イン
tunnel.NAME.local	string	gre か vxlan	-	トンネルに使用するローカルアド
tunnel.NAME.port	integer	vxlan	0	vxlan トンネルに使用するポート
tunnel.NAME.protocol	string	標準モード	-	トンネリングのプロトコル ("vxlan
tunnel.NAME.remote	string	gre か vxlan	-	トンネルに使用するリモートアド
tunnel.NAME.ttl	integer	vxlan	1	マルチキャストルーティングトポ
security.acls	string	-	-	このネットワークに接続された N
security.acls.default.ingress.action	string	security.acls	reject	どの ACL ルールにもマッチしない
security.acls.default.egress.action	string	security.acls	reject	どの ACL ルールにもマッチしない
security.acls.default.ingress.logged	boolean	security.acls	false	どの ACL ルールにもマッチしない
security.acls.default.egress.logged	boolean	security.acls	false	どの ACL ルールにもマッチしない

これらのキーは lxc コマンドで以下のように設定できます。

```
lxc network set <network> <key> <value>
```

systemd-resolved との統合

LXD が動いているシステムが DNS のルックアップに systemd-resolved を使用している場合、 LXD が名前解決できるドメインを systemd-resolved に指定することができます。これには systemd-resolved にどのブリッジ、ネームサーバのアドレス、そして DNS ドメインかを伝える必要があります。

例えば、 LXD が `lxdbr0` インターフェースを使用している場合、 `lxc network get lxdbr0 ipv4.address` コマンドで IPv4 アドレス（IPv4 アドレスの代わりに IPv6 アドレスを使うこともできますし、 IPv4 アドレスと IPv6 アドレスの両方を使うこともできます）と `lxc network get lxdbr0 dns.domain`（ドメインが設定されていない場合は上記の表に示されているデフォルト値の `lxd` が使用されます）でドメインを取得します。そして systemd-resolved に以下のように指定します。

```
systemd-resolve --interface lxdbr0 --set-domain '~lxd' --set-dns n.n.n.n
```

上記の `lxdbr0` は実際のブリッジの名前に、 `n.n.n.n` はネームサーバの実際の（サブネットマスクを除いた） アドレスに置き換えて実行してください。

さらに `lxd` はドメイン名に置き換えてください。ドメイン名の前の ~ が重要ですので注意してください。~ はこのドメインだけをルックアップするためにこのネームサーバを使うように systemd-resolved に指定します。実際のドメイン名が何であるかにかかわらず ~ を前につけるべきです。また、 ~ という文字はシェルが展開するかもしれ

ないので、クォートに囲んでエスケープする必要があるかもしれません。

systemd のより新しいリリースでは systemd-resolve コマンドは deprecated になっていますが、(これを書いている時点では) 後方互換性のためまだ提供されています。systemd-resolved に伝えるための新しい方法は resolvectl コマンドを使うことです。これは以下の 2 ステップで実行します。

```
resolvectl dns lxdbr0 n.n.n.n
resolvectl domain lxdbr0 '~lxd'
```

この systemd-resolved の設定はブリッジが存在する間のみ存続します。ですので、リブートと LXD が再起動するたびにこのコマンドを繰り返し実行する必要があります（これを自動化するには下記を参照してください)。また、これはブリッジの dns.mode が none でないときにしか機能しないことに注意してください。

dns.domain の使用に依存する場合 DNS の名前解決ができるように resolved の DNSSEC を無効にする必要があるかもしれないことに注意してください。これは resolved.conf の DNSSEC オプションで設定できます。

LXD が lxdbr0 インターフェースを作成した場合、systemd-resolved の DNS 設定をシステム起動時に適用するのを自動化するには以下のような設定を含む systemd の unit ファイル /etc/systemd/system/lxd-dns-lxdbr0.service を作成する必要があります。

```
[Unit]
Description=LXD per-link DNS configuration for lxdbr0
BindsTo=sys-subsystem-net-devices-lxdbr0.device
After=sys-subsystem-net-devices-lxdbr0.device

[Service]
Type=oneshot
ExecStart=/usr/bin/resolvectl dns lxdbr0 n.n.n.n
ExecStart=/usr/bin/resolvectl domain lxdbr0 '~lxd'

[Install]
WantedBy=sys-subsystem-net-devices-lxdbr0.device
```

n.n.n.n を lxdbr0 ブリッジの IP アドレスで必ず置き換えてください。

自動起動を有効にし、起動するには以下のようにします。

```
sudo systemctl daemon-reload
sudo systemctl enable --now lxd-dns-lxdbr0
```

lxdbr0 インタフェースが既に存在する（例: LXD が実行中である場合など）場合、以下のようにサービスが起動済みかを確認できます。

```
sudo systemctl status lxd-dns-lxdbr0.service
● lxd-dns-lxdbr0.service - LXD per-link DNS configuration for lxdbr0
     Loaded: loaded (/etc/systemd/system/lxd-dns-lxdbr0.service; enabled; vendor preset:␣
↪enabled)
     Active: inactive (dead) since Mon 2021-06-14 17:03:12 BST; 1min 2s ago
    Process: 9433 ExecStart=/usr/bin/resolvectl dns lxdbr0 n.n.n.n (code=exited,␣
↪status=0/SUCCESS)
    Process: 9434 ExecStart=/usr/bin/resolvectl domain lxdbr0 ~lxd (code=exited,␣
↪status=0/SUCCESS)
   Main PID: 9434 (code=exited, status=0/SUCCESS)
```

次に設定が適用されているかを以下のように確認します。

```
sudo resolvectl status lxdbr0
Link 6 (lxdbr0)
      Current Scopes: DNS
DefaultRoute setting: no
       LLMNR setting: yes
MulticastDNS setting: no
  DNSOverTLS setting: no
      DNSSEC setting: no
    DNSSEC supported: no
  Current DNS Server: n.n.n.n
         DNS Servers: n.n.n.n
          DNS Domain: ~lxd
```

IPv6 プリフィクスサイズ

最適な動作には 64 のプリフィクスサイズが望ましいです。より大きなサブネット（ 64 より小さいプリフィクス）も正しく動作するでしょうが、SLAAC 環境下では有用ではないことが多いでしょう。

IPv6 アドレスの割り当てにステートフル DHCPv6 を使用している場合は、より小さなサブネットも理論的には利用可能ですが、 dnsmasq にきちんとサポートされておらず問題が起きるかもしれません。これらの 1 つをどうしても使わなければならない場合、静的割り当てか別のスタンドアロンの RA デーモンを使用可能です。

Firewalld で DHCP, DNS を許可する

firewalld を使用しているホストで LXD が実行する DHCP と DNS サーバにインスタンスがアクセスできるようにするには、ホストのブリッジインターフェースを firewalld の `trusted` ゾーンに追加する必要があります。

（リブート後も設定が残るように）恒久的にこれを行うには以下のコマンドを実行してください。

```
firewall-cmd --zone=trusted --change-interface=<LXD network name> --permanent
```

例えばブリッジネットワークが `lxdbr0` という名前の場合、以下のコマンドを実行します。

```
firewall-cmd --zone=trusted --change-interface=lxdbr0 --permanent
```

これにより LXD 自身のファイアーウォールのルールが有効になります。

Firewalld に LXD の iptables ルールを制御させるには

firewalld と LXD を一緒に使う場合、 iptables のルールがオーバーラップするかもしれません。例えば firewalld が LXD デーモンより後に起動すると firewalld が LXD の iptables ルールを削除し、 LXD コンテナが外向きのインターネットアクセスが全くできなくなるかもしれません。これを修正する 1 つの方法は LXD の iptables ルールを firewalld に移譲し、 LXD の iptables ルールは無効にすることです。

最初のステップは Firewalld で DHCP, DNS を許可する ことです。

次に LXD に iptables ルールを設定しないように（firewalld が設定するので）伝えます。

```
lxc network set lxdbr0 ipv4.nat false
lxc network set lxdbr0 ipv6.nat false
lxc network set lxdbr0 ipv6.firewall false
lxc network set lxdbr0 ipv4.firewall false
```

最後に firewalld のルールを LXD の利用ケースに応じて有効にします（この例では、ブリッジインターフェースが `lxdbr0` で付与されている IP の範囲が `10.0.0.0/24` だとしています）。

```
firewall-cmd --permanent --direct --add-rule ipv4 filter INPUT 0 -i lxdbr0 -s 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 filter OUTPUT 0 -o lxdbr0 -d 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 filter FORWARD 0 -i lxdbr0 -s 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 nat POSTROUTING 0 -s 10.0.0.0/24 ! -d␣
↪10.0.0.0/24 -m comment --comment "generated by firewalld for LXD" -j MASQUERADE
firewall-cmd --reload
```

firewalld にルールが設定されたかを確認するには以下のようにします。

```
firewall-cmd --direct --get-all-rules
```

警告：上記の手順はフールプルーフなアプローチではなく、不注意にセキュリティリスクをもたらすことにつながる可能性があります。

ネットワーク: macvlan

macvlan ネットワークタイプではインスタンスを macvlan NIC を使って親のインターフェースに接続する際に使用するプリセットを指定可能です。これによりインスタンスの NIC 自体は下層の詳しい設定を一切知ることなく、接続する `network` を単に指定するだけで設定できます。

ネットワーク設定プロパティ:

キー	型	条件	デフォルト	説明
maas.subnet.ipv4	string	ipv4 アドレス	-	インスタンスを登録する MAAS IPv4 サブネット（nic の `network` プロパティを使用する場合）
maas.subnet.ipv6	string	ipv6 アドレス	-	インスタンスを登録する MAAS IPv6 サブネット（nic の `network` プロパティを使用する場合）
mtu	integer	-	-	作成するインターフェースの MTU
parent	string	-	-	macvlan NIC を作成する親のインターフェース
vlan	integer	-	-	アタッチする先の VLAN ID
gvrp	boolean	-	false	GARP VLAN Registration Protocol を使って VLAN を登録する

ネットワーク: sriov

sriov ネットワークタイプではインスタンスを sriov NIC を使って親のインターフェースに接続する際に使用するプリセットを指定可能です。これによりインスタンスの NIC 自体は下層の詳しい設定を一切知ることなく、接続する `network` を単に指定するだけで設定できます。

ネットワーク設定プロパティ:

キー	型	条件	デフォルト	説明
maas.subnet.ipv4	string	ipv4 アドレス	-	インスタンスを登録する MAAS IPv4 サブネット（nic の network プロパティを使用する場合）
maas.subnet.ipv6	string	ipv6 アドレス	-	インスタンスを登録する MAAS IPv6 サブネット（nic の network プロパティを使用する場合）
mtu	integer	-	-	作成するインターフェースの MTU
parent	string	-	-	sriov NIC を作成する親のインターフェース
vlan	integer	-	-	アタッチする先の VLAN ID

ネットワーク: ovn

ovn ネットワークタイプは OVN SDN を使って論理的なネットワークの作成を可能にします。これは複数の個別のネットワーク内で同じ論理ネットワークのサブネットを使うような検証環境やマルチテナントの環境で便利です。

LXD の OVN ネットワークはより広いネットワークへの外向きのアクセスを可能にするため既存の管理された LXD のブリッジネットワークに接続できます。OVN 論理ネットワークからの全ての接続は親のネットワークによって割り当てられた動的 IP に NAT されます。

スタンドアロンの LXD での OVN の設定

これは外向きの通信のために親のネットワーク lxdbr0 に接続されたスタンドアロンの OVN ネットワークを作成する手順です。

OVN のツールをインストールし、ローカルノードで OVN の統合ブリッジを設定します。

```
sudo apt install ovn-host ovn-central
sudo ovs-vsctl set open_vswitch . \
  external_ids:ovn-remote=unix:/var/run/ovn/ovnsb_db.sock \
  external_ids:ovn-encap-type=geneve \
  external_ids:ovn-encap-ip=127.0.0.1
```

以下を使用して OVN ネットワークとインスタンスを作成します。

```
lxc network set lxdbr0 ipv4.dhcp.ranges=... ipv4.ovn.ranges=... # OVN ゲートウェイに IP のレンジを割り当て
lxc network create ovntest --type=ovn network=lxdbr0
lxc init images:ubuntu/20.04 c1
```

(次のページに続く)

(前のページからの続き)

```
lxc config device override c1 eth0 network=ovntest
lxc start c1
lxc ls
+------+---------+----------------------+-----------------------------------------------+--
↪---------+-----------+
| NAME |  STATE  |         IPV4         |                     IPV6                      |
 TYPE    | SNAPSHOTS |
+------+---------+----------------------+-----------------------------------------------+--
↪---------+-----------+
| c1   | RUNNING | 10.254.118.2 (eth0) | fd42:887:cff3:5089:216:3eff:fef0:549f (eth0) |␣
↪CONTAINER | 0         |
+------+---------+----------------------+-----------------------------------------------+--
↪---------+-----------+
```

ネットワークフォワード:

OVN のネットワークサポートは **ネットワークフォワード** 参照。

ネットワークピア:

OVN のネットワークピアサポートは **ネットワークピア** 参照。

ネットワークの設定プロパティ:

キー	型	条件	デフォルト	説明
bridge.hwaddr	string	-	-	ブリッジの MAC アドレス
bridge.mtu	integer	-	1442	ブリッジの MTU (デフォルトではホストからホストへの geneve トンネルを許可します)
dns.domain	string	-	lxd	DHCP のクライアントに広告し DNS の名前解決に使用するドメイン
dns.search	string	-	-	完全なドメインサーチのカンマ区切りリスト（デフォルトは `dns.domain` の値）
dns.zone.forward	string	-	-	正引き DNS レコード用の DNS ゾーン名
dns.zone.reverse.ipv4	string	-	-	IPv4 逆引き DNS レコード用の DNS ゾーン名
dns.zone.reverse.ipv6	string	-	-	IPv6 逆引き DNS レコード用の DNS ゾーン名
ipv4.address	string	標準モード	自動（作成時のみ）	ブリッジの IPv4 アドレス (CIDR 形式)。 IPv4 をオフにするには "none" 、新しいランダムな未使用のサブネットを生成するには "auto" を指定。
ipv4.dhcp	boolean	ipv4 アドレス	true	DHCP を使ってアドレスを割り当てるかどうか
ipv4.nat	boolean	ipv4 アドレス	false	NAT するかどうか（ipv4.address が未設定の場合デフォルト値は true でランダムな ipv4.address が生成されます）
ipv4.nat.address	string	ipv4 アドレス	-	ネットワークからの外向きトラフィックに使用されるソースアドレス (アップリンクに `ovn.ingress_mode=routed` が必要)
ipv6.address	string	標準モード	自動（作成時のみ）	ブリッジの IPv6 アドレス (CIDR 形式)。 IPv6 をオフにするには "none" 、新しいランダムな未使用のサブネットを生成するには "auto" を指定。
ipv6.nat.address	string	ipv6 アドレス	-	ネットワークからの外向きトラフィックに使用されるソースアドレス (アップリンクに `ovn.ingress_mode=routed` が必要)
ipv6.dhcp	boolean	ipv6 アドレス	true	DHCP 上に追加のネットワーク設定を提供するかどうか
ipv6.dhcp.stateful	boolean	ipv6 dhcp	false	DHCP を使ってアドレスを割り当てるかどうか
ipv6.nat	boolean	ipv6 アドレス	false	NAT するかどうか（ipv6.address が未設定の場合デフォルト値は true でランダムな ipv6.address が生成されます）
network	string	-	-	外部ネットワークへの外向きのアクセスに使うアップリンクのネットワーク
security.acls	string	-	-	このネットワークに接続する NIC に適用するネットワーク ACL のカンマ区切りリスト
secu-	string	secu-	reject	どの ACL ルールにもマッチしない ingress トラフィックに使うア

ネットワーク: 物理

物理ネットワークは OVN ネットワークを親インターフェースに接続する際に使用するプリセットの設定を提供します。

ネットワーク設定プロパティ:

キー	型	条件	デフォルト	説明
bgp.peers.NAME.address	string	bgp server	-	ovn ダウンストリームネットワークで使用するピアアドレス (IPv4 か IPv6)
bgp.peers.NAME.asn	integer	bgp server	-	ovn ダウンストリームネットワークで使用する AS 番号
bgp.peers.NAME.password	string	bgp server	- (パスワード無し)	ovn ダウンストリームネットワークで使用するピアのセッションパスワード（省略可能）
maas.subnet.ipv4	string	ipv4 アドレス	-	インスタンスを登録する MAAS IPv4 サブネット (NIC で network プロパティを使う場合に有効)
maas.subnet.ipv6	string	ipv6 アドレス	-	インスタンスを登録する MAAS IPv6 サブネット (NIC で network プロパティを使う場合に有効)
mtu	integer	-	-	作成するインターフェースの MTU
parent	string	-	-	sriov NIC を作成する親のインターフェース
vlan	integer	-	-	アタッチする先の VLAN ID
gvrp	boolean	-	false	GARP VLAN Registration Protocol を使って VLAN を登録する
ipv4.gateway	string	標準モード	-	ゲートウェイとネットワークの IPv4 アドレス（CIDR 表記）
ipv4.ovn.ranges	string	-	-	子供の OVN ネットワークルーターに使用する IPv4 アドレスの範囲（開始-終了 形式) のカンマ区切りリスト
ipv4.routes	string	ipv4 アドレス	-	子供の OVN ネットワークの ipv4.routes.external 設定で利用可能な追加の IPv4 CIDR サブネットのカンマ区切りリスト
ipv4.routes.anycast	boolean	ipv4 アドレス	false	複数のネットワーク／ NIC で同時にオーバーラップするルートが使われることを許可するかどうか
ipv6.gateway	string	標準モード	-	ゲートウェイとネットワークの IPv6 アドレス（CIDR 表記）
ipv6.ovn.ranges	string	-	-	子供の OVN ネットワークルーターに使用する IPv6 アドレスの範囲（開始-終了 形式) のカンマ区切りリスト
ipv6.routes	string	ipv6 アドレス	-	子供の OVN ネットワークの ipv6.routes.external 設定で利用可能な追加の IPv6 CIDR サブネットのカンマ区切りリスト
ipv6.routes.anycast	boolean	ipv6 アドレス	false	複数のネットワーク／ NIC で同時にオーバーラップするルートが使われることを許可するかどうか

BGP の統合

LXD は BGP サーバとして機能でき、アップストリームの BGP ルーターとセッションを確立し LXD が使用しているアドレスとサブネットを広告できます。

これにより LXD サーバやクラスタが内部／外部のアドレス空間を直接使い、正しいホストにルーティングされた特定のサブネットやアドレスをターゲットインスタンスにフォワードできます。

このためには `core.bgp_address`, `core.bgp_asn` と `core.bgp_routerid` が設定されている必要があります。これらが設定されると LXD は BGP セッションのリッスンを開始します。

ピアは `bridged` と `physical` で管理されたネットワーク上に定義できます。さらに `bridged` の場合は next-hop をオーバーライドするためにサーバ毎の設定キーの組が利用できます。それらが指定されない場合は next-hop はデフォルトとして BGP セッションに使用されるアドレスになります。

`physical` ネットワークの場合はアップリンクのネットワークが利用可能なサブネットのリストと BGP 設定を持つような `ovn` ネットワークに使用されます。親のネットワークが一旦設定されると、子の OVN ネットワークは BGP で広告された外部のサブネットとアドレスを受け取り next-hop は問題のネットワークの OVN ルーターアドレスに設定されます。

現在公開されるアドレスとネットワークは以下のとおりです。

- `nat` プロパティが `true` に設定されない場合はネットワークの `ipv4.address` か `ipv6.address`
- `nat` プロパティが設定される場合はネットワークの `ipv4.address` と `ipv6.address`
- `ipv4.routes.external` か `ipv6.routes.external` 経由で定義されるインスタンスの NIC ルート

現時点では、特定のピアに特定のルートやアドレスのみを公開する方法はありません。代わりにアップストリームのルーターでプリフィクスをフィルターすることを現状ではお勧めします。

3.2.4 ネットワーク ACL を設定するには

注釈: ネットワーク ACL は *OVN NIC* **タイプ**、*OVN* **ネットワーク** と **ブリッジネットワーク** (いくつか制限あり、**ブリッジの制限** 参照) で利用できます。

ネットワーク ACL (Access Control Lists; アクセス制御リスト) は同じネットワークに接続された異なるインスタンス間のネットワークアクセスや、他のネットワークとのアクセスを制御するトラフィクルールを定義します。

ネットワーク ACL は インスタンスの NIC やネットワークに直接適用できます。ネットワークに適用するときは、ネットワークに接続された全ての NIC に ACL が適用されます。

特定の ACL を (明示的にあるいはネットワークから暗黙的に) 適用したインスタンス NIC は論理的なグループを形成し、他のルールから送信元あるいは送信先として参照できます。より詳細な情報は *ACL* **グループ** を参照して

ください。

ACL を作成する

ACL を作成するには以下のコマンドを使用します。

```
lxc network acl create <ACL_name> [configuration_options...]
```

このコマンドはルール無しの ACL を作成します。次のステップとして ACL に **ルールを追加します**。

有効なネットワーク ACL の名前は以下のルールに従う必要があります。

- 名前は 1 文字から 63 文字の間である
- 名前は ASCII の文字、数字、ハイフンからのみなる
- 名前は数字やハイフンから始まらない
- 名前はハイフンで終わらない

ACL のプロパティ

ACL のプロパティには次のものがあります。

プロパティ	型	必須	説明
name	string	yes	プロジェクト内でユニークなネットワーク ACL の名前
description	string	no	ネットワーク ACL の説明
ingress	rule list	no	ingress のトラフィックルールのリスト
egress	rule list	no	egress のトラフィックルールのリスト
config	string set	no	キー・バリューペア形式での設定オプション (`user.*` カスタムキーのみサポート)

ルールの追加と削除

それぞれの ACL はルールの 2 つのリストを含みます。

- **イングレス** *(ingress)* ルールは NIC に向かう内向きのトラフィックに適用されます。
- **イーグレス** *(egress)* ルールは NIC から出ていく外向きのトラフィックに適用されます。

ACL にルールを追加するには、以下のコマンドを使います。 `<direction>` には `ingress` か `egress` のどちらかを指定します。

```
lxc network acl rule add <ACL_name> <direction> [properties...]
```

このコマンドは指定した方向 (direction) に対応するリストにルールを追加します。

(*ACL* **全体を編集する** 場合を除き) ルールを編集することはできませんが、以下のコマンドでルールを削除はできます。

```
lxc network acl rule remove <ACL_name> <direction> [properties...]
```

ユニークにルールを特定するのに必要な全てのプロパティを指定するか、またはマッチした全てのルールを削除するためコマンドに `--force` を追加する必要があります。

ルールの順番と優先度

ルールはリストとして提供されます。しかしリスト内のルールの順番は重要ではなくフィルタリングには影響しません。

LXD は以下のように `action` プロパティに基づいてルールの順番を自動的に決めます。

- `drop`
- `reject`
- `allow`
- 上記の全てにマッチしなかったトラフィックに対する自動のデフォルトアクション (デフォルトでは `reject`、**デフォルトアクションの設定** 参照)。

これは NIC に複数の ACL を適用する際、組み合わせたルールの順番を指定する必要がないことを意味します。ACL 内のあるルールがマッチすれば、そのルールが採用され、他のルールは考慮されません。

ルールのプロパティ

ACL ルールには次のプロパティがあります。

プロパティ	型	必須	説明
action	string	yes	マッチしたトラフィックに適用するアクション (`allow`, `reject` または `drop`)
state	string	yes	ルールの状態 (`enabled`, `disabled` または `logged`)、未設定の場合のデフォルト値は `enabled`
description	string	no	ルールの説明
source	string	no	CIDR か IP の範囲、送信元の ACL の名前、あるいは (ingress ルールに対しての) ソースサブジェクト名セレクターのカンマ区切りリスト、または any の場合は空を指定
destination	string	no	CIDR か IP の範囲、送信先の ACL の名前、あるいは (egress ルールに対しての) デスティネーションサブジェクト名セレクターのカンマ区切りリスト、または any の場合は空を指定
protocol	string	no	マッチ対象のプロトコル (`icmp4`, `icmp6`, `tcp`, `udp`)、または any の場合は空を指定
source_port	string	no	protocol が `udp` か `tcp` の場合はポートかポートの範囲 (開始-終了で両端含む) のカンマ区切りリスト、または any の場合は空を指定
destination_port	string	no	protocol が `udp` か `tcp` の場合はポートかポートの範囲 (開始-終了で両端含む) のカンマ区切りリスト、または any の場合は空を指定
icmp_type	string	no	protocol が `icmp4` か `icmp6` の場合は ICMP の Type 番号、または any の場合は空を指定
icmp_code	string	no	protocol が `icmp4` か `icmp6` の場合は ICMP の Code 番号、または any の場合は空を指定

ルール内でセレクタを使う

注釈: この機能は *OVN NIC* **タイプ** と *OVN* **ネットワーク** でのみサポートされます。

(ingress ルールの) `source` フィールドと (egress ルールの) `destination` フィールドは CIDR や IP の範囲の代わりにセレクタの使用をサポートします。

この機能を使えば、 IP のリストを管理したり追加のサブネットを作ることなしに、 ACL グループかネットワークセレクタを使ってインスタンスのグループに対するルールを定義できます。

ACL グループ

(明示的にあるいはネットワーク経由で暗黙的に) 特定の ACL を適用されたインスタンス NIC は論理的なポートグループを形成します。

そのような ACL グループは **サブジェクト名セレクタ** と呼ばれ、他の ACL グループ内で ACL 名を用いて参照できます。

例えば `foo` という名前の ACL がある場合、この ACL が適用されたインスタンス NIC のグループを `source=foo` で参照できます。

ネットワークセレクタ

ネットワークサブジェクトセレクタ を用いて、ネットワーク上の外向きと内向きのトラフィックにルールを定義できます。

`@internal` と `@external` という 2 つの特別なネットワークサブジェクトセレクタがあります。これらはそれぞれネットワークのローカルと外向きのトラフィックを示します。例:

```
source=@internal
```

ネットワークが **ネットワークピア** をサポートする場合、ピア接続間のトラフィックを`@<network_name>/<peer_name>` という形式のネットワークサブジェクトセレクタで参照できます。例:

```
source=@ovn1/mypeer
```

ネットワークサブジェクトセレクターを使用する際は、 ACL 適用先のネットワークは指定されたピア接続を持っていなければなりません。持っていない場合 ACL は適用できません。

トラフィックのログ

一般的には ACL はインスタンスとネットワーク間のネットワークトラフィックを制御するためのものです。しかし、特定のネットワークトラフィックをログ出力するためにルールを使うこともできます。これはモニタリングや、ルールを実際に有効にする前にテストするのに役立ちます。

ログのためにルールを追加するには `state=logged` プロパティ付きでルールを作成してください。ACL 内の全てのログのルールに対するログ出力は以下のコマンドで表示できます。

```
lxc network acl show-log <ACL_name>
```

ACL を編集する

ACL を編集するには以下のコマンドを使用します。

```
lxc network acl edit <ACL_name>
```

このコマンドは ACL を編集用に YAML 形式でオープンします。ACL 設定とルールの両方を編集できます。

ACL の適用

ACL の設定が終わったらネットワークかインスタンス NIC に適用する必要があります。

そのためにはネットワークか NIC の設定の `security.acls` リストに ACL を追加してください。ネットワークの場合は、以下のコマンドを使います。

```
lxc network set <network_name> security.acls="<ACL_name>"
```

インスタンス NIC の場合は、以下のコマンドを使います。

```
lxc config device set <instance_name> <device_name> security.acls="<ACL_name>"
```

デフォルトアクションの設定

1 つ以上の ACL が NIC に (明示的にあるいはネットワーク経由で暗黙的に) 適用されると、 NIC にデフォルトの reject ルールが追加されます。このルールは適用された ACL 内のどのルールにもマッチしない全てのトラフィックを拒否 (reject) します。

この挙動はネットワークと NIC レベルの `security.acls.default.ingress.action` と `security.acls.default.egress.action` 設定で変更できます。NIC レベルの設定はネットワークレベルの設定を上書きします。

例えば、ネットワークに接続された全てのインスタンスの内向きのトラフィックを許可 (`allow`) するには以下のコマンドを使用します。

```
lxc config device set <instance_name> <device_name> security.acls.default.ingress.
↪action=allow
```

インスタンス NIC に同じデフォルトアクションを設定するには以下のコマンドを使用します。

```
lxc config device set <instance_name> <device_name> security.acls.default.ingress.
↪action=allow
```

ブリッジの制限

ブリッジネットワークにネットワーク ACL を使用する場合は以下の制限に気を付けてください。

- OVN ACL とは違い、ブリッジ ACL はブリッジと LXD ホストの間の境界のみに適用されます。これは外部へと外部からのトラフィックにネットワークポリシーを適用するために使うことしかできないことを意味します。ブリッジ間のファイアウォール、つまり同じブリッジに接続されたインスタンス間のトラフィックに対するファイアウォールには使えません。
- *ACL* **グループとネットワークセレクタ** はサポートされません。
- `iptables` ファイアウォールドライバーを使う際は、IP レンジサブジェクト（例：`192.168.1.1-192.168.1.10`）は使用できません。
- ベースラインのネットワークサービスルールが（対応する INPUT/OUTPUT チェイン内の） ACL ルールの前に適用されます。これは一旦 ACL チェインに入ってしまうと INPUT/OUTPUT と FORWARD トラフィックを区別できないからです。このため ACL ルールはベースラインのサービスルールをブロックするのには使えません。

3.2.5 ネットワークフォワードを設定するには

注釈: ネットワークフォワードは *OVN* **ネットワーク** と **ブリッジネットワーク** で利用できます。

ネットワークフォワードは外部 IP アドレス（あるいは外部 IP アドレスの特定のポート）をフォワード設定が属するネットワーク内の内部 IP アドレス（あるいは内部 IP アドレスの特定のポート）にフォワードする機能です。

この機能は外部 IP アドレスが限定されていて 1 つの外部アドレスを複数のインスタンスで共有したい場合に有用です。この場合にネットワークフォワードを 2 つの異なる方法で利用できます。

- 外部アドレスからの全てのトラフィックを 1 つのインスタンスの内部アドレスにフォワードします。この方法はネットワークフォワードを単に再設定することで外部アドレスに向けられたトラフィックを別のインスタンスに簡単に移動できます。
- 外部アドレスの異なるポートからのトラフィックを異なるインスタンスにフォワードします (さらにこれらのインスタンスの異なるポートにフォワードもできます)。この方法は外部 IP アドレスを「共有」し、複数のインスタンスを一度に公開できます。

ネットワークフォワードを作成する

ネットワークフォワードを作成するには以下のコマンドを使用します。

```
lxc network forward create <network_name> <listen_address> [configuration_options...]
```

それぞれのフォワードはネットワークに割り当てられます。フォワードには単一の外部リッスンアドレスが必要です (使用しているネットワークに応じてどのアドレスがフォワードできるかについて詳細は **リッスンアドレスの要件** を参照してください)。

さらに `target_address=<IP_address>` 設定オプションを追加することでデフォルトのターゲットアドレスを追加することもできます。こうするとポート指定にマッチしないトラフィックは全てこのアドレスにフォワードします。このターゲットアドレスはフォワードが関連付けられるネットワークと同じサブネット内でなければならないことに注意してください。

フォワードのプロパティ

ネットワークフォワードのプロパティには以下のものがあります。

プロパティ	型	必須	説明
listen_address	string	yes	リッスンする IP アドレス
description	string	no	ネットワークフォワードの説明
config	string set	no	キー/バリューペア形式の設定オプション （`target_address` と `user.*` カスタムキーのみサポート）
ports	port list	no	**ポート指定** のリスト

リッスンアドレスの要件

有効なリッスンアドレスの要件はフォワードがどのネットワークタイプに割り当てられるかに応じて異なります。

ブリッジネットワーク

- 任意の衝突しないリッスンアドレスが使用できます。
- リッスンアドレスは他のネットワークで使用中のサブネットとオーバーラップはできません。

OVN ネットワーク

- 利用可能なリッスンアドレスはアップリンクネットワークの `ipv{n}.routes` 設定か (設定されている場合は) プロジェクトの `restricted.networks.subnets` 設定で定義されていなければなりません。

- リッスンアドレスは他のネットワークで使用中のサブネットとオーバーラップはできません。

ポートを設定する

リッスンアドレスの特定のポートからターゲットアドレスの特定のポートにトラフィックをフォワードするためにネットワークフォワードにポート指定を追加できます。このターゲットアドレスはデフォルトターゲットアドレスとは異なるものである必要があります。またフォワードを割り当てるネットワークと同じサブネットである必要があります。

ポート指定を追加するには以下のコマンドを使用します。

```
lxc network forward port add <network_name> <listen_address> <protocol> <listen_ports>
↪<target_address> [<target_ports>]
```

単一のポートか一組のポートを指定できます。異なるポートにトラフィックをフォワードしたい場合、 2 つの選択肢があります。

- 単一のターゲットポートを指定し、全てのリッスンポートからのトラフィックをこのターゲットポートにフォワードします。
- リッスンポートと同じ数の一組のターゲットポートを指定し、最初のリッスンポートを最初のターゲットポートに、 2 番目のリッスンポートを 2 番目のターゲットポートに、以下同様というようにフォワードします。

ポートのプロパティ

ネットワークフォワードポートのプロパティには以下のものがあります。

プロパティ	型	必須	説明
protocol	string	yes	ポートのプロトコル (`tcp` or `udp`)
listen_port	string	yes	リッスンするポート (例 `80,90-100`)
target_address	string	yes	フォワード先の IP アドレス
target_port	string	no	ターゲットのポート (例 `70,80-90` or `90`)、 空の場合は `listen_port` と同じ
description	string	no	ポートの説明

ネットワークフォワードを編集する

ネットワークフォワードを編集するには以下のコマンドを使用します。

```
lxc network forward edit <network_name> <listen_address>
```

このコマンドはネットワークフォワードを編集用に YAML 形式でオープンします。全般の設定とポート指定の両方を編集できます。

3.2.6 ネットワークピアの設定

ネットワークピアは 2 つの OVN ネットワーク間でルーティングの関係を作成できます。これにより 2 つのネットワーク間での通信がアップリンクのネットワーク経由ではなく OVN サブシステム内で完結できます。

ピアリングが双方向であることを確実にするため、ピアリング内の両方のネットワークがセットアップの行程を完了する必要があります。

例。

```
lxc network peer create <local_network> foo <target_project/target_network> --
↪project=local_network
lxc network peer create <target_network> foo <local_project/local_network> --
↪project=target_project
```

ピアのセットアップの行程でプロジェクトかネットワーク名の指定が正しくない場合、対応するプロジェクトやネットワークが存在しないというエラーを上記のコマンドは表示しません。これは他のプロジェクトの（訳注：悪意の）ユーザーがプロジェクトやネットワークが存在するかを確認できないようにするためです。

プロパティ

ネットワークピアの設定は以下の通りです。

プロパティ	型	必須	説明
name	string	yes	ローカルネットワーク上のネットワークピアの名前
description	string	no	ネットワークピアの説明
config	string set	no	設定のキーバリューペアー (`user.*` のカスタムキーのみサポート)
ports	port list	no	ネットワークフォワードのポートリスト
target_project	string	yes	対象のネットワークがどのプロジェクト内に存在するか (作成時に必須)
target_network	string	yes	どのネットワークとピアを作成するか (作成時に必須)
status	string	--	作成中か作成完了 (対象のネットワークと相互にピアリングした状態) かを示すステータス

3.2.7 ネットワークゾーンを設定するには

注釈: ネットワークゾーンは *OVN* **ネットワーク** と **ブリッジネットワーク** で利用できます。

ネットワークゾーンは LXD のネットワークの DNS レコードを保持するのに使用します。

ネットワークゾーンを使うと全てのインスタンスの有効な正引きと逆引きのレコードを自動的に維持できます。多くのネットワークにまたがる複数のインスタンスからなる LXD クラスタを運用する際に有用です。

各インスタンスに DNS レコードを持つとインスタンス上のネットワークサービスにアクセスするのがより簡単になります。また例えば外部への SMTP サービスをホストする際にも重要です。インスタンスに正しい正引きと逆引きの DNS エントリがないと、送信されたメールが潜在的なスパムと判定されてしまうかもしれません。

各ネットワークは下記の最大 3 つに関連します。

- 正引き DNS レコード
- IPv4 逆引き DNS レコード
- IPv6 逆引き DNS レコード

LXD は全てのインスタンス、ネットワークゲートウェイ、ダウンストリーム (下流) のネットワークポートの全てに対して正引きと逆引きのレコードを自動で管理し、オペレータのプロダクションの DNS サーバへのゾーン転送のためのこれらのゾーンを提供します。

生成されるレコード

例えば、あなたのネットワークで lxd.example.net の正引き DNS レコードのゾーンを設定した場合、以下の DNS 名を解決するレコードを生成します。

- ネットワーク内の全てのインスタンスに対して: <instance_name>.lxd.example.net
- ネットワークゲートウェイに対して: <network_name>.gw.lxd.example.net
- ダウンストリームネットワークポートに対して (ダウンストリーム OVN ネットワークを持つアップリンクのネットワーク上に設定されれうネットワークゾーンに対して): <project_name>-<downstream_network_name>.uplink.lxd.example.net

ゾーン設定に対して生成されたレコードは dig コマンドで確認できます。例えば、 dig @<DNS_server_IP> -p 1053 axfr lxd.example.net と実行すると以下のように出力されます。

```
lxd.example.net.              3600  IN        SOA        lxd.example.net. hostmaster.lxd.
↪example.net. 1648118965 120 60 86400 30
default-my-ovn.uplink.lxd.example.net. 300 IN A 192.0.2.100
my-instance.lxd.example.net.  300   IN        A        192.0.2.76
my-uplink.gw.lxd.example.net. 300   IN        A        192.0.2.1
foo.lxd.example.net.          300        IN        A        8.8.8.8
lxd.example.net.              3600        IN        SOA        lxd.example.net.␣
↪hostmaster.lxd.example.net. 1648118965 120 60 86400 30
```

192.0.2.0/24 を使用するネットワークに 2.0.192.in-addr.arpa の IPv4 逆引き DNS レコードのゾーンを設定すると、例えば 192.0.2.100 に対する逆引き DNS レコードを生成します。

組み込みの DNS サーバを有効にする

ネットワークゾーンを使用するには、組み込みの DNS サーバを有効にする必要があります。

そのためには、 LXD サーバのローカルアドレスに core.dns_address 設定オプション (**サーバ設定** 参照) を設定してください。

これは DNS サーバがリッスンするアドレスです。LXD クラスタの場合、アドレスは各クラスタメンバーによって異なるかもしれないことに注意してください。

注釈: 組み込みの DNS サーバは AXFR 経由でのゾーン転送のみをサポートしており、DNS レコードへの直接の問い合わせはできません。つまりこの機能は外部の DNS サーバ (bind9, nsd, ...) の使用を前提としています。外部の DNS サーバが LXD からの全体のゾーンを転送し、有効期限を過ぎたら更新し、DNS 問い合わせに対する管理権限を持つ応答 (authoritative answers) を提供します。

ゾーン転送の認証はゾーン毎に設定され、各ゾーンでピアごとに IP アドレスと TSIG キーを設定して、TSIG キー

ベースの認証を行います。

ネットワークゾーンの作成と設定

ネットワークゾーンの作成には以下のコマンドを使用します。

```
lxc network zone create <network_zone> [configuration_options...]
```

以下の例は正引き DNS レコードのゾーン、IPv4 逆引き DNS レコードのゾーン、IPv6 逆引き DNS レコードのゾーンを作成する方法を示しています。

```
lxc network zone create lxd.example.net
lxc network zone create 2.0.192.in-addr.arpa
lxc network zone create 1.0.0.0.1.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
```

注釈: ゾーン名は複数のプロジェクトをまたいでグローバルにユニークでなければなりません。そのため、別のプロジェクト内の既存のゾーンのせいでゾーンの作成がエラーになることがありえます。

ネットワークを作成するときに設定オプションを指定できますし、後から以下のコマンドで設定もできます。

```
lxc network zone set <network_zone> <key>=<value>
```

YAML 形式でネットワークゾーンを編集するには以下のコマンドを使用します。

```
lxc network zone edit <network_zone>
```

設定オプション

ネットワークゾーンで利用可能な設定オプションは下記のとおりです。

キー	型	必須	デフォルト値	説明
peers.NAME.address	string	no	-	DNS サーバの IP アドレス
peers.NAME.key	string	no	-	サーバの TSIG キー
dns.nameservers	string set	no	-	(NS レコード用の) DNS サーバの FQDN のカンマ区切りリスト
network.nat	bool	no	true	NAT されたサブネットのレコードを生成するかどうか
user.*	*	no	-	ユーザー提供の自由形式のキー・バリューペア

ネットワークにネットワークゾーンを追加する

ネットワークにゾーンを追加するにはネットワーク設定内に対応する設定オプションを設定します。

- 正引き DNS レコードには: dns.zone.forward
- IPv4 逆引き DNS レコードには: dns.zone.reverse.ipv4
- IPv6 逆引き DNS レコードには: dns.zone.reverse.ipv6

例えば

```
lxc network set <network_name> dns.zone.forward="lxd.example.net"
```

ゾーンはプロジェクトに属し、プロジェクトの networks 機能に紐づきます。プロジェクトの restricted.networks.zones 設定キーを使ってプロジェクトを指定のドメインとサブドメインに制限できます。

カスタムレコードを追加する

ネットワークゾーンは、全てのインスタンス、ネットワークゲートウェイ、ダウンストリームネットワークポートに対して正引きと逆引きレコードを自動的に生成します。

そのためには lxc network zone record コマンドを使用します。

レコードを作成する

レコードを作成するには以下のコマンドを使用します。

```
lxc network zone record create <network_zone> <record_name>
```

このコマンドはエントリ無しの空のレコードを作成しネットワークゾーンに追加します。

レコードのプロパティ

レコードは以下のプロパティを持ちます。

プロパティ	型	必須	説明
name	string	yes	レコードのユニークな名前
description	string	no	レコードの説明
entries	entry list	no	DNS エントリのリスト
config	string set	no	キー／バリュー形式の設定オプション (user.* カスタムキーのみサポート)

エントリを追加または削除する

レコードにエントリを追加するには以下のコマンドを使います。

```
lxc network zone record entry add <network_zone> <record_name> <type> <value> [--ttl
↪<TTL>]
```

このコマンドはレコードに指定した型と値を持つ DNS エントリを追加します。

例えば、デュアルスタックのウェブサーバを作成するには以下のような 2 つのエントリを持つレコードを追加します。

```
lxc network zone record entry add <network_zone> <record_name> A 1.2.3.4
lxc network zone record entry add <network_zone> <record_name> AAAA 1234::1234
```

エントリにカスタムの time-to-live (秒で指定) を設定するには `--ttl` フラグが使えます。指定しない場合、デフォルトの 300 秒になります。

(`lxc network zone record edit` でレコード全体を編集するのを除いて) エントリを編集は出来ませんが、以下のコマンドでエントリを削除できます。

```
lxc network zone record entry remove <network_zone> <record_name> <type> <value>
```

3.2.8 プリシード YAML を使った非対話型設定

`lxd init` コマンドは `--preseed` コマンドラインフラグをサポートしており、LXD デーモンの設定、ストレージプール、ネットワークデバイスとプロファイルを非対話式にを完全に構成することができます。

例えば、 LXD を新規インストールした状態で、以下のコマンドを実行

```
    cat <<EOF | lxd init --preseed
config:
  core.https_address: 192.168.1.1:9999
  images.auto_update_interval: 15
networks:
- name: lxdbr0
  type: bridge
  config:
    ipv4.address: auto
    ipv6.address: none
EOF
```

すると 192.168.1.1 のアドレスのポート 9999 で HTTPS 接続をリッスンし、15 時間ごとにイメージを自動的にアップデートし、 `lxdbr0` という名前のネットワークデバイスを作成し、 IPV4 の アドレスを自動的に割り当てるように LXD デーモンを構成します。

新規インストールした LXD を設定する

新規インストールした LXD のインスタンスを設定する場合、 preseed コマンドは必ず成功し、希望の設定を適用できます (与えられた YAML が正しいキーと値を含んでいる限り)。というのは、希望の状態と衝突する既存の状態が存在しないからです。

既存の LXD を再構成する

既存の LXD インスタンスを preseed コマンドを使って再構成する場合、指定された YAML 設定は既存の設定を完全に上書きすることを意味します (新規インストールの LXD の場合と同様に、指定された設定が存在しない場合はそれらは単に作成されます)。

既存の設定を上書きする場合は、その設定の新しい希望の状態の完全な設定を指定する必要があります (つまり *RESTful API* での `PUT` と同じ考え方です)。

ロールバック

新しく希望する設定の一部が既存の設定と衝突する場合 (例えばストレージプールをドライバーを `dir` から `zfs` に変更しようとするなど)、preseed コマンドは失敗し、それまでに適用したあらゆる変更を自動的にロールバックしようとベストを尽くします。

例えば新しい設定で作られた設定を削除したり、上書きされた設定を元の状態に戻したりするでしょう。

設定を上書きするのに失敗した場合のモードは *RESTful API* の `PUT` リクエストの場合と同様です。

ただし、まれにではありますが (典型的にはバックエンドのバグか制限により)、ロールバック自体も失敗する可能性があることにご注意ください。ですので LXD デーモンをプリシードで再構成しようとするときは注意が必要です。

デフォルト・プロファイル

対話的な初期化モードをは異なり、指定した YAML のペイロードで明示的に変更を指示しない限り、 `lxd init --preseed` はデフォルト・プロファイルを特定の状態に変更することはしません。

例えば、典型的にはデフォルトプロファイルにルート・ディスク・デバイスとネットワーク・インターフェースをアタッチしたいでしょう。以下の例を参照してください。

設定の形式

さまざまな設定のサポートされるキーと値は *RESTful API* にドキュメントされているものと同じです。ただし、YAML が読みやすいように変換されたものになっています (YAML は JSON のスーパーセットなので JSON を使うこともできます)。

以下に設定可能な取っ手のほとんどを含むプリシードのペイロードの例を示します。あなた自身のペイロードのテンプレートとして使い、ここに必要な設定を追加、変更、削除して使うことができます。

```
# デーモンの設定
config:
  core.https_address: 192.168.1.1:9999
  core.trust_password: sekret
  images.auto_update_interval: 6

# ストレージプール
storage_pools:
- name: data
  driver: zfs
  config:
    source: my-zfs-pool/my-zfs-dataset

# ネットワークデバイス
networks:
- name: lxd-my-bridge
  type: bridge
  config:
    ipv4.address: auto
    ipv6.address: none

# プロファイル
profiles:
- name: default
  devices:
    root:
      path: /
      pool: data
      type: disk
- name: test-profile
  description: "Test profile"
  config:
```

(次のページに続く)

(前のページからの続き)

```
    limits.memory: 2GB
  devices:
    test0:
      name: test0
      nictype: bridged
      parent: lxd-my-bridge
      type: nic
```

3.2.9 プロファイル

プロファイルはインスタンスが保持できる（キー・バリューやデバイスなどの）あらゆる設定を保持することができ、プロファイルをいくつでもインスタンスに適用することができます。

プロファイルは指定された順番に適用され、その結果最後に指定したプロファイルが特定のキーを上書きします。

どのような場合でも、インスタンス固有の設定はプロファイル由来のものを上書きします。

デフォルトのプロファイル

まだ存在していない場合は、LXD は `default` プロファイルを作成します。

`default` プロファイルはリネームや削除はできません。`default` プロファイルは異なるプロファイルリストを指定せずに作られたあらゆる新規のインスタンスに設定されます。

設定

プロファイルはコンテナや仮想マシンに固有なものではないため、どちらのインスタンスタイプでも有効な設定やデバイスを含めることができます。

これはこれらの設定やデバイスをインスタンスに直接適用するときの挙動とは異なります。その場合はインスタンスタイプが考慮され、許可されないキーはエラーになります。

有効な設定のオプションについては **インスタンス設定** を参照してください。

3.2.10 プロジェクト設定

プロジェクトに何を含めるかは `features` 設定キーによって決められます。機能が無効の場合はプロジェクトは `default` プロジェクトから継承します。

デフォルトでは全ての新規プロジェクトは全体のフィーチャーセットを取得し、アップグレード時には既存のプロジェクトは新規のフィーチャーが有効にはなりません。

key/value 設定は現在サポートされている以下のネームスペースによって名前空間が分けられています。

- `features` プロジェクトのフィーチャーセットのどの部分が使用中か
- `limits` プロジェクトに属するコンテナと VM に適用されるリソース制限
- `user` ユーザーメタデータに対する自由形式の key/value

キー	型	条件	デフォルト値	説明
backups.compression_algorithm	string	-	-	プロジェクト内のバックアップに使う圧縮アルゴリズム（
features.images	boolean	-	true	プロジェクト用のイメージとイメージエイリアスのセット
features.networks	boolean	-	false	プロジェクトごとに個別のネットワークのセットを使うか
features.profiles	boolean	-	true	プロジェクト用のプロファイルを分離する
features.storage.volumes	boolean	-	true	プロジェクトごとに個別のストレージボリュームのセット
images.auto_update_cached	boolean	-	-	LXD がキャッシュするイメージを自動更新するかどうか
images.auto_update_interval	integer	-	-	キャッシュしたイメージの更新を確認する間隔（単位は時
images.compression_algorithm	string	-	-	プロジェクト内のイメージに使う圧縮アルゴリズム（bzip
images.default_architecture	string	-	-	アーキテクチャーが混在するクラスタ内で使用するデフォ
images.remote_cache_expiry	integer	-	-	プロジェクト内の使用されないリモートイメージのキャッ
limits.containers	integer	-	-	プロジェクト内に作成可能なコンテナの最大数
limits.cpu	integer	-	-	プロジェクトのインスタンスに設定する個々の "limits.cpu
limits.disk	string	-	-	プロジェクトの全てのインスタンスボリューム、カスタム
limits.instances	integer	-	-	プロジェクト内に作成できるインスタンスの合計数の最大
limits.memory	string	-	-	プロジェクトのインスタンスに設定する個々の "limits.me
limits.networks	integer	-	-	このプロジェクトが持てるネットワークの最大数
limits.processes	integer	-	-	プロジェクトのインスタンスに設定する個々の "limits.pro
limits.virtual-machines	integer	-	-	プロジェクト内に作成可能な VM の最大数
restricted	boolean	-	false	セキュリティセンシティブな機能へのアクセスをブロック
restricted.backups	string	-	block	インスタンスやボリュームのバックアップの作成を禁止す
restricted.cluster.groups	string	-	-	指定したグループ以外のクラスタグループにターゲットす
restricted.cluster.target	string	-	block	インスタンスを作成・移動する際にクラスタメンバーを直
restricted.containers.lowlevel	string	-	block	block と設定すると raw.lxc, raw.idmap, volatile などの低レ
restricted.containers.nesting	string	-	block	block と設定すると security.nesting=true と設定するのを防

キー	型	条件	デフォルト値	説明
restricted.containers.privilege	string	-	unpriviliged	unpriviliged と設定すると security.privileged=true と設定
restricted.containers.interception	string	-	block	システムコールのインターセプションオプションの使用を
restricted.devices.disk	string	-	managed	block と設定すると root 以外のディスクデバイスを使用で
restricted.devices.disk.paths	string	-	-	`restricted.devices.disk` が `allow` に設定された場合
restricted.devices.gpu	string	-	block	block と設定すると gpu タイプのデバイスの使用を防ぐ
restricted.devices.infiniband	string	-	block	block と設定すると infiniband タイプのデバイスの使用を
restricted.devices.nic	string	-	managed	block と設定すると全てのネットワークデバイスの使用を
restricted.devices.pci	string	-	block	"pci" タイプのデバイスの使用を防ぐ
restricted.devices.proxy	string	-	block	"proxy" タイプのデバイスの使用を防ぐ
restricted.devices.unix-block	string	-	block	block と設定すると unix-block タイプのデバイスの使用を
restricted.devices.unix-char	string	-	block	block と設定すると unix-char タイプのデバイスの使用を
restricted.devices.unix-hotplug	string	-	block	block と設定すると unix-hotplug タイプのデバイスの使用
restricted.devices.usb	string	-	block	block と設定すると usb タイプのデバイスの使用を防ぐ
restricted.idmap.uid	string	-	-	インスタンスの `raw.idmap` 設定で使用可能なホストの U
restricted.idmap.gid	string	-	-	インスタンスの `raw.idmap` 設定で使用可能なホストの G
restricted.networks.subnets	string	-	block	このプロジェクトで使用するために割り当てられるアップ
restricted.networks.uplinks	string	-	block	このプロジェクト内のネットワークでアップリンクとして
restricted.networks.zones	string	-	block	このプロジェクト内の使用可能なネットワークゾーン（ま
restricted.snapshots	string	-	block	インスタンスやボリュームのスナップショット作成を禁止
restricted.virtual-machines.lowlevel	string	-	block	block と設定すると raw.qemu, volatile などの低レベルの仮

これらのキーは lxc ツールを使って以下のように設定できます。

```
lxc project set <project> <key> <value>
```

プロジェクトの制限

注意: `limits.*` 設定キーの 1 つを設定する際はプロジェクト内の **全ての** インスタンスに直接あるいはプロファイル経由で同じ設定キーを設定 **する必要があります**。

それに加えて

- `limits.cpu` 設定キーを使うにはさらに CPU ピンニングが使用されて **いない** 必要があります。
- `limits.memory` 設定キーはパーセント **ではなく** 絶対値で設定する必要があります。

プロジェクトに設定された `limits.*` 設定キーは直接あるいはプロファイル経由でプロジェクト内のインスタンスに設定した個々の `limits.*` 設定キーの値の **合計値** に対しての hard な上限として振る舞います。

例えば、プロジェクトの `limits.memory` 設定キーを `50GB` に設定すると、プロジェクト内のインスタンスに設定

された全ての limits.memory 設定キーの個々の値の合計が 50GB 以下に維持されることを意味します。インスタンスの作成あるいは変更時に limits.memory の値を全体の合計が 50GB を超えるように設定しようとするとエラーになります。

同様にプロジェクトの limits.cpu 設定キーを 100 に設定すると、個々の limits.cpu の値の **合計** が 100 以下に維持されることを意味します。

プロジェクトのセキュリティ規制

restricted 設定キーが true に設定されると、プロジェクトのインスタンスはコンテナネスティングや生の LXC 設定といったセキュリティセンシティブな機能にアクセスできなくなります。

restricted 設定キーがブロックする機能の正確な組み合わせは LXD の今後のリリースに伴って、より多くの機能がセキュリティセンシテイブであると判断されて増えていく可能性があります。

さまざまな restricted.* サブキーを使うことで通常なら restricted でブロックされるはずの個々の機能を選んで許可し、プロジェクトのインスタンスで使えるようにできます。

例えば

```
lxc project set <project> restricted=true
lxc project set <project> restricted.containers.nesting=allow
```

はコンテナネスティング **以外の** 全てのセキュリティセンシティブな機能をブロックします。

それぞれのセキュリティセンシティブな機能は対応する restricted.* プロジェクト設定サブキーを持ち、その機能を許可しプロジェクトで使えるようにするにはデフォルト値から変更する必要があります。

個々の restricted.* 設定キーの値の変更が有効になるのはトップレベルの restricted キーが true に設定されているときのみであることに注意してください。restricted が false に設定されている場合、restricted.* サブキーを変更しても実質的には変更していないのと同じです。

ほとんどの restricted.* 設定キーは block（デフォルト値）か allow のいずれかの値を設定可能なバイナリースイッチです。しかし一部の restricted.* 設定キーはより細かい制御のために他の値をサポートします。

全ての restricted.* キーを allow に設定すると restricted 自体を false に設定するのと実質同じことになります。

3.2.11 サーバ設定

key/value 設定は現在サポートされている以下のネームスペースによって名前空間が分けられています。

- `backups` バックアップ設定
- `candid` Candid を使った外部のユーザー認証 (External user authentication through Candid)
- `cluster` クラスタ設定
- `core` コア・デーモン設定
- `images` イメージ設定
- `maas` MAAS 統合
- `rbac` 外部の Candid と Canonical の RBAC を使ったロールベースのアクセス制御 (Role Based Access Control)

キー	型	スコープ	デフォルト値	説明
backups.compression_algorithm	string	global	gzip	新規のイメージに用いる圧縮アル
candid.api.key	string	global	-	Candid サーバの公開鍵（HTTP の
candid.api.url	string	global	-	Candid を使用する外部認証エント
candid.domains	string	global	-	許可される Candid ドメインのカン
candid.expiry	integer	global	3600	Canded macaroon の有効期間 (秒で
cluster.https_address	string	local	-	クラスタのトラフィックに使用す
cluster.images_minimal_replica	integer	global	3	特定のイメージのコピーを持つべ
cluster.max_standby	integer	global	2	データベースのスタンバイの役割
cluster.max_voters	integer	global	3	データベースの投票者の役割を割
cluster.offline_threshold	integer	global	20	無反応なノードをオフラインとみ
core.bgp_address	string	local	-	BGP サーバをバインドさせるアド
core.bgp_asn	string	global	-	ローカルサーバに使用する BGP の
core.bgp_routerid	string	local		この BGP サーバのユニークな ID
core.debug_address	string	local	-	pprof デバッグサーバがバインドす
core.dns_address	string	local	-	権威 DNS サーバをバインドするア
core.https_address	string	local	-	リモート API がバインドするアド
core.https_allowed_credentials	boolean	global	-	Access-Control-Allow-Credentials
core.https_allowed_headers	string	global	-	Access-Control-Allow-Headers HT
core.https_allowed_methods	string	global	-	Access-Control-Allow-Methods HT
core.https_allowed_origin	string	global	-	Access-Control-Allow-Origin HTT
core.https_trusted_proxy	string	global	-	プロキシの connection ヘッダーで
core.metrics_address	string	global	-	メトリクスサーバをバインドさせ

表 4 – 前のページからの続き

キー	型	スコープ	デフォルト値	説明
core.metrics_authentication	boolean	global	true	メトリクスエンドポイントの認証
core.proxy_https	string	global	-	HTTPS プロキシを使用する場合
core.proxy_http	string	global	-	HTTP プロキシを使用する場合は
core.proxy_ignore_hosts	string	global	-	プロキシが不要なホスト (NO_PR
core.shutdown_timeout	integer	global	5	LXD サーバがシャットダウンを完
core.trust_ca_certificates	boolean	global	-	CA に署名されたクライアント証明
core.trust_password	string	global	-	信頼を確立するためにクライアン
images.auto_update_cached	boolean	global	true	LXD がキャッシュしているイメー
images.auto_update_interval	integer	global	6	キャッシュされているイメージが
images.compression_algorithm	string	global	gzip	新しいイメージに使用する圧縮ア
images.default_architecture	string	-	-	アーキテクチャーが混在するクラ
images.remote_cache_expiry	integer	global	10	キャッシュされたが未使用のイメ
maas.api.key	string	global	-	MAAS を管理するための API キー
maas.api.url	string	global	-	MAAS サーバの URL
maas.machine	string	local	hostname	この LXD ホストの MAAS での名
network.ovn.integration_bridge	string	global	br-int	OVN ネットワークに使用する OV
network.ovn.northbound_connection	string	global	unix:/var/run/ovn/ovnnb_db.sock	OVN northbound データベース接続
rbac.agent.public_key	string	global	-	RBAC 登録中に提供される Candi
rbac.agent.private_key	string	global	-	RBAC 登録中に提供される Candi
rbac.agent.url	string	global	-	RBAC 登録中に提供される Candi
rbac.agent.username	string	global	-	RBAC 登録中に提供される Candi
rbac.api.expiry	integer	global	-	RBAC の macaroon の有効期限 (秒
rbac.api.key	string	global	-	RBAC サーバの公開鍵 (HTTP のみ
rbac.api.url	string	global	-	外部の RBAC サーバの URL
storage.backups_volume	string	local	-	バックアップの tarball を保管する
storage.images_volume	string	local	-	イメージの tarball を保管するのに

これらのキーは lxc コマンドで次のように設定します。

```
lxc config set <key> <value>
```

クラスタの一部として動作するときは、上記の表でスコープが `global` のキーは全てのクラスタメンバーに即座に反映されます。スコープが `local` のキーはコマンドラインツールの `\-\-target` オプションを使ってメンバーごとに設定する必要があります。

LXD をネットワーク上に公開する

デフォルトでは LXD は UNIX ソケット経由でローカルのユーザーのみが使用できます。

LXD をネットワーク上に公開するには `core.https_address` を設定する必要があります。すると全てのリモートクライアントが LXD に接続でき、公開利用可能とマークされた全てのイメージにアクセスできます。

信頼されたクライアントはサーバのトラストストアーに手動で追加できます。`lxc config trust add` を実行するか `core.trust_password` キーを設定し、設定したパスワードを接続時に提供することでクライアントがトラストストアーに追加されます。

認証についての詳細は **セキュリティ** を参照してください。

外部認証

ネットワーク経由で LXD にアクセスする場合は Candid による外部認証を使うように設定できます。

上記の `candid.*` 設定キーをデプロイ済みの Candid に対応する値に設定することでユーザーはウェブブラウザーで認証し LXD に信頼されることができます。

Candid サーバの手前に Canonical RBAC サーバがある場合、 `candid.*` の代わりにそれらのスーパーセットである `rbac.*` 設定キーを設定でき、これにより LXD を RBAC サービスと統合できます。

RBAC と統合されると、個々のユーザーとグループはプロジェクト単位にさまざまなアクセスレベルで許可が与えられます。これらは全て RBAC サービスにより外部で制御されます。

認証についての詳細は **セキュリティ** を参照してください。

3.2.12 ストレージの設定

- **ディレクトリ** *(dir)*
- *ceph*
- *cephfs*
- *btrfs*
- *lvm*
- *zfs*

ストレージプールの設定は lxc ツールを使って次のように設定できます:

```
lxc storage set [<remote>:]<pool> <key> <value>
```

ストレージボリュームの設定は lxc ツールを使って次のように設定できます:

```
lxc storage volume set [<remote>:]<pool> <volume> <key> <value>
```

ストレージプールのデフォルトボリューム設定を設定するには、volume 接頭辞つきのストレージプール設定を設定します（例: volume.<VOLUME_CONFIGURATION>=<VALUE>）。例えば、デフォルトのボリュームサイズを lxc ツールで設定するには以下のようにします。

```
lxc storage set [<remote>:]<pool> volume.size <value>
```

ストレージボリュームのコンテンツタイプ

ストレージボリュームは filesystem か block のいずれかのタイプが指定可能です。

コンテナとコンテナイメージは常に filesystem を使います。仮想マシンと仮想マシンイメージは常に block を使います。

カスタムストレージボリュームはどちらのタイプも利用可能でデフォルトは filesystem です。タイプが block のカスタムストレージボリュームは仮想マシンにのみアタッチできます。

ブロックカスタムストレージボリュームは以下のようにして作成できます。

```
lxc storage volume create [<remote>]:<pool> <name> --type=block
```

LXD のデータをどこに保管するか

使用しているストレージバックエンドによって LXD はファイルシステムをホストと共有するかあるいはデータを分離しておくことができます。

ホストと共有する

これは通常最もスペース効率良く LXD を動かす方法で、管理もおそらく一番容易でしょう。以下の方法で実現できます。

- 任意のファイルシステム上の dir バックエンド
- btrfs バックエンドでホストが btrfs で LXD に専用のサブボリュームを与えている場合
- zfs バックエンドでホストが zfs で zpool 上で専用のデータセットを LXD に与えている場合

専用のディスク／パーティション

このモードでは LXD のストレージはホストから完全に独立しています。これはメインのディスク上で空のパーティションを LXD に使用させるか、ディスク全体を専用で使用させるかで実現できます。

これは `dir`, `ceph`, `cephfs` 以外の全てのストレージドライバーでサポートされます。

ループディスク

上記のどちらの選択肢も利用できない場合、 LXD はメインのドライブ上にループファイルを作成し、選択したストレージドライバーにそれを使わせることができます。

これはディスク／パーティションを使う方法と似ていますが、メインのドライブ上の大きなファイルを代わりに使います。この方法は全ての書き込みがストレージドライバーとさらにメインドライブのファイルシステムの両方で処理される必要があるため、パフォーマンス上のペナルティーを受けます。またループファイルは通常は縮小できません。設定した上限までサイズが拡大しますが、インスタンスやイメージを削除してもファイルは縮小しません。

ストレージバックエンドとサポートされる機能

機能比較

LXD では、イメージ、インスタンス、カスタムボリューム用のストレージとして ZFS、btrfs、LVM、単なるディレクトリが使えます。可能であれば、各システムの高度な機能を使って、LXD は操作を最適化しようとします。

機能	ディレクトリ	Btrfs	LVM	ZFS	CEPH
最適化されたイメージストレージ	no	yes	yes	yes	yes
最適化されたインスタンスの作成	no	yes	yes	yes	yes
最適化されたスナップショットの作成	no	yes	yes	yes	yes
最適化されたイメージの転送	no	yes	no	yes	yes
最適化されたインスタンスの転送	no	yes	no	yes	yes
コピーオンライト	no	yes	yes	yes	yes
ブロックデバイスベース	no	no	yes	no	yes
インスタントクローン	no	yes	yes	yes	yes
コンテナ内でストレージドライバの使用	yes	yes	no	no	no
古い（最新ではない）スナップショットからのリストア	yes	yes	yes	no	yes
ストレージクオータ	yes(*)	yes	yes	yes	yes

おすすめのセットアップ

LXD から使う場合のベストなオプションは ZFS と btrfs を使うことです。このふたつは同様の機能を持ちますが、お使いのプラットフォームで使えるのであれば、ZFS のほうがより信頼性が上です。

可能であれば、LXD のストレージプールにディスクかパーティション全体を与えるのが良いでしょう。LXD で loop ベースのストレージを作れますが、プロダクション環境ではおすすめしません。

同様に、ディレクトリバックエンドも最後の手段として考えるべきでしょう。LXD の主な機能すべてが使えますが、インスタントコピーやスナップショットが使えないので、毎回インスタンスのストレージ全体をコピーする必要があり、恐ろしく遅くて役に立たないでしょう。

セキュリティの考慮

現在、 Linux Kernel はブロックベースのファイルシステム（例: `ext4`）が別のオプションでマウント済みの場合マウントオプションは適用せずに黙って無視します。これは専用ディスクデバイスが異なるストレージプール間で共有されている時、2 つめのマウントは期待しているマウントオプションが設定されないかもしれないことを意味します。これは例えば 1 つめのストレージプールが `acl` サポートを提供する想定で、2 つめのストレージプールが `acl` サポートを提供しない想定であるようなときにセキュリティ上の問題になります。この理由により、現状はストレージプールごとに専用のディスクデバイスを持つか、同じ専用ディスクを共有する全てのストレージプールで同じマウントオプションを使うことを推奨します。

最適化されたイメージストレージ

ディレクトリ以外のすべてのバックエンドには、ある種の最適化されたイメージ格納フォーマットがあります。これは、一からイメージの tarball を展開するのではなく、あらかじめ作られたイメージボリュームから単にクローンして、瞬間的にインスタンスを作るのに使われます。

そのイメージで使えないストレージプールの上にそのようなボリュームを準備することは無駄なので、ボリュームはオンデマンドで作成されます。したがって、最初のインスタンスはあとで作るインスタンスよりは作成に時間がかかります。

最適化されたインスタンスの転送

ZFS、btrfs、Ceph RBD は内部で send/receive メカニズムを持っており、最適化されたボリュームの転送ができます。LXD はこのような機能を使い、サーバ間でインスタンスやスナップショットを転送します。

ストレージドライバーがこのような機能をサポートしていない場合や、転送元と転送先のサーバのストレージバックエンドが違う場合で、このような機能が使えない場合は、LXD は代わりに rsync を使った転送にフォールバックし、個々のファイルを転送します。

rsync を使う必要がある場合、LXD ではストレージプールのプロパティである `rsync.bwlimit` を 0 以外の値に設定することで、ソケット I/O の流量の上限を設定できます。

デフォルトのストレージプール

LXD にはデフォルトののストレージプールの概念はありません。代わりに、インスタンスのルートに使用するプールは、LXD 内で別の「ディスク」デバイスとして扱われます。

デバイスエントリーは次のようになります。

```
root:
  type: disk
  path: /
  pool: default
```

この設定はインスタンスに直接指定できますし（"-s"オプションを "lxc launch" と "lxc init" に与えて）、LXD プロファイル経由でも設定できます。

後者のオプションは、デフォルトの LXD セットアップ（"lxd init" で実行します）が設定するものです。同じことを次のように任意のプロファイルに対してマニュアルで実行できます:

```
lxc profile device add default root disk path=/ pool=default
```

I/O 制限

ストレージデバイスをインスタンスにアタッチする際に、IOPS や MB/s による I/O 制限を、ストレージデバイスに対して設定できます（詳しくは **インスタンス** をご覧ください）。

この制限は Linux の `blkio` cgroup コントローラーを使って適用します。ディスクレベルで I/O の制限ができます（それより粒度の細かい制限はできません）。

この制限は、パーティションやパスではなく、全物理ディスクに対して適用されるので、次のような制限があります:

- 制限は仮想デバイス（例えば device mapper）によって実現しているファイルシステムには適用されません
- 複数のブロックデバイス上に存在するファイルシステムの場合、それぞれのデバイスは同じ制限が適用されます
- 同じディスク上に存在するふたつのディスクデバイスをインスタンスに与えた場合、ふたつのデバイスの制限は平均化されます

すべての I/O 制限は、実際のブロックデバイスにのみ適用されるので、制限を設定する際には、ファイルシステム自身のオーバーヘッドを考慮する必要があるでしょう。このことは、キャッシュされたデータへのアクセスは、制限の影響を受けないことも意味します。

各ストレージバックエンドに対する注意と例

ディレクトリ (dir)

- このバックエンドでは全ての機能を使えますが、他のバックエンドに比べて非常に時間がかかります。これは、イメージを展開したり、インスタンスやスナップショットやイメージのその時点のコピーを作成する必要があるからです。
- ファイルシステムレベルでプロジェクトクォータが有効に設定されている ext4 もしくは XFS で実行している場合は、ディレクトリバックエンドでクォータがサポートされます。

ストレージプール設定

キー	型	デフォルト値	説明
rsync.bwlimit	string	0 (no limit)	ストレージエンティティの転送に rsync を使う必要があるときにソケット I/O に指定する上限を設定
rsync.compression	bool	true	ストレージプールのマイグレーションの際に圧縮を使うかどうか
source	string	-	ブロックデバイスかループファイルかファイルシステムエントリのパス

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

ディレクトリストレージプールを作成するコマンド

- "pool1" という新しいディレクトリプールを作成します

```
lxc storage create pool1 dir
```

- 既存のディレクトリ "pool2" を使います

```
lxc storage create pool2 dir source=/data/lxd
```

CEPH

- イメージとして RBD イメージを使い、インスタンスやスナップショットを作成するためにスナップショットやクローンを実行します
- RBD でコピーオンライトが動作するため、すべての子がなくなるまでは、親のファイルシステムは削除できません。その結果、LXD は削除されたにもかかわらずまだ参照されているオブジェクトに、自動的に `zombie_` というプレフィックスを付与します。そして、参照されなくなるまでそれを保持します。そして安全に削除します
- LXD は OSD ストレージプールを完全にコントロールできると仮定します。LXD OSD ストレージプール内に、LXD が所有しないファイルシステムエンティティを維持し続けないことをおすすめします。LXD がそれらを削除する可能性があるからです
- 複数の LXD インスタンス間で、同じストレージプールを共有することはサポートしないことに注意してください。`lxd import` を使って既存インスタンスをバックアップする目的のときのみ、OSD ストレージプールを複数の LXD インスタンスで共有できます。このような場合には、`ceph.osd.force_reuse` プロパティを true に設定する必要があります。設定しない場合、LXD は他の LXD インスタンスが OSD ストレージプールを使っていることを検出した場合には、OSD ストレージプールの再利用を拒否します
- LXD が使う Ceph クラスタを設定するときは、OSD ストレージプールを保持するために使うストレージエンティティ用のファイルシステムとして `xfs` の使用をおすすめします。ストレージエンティティ用のファイルシステムとして ext4 を使用することは、Ceph の開発元では推奨していません。LXD と関係ない予期しない不規則な障害が発生するかもしれません
- "erasure" タイプの ceph osd プールを使うためには事前に作成した osd pool とメタデータを保管するための "replicated" タイプの別の osd pool が必要です。これは RBD と CephFS が omap をサポートしないために必要となります。そのプールが "earasure coded" かを指定するにはリプリケートされたプールに `ceph.osd.data_pool_name=<erasure-coded-pool-name>` と `source=<replicated-pool-name>` を使用する必要があります。

ストレージプール設定

キー	型	デフォルト値	説明
ceph.cluster_name	string	ceph	新しいストレージプールを作成する ceph クラスタの名前
ceph.osd.data_pool_name	string	-	osd data pool の名前
ceph.osd.force_reuse	bool	false	別の LXD インスタンスで既に使用されている osd ストレージプールの使用を強制するか
ceph.osd.pg_num	string	32	osd ストレージプール用の placement グループの数
ceph.osd.pool_name	string	プールの名前	osd ストレージプールの名前
ceph.rbd.clone_copy	bool	true	フルのデータセットコピーではなく RBD のライトウェイトクローンを使うかどうか
ceph.rbd.du	bool	true	停止したインスタンスのディスク使用データを取得するのに rbd du を使用するかどうか
ceph.rbd.features	string	layering	ボリュームで有効にする RBD の機能のカンマ区切りリスト
ceph.user.name	string	admin	ストレージプールとボリュームの作成に使用する ceph ユーザー
volatile.pool.pristine	string	true	プールが作成時に空かどうか

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
block.filesystem	string	block based driver	volume.block.filesystem と同じ	ストレージボリュームのファイルシステム
block.mount_options	string	block based driver	volume.block.mount_options と同じ	ブロックデバイスのマウントオプション
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

Ceph ストレージプールを作成するコマンド

- Ceph クラスタ "ceph" 内に "pool1" という OSD ストレージプールを作成する

```
lxc storage create pool1 ceph
```

- Ceph クラスタ "my-cluster" 内に "pool1" という OSD ストレージプールを作成する

```
lxc storage create pool1 ceph ceph.cluster_name=my-cluster
```

- ディスク上の名前を "my-osd" で "pool1" という名前の OSD ストレージプールを作成する

```
lxc storage create pool1 ceph ceph.osd.pool_name=my-osd
```

- 既存の OSD ストレージプール "my-already-existing-osd" を使用する

```
lxc storage create pool1 ceph source=my-already-existing-osd
```

- 既存の osd イレージャーコードされたプール "ecpool" と osd リプリケートされたプール "rpl-pool" を使用する

```
lxc storage create pool1 ceph source=rpl-pool ceph.osd.data_pool_name=ecpool
```

CEPHFS

- カスタムストレージボリュームにのみ利用可能
- サーバサイドで許可されていればスナップショットもサポート

ストレージプール設定

キー	型	デフォルト値	説明
ceph.cluster_name	string	ceph	新しいストレージプールを作成する ceph クラスタの名前
ceph.user.name	string	admin	ストレージプールやボリュームを作成する際に使用する Ceph ユーザー名
cephfs.cluster_name	string	ceph	新しいストレージプールを作成する ceph のクラスタ名
cephfs.path	string	/	CEPHFS をマウントするベースのパス
cephfs.user.name	string	admin	ストレージプールとボリュームを作成する際に用いる ceph のユーザー
volatile.pool.pristine	string	true	プールが作成時に空かどうか

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

Btrfs

- インスタンス、イメージ、スナップショットごとにサブボリュームを使い、新しいオブジェクトを作成する際に btrfs スナップショットを作成します
- btrfs は、親コンテナ自身が btrfs 上に作成されているときには、コンテナ内のストレージバックエンドとして使えます（ネストコンテナ）（qgroup を使った btrfs クオータについての注意を参照してください）
- btrfs では qgroup を使ったストレージクオータが使えます。btrfs qgroup は階層構造ですが、新しいサブボリュームは自動的には親のサブボリュームの qgroup には追加されません。このことは、ユーザーが設定されたクオータをエスケープできるということです。もし、クオータを厳格に遵守させたいときは、ユーザーはこのことに留意し、refquota を使った zfs ストレージを使うことを検討してください。

- クオータを使用する際は btrfs のエクステントはイミュータブルであるためブロックが書かれるときにブロックが新しいエクステントに書き込まれ古いブロックはその中のデータが全て参照されなくなるか再書き込みされるまで残ることを考慮することが非常に重要です。これはサブボリューム内の現在のファイルが使用中のスペースの合計量がクオータより小さいにもかかわらずクオータに達することがあり得ることを意味します。これは btrfs サブボリュームの上に生のディスクイメージファイルを使うランダム I/O の性質のため BTRFS 上で VM を使うときによく発生します。VM と btrfs のストレージプールの組み合わせは使わないことを私達は推奨します。もしそれでも使いたい場合は、ディスクイメージファイル内の全てのブロックが qgroup クオータの制限にかかること無く再書き込みできるようにインスタンスのルートディスクの `size.state` プロパティをルートディスクサイズの 2 倍に設定してください。また `btrfs.mount_options=compress-force` ストレージオプションを使うことで圧縮を有効にする副作用として最大のエクステントサイズを縮小させブロックの再書き込みによりストレージの大部分が 2 倍の容量を消費するのを防ぐことができます。ただしこれはストレージプールのオプションですので、プール上の全てのボリュームに影響します。

ストレージプール設定

キー	型	条件	デフォルト値	説明
btrfs.mount_options	string	btrfs driver	user_subvol_rm_allowed	ブロックデバイスのマウントオプション

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

Btrfs ストレージプールを作成するコマンド

- "pool1" という名前の loop を使ったプールを作成する

```
lxc storage create pool1 btrfs
```

- `/some/path` の既存の `btrfs ファイルシステムを使って "pool1" という新しいプールを作成する。

```
lxc storage create pool1 btrfs source=/some/path
```

- `/dev/sdX` 上に "pool1" という新しいプールを作成する

```
lxc storage create pool1 btrfs source=/dev/sdX
```

ループバックデバイスを使った btrfs プールの拡張

LXD では、ループバックデバイスの btrfs プールを直接は拡張できませんが、次のように拡張できます:

```
sudo truncate -s +5G /var/lib/lxd/disks/<POOL>.img
sudo losetup -c <LOOPDEV>
sudo btrfs filesystem resize max /var/lib/lxd/storage-pools/<POOL>/
```

(注意: snap のユーザーは /var/lib/lxd/ の代わりに /var/snap/lxd/common/mntns/var/snap/lxd/common/lxd/ を使ってください)

- LOOPDEV はストレージプールイメージに関連付けられたマウントされたループデバイス（例: /dev/loop8）を参照します。
- マウントされたループデバイスは次のコマンドで確認できます。

```
losetup -l
```

LVM

- イメージ用に LV を使うと、インスタンスとインスタンススナップショット用に LV のスナップショットを使います
- LV で使われるファイルシステムは ext4 です（代わりに xfs を使うように設定できます）
- デフォルトでは、すべての LVM ストレージプールは LVM thin pool を使います。すべての LXD ストレージエンティティ（イメージやインスタンスなど）のための論理ボリュームは、その LVM thin pool 内に作られます。この動作は、lvm.use_thinpool を "false" に設定して変更できます。この場合、LXD はインスタンススナップショットではないすべてのストレージエンティティ（イメージやインスタンスなど）に、通常の論理ボリュームを使います。Thinpool 以外の論理ボリュームは、スナップショットのスナップショットをサポートしていないので、ほとんどのストレージ操作を rsync にフォールバックする必要があります。これは、LVM ドライバがスピードとストレージ操作の両面で DIR ドライバに近づくため、必然的にパフォーマンスに重大な影響を与えることに注意してください。このオプションは、必要な場合のみに選択してください。
- 頻繁にインスタンスとのやりとりが発生する環境（例えば継続的インテグレーション）では、/etc/lvm/lvm.conf 内の retain_min と retain_days を調整して、LXD とのやりとりが遅くならないようにすることが重要です。

ストレージプール設定

キー	型	デフォルト値	説明
lvm.thinpool_name	string	LXDThin-Pool	イメージを作る thin pool 名
lvm.thinpool_metadata_size	string	0 (auto)	thin pool メタデータボリュームのサイズ。デフォルトは LVM が適切なサイズを計算
lvm.use_thinpool	bool	true	ストレージプールは論理ボリュームに thin pool を使うかどうか
lvm.vg.force_reuse	bool	false	既存の空でないボリュームグループの使用を強制
lvm.vg_name	string	name of the pool	作成するボリュームグループ名
rsync.bwlimit	string	0 (no limit)	ストレージエンティティーの転送に rsync を使う場合、I/O ソケットに設定する上限を指定
rsync.compression	bool	true	ストレージプールをマイグレートする際に圧縮を使用するかどうか
source	string	-	ブロックデバイスかループファイルかファイルシステムエントリのパス

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
block.filesystem	string	block based driver	volume.block.filesystem と同じ	ストレージボリュームのファイルシステム
block.mount_options	string	block based driver	volume.block.mount_options と同じ	ブロックデバイスのマウントオプション
lvm.stripes	string	lvm driver	-	新しいボリューム (あるいは thinpool ボリューム) に使用するストライプ数
lvm.stripes.size	string	lvm driver	-	使用するストライプのサイズ (最低 4096 バイトで 512 バイトの倍数を指定)
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

LVM ストレージプールを作成するコマンド

- "pool1" というループバックプールを作成する。LVM ボリュームグループの名前も "pool1" になります

```
lxc storage create pool1 lvm
```

- "my-pool" という既存の LVM ボリュームグループを使う

```
lxc storage create pool1 lvm source=my-pool
```

- ボリュームグループ "my-vg" 内の "my-pool" という既存の LVM thin pool を使う

```
lxc storage create pool1 lvm source=my-vg lvm.thinpool_name=my-pool
```

- `/dev/sdX` に "pool1" という新しいプールを作成する。LVM ボリュームグループの名前も "pool1" になります

```
lxc storage create pool1 lvm source=/dev/sdX
```

- LVM ボリュームグループ名を "my-pool" と名付け `/dev/sdX` を使って "pool1" というプールを新たに作成する

```
lxc storage create pool1 lvm source=/dev/sdX lvm.vg_name=my-pool
```

ZFS

- LXD が ZFS プールを作成した場合は、デフォルトで圧縮が有効になります
- イメージ用に ZFS を使うと、インスタンスとスナップショットの作成にスナップショットとクローンを使います
- ZFS でコピーオンライトが動作するため、すべての子のファイルシステムがなくなるまで、親のファイルシステムを削除できません。ですので、削除されたけれども、まだ参照されているオブジェクトを、LXD はランダムな `deleted/` なパスに自動的にリネームし、参照がなくなりオブジェクトを安全に削除できるようになるまで、そのオブジェクトを保持します。
- 現時点では、ZFS では、プールの一部をコンテナユーザーに権限委譲できません。開発元では、この問題に積極的に取り組んでいます。
- ZFS では最新のスナップショット以外からのリストアはできません。しかし、古いスナップショットから新しいインスタンスを作成することはできます。これにより、新しいスナップショットを削除する前に、スナップショットが確実にリストアしたいものかどうか確認できます。

LXD はリストア中に新しいスナップショットを自動的に破棄するように設定することもできます。これは `volume.zfs.remove_snapshots` プールオプションを使って設定可能です。

しかしインスタンスのコピーも ZFS スナップショットを使うこと、その結果として全ての子孫も消すことなしには最後のコピーより前に取られたスナップショットにインスタンスをリストアすることもできないことに注意してください。

必要なスナップショットを新しいインスタンスにコピーした後に古いインスタンスを削除できますが、インスタンスが持っているかもしれない他のスナップショットを失ってしまいます。

- LXD は ZFS プールとデータセットがフルコントロールできると仮定していることに注意してください。LXD の ZFS プールやデータセット内に LXD と関係ないファイルシステムエンティティを維持しないことをおすすめします。LXD がそれらを消してしまう恐れがあるからです。
- ZFS データセットでクオータを使った場合、LXD は ZFS の "quota" プロパティを設定します。LXD に "refquota" プロパティを設定させるには、与えられたデータセットに対して "zfs.use_refquota" を "true" に設定するか、ストレージプール上で "volume.zfs.use_refquota" を "true" に設定するかします。前者のオプションは、与えられたストレージプールだけに refquota を設定します。後者のオプションは、ストレージプール内のストレージボリュームすべてに refquota を使うようにします。また、ボリュームに"zfs.reserve_space"、ストレージプールに"volume.zfs.reserve_space"を設定することで、ZFS の"quota"/"refquota"に加えて"reservation"/"refreservation"を使用することができます。
- I/O クオータ（IOps/MBs）は ZFS ファイルシステムにはあまり影響を及ぼさないでしょう。これは、ZFS が（SPL を使った）Solaris モジュールの移植であり、I/O に対する制限が適用される Linux の VFS API を使ったネイティブな Linux ファイルシステムではないからです。

ストレージプール設定

キー	型	デフォルト値	説明
size	string	0	ストレージプールのサイズ。バイト単位（suffix も使えます）（現時点では loop ベースのプールと zfs で有効）
source	string	-	ブロックデバイスかループファイルかファイルシステムエントリのパス
zfs.clone_copy	string	true	boolean の文字列を指定した場合は ZFS のフルデータセットコピーの代わりに軽量なクローンを使うかどうかを制御し、 "rebase" という文字列を指定した場合は初期イメージをベースにコピーします。
zfs.export	bool	true	アンマウントの実行中に zpool のエクスポートを無効にする
zfs.pool_name	string	プールの名前	Zpool 名

ストレージボリューム設定

キー	型	条件	デフォルト値	説明
security.shifted	bool	custom volume	false	id シフトオーバーレイを有効にする（複数の独立したインスタンスによるアタッチを許可する）
security.unmapped	bool	custom volume	false	ボリュームへの id マッピングを無効にする
size	string	appropriate driver	volume.size と同じ	ストレージボリュームのサイズ
snapshots.expiry	string	custom volume	-	スナップショットがいつ削除されるかを制御（`1M 2H 3d 4w 5m 6y` のような設定形式を想定）
snapshots.pattern	string	custom volume	snap%d	スナップショット名を表す Pongo2 テンプレート文字列（スケジュールされたスナップショットと名前指定なしのスナップショットに使用）
snapshots.schedule	string	custom volume	-	Cron の書式 (`<minute> <hour> <dom> <month> <dow>`)、またはスケジュールアイリアスのカンマ区切りリスト `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`
zfs.blocksize	string	zfs driver	volume.zfs.blocksize と同じ	ZFS ブロックのサイズを 512〜16MiB の範囲で指定します（2 の累乗でなければなりません）。ブロックボリュームでは、より大きな値が設定されていても、最大値の 128KiB が使用されます。
zfs.remove_snapshots	string	zfs driver	volume.zfs.remove_snapshots と同じ	必要に応じてスナップショットを削除するかどうか
zfs.use_refquota	string	zfs driver	volume.zfs.zfs_refquota と同じ	領域の quota の代わりに refquota を使うかどうか
zfs.reserve_space	string	zfs driver	false	qouta/refquota に加えて reservation/refreservation も使用するかどうか

ZFS ストレージプールを作成するコマンド

- "pool1" というループバックプールを作成する。ZFS の Zpool 名も "pool1" となります

```
lxc storage create pool1 zfs
```

- ZFS Zpool 名を "my-tank" とし、"pool1" というループバックプールを作成する

```
lxc storage create pool1 zfs zfs.pool_name=my-tank
```

- 既存の ZFS Zpool "my-tank" を使う

```
lxc storage create pool1 zfs source=my-tank
```

- 既存の ZFS データセット "my-tank/slice" を使う

```
lxc storage create pool1 zfs source=my-tank/slice
```

- /dev/sdX 上に "pool1" という新しいプールを作成する。ZFS Zpool 名も "pool1" となります

```
lxc storage create pool1 zfs source=/dev/sdX
```

- /dev/sdX 上に "my-tank" という ZFS Zpool 名で新しいプールを作成する

```
lxc storage create pool1 zfs source=/dev/sdX zfs.pool_name=my-tank
```

ループバックの ZFS プールの拡張

LXD からは直接はループバックの ZFS プールを拡張できません。しかし、次のようにすればできます:

```
sudo truncate -s +5G /var/lib/lxd/disks/<POOL>.img
sudo zpool set autoexpand=on lxd
sudo zpool online -e lxd /var/lib/lxd/disks/<POOL>.img
sudo zpool set autoexpand=off lxd
```

(注意: snap のユーザーは /var/lib/lxd/ の代わりに /var/snap/lxd/common/lxd/ を使ってください)

既存のプールで TRIM を有効にする

LXD は ZFS 0.8 以降で新規に作成された全てのプールに TRIM サポートを自動で有効にします。

これによりコントローラーによるブロック再利用を改善し SSD の寿命を延ばすことができます。さらにループバックの ZFS プールを使用している場合はルートファイルシステムの空きスペースを解放できます。

0.8 より古い ZFS を 0.8 にアップグレードしたシステムでは、以下の 1 度きりの操作で TRIM の自動実行を有効にできます。

- zpool upgrade ZPOOL-NAME
- zpool set autotrim=on ZPOOL-NAME
- zpool trim ZPOOL-NAME

これにより現在未使用のスペースに TRIM を実行するだけでなく、将来 TRIM が自動的に実行されるようになります。

3.2.13 仮想マシン

イントロダクション

仮想マシンはコンテナと共に LXD でサポートされる新しいインスタンスタイプです。

仮想マシンは `qemu` を使って実装されています。

現状はコンテナで利用可能な全ての機能が VM には実装されているわけではないことにご注意ください。しかし、私達はコンテナと同等の機能を目指して引き続き努力します。

設定

有効な設定項目については **インスタンスの設定** を参照してください。

3.3 イメージ

3.3.1 アーキテクチャ

LXD はちょうど LXC と同じように Linux カーネルと Go でサポートされるあらゆるアーキテクチャで稼働することができます。

コンテナ、コンテナのスナップショットやイメージのように LXD のいくつかのオブジェクトはアーキテクチャに依存しています。

このドキュメントではサポートされているアーキテクチャ、それらの（データベースで使われる）ユニークな識別子、それらがどのように名前付けされるべきかといくつかの注釈をリストアップします。

LXD が問題とするのはカーネルアーキテクチャであり、ツールチェインで決定される特定のユーザースペースのフレーバーではないことに注意してください。

これは LXD は armv7 hard-float を armv7 soft-float と同じとして扱い、両方を "armv7" として参照することを意味します。もしユーザーにとって有用であれば正確なユーザースペースの ABI がイメージとコンテナプロパティとして設定可能となり、簡単に問い合わせすることを許可します。

アーキテクチャ

ID	Name	Notes	Personalities
1	i686	32bit Intel x86	
2	x86_64	64bit Intel x86	x86
3	armv7l	32bit ARMv7 little-endian	
4	aarch64	64bit ARMv8 little-endian	armv7 (optional)
5	ppc	32bit PowerPC big-endian	
6	ppc64	64bit PowerPC big-endian	powerpc
7	ppc64le	64bit PowerPC little-endian	
8	s390x	64bit ESA/390 big-endian	
9	mips	32bit MIPS	
10	mips64	64bit MIPS	mips
11	riscv32	32bit RISC-V little-endian	
12	riscv64	64bit RISC-V little-endian	

上記のアーキテクチャ名は通常は Linux のカーネルアーキテクチャ名と揃えてあります。

3.3.2 cloud-init

- cloud-init.vendor-data
- cloud-init.user-data
- cloud-init.network-config

しかし、cloud-init を使おうとする前に、これから使おうとするイメージ・ソースをどれにするかをまず決めてください。というのも、全てのイメージに cloud-init パッケージがインストールされているわけではないからです。

ubuntu と ubuntu-daily の remote にあるイメージは全て cloud-init が有効です。images remote のイメージで cloud-init が有効なイメージがあるものは /cloud という接尾辞がつきます（例: images:ubuntu/20.04/cloud）。

vendor-data と user-data は同じルールに従いますが、以下の制約があります。

- ユーザーは vendordata に対して究極のコントロールが可能です。実行を無効化したりマルチパートの入力の特定のパートの処理を無効化できます。
- デフォルトでは初回ブート時のみ実行されます。
- vendordata はユーザーにより無効化できます。インスタンスの実行に vendordata の使用が必須な場合は vendordata を使うべきではありません。
- ユーザーが指定した cloud-config は vendordata の cloud-config の上にマージされます。

LXD のインスタンスではインスタンスの設定よりもプロファイル内の vendor-data を使うべきです。

cloud-config の例はこちらにあります。 https://cloudinit.readthedocs.io/en/latest/topics/examples.html

カスタムネットワーク設定

cloud-init は、network-config データを使い、Ubuntu リリースに応じて ifupdown もしくは netplan のどちらかを使って、システム上の関連する設定を行います。

デフォルトではインスタンスの eth0 インタフェースで DHCP クライアントを使うようになっています。

これを変更するためには設定ディクショナリ内の cloud-init.network-config キーを使ってあなた自身のネットワーク設定を定義する必要があります。その設定がデフォルトの設定をオーバーライドするでしょう（これはテンプレートがそのように構成されているためです）。

例えば、ある特定のネットワーク・インタフェースを静的 IPv4 アドレスを持ち、カスタムのネームスペースを使うようにするには、以下のようにします。

```
config:
  cloud-init.network-config: |
```

(次のページに続く)

(前のページからの続き)

```
version: 1
config:
  - type: physical
    name: eth1
    subnets:
      - type: static
        ipv4: true
        address: 10.10.101.20
        netmask: 255.255.255.0
        gateway: 10.10.101.1
        control: auto
  - type: nameserver
    address: 10.10.10.254
```

この結果、インスタンスの rootfs には以下のファイルが作られます。

- `/var/lib/cloud/seed/nocloud-net/network-config`
- `/etc/network/interfaces.d/50-cloud-init.cfg` (ifupdown を使う場合)
- `/etc/netplan/50-cloud-init.yaml` (netplan を使う場合)

3.3.3 イメージの扱い

LXD はイメージをベースとしたワークフローを使用します。 LXD にはビルトインのイメージ・ストアがあり、ユーザーが外部のツールがそこからイメージをインポートできます。

その後、それらのイメージからコンテナが起動されます。

ローカルのイメージを使ってリモートのインスタンスを起動できますし、リモートのイメージを使ってローカルのインスタンスを起動することもできます。こういったケースではイメージはターゲットの LXD にキャッシュされます。

ソース

LXD は 3 つの異なるソースからのイメージのインポートをサポートします。

- リモートのイメージサーバ (LXD か simplestreams)
- イメージファイルの direct push
- リモートのウェブサーバ上のファイル

リモートのイメージサーバ (LXD か simplestreams)

これは最も一般的なイメージソースで 3 つの選択肢のうちインスタンスの作成時に直接サポートされている唯一の選択肢です。

この選択肢では、イメージサーバは検証されるために必要な証明書（HTTPS のみがサポートされます）と共にターゲットの LXD サーバに提供されます。

次にイメージそのものがフィンガープリント (SHA256) あるいはエイリアスによって選択されます。

CLI の視点では、これは以下の一般的なアクションによって実行されます。

- lxc launch ubuntu:20.04 u1
- lxc launch images:centos/8 c1
- lxc launch my-server:SHA256 a1
- lxc image copy images:gentoo local: --copy-aliases --auto-update

上記の `ubuntu` と `images` のケースではリモートは simplestreams を読み取り専用のサーバプロトコルとして使用し、イメージの複数のエイリアスの 1 つによりイメージを選択します。

`my-server` リモートのケースでは別の LXD サーバがあり、上記の例ではフィンガープリントによってイメージを選択します。

イメージファイルの direct push

これは主に外部サーバから直接イメージを取得できない隔離された環境で有用です。

そのような状況ではイメージファイルは他のシステムで以下のコマンドを使ってダウンロードできます。

- lxc image export ubuntu:20.04

その後ターゲットのシステムにイメージを転送してローカルイメージストアーに手動でインポートします。

- lxc image import META ROOTFS --alias ubuntu-20.04

`lxc image import` は統合イメージ (単一ファイル) と分割イメージ (2 つのファイル) の両方をサポートします。上の例では後者を使用しています。

リモートのウェブサーバ上のファイル

単一のイメージをユーザーに配布するためだけにフルのイメージサーバを動かすことの代替として、LXD は URL を指定してイメージをインポートするのもサポートしています。

ただし、この方法にはいくつか制限があります。

- 統合ファイル（単一ファイル）のみがサポートされます
- リモートサーバが追加の http ヘッダーを返す必要があります

LXD はサーバに問い合わせをする際に以下のヘッダーを設定します。

- `LXD-Server-Architectures` にはクライアントがサポートするアーキテクチャーのカンマ区切りリストを設定します
- `LXD-Server-Version` には使用している LXD のバージョンを設定します

リモートサーバが `LXD-Image-Hash` と `LXD-Image-URL` を設定することを期待します。前者はダウンロードされるイメージの SHA256 ハッシュで後者はイメージをダウンロードする URL です。

これによりかなり複雑なイメージサーバがカスタムヘッダーをサポートする基本的なウェブサーバだけで実装できます。

クライアント側では以下のように使用できます。

```
lxc image import URL --alias some-name
```

インスタンスやスナップショットを新しいイメージとして公開する

インスタンスやスナップショットの 1 つを新しいイメージに変換できます。これは `lxc publish` で CLI 上で実行できます。

これを行う際には、たいていの場合公開する前にインスタンスのメタデータやテンプレートを `lxc config metadata` と `lxc config template` コマンドを使って整理するのが良いでしょう。さらにホストの SSH キーや dbus/systemd の machine-id などインスタンスに固有な状態も削除するのが良いでしょう。

インスタンスから tarball を生成した後圧縮する必要があるので、公開のプロセスはかなり時間がかかるかもしれません。この操作は特に I/O と CPU の負荷が高いので、公開操作は LXD により 1 つずつ順に実行されます。

キャッシュ

リモートのイメージからインスタンスを起動する時、リモートのイメージがローカルのイメージ・ストアにキャッシュ・ビットをセットした状態でダウンロードされます。イメージは、 images.remote_cache_expiry に設定された日数だけ使われない (新たなインスタンスが起動されない) か、イメージが期限を迎えるか、どちらか早いほうが来るまで、プライベートなイメージとしてローカルに保存されます。

LXD はイメージから新しいインスタンスが起動される度にイメージの last_used_at プロパティを更新することで、イメージの利用状況を記録しています。

自動更新

LXD はイメージを最新に維持できます。デフォルトではエイリアスで指定しリモートサーバから取得したイメージは LXD によって自動更新されます。これは images.auto_update_cached という設定で変更できます。

(images.auto_update_interval が設定されない限り) 起動時とその後 6 時間毎に、 LXD デーモンはイメージ・ストア内で自動更新対象となっていてダウンロード元のサーバが記録されている全てのイメージのより新しいバージョンがあるかを確認します。

新しいイメージが見つかったら、イメージ・ストアにダウンロードされ、古いイメージを指していたエイリアスは新しいイメージを指すように変更され、古いイメージはストアから削除されます。

リモート・サーバからイメージを手動でコピーする際に、特定のイメージを最新に維持するように設定することもできます。

ユーザーがイメージのキャッシュから新しいインスタンスを作成しようとした時に、アップストリームの新しいイメージ更新が公開されており、ローカルの LXD がキャッシュに古いイメージを持っている場合は、 LXD はインスタンスの作成を遅らせるのではなく、古いバージョンのイメージを使います。

この振る舞いは現在のイメージが自動更新されるように設定されている時のみに発生し、 images.auto_update_interval を 0 にすることで無効にできます。

プロファイル

lxc image edit コマンドを使ってイメージにプロファイルのリストを関連付けできます。イメージにプロファイルを関連付けた後に起動したインスタンスはプロファイルを順番に適用します。プロファイルのリストとして nil を指定すると default プロファイルのみがイメージに関連付けされます。空のリストを指定すると、 default プロファイルも含めて一切のプロファイルをイメージに適用しません。イメージに関連付けされたプロファイルは lxc launch の --profile と --no-profiles オプションを使ってインスタンス起動時にオーバーライドできます。

特別なイメージプロパティ

プレフィックス ***requirements*** で始まるイメージプロパティ (例：requirements.XYZ) は、LXD がホストシステムと当該イメージで生成されるインスタンスの互換性を判断するために使用されます。これらの互換性がない場合には LXD はそのインスタンスを起動しません。

現在のところ、以下の要件がサポートされています。

キー	タイプ	デフォルト	説明
requirements.secureboot	string	-	"false" に設定すると、イメージがセキュアブートで起動しないことを示します。
requirements.cgroup	string	-	"v1" "に設定されている場合、ホストで CGroupV1 が実行されている必要があることを示します。

イメージの形式

LXD は現状 2 つの LXD に特有なイメージの形式をサポートします。

1 つめは統合された tarball で、単一の tarball がインスタンスの root と必要なメタデータの両方を含みます。

2 つめは分離されたモデルで、 2 つのファイルを使い、 1 つは root を含み、もう一つはメタデータを含みます。

LXD 自身によって生成されるのは前者の形式で、 LXD 特有のイメージを使う際はこちらの形式を使うべきです。

後者は、今日既に利用可能なものとして存在している LXD 以外の rootfs tarball を使ってイメージを簡単に作成できるように想定されているものです。

統合された tarball

tarball は圧縮できます。そして次のものを含みます。

- `rootfs/`
- `metadata.yaml`
- `templates/` (省略可能)

このモードではイメージの識別子は tarball の SHA-256 です。

分離された tarball

2 つの (圧縮しても良い) tarball 。 1 つはメタデータ、もう 1 つは rootfs です。

`metadata.tar` は以下のものを含みます。

- `metadata.yaml`
- `templates/` (省略可能)

`rootfs.tar` は、そのルートに Linux の root ファイルシステムを含みます。

このモードではイメージの識別子はメタデータと rootfs の tarball を (この順番で) 結合したものの SHA-256 です。

サポートされている圧縮形式

LXD は広範な tarball の圧縮アルゴリズムをサポートしますが、互換性のために gzip か xz が望ましいです。

分離されたイメージではコンテナの場合は rootfs ファイルはさらに squashfs 形式でフォーマットすることもできます。仮想マシンでは `rootfs.img` ファイルは常に qcow2 であり、オプションで qcow2 のネイティブ圧縮を使って圧縮することもできます。

中身

コンテナでは rootfs のディレクトリ (あるいは tarball) は完全なファイルシステムのツリーを含み、それが / になります。VM ではこれは代わりに `rootfs.img` ファイルでメインのディスクデバイスになります。

テンプレートのディレクトリはコンテナ内で使用される pongo2 形式のテンプレート・ファイルを含みます。

`metadata.yaml` はイメージを (現状は) LXD で稼働されるために必要な情報を含んでおり、これは以下のものを含みます。

```
architecture: x86_64
creation_date: 1424284563
properties:
  description: Ubuntu 20.04 LTS Intel 64bit
  os: Ubuntu
  release: focal 20.04
templates:
  /etc/hosts:
    when:
      - create
      - rename
    template: hosts.tpl
```

(次のページに続く)

(前のページからの続き)

```
    properties:
      foo: bar
  /etc/hostname:
    when:
      - start
    template: hostname.tpl
  /etc/network/interfaces:
    when:
      - create
    template: interfaces.tpl
    create_only: true
```

architecture と creation_date の項目は必須です。 properties は単にイメージのデフォルト・プロパティの組です。 os, release, name と description の項目は必須ではないですが、記載されることが多いでしょう。

テンプレートで when キーは以下の 1 つあるいは複数が指定可能です。

- create (そのイメージから新しいインスタンスが作成されたときに実行される)
- copy (既存のインスタンスから新しいインスタンスが作成されたときに実行される)
- start (インスタンスが開始される度に実行される)

テンプレートは常に以下のコンテキストを受け取ります。

- trigger: テンプレートを呼び出したイベントの名前 (string)
- path: テンプレート出力先のファイルのパス (string)
- container: インスタンスのプロパティ (name, architecture, privileged そして ephemeral) の key/value の map (map[string]string) (廃止予定。代わりに instance を使用してください)
- instance: インスタンスのプロパティ (name, architecture, privileged そして ephemeral) の key/value の map (map[string]string)
- config: インスタンスの設定の key/value の map (map[string]string)
- devices: インスタンスに割り当てられたデバイスの key/value の map (map[string]map[string]string)
- properties: metadata.yaml に指定されたテンプレートのプロパティの key/value の map (map[string]string)

create_only キーを設定すると LXD が存在しないファイルだけを生成し、既存のファイルを上書きしないようにできます。

一般的な規範として、パッケージで管理されているファイルをテンプレートの生成対象とすべきではないです。そうしてしまうとインスタンスの通常の操作で上書きされてしまうでしょう。

利便性のため、以下の関数が pongo のテンプレートで利用可能となっています。

- `config_get("user.foo", "bar")` => `user.foo` の値か、未設定の場合は `"bar"` を返します。

3.4 オペレーション

3.4.1 LXD サーバをバックアップする

何をバックアップするか

LXD サーバのバックアップを計画する際は、 LXD に保管／管理されている全ての異なるオブジェクトについて考慮してください。

- インスタンス (データベースのレコードとファイルシステム)
- イメージ (データベースのレコード、イメージファイル、そしてファイルシステム)
- ネットワーク (データベースのレコードと状態ファイル)
- プロファイル (データベースのレコード)
- ストレージボリューム (データベースのレコードとファイルシステム)

データベースだけをバックアップあるいはインスタンスだけをバックアップしても完全に機能するバックアップにはなりません。

ディザスターリカバリのシナリオによっては、上記のようなバックアップも妥当かもしれませんが、素早くオンラインに復帰することが目標なら、使用している LXD の全ての異なるピースを考慮してください。

フルバックアップ

フルバックアップは `/var/lib/lxd` あるいは snap ユーザーの場合は `/var/snap/lxd/common/lxd` の全体を含みます。

LXD が外部ストレージを使用している場合はそれらも適切にバックアップする必要があります。これは LVM ボリュームグループや ZFS プールなど LXD に直接含まれていないあらゆる外部のリソースです。

リストアにはリストア先のサーバ上の LXD の停止、 LXD ディレクトリの削除、そしてバックアップと必要な外部リソースのリストアを含みます。

snap パッケージを使っておらず、かつシステムに /etc/subuid と /etc/subgid ファイルがある場合、 `lxd` と `root` ユーザーの両方についてこれらのファイルあるいは少なくともこれらのファイル内のエントリーを復元することも良い考えです（コンテナのファイルシステムの不要なシフトを防ぎます）。

その後再び LXD を起動し、全てが正常に動作するか確認してください。

LXD サーバのセカンダリバックアップ

LXD は 2 つのホスト間でインスタンスとストレージボリュームのコピーと移動をサポートしています。

ですので予備のサーバがあれば、インスタンスとストレージボリュームを時々そのセカンダリサーバにコピーしておき、オフラインの予備あるいは単なるストレージサーバとして稼働させることが可能です。そして必要ならばそこからインスタンスをコピーして戻すことができます。

インスタンスのバックアップ

`lxc export` コマンドがインスタンスをバックアップの tarball にエクスポートするのに使えます。これらの tarball はデフォルトで全てのスナップショットを含みますが、同じストレージプールバックエンドを使っている LXD サーバにリストアすることがわかっていれば「最適化」された tarball を取得することもできます。

サーバ上にインストールされたどんな圧縮ツールでも `--compression` を指定することで利用可能です。LXD 側でのバリデーションはなく、 LXD から実行可能で `-c` オプションで標準出力への出力をサポートしているコマンドであれば動作します。

これらの tarball はあなたが望むどんなファイルシステム上にどのようにでも保存することができ、 `lxc import` コマンドを使って LXD にインポートして戻すことができます。

ディザスタリカバリ

LXD は `lxd recover` コマンドを提供しています（通常の `lxc` コマンドではなく `lxd` コマンドであることに注意）。これはインタラクティブな CLI ツールでデータベース内に存在する全てのストレージプールをスキャンしリカバリー可能な焼失したボリュームを探します。また（ディスク上には存在するがデータベース内には存在しない）任意の未知のストレージプールの詳細をユーザーが指定してそれらに対してもスキャンを試みることもできます。

指定されたインスタンスを復元するのに必要な全ての（インスタンス設定、アタッチしたデバイス、ストレージボリューム、プール設定も含めた）情報を含む各インスタンスのストレージボリューム内の `backup.yaml` ファイルを LXD は保管しているため、それをインスタンス、ストレージボリューム、ストレージプールのデータベースレコードをリビルドするのに使用できます。

`lxd recover` ツールはストレージプールを（まだマウントされていなければ）マウントし、 LXD に関係すると思われる未知のボリュームをスキャンしようと試みます。各インスタンスボリュームについては LXD はマウントして `backup.yaml` ファイルにアクセスしようと試みます。その後 `backup.yaml` ファイルの内容と（対応するスナップショットなど）ディスク上に実際に存在するものとを比較してある程度の整合性チェックを行い、問題なければデータベースのレコードを再生成します。

ストレージプールのデータベースレコードも作成が必要な場合、ディスカバリーフェーズにユーザーが入力した情報よりも、インスタンスの `backup.yaml` ファイルを設定のベースとして優先して使用します。ただし、それが無い場合はユーザーが入力した情報をもとにプールのデータベースレコードを復元するようにフォールバックします。

3.4.2 クラスタリング

LXD はクラスタリングモードで実行できます。クラスタリングモードでは複数台の LXD サーバが同じ分散データベースを共有し、REST API や lxc クライアントで統合管理できます。

この機能は API 拡張の "clustering" の一部として導入しました。

クラスタの形成

まず、ブートストラップノードを選択する必要があります。既存の LXD サーバでも新しいインスタンスでもブートストラップノードになれます。ブートストラップノードとなるサーバを決めた後は、ブートストラップノードを初期化し、それからクラスタへ追加ノードを参加させます。この処理はインタラクティブに行えますし、前もって定義ファイルを作成しても行えます。

クラスタに追加するノードはすべて、ストーレージプールとネットワークについて、ブートストラップノードと同じ構成を持たなければなりません。ノード特有の設定として持てる唯一の設定は、ストレージプールに対する `source` と `size`、ネットワークに対する `bridge.external_interface` です。

クラスタ内のノード数としては 3 以上を強く推奨します。これは少なくとも 1 ノードが落ちても分散状態のクオラムを確立できるからです（分散状態は Raft アルゴリズムを使ってレプリケーションされている SQLite データベースに保管されています）。ノード数が 3 より小さくなるとクラスタ内のただ 1 つのノードだけが SQLite データベースを保管します。第 3 のノードがクラスタに参加したときに、第 2 と第 3 のノードがデータベースの複製を受け取ります。

インタラクティブに行う方法

`lxd init` を実行し、最初の質問（"Would you like to use LXD clustering?"）に `yes` と答えます。そして、そのノードを特定する名前、他のノードが接続するための IP もしくは DNS アドレスを選択します。そして、既存のクラスタに加わるかどうかの質問には no と答えます。最後に、オプショナルでストレージプールとネットワークブリッジを作成できます。これで、最初のクラスタノードが起動し、ネットワークが利用できるようになります。

更に追加のノードをクラスタに追加できます。しかし、追加ノードの既存データはすべて失われるため、追加のノードは完全に新しい LXD サーバであるか、追加前にすべての情報をクリアしたノードである必要があります。

既存のクラスタにメンバーを追加するには 2 つの方法があります。トラスト・パスワードを使うか参加トークンを使うかです。新規メンバーの参加トークンは既存のクラスタで次のコマンドを使って事前に生成します。

```
lxc cluster add <new member name>
```

これで 1 度だけ使える参加トークンが生成されます。これは `lxd init` の参加トークンを質問するプロンプトで使えます。参加トークンは既存のオンラインメンバーのアドレスと 1 度だけ使えるシークレットとクラスタの証明書のフィンガープリントを含んでいます。参加トークンは `lxd init` 実行時に聞かれる複数の質問に自動的に回答するのに使われるので、回答が必要な質問の数を減らすことができます。

あるいは参加トークンの代わりにトラスト・パスワードを使うこともできます。

ノードを追加するために、`lxd init` を実行し、クラスタリングを使うかどうかの質問に `yes` と答えます。ブートストラップノード、それまでに参加したノードとは異なる名前を指定します。IP もしくは DNS アドレスを指定し、既存のクラスタに加わるかどうかの質問には `yes` と答えます。

参加トークンがある場合は参加トークンを持っているかの質問に `yes` と回答し、参加トークンを求めるプロンプトに対してトークンをコピーします。

参加トークンは持っていないがトラスト・パスワードを持っている場合は参加トークンを持っているかの質問に `no` と答えます。その後クラスタ内の既存のノードのアドレスを 1 つ選び表示されたフィンガープリントが既存メンバーのクラスタ証明書にマッチするかをチェックします。

サーバごとの設定

上で述べたように LXD のクラスタメンバーはたいていは同一のシステムであると想定されます。

しかしディスクの多少の順序の違いやネットワークインターフェースの名前が違いに適応するため、 LXD は一部の設定をサーバごとに記録します。クラスタにそのような設定が存在するときは、新しく追加されるサーバにはその設定に対する値を指定する必要があります。

これはたいていの場合インタラクティブな `lxd init` の実行時に、ストレージやネットワークに関連するいくつかの設定キーの値をユーザーに尋ねることで実現されます。

典型的には以下のような項目が対象になります。

- ストレージプールのソースデバイス（空にするとループデバイスを作成）
- ZFS zpool の名前（デフォルトは LXD プールの名前）
- ブリッジネットワークの外部インターフェース（空にすると追加しない）
- マネージドされた物理あるいは macvlan ネットワークの親のネットワークデバイスの名前（設定必須）

どういう質問があるかは `/1.0/cluster` API エンドポイントに問い合わせることで事前に確認できます（スクリプトを書く際に有効です）。これは `lxc query /1.0/cluster` や他の API クライアントを使って実行できます。

事前に定義して行う方法

事前にブートストラップノードの設定内容を書いた定義ファイルを作成できます。例えば:

```
config:
  core.trust_password: sekret
  core.https_address: 10.55.60.171:8443
  images.auto_update_interval: 15
```

(次のページに続く)

(前のページからの続き)

```
storage_pools:
- name: default
  driver: dir
networks:
- name: lxdbr0
  type: bridge
  config:
    ipv4.address: 192.168.100.14/24
    ipv6.address: none
profiles:
- name: default
  devices:
    root:
      path: /
      pool: default
      type: disk
    eth0:
      name: eth0
      nictype: bridged
      parent: lxdbr0
      type: nic
cluster:
  server_name: node1
  enabled: true
```

定義ファイルを作成したあと、`cat <preseed-file> | lxd init --preseed` を実行し、最初のノードを作成します。

次に、他のノードのブートストラップファイルを作成します。`cluster` セクションに、追加するノード固有のデータと設定値を指定するだけです。

ターゲットとなるブートストラップノードのアドレスと証明書を必ず含めてください。`cluster_certificate` に対する YAML 互換のエントリーを作成するには、`sed ':a;N;$!ba;s/\n/\n\n/g' /var/lib/lxd/cluster.crt` （あるいは snap ユーザーは `sed ':a;N;$!ba;s/\n/\n\n/g' /var/snap/lxd/common/lxd/cluster.crt`）のようにコマンドを実行します。このコマンドはブートストラップノードで実行する必要があります。 `cluster_certificate_path` キー（これにはクラスタ証明書の有効なパスを設定します）を `cluster_certificate` キーの代わりに使うこともできます。

例えば:

```
cluster:
  enabled: true
  server_name: node2
  server_address: 10.55.60.155:8443
  cluster_address: 10.55.60.171:8443
  cluster_certificate: "-----BEGIN CERTIFICATE-----

opyQ1VRpAg2sV2C4W8irbNqeUsTeZZxhLqp4vNOXXBBrSqUCdPu1JXADV0kavg1l

2sXYoMobyV3K+RaJgsr1OiHjacGiGCQT3YyNGGY/n5zgT/8xI0Dquvja0bNkaf6f

...

-----END CERTIFICATE-----
"
  cluster_password: sekret
  member_config:
  - entity: storage-pool
    name: default
    key: source
    value: ""
```

クラスタ参加トークンを使用してクラスタに参加する際は、下記のフィールドは省略できます。

- server_name
- cluster_address
- cluster_certificate
- cluster_password

そして代わりにフルのトークンを `cluster_token` フィールドを使って渡せます。

クラスタの管理

クラスタが形成されると、`lxc cluster list` を実行して、ノードのリストと状態を見ることができます。ノードそれぞれのもっと詳細な情報は `lxc cluster show <node name>` を実行して取得できます。

クラスタメンバーの設定

各クラスタメンバーは以下のサポートされるネームスペース内で独自のキー・バリュー設定を持てます。

- `user` (ユーザーのメタデータ用に自由形式のキー・バリュー)
- `scheduler` (メンバーが自クラスタによりどのように動的にターゲットされるかに関連するオプション)

現状サポートされるキーは以下の通りです。

キー	型	デフォルト値	説明
scheduler.instance	string	all	`all` の場合、インスタンスの最も少ないメンバーが、インスタンス作成の対象として自動的に選択されます。 `manual` の場合、インスタンスは `--target` が指定されたときのみメンバーにターゲットされます。`group` の場合、インスタンスは `--target=@<group>` が指定されたときのみメンバーにターゲットされます。
user.*	string	-	自由形式のユーザーのキー・バリュー・ストレージ (検索で使用可能)

クラスタメンバーの役割

以下の役割が LXD クラスタメンバーに設定できます。自動的な役割は LXD 自身によって設定され、ユーザーは変更できません。

役割	自動的	説明
database	yes	分散データベースの投票メンバー
database-leader	yes	分散データベースの現在のリーダー
database-standby	yes	分散データベースのスタンバイ (投票しない) メンバー
event-hub	no	LXD 内部イベントの交換点 (ハブ) (最低 2 つ必要)
ovn-chassis	no	OVN ネットワークのアップリンクゲートウェイ候補

投票 (voting) メンバーとスタンバイメンバー

クラスタは状態を保管するために分散 **データベース** を使用します。クラスタ内の全てのノードはユーザーのリクエストに応えるためにそのような分散データベースにアクセスする必要があります。

クラスタ内に多くのノードがある場合、それらのうちいくつかだけがデータベースのデータを複製するために選ばれます。選ばれた各オンードは投票者 (voter) としてあるいはスタンバイとしてデータを複製できます。データベース（とそれに由来するクラスタ）は投票者の過半数がオンラインである限り利用可能です。別の投票者が正常にシャットダウンした時やオフラインであると検出された時はスタンバイノードが自動的に投票者に昇格されます。

投票ノードのデフォルト数は 3 で、スタンバイノードのデフォルト数は 2 です。これは 1 度に最大で 1 つの投票ノードの電源を切る限りあなたのクラスタは稼働し続けることを意味します。

投票ノードとスタンバイノードの望ましい数は以下のように変更できます。

```
lxc config set cluster.max_voters <n>
```

そして

```
lxc config set cluster.max_standby <n>
```

投票者の最大数は奇数で最低でも 3 であるという制約があります。一方、スタンバイノードは 0 から 5 の間でなければなりません。

ノードの削除

クラスタからノードをクリーンに削除するには、`lxc cluster remove <node name>` を使います。

オフラインノードとフォールトトレランス

都度、選出されたクラスタリーダーが存在し、そのリーダーが他のノードの健全性をモニタリングします。20 秒以上ノードがダウンした場合は、ステータスは OFFLINE とマークされ、そのノード上での操作はできなくなります。また、すべてのノードで状態の変更が必要な操作が可能です。

リーダーがオフラインに移行した場合、他のノードが新しいリーダーに選出されます。

オフラインノードがオンラインに戻るとすぐに、ふたたび操作できるようになります。

ノードをオンラインに戻せないとき、ノードをオンラインに戻したくないときは、`lxc cluster remove --force <node name>` を使ってクラスタからノードを削除できます。

反応しないノードがオフラインと認識されるまでの秒数は以下のようにして変更できます。

```
lxc config set cluster.offline_threshold <n seconds>
```

最小値は 10 秒です。

ノードのアップグレード

クラスタをアップグレードするためには、すべてのノードをアップグレードし、すべてが確実に同じバージョンの LXD にする必要があります。

単一のノードをアップグレードするには、単にホスト上で（snap や他のパッケージ管理システムを使って）lxd/lxc バイナリをアップグレードし、lxd デーモンを再起動します。

デーモンの新バージョンでデータベーススキーマや API が変更になった場合は、再起動したノードは Blocked 状態に移行する可能性があります。これは、クラスタ内にまだアップグレードされていないノードが存在し、その上で古いバージョンが動作している場合に起こります。ノードが Blocked 状態にあるとき、このノードは LXD API リクエストを一切受け付けません（詳しく言うと、実行中のインスタンスは動き続けますが、ノード上の lxc コマンドは動きません）。

ブロックされていないノード上で `lxc cluster list` を実行すると、ノードがブロックされているかどうかを確認できます。

残りのノードのアップグレードを進めると、最後のノードをアップグレードするまでは、ノードはすべて Blocked 状態に移行します。その時点で、Blocked ノードは古いノードが残っていないかを確認し、再度操作できるようになります。

クラスタメンバーの待避と復元

再起動が必要なシステムアップデートを適用する定例メンテナンスやハードウェアの構成変更などで、指定したサーバ上の全てのインスタンスを空にしたいことが時々あります。

これは `lxc cluster evacuate <NAME>` で実行できます。このコマンドはそのサーバ上の全てのインスタンスをマイグレートし、他のクラスタメンバーに移動します。待避が行われたクラスタメンバーは "evacuated" 状態に遷移し、そこではインスタンスの生成は禁止されます。

メンテナンスが完了したら `lxc cluster restore <NAME>` を実行するとサーバを通常の実行状態に戻し、このサーバ上に元々あって一時的に他のサーバに移動していたインスタンスをこのサーバ上に戻します。

指定のインスタンスの挙動は `cluster.evacuate` インスタンス設定キーで指定できます。`boot.host_shutdown_timeout` 設定キーを尊重してインスタンスはクリーンにシャットダウンされます。

Failure domains

Failure domain はシャットダウンしたかクラッシュしたクラスタメンバーに role を割り当てる際にどのノードが優先されるかを指示するのに使います。例えば、現在 database role を持つクラスタメンバーがシャットダウンした場合、 LXD は同じ failure domain 内の別のクラスタメンバーが存在すればそれに database role を割り当てようと試みます。

クラスタメンバーの failure domain を変更するには `lxc cluster edit <member>` コマンドラインツールか、`PUT /1.0/cluster/<member>` REST API が使用できます。

クォーラム消失からの復旧

各 LXD クラスタはデータベースノードとして機能するメンバーを最大 3 つまで持つことができます。恒久的にデータベースノードとして機能するクラスタメンバーの過半数を失った場合 (例えば 3 メンバーのクラスタで 2 メンバーを失った場合)、クラスタは利用不可能になります。しかし、 1 つでもデータベースノードが生き残っていれば、クラスタをリカバーすることができます。

クラスタメンバーがデータベースノードとして設定されているかどうかをチェックするには、クラスタのいずれかの生き残っているメンバーにログオンして以下のコマンドを実行します。

```
lxd cluster list-database
```

これは LXD デーモンが実行中でなくても実行できます。

一覧表示されたメンバーの中で、生き残っていてログインしたものを選びます (コマンドを実行したメンバーと異なる場合)。

LXD デーモンが実行していないことを確認したうえで次のコマンドを実行します。

```
lxd cluster recover-from-quorum-loss
```

この時点で LXD デーモンを再起動できるようになり、データベースはオンラインに復帰するはずです。

データベースからは何の情報も削除されていないことに注意してください。特に失われたクラスタメンバーに関する情報は、それらのインスタンスについてのメタデータも含めて、まだそこに残っています。この情報は失われたインスタンスを再度作成する必要がある場合に、さらなるリカバーのステップで利用することができます。

失われたクラスタメンバーを恒久的に削除するためには、次のコマンドが利用できます。

```
lxc cluster remove <name> --force
```

ここでは `lxd` ではなく通常の `lxc` コマンドを使う必要があることに注意してください。

アドレスを変更してクラスタメンバーをリカバーする

クラスタの一部のメンバーが到達不可能になった場合や、IP アドレスやリッスンするポート番号の変更によりクラスタ自体が到達不可能になった場合、クラスタは再設定できます。

クラスタの各メンバー上で、 LXD が実行していないときに、以下のコマンドを実行します。

```
lxd cluster edit
```

このセクションの全てのコマンドは `lxc` ではなく `lxd` を使うことに注意してください。

このコマンドはこのノード上で最後に記録されたクラスタの他のノードに関する情報を YAML 形式で表示します。

```
# Latest dqlite segment ID: 1234 # 最新の dqlite のセグメント ID

members:
  - id: 1              # このノードの内部 ID（読み取り専用）
    name: node1        # クラスタメンバーの名前（読み取り専用）
    address: 10.0.0.10:8443 # このノードの最新のアドレス（書き込み可）
    role: voter              # このノードの最新のロール（書き込み可）
  - id: 2
   name: node2
    address: 10.0.0.11:8443
    role: stand-by
  - id: 3
   name: node3
    address: 10.0.0.12:8443
    role: spare
```

この設定からメンバーを削除したり、スタンバイのノードを投票者 (voter) に変えたりは出来ません。それらの変更にはグローバルデータベースが必要になるかもしれないからです。重要なこととして、最低 2 つのノードが投票者 (voter) であること (メンバー数が 2 のクラスタのケースを除いて。メンバー数が 2 のクラスタは 1 つの投票者 (voter) で十分です) を覚えておいてください。そうでないとクォーラムが成立しません。

必要な変更を終えたら、クラスタ内の各メンバー上で同様に変更を行います。各メンバー上で LXD をリロードすると、設定に書かれた全てのノードを含んだ状態でクラスタ全体がオンラインに戻るはずです。

データベースからは何の情報も削除されていないので、クラスタメンバーとインスタンスの全ての情報はデータベースに残っていることに注意してください。

インスタンス

クラスタ上の任意のノード上でインスタンスを起動できます。例えば、node1 から:

```
lxc launch --target node2 ubuntu:18.04 c1
```

のように実行すれば、node2 上で Ubuntu 20.04 コンテナが起動します。

ターゲットを指定せずにインスタンスを起動したときは、インスタンスの数が一番少ないサーバ上でインスタンスが起動されます。全てのサーバが同じ数のインスタンスを持っている場合はランダムに選ばれます。

以下のように実行すると、インスタンス上のすべてのコンテナをリストできます:

```
lxc list
```

NODE 列がコンテナが実行中のノードを示します。

インスタンスが起動後、任意のノードからそのコンテナを操作できます。例えば、node1 から:

```
lxc exec c1 ls /
lxc stop c1
lxc delete c1
lxc pull file c1/etc/hosts .
```

のように操作できます。

Raft メンバーシップの手動での変更

何か予期せぬ出来事が起こった場合など、クラスタの Raft メンバーシップの設定を手動で変更する必要がある状況があるかもしれません。

例えばクリーンに削除できなかったクラスタメンバーがある場合、 `lxc cluster list` に表示されないですが、引き続き Raft 設定の一部になってしまう場合があるかもしれません（この状況は `lxd sql local "SELECT * FROM raft_nodes"` で確認できます）。

この場合は以下のように実行すると

```
lxd cluster remove-raft-node <address>
```

残ってしまったノードを削除できます。

イメージ

デフォルトではデータベースメンバを持っているのと同じ数のクラスタに LXD はイメージを複製します。これは通常はクラスタ内で最大 3 つのコピーを持つことを意味します。

耐障害性とイメージがローカルにある可能性を上げるためにこの数を増やすことができます。

特別な値である "-1" は全てのノードにイメージをコピーするために使用できます。

この数を 1 に設定することでイメージの複製を無効にできます。

```
lxc config set cluster.images_minimal_replica 1
```

ストレージプール

先に述べたように、すべてのノードは同一のストレージプールを持たなければなりません。異なるノード上のプール間の違いは、設定項目、source、size、zfs.pool_name のみです。

新たにストレージプールを作成するためには、すべてのノードでストレージプールをを定義する必要があります。例えば:

```
lxc storage create --target node1 data zfs source=/dev/vdb1
lxc storage create --target node2 data zfs source=/dev/vdc1
```

のようにします。

新しいストレージプールをノード上に定義する際、ノード固有で与えることのできる設定項目は上記設定のみです。

この時点ではプールはまだ実際には作られていませんが、定義はされています（lxc storage list を実行すると、状態が Pending とマークされています)。

次のように実行しましょう:

```
lxc storage create data zfs
```

するとストレージがすべてのノードでインスタンス化されます。特定のノードで定義を行っていない場合、もしくはノードがダウンしている場合は、エラーが返ります。

この最後の storage create コマンドには、ノード固有ではない（上記参照）任意の設定項目を与えることができます。

ストレージボリューム

各ボリュームは特定のノード上に存在しています。`lxc storage volume list` は、特定のボリュームがどのノードにあるかを示す `NODE` 列を表示します。

異なるボリュームは、異なるノード（例えば image volumes）上に存在する限りは同じ名前を持てます。複数のノードが与えた名前のボリュームを持つ場合には、ボリュームコマンドに `--target <node name>` を与える必要がある点を除いて、ストレージボリュームはクラスタ化されていない場合と同じ方法で管理できます。

例えば:

```
# Create a volume on the node this client is pointing at
lxc storage volume create default web

# Create a volume with the same node on another node
lxc storage volume create default web --target node2

# Show the two volumes defined
lxc storage volume show default web --target node1
lxc storage volume show default web --target node2
```

ネットワーク

先に述べたように、すべてのノードは同じネットワークを定義しなければなりません。

異なるノード間のネットワークで異なっても良い設定は、それらのオプショナルの設定項目だけです。特定のクラスタノード上に新しいネットワークを定義する際、設定可能な有効なオプショナルな設定項目は `bridge.external_interfaces` と `parent` だけです。これらは各ノード上で異なる値が設定可能です（それぞれの定義については **ネットワーク設定** の文書を参照してください）。

新しいネットワークを作成するには、最初にすべてのノードで以下のように定義を行う必要があります:

```
lxc network create --target node1 my-network
lxc network create --target node2 my-network
```

この時点では、ネットワークはまだ実際には作成されていません。しかし定義はされています（`lxc network list` を実行すると、状態が Pending とマークされています）。

次のように実行しましょう:

```
lxc network create my-network
```

するとネットワークがすべてのノード上でインスタンス化されます。特定のノードで定義していない場合、もしくはノードがダウンしている場合は、エラーが返ります。

この最後の network create コマンドには、ノード固有ではない（上記参照）任意の設定項目を与えることができます。

分離した REST API とクラスタネットワーク

クライアントの REST API エンドポイントとクラスタ内のノード間の内部的なトラフィック（例えば REST API に DNS ラウンドロビンとともに仮想 IP アドレスを使うために）で別のネットワークを設定できます。

このためには、クラスタの最初のノードを cluster.https_address 設定キーを使ってブートストラップする必要があります。例えば以下の定義ファイルを使うと

```
config:
  core.trust_password: sekret
  core.https_address: my.lxd.cluster:8443
  cluster.https_address: 10.55.60.171:8443
...
```

（YAML 定義ファイルの残りは上記と同じ）。

新しいノードを参加させるには、まず REST API のアドレスを設定します。例えば lxc クライアントを使って以下のように実行し

```
lxc config set core.https_address my.lxd.cluster:8443
```

そして通常通り PUT /1.0/cluster API エンドポイントを使って、server_address フィールドで参加するノードのアドレスを設定します。定義ファイルを使うなら YAML のペイロードは完全に上記のものと同じになるでしょう。

クラスタ証明書の更新

LXD のクラスタ内の全てのサーバは同じ共有された証明書で応答します。これは通常は有効期限が 10 年の標準的な自己署名証明書です。

何か他のもの、例えば Let's Encrypt で取得した有効な証明書、に置き換えたい場合は lxc cluster update-certificate を使ってクラスタ内の全てのサーバの証明書を置き換えることが出来ます。

クラスタグループ

LXD のクラスタでは、メンバーはクラスタグループに追加できます。デフォルトでは全てのメンバーは default グループに属します。

クラスタメンバーは lxc cluster group assign コマンドを使ってグループに割り当てられます。

```
lxc cluster group create gpu
lxc cluster group assign cluster:node1 gpu
```

クラスタグループがあると、個別のメンバーの代わりに特定のグループをターゲットにできます。これは --target を使うときに @ の接頭辞を使うことで実現できます。

例

```
lxc launch images:ubuntu/20.04 cluster:ubuntu --target=@gpu
```

もし scheduler.instance が all (デフォルト) か group に設定されていれば、上のコマンドにより gpu グループに属するクラスタメンバー上にインスタンスが作成されます。

3.4.3 インスタンスのコマンド実行

LXD は与えられたインスタンス内でコマンドを実行することを容易にします。コンテナでは、これは常に動作し、LXD によって直接処理されます。仮想マシンでは、これは仮想マシン内で動作する lxd-agent プロセスに依存します。

CLI レベルでは、これは lxc exec コマンドによって達成されます。実行するコマンドだけでなく、実行モード、ユーザー、グループ、作業を指定することができます。実行モード、ユーザー、グループ、作業ディレクトリの指定をサポートします。

API レベルでは、/1.0/instances/NAME/exec で実現されます。

実行モード

LXD はコマンドを対話的にも非対話的にも実行できます。

インタラクティブモードでは、入力 (stdin) と出力 (stdout, stderr) を扱うために疑似端末装置 (PTS) が使用されます。これは、ターミナル・エミュレータに接続されている場合（スクリプトから実行されていない場合)、CLI によって自動的に選択されます。

非インタラクティブ・モードでは、代わりにパイプが割り当てられ、stdin、stdout、stderr のそれぞれに 1 つずつ割り当てられます。これにより、多くのスクリプトで必要とされるように、コマンドを実行しながら、stdin、stdout、stderr を別々に適切に取得することができます。

ユーザー、グループ、作業ディレクトリ

LXD はインスタンス内のデータを読まない、あるいはその中にあるものを信用しないというポリシーを持っています。これは、LXD がユーザーやグループの解決を処理するために、/etc/passwd、/etc/group や/etc/nsswitch.conf のようなものを解析しないことを意味しています。

結果として、LXD はユーザのホームディレクトリがどこにあるか、あるいはどのような補助的なグループがあるかを知りません。

デフォルトでは、LXD は root (uid 0)、デフォルトのグループ (gid 0) としてコマンドを実行します。作業ディレクトリは/root に設定されています。

ユーザー、グループ、作業ディレクトリはすべて上書きすることができますが、絶対値（uid、gid、パス）は LXD が解決してくれるわけではないので、利用者が指定する必要があります。

環境

exec セッション中に設定される環境変数は、いくつかのソースから得られます。

- インスタンスに直接設定される environment.KEY=VALUE
- exec セッション中に直接渡される環境変数
- LXD によって設定されるデフォルト変数

最後のカテゴリでは、LXD は PATH を/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin に設定し、必要に応じて /snap と /etc/NIXOS で拡張されます。さらに、LANG には C.UTF-8 が設定されます。

root (uid 0) として実行する場合は、以下の変数も設定されます。

- HOME に /root が設定されます。
- USER に root が設定される。

他のユーザーとして実行する場合、正しい値を指定するのはそのユーザーの責任です。

これらのデフォルト値が設定されるのは、インスタンス構成に含まれていなかったり、exec セッションで直接オーバーライドされていない場合のみです。

3.4.4 プロダクション環境のセットアップ

あなたは LXD live online か、なんらかのサーバで LXD を試してみました。結果に満足して、今度は LXD で本格的な作業を試してみたいと思います。

大多数の Linux ディストリビューションでは大量のコンテナを稼働させるのに最適化されたカーネルの設定はされていません。このドキュメントでの指示はコンテナを稼働させる際にひっかかりがちな制限値のほとんどとお勧めの変更後の値をカバーしています。

よく遭遇するエラー

```
Failed to allocate directory watch:  Too many open files

<Error> <Error>:  Too many open files

failed to open stream:  Too many open files in...

neighbour:  ndisc_cache:  neighbor table overflow!
```

サーバの変更

/etc/security/limits.conf

ドメイン	種別	項目	値	デフォルト	説明
*	soft	nofile	1048576	未設定	オープンするファイルの最大数
*	hard	nofile	1048576	未設定	オープンするファイルの最大数
root	soft	nofile	1048576	未設定	オープンするファイルの最大数
root	hard	nofile	1048576	未設定	オープンするファイルの最大数
*	soft	memlock	unlimited	未設定	ロックされたメモリ内の最大のアドレス空間 (KB)
*	hard	memlock	unlimited	未設定	ロックされたメモリ内の最大のアドレス空間 (KB)
root	soft	memlock	unlimited	未設定	ロックされたメモリ内の最大のアドレス空間 (KB) (bpf システムコール監視にのみ必要)
root	hard	memlock	unlimited	未設定	ロックされたメモリ内の最大のアドレス空間 (KB) (bpf システムコール監視にのみ必要)

注意: snap ユーザーの場合はこれらの制限は snap/LXD によって自動的に上げられます。

/etc/sysctl.conf

パラメータ	値	デフォルト	説明
fs.aio-max-nr	524288	65536	これは並行に実行される非同期 I/O 操作の最大数です。 AIO サブシステムを使うワークロードが大量にある場合（例: MySQL ）、これを増やす必要があるかもしれません。
fs.inotify.max_queued_events	1048576	16384	これは対応する inotify のインスタンスにキューイングされるイベント数の上限を指定します。 1
fs.inotify.max_user_instances	1048576	128	これは実ユーザー ID ごとに作成可能な inotify のインスタンス数の上限を指定します。 1
fs.inotify.max_user_watches	1048576	8192	これは実ユーザー ID ごとに作成可能な watch 数の上限を指定します。 1
kernel.dmesg_restrict	1	0	この設定を有効にするとコンテナがカーネルのリングバッファー内のメッセージにアクセスするのを拒否します。この設定はホスト・システム上の非 root ユーザーへのアクセスも拒否することに注意してください。
kernel.keys.maxbytes	2000000	20000	非 root ユーザーが使用できる keyring の最大サイズ
kernel.keys.maxkeys	2000	200	非 root ユーザーが使用できるキーの最大数で、コンテナ数より大きくなければなりません
net.ipv4.neigh.default.gc_thresh3	8192	1024	これは IPv4 ARP テーブル (IPv4) 内のエントリーの最大数です。1024 個を超えるコンテナを作成するなら増やすべきです。増やさなければ ARP テーブルがフルになったときに `neighbour: ndisc_cache: neighbor table overflow!` というエラーが発生し、コンテナがネットワーク設定を取得できなくなります。 2
net.ipv6.neigh.default.gc_thresh3	8192	1024	これは IPv6 ARP テーブル (IPv6) 内のエントリーの最大数です。1024 個を超えるコンテナを作成するなら増やすべきです。増やさなければ ARP テーブルがフルになったときに `neighbour: ndisc_cache: neighbor table overflow!` というエラーが発生し、コンテナがネットワーク設定を取得できなくなります。 2
vm.max_map_count	262144	65530	このファイルはプロセスが持つメモリマップ領域の最大数を含みます。malloc の呼び出しの副作用として、 直接的には mmap と mprotect によって、また、共有ライブラリーをロードすることによって、メモリマップ領域を使います。

設定後、サーバの再起動が必要です。

コンテナ名の漏洩防止

/sys/kernel/slab と /proc/sched_debug はともにシステム上の全ての cgroup の一覧を表示し、拡張を使えばコンテナ一覧を表示するのを容易にします。

一覧が見られるのを防ぐためには、コンテナを開始する前に以下のコマンドを忘れずに実行してください。

- chmod 400 /proc/sched_debug
- chmod 700 /sys/kernel/slab/

ネットワーク帯域の調整

大量の (コンテナ・コンテナ間、あるいはホスト・コンテナ間の) ローカル・アクティビティを持つ LXD ホスト上に 1GbE 以上の NIC をお持ちか、 LXD ホストに 1GbE 以上のインターネット接続をお持ちでしたら、 txqueuelen を調整する価値があります。これらの設定は 10GbE NIC ではさらによく機能します。

サーバの変更

txqueuelen

(あなたにとっての最適な値はわかりませんが) あなたの実 NIC の `txqueuelen` を 10000 に変える必要がある場合、 lxdbr0 インタフェースの `txqueuelen` も 10000 に変更してください。

Debian ベースのディストリビューションでは `/etc/network/interfaces` 内で `txqueuelen` を恒久的に変更できます。例えば `up ip link set eth0 txqueuelen 10000` という設定を加えることで起動時にインタフェースの txqueuelen の値を設定できます。

/etc/sysctl.conf

`net.core.netdev_max_backlog` の値も増やす必要があります。`/etc/sysctl.conf` に `net.core.netdev_max_backlog = 182757` という設定を加えれば (再起動後に) 恒久的に設定できます。(テストの目的で) `netdev_max_backlog` を一時的に設定するには `echo 182757 > /proc/sys/net/core/netdev_max_backlog` と実行します。注意: この値が大きすぎると思うかもしれません。多くの人は `netdev_max_backlog` = `net.ipv4.tcp_mem` の最小値と設定することを好んでいます。例えば私は `net.ipv4.tcp_mem = 182757 243679 365514` という値を使用しています。

コンテナの変更

コンテナ内のイーサネット・インタフェース全ての txqueuelen の値を変更する必要もあります。Debian ベースのディストリビューションでは `/etc/network/interfaces` 内で恒久的に txqueuelen を変更できます。例えば `up ip link set eth0 txqueuelen 10000` という設定を加えることで起動時にインタフェースの txqueuelen の値を設定できます。

この変更についての注意

10000 という txqueuelen の値は 10GbE NIC ではよく使われます。基本的には、 小さな txqueuelen の値は高レイテンシで低速なデバイスと低レイテンシで高速なデバイスで使われます。個人的にはこれらの設定で (ホスト・コンテナ間、コンテナ・コンテナ間の) ローカル通信とインターネット接続が 3〜5% 改善しています。txqueuelen の値の調整の良いところは、使用するコンテナ数が増えれば増えるほど、この調整の恩恵を受けられることです。そして、この値はいつでも一時的に変更することができ、あなたの環境で LXD ホストの再起動無しに変更の結果を確認することができます。

3.4.5 Remotes

イントロダクション

リモートは LXD のコマンドラインクライアント内の概念でありさまざまな LXD サーバやクラスタを参照するのに使用します。リモートは実質的には特定の LXD サーバをサーバにログインしたり認証するのに必要な認証情報も含めて指定する URL を指す名前です。LXD には次の 4 種類のリモートがあります。

- Static
- Default
- Global (システムごと)
- Local (ユーザーごと)

Static

Static リモートは

- local (default)
- ubuntu
- ubuntu-daily

これらはハードコードされておりユーザーが変更できません。

Default

初回の使用時に自動的に追加されます。

Global (システムごと)

デフォルトでは global の設定ファイルは /etc/lxc/config.yml に置かれます。LXD_GLOBAL_CONF 環境変数でパスを変更できます。この設定ファイルを手動で編集して global リモートを追加できます。これらのリモートの証明書は servercerts ディレクトリ (例: /etc/lxc/servercerts/) に置き、リモートの名前にマッチ (例: foo.crt) させます。

設定例を以下に示します。

```
remotes:
  foo:
    addr: https://10.0.2.4:8443
    auth_type: tls
    project: default
    protocol: lxd
    public: false
  bar:
    addr: https://10.0.2.5:8443
    auth_type: tls
    project: default
    protocol: lxd
    public: false
```

Local (ユーザーごと)

local レベルのリモートは CLI (lxc) で次のように管理します。lxc remote [command]

デフォルトでは local の設定ファイルは ~/.config/lxc/config.yml に置かれます。LXD_CONF 環境変数でパスを変更できます。ユーザーはシステムのリモートを (例: lxc remote name や lxc remote set-url を実行することで) オーバーライドすることができます。その場合リモートの設定は関連する証明書と共にコピーされます。

3.4.6 リモート API 認証

LXD デーモンとのリモート通信は、HTTPS 上の JSON を使って行われます。

リモート API にアクセスするためには、クライアントは LXD サーバとの間で認証を行う必要があります。以下の認証方法がサポートされています。

- *TLS* **クライアント証明書**
- *Candid* **ベースの認証**
- **役割ベースのアクセスコントロール** *(RBAC)*

TLS クライアント証明書

認証に TLS クライアント証明書を使用する場合、クライアントとサーバーの両方が最初に起動したときにキーペアを生成します。サーバはそのキーペアを LXD ソケットへの全ての HTTPS 接続に使用します。クライアントは、その証明書をクライアント証明書として、あらゆるクライアント・サーバ間の通信に使用します。

証明書を再生成させるには、単に古いものを削除します。次の接続時には、新しい証明書が生成されます。

通信プロトコル

通信プロトコルは TLS1.2 以上に対応しています。すべての通信には完全な前方秘匿を使用し、暗号は強力な楕円曲線（ECDHE-RSA や ECDHE-ECDSA など）に限定してください。

生成される鍵は最低でも 4096 ビットの RSA、できれば 384 ビットの ECDSA が望ましいです。署名を使用する場合は、SHA-2 署名のみを信頼すべきです。

我々はクライアントとサーバーの両方を管理しているので、壊れたプロトコルや暗号の下位互換をサポートする理由はありません。

信頼できる TLS クライアント

LXD サーバが信頼する TLS 証明書のリストは、`lxc config trust list` で取得できます。

信頼できるクライアントは以下のいずれかの方法で追加できます。

- **信頼できる証明書をサーバーに追加する**
- **トラストパスワードを使ったクライアント証明書の追加**
- **トークンを使ったクライアント証明書の追加**

サーバーとの認証を行うワークフローは、SSH の場合と同様で、未知のサーバーへの初回接続時にプロンプトが表示されます。

1. ユーザーが `lxc remote add` でサーバーを追加すると、HTTPS でサーバーに接続され、その証明書がダウンロードされ、フィンガープリントがユーザーに表示されます。
2. ユーザーは、これが本当にサーバーのフィンガープリントであることを確認するよう求められます。これは、サーバーに接続して手動で確認するか、サーバーにアクセスできる人に info コマンドを実行してフィンガープリントを比較してもらうことで確認できます。
3. サーバーはクライアントの認証を試みます。
   - クライアント証明書がサーバーのトラストストアにある場合は、接続が許可されます。
   - クライアント証明書がサーバーのトラストストアにない場合、サーバーはユーザーにトークンまたはトラストパスワードの入力を求めます。提供されたトークンまたはトラストパスワードが一致した場合、クライアント証明書はサーバーのトラストストアに追加され、接続が許可されます。そうでない場合は、接続が拒否されます。

クライアントへの信頼を取り消すには、`lxc config trust remove FINGERPRINT` でそのクライアント証明書をサーバーから削除します。

TLS クライアントを 1 つまたは複数のプロジェクトに制限することが可能です。この場合、クライアントは、グローバルな構成変更の実行や、アクセスを許可されたプロジェクトの構成（制限、制約）の変更もできなくなります。

アクセスを制限するには、`lxc config trust edit FINGERPRINT` を使用します。restricted **キーを** true`に設定し、クライアントのアクセスを制限するプロジェクトのリストを指定します。プロジェクトのリストが空の場合、クライアントはどのプロジェクトへのアクセスも許可されません。

信頼できる証明書をサーバーに追加する

信頼できるクライアントを追加するには、そのクライアント証明書をサーバーのトラストストアに直接追加するのが望ましい方法です。これを行うには、クライアント証明書をサーバーにコピーし、`lxc config trust add <file>`で登録します。

トラストパスワードを使ったクライアント証明書の追加

クライアント側から新しい信頼関係を確立できるようにするには、サーバーに信頼パスワード（`core.trust_password`、**サーバ設定**参照）を設定する必要があります。クライアントは、プロンプト時にトラストパスワードを入力することで、自分の証明書をサーバのトラストストアに追加することができます。

本番環境では、すべてのクライアントが追加された後に、`core.trust_password` の設定を解除してください。これにより、パスワードを推測しようとするブルートフォース攻撃を防ぐことができます。

トークンを使ったクライアント証明書の追加

トークンを使って新しいクライアントを追加することもできます。トークンは一度使用すると無効になるため、トラストパスワードを使用するよりも安全な方法です。

この方法を使用するには，クライアント名の入力を促す `lxc config trust add` を呼び出して，各クライアント用のトークンを生成します。その後，クライアントは，トラストパスワードの入力を求められたときに生成されたトークンを提供することで，自分の証明書をサーバのトラストストアに追加することができます。あるいは、クライアントはリモートの追加時にトークンを直接提供することもできます。`lxc remote add <name> <token>`.

PKI システムの使用

PKI (Public key infrastructure) の設定では、システム管理者が中央の PKI を管理し、すべての lxc クライアント用のクライアント証明書とすべての LXD デーモン用のサーバー証明書を発行します。

PKI モードを有効にするには、以下の手順を実行します。

1. すべてのマシンに CA（認証局）の証明書を追加します。
   - クライアントの設定ディレクトリ（`~/.config/lxc`）に `client.ca` ファイルを配置する。
   - `server.ca` ファイルをサーバの設定ディレクトリ（`/var/lib/lxd` または snap ユーザの場合は`/var/snap/lxd/common/lxd`）に置く。
2. CA から発行された証明書をクライアントとサーバーに配置し、自動生成された証明書を置き換える。
3. サーバーを再起動します。

このモードでは、LXD デーモンへの接続はすべて、事前に発行された CA 証明書を使って行われます。

もしサーバ証明書が CA によって署名されていなければ、接続は単に通常の認証メカニズムを通過します。サーバ証明書が有効で CA によって署名されていれば、ユーザに証明書を求めるプロンプトを出さずに接続を続行します。

生成された証明書は自動的には信頼されないことに注意してください。そのため、**信頼できる *TLS* クライアント**で説明している方法のいずれかで、サーバーに追加する必要があります。

Candid ベースの認証

LXD が Candid 認証を使用するように設定されている場合、サーバーで認証を行おうとするクライアントは、`candid.api.url` 設定（**サーバ設定**参照）で指定された認証サーバーからディスチャージトークンを取得しなければなりません。

認証サーバーの証明書は、LXD サーバーから信頼されていなければなりません。

Candid/Macaroon 認証を設定した LXD サーバにリモートポインティングを追加するには、`lxc remote add REMOTE ENDPOINT --auth-type=candid` を実行します。ユーザーを確認するために、クライアントは認証サーバーが要求する認証情報の入力を求められます。認証が成功した場合、クライアントは LXD サーバに接続し、認証サーバから受け取ったトークンを提示します。LXD サーバはトークンを検証し、リクエストを認証します。トークンはクッキーとして保存され、クライアントが LXD にリクエストするたびに提示されます。

Candid ベースの認証を設定する方法については、チュートリアルの Candid authentication for LXD を参照してください。

役割ベースのアクセスコントロール (RBAC)

LXD は Canonical の RBAC サービスとの連携をサポートしています。Candid ベースの認証と組み合わせることで、RBAC (Role Based Access Control) は、API クライアントが LXD 上でできることを制限するために使うことができます。

このような設定では、認証は Candid を通して行われ、RBAC サービスはユーザー/グループの関係に役割を維持します。ロールは個々のプロジェクトにも、すべてのプロジェクトにも、あるいは LXD インスタンス全体にも割り当てることができます。

プロジェクトに適用された場合のロールの意味は以下の通りです。

- 監査役: プロジェクトへの読み取り専用のアクセス権
- ユーザー: 通常のライフサイクルアクション（開始、停止、...）を実行する能力。インスタンスでのコマンド実行、コンソールへのアタッチ、スナップショットの管理など。
- オペレーター: 上記のすべての機能に加え、インスタンスやイメージの作成、再設定、削除を行う機能インスタンスとイメージの作成、再設定、削除
- 管理者: 上記の機能に加えて、プロジェクト自体を再構成する機能を持つ

重要: 制限のないプロジェクトでは、`auditor` と `user` のロールだけが、ホストへのルートアクセスを任せられないユーザーに適しています。

また、**制限付きプロジェクト** では、適切に設定されていれば、`operator` ロールも安全に使用することができます。

失敗のシナリオ

以下のようなシナリオでは、認証に失敗することが予想されます。

サーバー証明書の変更

以下のような場合、サーバーの証明書が変更されている可能性があります。

- サーバを完全に再インストールしたため、新しい証明書が発行された。
- 接続が傍受されている（MITM (Man in the middle)）。

このような場合、クライアントは、証明書のフィンガープリントが、このリモート用の設定にあるフィンガープリントと一致しないため、サーバーへの接続を拒否します。

この場合、ユーザーはサーバー管理者に連絡して、証明書が実際に変更されたかどうかを確認する必要があります。証明書が変更された場合は、証明書を新しいものに置き換えるか、リモートを完全に削除して再度追加することができます。

サーバーの信頼関係が取り消された（revoke された）場合

信頼されている他のクライアントや、ローカルのサーバー管理者が、サーバー上のクライアントの信頼エントリを削除すると、そのクライアントに対するサーバーの信頼関係は失効します。

この場合、サーバーは引き続き同じ証明書を使用しますが、すべての API 呼び出しは、クライアントが信頼されていないことを示すエラーである 403 コードを返します。

3.5 REST API

3.5.1 REST API

イントロダクション

LXD とクライアントの間の全ての通信は HTTP 上の RESTful API を使って行います。リモートの操作は SSL で暗号化して通信し、ローカルの操作は Unix ソケットを使って通信します。

API のバージョニング

サポートされている API のメジャーバージョンのリストは GET / を使って取得できます。

後方互換性を壊す場合は API のメジャーバージョンが上がります。

後方互換性を壊さずに追加される機能は api_extensions の追加という形になり、特定の機能がサーバでサポートされているかクライアントがチェックすることで利用できます。

戻り値

次の 3 つの標準的な戻り値の型があります。

- 標準の戻り値
- バックグラウンド操作
- エラー

標準の戻り値

標準の同期的な操作に対しては以下のような dict が返されます。

```
{
    "type": "sync",
    "status": "Success",
    "status_code": 200,
    "metadata": {}                          // リソースやアクションに固有な追加のメタデータ
}
```

HTTP ステータスコードは必ず 200 です。

バックグラウンド操作

リクエストの結果がバックグラウンド操作になる場合、 HTTP ステータスコードは 202 (Accepted) になり、操作の URL を指す HTTP の Location ヘッダが返されます。

レスポンスボディは以下のような構造を持つ dict です。

```
{
    "type": "async",
    "status": "OK",
    "status_code": 100,
    "operation": "/1.0/instances/<id>",                    // バックグラウンド操作の URL
```

(次のページに続く)

(前のページからの続き)

```
    "metadata": {}                                          // 操作のメタデータ (下記参照)
}
```

操作のメタデータの構造は以下のようになります。

```
{
    "id": "a40f5541-5e98-454f-b3b6-8a51ef5dbd3c",           // 操作の UUID
    "class": "websocket",                                   // 操作の種別 (task, websocket,
 token のいずれか)
    "created_at": "2015-11-17T22:32:02.226176091-05:00",    // 操作の作成日時
    "updated_at": "2015-11-17T22:32:02.226176091-05:00",    // 操作の最終更新日時
    "status": "Running",                                    // 文字列表記での操作の状態
    "status_code": 103,                                     // 整数表記での操作の状態␣
↪(status ではなくこちらを利用してください。訳注: 詳しくは下記のステータスコードの項を参照)
    "resources": {                                          // リソース種別 (container,␣
↪snapshots, images のいずれか) の dict を影響を受けるリソース
      "containers": [
        "/1.0/instances/test"
      ]
    },
    "metadata": {                                           // 対象となっている (この例では␣
↪exec) 操作に固有なメタデータ
      "fds": {
        "0": "2a4a97af81529f6608dca31f03a7b7e47acc0b8dc6514496eb25e325f9e4fa6a",
        "control": "5b64c661ef313b423b5317ba9cb6410e40b705806c28255f601c0ef603f079a7"
      }
    },
    "may_cancel": false,                                    // (REST で DELETE を使用して)␣
↪操作がキャンセル可能かどうか
    "err": ""                                               // 操作が失敗した場合にエラー文字
列が設定されます
}
```

対象の操作に対して追加のリクエストを送って情報を取り出さなくても、何が起こっているかユーザーにとってわかりやすい形でボディは構成されています。ボディに含まれる全ての情報はバックグラウンド操作の URL から取得することもできます。

エラー

さまざまな状況によっては操作を行う前に直ぐに問題が起きる場合があり、そういう場合には以下のような値が返されます。

```
{
    "type": "error",
    "error": "Failure",
    "error_code": 400,
    "metadata": {}                          // エラーについてのさらなる詳細
}
```

HTTP ステータスコードは 400, 401, 403, 404, 409, 412, 500 のいずれかです。

ステータスコード

LXD REST API はステータス情報を返す必要があります。それはエラーの理由だったり、操作の現在の状態だったり、 LXD が提供する様々なリソースの状態だったりします。

デバッグをシンプルにするため、ステータスは常に文字列表記と整数表記で重複して返されます。ステータスの整数表記の値は将来に渡って不変なので API クライアントが個々の値に依存できます。文字列表記のステータスは人間が API を手動で実行したときに何が起きているかをより簡単に判断できるように用意されています。

ほとんどのケースでこれらは `status` と `status_code` と呼ばれ、前者はユーザーフレンドリーな文字列表記で後者は固定の数値です。

整数表記のコードは常に 3 桁の数字で以下の範囲の値となっています。

- 100 to 199: リソースの状態 (started, stopped, ready, ...)
- 200 to 399: 成功したアクションの結果
- 400 to 599: 失敗したアクションの結果
- 600 to 999: 将来使用するために予約されている番号の範囲

現在使用されているステータスコード一覧

コード	意味
100	操作が作成された
101	開始された
102	停止された
103	実行中
104	キャンセル中
105	ペンディング
106	開始中
107	停止中
108	中断中
109	凍結中
110	凍結された
111	解凍された
112	エラー
200	成功
400	失敗
401	キャンセルされた

再帰

巨大な一覧のクエリを最適化するために、コレクションに対して再帰が実装されています。コレクションに対するクエリの GET リクエストに `recursion` パラメータを指定できます。

デフォルト値は 0 でコレクションのメンバーの URL が返されることを意味します。1 を指定するとこれらの URL がそれが指すオブジェクト (通常は dict 形式) で置き換えられます。

再帰はジョブへのポインタ (URL) をオブジェクトそのもので単に置き換えるように実装されています。

フィルタ

検索結果をある値でフィルタするために、コレクションにフィルタが実装されています。コレクションに対する GET クエリに `filter` 引数を渡せます。

フィルタはインスタンス、イメージ、ストレージボリュームのエンドポイントに提供されています。

フィルタにはデフォルト値はありません。これは見つかった全ての結果が返されることを意味します。フィルタの引数には以下のような言語を設定します。

?filter=field_name eq desired_field_assignment

この言語は REST API のフィルタロジックを構成するための OData の慣習に従います。フィルタは下記の論理演算子もサポートします。not(not), equals(eq), not equals(ne), and(and), or(or) フィルタは左結合で評価されます。空白を含む値はクォートで囲むことができます。ネストしたフィルタもサポートされます。例えば config 内のフィールドに対してフィルタするには以下のように指定します。

?filter=config.field_name eq desired_field_assignment

device の属性についてフィルタするには以下のように指定します。

?filter=devices.device_name.field_name eq desired_field_assignment

以下に上記の異なるフィルタの方法を含む GET クエリをいくつか示します。

containers?filter=name eq "my container" and status eq Running

containers?filter=config.image.os eq ubuntu or devices.eth0.nictype eq bridged

images?filter=Properties.os eq Centos and not UpdateSource.Protocol eq simplestreams

非同期操作

完了までに 1 秒以上かかるかもしれない操作はバックグラウンドで実行しなければなりません。そしてクライアントにはバックグラウンド操作 ID を返します。

クライアントは操作のステータス更新をポーリングするか long-poll API を使って通知を待つことが出来ます。

通知

通知のために Websocket ベースの API が利用できます。クライアントへ送られるトラフィックを制限するためにいくつかの異なる通知種別が存在します。

リモート操作の状態をポーリングしなくて済むように、リモート操作を開始する前に操作の通知をクライアントが常に購読しておくのがお勧めです。

PUT と PATCH の使い分け

LXD API は既存のオブジェクトを変更するのに PUT と PATCH の両方をサポートします。

PUT はオブジェクト全体を新しい定義で置き換えます。典型的には GET で現在のオブジェクトの状態を取得した後に PUT が呼ばれます。

レースコンディションを避けるため、 GET のレスポンスから ETag ヘッダを読み取り PUT リクエストの If-Match ヘッダに設定するべきです。こうしておけば GET と PUT の間にオブジェクトが他から変更されていた場合は更新が失敗するようになります。

PATCH は変更したいプロパティだけを指定することでオブジェクト内の単一のフィールドを変更するのに用いられます。キーを削除するには通常は空の値を設定すれば良いようになっていますが、 PATCH ではキーの削除は出来ず、代わりに PUT を使う必要がある場合もあります。

インスタンス、コンテナと仮想マシン

このドキュメントでは `/1.0/instances/...` のようなパスを常に示します。これらはかなり新しく、仮想マシンがサポートされた LXD 3.19 で導入されました。

コンテナのみをサポートする古いリリースでは全く同じ API を `/1.0/containers/...` で利用します。

後方互換性の理由で LXD は `/1.0/containers` API を引き続き公開しサポートしますが、簡潔さのため以下では両方をドキュメントはしないことにしました。

`/1.0/virtual-machines` に追加のエンドポイントも存在し、 `/1.0/containers` とほぼ同様ですが、仮想マシンのタイプのインスタンスのみを表示します。

API 構造

LXD は API エンドポイントを記述する Swagger 仕様を自動生成しています。この API 仕様の YAML 版が rest-api.yaml にあります。手軽にウェブで見る場合は https://linuxcontainers.org/lxd/api/master/ を参照してください。

3.5.2 API 仕様

3.5.3 API 拡張

それらの変更は全て後方互換であり、 `GET /1.0/` の `api_extensions` を見ることでクライアントツールにより検出可能です。

storage_zfs_remove_snapshots

`storage.zfs_remove_snapshots` というデーモン設定キーが導入されました。

値の型は boolean でデフォルトは false です。 true にセットすると、スナップショットを復元しようとするときに必要なスナップショットを全て削除するように LXD に指示します。

ZFS でスナップショットの復元が出来るのは最新のスナップショットに限られるので、この対応が必要になります。

container_host_shutdown_timeout

`boot.host_shutdown_timeout` というコンテナ設定キーが導入されました。

値の型は integer でコンテナを停止しようとした後 kill するまでどれだけ待つかを LXD に指示します。

この値は LXD デーモンのクリーンなシャットダウンのときにのみ使用されます。デフォルトは 30s です。

container_stop_priority

`boot.stop.priority` というコンテナ設定キーが導入されました。

値の型は integer でシャットダウン時のコンテナの優先度を指示します。

コンテナは優先度レベルの高いものからシャットダウンを開始します。

同じ優先度のコンテナは並列にシャットダウンします。デフォルトは 0 です。

container_syscall_filtering

コンテナ設定キーに関するいくつかの新しい syscall が導入されました。

- `security.syscalls.blacklist_default`
- `security.syscalls.blacklist_compat`
- `security.syscalls.blacklist`
- `security.syscalls.whitelist`

使い方は **インスタンスの設定** を参照してください。

auth_pki

これは PKI 認証モードのサポートを指示します。

このモードではクライアントとサーバは同じ PKI によって発行された証明書を使わなければなりません。

詳細は security.md を参照してください。

container_last_used_at

GET /1.0/containers/<name> エンドポイントに last_used_at フィールドが追加されました。

これはコンテナが開始した最新の時刻のタイムスタンプです。

コンテナが作成されたが開始はされていない場合は last_used_at フィールドは 1970-01-01T00:00:00Z になります。

etag

関連性のある全てのエンドポイントに ETag ヘッダのサポートが追加されました。

この変更により GET のレスポンスに次の HTTP ヘッダが追加されます。

- ETag (ユーザーが変更可能なコンテンツの SHA-256)

また PUT リクエストに次の HTTP ヘッダのサポートが追加されます。

- If-Match (前回の GET で得られた ETag の値を指定)

これにより GET で LXD のオブジェクトを取得して PUT で変更する際に、レースコンディションになったり、途中で別のクライアントがオブジェクトを変更していた (訳注: のを上書きしてしまう) というリスク無しに PUT で変更できるようになります。

patch

HTTP の PATCH メソッドのサポートを追加します。

PUT の代わりに PATCH を使うとオブジェクトの部分的な変更が出来ます。

usb_devices

USB ホットプラグのサポートを追加します。

https_allowed_credentials

LXD API を全てのウェブブラウザで (SPA 経由で) 使用するには、 XHR の度に認証情報を送る必要があります。それぞれの XHR リクエストで"withCredentials=true" とセットします。

Firefox や Safari などいくつかのブラウザは Access-Control-Allow-Credentials: true ヘッダがないレスポンスを受け入れることができません。サーバがこのヘッダ付きのレスポンスを返すことを保証するには core.https_allowed_credentials=true と設定してください。

image_compression_algorithm

この変更はイメージを作成する時 (`POST /1.0/images`) に `compression_algorithm` というプロパティのサポートを追加します。

このプロパティを設定するとサーバのデフォルト値 (`images.compression_algorithm`) をオーバーライドします。

directory_manipulation

LXD API 経由でディレクトリを作成したり一覧したりでき、ファイルタイプを X-LXD-type ヘッダに付与するようになります。現状はファイルタイプは "file" か "directory" のいずれかです。

container_cpu_time

この拡張により実行中のコンテナの CPU 時間を取得できます。

storage_zfs_use_refquota

この拡張により新しいサーバプロパティ `storage.zfs_use_refquota` が追加されます。これはコンテナにサイズ制限を設定する際に "quota" の代わりに "refquota" を設定するように LXD に指示します。また LXD はディスク使用量を調べる際に "used" の代わりに"usedbydataset" を使うようになります。

これはスナップショットによるディスク消費をコンテナのディスク利用の一部とみなすかどうかを実質的に切り替えることになります。

storage_lvm_mount_options

この拡張は `storage.lvm_mount_options` という新しいデーモン設定オプションを追加します。デフォルト値は "discard" で、このオプションにより LVM LV で使用するファイルシステムの追加マウントオプションをユーザーが指定できるようになります。

network

LXD のネットワーク管理 API 。

次のものを含みます。

- `/1.0/networks` エントリーに "managed" プロパティを追加
- ネットワーク設定オプションの全て (詳細は **ネットワーク設定** を参照)
- `POST /1.0/networks` (詳細は *RESTful API* を参照)
- `PUT /1.0/networks/<entry>` (詳細は *RESTful API* を参照)

- `PATCH /1.0/networks/<entry>` (詳細は *RESTful API* を参照)
- `DELETE /1.0/networks/<entry>` (詳細は *RESTful API* を参照)
- "nic" タイプのデバイスの `ipv4.address` プロパティ (nictype が "bridged" の場合)
- "nic" タイプのデバイスの `ipv6.address` プロパティ (nictype が "bridged" の場合)
- "nic" タイプのデバイスの `security.mac_filtering` プロパティ (nictype が "bridged" の場合)

profile_usedby

プロファイルを使用しているコンテナをプロファイルエントリーの一覧の used_by フィールドとして新たに追加します。

container_push

コンテナが push モードで作成される時、クライアントは作成元と作成先のサーバ間のプロキシとして機能します。作成先のサーバが NAT やファイアウォールの後ろにいて作成元のサーバと直接通信できず pull モードで作成できないときにこれは便利です。

container_exec_recording

新しい boolean 型の "record-output" を導入します。これは `/1.0/containers/<name>/exec` のパラメータでこれを "true" に設定し "wait-for-websocket" を fales に設定すると標準出力と標準エラー出力をディスクに保存し logs インタフェース経由で利用可能にします。

記録された出力の URL はコマンドが実行完了したら操作のメタデータに含まれます。

出力は他のログファイルと同様に、通常は 48 時間後に期限切れになります。

certificate_update

REST API に次のものを追加します。

- 証明書の GET に ETag ヘッダ
- 証明書エントリーの PUT
- 証明書エントリーの PATCH

container_exec_signal_handling

クライアントに送られたシグナルをコンテナ内で実行中のプロセスにフォワーディングするサポートを `/1.0/containers/<name>/exec` に追加します。現状では SIGTERM と SIGHUP がフォワードされます。フォワード出来るシグナルは今後さらに追加されるかもしれません。

gpu_devices

コンテナに GPU を追加できるようにします。

container_image_properties

設定キー空間に新しく `image` を導入します。これは読み取り専用で、親のイメージのプロパティを含みます。

migration_progress

転送の進捗が操作の一部として送信側と受信側の両方に公開されます。これは操作のメタデータの"fs_progress" 属性として現れます。

id_map

`security.idmap.isolated`, `security.idmap.isolated`, `security.idmap.size`, `raw.id_map` のフィールドを設定できるようにします。

network_firewall_filtering

`ipv4.firewall` と `ipv6.firewall` という 2 つのキーを追加します。false に設置すると iptables の FORWARDING ルールの生成をしないようになります。 NAT ルールは対応する `ipv4.nat` や `ipv6.nat` キーが true に設定されている限り引き続き追加されます。

ブリッジに対して dnsmasq が有効な場合、 dnsmasq が機能する (DHCP/DNS) ために必要なルールは常に適用されます。

network_routes

`ipv4.routes` と `ipv6.routes` を導入します。これらは LXD ブリッジに追加のサブネットをルーティングできるようにします。

storage

LXD のストレージ管理 API 。

これは次のものを含みます。

- GET /1.0/storage-pools
- POST /1.0/storage-pools (詳細は *RESTful API* を参照)
- GET /1.0/storage-pools/<name> (詳細は *RESTful API* を参照)
- POST /1.0/storage-pools/<name> (詳細は *RESTful API* を参照)
- PUT /1.0/storage-pools/<name> (詳細は *RESTful API* を参照)
- PATCH /1.0/storage-pools/<name> (詳細は *RESTful API* を参照)
- DELETE /1.0/storage-pools/<name> (詳細は *RESTful API* を参照)
- GET /1.0/storage-pools/<name>/volumes (詳細は *RESTful API* を参照)
- GET /1.0/storage-pools/<name>/volumes/<volume_type> (詳細は *RESTful API* を参照)
- POST /1.0/storage-pools/<name>/volumes/<volume_type> (詳細は *RESTful API* を参照)
- GET /1.0/storage-pools/<pool>/volumes/<volume_type>/<name> (詳細は *RESTful API* を参照)
- POST /1.0/storage-pools/<pool>/volumes/<volume_type>/<name> (詳細は *RESTful API* を参照)
- PUT /1.0/storage-pools/<pool>/volumes/<volume_type>/<name> (詳細は *RESTful API* を参照)
- PATCH /1.0/storage-pools/<pool>/volumes/<volume_type>/<name> (詳細は *RESTful API* を参照)
- DELETE /1.0/storage-pools/<pool>/volumes/<volume_type>/<name> (詳細は *RESTful API* を参照)
- 全てのストレージ設定オプション (詳細は **ストレージの設定** を参照)

file_delete

/1.0/containers/<name>/files の DELETE メソッドを実装

file_append

X-LXD-write ヘッダを実装しました。値は overwrite か append のいずれかです。

network_dhcp_expiry

ipv4.dhcp.expiry と ipv6.dhcp.expiry を導入します。 DHCP のリース期限を設定できるようにします。

storage_lvm_vg_rename

storage.lvm.vg_name を設定することでボリュームグループをリネームできるようにします。

storage_lvm_thinpool_rename

storage.thinpool_name を設定することで thinpool をリネームできるようにします。

network_vlan

macvlan ネットワークデバイスに vlan プロパティを新たに追加します。

これを設定すると、指定した VLAN にアタッチするように LXD に指示します。LXD はホスト上でその VLAN を持つ既存のインタフェースを探します。もし見つからない場合は LXD がインタフェースを作成して macvlan の親として使用します。

image_create_aliases

POST /1.0/images に aliases フィールドを新たに追加します。イメージの作成／インポート時にエイリアスを設定できるようになります。

container_stateless_copy

POST /1.0/containers/<name> に live という属性を新たに導入します。false に設定すると、実行状態を転送しようとしないように LXD に伝えます。

container_only_migration

container_only という boolean 型の属性を導入します。 true に設定するとコンテナだけがコピーや移動されるようになります。

storage_zfs_clone_copy

ZFS ストレージプールに storage_zfs_clone_copy という boolean 型のプロパティを導入します。false に設定すると、コンテナのコピーは zfs send と receive 経由で行われるようになります。これにより作成先のコンテナは作成元のコンテナに依存しないようになり、 ZFS プールに依存するスナップショットを維持する必要がなくなります。しかし、これは影響するプールのストレージの使用状況が以前より非効率的になるという結果を伴います。このプロパティのデフォルト値は true です。つまり明示的に "false" に設定しない限り、空間効率の良いスナップショットが使われます。

unix_device_rename

path を設定することによりコンテナ内部で unix-block/unix-char デバイスをリネームできるようにし、ホスト上のデバイスを指定する source 属性が追加されます。path を設定せずに source を設定すると、 path は source と同じものとして扱います。 source や major/minor を設定せずに path を設定すると source は path と同じものとして扱います。ですので、最低どちらか 1 つは設定しなければなりません。

storage_rsync_bwlimit

ストレージエンティティを転送するために rsync が起動される場合に rsync.bwlimit を設定すると使用できるソケット I/O の量に上限を設定します。

network_vxlan_interface

ネットワークに tunnel.NAME.interface オプションを新たに導入します。

このキーは VXLAN トンネルにホストのどのネットワークインタフェースを使うかを制御します。

storage_btrfs_mount_options

btrfs ストレージプールに `btrfs.mount_options` プロパティを導入します。

このキーは btrfs ストレージプールに使われるマウントオプションを制御します。

entity_description

これはエンティティにコンテナ、スナップショット、ストレージプール、ボリュームのような説明を追加します。

image_force_refresh

既存のイメージを強制的にリフレッシュできます。

storage_lvm_lv_resizing

これはコンテナの root ディスクデバイス内に `size` プロパティを設定することで論理ボリュームをリサイズできるようにします。

id_map_base

これは `security.idmap.base` を新しく導入します。これにより分離されたコンテナに map auto-selection するプロセスをスキップし、ホストのどの uid/gid をベースとして使うかをユーザーが指定できるようにします。

file_symlinks

これは file API 経由でシンボリックリンクを転送するサポートを追加します。X-LXD-type に "symlink" を指定できるようになり、リクエストの内容はターゲットのパスを指定します。

container_push_target

`POST /1.0/containers/<name>` に `target` フィールドを新たに追加します。これはマイグレーション中に作成元の LXD ホストが作成先に接続するために利用可能です。

network_vlan_physical

`physical` ネットワークデバイスで `vlan` プロパティが使用できるようにします。

設定すると、 `parent` インタフェース上で指定された VLAN にアタッチするように LXD に指示します。 LXD はホスト上でその `parent` と VLAN を既存のインタフェースで探します。見つからない場合は作成します。その後コンテナにこのインタフェースを直接アタッチします。

storage_images_delete

これは指定したストレージプールからイメージのストレージボリュームをストレージ API で削除できるようにします。

container_edit_metadata

これはコンテナの metadata.yaml と関連するテンプレートを`/1.0/containers/<name>/metadata` 配下の URL にアクセスすることにより API で編集できるようにします。コンテナからイメージを発行する前にコンテナを編集できるようになります。

container_snapshot_stateful_migration

これは stateful なコンテナのスナップショットを新しいコンテナにマイグレートできるようにします。

storage_driver_ceph

これは ceph ストレージドライバを追加します。

storage_ceph_user_name

これは ceph ユーザーを指定できるようにします。

instance_types

これはコンテナの作成リクエストに `instance_type` フィールドを追加します。値は LXD のリソース制限に展開されます。

storage_volatile_initial_source

これはストレージプール作成中に LXD に渡された実際の作成元を記録します。

storage_ceph_force_osd_reuse

これは ceph ストレージドライバに `ceph.osd.force_reuse` プロパティを導入します。 `true` に設定すると LXD は別の LXD インスタンスで既に使用中の osd ストレージプールを再利用するようになります。

storage_block_filesystem_btrfs

これは ext4 と xfs に加えて btrfs をストレージボリュームファイルシステムとしてサポートするようになります。

resources

これは LXD が利用可能なシステムリソースを LXD デーモンに問い合わせできるようにします。

kernel_limits

これは `nofile` でコンテナがオープンできるファイルの最大数といったプロセスのリミットを設定できるようにします。形式は `limits.kernel.[`**リミット名**`]` です。

storage_api_volume_rename

これはカスタムストレージボリュームをリネームできるようにします。

external_authentication

これは Macaroons での外部認証をできるようにします。

network_sriov

これは SR-IOV を有効にしたネットワークデバイスのサポートを追加します。

console

これはコンテナのコンソールデバイスとコンソールログを利用可能にします。

restrict_devlxd

security.devlxd コンテナ設定キーを新たに導入します。このキーは /dev/lxd インタフェースがコンテナで利用可能になるかを制御します。false に設定すると、コンテナが LXD デーモンと連携するのを実質無効にすることになります。

migration_pre_copy

これはライブマイグレーション中に最適化されたメモリ転送をできるようにします。

infiniband

これは infiniband ネットワークデバイスを使用できるようにします。

maas_network

これは MAAS ネットワーク統合をできるようにします。

デーモンレベルで設定すると、 "nic" デバイスを特定の MAAS サブネットにアタッチできるようになります。

devlxd_events

これは devlxd ソケットに websocket API を追加します。

devlxd ソケット上で /1.0/events に接続すると、 websocket 上でイベントのストリームを受け取れるようになります。

proxy

これはコンテナに proxy という新しいデバイスタイプを追加します。これによりホストとコンテナ間で接続をフォワーディングできるようになります。

network_dhcp_gateway

代替のゲートウェイを設定するための ipv4.dhcp.gateway ネットワーク設定キーを新たに追加します。

file_get_symlink

これは file API を使ってシンボリックリンクを取得できるようにします。

network_leases

/1.0/networks/NAME/leases API エンドポイントを追加します。 LXD が管理する DHCP サーバが稼働するブリッジ上のリースデータベースに問い合わせできるようになります。

unix_device_hotplug

これは unix デバイスに "required" プロパティのサポートを追加します。

storage_api_local_volume_handling

これはカスタムストレージボリュームを同じあるいは異なるストレージプール間でコピーしたり移動したりできるようにします。

operation_description

全ての操作に "description" フィールドを追加します。

clustering

LXD のクラスタリング API 。

これは次の新しいエンドポイントを含みます (詳細は *RESTful API* を参照)。

- `GET /1.0/cluster`
- `UPDATE /1.0/cluster`
- `GET /1.0/cluster/members`
- `GET /1.0/cluster/members/<name>`

- POST /1.0/cluster/members/<name>
- DELETE /1.0/cluster/members/<name>

次の既存のエンドポイントは以下のように変更されます。

- POST /1.0/containers 新しい target クエリパラメータを受け付けるようになります。
- POST /1.0/storage-pools 新しい target クエリパラメータを受け付けるようになります
- GET /1.0/storage-pool/<name> 新しい target クエリパラメータを受け付けるようになります
- POST /1.0/storage-pool/<pool>/volumes/<type> 新しい target クエリパラメータを受け付けるようになります
- GET /1.0/storage-pool/<pool>/volumes/<type>/<name> 新しい target クエリパラメータを受け付けるようになります
- POST /1.0/storage-pool/<pool>/volumes/<type>/<name> 新しい target クエリパラメータを受け付けるようになります
- PUT /1.0/storage-pool/<pool>/volumes/<type>/<name> 新しい target クエリパラメータを受け付けるようになります
- PATCH /1.0/storage-pool/<pool>/volumes/<type>/<name> 新しい target クエリパラメータを受け付けるようになります
- DELETE /1.0/storage-pool/<pool>/volumes/<type>/<name> 新しい target クエリパラメータを受け付けるようになります
- POST /1.0/networks 新しい target クエリパラメータを受け付けるようになります
- GET /1.0/networks/<name> 新しい target クエリパラメータを受け付けるようになります

event_lifecycle

これはイベント API に lifecycle メッセージ種別を新たに追加します。

storage_api_remote_volume_handling

これはリモート間でカスタムストレージボリュームをコピーや移動できるようにします。

nvidia_runtime

コンテナに nvidia_runtime という設定オプションを追加します。これを true に設定すると NVIDIA ランタイムと CUDA ライブラリーがコンテナに渡されます。

container_mount_propagation

これはディスクデバイスタイプに "propagation" オプションを新たに追加します。これによりカーネルのマウントプロパゲーションの設定ができるようになります。

container_backup

コンテナのバックアップサポートを追加します。

これは次のエンドポイントを新たに追加します (詳細は *RESTful API* を参照)。

- GET /1.0/containers/<name>/backups
- POST /1.0/containers/<name>/backups
- GET /1.0/containers/<name>/backups/<name>
- POST /1.0/containers/<name>/backups/<name>
- DELETE /1.0/containers/<name>/backups/<name>
- GET /1.0/containers/<name>/backups/<name>/export

次の既存のエンドポイントは以下のように変更されます。

- POST /1.0/containers 新たな作成元の種別 backup を受け付けるようになります

devlxd_images

コンテナに security.devlxd.images 設定オプションを追加します。これにより devlxd 上で /1.0/images/FINGERPRINT/export API が利用可能になります。 nested LXD を動かすコンテナがホストから生のイメージを取得するためにこれは利用できます。

container_local_cross_pool_handling

これは同じ LXD インスタンス上のストレージプール間でコンテナをコピー・移動できるようにします。

proxy_unix

proxy デバイスで unix ソケットと abstract unix ソケットの両方のサポートを追加します。これらは `unix:/path/to/unix.sock` (通常のソケット) あるいは `unix:@/tmp/unix.sock` (abstract ソケット) のようにアドレスを指定して利用可能です。

現状サポートされている接続は次のとおりです。

- `TCP <-> TCP`
- `UNIX <-> UNIX`
- `TCP <-> UNIX`
- `UNIX <-> TCP`

proxy_udp

proxy デバイスで udp のサポートを追加します。

現状サポートされている接続は次のとおりです。

- `TCP <-> TCP`
- `UNIX <-> UNIX`
- `TCP <-> UNIX`
- `UNIX <-> TCP`
- `UDP <-> UDP`
- `TCP <-> UDP`
- `UNIX <-> UDP`

clustering_join

これにより GET /1.0/cluster がノードに参加する際にどのようなストレージプールとネットワークを作成する必要があるかについての情報を返します。また、それらを作成する際にどのノード固有の設定キーを使う必要があるかについての情報も返します。同様に PUT /1.0/cluster エンドポイントも同じ形式でストレージプールとネットワークについての情報を受け付け、クラスタに参加する前にこれらが自動的に作成されるようになります。

proxy_tcp_udp_multi_port_handling

複数のポートにトラフィックをフォワーディングできるようにします。フォワーディングはポートの範囲が転送元と転送先で同じ (例えば `1.2.3.4 0-1000 -> 5.6.7.8 1000-2000`) 場合か転送元で範囲を指定し転送先で単一のポートを指定する (例えば `1.2.3.4 0-1000 -> 5.6.7.8 1000`) 場合に可能です。

network_state

ネットワークの状態を取得できるようになります。

これは次のエンドポイントを新たに追加します (詳細は *RESTful API* を参照)。

- `GET /1.0/networks/<name>/state`

proxy_unix_dac_properties

これは抽象的 unix ソケットではない unix ソケットに gid, uid, パーミションのプロパティを追加します。

container_protection_delete

`security.protection.delete` フィールドを設定できるようにします。 true に設定するとコンテナが削除されるのを防ぎます。スナップショットはこの設定により影響を受けません。

proxy_priv_drop

proxy デバイスに security.uid と security.gid を追加します。これは root 権限を落とし (訳注: 非 root 権限で動作させるという意味です)、 Unix ソケットに接続する際に用いられる uid/gid も変更します。

pprof_http

これはデバッグ用の HTTP サーバを起動するために、新たに core.debug_address オプションを追加します。

このサーバは現在 pprof API を含んでおり、従来の cpu-profile, memory-profile と print-goroutines デバッグオプションを置き換えるものです。

proxy_haproxy_protocol

proxy デバイスに proxy_protocol キーを追加します。これは HAProxy PROXY プロトコルヘッダの使用を制御します。

network_hwaddr

ブリッジの MAC アドレスを制御する bridge.hwaddr キーを追加します。

proxy_nat

これは最適化された UDP/TCP プロキシを追加します。設定上可能であればプロキシ処理は proxy デバイスの代わりに iptables 経由で行われるようになります。

network_nat_order

LXD ブリッジに `ipv4.nat.order` と `ipv6.nat.order` 設定キーを導入します。これらのキーは LXD のルールをチェイン内の既存のルールの前に置くか後に置くかを制御します。

container_full

これは `GET /1.0/containers` に recursion=2 という新しいモードを導入します。これにより状態、スナップショットとバックアップの構造を含むコンテナの全ての構造を取得できるようになります。

この結果 "lxc list" は必要な全ての情報を 1 つのクエリで取得できるようになります。

candid_authentication

これは新たに candid.api.url 設定キーを導入し core.macaroon.endpoint を削除します。

backup_compression

これは新たに `backups.compression_algorithm` 設定キーを導入します。これによりバックアップの圧縮の設定が可能になります。

candid_config

これは `candid.domains` と `candid.expiry` 設定キーを導入します。前者は許可された／有効な Candid ドメインを指定することを可能にし、後者は macaroon の有効期限を設定可能にします。 `lxc remote add` コマンドに新たに `--domain` フラグが追加され、これにより Candid ドメインを指定可能になります。

nvidia_runtime_config

これは nvidia.runtime と libnvidia-container ライブラリーを使用する際に追加のいくつかの設定キーを導入します。これらのキーは nvidia-container の対応する環境変数にほぼそのまま置き換えられます。

- nvidia.driver.capabilities => NVIDIA_DRIVER_CAPABILITIES
- nvidia.require.cuda => NVIDIA_REQUIRE_CUDA
- nvidia.require.driver => NVIDIA_REQUIRE_DRIVER

storage_api_volume_snapshots

ストレージボリュームスナップショットのサポートを追加します。これらはコンテナスナップショットのように振る舞いますが、ボリュームに対してのみ作成できます。

これにより次の新しいエンドポイントが追加されます (詳細は *RESTful API* を参照)。

- `GET /1.0/storage-pools/<pool>/volumes/<type>/<name>/snapshots`
- `POST /1.0/storage-pools/<pool>/volumes/<type>/<name>/snapshots`
- `GET /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `PUT /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `POST /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `DELETE /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`

storage_unmapped

ストレージボリュームに新たに `security.unmapped` という設定を導入します。

true に設定するとボリューム上の現在のマップをフラッシュし、以降の idmap のトラッキングとボリューム上のリマッピングを防ぎます。

これは隔離されたコンテナ間でデータを共有するために使用できます。この際コンテナを書き込みアクセスを要求するコンテナにアタッチした後にデータを共有します。

projects

新たに project API を追加します。プロジェクトの作成、更新、削除ができます。

現時点では、プロジェクトは、コンテナ、プロファイル、イメージを保持できます。そして、プロジェクトを切り替えることで、独立した LXD リソースのビューを見せられます。

candid_config_key

新たに `candid.api.key` オプションが使えるようになります。これにより、エンドポイントが期待する公開鍵を設定でき、HTTP のみの Candid サーバを安全に利用できます。

network_vxlan_ttl

新たにネットワークの設定に `tunnel.NAME.ttl` が指定できるようになります。これにより、VXLAN トンネルの TTL を増加させることができます。

container_incremental_copy

新たにコンテナのインクリメンタルコピーができるようになります。`--refresh` オプションを指定してコンテナをコピーすると、見つからないファイルや、更新されたファイルのみをコピーします。コンテナが存在しない場合は、通常のコピーを実行します。

usb_optional_vendorid

名前が暗示しているように、コンテナにアタッチされた USB デバイスの `vendorid` フィールドが省略可能になります。これにより全ての USB デバイスがコンテナに渡されます (GPU に対してなされたのと同様)。

snapshot_scheduling

これはスナップショットのスケジューリングのサポートを追加します。これにより 3 つの新しい設定キーが導入されます。 snapshots.schedule, snapshots.schedule.stopped, そして snapshots.pattern です。スナップショットは最短で 1 分間隔で自動的に作成されます。

snapshots_schedule_aliases

スナップショットのスケジュールはスケジュールエイリアスのカンマ区切りリストで設定できます。インスタンスには <@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly> <@startup>、ストレージボリュームには <@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly> のエイリアスが利用できます。

container_copy_project

コピー元のコンテナの dict に project フィールドを導入します。これによりプロジェクト間でコンテナをコピーあるいは移動できるようになります。

clustering_server_address

これはサーバのネットワークアドレスを REST API のクライアントネットワークアドレスと異なる値に設定することのサポートを追加します。クライアントは新しい cluster.https_address 設定キーを初期のサーバのアドレスを指定するためにに設定できます。新しいサーバが参加する際、クライアントは参加するサーバの core.https_address 設定キーを参加するサーバがリッスンすべきアドレスに設定でき、PUT /1.0/cluster API の server_address キーを参加するサーバがクラスタリングトラフィックに使用すべきアドレスに設定できます (server_address の値は自動的に参加するサーバの cluster.https_address 設定キーにコピーされます)。

clustering_image_replication

クラスタ内のノードをまたいだイメージのレプリケーションを可能にします。新しい cluster.images_minimal_replica 設定キーが導入され、イメージのリプリケーションに対するノードの最小数を指定するのに使用できます。

container_protection_shift

security.protection.shift の設定を可能にします。これによりコンテナのファイルシステム上で uid/gid をシフト (再マッピング) させることを防ぎます。

snapshot_expiry

これはスナップショットの有効期限のサポートを追加します。タスクは 1 分おきに実行されます。`snapshots.expiry` 設定オプションは、`1M 2H 3d 4w 5m 6y`（それぞれ 1 分、2 時間、3 日、4 週間、5 ヶ月、6 年）といった形式を取ります。この指定ではすべての部分を使う必要はありません。

作成されるスナップショットには、指定した式に基づいて有効期限が設定されます。`expires_at` で定義される有効期限は、API や `lxc config edit` コマンドを使って手動で編集できます。有効な有効期限が設定されたスナップショットはタスク実行時に削除されます。有効期限は `expires_at` に空文字列や `0001-01-01T00:00:00Z`（zero time）を設定することで無効化できます。`snapshots.expiry` が設定されていない場合はこれがデフォルトです。

これは次のような新しいエンドポイントを追加します（詳しくは *RESTful API* をご覧ください）：

- `PUT /1.0/containers/<name>/snapshots/<name>`

snapshot_expiry_creation

コンテナ作成に `expires\_at` を追加し、作成時にスナップショットの有効期限を上書きできます。

network_leases_location

ネットワークのリースリストに "Location" フィールドを導入します。これは、特定のリースがどのノードに存在するかを問い合わせるときに使います。

resources_cpu_socket

ソケットの情報が入れ替わる場合に備えて CPU リソースにソケットフィールドを追加します。

resources_gpu

サーバリソースに新規に GPU 構造を追加し、システム上で利用可能な全ての GPU を一覧表示します。

resources_numa

全ての CPU と GPU に対する NUMA ノードを表示します。

kernel_features

サーバの環境からオプショナルなカーネル機能の使用可否状態を取得します。

id_map_current

内部的な `volatile.idmap.current` キーを新規に導入します。これはコンテナに対する現在のマッピングを追跡するのに使われます。

実質的には以下が利用可能になります。

- `volatile.last_state.idmap` => ディスク上の idmap
- `volatile.idmap.current` => 現在のカーネルマップ
- `volatile.idmap.next` => 次のディスク上の idmap

これはディスク上の map が変更されていないがカーネルマップは変更されている (例: shiftfs) ような環境を実装するために必要です。

event_location

API イベントの世代の場所を公開します。

storage_api_remote_volume_snapshots

ストレージボリュームをそれらのスナップショットを含んで移行できます。

network_nat_address

これは LXD ブリッジに `ipv4.nat.address` と `ipv6.nat.address` 設定キーを導入します。これらのキーはブリッジからの送信トラフィックに使うソースアドレスを制御します。

container_nic_routes

これは "nic" タイプのデバイスに `ipv4.routes` と `ipv6.routes` プロパティを導入します。ホストからコンテナへの nic への静的ルートが追加できます。

rbac

RBAC (role based access control; ロールベースのアクセス制御) のサポートを追加します。これは以下の設定キーを新規に導入します。

- rbac.api.url
- rbac.api.key
- rbac.api.expiry
- rbac.agent.url
- rbac.agent.username
- rbac.agent.private_key
- rbac.agent.public_key

cluster_internal_copy

これは通常の "POST /1.0/containers" を実行することでクラスタノード間でコンテナをコピーすることを可能にします。この際 LXD はマイグレーションが必要かどうかを内部的に判定します。

seccomp_notify

カーネルが seccomp ベースの syscall インターセプトをサポートする場合に登録された syscall が実行されたことをコンテナから LXD に通知することができます。 LXD はそれを受けて様々なアクションをトリガーするかを決定します。

lxc_features

これは `GET /1.0/` ルート経由で `lxc info` コマンドの出力に `lxc_features` セクションを導入します。配下の LXC ライブラリーに存在するキー・フィーチャーに対するチェックの結果を出力します。

container_nic_ipvlan

これは "nic" デバイスに `ipvlan` のタイプを導入します。

network_vlan_sriov

これは SR-IOV デバイスに VLAN (`vlan`) と MAC フィルタリング (`security.mac_filtering`) のサポートを導入します。

storage_cephfs

ストレージプールドライバとして CEPHFS のサポートを追加します。これはカスタムボリュームとしての利用のみが可能になり、イメージとコンテナは CEPHFS ではなく CEPH (RBD) 上に構築する必要があります。

container_nic_ipfilter

これは `bridged` の NIC デバイスに対してコンテナの IP フィルタリング (`security.ipv4_filtering` and `security.ipv6_filtering`) を導入します。

resources_v2

/1.0/resources のリソース API を見直しました。主な変更は以下の通りです。

- CPU
  - ソケット、コア、スレッドのトラッキングのレポートを修正しました
  - コア毎の NUMA ノードのトラッキング
  - ソケット毎のベースとターボの周波数のトラッキング
  - コア毎の現在の周波数のトラッキング
  - CPU のキャッシュ情報の追加
  - CPU アーキテクチャをエクスポート
  - スレッドのオンライン／オフライン状態を表示
- メモリ
  - HugePages のトラッキングを追加
  - NUMA ノード毎でもメモリ消費を追跡
- GPU
  - DRM 情報を別の構造体に分離
  - DRM 構造体内にデバイスの名前とノードを公開
  - NVIDIA 構造体内にデバイスの名前とノードを公開

- SR-IOV VF のトラッキングを追加

container_exec_user_group_cwd

`POST /1.0/containers/NAME/exec` の実行時に User, Group と Cwd を指定するサポートを追加

container_syscall_intercept

`security.syscalls.intercept.\*` 設定キーを追加します。これはどのシステムコールを LXD がインターセプトし昇格された権限で処理するかを制御します。

container_disk_shift

`disk` デバイスに `shift` プロパティを追加します。これは shiftfs のオーバーレイの使用を制御します。

storage_shifted

ストレージボリュームに新しく `security.shifted` という boolean の設定を導入します。

これを true に設定すると複数の隔離されたコンテナが、それら全てがファイルシステムに書き込み可能にしたまま、同じストレージボリュームにアタッチするのを許可します。

これは shiftfs をオーバーレイファイルシステムとして使用します。

resources_infiniband

リソース API の一部として infiniband キャラクタデバイス (issm, umad, uverb) の情報を公開します。

daemon_storage

これは `storage.images_volume` と `storage.backups_volume` という 2 つの新しい設定項目を導入します。これらは既存のプール上のストレージボリュームがデーモン全体のイメージとバックアップを保管するのに使えるようにします。

instances

これはインスタンスの概念を導入します。現状ではインスタンスの唯一の種別は "container" です。

image_types

これはイメージに新しく Type フィールドのサポートを導入します。 Type フィールドはイメージがどういう種別かを示します。

resources_disk_sata

ディスクリソース API の構造体を次の項目を含むように拡張します。

- sata デバイス (種別) の適切な検出
- デバイスパス
- ドライブの RPM
- ブロックサイズ
- ファームウェアバージョン
- シリアルナンバー

clustering_roles

これはクラスタのエントリーに `roles` という新しい属性を追加し、クラスタ内のメンバーが提供する role の一覧を公開します。

images_expiry

イメージの有効期限を設定できます。

resources_network_firmware

ネットワークカードのエントリーに FirmwareVersion フィールドを追加します。

backup_compression_algorithm

バックアップを作成する (`POST /1.0/containers/<name>/backups`) 際に `compression_algorithm` プロパティのサポートを追加します。

このプロパティを設定するとデフォルト値 (`backups.compression_algorithm`) をオーバーライドすることができます。

ceph_data_pool_name

Ceph RBD を使ってストレージプールを作成する際にオプショナルな引数 (`ceph.osd.data_pool_name`) のサポートを追加します。この引数が指定されると、プールはメタデータは `pool_name` で指定されたプールに保持しつつ実際のデータは `data_pool_name` で指定されたプールに保管するようになります。

container_syscall_intercept_mount

`security.syscalls.intercept.mount`, `security.syscalls.intercept.mount.allowed`, `security.syscalls.intercept.mount.shift` 設定キーを追加します。これらは mount システムコールを LXD にインターセプトさせるかどうか、昇格されたパーミションでどのように処理させるかを制御します。

compression_squashfs

イメージやバックアップを SquashFS ファイルシステムの形式でインポート／エクスポートするサポートを追加します。

container_raw_mount

ディスクデバイスに raw mount オプションを渡すサポートを追加します。

container_nic_routed

`routed` "nic" デバイスタイプを導入します。

container_syscall_intercept_mount_fuse

`security.syscalls.intercept.mount.fuse` キーを追加します。これはファイルシステムのマウントを fuse 実装にリダイレクトするのに使えます。このためには例えば `security.syscalls.intercept.mount.fuse=ext4=fuse2fs` のように設定します。

container_disk_ceph

既存の CEPH RDB もしくは FS を直接 LXD コンテナに接続できます。

virtual_machines

仮想マシンサポートが追加されます。

image_profiles

新しいコンテナを起動するときに、イメージに適用するプロファイルのリストが指定できます。

clustering_architecture

クラスタメンバーに `architecture` 属性を追加します。この属性はクラスタメンバーのアーキテクチャを示します。

resources_disk_id

リソース API のディスクのエントリーに device_id フィールドを追加します。

storage_lvm_stripes

通常のボリュームと thin pool ボリューム上で LVM ストライプを使う機能を追加します。

vm_boot_priority

ブートの順序を制御するため nic とディスクデバイスに `boot.priority` プロパティを追加します。

unix_hotplug_devices

UNIX のキャラクタデバイスとブロックデバイスのホットプラグのサポートを追加します。

api_filtering

インスタンスとイメージに対する GET リクエストの結果をフィルタリングする機能を追加します。

instance_nic_network

NIC デバイスの `network` プロパティのサポートを追加し、管理されたネットワークへ NIC をリンクできるようにします。これによりネットワーク設定の一部を引き継ぎ、 IP 設定のより良い検証を行うことができます。

clustering_sizing

データベースの投票者とスタンバイに対してカスタムの値を指定するサポートです。`cluster.max_voters` と `cluster.max_standby` という新しい設定キーが導入され、データベースの投票者とスタンバイの理想的な数を指定できます。

firewall_driver

ServerEnvironment 構造体にファイアーウォールのドライバーが使用されていることを示す `Firewall` プロパティを追加します。

storage_lvm_vg_force_reuse

既存の空でないボリュームグループからストレージボリュームを作成する機能を追加します。このオプションの使用には注意が必要です。というのは、同じボリュームグループ内に LXD 以外で作成されたボリュームとボリューム名が衝突しないことを LXD が保証できないからです。このことはもし名前の衝突が起きたときは LXD 以外で作成されたボリュームを LXD が削除してしまう可能性があることを意味します。

container_syscall_intercept_hugetlbfs

mount システムコール・インターセプションが有効にされ hugetlbfs が許可されたファイルシステムとして指定された場合、 LXD は別の hugetlbfs インスタンスを uid と gid をコンテナの root の uid と gid に設定するマウントオプションを指定してコンテナにマウントします。これによりコンテナ内のプロセスが hugepage を確実に利用できるようにします。

limits_hugepages

コンテナが使用できる hugepage の数を hugetlb cgroup を使って制限できるようにします。この機能を使用するには hugetlb cgroup が利用可能になっている必要があります。注意: hugetlbfs ファイルシステムの mount システムコールをインターセプトするときは、ホストの hugepage のリソースをコンテナが使い切ってしまわないように hugepage を制限することを推奨します。

container_nic_routed_gateway

この拡張は `ipv4.gateway` と `ipv6.gateway` という NIC の設定キーを追加します。指定可能な値は auto か none のいずれかです。値を指定しない場合のデフォルト値は auto です。auto に設定した場合は、デフォルトゲートウェイがコンテナ内部に追加され、ホスト側のインタフェースにも同じゲートウェイアドレスが追加されるという現在の挙動と同じになります。none に設定すると、デフォルトゲートウェイもアドレスもホスト側のインターフェースには追加されません。これにより複数のルートを持つ NIC デバイスをコンテナに追加できます。

projects_restrictions

この拡張はプロジェクトに `restricted` という設定キーを追加します。これによりプロジェクト内でセキュリティセンシティブな機能を使うのを防ぐことができます。

custom_volume_snapshot_expiry

この拡張はカスタムボリュームのスナップショットに有効期限を設定できるようにします。有効期限は `snapshots.expiry` 設定キーにより個別に設定することも出来ますし、親のカスタムボリュームに設定してそこから作成された全てのスナップショットに自動的にその有効期限を適用することも出来ます。

volume_snapshot_scheduling

この拡張はカスタムボリュームのスナップショットにスケジュール機能を追加します。`snapshots.schedule` と `snapshots.pattern` という 2 つの設定キーが新たに追加されます。スナップショットは最短で 1 分毎に作成可能です。

trust_ca_certificates

この拡張により提供された CA (`server.ca`) によって信頼されたクライアント証明書のチェックが可能になります。`core.trust_ca_certificates` を true に設定すると有効にできます。有効な場合、クライアント証明書のチェックを行い、チェックが OK であれば信頼されたパスワードの要求はスキップします。ただし、提供された CRL (`ca.crl`) に接続してきたクライアント証明書が含まれる場合は例外です。この場合は、パスワードが求められます。

snapshot_disk_usage

この拡張はスナップショットのディスク使用量を示す `/1.0/instances/<name>/snapshots/<snapshot>` の出力に `size` フィールドを新たに追加します。

clustering_edit_roles

この拡張はクラスタメンバーに書き込み可能なエンドポイントを追加し、ロールの編集を可能にします。

container_nic_routed_host_address

この拡張は NIC の設定キーに `ipv4.host_address` と `ipv6.host_address` を追加し、ホスト側の veth インターフェースの IP アドレスを制御できるようにします。これは同時に複数の routed NIC を使用し、予測可能な next-hop のアドレスを使用したい場合に有用です。

さらにこの拡張は `ipv4.gateway` と `ipv6.gateway` の NIC 設定キーの振る舞いを変更します。auto に設定するとコンテナはデフォルトゲートウェイをそれぞれ `ipv4.host_address` と `ipv6.host_address` で指定した値にします。

デフォルト値は次の通りです。

`ipv4.host_address`: 169.254.0.1 `ipv6.host_address`: fe80::1

これは以前のデフォルトの挙動と後方互換性があります。

container_nic_ipvlan_gateway

この拡張は `ipv4.gateway` と `ipv6.gateway` の NIC 設定キーを追加し auto か none の値を指定できます。指定しない場合のデフォルト値は auto です。この場合は従来同様の挙動になりコンテナ内部に追加されるデフォルトゲートウェイと同じアドレスがホスト側のインターフェースにも追加されます。none に設定された場合、ホスト側のインターフェースにはデフォルトゲートウェイもアドレスも追加されません。これによりコンテナに ipvlan の NIC デバイスを複数追加することができます。

resources_usb_pci

この拡張は `/1.0/resources` の出力に USB と PC デバイスを追加します。

resources_cpu_threads_numa

この拡張は numa_node フィールドをコアごとではなくスレッドごとに記録するように変更します。これは一部のハードウェアでスレッドを異なる NUMA ドメインに入れる場合があるようなのでそれに対応するためのものです。

resources_cpu_core_die

それぞれのコアごとに `die_id` 情報を公開します。

api_os

この拡張は `/1.0` 内に `os` と `os_version` の 2 つのフィールドを追加します。

これらの値はシステム上の os-release のデータから取得されます。

container_nic_routed_host_table

この拡張は `ipv4.host_table` と `ipv6.host_table` という NIC の設定キーを導入します。これで指定した ID のカスタムポリシーのルーティングテーブルにインスタンスの IP のための静的ルートを追加できます。

container_nic_ipvlan_host_table

この拡張は `ipv4.host_table` と `ipv6.host_table` という NIC の設定キーを導入します。これで指定した ID のカスタムポリシーのルーティングテーブルにインスタンスの IP のための静的ルートを追加できます。

container_nic_ipvlan_mode

この拡張は `mode` という NIC の設定キーを導入します。これにより `ipvlan` モードを `l2` か `l3s` のいずれかに切り替えられます。指定しない場合、デフォルトは `l3s` （従来の挙動）です。

`l2` モードでは `ipv4.address` と `ipv6.address` キーは CIDR か単一アドレスの形式を受け付けます。単一アドレスの形式を使う場合、デフォルトのサブネットのサイズは IPv4 では /24 、 IPv6 では /64 となります。

`l2` モードでは `ipv4.gateway` と `ipv6.gateway` キーは単一の IP アドレスのみを受け付けます。

resources_system

この拡張は `/1.0/resources` の出力にシステム情報を追加します。

images_push_relay

この拡張はイメージのコピーに push と relay モードを追加します。また以下の新しいエンドポイントも追加します。

- `POST 1.0/images/<fingerprint>/export`

network_dns_search

この拡張はネットワークに `dns.search` という設定オプションを追加します。

container_nic_routed_limits

この拡張は routed NIC に `limits.ingress`, `limits.egress`, `limits.max` を追加します。

instance_nic_bridged_vlan

この拡張は `bridged` NIC に `vlan` と `vlan.tagged` の設定を追加します。

`vlan` には参加するタグなし VLAN を指定し、 `vlan.tagged` は参加するタグ VLAN のカンマ区切りリストを指定します。

network_state_bond_bridge

この拡張は /1.0/networks/NAME/state API に bridge と bond のセクションを追加します。

これらはそれぞれの特定のタイプに関連する追加の状態の情報を含みます。

Bond:

- Mode
- Transmit hash
- Up delay
- Down delay
- MII frequency
- MII state
- Lower devices

Bridge:

- ID
- Forward delay
- STP mode
- Default VLAN
- VLAN filtering
- Upper devices

resources_cpu_isolated

この拡張は CPU スレッドに `Isolated` プロパティを追加します。これはスレッドが物理的には `Online` ですがタスクを受け付けないように設定しているかを示します。

usedby_consistency

この拡張により、可能な時は UsedBy が適切な ?project= と ?target= に対して一貫性があるようになるはずです。

UsedBy を持つ 5 つのエンティティーは以下の通りです。

- プロファイル
- プロジェクト
- ネットワーク
- ストレージプール
- ストレージボリューム

custom_block_volumes

この拡張によりカスタムブロックボリュームを作成しインスタンスにアタッチできるようになります。カスタムストレージボリュームの作成時に `--type` フラグが新規追加され、 `fs` と `block` の値を受け付けます。

clustering_failure_domains

この拡張は PUT /1.0/cluster/<node> API に failure_domain フィールドを追加します。これはノードの failure domain を設定するのに使えます。

container_syscall_filtering_allow_deny_syntax

いくつかのシステムコールに関連したコンテナの設定キーが更新されました。

- security.syscalls.deny_default
- security.syscalls.deny_compat
- security.syscalls.deny
- security.syscalls.allow

resources_gpu_mdev

/1.0/resources の利用可能な媒介デバイス (mediated device) のプロファイルとデバイスを公開します。

console_vga_type

この拡張は /1.0/console エンドポイントが ?type= 引数を取るように拡張します。これは console (デフォルト) か vga (この拡張で追加される新しいタイプ) を指定可能です。

/1.0/<instance name>/console?type=vga に POST する際はメタデータフィールド内の操作の結果ウェブソケットにより返されるデータはターゲットの仮想マシンの SPICE unix ソケットにアタッチされた双方向のプロキシになります。

projects_limits_disk

利用可能なプロジェクトの設定キーに `limits.disk` を追加します。これが設定されるとプロジェクト内でインスタンスボリューム、カスタムボリューム、イメージボリュームが使用できるディスクスペースの合計の量を制限できます。

network_type_macvlan

ネットワークタイプ `macvlan` のサポートを追加し、このネットワークタイプに `parent` 設定キーを追加します。これは NIC デバイスインターフェースを作る際にどの親インターフェースを使用するべきかを指定します。

さらに `macvlan` の NIC に `network` 設定キーを追加します。これは NIC デバイスの設定の基礎として使う network を指定します。

network_type_sriov

ネットワークタイプ `sriov` のサポートを追加し、このネットワークタイプに `parent` 設定キーを追加します。これは NIC デバイスインターフェースを作る際にどの親インターフェースを使用するべきかを指定します。

さらに `sriov` の NIC に `network` 設定キーを追加します。これは NIC デバイスの設定の基礎として使う network を指定します。

container_syscall_intercept_bpf_devices

この拡張はコンテナ内で bpf のシステムコールをインターセプトする機能を提供します。具体的には device cgroup の bpf のプログラムを管理できるようにします。

network_type_ovn

ネットワークタイプ `ovn` のサポートを追加し、 `bridge` タイプのネットワークを `parent` として設定できるようにします。

`ovn` という新しい NIC のデバイスタイプを追加します。これにより `network` 設定キーにどの `ovn` のタイプのネットワークに接続すべきかを指定できます。

さらに全ての `ovn` ネットワークと NIC デバイスに適用される 2 つのグローバルの設定キーを追加します。

- network.ovn.integration_bridge - 使用する OVS 統合ブリッジ
- network.ovn.northbound_connection - OVN northbound データベース接続文字列

projects_networks

プロジェクトに features.networks 設定キーを追加し、プロジェクトがネットワークを保持できるようにします。

projects_networks_restricted_uplinks

プロジェクトに restricted.networks.uplinks 設定キーを追加し、プロジェクト内で作られたネットワークがそのアップリンクのネットワークとしてどのネットワークが使えるかを（カンマ区切りリストで）指定します。

custom_volume_backup

カスタムボリュームのバックアップサポートを追加します。

この拡張は以下の新しい API エンドポイント （詳細は *RESTful API* を参照）を含みます。

- GET /1.0/storage-pools/<pool>/<type>/<volume>/backups
- POST /1.0/storage-pools/<pool>/<type>/<volume>/backups
- GET /1.0/storage-pools/<pool>/<type>/<volume>/backups/<name>
- POST /1.0/storage-pools/<pool>/<type>/<volume>/backups/<name>
- DELETE /1.0/storage-pools/<pool>/<type>/<volume>/backups/<name>
- GET /1.0/storage-pools/<pool>/<type>/<volume>/backups/<name>/export

以下の既存のエンドポイントが変更されます。

- POST /1.0/storage-pools/<pool>/<type>/<volume> が新しいソースタイプとして backup を受け付けます

backup_override_name

InstanceBackupArgs に Name フィールドを追加し、バックアップをリストアする際に別のインスタンス名を指定できるようにします。

StoragePoolVolumeBackupArgs に Name と PoolName フィールドを追加し、カスタムボリュームのバックアップをリストアする際に別のボリューム名を指定できるようにします。

storage_rsync_compression

ストレージプールに rsync.compression 設定キーを追加します。このキーはストレージプールをマイグレートする際に rsync での圧縮を無効にするために使うことができます。

network_type_physical

新たに physical というネットワークタイプのサポートを追加し、 ovn ネットワークのアップリンクとして使用できるようにします。

physical ネットワークの parent で指定するインターフェースは ovn ネットワークのゲートウェイに接続されます。

network_ovn_external_subnets

ovn ネットワークがアップリンクネットワークの外部のサブネットを使用できるようにします。

physical ネットワークに ipv4.routes と ipv6.routes の設定を追加します。これは子供の OVN ネットワークで ipv4.routes.external と ipv6.routes.external の設定で使用可能な外部のルートを指定します。

プロジェクトに restricted.networks.subnets 設定を追加します。これはプロジェクト内の OVN ネットワークで使用可能な外部のサブネットを指定します（未設定の場合はアップリンクネットワークで定義される全てのルートが使用可能です）。

network_ovn_nat

ovn ネットワークに ipv4.nat と ipv6.nat の設定を追加します。

これらの設定（訳注: ipv4.nat や ipv6.nat）を未設定でネットワークを作成する際、（訳注: ipv4.address や ipv6.address が未設定あるいは auto の場合に）対応するアドレス （訳注: ipv4.nat であれば ipv4.address、ipv6.nat であれば ipv6.address）がサブネット用に生成される場合は適切な NAT が生成され、ipv4.nat や ipv6.nat は true に設定されます。

この設定がない場合は値は false として扱われます。

network_ovn_external_routes_remove

ovn ネットワークから ipv4.routes.external と ipv6.routes.external の設定を削除します。

ネットワークと NIC レベルの両方で指定するのではなく、 ovn NIC タイプ上で等価な設定を使えます。

tpm_device_type

`tpm` デバイスタイプを導入します。

storage_zfs_clone_copy_rebase

zfs.clone_copy に `rebase` という値を導入します。この設定で LXD は先祖の系列上の "image" データセットを追跡し、その最上位に対して send/receive を実行します。

gpu_mdev

これは仮想 CPU のサポートを追加します。GPU デバイスに `mdev` 設定キーを追加し、i915-GVTg_V5_4 のようなサポートされる mdev のタイプを指定します。

resources_pci_iommu

これはリソース API の PCI エントリーに IOMMUGroup フィールドを追加します。

resources_network_usb

リソース API のネットワークカードエントリーに usb_address フィールドを追加します。

resources_disk_address

リソース API のディスクエントリーに usb_address と pci_address フィールドを追加します。

network_physical_ovn_ingress_mode

`physical` ネットワークに `ovn.ingress_mode` 設定を追加します。

OVN NIC ネットワークの外部 IP アドレスがアップリンクネットワークにどのように広告されるかの方法を設定します。

`l2proxy` (proxy ARP/NDP) か `routed` のいずれかを指定します。

network_ovn_dhcp

ovn ネットワークに ipv4.dhcp と ipv6.dhcp の設定を追加します。

DHCP (と IPv6 の RA) を無効にできます。デフォルトはオンです。

network_physical_routes_anycast

physical ネットワークに ipv4.routes.anycast と ipv6.routes.anycast の boolean の設定を追加します。デフォルトは false です。

ovn.ingress_mode=routed と共に使うと physical ネットワークをアップリンクとして使う OVN ネットワークでサブネット／ルートのオーバーラップ検出を緩和できます。

projects_limits_instances

limits.instances を利用可能なプロジェクトの設定キーに追加します。設定するとプロジェクト内で使われるインスタンス（VM とコンテナ）の合計数を制限します。

network_state_vlan

これは /1.0/networks/NAME/state API に "vlan" セクションを追加します。

これらは VLAN インターフェースに関連する追加の状態の情報を含みます。

- lower_device
- vid

instance_nic_bridged_port_isolation

これは bridged NIC に security.port_isolation のフィールドを追加します。

instance_bulk_state_change

一括状態変更（詳細は *REST API* を参照）のために次のエンドポイントを追加します。

- PUT /1.0/instances

network_gvrp

これはオプショナルな gvrp プロパティを macvlan と physical ネットワークに追加し、さらに ipvlan, macvlan, routed, physical NIC デバイスにも追加します。

設定された場合は、これは VLAN が GARP VLAN Registration Protocol を使って登録すべきかどうかを指定します。デフォルトは false です。

instance_pool_move

これは POST /1.0/instances/NAME API に pool フィールドを追加し、プール間でインスタンスのルートディスクを簡単に移動できるようにします。

gpu_sriov

これは SR-IOV を有効にした GPU のサポートを追加します。これにより sriov という GPU タイプのプロパティが追加されます。

pci_device_type

これは pci デバイスタイプを追加します。

storage_volume_state

/1.0/storage-pools/POOL/volumes/VOLUME/state API エンドポイントを新規追加しボリュームの使用量を取得できるようにします。

network_acl

これは /1.0/network-acls の API エンドポイントプリフィクス以下の API にネットワークの ACL のコンセプトを追加します。

migration_stateful

migration.stateful という設定キーを追加します。

disk_state_quota

これは disk デバイスに size.state というデバイス設定キーを追加します。

storage_ceph_features

ストレージプールに ceph.rbd.features 設定キーを追加し、新規ボリュームに使用する RBD の機能を制御します。

projects_compression

backups.compression_algorithm と images.compression_algorithm 設定キーを追加します。これらによりプロジェクトごとのバックアップとイメージの圧縮の設定が出来るようになります。

projects_images_remote_cache_expiry

プロジェクトに images.remote_cache_expiry 設定キーを追加します。これを設定するとキャッシュされたリモートのイメージが指定の日数使われない場合は削除されるようになります。

certificate_project

API 内の証明書に restricted と projects プロパティを追加します。projects は証明書がアクセスしたプロジェクト名の一覧を保持します。

network_ovn_acl

OVN ネットワークと OVN NIC に security.acls プロパティを追加します。これにより ネットワークに ACL をかけられるようになります。

projects_images_auto_update

images.auto_update_cached と images.auto_update_interval 設定キーを追加します。これらによりプロジェクト内のイメージの自動更新を設定できるようになります。

projects_restricted_cluster_target

プロジェクトに restricted.cluster.target 設定キーを追加します。これによりどのクラスタメンバーにワークロードを配置するかやメンバー間のワークロードを移動する能力を指定する --target オプションをユーザーに使わせないように出来ます。

images_default_architecture

images.default_architecture をグローバルの設定キーとプロジェクトごとの設定キーとして追加します。これはイメージリクエストの一部として明示的に指定しなかった場合にどのアーキテクチャーを使用するかを LXD に指定します。

network_ovn_acl_defaults

OVN ネットワークと NIC に security.acls.default.{in,e}gress.action と security.acls.default.{in,e}gress.logged 設定キーを追加します。これは削除された ACL の default.action と default.logged キーの代わりになるものです。

gpu_mig

これは NVIDIA MIG のサポートを追加します。mig gputype と関連する設定キーを追加します。

project_usage

プロジェクトに現在のリソース割り当ての情報を取得する API エンドポイントを追加します。API の GET /1.0/projects/<name>/state で利用できます。

network_bridge_acl

bridge ネットワークに security.acls 設定キーを追加し、ネットワーク ACL を適用できるようにします。

さらにマッチしなかったトラフィックに対するデフォルトの振る舞いを指定する security.acls.default.{in,e}gress.action と security.acls.default.{in,e}gress.logged 設定キーを追加します。

warnings

LXD の警告 API です。

この拡張は次のエンドポイントを含みます（詳細は *Restful API* 参照）。

- `GET /1.0/warnings`
- `GET /1.0/warnings/<uuid>`
- `PUT /1.0/warnings/<uuid>`
- `DELETE /1.0/warnings/<uuid>`

projects_restricted_backups_and_snapshots

プロジェクトに `restricted.backups` と `restricted.snapshots` 設定キーを追加し、ユーザーがバックアップやスナップショットを作成できないようにします。

clustering_join_token

トラスト・パスワードを使わずに新しいクラスタメンバーを追加する際に使用する参加トークンをリクエストするための `POST /1.0/cluster/members` API エンドポイントを追加します。

clustering_description

クラスタメンバーに編集可能な説明を追加します。

server_trusted_proxy

`core.https_trusted_proxy` のサポートを追加します。 この設定は、LXD が HAProxy スタイルの connection ヘッダーをパースし、そのような（HAProxy などのリバースプロキシサーバが LXD の前面に存在するような）接続の場合でヘッダーが存在する場合は、プロキシサーバが（ヘッダーで）提供するリクエストの（実際のクライアントの）ソースアドレスへ（LXD が）ソースアドレスを書き換え（て、LXD の管理するクラスタにリクエストを送出し）ます。（LXD のログにもオリジナルのアドレスを記録します）

clustering_update_cert

クラスタ全体に適用されるクラスタ証明書を更新するための `PUT /1.0/cluster/certificate` エンドポイントを追加します。

storage_api_project

これはプロジェクト間でカスタムストレージボリュームをコピー／移動できるようにします。

server_instance_driver_operational

これは `/1.0` エンドポイントの `driver` の出力をサーバ上で実際にサポートされ利用可能であるドライバーのみを含めるように修正します（LXD に含まれるがサーバ上では利用不可なドライバーも含めるのとは違って）。

server_supported_storage_drivers

これはサーバの環境情報にサポートされているストレージドライバーの情報を追加します。

event_lifecycle_requestor_address

lifecycle requestor に address のフィールドを追加します。

resources_gpu_usb

リソース API 内の ResourcesGPUCard (GPU エントリ) に USBAddress (usb_address) を追加します。

clustering_evacuation

クラスタメンバーを待避と復元するための `POST /1.0/cluster/members/<name>/state` エンドポイントを追加します。また設定キー `cluster.evacuate` と `volatile.evacuate.origin` も追加します。これらはそれぞれ待避の方法 (`auto`, `stop` or `migrate`) と移動したインスタンスのオリジンを設定します。

network_ovn_nat_address

これは LXD の `ovn` ネットワークに `ipv4.nat.address` と `ipv6.nat.address` 設定キーを追加します。これらのキーで OVN 仮想ネットワークからの外向きトラフィックのソースアドレスを制御します。これらのキーは OVN ネットワークのアップリンクネットワークが `ovn.ingress_mode=routed` という設定を持つ場合にのみ指定可能です。

network_bgp

これは LXD を BGP ルーターとして振る舞わせルートを bridge と ovn ネットワークに広告するようにします。

以下のグローバル設定が追加されます。

- core.bgp_address
- core.bgp_asn
- core.bgp_routerid

以下のネットワーク設定キーが追加されます（bridge と physical）。

- bgp.peers.<name>.address
- bgp.peers.<name>.asn
- bgp.peers.<name>.password
- bgp.ipv4.nexthop
- bgp.ipv6.nexthop

そして下記の NIC 特有な設定が追加されます（nictype が bridged の場合）。

- ipv4.routes.external
- ipv6.routes.external

network_forward

これはネットワークアドレスのフォワード機能を追加します。bridge と ovn ネットワークで外部 IP アドレスを定義して対応するネットワーク内の内部 IP アドレス (複数指定可能) にフォワード出来ます。

custom_volume_refresh

ボリュームマイグレーションに refresh オプションのサポートを追加します。

network_counters_errors_dropped

これはネットワークカウンターに受信エラー数、送信エラー数とインバウンドとアウトバウンドのドロップしたパケット数を追加します。

metrics

これは LXD にメトリクスを追加します。実行中のインスタンスのメトリクスを OpenMetrics 形式で返します。

この拡張は次のエンドポイントを含みます。

- `GET /1.0/metrics`

image_source_project

`POST /1.0/images` に `project` フィールドを追加し、イメージコピー時にコピー元プロジェクトを設定できるようにします。

clustering_config

クラスタメンバーに `config` プロパティを追加し、キー・バリュー・ペアを設定可能にします。

network_peer

ネットワークピアリングを追加し、 OVN ネットワーク間のトラフィックが OVN サブシステムの外に出ずに通信できるようにします。

linux_sysctl

`linux.sysctl.*` 設定キーを追加し、ユーザーが一コンテナ内の一部のカーネルパラメータを変更できるようにします。

network_dns

組み込みの DNS サーバとゾーン API を追加し、 LXD インスタンスに DNS レコードを提供します。

以下のサーバ設定キーが追加されます。

- `core.dns_address`

以下のネットワーク設定キーが追加されます。

- `dns.zone.forward`
- `dns.zone.reverse.ipv4`
- `dns.zone.reverse.ipv6`

以下のプロジェクト設定キーが追加されます。

- `restricted.networks.zones`

DNS ゾーンを管理するために下記の REST API が追加されます。

- /1.0/network-zones (GET, POST)
- /1.0/network-zones/<name> (GET, PUT, PATCH, DELETE)

ovn_nic_acceleration

OVN NIC に acceleration 設定キーを追加し、ハードウェアオフロードを有効にするのに使用できます。設定値は none または sriov です。

certificate_self_renewal

これはクライアント自身の信頼証明書の更新のサポートを追加します。

instance_project_move

これは POST /1.0/instances/NAME API に project フィールドを追加し、インスタンスをプロジェクト間で簡単に移動できるようにします。

storage_volume_project_move

これはストレージボリュームのプロジェクト間での移動のサポートを追加します。

cloud_init

これは以下のキーを含む project 設定キー名前空間を追加します。

- cloud-init.vendor-data
- cloud-init.user-data
- cloud-init.network-config

これはまた devlxd にインスタンスのデバイスを表示する /1.0/devices エンドポイントを追加します。

network_dns_nat

これはネットワークゾーン (DNS) に `network.nat` を設定オプションとして追加します。

デフォルトでは全てのインスタンスの NIC のレコードを生成するという現状の挙動になりますが、`false` に設定すると外部から到達可能なアドレスのレコードのみを生成するよう LXD に指示します。

database_leader

クラスタ・リーダーに設定される "database-leader" ロールを追加します。

instance_all_projects

全てのプロジェクトのインスタンス表示のサポートを追加します。

clustering_groups

クラスタ・メンバーのグループ化のサポートを追加します。

これは以下の新しいエンドポイントを追加します。

- `/1.0/cluster/groups` (GET, POST)
- `/1.0/cluster/groups/<name>` (GET, POST, PUT, PATCH, DELETE)

以下のプロジェクトの制限が追加されます。

- `restricted.cluster.groups`

ceph_rbd_du

Ceph ストレージプールに `ceph.rbd.du` という boolean の設定を追加します。実行に時間がかかるかもしれない `rbd du` の呼び出しの使用を無効化できます。

instance_get_full

これは `GET /1.0/instances/{name}` に recursion=1 のモードを追加します。これは状態、スナップショット、バックアップの構造体を含む全てのインスタンスの構造体が取得できます。

qemu_metrics

これは `security.agent.metrics` という boolean 値を追加します。デフォルト値は `true` です。`false` に設定するとメトリクスや他の状態の取得のために lxd-agent に接続することはせず、 QEMU からの統計情報に頼ります。

gpu_mig_uuid

Nvidia `470+` ドライバー (例. `MIG-74c6a31a-fde5-5c61-973b-70e12346c202`) で使用される MIG UUID 形式のサポートを追加します。`MIG-` の接頭辞は省略できます。

この拡張が古い `mig.gi` と `mig.ci` パラメーターに取って代わります。これらは古いドライバーとの互換性のため残されますが、同時には設定できません。

event_project

イベントの API にイベントが属するプロジェクトを公開します。

clustering_evacuation_live

`cluster.evacuate` への設定値 `live-migrate` を追加します。これはクラスタ待避の際にインスタンスのライブマイグレーションを強制します。

instance_allow_inconsistent_copy

`POST /1.0/instances` のインスタンスソースに `allow_inconsistent` フィールドを追加します。true の場合、rsync はコピーからインスタンスを生成するときに `Partial transfer due to vanished source files` (code 24) エラーを無視します。

network_state_ovn

これにより、/1.0/networks/NAME/state API に "ovn "セクションが追加されます。これには OVN ネットワークに関連する追加の状態情報が含まれます:

- chassis (シャーシ)

storage_volume_api_filtering

ストレージボリュームの GET リクエストの結果をフィルタリングする機能を追加します。

image_restrictions

この拡張機能は、イメージのプロパティに、イメージの制限やホストの要件を追加します。これらの要件はインスタンスとホストシステムとの互換性を決定するのに役立ちます。

storage_zfs_export

`zfs.export` を設定することで、プールのアンマウント時に zpool のエクスポートを無効にする機能を導入しました。

network_dns_records

network zones (DNS) API を拡張し、カスタムレコードの作成と管理機能を追加します。

これにより、以下が追加されます。

- `GET /1.0/network-zones/ZONE/records`
- `POST /1.0/network-zones/ZONE/records`
- `GET /1.0/network-zones/ZONE/records/RECORD`
- `PUT /1.0/network-zones/ZONE/records/RECORD`
- `PATCH /1.0/network-zones/ZONE/records/RECORD`
- `DELETE /1.0/network-zones/ZONE/records/RECORD`

storage_zfs_reserve_space

quota/refquota に加えて、ZFS プロパティの reservation/refreservation を設定する機能を追加します。

network_acl_log

ACL ファイアウォールのログを取得するための API `GET /1.0/networks-acls/NAME/log` を追加します。

storage_zfs_blocksize

ZFS ストレージボリュームに新しい `zfs.blocksize` プロパティを導入し、ボリュームのブロックサイズを設定できるようになります。

metrics_cpu_seconds

LXD が使用する CPU 時間をミリ秒ではなく秒単位で出力するように修正されたかどうかを検出するために使用されます。

instance_snapshot_never

`snapshots.schedule` に`@never` オプションを追加し、継承を無効にすることができます。

certificate_token

トラストストアに、トラストパスワードに代わる安全な手段として、トークンベースの証明書を追加します。

これは `POST /1.0/certificates` に `token` フィールドを追加します。

instance_nic_routed_neighbor_probe

これは `routed` NIC が親のネットワークが利用可能かを調べるために IP 近傍探索するのを無効化できるようにします。

`ipv4.neighbor_probe` と `ipv6.neighbor_probe` の NIC 設定を追加します。未指定の場合のデフォルト値は `true` です。

event_hub

これは event-hub というクラスタメンバの役割と ServerEventMode 環境フィールドを追加します。

agent_nic_config

これを true に設定すると、仮想マシンの起動時に lxd-agent がインスタンスの NIC デバイスの名前と MTU を変更するための NIC 設定を適用します。

projects_restricted_intercept

restricted.container.intercept という設定キーを追加し通常は安全なシステムコールのインターセプションオプションを可能にします。

metrics_authentication

core.metrics_authentication というサーバ設定オプションを追加し /1.0/metrics のエンドポイントをクライアント認証無しでアクセスすることを可能にします。

images_target_project

コピー元とは異なるプロジェクトにイメージをコピーできるようにします。

cluster_migration_inconsistent_copy

POST /1.0/instances/<name> に allow_inconsistent フィールドを追加します。 true に設定するとクラスタメンバー間で不整合なコピーを許します。

cluster_ovn_chassis

ovn-chassis というクラスタロールを追加します。これはクラスタメンバーが OVN シャーシとしてどう振る舞うかを指定できるようにします。

container_syscall_intercept_sched_setscheduler

security.syscalls.intercept.sched_setscheduler を追加し、コンテナ内の高度なプロセス優先度管理を可能にします。

storage_lvm_thinpool_metadata_size

`storage.thinpool_metadata_size` により thinpool のメタデータボリュームサイズを指定できるようにします。

指定しない場合のデフォルトは LVM が適切な thinpool のメタデータボリュームサイズを選択します。

3.5.4 インスタンス〜ホスト間の通信

ホストされているワークロード (インスタンス) とそのホストのコミュニケーションは厳密には必要とされているわけではないですが、とても便利な機能です。

LXD ではこの機能は `/dev/lxd/sock` というノードを通して実装されており、このノードは全ての LXD のインスタンスに対して作成、セットアップされます。

このファイルはインスタンス内部のプロセスが接続できる Unix ソケットです。マルチスレッドで動いているので複数のクライアントが同時に接続できます。

実装詳細

ホストでは LXD は `/var/lib/lxd/devlxd/sock` をバインドして新しいコネクションのリッスンを開始します。

このソケットは、LXD が開始させたすべてのインスタンス内の `/dev/lxd/sock` に公開されます。

4096 を超えるインスタンスを扱うのに単一のソケットが必要です。そうでなければ、LXD は各々のインスタンスに異なるソケットをバインドする必要があり、ファイルディスクリプタ数の上限にすぐ到達してしまいます。

認証

`/dev/lxd/sock` への問い合わせは依頼するインスタンスに関連した情報のみを返します。リクエストがどこから来たかを知るために、 LXD は初期のソケットの ucred 構造体を取り出し、 LXD が管理しているインスタンスのリストと比較します。

プロトコル

`/dev/lxd/sock` のプロトコルは JSON メッセージを用いたプレーンテキストの HTTP であり、 LXD プロトコルのローカル版に非常に似ています。

メインの LXD API とは異なり、 `/dev/lxd/sock` API にはバックグラウンド処理と認証サポートはありません。

REST-API

API の構造

- /
  - /1.0
    - /1.0/config
      - /1.0/config/{key}
    - /1.0/devices
    - /1.0/events
    - /1.0/images/{fingerprint}/export
    - /1.0/meta-data

API の詳細

/

GET

- 説明: サポートされている API のリスト
- 出力: サポートされている API エンドポイント URL のリスト (デフォルトでは ['/1.0']`)

戻り値:

```
[
    "/1.0"
]
```

/1.0

GET

- 説明: 1.0 API についての情報
- 出力: dict 形式のオブジェクト

戻り値:

```
{
    "api_version": "1.0"
}
```

/1.0/config

GET

- 説明: 設定キーの一覧
- 出力: 設定キー URL のリスト

設定キーの名前はインスタンスの設定の名前と一致するようにしています。しかし、設定の namespace の全てが /dev/lxd/sock にエクスポートされているわけではありません。現在は cloud-init.* と user.* キーのみがインスタンスにアクセス可能となっています。

現時点ではインスタンスが書き込み可能な名前空間はありません。

戻り値:

```
[
    "/1.0/config/user.a"
]
```

/1.0/config/<KEY>

GET

- 説明: そのキーの値
- 出力: プレーンテキストの値

戻り値:

```
blah
```

/1.0/devices

GET

- 説明: インスタンスのデバイスのマップ
- 出力: dict

戻り値:

```
{
    "eth0": {
        "name": "eth0",
        "network": "lxdbr0",
        "type": "nic"
    },
    "root": {
        "path": "/",
        "pool": "default",
        "type": "disk"
    }
}
```

/1.0/events

GET

- 説明: この API ではプロトコルが websocket にアップグレードされます。
- 出力: 無し (イベントのフローが終わることがなくずっと続く)

サポートされる引数は以下の通りです。

- type: 購読する通知の種別のカンマ区切りリスト (デフォルトは all)

通知の種別には以下のものがあります。

- config (あらゆる user.* 設定キーの変更)
- device (あらゆるデバイスの追加、変更、削除)

この API は決して終了しません。それぞれの通知は別々の JSON の dict として送られます。

```
{
    "timestamp": "2017-12-21T18:28:26.846603815-05:00",
    "type": "device",
    "metadata": {
        "name": "kvm",
        "action": "added",
        "config": {
            "type": "unix-char",
            "path": "/dev/kvm"
```

(次のページに続く)

(前のページからの続き)

```
        }
    }
}
```


```
{
    "timestamp": "2017-12-21T18:28:26.846603815-05:00",
    "type": "config",
    "metadata": {
        "key": "user.foo",
        "old_value": "",
        "value": "bar"
    }
}
```


/1.0/images/<FINGERPRINT>/export

GET

- 説明: 公開されたあるいはキャッシュされたイメージをホストからダウンロードする
- 出力: 生のイメージあるいはエラー
- アクセス権: security.devlxd.images を true に設定する必要があります

戻り値:

```
LXD デーモン API の /1.0/images/<FINGERPRINT>/export を参照してください。
```

/1.0/meta-data

GET

- 説明: cloud-init と互換性のあるコンテナのメタデータ
- 出力: cloud-init のメタデータ

戻り値:

```
#cloud-config
instance-id: af6a01c7-f847-4688-a2a4-37fddd744625
local-hostname: abc
```

3.5.5 イベント

はじめに

イベントとは LXD 上で発生したアクションに関するメッセージです。 `/1.0/events` の API エンドポイントを直接使うか `lxc monitor` コマンドを使うことでウェブソケットに接続しログとライフサイクルメッセージがストリーム出力されます。

イベント種別

LXD は現在 3 つのイベント種別をサポートします。

- **Logging**: サーバのログレベルに関係なく全てのログメッセージを表示します。
- **Operation**: 作成から完了までの（状態と進捗メタデータの更新を含む）全ての実行中のオペレーションを表示します。
- **Lifecycle**: LXD 上で発生する特定のアクションの監査証跡を表示します。

イベント構造

例:

```
location: cluster_name
metadata:
  action: network-updated
  requestor:
    protocol: unix
    username: root
  source: /1.0/networks/lxdbr0
timestamp: "2021-03-14T00:00:00Z"
type: lifecycle
```

- `location`: クラスタメンバー名（クラスタであれば）。
- `timestamp`: RFC3339 形式のイベント発生時刻。
- `type`: イベント種別（`logging`, `operation`, `lifecycle` のいずれか）。
- `metadata`: 特定のイベント種別に関する情報。

logging イベントの構造

- message: ログメッセージ。
- level: ログのログレベル。
- context: イベントに含まれる追加情報。

operation イベントの構造

- id: オペレーションの UUID
- class: オペレーション種別（task, token, websocket のいずれか）。
- description: オペレーションの説明。
- created_at: オペレーションの作成日時。
- updated_at: オペレーションの更新日時。
- status: オペレーションの現在の状態。
- status_code: オペレーションのステータスコード。
- resources: このオペレーションで影響を受けるリソース。
- metadata: オペレーション特有のメタデータ。
- may_cancel: オペレーションがキャンセル可能か。
- err: オペレーションのエラーメッセージ。
- location: クラスタメンバー名（クラスタであれば）。

lifecycle イベントの構造

- action: 発生したライフサイクルアクション。
- requestor: 誰がリクエストを作成したかの情報（該当するものがあれば）。
- source: アクションの対象のパス。
- context: イベントに含まれる追加情報。

サポートされる lifecycle イベント

| 名前 | 説明 | 追加情報 | - | :---------- | :---------- | :---------- | | certificate-created | サーバのトラスト・ストアに新しい証明書が追加された。 | | | certificate-deleted | トラスト・ストアから証明書が削除された。 | | | certificate-updated | 証明書の設定が更新された。 | | | cluster-certificate-updated | クラスタ全体の証明書が変更された。 | | | cluster-disabled | このマシンに対してクラスタリングが無効化された。 | | | cluster-enabled | このマシンに対してクラスタリングが有効化された。 | | | cluster-member-added | 新しいマシンがクラスタに参加した。 | | | cluster-member-removed | クラスタからクラスタメンバーが削除された。 | | | cluster-member-renamed | クラスタメンバーがリネームされた。 | old_name: 以前の名前。 | | cluster-member-updated | クラスタメンバーの設定が変更された。 | | | cluster-token-created | クラスタメンバー追加のための参加トークンが作成された。 | | | config-updated | サーバの設定が変更された。 | | | image-alias-created | 既存イメージのエイリアスが作成された。 | target: オリジナルのインスタンス。 | | image-alias-deleted | 既存イメージのエイリアスが削除された。 | target: オリジナルのインスタンス。 | | image-alias-renamed | 既存イメージのエイリアスがリネームされた。 | old_name: 以前の名前。 | | image-alias-updated | イメージ・エイリアスの設定が変更された。 | target: オリジナルのインスタンス。 | | image-created | イメージ・ストアに新しいイメージが追加された。 | type: container か vm。 | | image-deleted | イメージ・ストアからイメージが削除された。 | | | image-refreshed | ローカルのイメージコピーが現在のソースイメージのバージョンに更新された。 | | | image-retrieved | raw イメージファイルがサーバからダウンロードされた。 | target: ダウンロード先のサーバ。 | | image-secret-created | イメージ取得用のワンタイム・キーが作成された。 | | | image-updated | イメージの設定が変更された。 | | | instance-backup-created | インスタンスのバックアップが作成された。 | | | instance-backup-deleted | インスタンスのバックアップが削除された。 | | | instance-backup-renamed | インスタンスのバックアップがリネームされた。 | old_name: 以前の名前。 | | instance-backup-retrieved | raw インスタンス・バックアップ・ファイルがダウンロードされた。 | | | instance-console | インスタンスのコンソールに接続された。 | type: console か vga。 | | instance-console-reset | コンソール・バッファーがリセットされた。 | | | instance-console-retrieved | コンソール・ログがダウンロードされた。 | | | instance-created | 新しいインスタンスが作成された。 | | | instance-deleted | インスタンスが削除された。 | | | instance-exec | インスタンス上でコマンドが実行された。 | command: 実行されたコマンド。 | | instance-file-deleted | インスタンス上のファイルが削除された。 | file: ファイルのパス。 | | instance-file-pushed | インスタンスにファイルがアップロードされた。 | file-source: ローカルのファイルパス。 file-destination: コピー先のファイルパス。 info: ファイルの情報。 | | instance-file-retrieved | インスタンスからファイルがダウンロードされた。 | file-source: インスタンスのファイルパス。 file-destination: コピー先のファイルパス。 | | instance-log-deleted | インスタンスの指定のログファイルが削除された。 | | | instance-log-retrieved | インスタンスの指定のログファイルがダウンロードされた。 | | | instance-metadata-retrieved | インスタンスのイメージメタデータがダウンロードされた。 | | | instance-metadata-updated | インスタンスのイメージメタデータが変更された。 | | | instance-metadata-template-created | インスタンスの新しいイメージテンプレートファイルが作成された。 | path: ファイルの相対パス。 | | instance-metadata-template-deleted | インスタンスのイメージテンプレートファイルが削除された。 | path: ファイルの相対パス。 | | instance-metadata-template-retrieved | インスタンスのイメージテンプレートファイルがダウンロードされた。 | path: ファイルの相対パス。 | | instance-paused

| インスタンスが休止状態にされた。 ||| instance-renamed | インスタンスがリネームされた。 | old_name: 以前の名前。 || instance-restarted | インスタンスが再起動された。 ||| instance-restored | インスタンスがスナップショットから復元された。 | snapshot: 復元されたスナップショット名。 || instance-resumed | インスタンスが休止状態から復帰された。 ||| instance-shutdown | インスタンスがシャットダウンされた。 || | instance-started | インスタンスが起動された。 ||| instance-stopped | インスタンスが停止された。 ||| instance-updated | インスタンスの設定が変更された。 ||| instance-snapshot-created | インスタンスのスナップショットが作成された。 ||| instance-snapshot-deleted | インスタンスのスナップショットが削除された。 ||| instance-snapshot-renamed | インスタンスのスナップショットがリネームされた。 | old_name: 以前の名前。 || instance-snapshot-updated | インスタンス・スナップショットの設定が変更された。 ||| network-acl-created | 新しいネットワーク ACL が作成された。 ||| network-acl-deleted | ネットワーク ACL が削除された。 ||| network-acl-renamed | ネットワーク ACL がリネームされた。 | old_name: 以前の名前。 || network-acl-updated | ネットワーク ACL の設定が変更された。 ||| network-created | ネットワークデバイスが作成された。 ||| network-deleted | ネットワークデバイスが削除された。 ||| network-renamed | ネットワークデバイスがリネームされた。 | old_name: 以前の名前。 || network-updated | ネットワークデバイスの設定が変更された。 ||| operation-cancelled | オペレーションがキャンセルされた。 ||| profile-created | 新しいプロファイルが作成された。 ||| profile-deleted | プロファイルが削除された。 ||| profile-renamed | プロファイルがリネームされた。 | old_name: 以前の名前。 || profile-updated | プロファイルの設定が変更された。 ||| project-created | 新しいプロジェクトが作成された。 ||| project-deleted | プロジェクトが削除された。 ||| project-renamed | プロジェクトがリネームされた。 | old_name: 以前の名前。 || project-updated | プロジェクトの設定が変更された。 ||| storage-pool-created | 新しいストレージプールが作成された。 | target: クラスタメンバー名。 || storage-pool-deleted | ストレージプールが削除された。 ||| storage-pool-updated | ストレージプールの設定が変更された。 | target: クラスタメンバー名。 || storage-volume-backup-created | ストレージボリュームの新しいバックアップが作成された。 | type: container, virtual-machine, image, custom のいずれか。 || storage-volume-backup-deleted | ストレージボリュームのバックアップが削除された。 || | storage-volume-backup-renamed | ストレージボリュームのバックアップがリネームされた。 | old_name: 以前の名前。 || storage-volume-backup-retrieved | ストレージボリュームのバックアップがダウンロードされた。 ||| storage-volume-created | 新しいストレージボリュームが作成された。 | type: container, virtual-machine, image, custom のいずれか。 || storage-volume-deleted | ストレージボリュームが削除された。 ||| storage-volume-renamed | ストレージボリュームがリネームされた。 | old_name: 以前の名前。 || storage-volume-restored | ストレージボリュームがスナップショットから復元された。 | snapshot: 復元されたスナップショット名。 || storage-volume-updated | ストレージボリュームの設定が変更された。 ||| storage-volume-snapshot-created | 新しいストレージボリュームスナップショットが作成された。 | type: container, virtual-machine, image, custom のいずれか。 || storage-volume-snapshot-deleted | ストレージボリュームのスナップショットが削除された。 ||| storage-volume-snapshot-renamed | ストレージボリュームのスナップショットがリネームされた。 | old_name: 以前の名前。 || storage-volume-snapshot-updated | ストレージボリュームのスナップショットの設定が変更された。 ||| warning-acknowledged | 警告の状態が "acknowledged" （確認済み）に設定された。 ||| warning-deleted | 警告が削除された。 ||| warning-reset | 警告の状態が "new" （新規）に設定された。 ||

3.5.6 インスタンスメトリクス

LXD は全ての実行中のインスタンスについてのメトリクスを提供します。これは CPU、メモリー、ネットワーク、ディスク、プロセスの使用量を含み、Prometheus で読み取って Grafana でグラフを表示するのに使うことを想定しています。クラスタ環境では、 LXD はアクセスされているサーバ上で稼働中のインスタンスの値だけを返します。各クラスタメンバーから別々にデータを取得する想定です。インスタンスメトリクスは `/1.0/metrics` エンドポイントを呼ぶと更新されます。メトリクスは複数のスクレイパーに対応するため 8 秒キャッシュします。メトリクスの取得は比較的重い処理ですので、影響が大きすぎるようならデフォルトの間隔より長い間隔でスクレイピングすることを検討してください。

メトリクス用証明書の作成

`1.0/metrics` エンドポイントは他の証明書に加えて `metrics` タイプの証明書を受け付けるという点で特別なエンドポイントです。このタイプの証明書はメトリクス専用で、インスタンスや他の LXD のオブジェクトの操作には使用できません。

新しい証明書は以下のように作成します（この手順はメトリクス用の証明書に限ったものではありません）。

```
openssl req -x509 -newkey ec -pkeyopt ec_paramgen_curve:secp384r1 -sha384 -keyout␣
↪metrics.key -nodes -out metrics.crt -days 3650 -subj "/CN=metrics.local"
```

作成後、証明書を信頼済みクライアントのリストに追加する必要があります。

```
lxc config trust add metrics.crt --type=metrics
```

Prometheus にターゲットを追加

Prometheus が LXD からメトリクスを取得するためには、 LXD をターゲットに追加する必要があります。

まず、 LXD にネットワーク越しにアクセスできるように `core.https_address` を設定しているかを確認してください。これは以下のコマンドを実行することで設定できます。

```
lxc config set core.https_address ":8443"
```

あるいは、メトリクス用途専用の `core.metrics_address` を使うことも出来ます。

次に、新しく作成した証明書と鍵を LXD のサーバ証明書とともに Prometheus からアクセスできるようにする必要があります。これは以下の 3 つのファイルを `/etc/prometheus/tls` にコピーすればできます。

```
# tls ディレクトリーを新規に作成
mkdir /etc/prometheus/tls
```

(次のページに続く)

(前のページからの続き)

```
# 新規に作成された証明書と鍵を tls ディレクトリーにコピー
cp metrics.crt metrics.key /etc/prometheus/tls

# LXD サーバ証明書を tls ディレクトリーにコピー
cp /var/snap/lxd/common/lxd/server.crt /etc/prometheus/tls

# これらのファイルを Prometheus が読めるようにする（通常 Prometheus は "prometheus" ユーザーで稼働しています）
chown -R prometheus:prometheus /etc/prometheus/tls
```

最後に、 LXD をターゲットに追加する必要があります。これは /etc/prometheus/prometheus.yaml を編集する必要があります。設定を以下のようにします。

```
scrape_configs:
  - job_name: lxd
    metrics_path: '/1.0/metrics'
    scheme: 'https'
    static_configs:
      - targets: ['127.0.0.1:8443']
    tls_config:
      ca_file: 'tls/lxd.crt'
      cert_file: 'tls/metrics.crt'
      key_file: 'tls/metrics.key'
```

3.6 内部動作とデバッグ

3.6.1 コンテナ実行環境

LXD は実行するコンテナに一貫性のある環境を提供しようとします。

正確な環境はカーネルの機能やユーザーの設定によって若干異なりますが、それ以外は全てのコンテナに対して同一です。

PID1

LXD は何であれ /sbin/init に置かれているものをコンテナの初期プロセス (PID 1) として起動します。このバイナリは親が変更されたプロセス (訳注: ゾンビプロセスなど) の処理を含めて適切な init システムとして振る舞う必要があります。

LXD がコンテナの PID1 とコミュニケーションするのは以下の 2 つのシグナルだけです。

- SIGINT コンテナのリブートをトリガーする
- SIGPWR (かあるいは SIGRTMIN+3) コンテナのクリーンなシャットダウンをトリガーする

PID1 の初期環境は container=lxc 以外は空です。 init システムは container=lxc をランタイムを検出する (訳注: lxc で動いていることを知る) ために使用できます。

デフォルトの 3 個 (訳注: stdin, stdout, stderr) より上の全てのファイルディスクリプタは PID1 が起動される前に閉じられます。

ファイルシステム

LXD は使用するどのイメージから生成する新規のコンテナは少なくとも以下のファイルシステムを含むことを前提とします。

- /dev (空のディレクトリ)
- /proc (空のディレクトリ)
- /sbin/init (実行ファイル)
- /sys (空のディレクトリ)

デバイス

LXD のコンテナは tmpfs ファイルシステムをベースとする最低限で一時的な /dev を持ちます。これは tmpfs であって devtmpfs ではないので、デバイスノードは手動で作成されたときのみ現れます。

デバイスノードの標準セットでは以下のデバイスがセットアップされます。

- /dev/console
- /dev/fd
- /dev/full
- /dev/log
- /dev/null
- /dev/ptmx

- /dev/random
- /dev/stdin
- /dev/stderr
- /dev/stdout
- /dev/tty
- /dev/urandom
- /dev/zero

標準セットのデバイスに加えて、以下のデバイスも利便性のためにセットアップされます。

- /dev/fuse
- /dev/net/tun
- /dev/mqueue

マウント

LXD では以下のマウントがデフォルトでセットアップされます。

- /proc (proc)
- /sys (sysfs)
- /sys/fs/cgroup/* (cgroupfs) (cgroup namespace サポートを欠くカーネルの場合のみ)

以下のパスがホスト上に存在する場合は自動的にマウントされます。

- /proc/sys/fs/binfmt_misc
- /sys/firmware/efi/efivars
- /sys/fs/fuse/connections
- /sys/fs/pstore
- /sys/kernel/debug
- /sys/kernel/security

これらを引き渡す理由は、これらがマウントされているか、コンテナ内でマウントできるようになっているかが必要とされているレガシーな init システムのためです。

これらのほとんどは非特権コンテナ内からは書き込み可能ではなく (あるいは読み取り可能ですらなく)、特権コンテナ内では LXD の AppArmor ポリシーによってブロックされます。

ネットワーク

LXD コンテナはネットワークデバイスをいくつでもアタッチできます。これらの名前はユーザーにオーバーライドされない限りは ethX で X は連番です。

コンテナからホストへのコミュニケーション

LXD は /dev/lxd/sock にソケットをセットアップし、コンテナ内の root ユーザーはこれを使ってホストの LXD とコミュニケーションできます。

API は **ここにドキュメント化されています**.

LXCFS

ホストに LXCFS がある場合は、コンテナ用に自動的にセットアップされます。

これは通常いくつかの /proc ファイルになり、それらは bind mount を通してオーバーライドされます。古いカーネルでは /sys/fs/cgroup の仮想バージョンも LXCFS によりセットアップされるかもしれません。

3.6.2 LXD でのライブマイグレーション

マイグレーションには 2 つの要素があります。1 つは「ソース」、つまり既にインスタンスを保持しているホストです。もう 1 つは「シンク」、インスタンスを受け取るホストです。現在、 pull モードでは、ソースが操作をセットアップし、シンクがソースに接続してインスタンスを pull します。

マイグレーションでは以下の 3 つの websocket (チャンネル) を使用します。

1. コントロール・ストリーム
2. criu イメージ・ストリーム
3. ファイルシステム・ストリーム

マイグレーションが開始されると、インスタンスに関する情報、インスタンスの設定などがコントロール・チャンネル上を流れます (このプロセスの完全な説明は後述します)。criu イメージとインスタンスのファイルシステムはそれぞれ個別のチャンネルを使って同期され、リストア操作の結果はシンクからソースにコントロール・チャンネル上で送られます。

特に、 criu チャンネルとファイルシステム・チャンネルの上で話されるプロトコルはコントロール・ソケット上で交渉されたものによって異なる場合があります。例えば、ソースとシンクの両方の LXD ディレクトリが btrfs 上にある場合、ファイルシステム・ソケットは btrfs の send/receive を話せます。さらに、現時点では我々は「ストップ・ザ・ワールド」タイプのマイグレーションを実行しますが、 criu の p.haul プロトコルはいつか criu ソケット上で実現されるでしょう。

コントロール・ソケット

2 つのエンドポイント間で 3 つの websocket が全て接続されたら、ソースは MigrationHeader (protobuf の記述が `/lxd/migration/migrate.proto` にあります) を送ります。このヘッダはインスタンスの設定を含んでおり、それは新しいインスタンスに追加されます。

話す予定のファイルシステムと criu のプロトコルを示す 2 つのフィールドもあります。例えば、サーバが btrfs ファイルシステム上にホストされている場合、単純な rsync の代わりに `btrfs send` を使いたいと示すことができます (同様に単にイメージを rsync で低速に転送する代わりに p.haul プロトコルを話したいと示すこともできるかもしれません)。

次にシンクはこのメッセージを調べてシンクがサポートするもので応答します。上の例を続けると、シンクが btrfs ファイルシステム上にない場合、最小公倍数 (この場合は rsync) で応答し、ソースはルート・ファイルシステムを rsync で送ることになります。同様に criu コネクションの例でシンクが p.haul プロトコル (や他の何か) をサポートしない場合は、 rsync にフォールバックします。

3.6.3 デーモンの動作

この仕様書は特定のシグナルに対する反応やクラッシュなどのデーモンの振る舞いの一部を取り扱います。

起動

起動する度に LXD はディレクトリ構造が存在することをチェックします。もし存在しない場合は、必要なディレクトリを作成し、キーペアを生成し、データベースを初期化します。

ひとたびデーモンが動作の準備が出来ると、 LXD はデータベース内のインスタンスのテーブルから対象のテーブルを検索し、電源状態が実際の状態と異なっていないかを確認します。もしインスタンスの電源状態が稼働中と記録されているのにインスタンスが稼働していない場合は LXD はそのインスタンスを開始します。

シグナル処理

SIGINT, SIGQUIT, SIGTERM

これらのシグナルについては LXD は一時的に停止し、後に再開してインスタンスの処理を継続することを想定しています。

インスタンスは稼働し続けて LXD は全ての接続を閉じ、クリーンな状態で終了するでしょう。

SIGPWR

LXD にホストがシャットダウンしようとしていることを伝えます。

LXD は全てのインスタンスをクリーンにシャットダウンしようと試みます。30 秒後、LXD は残りのインスタンスを kill します。

ホストがリブートを完了後に LXD がインスタンスを元の状態に戻せるように、データベース内のインスタンスのテーブルの `power_state` カラムにインスタンスの元の電源状態を記録しておきます。

SIGUSR1

メモリプロファイルを `--memprofile` で指定したファイルにダンプします。

3.6.4 データベース

イントロダクション

そもそも、なぜデータベースなのでしょう？

従来 LXC で行われていたように設定と状態をそれぞれのインスタンスのディレクトリに保存するのではなく、LXD ではそれら全ての情報を保管する内部的なデータベースを持っています。これによりすべてのインスタンスの設定に対する問い合わせをとても高速に行えます。

例えば、「どのインスタンスが br0 を使っているのか」というかなり分かりやすい問いがあります。この問いにデータベース無しで答えるとすると、 LXD は一つ一つのインスタンスに対して、設定を読み込んでパースし、そこにどのネットワークデバイスが定義されているかを見るということを繰り返し行わなければなりません。

インスタンスの数が少なければ、その処理は速いかもしれませんが、2000 個のインスタンスに対してどれだけ多くのファイルシステムへのアクセスが必要かを想像してみてください。代わりにデータベースを使うことで、非常に単純なクエリでキャッシュ済みのデータベースにアクセスするだけで良くなるのです。

データベースエンジン

LXD はクラスタリングをサポートし、クラスタの全てのメンバは同じデータベースの状態を共有する必要があるため、データベースエンジンは SQLite の分散対応バージョン をベースにしています。それは外部のデータベースのプロセスを必要とせずに、 レプリケーション、フォールトトレランス、自動フェールオーバーの機能を提供します。このデータベースを「グローバル」 LXD データベースと呼びます。

単一の非クラスタノードとして LXD を使う場合であっても、やはりグローバルデータベースを使用します。ただし、その場合は実質的には通常の SQLite データベースとして振る舞います。

グローバルデータベースのファイルは LXD のデータディレクトリ (例 `/var/lib/lxd/database/global` か snap ユーザーは `/var/snap/lxd/common/lxd/database/global`) の `./database/global` サブディレクトリの下に

格納されます。

クラスタの各メンバもそのメンバ固有の何らかのデータを保持する必要があるため、LXD は単なる SQLite のデータベース (「ローカル」データベース」も使用します。これは `./database/local.db` に置かれます。

アップグレードの前にはグローバルデータベースのディレクトリとローカルデータベースのファイルのバックアップが作成され、`.bak` のサフィックス付きでタグ付けされます。アップグレード前の状態に戻す必要がある場合は、このバックアップを使うことができます。

データベースのデータとスキーマをダンプする

データベースのデータまたはスキーマの SQL テキスト形式でのダンプを取得したい場合は、`lxd sql <local|global> [.dump|.schema]` コマンドを使ってください。これにより sqlite3 コマンドラインツールの `.dump` または `.schema` ディレクティブと同じ出力を生成できます。

コンソールからカスタムクエリを実行する

ローカルまたはグローバルデータベースに SQL クエリ (例 `SELECT`, `INSERT`, `UPDATE`) を実行する必要がある場合、`lxd sql` コマンドを使うことができます (詳細は `lxd sql --help` を実行してください)。

ただ、これが必要になるのは壊れたアップデートかバグからリカバーするときだけでしょう。その場合、まず LXD チームに相談してみてください (GitHub のイシュー を作成するかフォーラム に投稿)。

LXD デーモン起動時にカスタムクエリを実行する

SQL のデータマイグレーションのバグあるいは関連する問題のためにアップグレード後に LXD デーモンが起動に失敗する場合、壊れたアップデートを修復するクエリを含んだ `.sql` ファイルを作成することで、その状況からリカバーできます。

ローカルデータベースに対して修復を実行するには、修復に必要なクエリを含む`./database/patch.local.sql` というファイルを作成してください。同様にグローバルデータベースの修復には `./database/patch.global.sql` というファイルを作成してください。

これらのファイルはデーモンの起動シーケンスの非常に早い段階で読み込まれ、クエリが成功したときは削除されます (クエリは SQL トランザクション内で実行されるので、クエリが失敗したときにデータベースの状態が変更されることはありません)。

上記の通り、まず LXD チームに相談してみてください。

クラスタデータベースをディスクに同期

クラスタデータベースの内容をディスクにフラッシュしたいなら、`lxd sql global .sync` コマンドを使ってください。これは SQLite そのままの形式のデータベースのファイルを `./database/global/db.bin` に書き込みます。その後 `sqlite3` コマンドラインツールを使って中身を見ることが出来ます。

3.6.5 デバッグ

インスタンスの問題をデバッグする際の情報については、*FAQ* を参照してください。

lxc と lxd のデバッグ

`lxc` と `lxd` のコードをトラブルシューティングするのに役立ついくつかの異なる方法を説明します。

lxc --debug

クライアントのどのコマンドにも `--debug` フラグを追加することで内部についての追加情報を出力することができます。もし有用な情報がない場合はログ出力の呼び出しで追加することができます。

```
logger.Debugf("Hello: %s", "Debug")
```

lxc monitor

このコマンドはメッセージがリモートのサーバに現れるのをモニターします。

lxd --debug

`lxd` サーバを停止して `--debug` フラグでフォアグラウンドで実行することでたくさんの（願わくは）有用な情報が出力されます。

```
systemctl stop lxd lxd.socket
lxd --debug --group lxd
```

上記の `--group lxd` は非特権ユーザーにアクセス権限を与えるために必要です。

ローカルソケット経由での REST API

サーバサイドで LXD とやりとりするのに最も簡単な方法はローカルソケットを経由することです。以下のコマンドは GET /1.0 にアクセスし、jq ユーティリティを使って JSON を人間が読みやすいように整形します。

```
curl --unix-socket /var/lib/lxd/unix.socket lxd/1.0 | jq .
```

あるいは snap ユーザーの場合は

```
curl --unix-socket /var/snap/lxd/common/lxd/unix.socket lxd/1.0 | jq .
```

利用可能な API については *RESTful API* をご参照ください。

HTTPS 経由での REST API

LXD への *HTTPS* 接続には有効なクライアント証明書が必要です。証明書は初回に lxc remote add を実行したときに~/.config/lxc/client.crt に生成されます。この証明書は認証と暗号化のために接続ツールに渡す必要があります。

証明書の中身に興味がある場合は以下のコマンドで確認できます。

```
openssl x509 -in client.crt -purpose
```

コマンドの出力の中に以下の情報を読み取ることが出来るはずです。

```
Certificate purposes:
SSL client : Yes
```

コマンドラインツールを使う

```
wget --no-check-certificate https://127.0.0.1:8443/1.0 --certificate=$HOME/.config/lxc/
↪client.crt --private-key=$HOME/.config/lxc/client.key -O - -q
```

ブラウザを使う

いくつかのブラウザ拡張はウェブのリクエストを作成、修正、リプレイするための便利なインターフェースを提供しています。LXD サーバに対して認証するには lxc のクライアント証明書をインポート可能な形式に変換しブラウザにインポートしてください。

```
openssl pkcs12 -clcerts -inkey client.key -in client.crt -export -out client.pfx
```

上記のコマンドを実行し、（訳注：変換後の証明書をインポートしてから）ブラウザで https://127.0.0.1:8443/1.0 を開けば期待通り動くはずです。

3.6.6 環境変数

以下の環境変数を設定することで、 LXD のクライアントとデーモンをユーザーの環境に適合させることができ、いくつかの高度な機能を有効または無効にすることができます。

クライアントとサーバ共通の環境変数

名前	説明
LXD_DIR	LXD のデータディレクトリ
PATH	実行ファイルの検索対象のパスのリスト
http_proxy	HTTP 用のプロキシサーバの URL
https_proxy	HTTPS 用のプロキシサーバの URL
no_proxy	プロキシが不要なドメイン、IP アドレスあるいは CIDR レンジのリスト

クライアントの環境変数

名前	説明
EDITOR	使用するテキストエディタ
VISUAL	(EDITOR が設定されてないときに) 使用するテキストエディタ
LXD_CONF	LXC 設定ディレクトリーのパス
LXD_GLOBAL_CONF	LXC グローバル設定ディレクトリーのパス
LXC_REMOTE	使用するリモートの名前（設定されたデフォルトのリモートよりも優先されます）

サーバの環境変数

名前	説明
LXD_EXEC_PATH	(サブコマンド実行時に使用される) LXD 実行ファイルのフルパス
LXD_LXC_TEMPLATE_CONFIG	LXC テンプレート設定ディレクトリ
LXD_SECURITY_APPARMOR	false に設定すると AppArmor を無効にします
LXD_UNPRIVILEGED_ONLY	true に設定すると非特権コンテナしか作れなくなるように強制します。LXD_UNPRIVILEGED_ONLY を設定する前に作られた特権コンテナだけが引き続き特権を持つことに注意してください。このオプションを LXD デーモンを最初にセットアップするときに設定するのが実用的です。
LXD_OVMF_PATH	OVMF_CODE.fd と OVMF_VARS.ms.fd を含む OVMF ビルドへのパス
LXD_SHIFTFS_DISABLE	shiftfs サポートを無効にする（従来の UID シフトを試す際に有用です）
LXD_IDMAPPED_MOUNTS_DISABLE	idmapped マウントを無効にする（従来の UID シフトを試す際に有用です）
LXD_DEVMONITOR_DIR	fsnotify でモニターでモニターするパス。主にテスト用。

3.6.7 システムコールのインターセプション

LXD では非特権コンテナで、いくつか特定のシステムコールをインターセプトできます。もし、それが安全であると見なせるのであれば、ホスト上で特権を昇格させて実行します。

これを行うことで、対象のシステムコールではパフォーマンスに影響があり、LXD ではリクエストを評価するための作業が必要となり、もし許可されれば昇格した特権で実行されます。

特定のシステムコールインターセプションのオプションの有効化はコンテナの設定オプションを使ってコンテナ単位で行われます。

利用できるシステムコール

mknod と mknodat システムコールを使用して、色々なスペシャルファイルを作成できます。

もっとも一般的にはコンテナ内部で、ブロックデバイスやキャラクターデバイスを作成するために呼び出されます。このようなデバイスを作成することは、非特権コンテナ内では許可されません。これは、ディスクやメモリのようなリソースに直接書き込みのアクセスを許可することになり、特権を昇格するのに非常に簡単な方法であるためです。

しかし、作成しても安全であるファイルもあります。このような場合に、システムコールをインターセプトすることで、特定の処理のブロックが解除され、非特権コンテナ内部で実行できるようになります。

現時点で許可されているデバイスは次のものです:

- overlayfs whiteout (char 0:0)
- /dev/console (char 5:1)

- /dev/full (char 1:7)
- /dev/null (char 1:3)
- /dev/random (char 1:8)
- /dev/tty (char 5:0)
- /dev/urandom (char 1:9)
- /dev/zero (char 1:5)

キャラクターデバイス以外のすべてのファイルタイプは、現時点では通常通りカーネルに送られるので、この機能を有効にしても動作は全く変わりません。

この機能は `security.syscalls.intercept.mknod` を `true` に設定することで有効に出来ます。

bpf

カーネル内の eBPF プログラムを管理するために `bpf` システムコールを使用します。これらは様々なカーネルサブシステムにアタッチされます。

一般に、信頼していない eBPF プログラムをロードするのはタイミングベースの攻撃を容易にするので問題です。

LXD の eBPF サポートは現在のところデバイスの cgroup エントリを管理するプログラムに限定しています。有効にするには `security.syscalls.intercept.bpf` と `security.syscalls.intercept.bpf.devices` の両方を true に設定する必要があります。

mount

`mount システムコールは物理と仮想ファイルシステムの両方のマウントを可能にします。デフォルトでは、非特権コンテナはカーネルにより制限され、いくつかの仮想とネットワークファイルシステムのみに限定されています。

物理ファイルシステムをマウントできるようにするにはシステムコールインターセプションが使えます。LXD はこれを取り扱うために様々な選択肢を提供しています。

`security.syscalls.intercept.mount` は全体の機能を制御するのに使用され、他のいずれかの選択肢を機能させるためには有効にする必要があります。

`security.syscalls.intercept.mount.allowed` はコンテナ内に直接マウント可能なファイルシステムのリストを指定できます。これはユーザが信頼できないデータをカーネルに送り込むことを許すため最も危険な選択肢です。これにより簡単にホストシステムをクラッシュさせたり攻撃が出来てしまいます。そのため信頼された環境でのみ使うようにすべきです。

`security.syscalls.intercept.mount.shift` は上記に加えてコンテナで使用される UID/GID マップでマウントした結果をシフトさせるのに使用できます。これは非特権コンテナ内で全てが nobody/nogroup として表示されるのを回避するために必要です。

これらよりもっと安全な代替は security.syscalls.intercept.mount.fuse でファイルシステムの名前と FUSE ハンドラの組を指定できます。これが設定されると指定されたファイルシステムのどれかをマウントしようとするとそのファイルシステムに対応する FUSE ハンドラ呼び出しにリダイレクトされます。

これは全て呼び出し側として実行されるので、カーネルのアタックサーフェスにまつわる全ての問題を回避し、そのため一般的に安全と考えられます。しかし、あらゆる種類のシステムコールインターセプションはホストシステムに過大な負荷をかける簡単な方法になることを留意しておくべきです。

sched_setscheduler

sched_setscheduler システムコールはプロセスの優先度を管理するのに使えます。

これを許可するとユーザが自分のプロセスの優先度を著しく上げることを許すため、潜在的に多くのシステムリソースを使われることになります。

これはまた SCHED_FIFO のようなスケジューラへのアクセスを許すので、一般的には欠陥と考えられ、システム全体の安定性に著しく影響を与える可能性があります。このため通常の状況下では、真の root ユーザ (あるいはグローバルの CAP_SYS_NICE) のみがこれの使用を許すべきです。

setxattr

setxattr システムコールは、拡張ファイル属性を設定するのに使われます。

現時点で、これにより処理される属性は次のものです:

- trusted.overlay.opaque (overlayfs directory whiteout)

この介入は多数の文字列で行う必要があるため、現在のところ、対象の少数の属性のみインターセプトする簡単な方法がありません。上記の属性のみを許可しているため、カーネルが以前に許可していた他の属性を破損する可能性があります。

この機能は security.syscalls.intercept.setxattr を true に設定することで有効にできます。

3.6.8 ユーザー名前空間 (user namespace) 用の ID のマッピング

LXD は安全なコンテナを実行します。これは主にユーザー・ネームスペースの使用によって実現されています。ユーザー・ネームスペースはコンテナを非特権で実行することを可能にし、攻撃対象を大幅に限定します。

ユーザー・ネームスペースはコンテナの uid と gid の組をホストの uid と gid の組にマッピングすることで機能します。

例えば、 100000 から 165535 までのホストの uid と gid を LXD が使用できるようにし、コンテナで 0 から 65535 までの uid/gid にマッピングするように設定できます。

この結果、コンテナ内で 0 の uid で動くプロセスが実際には uid 100000 で動くことになります。

root (0) と nobody (65534) の POSIX の範囲をカバーするため、割当は必ず最低 65535 個の uid と gid であるべきです。

カーネルのサポート

ユーザー・ネームスペースの使用にはカーネル 3.12 以上が必要です。 LXD は古いカーネルでも起動しますが、コンテナを起動するのは拒否します。

使用可能な範囲

ほとんどのホストでは、 LXD は初回起動時に "lxd" ユーザーの割当のために`/etc/subuid` と `/etc/subgid` をチェックし、そこで指定されている範囲の最初の 65536 個の uid と gid をデフォルト・プロファイルで使用するように設定します。

範囲が 65536 より小さい場合 (範囲が全く無い場合を含む)、これが修正されるまで LXD はコンテナの作成と起動に失敗します。

`/etc/subuid` 、 `/etc/subgid` 、 `newuidmap` (パスを検索)、 `newgidmap` (パスを検索) のいくつか (ただし全部ではない) がシステムに存在する場合、これは shadow の設定が間違っていることを示しているので、これが修正されるまで LXD はコンテナの起動に失敗します。

これらのファイルが 1 つも無い場合、 LXD は 1000000 の基点の uid/gid から開始する 1000000000 の uid/gid の範囲を想定します。

これは最もよくあるケースであり、完全に非特権なコンテナをホストするシステム上で稼働するのではない場合（コンテナランタイム自身はユーザー権限で実行するような場合）に、通常は推奨される設定です。

ホスト間で異なる範囲の使用

ホスト間でコンテナを移動する時、送信側のマッピングが送られるので、受信側のホストで異なる範囲にマッピング可能です。

コンテナ毎に異なる ID マッピング

コンテナを他のコンテナからより一層隔離するために、 LXD はコンテナ毎に異なる ID マッピングを使用することをサポートしています。これはコンテナ毎に `security.idmap.isolated` と `security.idmap.size` という 2 つの設定項目で制御できます。

`security.idmap.isolated` が設定されたコンテナは `security.idmap.isolated` が設定された他のコンテナと衝突しないユニークな ID の範囲を持つように設定されます (もしそのようなコンテナが 1 つも存在しない場合、このキーを設定しようとしても失敗します)。

`security.idmap.size` が設定されたコンテナはこのサイズに ID の範囲が設定されます。このプロパティが設定されていない隔離されたコンテナは ID の範囲がデフォルトのサイズ 65536 に設定されます。これにより POSIX

に準拠し、コンテナ内で"nobody" ユーザーが使用できます。

特定のマッピングを選択するには `security.idmap.base` を設定すると自動検出機構をオーバーライドし、コンテナでベースとして使用したいホストの uid/gid を LXD に伝えることができます。

これらのプロパティを反映するにはコンテナの再起動が必要です。

カスタムの ID マッピング

さらに LXD は ID マッピングの一部をカスタマイズすることをサポートします。例えば、uid を変更するファイルシステムを必要とせずに、ホストのファイルシステムの一部をコンテナに bind mount することをユーザーに許可できます。このためのコンテナ毎の設定項目は `raw.idmap` で、設定例は以下のようになります。

```
both 1000 1000
uid 50-60 500-510
gid 100000-110000 10000-20000
```

1 行目は、ホストの uid と gid 1000 の両方をコンテナ内の uid 1000 にマッピングする設定です (これは例えばユーザーのホームディレクトリをコンテナ内に bind mount するのに使用できます)。

2 行目と 3 行目は uid または gid のどちらかだけをコンテナ内にマッピングする設定です。行の中の 2 番目のエントリーはソース ID 、 つまりホスト上の ID で、 3 番目のエントリーはコンテナ内部での範囲です。これらの範囲は同じサイズでなければなりません。

このプロパティを反映するにはコンテナの再起動が必要です。

3.7 外部リソース