はじめに

DockerEngineユーザガイド〜活用編

このガイドについて

Docker Engine ドキュメント（https://docs.docker.com/）の日本語翻訳版（http://docs.docker.jp/）をもとに電子データ化しました。

現時点ではベータ版であり、内容に関しては無保証です。PDF の評価用として公開しました。Docker は活発な開発と改善が続いています。同様に、ドキュメントもオリジナルですら日々書き換えが進んでいますので、あらかじめご了承願います。

このドキュメントに関する問題や翻訳に対するご意見がありましたら、GitHub リポジトリ上の Issue までご連絡いただくか、pull request をお願いいたします。

- https://github.com/zembutsu/docs.docker.jp

履歴

- 2016 年 6 月 4 日　Docker Engine 活用編 beta1 を公開

目次

アプリケーションの Docker 化

6.1 MongoDB の Docker 化

この例では、MongoDB(モンゴデービー)[*1] がインストール済みの Docker イメージを、どのようにして構築するかを学びます。また、イメージを Docker Hub レジストリに送信して、他人と共有する方法も理解します。

このガイドでは MongoDB コンテナを構築する仕組みを紹介しますが、Docker Hub の公式イメージ[*2] を使っても構いません。

Docker を使い MongoDB インスタンスをデプロイしたら、次のメリットがあります。

- 簡単なメンテナンス、MongoDB インスタンスの高い設定性
- ミリ秒以内で実行と開始
- どこからでも接続可能で共有できるイメージに基づく

構築用の `Dockerfile` を作成しましょう。

```
$ nano Dockerfile
```

オプションですが、`Dockerfile` の冒頭に自身の役割などをコメントしておくと便利です。

```
# MongoDB の Docker 化：MongoDB イメージを構築する Dockerfile
# ubuntu:latest をベースとし、MongoDB は以下の手順でインストール：
# http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/
```

`Dockerfile` は柔軟性がありますが、特定の書式に従う必要があります。最初に必要になるのは、これから作成する MongoDB を Docker 化したイメージの親にあたるイメージ名の定義です。

*1 https://www.mongodb.org/

*2 https://hub.docker.com/_/mongo/

Docker Hub Ubuntu リポジトリ[*1] にある Ubuntu の レイテスト バージョンを使い、イメージを構築します。

```
# 書式：FROM　　リポジトリ[:バージョン]
FROM　　ubuntu:latest
```

続けて、Dockerfile の メ ン テ ナ を宣言します。

```
# 書式：MAINTAINER 名前　<email@addr.ess>
MAINTAINER M.Y. Name <myname@addr.ess>
```

Ubuntu システムにも MongoDB パッケージがありますが、作成日が古いものです。そのため、この例では公式の MongoDB パッケージを使います。

MongoDB 公開 GPG 鍵を取り込みます。また、パッケージ・マネージャ用に MongoDB リポジトリ・ファイルも作成します。

```
# インストール：
# MongoDB 公開 GPG 鍵を取り込み、MongoDB リストファイルを作成
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" \
    | tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

この初期準備が終われば、パッケージを更新し、MongoDB をインストールできます。

```
# apt-get ソースを更新し、MongoDB をインストール
RUN apt-get update && apt-get install -y mongodb-org
```

MongoDB のバージョンを指定したインストールできます。そのためには、次の例のようにパッケージのバージョン番号のリストが必要です。

```
RUN apt-get update && apt-get install -y mongodb-org=3.0.1 mongodb-org-server=3.0.1 \
 mongodb-org-shell=3.0.1 mongodb-org-mongos=3.0.1 mongodb-org-tools=3.0.1
```

MongoDB はデータ・ディレクトリが必要です。インストール命令ステップで作成を命令しましょう。

```
# MongoDB データ・ディレクトリの作成
RUN mkdir -p /data/db
```

最後に ENTRYPOINT を設定します。これは Docker に対して MongoDB イメージでコンテナを起動する時、コンテナ内で mongod を実行するよう命令します。そして、ポートを公開するために EXPOSE 命令を使います。

```
# コンテナのポート 27017 をホスト側に露出（EXPOSE）
```

*1 https://hub.docker.com/_/ubuntu/

```
EXPOSE 27017

# usr/bin/mongod を Docker 化アプリケーションのエントリ・ポイントに設定
ENTRYPOINT ["/usr/bin/mongod"]
```

ファイルを保存したら、イメージを構築しましょう。

この Dockerfile の完成版はこちら[*1] をご覧ください。

6.1.2 MongoDB Docker イメージの構築

作成した Dockerfile を使い、新しい MongoDB イメージを Docker で構築できます。テスト用でない限り、docker build コマンドに --tag オプションを通して Docker イメージをタグ付けするのが良い手法です。

```
# 書式：docker build --tag/-t <ユーザ名>/<リポジトリ>
# 例
$ docker build --tag my/repo .
```

コマンドを実行したら、Docker は Dockerfile を処理してイメージを構築します。イメージは最終的に my/repo とタグ付けされます。

全ての Docker イメージ・リポジトリを Docker Hub で保管・共有できるようにするには、docker push コマンドを使います。送信するためには、ログインの必要があります。

```
# ログイン
$ docker login
Username:
..

# イメージを送信
# 書式：docker push <ユーザ名>/<リポジトリ>
$ docker push my/repo
The push refers to a repository [my/repo] (len: 1)
Sending image list
Pushing repository my/repo (1 tags)
..
```

6.1.4 MongoDB イメージを使う

作成した MongoDB イメージを使い、他の MongoDB インスタンスをデーモン・プロセスとして実行できます。

```
# 基本的な方法
# 使い方：docker run --name <コンテナ名> -d <ユーザ名>/<リポジトリ>
$ docker run -p 27017:27017 --name mongo_instance_001 -d my/repo

# Docker 化した Mongo DB 、学び理解しました！
# 使い方：docker run --name <コンテナ名> -d <ユーザ名>/<リポジトリ> --noprealloc --smallfiles
```

*1 https://github.com/docker/docker/blob/master/docs/examples/mongodb/Dockerfile

```
$ docker run -p 27017:27017 --name mongo_instance_001 -d my/repo --smallfiles

# MongoDB コンテナのログを確認
# 使い方：docker logs <コンテナ名>
$ docker logs mongo_instance_001

# MongoDB を使う
# 使い方：mongo --port <`docker ps` で得られるポート>
$ mongo --port 27017

# If using docker-machine
# docker-machine を使う場合
# 使い方：mongo --port <`docker ps` で得られるポート> --host <`docker-machine ip VM 名`の IP アドレス>
$ mongo --port 27017 --host 192.168.59.103
```

ちなみに、もし同じエンジン上で２つのコンテナを実行したい場合、ホスト側は２つの異なったポートを各コンテナに割り当てる必要があります。

```
# ２つのコンテナを起動し、ポートを割り当て
$ docker run -p 28001:27017 --name mongo_instance_001 -d my/repo
$ docker run -p 28002:27017 --name mongo_instance_002 -d my/repo

# 各 MongoDB インスタンスのポートに接続できる
$ mongo --port 28001
$ mongo --port 28002
```

6.2 PostgreSQL の Docker 化

6.1.2 Docker に PostgreSQL をインストール

ここでは Docker Hub で配布されている Docker イメージを使っていますが、自分自身で作成しても構いません。

まず、新しい Dockerfile を作成します。

この PostgreSQL のセットアップは、開発用途専用です。より安全にするためには、PostgreSQL のドキュメントを参照し、適切に調整ください。

```
#
# https://docs.docker.com/examples/postgresql_service/ にあるサンプルの Dockerfile です
#

FROM ubuntu
MAINTAINER SvenDowideit@docker.com

# Add the PostgreSQL PGP key to verify their Debian packages.
# It should be the same key as https://www.postgresql.org/media/keys/ACCC4CF8.asc
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8

# Add PostgreSQL's repository. It contains the most recent stable release
#     of PostgreSQL, ``9.3``.
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" >
/etc/apt/sources.list.d/pgdg.list

# Install ``python-software-properties``, ``software-properties-common`` and PostgreSQL 9.3
#  There are some warnings (in red) that show up during the build. You can hide
#  them by prefixing each apt-get statement with DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y python-software-properties software-properties-common
postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3

# Note: The official Debian and Ubuntu images automatically ``apt-get clean``
# after each ``apt-get``

# Run the rest of the commands as the ``postgres`` user created by the ``postgres-9.3`` package when it was
``apt-get installed``
USER postgres

# Create a PostgreSQL role named ``docker`` with ``docker`` as the password and
# then create a database `docker` owned by the ``docker`` role.
# Note: here we use ``&&\`` to run commands one after the other - the ``\``
#       allows the RUN command to span multiple lines.
RUN    /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

# Adjust PostgreSQL configuration so that remote connections to the
# database are possible.
RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/9.3/main/pg_hba.conf

# And add ``listen_addresses`` to ``/etc/postgresql/9.3/main/postgresql.conf``
```

```
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

# Expose the PostgreSQL port
EXPOSE 5432

# Add VOLUMEs to allow backup of config, logs and databases
VOLUME  ["/etc/postgresql", "/var/log/postgresql", "/var/lib/postgresql"]

# Set the default command to run when starting the container
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D", "/var/lib/postgresql/9.3/main", "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

Dockerfile で構築するイメージに名前を割り当てます。

```
$ docker build -t eg_postgresql .
```

それから PostgreSQL サーバコンテナを（フォアグラウンドで）実行します。

```
$ docker run --rm -P --name pg_test eg_postgresql
```

PostgreSQL サーバに接続するには２つの方法があります。 コンテナをリンクするか、ホスト側（あるいはネットワーク側）から接続できます。

--rm

クライアントの docker run 時に -link リモート名:ローカル・エイリアス を指定時、コンテナが他のコンテナのポートに直接接続できるようになります。これは接続のために使える複数の環境変数を作成します。

```
$ docker run --rm -t -i --link pg_test:pg eg_postgresql bash

postgres@7ef98b1b7243:/$ psql -h $PG_PORT_5432_TCP_ADDR -p $PG_PORT_5432_TCP_PORT -d docker -U docker
--password
```

ホストシステムから接続

postgresql クライアントがインストールされていれば、ホスト側に割り当てられたポートに対しても、同様にテストできます。docker ps でコンテナがどこのポートに割り当てられているか確認します。

```
$ docker ps
CONTAINER ID        IMAGE                  COMMAND                CREATED             STATUS
   PORTS                                      NAMES
5e24362f27f6        eg_postgresql:latest   /usr/lib/postgresql/   About an hour ago   Up About an hour
  0.0.0.0:49153->5432/tcp                     pg_test
$ psql -h localhost -p 49153 -d docker -U docker --password
```

データベースのテスト

認証すると docker =# プロンプトが表示され、テーブル作成を処理できます。

```
psql (9.3.1)
Type "help" for help.

$ docker=# CREATE TABLE cities (
docker(#     name            varchar(80),
docker(#     location        point
docker(# );
CREATE TABLE
$ docker=# INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
INSERT 0 1
$ docker=# select * from cities;
     name      | location
---------------+-----------
 San Francisco | (-194,53)
(1 row)
```

コンテナ・ボリュームを使う

PostgreSQL のログファイル調査や、設定やデータのバックアップのために、ボリュームを定義できます。

```
$ docker run --rm --volumes-from pg_test -t -i busybox sh

/ # ls
bin      etc      lib      linuxrc  mnt      proc     run      sys      usr
dev      home     lib64    media    opt      root     sbin     tmp      var
/ # ls /etc/postgresql/9.3/main/
environment      pg_hba.conf      postgresql.conf
pg_ctl.conf      pg_ident.conf    start.conf
/tmp # ls /var/log
ldconfig    postgresql
```

6.3 CouchDB サービスの Docker 化

２つの CouchDB コンテナ間で同じデータを共有するため、ここでは例としてデータ・ボリュームを使います。これはホット・アップグレード、同じデータを使った異なった CouchDB バージョンのテストなどに便利です。

/var/lib/couchdb をデータ・ボリュームとして作成するのに注意してください。

```
$ COUCH1=$(docker run -d -p 5984 -v /var/lib/couchdb shykes/couchdb:2013-05-03)
```

Docker ホストは到達可能な localhost を想定しています。もしそうでなければ、localhost を Docker ホストのパブリック IP アドレスに置き換えてください。

```
$ HOST=localhost
$ URL="http://$HOST:$(docker port $COUCH1 5984 | grep -o '[1-9][0-9]*$')/_utils/"
$ echo "Navigate to $URL in your browser, and use the couch interface to add data"
```

今回は$COUCH1 のボリュームに対する共有アクセスをリクエストします。

```
$ COUCH2=$(docker run -d -p 5984 --volumes-from $COUCH1 shykes/couchdb:2013-05-03)
```

```
$ HOST=localhost
$ URL="http://$HOST:$(docker port $COUCH2 5984 | grep -o '[1-9][0-9]*$')/_utils/"
$ echo "Navigate to $URL in your browser. You should see the same data as in the first database"'!'
```

おめでとうございます。２つの Couchdb コンテナを実行し、お互いのデータを完全に隔離しました。

6.4 Couchbase サービスの Docker 化

この例では Docker Compose を使い Couchbase[*1] サーバを起動し、 REST API[*2] を使えるように設定し、結果を確認します。

Couchbase はオープンソースです。そして、最近のウェブ、モバイル、IoT アプリケーション向けのドキュメント指向 NoSQL データベースです。簡単な開発とインターネットで性能をスケールできるように設計されています。

Couchbase Docker イメージは Docker Hub 上[*3] で公開されています。
Couchbase サーバは次のように起動します：

```
docker run -d --name db -p 8091-8093:8091-8093 -p 11210:11210 couchbase
```

各ポートを公開する理由は、Couchbase Developer Portal - Network Configuration [*4] をご覧ください。

ログには次のように表示されます：

```
docker logs db
Starting Couchbase Server -- Web UI available at http://<ip>:8091
```

このページの例では、接続先の Docker ホストの IP アドレスは 192.168.99.100 を前提にしています。192.168.88.100 は実際の Docker ホストの IP アドレスに置き換えてください。Docker Machine で Docker を実行している場合は、次のコマンドで IP アドレスを確認できます。
docker-machine ip <マシン名>

Couchbase コンソールには `http://192.168.99.100:8091` でアクセスできます。デフォルトのユーザ名は `Administrator` 、パスワードは `password` です。

通常は Couchbase サーバを使う前にコンソール上での設定が必要です。これを REST API を使えば簡単に設定できます。

Couchbase インスタンス上では、Data・Query・Index は別々のサービスです。各サービスは別々の設定が必要です。例えば、Query は CPU の処理が集中するため、より速い CPU が必要です。 Index はディスクが重いため、速い SSD が必要です。Data は読み書きを速くするため、より多くのメモリが必要です。

メモリが必要になる設定は Data と Index サービスのみです。

*1 http://couchbase.com/

*2 http://developer.couchbase.com/documentation/server/4.0/rest-api/rest-endpoints-all.html

*3 https://hub.docker.com/_/couchbase/

*4 http://developer.couchbase.com/documentation/server/4.1/install/install-ports.html

```
curl -v -X POST http://192.168.99.100:8091/pools/default -d memoryQuota=300 -d indexMemoryQuota=300
* Hostname was NOT found in DNS cache
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8091 (#0)
> POST /pools/default HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 192.168.99.100:8091
> Accept: */*
> Content-Length: 36
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 36 out of 36 bytes
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Basic realm="Couchbase Server Admin / REST"
* Server Couchbase Server is not blacklisted
< Server: Couchbase Server
< Pragma: no-cache
< Date: Wed, 25 Nov 2015 22:48:16 GMT
< Content-Length: 0
< Cache-Control: no-cache
<
* Connection #0 to host 192.168.99.100 left intact
```

これは REST エンドポイント `/pools/default` に HTTP POST リクエストを送信した結果です。ホストとは Docker Machine の IP アドレスです。ポートは Couchbase サーバによって公開されているものです。サーバに対して、メモリとインデックスに対する制限（quota）をリクエストしています。

3 つの全サービス、または、1 つに対しての設定が可能です。これにより、それぞれのアフィニティ（ハードウェア要件等)、サービスを適切にセットアップします。例えば、Data サービスが開始できるのは、Docker ホストが SSD マシン上で動作している場所といった指定です。

```
curl -v http://192.168.99.100:8091/node/controller/setupServices -d 'services=kv%2Cn1ql%2Cindex'
* Hostname was NOT found in DNS cache
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8091 (#0)
> POST /node/controller/setupServices HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 192.168.99.100:8091
> Accept: */*
> Content-Length: 26
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 26 out of 26 bytes
< HTTP/1.1 200 OK
* Server Couchbase Server is not blacklisted
< Server: Couchbase Server
< Pragma: no-cache
< Date: Wed, 25 Nov 2015 22:49:51 GMT
< Content-Length: 0
< Cache-Control: no-cache
<
* Connection #0 to host 192.168.99.100 left intact
```

これは REST エンドポイント /node/controller/setupServices に HTTP POST リクエストを送信した結果です。コマンドの結果は、Couchbase サーバ用に３つのサービスが設定されています。Data サービスは kv、Query サービスは n1ql、Index サービスは index なのが分かります。

あとで Couchbase サーバを管理するため、ユーザ名とパスワードの認証情報を設定します。

```
curl -v -X POST http://192.168.99.100:8091/settings/web -d port=8091 -d username=Administrator -d
password=password
* Hostname was NOT found in DNS cache
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8091 (#0)
> POST /settings/web HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 192.168.99.100:8091
> Accept: */*
> Content-Length: 50
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 50 out of 50 bytes
< HTTP/1.1 200 OK
* Server Couchbase Server is not blacklisted
< Server: Couchbase Server
< Pragma: no-cache
< Date: Wed, 25 Nov 2015 22:50:43 GMT
< Content-Type: application/json
< Content-Length: 44
< Cache-Control: no-cache
<
* Connection #0 to host 192.168.99.100 left intact
{"newBaseUri":"http://192.168.99.100:8091/"}
```

これは REST エンドポイント /settings/web に HTTP POST リクエストを送信した結果です。ユーザ名とパスワードの認証情報がリクエスト中に含まれています。

サンプル・データのインストール

Couchbase サーバは couchbase インスタンス内で簡単にサンプル・データを読み込めます。

```
curl -v -u Administrator:password -X POST http://192.168.99.100:8091/sampleBuckets/install -d
'["travel-sample"]'
* Hostname was NOT found in DNS cache
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8091 (#0)
* Server auth using Basic with user 'Administrator'
> POST /sampleBuckets/install HTTP/1.1
> Authorization: Basic QWRtaW5pc3RyYXRvcjpwYXNzd29yZA==
> User-Agent: curl/7.37.1
> Host: 192.168.99.100:8091
> Accept: */*
> Content-Length: 17
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 17 out of 17 bytes
```

```
< HTTP/1.1 202 Accepted
* Server Couchbase Server is not blacklisted
< Server: Couchbase Server
< Pragma: no-cache
< Date: Wed, 25 Nov 2015 22:51:51 GMT
< Content-Type: application/json
< Content-Length: 2
< Cache-Control: no-cache
<
* Connection #0 to host 192.168.99.100 left intact
[]
```

これは REST エンドポイント `/sampleBuckets/install` に HTTP POST リクエストを送信した結果です。サンプル・バケット名をリクエスト中に指定します。

おつかれさまでした。Couchbase コンテナの設定を、全て REST API を使って行いました。

CBQ を使って Couchbase に問い合わせ

CBQ[*1] は Couchbase への問い合わせを省略するコマンドライン・ツールです。これは Couchbase サーバに対して JSON ドキュメントの作成・読み込み・更新・削除が可能です。ツールは Couchbase Docker イメージに同梱されています。

CBQ ツールの実行：

```
docker run -it --link db:db couchbase cbq --engine http://db:8093
Couchbase query shell connected to http://db:8093/ . Type Ctrl-D to exit.
cbq>
```

`--engine` パラメータは、CBQ に Docker ホスト上で動いている Couchbase サーバのホストとポートを指定します。ホストとは、通常、Couchbase サーバを実行しているホストの名前もしくは IP アドレスです。今回の例では、コンテナを起動時に指定したコンテナ名 `db` とポート `8093` が全てのクエリを受け付けます。

Couchbase には N1QL[*2] を使う JSON ドキュメントで問い合わせます。N1QL は包括的な宣言型クエリ言語であり、JSON ドキュメントに SQL のような機能を持たせます。

N1QL クエリを使ってデータベースに問い合わせます：

```
cbq> select * from `travel-sample` limit 1;
{
    "requestID": "97816771-3c25-4a1d-9ea8-eb6ad8a51919",
    "signature": {
        "*": "*"
    },
    "results": [
        {
            "travel-sample": {
                "callsign": "MILE-AIR",
                "country": "United States",
```

*1 http://developer.couchbase.com/documentation/server/4.1/cli/cbq-tool.html

* 2 http://developer.couchbase.com/documentation/server/4.1/n1ql/n1ql-language-reference/index.html

```
                    "iata": "Q5",
                    "icao": "MLA",
                    "id": 10,
                    "name": "40-Mile Air",
                    "type": "airline"
                }
            }
        ],
        "status": "success",
        "metrics": {
            "elapsedTime": "60.872423ms",
            "executionTime": "60.792258ms",
            "resultCount": 1,
            "resultSize": 300
        }
    }
```


6.4.3 Couchbase ウェブ・コンソール

Couchbase ウェブ・コンソール[*1] は Couchbase インスタンスを管理できるコンソールです。次の URL で表示します。

http://192.168.99.100:8091/

この IP アドレスの部分は Docker Machine の IP アドレスか、ローカルで動かしている場合は `localhost` です。

*1 http://developer.couchbase.com/documentation/server/4.1/admin/ui-intro.html

6.5 Node.js ウェブ・アプリの Docker 化

この例のゴールは、Dockerfile を使い、親イメージから自分の Docker イメージを構築できるようにする方法を理解します。ここでは CentOS 上で簡単な Node.js の hello world ウェブ・アプリケーションを実行します。ソースコード全体は https://github.com/enokd/docker-node-hello/ から入手できます。

6.5.1 Node.js アプリの作成

まず、全てのファイルを置く src ディレクトリを作成します。それから package.json ファイルを作成し、アプリケーションと依存関係について記述します。

```
{
  "name": "docker-centos-hello",
  "private": true,
  "version": "0.0.1",
  "description": "Node.js Hello world app on CentOS using docker",
  "author": "Daniel Gasienica <daniel@gasienica.ch>",
  "dependencies": {
    "express": "3.2.4"
  }
}
```

次に index.js ファイルを作成し、ウェブアプリが Express.js[*1] フレームワークを使うように定義します。

```
var express = require('express');

// Constants
var PORT = 8080;

// App
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

次のステップでは、Docker が CentOS コンテナの中で、どのようにこのアプリを実行するかを理解していきます。まず、自分のアプリを動かす Docker イメージを作成します。

Dockerfile という名称の空ファイルを作成します。

```
touch Dockerfile
```

好みのエディタで Dockerfile を開きます。

*1 http://expressjs.com/

自分のイメージの構築に使いたい親イメージを定義します。ここでは Docker Hub 上で利用可能な CentOS[*1]（タグ：centos6）を使います。

```
FROM    centos:centos6
```

Node.js アプリを作るために、CentOS イメージ上に Node.js と npm をインストールします。アプリケーションの実行には Node.js が必要です。また、package.json で定義したアプリケーションをインストールするために npm も必要です。CentOS 用の適切なパッケージをインストールするため、Node.js wiki[*2] の指示に従って作業します。

```
# Enable Extra Packages for Enterprise Linux (EPEL) for CentOS
RUN     yum install -y epel-release
# Install Node.js and npm
RUN     yum install -y nodejs npm
```

npm バイナリでアプリケーションの依存関係をインストールします。

```
# Install app dependencies
COPY package.json /src/package.json
RUN cd /src; npm install --production
```

アプリケーションのソースコードを Docker イメージに取り込むため、COPY 命令を使います。

```
# Bundle app source
COPY . /src
```

アプリケーションはポート 8080 を利用のため、EXPOSE 命令を使い docker デーモンがポートを割り当てるようにします。

```
EXPOSE  8080
```

最後にあと少し、実行時にアプリケーションを実行できるよう CMD 命令でコマンドを定義します。例えば node と、アプリケーション、例えば src/index.js を指定します（ソースファイルは前の手順でコンテナに加えていました)。

```
CMD ["node", "/src/index.js"]
```

これで Dockerfile は次のようになります。

*1 https://registry.hub.docker.com/_/centos/

* 2 https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager#rhelcentosscientific-linux-6

```
FROM    centos:centos6

# Enable Extra Packages for Enterprise Linux (EPEL) for CentOS
RUN     yum install -y epel-release
# Install Node.js and npm
RUN     yum install -y nodejs npm

# Install app dependencies
COPY package.json /src/package.json
RUN cd /src; npm install --production

# Bundle app source
COPY . /src

EXPOSE  8080
CMD ["node", "/src/index.js"]
```

6.5.3 イメージを構築

Dockerfile のあるディレクトリに移動し、Docker イメージを構築するため次のコマンドを実行します。-t フラグを使いイメージにタグを付ければ、あとから docker images コマンドで簡単に探せます。

```
$ docker build -t <自分のユーザ名>/centos-node-hello .
```

作成したイメージは、Docker のイメージ一覧に表示されます。

```
$ docker images

# Example
REPOSITORY                          TAG         ID              CREATED
centos                              centos6     539c0211cd76    8 weeks ago
<your username>/centos-node-hello   latest      d64d3505b0d2    2 hours ago
```

イメージに -d を付けて実行したら、コンテナはデタッチド・モードで動作します。これは、コンテナをバックグラウンドで動作するものです。 -p フラグで、コンテナ内のプライベートなポートを公開ポートに渡します。

```
$ docker run -p 49160:8080 -d <自分のユーザ名>/centos-node-hello
```

アプリケーションの出力を表示します。

```
# Get container ID
$ docker ps

# Print app output
$ docker logs <container id>

# Example
Running on http://localhost:8080
```

6.5.5 テスト

アプリケーションをテストするには、Docker でアプリケーションにポートを割り当てます。

```
$ docker ps

# Example
ID            IMAGE                                  COMMAND             ...   PORTS
ecce33b30ebf  <your username>/centos-node-hello:latest  node /src/index.js        49160->8080
```

この例は、Docker はコンテナのポート 8080 をポート 49160 に割り当てます。

これで curl を使ってアプリケーションを呼び出せます（インストールの必要があれば sudo apt-get install curl を実行します。 ）。

```
$ curl -i localhost:49160

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Sun, 02 Jun 2013 03:53:22 GMT
Connection: keep-alive

Hello world
```

OS X 上で Docker Machine を使っている場合は、ポートが実際に割り当てられているのは Docker ホストの VM 側であり、次のコマンドを使う必要があります。

```
$ curl $(docker-machine ip VM_NAME):49160
```

私たちはこのチュートリアルが、Docker 上で Node.js と CentOS を動かす手助けになるのを望みます。全てのソースコードは https://github.com/enokd/docker-node-hello/ にあります。

6.6 Redis サービスの Docker 化

非常にシンプルで飾り気の無い Redis サービスを、リンク機能を使ってウェブ・アプリケーションにアタッチします。

6.6.1 Redis 用の Docker コンテナを作成

まず、新しい Redis イメージ用の Dockerfile を作成します。

```
FROM        ubuntu:14.04
RUN         apt-get update && apt-get install -y redis-server
EXPOSE      6379
ENTRYPOINT  ["/usr/bin/redis-server"]
```

次に、Dockerfile からイメージを構築します。 <自分の名前> の場所は、自分自身のリポジトリ名に置き換えてください。

```
$ docker build -t <自分のユーザ名>/redis .
```

先ほど作成したイメージを使い、コンテナ名を redis とします。
コンテナに -d を付けて実行したら、デタッチド・モードとして実行され、コンテナはバックグラウンドで動作します。
重要なのは、コンテナ内のポートを全く公開していない点です。そのかわり、Redis データベースに接続するには、コンテナに対するリンク機能を使います。

```
$ docker run --name redis -d <自分の名前>/redis
```

6.6.3 ウェブ・アプリケーションのコンテナを作成

次にアプリケーションのコンテナを作成します。 --link フラグを使い、redis コンテナに対するリンクを作成します。ここでは db というエイリアス（別名）で作成します。これにより、redis コンテナと安全なトンネルが作成されます。Redis インスタンスが公開しているコンテナ内のポートには、ここで指定したコンテナだけ接続できるようになります。

```
$ docker run --link redis:db -i -t ubuntu:14.04 /bin/bash
```

新しく作成したコンテナの中では、接続をテストするために redis-cli バイナリの取得・インストールが必要です。

```
$ sudo apt-get update
$ sudo apt-get install redis-server
$ sudo service redis-server stop
```

それから --link redis:db オプションを使い、Docker がウェブ・アプリケーションのコンテナ内で利用可能な環境変数を作成します。

```
$ env | grep DB_

# Should return something similar to this with your values
DB_NAME=/violet_wolf/db
DB_PORT_6379_TCP_PORT=6379
DB_PORT=tcp://172.17.0.33:6379
DB_PORT_6379_TCP=tcp://172.17.0.33:6379
DB_PORT_6379_TCP_ADDR=172.17.0.33
DB_PORT_6379_TCP_PROTO=tcp
```

ここでは DB が接頭語となっている複数の環境変数が見えます。DB とはコンテナ起動時に指定した、リンクのエイリアスです。DB_PORT_6379_TCP_ADDR を使って Redis コンテナに接続してみましょう。

```
$ redis-cli -h $DB_PORT_6379_TCP_ADDR
$ redis 172.17.0.33:6379>
$ redis 172.17.0.33:6379> set docker awesome
OK
$ redis 172.17.0.33:6379> get docker
"awesome"
$ redis 172.17.0.33:6379> exit
```

ウェブ・アプリケーションが redis コンテナに接続するために、この環境変数や他の環境変数を利用できます。

6.7 Riak の Docker 化

この例は、Riak がインストール済みの Docker イメージをどのように構築するかを紹介するのが目的です。

Dockerfile という名称の空ファイルを作成します。

```
$ touch Dockerfile
```

次に、自分のイメージを構築するにあたり、元になる親イメージを定義します。ここでは Docker Hub で利用可能な Ubuntu[*1]（タグ：trusty）を使います。

```
# Riak
#
# VERSION       0.1.1

# Use the Ubuntu base image provided by dotCloud
FROM ubuntu:trusty
MAINTAINER Hector Castro hector@basho.com
```

次は、curl をインストールします。curl はリポジトリのセットアップ・スクリプトをダウンロードし、実行するためです。

```
# Install Riak repository before we do apt-get update, so that update happens
# in a single step
RUN apt-get install -q -y curl && \
    curl -fsSL https://packagecloud.io/install/repositories/basho/riak/script.deb | sudo bash
```

それから、いくつかの依存関係のインストールとセットアップをします。

- supervisor は Riak プロセスの管理に使用
- riak-2.0.5-1 は Riack バージョン 2.0.5 のパッケージ

```
# プロジェクトの依存関係をインストール・セットアップ
RUN apt-get update && \
    apt-get install -y supervisor riak=2.0.5-1

RUN mkdir -p /var/log/supervisor

RUN locale-gen en_US en_US.UTF-8

COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

それから、Riak の設定を変更します。

```
# Riak が、あらゆるホストから接続できるよう設定
```

*1 https://hub.docker.com/_/ubuntu/

```
RUN sed -i "s|listener.http.internal = 127.0.0.1:8098|listener.http.internal = 0.0.0.0:8098|" /etc/riak/riak.conf
RUN sed -i "s|listener.protobuf.internal = 127.0.0.1:8087|listener.protobuf.internal = 0.0.0.0:8087|" /etc/riak/riak.conf
```

それから、Riak プロトコル・バッファ用ポートと HTTP インターフェースを公開します。

```
# Riack プロトコル・バッファと HTTP インターフェースの露出
EXPOSE 8087 8098
```

最後に supervisord を実行し、Riack を開始します。

```
CMD ["/usr/bin/supervisord"]
```

supervisord.conf という空のファイルを作成します。Dockerfile がある同じディレクトリかどうか確認してください。

```
touch supervisord.conf
```

以下のプログラム定義を投入します。

```
[supervisord]
nodaemon=true

[program:riak]
command=bash -c "/usr/sbin/riak console"
numprocs=1
autostart=true
autorestart=true
user=riak
environment=HOME="/var/lib/riak"
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s.log
```

6.7.3 Riak 用の Docker イメージを構築

これで Riak 用の Docker イメージを構築できます。

```
$ docker build -t "<自分のユーザ名>/riak" .
```

次のステップ

Riak は分散データベースです。多くのプロダクションへのデプロイには、 少なくとも 5 ノードが必要[*1] と考えられています。docker-riak プロジェクト[*2] に、Docker と Pipework を使った Riak クラスタのデプロイ方法の詳細があります。

*1 http://basho.com/why-your-riak-cluster-should-have-at-least-five-nodes/

*2 https://github.com/hectcastro/docker-riak

6.8 SSH デーモン用サービスの Docker 化

6.8.1 eg_sshd イメージの構築

以下の Dockerfile はコンテナ内に SSHd サービスをセットアップします。これは他のコンテナ用ボリュームの調査や、テスト用コンテナに対する迅速なアクセスを提供します。

```
# sshd
#
# VERSION               0.0.2

FROM ubuntu:14.04
MAINTAINER Sven Dowideit <SvenDowideit@docker.com>

RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd
RUN sed -i 's/PermitRootLogin without-password/PermitRootLogin yes/' /etc/ssh/sshd_config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /etc/pam.d/sshd

ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile

EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

イメージを構築するには：

```
$ docker build -t eg_sshd .
```

次は実行します。docker port を使い、コンテナのポート 22 がホスト側のどのポートに割り当て（マップ）されているか確認できます。

```
$ docker run -d -P --name test_sshd eg_sshd
$ docker port test_sshd 22
0.0.0.0:49154
```

それから、コンテナに 対して root として SSH できます。接続先は、IP アドレス（これは docker inspect で確認できます）か、Docker デーモンの IP アドレス（ip address や ifconfig で確認できます ）上のポート 49154 か、Docker デーモンのホスト localhost です。

```
$ ssh root@192.168.1.2 -p 49154
# パスワードは ``screencast`` です。
$$
```

6.8.3 環境変数

sshd デーモンでシェルを呼び出すのは複雑です。シェルを起動する前に sshd 環境を調整し、 環境変数をユーザのシェルから通常の Docker のメカニズムに対して渡す必要があるためです。

値を設定するためには、Dockerfile で ENV を使います。先ほどの Dockerfile の例では、シェルの初期化のために /etc/profile を送る必要があるでしょう。

docker run -e ENV=value で値を渡す必要がある場合は、同様の処理を行うスクリプトを準備する必要があります。そのためには sshd -D を CMD 命令に置き換え、その準備したスクリプトを実行します。

6.8.4 クリーンアップ

最後に、テストの後でコンテナの停止と削除をし、そのイメージを削除します。

```
$ docker stop test_sshd
$ docker rm test_sshd
$ docker rmi eg_sshd
```

6.9 apt-cacher-ng サービスの Docker 化

複数の Docker サーバを持っている場合や、Docker 構築キャッシュを使わない Docker コンテナを構築する場合は、パッケージ用のキャッシュ・プロキシを使うと便利です。このコンテナは、パッケージを既にダウンロード済みの環境から二次ダウンロードできるようにします。

以下の Dockerfile を使います。

```
#
# Build: docker build -t apt-cacher .
# Run: docker run -d -p 3142:3142 --name apt-cacher-run apt-cacher
#
# and then you can run containers with:
#   docker run -t -i --rm -e http_proxy http://dockerhost:3142/ debian bash
#
# Here, `dockerhost` is the IP address or FQDN of a host running the Docker daemon
# which acts as an APT proxy server.
FROM        ubuntu
MAINTAINER  SvenDowideit@docker.com

VOLUME      ["/var/cache/apt-cacher-ng"]
RUN     apt-get update && apt-get install -y apt-cacher-ng

EXPOSE      3142
CMD     chmod 777 /var/cache/apt-cacher-ng && /etc/init.d/apt-cacher-ng start && \
  tail -f /var/log/apt-cacher-ng/*
```

イメージを構築するには、次のようにします。

```
$ docker build -t eg_apt_cacher_ng .
```

それから実行します。公開されたポートをホスト側に割り当てます。

```
$ docker run -d -p 3142:3142 --name test_apt_cacher_ng eg_apt_cacher_ng
```

ログファイルを参照します。デフォルトのコマンドに `-f` （末尾を表示）オプションを付けます。

```
$ docker logs -f test_apt_cacher_ng
```

Debian をベースとしたコンテナで proxy を使うには、以下の手順とオプションを進めます。

1. apt プロキシ設定を追加します。

```
echo 'Acquire::http { Proxy "http://dockerhost:3142"; };' >> /etc/apt/conf.d/01proxy
```

2. 環境変数を設定します： `http_proxy=http://dockerhost:3142/`
3. `sources.list` エントリを変更し、 `http://dockerhost:3142/` から始めるようにします。
4. `--link` を使って APT proxy コンテナを Debian ベースのコンテナにリンクします。
5. Debian ベースのコンテナで、APT proxy コンテナに接続するカスタム・ネットワークを作成します。

オプション１：apt 設定を安全に行うには、共通の基盤となるバージョンで apt 設定を行う方法があります。

```
FROM ubuntu
RUN  echo 'Acquire::http { Proxy "http://dockerhost:3142"; };' >> /etc/apt/apt.conf.d/01proxy
RUN apt-get update && apt-get install -y vim git

# docker build -t my_ubuntu .
```

オプション２：http_proxy 設定はテストに便利ですが、curl と wget のような HTTP クライアントでは動作しない場合があります。

```
$ docker run --rm -t -i -e http_proxy=http://dockerhost:3142/ debian bash
```

オプション３：これは最新版を取り入れるためですが、 Dockerfile では何度が記述が必要になるかもしれません。

オプション４：Debian コンテナを proxy サーバに次のコマンドでリンクします。

```
$ docker run -i -t --link test_apt_cacher_ng:apt_proxy -e http_proxy=http://apt_proxy:3142/ debian bash
```

オプション５：APT　proxy サーバと Debian ベースのコンテナが接続するカスタム・ネットワークを作成します。

```
$ docker network create mynetwork
$ docker run -d -p 3142:3142 --net=mynetwork --name test_apt_cacher_ng eg_apt_cacher_ng
$ docker run --rm -it --net=mynetwork -e http_proxy=http://test_apt_cacher_ng:3142/ debian bash
```

apt-cacher-ng はリポジトリを管理するのと同じツールを持っています。VOLUME 命令を使い、サービスを実行するイメージを構築します。

```
$ docker run --rm -t -i --volumes-from test_apt_cacher_ng eg_apt_cacher_ng bash

$$ /usr/lib/apt-cacher-ng/distkill.pl
Scanning /var/cache/apt-cacher-ng, please wait...
Found distributions:
bla, taggedcount: 0
     1. precise-security (36 index files)
     2. wheezy (25 index files)
     3. precise-updates (36 index files)
     4. precise (36 index files)
     5. wheezy-updates (18 index files)

Found architectures:
     6. amd64 (36 index files)
     7. i386 (24 index files)

WARNING: The removal action may wipe out whole directories containing
         index files. Select d to see detailed list.

(Number nn: tag distribution or architecture nn; 0: exit; d: show details; r: remove tagged; q: quit): q
```

最後に、コンテナのテストが終わったら、クリーンアップのためにコンテナを停止・削除し、イメージを削除します。

```
$ docker stop test_apt_cacher_ng
$ docker rm test_apt_cacher_ng
$ docker rmi eg_apt_cacher_ng
```

7章

Docker Engine 管理

Docker 1.2 から が Docker の機能に組み込まれました。これはコンテナ終了時に再起動するための仕組みです。再起動ポリシーを設定しておけば、Docker デーモンの起動時、典型的なのはシステムの起動時に自動的にコンテナを開始します。リンクされたコンテナであっても、再起動ポリシーは適切な順番で起動します。

必要に応じて再起動ポリシーを使わないでください（例：Docker ではないプロセスが Docker コンテナに依存する場合)。そのような場合は、再起動ポリシーの代わりに upstart、systemd、supervisor といったプロセス・マネージャをお使いください。

7.1.1 プロセス・マネージャを使う場合

デフォルトの Docker は再起動ポリシーを設定しません。しかし、多くのプロセス・マネージャと衝突を引き起こす可能性については知っておいてください。もしプロセス・マネージャを使うのであれば、再起動ポリシーを使わない方が良いでしょう。

イメージのセットアップが完了したら、コンテナを実行できるようになり満足でしょう。この実行をプロセス・マネージャに委ねられます。 `docker start -a` を実行したら、Docker は自動的に実行中のコンテナにアタッチします。そして、実行後は必要に応じて全てのシグナルを転送しますので、コンテナの停止をプロセス・マネージャが検出したら、適切に再起動するでしょう。

以下は systemd と upstart を Docker と連携する例を紹介します。

7.1.2 例

この例では２つの有名なプロセス・マネージャ、upstart と systemd 向けの設定ファイルを扱います。これらの例では、既に作成しているコンテナ `--name=redis_server` を使って Redis を起動するのを想定しています。これらのファイルは新しいサービスを定義し、docker サービスが起動した後で自動的に起動するものです。

upstart

```
description "Redis container"
author "Me"
start on filesystem and started docker
stop on runlevel [!2345]
respawn
script
  /usr/bin/docker start -a redis_server
end script
```

systemd

```
[Unit]
Description=Redis container
Requires=docker.service
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker start -a redis_server
ExecStop=/usr/bin/docker stop -t 2 redis_server

[Install]
WantedBy=local.target
```

redis コンテナに（ --env のような）オプションを渡したい場合は、 docker run に代わって docker start を使う必要があります。次の例は、起動したコンテナのサービスが停止、または、サービス停止によってコンテナが削除されたとしても、新しいコンテナを毎回作成します。

```
[Service]
...
ExecStart=/usr/bin/docker run --env foo=bar --name redis_server redis
ExecStop=/usr/bin/docker stop -t 2 redis_server ; /usr/bin/docker rm -f redis_server
...
```

多くの Linux ディストリビューションは systemd を使って Docker デーモンを起動します。このドキュメントは、様々な Docker の設定例を紹介します。

Docker をインストールしたら、Docker デーモンを起動する必要があります。

```
$ sudo systemctl start docker
# 他のディストリビューションでは、次のように実行します
$ sudo service docker start
```

また、Docker をブート時に自動起動するには、次のように実行すべきです。

```
$ sudo systemctl enable docker
# 他のディストリビューションでは、次のように実行します
$ sudo chkconfig docker on
```

7.2.2 Docker デーモンのオプション変更

Docker デーモンの設定を変更するには、多くのフラグを使う方法と、環境変数を使う方法があります。

推奨する方法は、systemd 用のファイルを使うことです。ローカルの設定ファイルは /etc/systemd/system/docker.service.d ディレクトリにあります。あるいは/etc/systemd/system/docker.service かもしれません。これは /lib/systemd/system/docker.service にあるデフォルト設定を上書きします。

一方で、既にパッケージを使ってインストールしていた場合は、環境設定ファイル（通常は /etc/sysconfig/docker ） があるかもしれません。これは後方互換性のためです。このファイルの内容は、/etc/systemd/system/docker.service.d ディレクトリにあるファイルに落とし込めます。

```
[Service]
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// $OPTIONS \
          $DOCKER_STORAGE_OPTIONS \
          $DOCKER_NETWORK_OPTIONS \
          $BLOCK_REGISTRY \
          $INSECURE_REGISTRY
```

docker.service が環境設定ファイルを使っているか確認します。

```
$ systemctl show docker | grep EnvironmentFile
EnvironmentFile=-/etc/sysconfig/docker (ignore_errors=yes)
```

あるいは、サービス用のファイルがどこにあるか探します。

```
$ systemctl status docker | grep Loaded
FragmentPath=/usr/lib/systemd/system/docker.service
```

```
$ grep EnvironmentFile /usr/lib/systemd/system/docker.service
EnvironmentFile=-/etc/sysconfig/docker
```

Docker デーモンのオプションは、以下の HTTP Proxy 例で説明するようなファイルを使って上書き可能です。このファイルは /usr/lib/systemd/system か /lib/systemd/system にありますが、デフォルトのオプション設定は変更すべきではありません。

Docker イメージ、コンテナ、ボリュームを別々のパーティションのディスク・スペースで管理したくなるでしょう。

この例では、次のような docker.service ファイルがあるものとします。

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network.target docker.socket
Requires=docker.socket

[Service]
Type=notify
ExecStart=/usr/bin/docker daemon -H fd://
LimitNOFILE=1048576
LimitNPROC=1048576
TasksMax=1048576

[Install]
Also=docker.socket
```

これはドロップイン・ファイル（先ほど扱いました）を経由して外部フラグを追加できます。以下の内容を含むファイルを /etc/systemd/system/docker.service.d に作成します。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// --graph="/mnt/docker-data" --storage-driver=overlay
```

このファイルに他の環境変数も設定できます。例えば、HTTP_PROXY 環境変数を下に追加することもできるでしょう。

ExecSart 設定を変更するには、空の設定の次の行に、新しい設定を追加します。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// --bip=172.17.42.1/16
```

空の設定があると失敗しますので、次のように表示されるでしょう。

```
docker.service has more than one ExecStart= setting, which is only allowed for Type=oneshot services.
Refusing.
```

HTTP プロキシ

この例はデフォルトの docker.service ファイルを上書きします。

HTTP プロキシサーバの背後にいる場合、ここではオフィスで設定する例として、Docker の systemd サービス・ファイルに設定を追加する必要があるものとします。

まず、docker サービス向けの systemd ドロップイン・ディレクトリを作成します。

```
mkdir /etc/systemd/system/docker.service.d
```

次は /etc/systemd/system/docker.service.d/http-proxy.conf ファイルを作成し、 HTTP_PROXY 環境変数を追加します。

```
[Service]
Environment="HTTP_PROXY=http://proxy.example.com:80/"
```

内部の Docker レジストリがあれば、プロキシを通さずに通信できるようにするため、 NO_PROXY 環境変数を指定します。

```
Environment="HTTP_PROXY=http://proxy.example.com:80/"
"NO_PROXY=localhost,127.0.0.1,docker-registry.somecorporation.com"
```

設定を反映します。

```
$ sudo systemctl daemon-reload
```

設定ファイルが読み込まれたのを確認します。

```
$ systemctl show --property=Environment docker
Environment=HTTP_PROXY=http://proxy.example.com:80/
```

Docker を再起動します。

```
$ sudo systemctl restart docker
```

7.2.3 systemd ユニットファイルの手動作成

パッケージを使わずにバイナリをインストールした場合でも、Docker と systemd を連動したくなるでしょう。簡単に実現するには、単純に GitHub リポジトリ にある２つのユニットファイル[*1]（サービスとソケット用）を /etc/systemd/system に置くだけです。

*1 https://github.com/docker/docker/tree/master/contrib/init/systemd

7.3 PowerShell DSC で使う

Windows PowerShell DSC (Desired State Configuration) は設定管理ツールです。これは Windows PowerShell の機能を拡張します。DSC は宣言型の構文を使いターゲットがどのような状態になるかを設定します。PowerShell DSC に関する詳しい情報は、Microsoft 社のサイト[*1] をご覧ください。

7.3.1 動作条件

このガイドの利用にあたり、Windows ホストは PowerShell v4.0 以上の必要があります。

DSC 設定に含まれるスクリプトは、公式では Ubuntu ターゲットのみサポートされています。 Ubuntu ターゲットは OMI サーバと PowerShell DSC for Linux providers のインストールが必要です。詳しい情報は https://github.com/MSFTOSSMgmt/WPSDSCLinux をご覧ください。ソース・リポジトリの一覧に、PowerShell DSC for Linux のインストール方法や、初期化スクリプトに関するより詳しい情報があります。

7.3.2 インストール

DSC 設定例のソースは次のリポジトリ https://github.com/anweiss/DockerClientDSC から利用可能です。クローンも可能です。

```
$ git clone https://github.com/anweiss/DockerClientDSC.git
```

7.3.3 使い方

DSC 設定はシェルスクリプトのセットを使い、どこに Docker の構成物を置くかや、ターゲット・ノードの設定を行います。ソース・リポジトリはスクリプト（RunDockerClientConfig.ps1 があり)、CIM セッションに必要な接続と、 Set-DscConfiguration cmdlet の実行に使います。

より詳細な情報は次の URL をご覧ください。https://github.com/anweiss/DockerClientDSC

Docker インストール

```
apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys\
36A1D7869245C8950F966E92D8576A8BA88D21E9
sh -c "echo deb https://apt.dockerproject.org/repo ubuntu-trusty main\
> /etc/apt/sources.list.d/docker.list"
apt-get update
apt-get install docker-engine
```

現在の作業ディレクトリが DockerClientDSC ソースに設定されていることを確認し、Docker クライアント設定を現在の PowerShell セッションに反映します。

```
. .\DockerClient.ps1
```

ターゲット・ノード用の DSC 設定に必要な .mof ファイルを生成します。

```
DockerClient -Hostname "myhost"
```

*1 https://technet.microsoft.com/ja-jp/library/dn249912.aspx?f=255&mspperror=-2147217396

サンプルの DSC 設定データ・ファイルは、 Hostname パラメータの情報を元に連結されます。

```
DockerClient -ConfigurationData .\DockerConfigData.psd1
```

ターゲット・ノードでアプリケーション設定プロセスを開始します。

```
.\RunDockerClientConfig.ps1 -Hostname "myhost"
```

RunDockerClientConfig.ps1 スクリプトは DSC 設定データファイルもパースして、次のように複数のノードに対して設定を反映します。

```
.\RunDockerClientConfig.ps1 -ConfigurationData .\DockerConfigData.psd1
```

7.3.4 イメージ

イメージ設定とは docker pull [image] あるいは docker rmi -f [IMAGE] 処理と同等です。
先ほどのステップで定義したファイルを使い、DockerClient の Image パラメータで設定を追加します。

```
DockerClient -Hostname "myhost" -Image "node"
.\RunDockerClientConfig.ps1 -Hostname "myhost"
```

ホストに対して複数のイメージを取得する設定も可能です。

```
DockerClient -Hostname "myhost" -Image "node","mongo"
.\RunDockerClientConfig.ps1 -Hostname "myhost"
```

イメージを削除するには、次のようにハッシュ・テーブルを使います。

```
DockerClient -Hostname "myhost" -Image @{Name="node"; Remove=$true}
.\RunDockerClientConfig.ps1 -Hostname $hostname
```

7.3.5 コンテナ

コンテナの設定は次のように行います。

```
docker run -d --name="[containername]" -p '[port]' -e '[env]' --link '[link]'\
'[image]' '[command]'
```

あるいは

```
docker rm -f [containername]
```

コンテナを作成・削除するには、 1 つまたは複数コンテナをハッシュ・テーブルで指定します。ハッシュ・テーブルは次のプロパティのパラメータを指定します。

- Name（必須）

- Image（Remove プロパティが $true の以外は必要）
- Port
- Env
- Link
- Command
- Remove

例えば、ハッシュテーブルの設定でコンテナを作成するには、次のようにします。

```
$webContainer = @{Name="web"; Image="anweiss/docker-platynem"; Port="80:80"}
```

それから、先補との定義と同じ手順で DockerClient に -Image と -Container パラメータを使います。

```
DockerClient -Hostname "myhost" -Image node -Container $webContainer
.\RunDockerClientConfig.ps1 -Hostname "myhost"
```

既存のコンテナは次のように削除できます。

```
$containerToRemove = @{Name="web"; Remove=$true}
DockerClient -Hostname "myhost" -Container $containerToRemove
.\RunDockerClientConfig.ps1 -Hostname "myhost"
```

このハッシュテーブルは全てのパラメータを使い、コンテナを作成しています。

```
$containerProps = @{Name="web"; Image="node:latest"; Port="80:80"; `
Env="PORT=80"; Link="db:db"; Command="grunt"}
```

7.4 CFEngine でプロセス管理

プロセスを管理する Docker コンテナを作成します。

Docker は各実行中のコンテナで 1 つのプロセスを監視します。そして、コンテナの生死とは、そのプロセスの生死を指します。Docker コンテナ内での CFEngine（シーエフエンジン）の動かし方を紹介します。これは、次の問題を軽減するものです。

- 1 つのコンテナの中で、複数のコンテナを簡単に起動できるかもしれません。通常の `docker run` コマンドを実行するだけで、全てを自動的に管理します。
- 停止またはクラッシュしたプロセスを管理し、1 分以内に CFEngine が対象プロセスを再起動します。
- CFEngine スケジューリング・デーモン（cf-execd）が動いている限り、コンテナ自身も動かし続けます。CFEngine によって、コンテナの生存をサービスの状況と切り離せるようになります。

7.4.1 どのように動作するのか

CFEngine は、cfe-docker 統合ポリシー（integration policies）と一緒に `Dockerfile` の中でインストールします。Docker イメージの中で CFEngine が構築されています。

Dockerfile の `ENTRYPOINT` には任意のコマンド（と任意の引数）をパラメータとして指定できます。Docker コンテナが CFEngine ポリシーの書かれたパラメータを受け取り実行する時、CFEngine はコンテナ内で任意のプロセスが間違いなく実行されるようにします。

CFEngine は `ENTRYPOINT` で指定したコマンドの**ベースネーム**（最終的に実行するコマンドの名前）にあたるプロセス・テーブルをスキャンし、 ベースネームのプロセスが見つからなければ、それを開始しようとします。例えば、コンテナを `docker run "/path/to/my/application パラメータ"` で開始したら、CFEngine は `application` という名前のプロセスを探し、コマンドを実行します。もし `application` にあたるエントリがプロセス・テーブルに無ければ、CFEngine は `/path/to/my/application パラメータ` でアプリケーションを再度起動しようとします。このプロセス・テーブルの確認は、毎分実行されます。

従って重要になるのは、アプリケーションを開始するにあたり、そのプロセスにベースネームが含まれている必要があるため、ご注意ください。

7.4.2 使い方

この例は、Docker をインストール済みであり、動くものと想定しています。これから 1 つのコンテナの中に `apache2` と `sshd` をインストール・管理します。

3 つの手順を踏みます：

1. コンテナの中に CFEngine をインストールします。
2. CFEngine の Docker プロセス管理ポリシーを、CFEngine をインストールしたコンテナにコピーします。
3. `docker run` コマンドの一部として、アプリケーション・プロセスを開始します。

イメージの構築

以下のように、Dockerfile の 1 ～ 2 ステップで設定できます。

```
FROM ubuntu
MAINTAINER Eystein M å løy Stenberg <eytein.stenberg@gmail.com>
```

```
RUN apt-get update && apt-get install -y wget lsb-release unzip ca-certificates

# install latest CFEngine
RUN wget -qO- http://cfengine.com/pub/gpg.key | apt-key add -
RUN echo "deb http://cfengine.com/pub/apt $(lsb_release -cs) main" >
/etc/apt/sources.list.d/cfengine-community.list
RUN apt-get update && apt-get install -y cfengine-community

# install cfe-docker process management policy
RUN wget https://github.com/estenberg/cfe-docker/archive/master.zip -P /tmp/ && unzip /tmp/master.zip
-d /tmp/
RUN cp /tmp/cfe-docker-master/cfengine/bin/* /var/cfengine/bin/
RUN cp /tmp/cfe-docker-master/cfengine/inputs/* /var/cfengine/inputs/
RUN rm -rf /tmp/cfe-docker-master /tmp/master.zip

# apache2 and openssh are just for testing purposes, install your own apps here
RUN apt-get update && apt-get install -y openssh-server apache2
RUN mkdir -p /var/run/sshd
RUN echo "root:password" | chpasswd  # need a password for ssh

ENTRYPOINT ["/var/cfengine/bin/docker_processes_run.sh"]
```

作業ディレクトリでこのファイルを Dockerfile として保存します。イメージを構築するには docker build コマンドを使います。例： `docker build -t managed_image` 。

コンテナのテスト

apache2 と sshd を実行するコンテナを開始し、SSH インスタンスのポート転送を管理します。

```
$ docker run -p 127.0.0.1:222:22 -d managed_image "/usr/sbin/sshd" "/etc/init.d/apache2 start"
```

これが、まさに cfe-docker 統合における明確な利点です。通常の `docker run` コマンドの一部として、複数のプロセスを開始します。

新しいコンテナ内にログインしたら、apache2 と sshd の両方が動作しています。先ほどの Dockerfile で root パスワードを“password”と指定しましたので、これを使って SSH でログインします。

```
ssh -p222 root@127.0.0.1

ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 07:48 ?        00:00:00 /bin/bash /var/cfengine/bin/docker_processes_run.sh
/usr/sbin/sshd /etc/init.d/apache2 start
root        18     1  0 07:48 ?        00:00:00 /var/cfengine/bin/cf-execd -F
root        20     1  0 07:48 ?        00:00:00 /usr/sbin/sshd
root        32     1  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    34    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    35    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
www-data    36    32  0 07:48 ?        00:00:00 /usr/sbin/apache2 -k start
root        93    20  0 07:48 ?        00:00:00 sshd: root@pts/0
root       105    93  0 07:48 pts/0    00:00:00 -bash
root       112   105  0 07:49 pts/0    00:00:00 ps -ef
```

もし apache2 を停止しても、CFEngine が 1 分以内に再起動します。

```
service apache2 status
 Apache2 is running (pid 32).
service apache2 stop
          * Stopping web server apache2 ... waiting    [ OK ]
service apache2 status
 Apache2 is NOT running.
# ... wait up to 1 minute...
service apache2 status
 Apache2 is running (pid 173).
```

自分のアプリケーションを同じように設定できます。上記の例で調整が必要なのは、２ヵ所だけです。

- 上記の Dockerfile を使い、apache2 と sshd の代わりに自分のアプリケーションをインストールします。
- docker run でコンテナ起動時に、apache2 と sshd ではなく、自分のアプリケーション用のコマンドライン引数を指定します。

7.5 Chef を使う

このインストール方法に関するドキュメントは、コミュニティからの貢献です。

7.5.1 動作条件

このガイドを使う前に、Chef[*1] のインストール作業が必要です。cookbook は様々なオペレーティング・システムに対応しています。

7.5.2 インストール

Chef Supermarket[*2] 上の cookbook が利用可能です。そして、好みの cookbook 依存関係マネージャ（dependency manager）を使ってインストールできます。

ソースは GitHub 上（https://github.com/someara/chef-docker）にあります。

7.5.3 使い方

- 自分の cookbook の metadata.rb に `depends 'docker', '~> 2.0'` を追加します。
- cookbook のレシピに送信するリソースを指定します。同様にコア Chef リソースも使えます（file、template、directory、package 等）。

```
docker_service 'default' do
  action [:create, :start]
end

docker_image 'busybox' do
  action :pull
end

docker_container 'an echo server' do
  repo 'busybox'
  port '1234:1234'
  command "nc -ll -p 1234 -e /bin/cat"
end
```

7.5.4 はじめましょう

こちらの例は、最新のイメージを取得し、コンテナ実行時にポートを公開します。

```
# Pull latest image
docker_image 'nginx' do
  tag 'latest'
  action :pull
```

*1 http://www.chef.io/

*2 https://supermarket.chef.io/cookbooks/docker

```
end

# Run container exposing ports
docker_container 'my_nginx' do
  repo 'nginx'
  tag 'latest'
  port '80:80'
  binds [ '/some/local/files/:/etc/nginx/conf.d' ]
  host_name 'www'
  domain_name 'computers.biz'
  env 'FOO=bar'
  subscribes :redeploy, 'docker_image[nginx]'
end
```

7.6 Puppet を使う

このインストール方法は、コミュニティにからの貢献です。 公式のインストール方法は Ubuntu を使います。このバージョンは情報が古いかもしれません。

7.6.1 動作条件

このガイドを利用するには、Puppet Labs（パケットラブズ）が提供する Puppet（パペット）[*1] をインストールする必要があります。

また、現時点の公式パッケージが利用できるモジュールは、Ubuntu 向けのみです。

7.6.2 インストール

モジュールは Puppet Forge（フォージ）[*2] のものが使えます。内部モジュールのツールを使ってインストールします。

```
$ puppet module install garethr/docker
```

また、ソースをダウンロードするには、GitHub （https://github.com/garethr/garethr-docker）も使えます。

7.6.3 使い方

モジュールは Docker をインストールする puppet クラスを提供し、イメージとコンテナを管理するために 2 つのタイプを定義します。

インストール

```
include 'docker'
```

イメージ

次の手順でほとんどの Docker イメージをインストールします。この例では、次のように使用するタイプを定義しています。

```
docker::image { 'ubuntu': }
```

これは、次のコマンドと同等です。

```
$ docker pull ubuntu
```

```
docker::image { 'ubuntu':
  ensure => 'absent',
}
```

*1 https://puppetlabs.com/

*2 https://forge.puppetlabs.com/garethr/docker/

コンテナ

これで、Docker によって管理されるコンテナ内で、コマンドを実行するイメージができます。

```
docker::run { 'helloworld':
  image   => 'ubuntu',
  command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',
}
```

これは upstart 下での次のコマンドと同等です。

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

ports 、 env 、 dns 、 volumes の属性は文字で指定するか、先ほどの の値で指定します。

7.7 Supervisor を Docker で使う

伝統的に Docker コンテナは起動時に 1 つのプロセスを実行します。例えば、Apache デーモンや SSH サーバのデーモンです。しかし、コンテナ内で複数のプロセスを起動したいこともあるでしょう。これを実現するにはいくつもの方法があります。プロセス管理ツールをインストールすることで、コンテナの CMD 命令で単純な Bash スクリプトを使えるようにします。

ここでは例としてプロセス管理ツール Supervisor(スーパーバイザー)[*1] を使い、コンテナ内で複数のプロセスを管理します。Supervisor を使うことにより、制御・管理しやすくし、実行したいプロセスを再起動できます。デモンストレーションとして、SSH デーモンと Apache デーモンの両方をインストール・管理します。

基本的な Dockerfile から新しいイメージを作りましょう。

```
FROM ubuntu:13.04
MAINTAINER examples@docker.com
```

7.7.2 Supervisor のインストール

SSH と Apache デーモンと同じように、Supervisor をコンテナにインストールできます。

```
RUN apt-get update && apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
```

ここでインストールしたパッケージは openssh-server、apache2、supervisor（Supervisor デーモンを提供）です。それから、SSH デーモンと Supervisor を実行するための新しいディレクトリの作成も必要です。

次は Supervisor の設定ファイルを追加しましょう。デフォルトのファイルは supervisord.conf であり、/etc/supervisor/conf.d/ に置きます。

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

superviserd.conf の内容を見ましょう。

```
[supervisord]
nodaemon=true

[program:sshd]
command=/usr/sbin/sshd -D

[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

supervisord.conf 設定ファイルにはディレクティブ（命令）を記述します。これは Supervisor とプロセスを管

*1 http://supervisord.org/

理するためです。始めのブロック [supervisord] は Supervisord 自身の設定を指定します。ここで使ったディレクティブ nodaemon は、Supervisor をデーモン化するのではなく、インタラクティブに実行します。

次の２つのブロックは制御したいサービスを管理します。各ブロックは、別々のプロセスです。ブロックには command というディレクティブが１つあり、各プロセスで何のコマンドを起動するか指定します。

Dockerfile を仕上げるには、コンテナの実行時に必要な公開ポートや、 Supervisor 起動のための CMD 命令を追加します。

```
EXPOSE 22 80
CMD ["/usr/bin/supervisord"]
```

ここでは、コンテナのポート 22 と 80 を公開し、コンテナの起動時に /usr/bin/supervisord バイナリを実行します。

7.7.5 イメージの構築

これで新しいイメージを構築できます。

```
$ docker build -t <yourname>/supervisord .
```

イメージを構築したら、これを使ってコンテナを起動します。

```
$ docker run -p 22 -p 80 -t -i <yourname>/supervisord
2013-11-25 18:53:22,312 CRIT Supervisor running as root (no user in config file)
2013-11-25 18:53:22,312 WARN Included extra file "/etc/supervisor/conf.d/supervisord.conf" during
parsing
2013-11-25 18:53:22,342 INFO supervisord started with pid 1
2013-11-25 18:53:23,346 INFO spawned: 'sshd' with pid 6
2013-11-25 18:53:23,349 INFO spawned: 'apache2' with pid 7
...
```

docker run コマンドを実行することで、新しいコンテナをインタラクティブに起動しました。このコンテナは Supervisor を実行し、一緒に SSH と Apache デーモンを起動します。 -p フラグを指定し、ポート 22 と 80 を公開します。ここで、SSH と Apache デーモンの両方に接続できるようにするため、公開ポートを個々に指定しています。

Docker のインストールに成功したら、docker デーモンはデフォルトの設定で動いています。

プロダクション環境では、システム管理者は組織における必要性に従い、docker デーモンの設定を変更し、起動・停止するでしょう。ほとんどの場合、システム管理者は docker デーモンの起動・停止のために SysVinit、Upstart 、systemd といったプロセス・マネージャを設定するでしょう。

docker デーモンは docker daemon コマンドで直接操作できます。デフォルトでは Unix ソケット unix:///var/run/docker.sock をリッスンします。

```
$ docker daemon

INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
...
...
```

プロセス・マネージャの代わりに、docker daemon コマンドを使って docker デーモンを直接実行できます。docker をコマンドで直接実行時に、オプション設定を追加可能です。docker デーモンを設定するための他のオプションも追加できます。

デーモンで利用可能なオプション：

フラグ	説明
-D , --debug=false	デバッグ・モードの有効化と無効化。デフォルトは false
-H , --host=[]	デーモンが接続するソケット
--tls=false	TLS の有効化と無効化。デフォルトは false

以下は docker デーモンに設定オプションを付けて実行する例です。

```
$ docker daemon -D --tls=true --tlscert=/var/docker/server.pem --tlskey=/var/docker/serverkey.pem \
    -H tcp://192.168.59.3:2376
```

３つのオプションがあります：

- -D（デバッグ）モードの有効化
- tls を有効にするため、サーバ証明書と鍵を --tlscert と --tlskey で個々に指定
- tcp://192.168.59.3:2376 への接続をリッスン

コマンドライン・リファレンスのデーモンのフラグ一覧に説明があります。

デーモンのデバッグ

更に捕捉しますと、管理者やオペレータがデーモンの実行時の挙動に関して、更に詳細な情報を得るには、デー

モンのログレベルを「debug」に設定するか、 -D オプションを付けてデバッグ・モードにします。デーモンからの応答が無くても、Docker デーモンに対して SIGUSR1 シグナルを送信することで、デーモンのログに追加された全てのスレッドを強制的に追跡します。Linux システム上でシグナルを送る一般的な方法は kill シグナルを使います。例えば kill -USR1 <デーモンの pid> を実行したら、デーモンのプロセスに SIGUSR1 シグナルを送信し、スタック・ダンプをデーモンのログに追加します。

スタック・トレースをログに保存するには、 デーモンのログレベルの設定は少なくとも「info」レベル以上にする必要があります。デフォルトのデーモンのログレベルは「info」に設定されています。

デーモンは SIGUSR1 シグナルを受け取った後、スタック・トレースをダンプしてログに出力します。スタック・トレースはデーモン内部の全ての goroutine の状態とスレッドの把握に使えます。

7.8.3 Ubuntu

Ubunt 14.04 からはプロセス・マネージャに Upstart（アップスタート） を使います。デフォルトでは、Upstart のジョブは /etc/init に保管され、 docker Upstart ジョブは /etc/init/docker.conf にあります。

Ubuntu で Docker のインストールに成功した後、Upstart で稼働状態を確認するには、次のようにします。

```
$ sudo status docker
docker start/running, process 989
```

docker デーモンは次のように開始・停止・再起動できます。

```
$ sudo start docker
```

```
$ sudo stop docker
```

```
$ sudo restart docker
```

以下の例は、プロセス・マネージャに upstart を使い Docker システムを設定する方法です。Ubuntu 15.04 以降はプロセス・マネージャに systemd（システムディー） を使います。Ubuntu 15.04 以降は、「systemd で Docker の管理・設定」のセクションをご覧ください。

システム上にある docker デーモンの設定は、/etc/default/docker ファイルを編集します。ここに DOCKER_OPTS 環境変数を指定可能です。

Docker オプションの設定を変更するには：

1. ホストに sudo や root 特権を持つユーザでログインします。

2. ホスト上に /etc/default/docker ファイルが無ければ作成します。Docker のインストール方法によっては、既にファイルが作成されている場合があります。

3. 任意のエディタでファイルを開きます。

```
$ sudo vi /etc/default/docker
```

4. DOCKER_OPTS 変数に、次のオプションを指定します。これらのオプションは docker デーモンを実行する時に追加されます。

```
DOCKER_OPTS="-D --tls=true --tlscert=/var/docker/server.pem --tlskey=/var/docker/serverkey.pem \
    -H tcp://192.168.59.3:2376"
```

これらのオプションの意味は：

- -D（デバッグ）モードの有効化
- tls を有効にするため、サーバ証明書と鍵を --tlscert と --tlskey で個々に指定
- tcp://192.168.59.3:2376 への接続をリッスン

コマンドライン・リファレンスのデーモンのフラグ一覧に説明があります。

5. ファイルを保存して閉じます。

6. docker デーモンを再起動します。

```
$ sudo restart docker
```

7. docker デーモンが指定したオプションで実行しているか、 ps コマンドで確認します。

```
$ ps aux | grep docker | grep -v grep
```

ログ

Upstart ジョブのログは、デフォルトでは /var/log/upstart に保管されており、docker デーモンのログは /var/log/upstart/docker.log にあります。

```
$ tail -f /var/log/upstart/docker.log
INFO[0000] Loading containers: done.
INFO[0000] Docker daemon commit=1b09a95-unsupported graphdriver=aufs version=1.11.0-dev
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
INFO[0000] Daemon has completed initialization
```

7.8.4 CentOS / Red Hat Enterprise Linux / Fedora

CentOS と RHEL の 7.x 以降では、プロセス・マネージャに systemd を使います。Fedora 21 以降は、プロセス・マネージャに systemd を使います。

CentOS、Red Hat Enterprise Linux、Fedora に Docker をインストール後は、次のように稼働状態を確認できます。

```
$ sudo systemctl status docker
```

docker デーモンは次のように開始・停止・再起動できます。

```
$ sudo systemctl start docker

$ sudo systemctl stop docker

$ sudo systemctl restart docker
```

Docker をブート時に起動するようにするには、次のように実行すべきです。

```
$ sudo systemctl enable docker
```

CentOS 7.x と RHEL 7.x では systemd で Docker を管理・設定できます 。

以前の CentOS 6.x や RHEL 6.x の場合は、システム上にある docker デーモンの設定は /etc/default/docker ファイルを編集し、ここで様々な変数を設定します。CentOS 7.x と RHEL 7.x では、この変数名が OPTIONS になります。CentOS 6.x と RHEL 6.x では、この変数名は other_args です。このセクションでは CentOS 7 の docker デーモンを例に説明します。

Docker オプションの設定を変更するには：

1. ホストに sudo や root 特権を持つユーザでログインします。

2. /etc/systemd/system/docker.service.d ディレクトリを作成します。

```
$ sudo mkdir /etc/systemd/system/docker.service.d
```

3. /etc/systemd/system/docker.service.d/docker.conf ファイルを作成します。

4. 任意のエディタでファイルを開きます。

```
$ sudo vi /etc/systemd/system/docker.service.d/docker.conf
```

5. docker デーモンの設定を変更するため、docker.service ファイルの ExecStart 設定を上書きします。ExecStart 設定を変更するためには、新しい設定行を追加する前に、次のように空の設定行を追加します。

```
[Service]
ExecStart=
ExecStart=/usr/bin/docker daemon -H fd:// -D --tls=true --tlscert=/var/docker/server.pem
--tlskey=/var/docker/serverkey.pem -H tcp://192.168.59.3:2376
```

これらのオプションの意味は：

- -D（デバッグ）モードの有効化

- tls を有効にするため、サーバ証明書と鍵を --tlscert と --tlskey で個々に指定
- tcp://192.168.59.3:2376 への接続をリッスン

コマンドライン・リファレンスのデーモンのフラグ一覧に説明があります。

6. ファイルを保存して閉じます。

7. 変更を反映（フラッシュ）します。

```
$ sudo systemctl daemon-reload
```

8. docker デーモンを再起動します。

```
$ sudo systemctl restart docker
```

9. docker デーモンが指定したオプションで実行しているか、ps コマンドで確認します。

```
$ ps aux | grep docker | grep -v grep
```

ログ

systemd は自身で journal と呼ばれるロギング・システムを持っています。docker デーモンのログ表示は journalctl -u docker を使います。

```
$ sudo journalctl -u docker
May 06 00:22:05 localhost.localdomain systemd[1]: Starting Docker Application Container Engine...
May 06 00:22:05 localhost.localdomain docker[2495]: time="2015-05-06T00:22:05Z" level="info" msg="+job
serveapi(unix:///var/run/docker.sock)"
May 06 00:22:05 localhost.localdomain docker[2495]: time="2015-05-06T00:22:05Z" level="info"
msg="Listening for HTTP on unix (/var/run/docker.sock)"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info" msg="+job
init_networkdriver()"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info" msg="-job
init_networkdriver() = OK (0)"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info"
msg="Loading containers: start."
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info"
msg="Loading containers: done."
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info"
msg="Docker daemon commit=1b09a95-unsupported graphdriver=aufs version=1.11.0-dev""
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info" msg="+job
acceptconnections()"
May 06 00:22:06 localhost.localdomain docker[2495]: time="2015-05-06T00:22:06Z" level="info" msg="-job
acceptconnections() = OK (0)"
```

journal の使い方や設定方法は高度なトピックのため、このドキュメントの範囲では扱いません。

コンテナのラインタイム・メトリクス（訳注；コンテナ実行時の、様々なリソース指標や数値データ）をライブ（生）で表示するには、docker stats（スタッツ）コマンドを使います。コマンドがサポートしているのは、CPU、メモリ使用率、メモリ上限、ネットワーク I/O のメトリクスです。

以下は docker stats コマンドを実行した例です。

```
$ docker stats redis1 redis2
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O
BLOCK I/O
redis1              0.07%               796 KB / 64 MB        1.21%               788 B / 648 B
3.568 MB / 512 KB
redis2              0.07%               2.746 MB / 64 MB      4.29%               1.266 KB / 648 B
12.4 MB / 0 B
```

docker stats コマンドのより詳細な情報は、docker stats リファレンス・ページ をご覧ください。

7.9.1 コントロール・グループ

Linux はプロセス・グループの追跡だけでなく、CPU・メモリ・ブロック I/O　のメトリクス表示は、コントロール・グループ[1] に依存しています。これらのメトリクスやネットワーク使用量のメトリクスも同様に取得できます。これらは「純粋な」 LXC コンテナ用であり、Docker コンテナ用でもあります。

コントロール・グループは　　を通して公開されています。最近のディストリビューションでは、 /sys/fs/cgroup 以下で見つかるでしょう。このディレクトリの下に、device・freezer・blkio 等の複数のサブディレクトリがあります。各サブディレクトリは、それぞれ異なった cgroup 階層に相当します。

古いシステムでは、コントロール・グループが /cgroup にマウントされており、その下に明確な階層が無いかもしれません。そのような場合、サブディレクトリが見える代わりに、たくさんのファイルがあるでしょう。あるいは、存在しているコンテナに相当するディレクトリがあるかもしれません。

どこにコントロール・グループがマウントされているかを調べるには、次のように実行します。

```
$ grep cgroup /proc/mounts
```

7.9.2 コントロール・グループの列挙

/proc/cgroups を調べれば、システム上の様々に異なるコントロール・グループのサブシステムが見えます。それぞれに階層がサブシステムに相当しており、多くのグループが見えるでしょう。

コントロール・グループのプロセスに属する情報は、 /proc/<pic>/cgroup からも確認できます。コントロール・グループは階層のマウントポイントからの相対パス上に表示されます。例えば、 / が意味するのは「対象のプロセスは特定のグループに割り当てられていない」であり、 /lxc/pumpkin が意味するのはプロセスが pumpkin と呼ばれるコンテナのメンバであると考えられます。

コンテナごとに、それぞれの階層に cgroup が作成されます。古いシステム上のバージョンが古い LXC　userland

*1 https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

tools の場合、cgroups の名前はコンテナ名になっています。より最近のバージョンの LXC ツールであれば、cgroup は lxc/<コンテナ名> になります。

Docker コンテナは cgroups を使うため、コンテナ名はフル ID か、コンテナのロング ID になります。 docker ps コマンドで表示されたコンテナ ID が ae836c95b4c3 のように見えるのであれば、このコンテナのロング ID は ae836c95b4c3c9e9179e0e91015512da89fdec91612f63cebae57df9a5444c79 のようなものです。この情報を調べるには、docker inspect か docker ps --no-trunc を使います。

Docker コンテナが利用するメモリのメトリクスは、 /sys/fs/cgroup/memory/docker/<ロング ID>/ から全て参照できます。

各サブシステム（メモリ、CPU、ブロック I/O）ごとに、1 つまたは複数の に統計情報が含まれます。

メモリ・メトリクスは「memory」cgroups にあります。メモリのコントロール・グループは少々のオーバーヘッドが増えるのを覚えておいてください。これはホスト上における詳細なメモリ使用情報を計算するためです。そのため、多くのディストリビューションではデフォルトでは無効です。一般的に、有効にするためには、カーネルのコマンドライン・パラメータに cgroup_enable=memory swapaccount=1 を追加します。

メトリクスは疑似ファイル memory.stat にあります。次のように表示されます。

```
cache 11492564992
rss 1930993664
mapped_file 306728960
pgpgin 406632648
pgpgout 403355412
swap 0
pgfault 728281223
pgmajfault 1724
inactive_anon 46608384
active_anon 1884520448
inactive_file 7003344896
active_file 4489052160
unevictable 32768
hierarchical_memory_limit 9223372036854775807
hierarchical_memsw_limit 9223372036854775807
total_cache 11492564992
total_rss 1930993664
total_mapped_file 306728960
total_pgpgin 406632648
total_pgpgout 403355412
total_swap 0
total_pgfault 728281223
total_pgmajfault 1724
total_inactive_anon 46608384
total_active_anon 1884520448
total_inactive_file 7003344896
total_active_file 4489052160
total_unevictable 32768
```

前半（total_が先頭に無い ）は、cgroup 中にあるプロセス関連の統計情報を表示します。サブグループは除外しています。後半（先頭に total_がある ）は、サブグループも含めたものです。

いくつかのメトリクスは「gauges」（ゲージ；計測した値そのものの意味）であり、例えば、値が増減するものです（例：swap は cgroup のメンバによって使われている swap 領域の容量です）。あるいは「counter」（カウンタ）は、特定のイベント発生後に増えた値のみ表示します（例：pgfault はページ・フォルトの回数を表しますが、cgroup が作成された後の値です。この値は決して減少しません。）。

- **cache**：コントロール・グループのプロセスによって使用されるメモリ容量であり、ブロック・デバイス上のブロックと密接に関わりがあります。ディスクからファイルを読み書きしたら、この値が増えます。値が増えるのは「通常」の I/O（`open`、`read`、`write` システムコール）だけでなく、ファイルのマップ（`mmap` を使用）でも同様です。あるいは `tmpfs` マウントでメモリを使う場合も、理由が明確でなくともカウントされます。
- **rss**：ディスクに関連 しない メモリ使用量です。例えば、stacks、heaps、アノニマスなメモリマップです。
- **mapped_file**：コントロール・グループ上のプロセスに割り当てられるファイル容量です。 メモリを**どのように**使用しているかの情報は得られません。どれだけ使っているかを表示します。
- **pgfault と pgmajfault**：cgroup のプロセスが「page fault」と「major fault」の回数を個々に表示します。page fault とは、存在しないかプロテクトされた仮想メモリスペースにプロセスがアクセスした時に発生します。かつては、プロセスにバグがあり、無効なアドレスにアクセスしようとした時に発生しました（`SIGSEGV` シグナルが送信されます。典型的なのは `Segmentation fault` メッセージを表示して kill される場合です ）。最近であれば、プロセスがスワップ・アウトされたメモリ領域を読み込みに行くか、あるいはマップされたファイルに相当する時に発生します。そのような場合、カーネルはページをディスクから読み込み、CPU がメモリへのアクセスを処理します。これはまた、プロセスがコピー・オン・ライト（copy-on-write）のメモリ領域に書き込んだ時にも発生します。これはカーネルがプロセスの実行を阻止するのと同じであり、メモリページを複製し、プロセスが自身のページをコピーして書き込み作業を再開しようとします。「メジャー」な失敗が起こるのは、カーネルが実際にディスクからデータを読み込む時点です。読み込みによって、既存のページと重複するか、空のページが割り当てられると一般的な（あるいは「マイナー」な）エラーが発生します。
- **swap**：対象の cgroup にあるプロセスが、現在どれだけ swap を使っているかの量です。
- **active_anon と inactive_anon**：カーネルによって active と inactive に区分される anonymous メモリ容量です。anonymous メモリとは、ディスク・ページにリンクされないメモリです。言い換えれば、先ほど説明した rss カウンタと同等なものです。実際、rss カウンタの厳密な定義は、**active_anon + inactive_anon - tmpfs** です（tmpfs のメモリ容量とは、このコントロール・グループの `tmpfs` ファイルシステムがマウントして使っている容量です）。では次に、「active」と「inactive」の違いは何でしょうか？ ページは「active」として始まりますが、一定の時間が経てば、カーネルがメモリを整理（sweep）して、いくつかのページを「inactive」にタグ付けします。再度アクセスがあれば、直ちに「active」に再度タグ付けされます。カーネルがメモリ不足に近づくか、ディスクへのスワップアウト回数により、カーネルは「inactive」なページをスワップします。
- ：キャッシュメモリの active と inactive は、先ほどの anonymou メモリの説明にあるものと似ています。正確な計算式は、キャッシュ = **active_file + inactive_file + tmpfs** です。この正確なルールが使われるのは、カーネルがメモリページを active から inactive にセットする時です。これは anonymous メモリとして使うのとは違って、一般的な基本原理によるものと同じです。注意点としては、カーネルがメモリを再要求（reclaim）するするとき、直ちに再要求（anonymous ページや汚れた/変更されたページをディスクに書き込む）よりも、プール上のクリーンな（＝変更されていない）ページを再要求するほうが簡単だからです。
- **unevictable**：再要求されないメモリの容量です。一般的に `mlock` で「ロックされた」メモリ容量です。暗号化フレームワークによる秘密鍵の作成や、ディスクにスワップさせたくないような繊細な素材に使われます。
- **memory と memsw の limits**：これらは実際のメトリクスではありませんが、対象の cgroup に適用される上限の確認に使います。「memory」はこのコントロール・グループのプロセスによって使われる最大の物理メモリを示します。「memsw」は RAM+swap の最大容量を示します。

ページキャッシュ中のメモリ計算は非常に複雑です。もし２つのプロセスが異なったコントロール・グループ上にあるなら、それぞれの同じファイル（結局はディスク上の同じブロックに依存しますが）を読み込む必要があり

ます。割り当てられたメモリは、コントロール・グループごとの容量に依存します。これは良さそうですが、cgroup を削除したら、メモリページとして消費していた領域は使わなくなり、他の cgroup のメモリ容量を増加させることをも意味します。

これまではメモリのメトリクスを見てきました。メモリに比べると他のものは非常に簡単に見えるでしょう。CPU メトリクスは cpuacct コントローラにあります。

コンテナごとに疑似ファイル `cpuacct.stat` があり、ここにコンテナにあるプロセスの CPU 使用率を、`user` 時間と `system` 時間に分割して記録されます。いずれも慣れていなければ、`user` とはプロセスが CPU を直接制御する時間のこと（例：プロセス・コードの実行）であり、`system` とはプロセスに代わり CPU のシステムコールを実行する時間です。

これらの時間は 100 分の 1 秒の周期（tick）で表示されます。実際にはこれらは「user jiffies」として表示されます。`USER_HZ`「jillies」が毎秒かつ x86 システムであれば、`USER_HZ` は 100 です。これは 1 秒の「周期」で、スケジューラが実際に割り当てる時に使いますが、tickless kernels[*1] にあるように、多くのカーネルで ticks は適切ではありません。まだ残っているのは、主に遺産（レガシー）と互換性のためです。

Block I/O メトリクス

Block I/O は `blkio` コントローラを算出します。異なったメトリックスが別々のファイルに散在しています。より詳細な情報を知りたい場合は、カーネル・ドキュメントの blkio-controller[*2] をご覧ください。ここでは最も関係が深いものをいくつか扱います。

- **blkio.sectors**：cgroups のプロセスのメンバが、512 バイトのセクタをデバイスごとに読み書きするものです。読み書きは単一のカウンタに合算されます。
- **blkio.io_service_bytes**：cgroup で読み書きしたバイト数を表示します。デバイスごとに 4 つのカウンタがあります。これは、デバイスごとに同期・非同期 I/O と、読み込み・書き込みがあるからです。
- **blkio.io_serviced**：サイズに関わらず I/O 操作の実行回数です。こちらもデバイスごとに 4 つのカウンタがあります。
- **blkio.io_queued**：このグループ上で I/O 動作がキュー（保留）されている数を表示します。言い換えれば、cgroup が何ら I/O を処理しなければ、この値は 0 になります。ただし、その逆の場合は違うので気を付けてください。つまり、I/O キューが発生していなくても、cgroup がアイドルだとは言えません。これは、キューが無くても、純粋に停止しているデバイスからの同期読み込みを行い、直ちに処理することができるためです。また、cgroup は I/O サブシステムに対するプレッシャーを、相対的な量に保とうとする手助けになります。プロセスのグループが更に I/O が必要になれば、キューサイズが増えることにより、他のデバイスとの負荷が増えるでしょう。

7.9.5 ネットワーク・メトリクス

ネットワークのメトリクスは、コントロール・グループから直接表示されません。ここに良いたとえがあります。ネットワーク・インターフェースとはネットワーク名前空間（network namespaces）内のコンテクスト（内容）として存在します。カーネルは、プロセスのグループが送受信したパケットとバイト数を大まかに計算できます。しかし、これらのメトリックスは使いづらいものです。インターフェースごとのメトリクスが欲しいでしょう（なぜなら、ローカルの `lo` インターフェスに発生するトラフィックが実際に計測できないためです ）。ですが、単一の cgroup 内のプロセスは、複数のネットワーク名前空間に所属するようになりました。これらのメトリクスの解釈は

*1 http://lwn.net/Articles/549580/

*2 https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt

大変です。複数のネットワーク名前空間が意味するのは、複数の lo インターフェース、複数の eth0 インターフェース等を持ちます。つまり、コントロール・グループからネットワーク・メトリクスを簡単に取得する方法はありません。

そのかわり、他のソースからネットワークのメトリクスを集められます。

IPtables

IPtables を使えば（というよりも、インターフェースに対する iptables の netfilter フレームワークを使うことにより)、ある程度正しく計測できます。

例えば、ウェブサーバの外側に対する(outbound) HTTP トラフィックの計算のために、次のようなルールを作成できます。

```
$ iptables -I OUTPUT -p tcp --sport 80
```

ここには何ら -j や -g フラグはありませんが、ルールがあることにより、一致するパケットは次のルールに渡されます。

それから、次のようにしてカウンタの値を確認できます。

```
$ iptables -nxvL OUTPUT
```

技術的には -n は不要なのですが、今回の例では、不要な DNS 逆引きの名前解決をしないために付けています。

カウンタにはパケットとバイト数が含まれます。これを使ってコンテナのトラフィック用のメトリクスをセットアップしたければ、 コンテナの IP アドレスごとに（内外の方向に対する）２つの iptables ルールの for ループを FORWARD チェーンに追加します。これにより、NAT レイヤを追加するトラフィックのみ計測します。つまり、ユーザランド・プロキシを通過しているトラフィックも加えなくてはいけません。

後は通常の方法で計測します。 collectd（コレクトディー） を使ったことがあるのなら、自動的に iptables のカウンタを収集する便利なプラグイン[*1] があります。

インターフェース・レベルのカウンタ

各コンテナは仮想イーサネット・インターフェースを持つため、そのインターフェースから直接 TX・RX カウンタを取得したくなるでしょう。各コンテナが vethKk8Zq のような仮想イーサネット・インターフェースに割り当てられているのに気を付けてください。コンテナに対応している適切なインターフェースを見つけることは、残念ながら大変です。

しかし今は、 コンテナを通さなくても 数値を確認できる良い方法があります、ホスト環境上で
力を使い、ネットワーク名前空間内のコンテナの情報を確認します。

ip netns exec コマンドは、あらゆるネットワーク名前空間内で、あらゆるプログラムを実行し（対象のホスト上の)、現在のプロセス状況を表示します。つまり、ホストがコンテナのネットワーク名前空間に入れますが、コンテナはホスト側にアクセスできないだけでなく、他のコンテナにもアクセスできません。次のサブコマンドを通すことで、コンテナが「見える」用になります。

正確なコマンドの形式は、次の通りです。

```
$ ip netns exec <nsname> <command...>
```

*1 https://collectd.org/wiki/index.php/Table_of_Plugins

例：

```
$ ip netns exec mycontainer netstat -i
```

ip netns は「mycontainer」コンテナを名前空間の疑似ファイルから探します。各プロセスは 1 つのネットワーク名前空間、PID の名前空間、mnt 名前空間等に属しています。これらの名前空間は /proc/<pid>/ns/ 以下にあります。例えば、PID 42 のネットワーク名前空間に関する情報は、疑似ファイル /proc/42/ns/net です。

ip netns exec mycontainer ... を実行したら、 /var/run/netns/mycontainer が疑似ファイルの 1 つとなるでしょう（シンボリック・リンクが使えます)。

言い換えれば、私たちが必要であれば、ネットワーク名前空間の中でコマンドを実行できるのです。

- 調査したいコンテナに入っている、あらゆる PID を探し出します
- /var/run/netns/<何らかの名前> から /proc/<thepid>/ns/net へのシンボリック・リンクを作成します。
- ip netns exec <何らかの名前> .... を実行します。

ネットワーク使用状況を調査したいコンテナがあり、そこで実行しているプロセスを見つける方法を学ぶには、「コントロール・グループの列挙」のセクションを読み直してください。ここからは tasks と呼ばれる疑似ファイルを例に、コントロール・グループ（つまり、コンテナ）の中にどのような PID があるかを調べましょう。

これらを一度に実行したら、取得したコンテナの「ショート ID」は変数$CID に入れて処理されます。

```
$ TASKS=/sys/fs/cgroup/devices/docker/$CID*/tasks
$ PID=$(head -n 1 $TASKS)
$ mkdir -p /var/run/netns
$ ln -sf /proc/$PID/ns/net /var/run/netns/$CID
$ ip netns exec $CID netstat -i
```

新しいプロセスごとに毎回メトリクスを更新するのは、(比較的）コストがかかるので注意してください。メトリクスを高い解像度で収集したい場合、そして／または、大量のコンテナを扱う場合（1 ホスト上に 1,000 コンテナと考えます)、毎回新しいプロセスをフォークしようとは思わないでしょう。

ここでは 1 つのプロセスでメトリクスを収集する方法を紹介します。メトリクス・コレクションを C 言語で書く必要があります（あるいは、ローレベルなシステムコールが可能な言語を使います)。setns()という特別なシステムコールを使えば、任意の名前空間上にある現在のプロセスを返します。必要があれば、他にも名前空間疑似ファイルのファイル・ディスクリプタ（file descriptor）を開けます（思い出してください：疑似ファイルは /proc/<pid>/ns/net です)。

しかしながら、これはキャッチするだけです。ファイルをオープンのままにできません。つまり、そのままにしておけば、コントロール・グループが終了しても名前空間を破棄できず、ネットワーク・リソース（コンテナの仮想インターフェース等）が残り続けるでしょう（あるいはファイル・ディスクリプタを閉じるまで)。

適切なアプローチで、コンテナごとの最初の PID と、都度、名前空間の疑似ファイルが開かれるたびに、追跡し続ける必要があります。

時々、リアルタイムなメトリクス収集に気を配っていなくても、コンテナ終了時に、どれだけ CPU やメモリ等を使用したか知りたい時があるでしょう。

Docker は lxc-start に依存しており、終了時は丁寧に自分自身をクリーンアップするため困難です。しかし、

他にも方法があります。定期的にメトリクスを集める方法（例：毎分 collectd　LXC プラグインを実行）が簡単です。

しかし、停止したコンテナに関する情報を集めたい時もあるでしょう。次のようにします。

各コンテナで収集プロセスを開始し、コントロール・グループに移動します。これは対象の cgroup のタスクファイルに PID が書かれている場所を監視します。収集プロセスは定期的にタスクファイルを監視し、コントロール・グループの最新プロセスを確認します（先ほどのセクションで暑かったネットワーク統計情報も取得したい場合は、プロセスを適切なネットワーク名前空間にも移動します)。

コンテナが終了すると、`lxc-start` はコントロール・グループを削除しようとします。コントロール・グループが使用中のため、処理は失敗しますが問題ありません。自分で作ったプロセスは、対象のグループ内に自分しかいないことが分かります。それが必要なメトリックスを取得する適切なタイミングです。

最後に、自分のプロセスをルート・コントロール・グループに移動し、コンテナのコントロール・グループを削除します。コントロール・グループの削除は、ディレクトリを `rmdir` するだけです。感覚的にディレクトリに対する `rmdir` は、まだ中にファイルのではと思うかもしれませんが、これは疑似ファイルシステムのため、通常のルールは適用されません。クリーンアップが完了したら、これで収集プロセスを安全に終了できます。

7.10 アンバサダを経由したリンク

7.10.1 はじめに

サービスの利用者とプロバイダ間をネットワーク・リンクで固定するよりも、サービスのポータビリティを Docker は推奨します。

(利用者) --> (redis)

利用者が別の redis サービスに接続するには再起動が必用です。そこで、**アンバサダ**（ambassador；大使、使節の意味）を追加できます。

(利用者) --> (redis-ambassador) --> (redis)

(利用者) --> (redis-ambassador) ---network---> (redis-ambassador) --> (redis)

利用者が別の Redis サーバに通信するよう書き換える必要がある時、コンテナを接続する redis-ambassador しか再起動が不要です。

このパターンは利用者を別の docker ホスト上に対し、透過的に移動させるのにも使えます。

svendowideit/ambassador コンテナを使い、docker run パラメータで全体を制御するリンクを追加してみましょう。

Docker ホスト上で実際の Redis サーバを開始します。

```
big-server $ docker run -d --name redis crosbymichael/redis
```

それから Redis サーバにリンクしてポートを公開するアンバサダを追加します。

```
big-server $ docker run -d --link redis:redis --name redis_ambassador -p 6379:6379  \
                 svendowideit/ambassador
```

別のホスト上で、更にアンバサダをセットアップできます。ここではプロキシしたい big-server のリモート・ポートを環境変数に設定します。

```
client-server $ docker run -d --name redis_ambassador --expose 6379 \
    -e REDIS_PORT_6379_TCP=tcp://192.168.1.52:6379 svendowideit/ambassador
```

それから client-server ホスト上で Redis クライアント・コンテナがリモートの Redis サーバに通信できるようにするため、ローカルの Redis アンバサダにリンクします。

```
client-server $ docker run -i -t --rm --link redis_ambassador:redis relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

7.10.3 動作内容

以下の例で、svendowideit/ambassador コンテナが自動的に（多少の sed の力を使い）何を行っているか見ていきましょう。

Docker ホスト（192.168.1.52）上では Redis が実行されています。

```
# 実際の redis サーバを起動
$ docker run -d --name redis crosbymichael/redis

# 接続テスト用の redis-cli コンテナを取得
$ docker pull relateiq/redis-cli

# redis サーバと直接通信してテスト
$ docker run -t -i --rm --link redis:redis relateiq/redis-cli
redis 172.17.0.136:6379> ping
PONG
^D

# redis アンバサダを追加
$ docker run -t -i --link redis:redis --name redis_ambassador -p 6379:6379 alpine:3.2 sh
```

redis_ambassador コンテナ内では、リンクされた Redis コンテナの状態を env で確認できます。

```
/ # env
REDIS_PORT=tcp://172.17.0.136:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.136
REDIS_NAME=/redis_ambassador/redis
HOSTNAME=19d7adf4705e
SHLVL=1
HOME=/root
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_PORT_6379_TCP=tcp://172.17.0.136:6379
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
/ # exit
```

この環境変数は、アンバサダの socat スクリプトが Redis を公開するために使います（ -p 6379:6369 でポートを割り当てます)。

```
$ docker rm redis_ambassador
$ CMD="apk update && apk add socat && sh"
$ docker run -t -i --link redis:redis --name redis_ambassador -p 6379:6379 alpine:3.2 sh -c "$CMD"
[...]
/ # socat -t 100000000 TCP4-LISTEN:6379,fork,reuseaddr TCP4:172.17.0.136:6379
```

次は Redis サーバにアンバサダ経由で ping します。

次は別のサーバに移動します。

```
$ CMD="apk update && apk add socat && sh"
$ docker run -t -i --expose 6379 --name redis_ambassador alpine:3.2 sh -c "$CMD"
```

```
[...]
/ # socat -t 100000000 TCP4-LISTEN:6379,fork,reuseaddr TCP4:192.168.1.52:6379
```

redis-cli イメージを取得し、アンバサダ・ブリッジを経由して通信します。

```
$ docker pull relateiq/redis-cli
$ docker run -i -t --rm --link redis_ambassador:redis relateiq/redis-cli
redis 172.17.0.160:6379> ping
PONG
```

7.10.4 svendowideit/ambassador Dockerfile

svendowideit/ambassador イメージは socat がインストールされた alpine:3.2 イメージをベースとしています。コンテナを実行すると、小さな sed スクリプトが（利用可能な複数の）リンク環境変数をポート転送用に使います。リモートホストであれば、コマンドラインのオプション実行に -e で環境変数を指定する必要があります。

```
--expose 1234 -e REDIS_PORT_1234_TCP=tcp://192.168.1.52:6379
```

ローカルの 1234 ポートをリモートの IP とポートに転送します。この例では 192.168.1.52:6379 です。

```
#
# do
#   docker build -t svendowideit/ambassador .
# then to run it (on the host that has the real backend on it)
#   docker run -t -i -link redis:redis -name redis_ambassador -p 6379:6379 svendowideit/ambassador
# on the remote host, you can set up another ambassador
#    docker run -t -i -name redis_ambassador -expose 6379 -e REDIS_PORT_6379_TCP=tcp://192.168.1.52:6379
# svendowideit/ambassador sh
# you can read more about this process at https://docs.docker.com/articles/ambassador_pattern_linking/

# use alpine because its a minimal image with a package manager.
# prettymuch all that is needed is a container that has a functioning env and socat (or equivalent)
FROM    alpine:3.2
MAINTAINER  SvenDowideit@home.org.au

RUN apk update && \
    apk add socat && \
    rm -r /var/cache/

CMD env | grep _TCP= | sed 's/.*_PORT_\([0-9]*\)_TCP=tcp:\/\/\(.*\):\(.*\)/socat -t 100000000
TCP4-LISTEN:\1,fork,reuseaddr TCP4:\2:\3 \&/' && echo wait) | sh
```

7.11 ログ保存

コンテナには、Docker デーモンよりも多い様々なロギング（ログ保存）ドライバがあります。ロギング・ドライバを指定するには、docker run コマンドで --log-driver=VALUE を使います。以下のオプションをサポートしています。

none	コンテナ用のロギング・ドライバを無効化します。このドライバを指定したら docker logs は無効化されます。
json-file	Docker 用のデフォルト・ロギング・ドライバです。JSON メッセージをファイルに記録します。
syslog	Docker 用の syslog ロギング・ドライバです。ログ・メッセージを syslog に記録します。
journald	Docker 用の journald ロギング・ドライバです。ログ・メッセージを journald に記録します。
gelf	Docker 用の Graylog Extendef ログ・フォーマット（GELF）ロギング・ドライバです。ログ・メッセージを Graylog のエンドポイントや Logstash に記録します。
fluentd	Docker 用の fluentd ロギング・ドライバです。ログ・メッセージを fluentd に記録します (forward input)。
awslogs	Docker 用の Amazon CloudWatch Logs ロギング・ドライバです。ログ・メッセージを Amazon CloudWatch Logs に記録します。
splunk	Docker 用の Splunk ロギング・ドライバです。HTTP イベント・コレクタを使いログを splunk に書き込みます。
etwlogs	Docker 用の ETW ロギング・ドライバです。ログメッセージを ETW イベントとして書き込みます。
gcplogs	Docker 用の Google Cloud ロギング・ドライバです。ログメッセージを Google Cloud Logging に書き込みます。

docker logs コマンドが使えるのは json-file か journald ロギング・ドライバ使用時のみです。

label と env オプションは、ロギング・ドライバで利用可能な追加属性を加えます。各オプションはキーのリストをカンマで区切ります。label と env キーの間に衝突があれば、env を優先します。

各種オプションは、Docker デーモン起動時に指定します。

```
docker daemon --log-driver=json-file --log-opt labels=foo --log-opt env=foo,fizz
```

それから、label や env の値を指定してコンテナを起動します。例えば、次のように指定するでしょう。

```
docker run --label foo=bar -e fizz=buzz -d -P training/webapp python app.py
```

これはドライバ上のログに依存する追加フィールドを加えるものです。次の例は json-file の場合です。

```
"attrs":{"fizz":"buzz","foo":"bar"}
```

JSON ファイルのオプション

json-file ロギング・ドライバがサポートしているロギング・オプションは以下の通りです。

```
--log-opt max-size=[0-9+][k|m|g]
--log-opt max-file=[0-9+]
--log-opt labels=label1,label2
--log-opt env=env1,env2
```

ログが max-size に到達したら、ロールオーバされます (別のファイルに繰り出されます)。設定できるサイズは、

キロバイト(k)、メガバイト(m)、ギガバイト(g) です。例えば、 --log-opt max-size=50m のように指定します。もしも max-size を設定しなければ、ログはロールオーバされません。

max-file で指定するのは、ログが何回ロールオーバされたら破棄するかです。例えば --log-opt max-file=100 のように指定します。もし max-size を設定しなければ、max-file は無効です。

max-size と max-file をセットしたら、docker logs は直近のログファイルのログ行だけ表示します。

syslog のオプション

syslog ロギング・ドライバがサポートしているロギング・オプションは以下の通りです。

```
--log-opt syslog-address=[tcp|udp|tcp+tls]://host:port
--log-opt syslog-address=unix://path
--log-opt syslog-facility=daemon
--log-opt syslog-tls-ca-cert=/etc/ca-certificates/custom/ca.pem
--log-opt syslog-tls-cert=/etc/ca-certificates/custom/cert.pem
--log-opt syslog-tls-key=/etc/ca-certificates/custom/key.pem
--log-opt syslog-tls-skip-verify=true
--log-opt tag="mailer"
--log-opt syslog-format=[rfc5424|rfc3164]
```

syslog-address は、ドライバが接続するリモートの syslog サーバのアドレスを指定します。指定しなければ、デフォルトでは実行中システム上にあるローカルの unix ソケットを使います。tcp や udp で port を指定しなければ、デフォルトは 514 になります。以下の例は syslog ドライバを使い、リモートの 192.168.0.42 サーバ上のポート 123 に接続する方法です。

```
$ docker run --log-driver=syslog --log-opt syslog-address=tcp://192.168.0.42:123
```

syslog-facility オプションは syslog のファシリティを設定します。デフォルトでは、システムは daemon 値を使います。これを上書きするには、0 から 23 までの整数か、以下のファシリティ名を指定します。

- kern
- user
- mail
- daemon
- auth
- syslog
- lpr
- news
- uucp
- cron
- authpriv
- ftp
- local0
- local1
- local2
- local3
- local4
- local5

- local6
- local7

認証局（CA）によって署名済みの、信頼できる証明書への絶対パスを syslog-tls-ca-cert で指定します。このオプションは tcp+tls 以外のプロトコルを使う場合は無視されます。

syslog-tls-cert は TLS 証明書用ファイルに対する絶対パスです。このオプションは tcp+tls 以外のプロトコルを使う場合は無視されます。

syslog-tls-key は TLS 鍵ファイルに対する絶対パスを指定します。このオプションは tcp+tls 以外のプロトコルを使う場合は無視されます。

syslog-tls-skip-verify は TLS 認証を設定します。デフォルトでは認証が有効ですが、オプションの値を true に指定したら、この設定を上書きします。このオプションは tcp+tls 以外のプロトコルを使う場合は無視されます。

デフォルトでは、Docker はコンテナ ID の冒頭 12 文字のみログ・メッセージにタグ付けします。タグ・フォーマットの記録方式をカスタマイズするには、「log tag オプション」のセクションをご覧ください。

syslog-format は syslog メッセージを書き込み時の書式を指定します。何も指定しなければ、デフォルトではホスト名を指定しないローカルの unix　syslog 形式です。rfc3164 を指定したら、RFC-3164 互換形式でログを記録します。rfc5424 を指定したら、RFC-5424 互換形式で記録します。

journald オプション

journald ロギング・ドライバは journal の CONTAINER_ID フィールドにコンテナ ID を記録します。ロギング・ドライバの詳細な動作については、「journald ロギング・ドライバ」リファレンスをご覧ください。

gelf オプション

GELF ロギングドライバは以下のオプションをサポートしています。

```
--log-opt gelf-address=udp://host:port
--log-opt tag="database"
--log-opt labels=label1,label2
--log-opt env=env1,env2
```

gelf-address オプションは、接続先のリモート GELF サーバのアドレスを指定します。現時点では udp が転送用にサポートされており、利用時に port を指定する必要があります。次の例は gelf ドライバで GELF リモートサーバ 192.168.0.42 のポート 12201 に接続します。

```
$ docker run --log-driver=gelf --log-opt gelf-address=udp://192.168.0.42:12201
```

デフォルトでは、Docker はコンテナ ID の冒頭 12 文字のみログ・メッセージにタグ付けします。タグ・フォーマットの記録方式をカスタマイズするには、「log tag オプション」のドキュメントをご覧ください。

label と env オプションが gelf ロギング・ドライバでサポートされています。これは extra フィールドに、冒頭がアンダースコア（ _ ）で始まるキーを追加するものです。

```
// […]
"_foo": "bar",
"_fizz": "buzz",
// […]
```

--log-opt NAME=VALUE フラグを使い、以下の Fluentd ロギング・ドライバのオプションを追加できます。

- fluentd-address ： 接続先を host:port の形式で指定。[localhost:24224]
- tag ： fluentd メッセージのタグを指定。
- fluentd-buffer-limit ： fluentd ログバッファの最大サイズを指定します。 [8MB]
- fluentd-retry-wait ： 接続リトライ前の初回遅延時間です（以降は指数関数的に増えます） [1000ms]
- fluentd-max-retries ： docker で不意の障害が発生時、最大のリトライ数を指定します。 [1073741824]
- fluentd-async-connect ： 初期接続をブロックするかどうかを指定します。 [false]

例えば、両方のオプションを指定したら、次のようになります。

```
docker run --log-driver=fluentd --log-opt fluentd-address=localhost:24224 \
  --log-opt tag=docker.{{.Name}}
```

コンテナは指定した場所にある Fluentd デーモンに接続できなければ、コンテナは直ちに停止します。このロギング・ドライバの動作に関する詳細情報は「fluentd ロギング・ドライバ」をご覧ください。

Amazon CloudWatch Logs オプションの指定

Amazon CloudWatch ロギングドライバは、以下のオプションをサポートしています。

```
--log-opt awslogs-region=<aws_region>
--log-opt awslogs-group=<log_group_name>
--log-opt awslogs-stream=<log_stream_name>
```

このロギング・ドライバの動作に関する詳細情報は「awslogs ロギング・ドライバ」をご覧ください。

ETW ロギング・ドライバのオプション

etwlogs ロギング・ドライバには必須のオプションはありません。このロギング・ドライバは各ログメッセージを ETW イベントとして転送します。ETW 受信側（リスナー）は受信したイベントを作成できます。
このロギング・ドライバの動作に関する詳細情報は「ETW ロギング・ドライバ」をご覧ください。

Google Cloud ロギング

Google Cloud ロギング・ドライバはいかのオプションをサポートしています。

```
--log-opt gcp-project=<gcp_projext>
--log-opt labels=<label1>,<label2>
--log-opt env=<envvar1>,<envvar2>
--log-opt log-cmd=true
```

このロギング・ドライバの動作に関する詳細情報は Google Cloud ロギング・ドライバ をご覧ください。

7.11.2 ログ用のタグ

tag ログ・オプションは、コンテナのログ・メッセージを識別するため、どのような形式のタグを使うか指定します。デフォルトでは、システムはコンテナ ID の冒頭 12 文字を使います。この動作を上書きするには、tag オプションを使います。

```
docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224 --log-opt tag="mailer"
```

Docker はタグの値を指定するために、特別なテンプレート・マークアップをサポートしています。

マークアップ	説明
{{.ID}}	コンテナ ID の冒頭 12 文字
{{.FullID}}	コンテナの完全 ID
{{.Name}}	コンテナ名
{{.ImageID}}	イメージ ID の冒頭 12 文字
{{.ImageFullId}}	コンテナの完全 ID
{{.ImageName}}	コンテナが使っているイメージ名

例えば、 `--log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}"` を値に指定したら、syslog のログ行は次のようになります。

```
Aug  7 18:33:19 HOSTNAME docker/hello-world/foobar/5790672ab6a0[9103]: Hello from Docker.
```

起動時に、システムは container_name フィールドと{{.Name}}をタグに設定します。docker rename でコンテナ名を変更しても、ログメッセージに新しいコンテナ名は反映されません。そのかわり、これらのメッセージは元々のコンテナ名を使って保存され続けます。

高度な使い方は、go テンプレート[*1] のタグ生成や、コンテナのログ内容[*2] をご覧ください。

以下は syslog ロガーを使う例です：

```
$ docker run -it --rm \
    --log-driver syslog \
    --log-opt tag="{{ (.ExtraAttributes nil).SOME_ENV_VAR }}" \
    --log-opt env=SOME_ENV_VAR \
    -e SOME_ENV_VAR=logtester.1234 \
    flyinprogrammer/logtester
```

ログの結果は次のようになります。

```
Apr  1 15:22:17 ip-10-27-39-73 docker/logtester.1234[45499]: + exec app
Apr  1 15:22:17 ip-10-27-39-73 docker/logtester.1234[45499]: 2016-04-01 15:22:17.075416751 +0000 UTC
stderr msg: 1
```

ドライバがログのオプション syslog-tag、fluentd-tag、gelf-tag を指定しても後方互換性があります。ですが、これらの代わりに、標準化のため一般的な tag ログ・オプションを使うべきです。

7.11.3 Amazon Cloud Watch ロギング・ドライバ

awslogs ロギングドライバは、コンテナのログを Amazon CloudWatch ログ[*3] に送信します。ログのエントリは、AWS マネジメント・コンソール[*4] や AWS SDK やコマンドライン・ツール[*5] を通して確認できます。

*1 http://golang.org/pkg/text/template/

*2 https://github.com/docker/docker/blob/master/daemon/logger/context.go

*3 https://aws.amazon.com/cloudwatch/details/#log-monitoring

*4 https://console.aws.amazon.com/cloudwatch/home#logs:

*5 http://docs.aws.amazon.com/cli/latest/reference/logs/index.html

使い方

デフォルトのロギング・ドライバを指定するには、Docker デーモンで --log-driver オプションを使います。

```
docker daemon --log-driver=awslogs
```

特定のコンテナに対するロギング・ドライバの指定は、docker run で --log-driver オプションを使います。

```
docker run --log-driver=awslogs ...
```

Amazon CloudWatch ログのオプション

Amazon CloudWatch Logs ロギング・ドライバのオプションを指定するには、 --log-opt NAME=VALUE フラグを使います。

awslogs-region

awslogs ロギング・ドライバを使うには、リージョンの指定が必須です。リージョンを指定するにはログのオプションで awslogs-region を指定するか、環境変数 AWS_REGION を使います。

```
docker run --log-driver=awslogs --log-opt awslogs-region=us-east-1 ...
```

awslogs-group

log group[*1] を使う場合は、awslogs-group ログ・オプションを指定します。awslogs-group ログ・オプションで log group を指定します。

```
docker run --log-driver=awslogs --log-opt awslogs-region=us-east-1 \
  --log-opt awslogs-group=myLogGroup ...
```

awslogs-stream

log stream[*2] を使う場合は、awslogs-stream ログ・オプションを指定します。指定しなければ、コンテナ ID がログ・ストリームのために使われます。

ログ・ストリームに使うログ・グループはコンテナごとに指定すべきです。複数のコンテナが同じログ・ストリームを並行して使用すると、ログ記録性能が低下します。

Docker デーモンが awslogs ロギング・ドライバを使う時は、AWS の認証情報（credentials）の指定が必要です。認証情報とは環境変数 AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN 境変数です。デフォルトは AWS 共有認証ファイル（root ユーザであれば ~/.aws/credentials）か、（Amazon EC2 インスタンス上で Docker デーモンを実行するのであれば）Amazon EC2 インスタンス・プロファイルです。

認証情報には、次の例のように logs:CreateLogStream と logs:PutLogEvents の各アクションに対するポリシー追加が必要です。

*1 http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatchLogs.html

*2 http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatchLogs.html

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

7.11.4 ETW ロギング・ドライバ

ETW ロギング・ドライバはコンテナのログを ETW イベントとして転送します。ETW は Windows におけるイベント・トレース（追跡）であり、Windows 上のアプリケーションをトレースする共通フレームワークです。各 ETW イベントにはログとコンテクスト情報の両方があります。クライアントは ETW リスナーを通して、これらのイベントを確認できます。

ETW はロギング・ドライバを {a3693192-9ed6-46d2-a981-f8226c8363bd} のような GUID 識別子で Windows に登録します。クライアントは新しい ETW リスナーを作成し、ロギング・ドライバが送信するイベントを受信・登録できます。

使い方

ここでは大部分の Windows にインストール済みの logman ユーティリティ・プログラムを使い、これらのイベントのリッスン例を扱います。

1. `logman start -ets DockerContainerLogs -p {a3693192-9ed6-46d2-a981-f8226c8363bd} 0 0 -o trace.etl`

2. コンテナを etwlog ドライバと一緒に起動します。`docker run` コマンドに`--log-driver=etwlogs` を追加します。

3. `logman stop -ets DockerContainerLogs`

4. 実行するとイベントを含む etl ファイルを作成します。人間が読める形式に変換する方法の 1 つが `tracerpt -y trace.etl` の実行です。

各 ETW イベントには、次の形式のように構造化されたメッセージを含みます。

```
container_name: %s, image_name: %s, container_id: %s, image_id: %s, source: [stdout | stderr], log: %s
```

各メッセージの詳細は以下の通りです。

フィールド	説明
container_name	開始のコンテナ名
image_name	コンテナのイメージ名
container_id	64 文字のコンテナ ID
image_id	コンテナ・イメージのフル ID
source	stdout（標準出力）または stderr（標準エラー出力）
log	コンテナのログ・メッセージ

以下はイベント・メッセージの例です。

```
container_name: backstabbing_spence,
image_name: windowsservercore,
container_id: f14bb55aa862d7596b03a33251c1be7dbbec8056bbdead1da8ec5ecebbe29731,
image_id: sha256:2f9e19bd998d3565b4f345ac9aaf6e3fc555406239a4fb1b1ba879673713824b,
source: stdout,
log: Hello world!
```

クライアントはこのメッセージ文字列をログメッセージごとにパース可能です。また、コンテクスト情報も同様です。ETW イベント無いのタイムスタンプも利用可能です。

ETW プロバイダはメッセージ文字列のみ転送するだけであり、特別な ETW イベント構造ではありません。そのため、システムが ETW イベントの読み込み・受信のため、マニフェスト・ファイルを登録する必要がありません。

7.11.5 Fluentd ロギング・ドライバ

fluentd ロギング・ドライバは、コンテナのログを Fluentd[*1] コレクタに構造化したログ・データとして送信します。それから、ユーザは Fluentd の様々な出力プラグイン[*2] を使い、ログを様々な送信先に送れます。

ログ・メッセージ自身に加え、fluent ログ・ドライバは以下のメタデータを構造化ログ・メッセージの中に入れて送信できます。

フィールド	説明
container_id	64 文字の完全コンテナ ID
container_name	開始時のコンテナ名。docker rename でコンテナの名称を変えても、新しい名前は journal エントリに反映されない。
source	stdout か stderr

このロギング・ドライバの使用時は、docker logs コマンドを利用できません。

使い方

必要であれば、同じ --log-opt オプションを何度も指定可能です。

- fluentd-address ： localhost:24224 に接続する host:port を指定します。
- tag ： fluentd メッセージに送るタグを指定します。 {{.ID}}、{{.FullID}}、{{.Name}}、docker.{{.ID}} のようなマークアップ形式です。

デフォルトのロギング・ドライバを設定するには、Docker デーモンに --log-driver オプションを使います。

```
docker daemon --log-driver=fluentd
```

特定のコンテナに対してロギング・ドライバを指定する場合は、docker run に --log-driver オプションを指定します。

*1 http://www.fluentd.org/

*2 http://www.fluentd.org/plugins

```
docker run --log-driver=fluentd ...
```

このロギング・ドライバを使う前に、Fluentd デーモンを起動します。ロギング・ドライバは、デフォルトで localhost:24224 のデーモンに接続を試みます。fluentd-address オプションを使えば、異なったアドレスに接続できます。

```
docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224
```

コンテナが Fluentd デーモンに接続できなければ、コンテナは直ちに停止します。

オプション

--log-opt NAME=VALUE フラグで Fluentd ロギング・ドライバのオプションを追加できます。

fluentd-address

デフォルトでは、ロギング・ドライバは localhost:24224 に接続します。fluentd-address オプションを指定すると、異なったアドレスに接続します。

```
docker run --log-driver=fluentd --log-opt fluentd-address=myhost.local:24224
```

tag

デフォルトでは、Docker はコンテナ ID の冒頭 12 文字を tag log メッセージで使います。このログフォーマットをカスタマイズするには、「log tag オプションのドキュメント」をご覧ください。

label と env

label と env オプションは、どちらもカンマ区切りでキーを指定できます。label と env キーが重複する場合は、env の値が優先されます。どちらのオプションもロギング・メッセージの特別属性（extra attributes）に追加フィールドを加えます。

fluentd-acync-connect

Docker は Fluentd にバックグラウンドで接続します。接続が確立できるまでメッセージはバッファされます。

Docker と Fluentd デーモンの管理

Fluentd そのものについては、プロジェクトのウェブページ[*1] とドキュメント[*2] をご覧ください。

このロギング・ドライバを使うには、ホスト上に fluentd デーモンを起動します。私たちは Fluentd docker イメージ[*3] の利用を推奨します。このイメージが特に役立つのは、各ホスト上にある複数のコンテナのログを統合する場合です。そして、ログはデータを統合する用途として作成した、別の Fluentd ノードに転送できます。

1. 設定ファイル（test.conf）に入力ログをダンプするよう記述します。

```
<source>
  @type forward
</source>
```

*1 http://www.fluentd.org/

*2 http://docs.fluentd.org/

*3 https://registry.hub.docker.com/u/fluent/fluentd/

```
<match docker.**>
  @type stdout
</match>
```

2. Fluentd コンテナを、この設定を使って起動します。

```
$ docker run -it -p 24224:24224 -v /path/to/conf/test.conf:/fluentd/etc \
  -e FLUENTD_CONF=test.conf fluent/fluentd:latest
```

3. fluentd ロギング・ドライバを使うコンテナを更に起動します。

```
$ docker run --log-driver=fluentd your/application
```

7.11.6 Google Cloud ロギング・ドライバ

Google Cloud ロギング・ドライバはコンテナのログを Google Cloud Logging[*1] に送ります。

使い方

デフォルトのロギング・ドライバを設定するには、Docker デーモンに --log-driver オプションを使います。

```
docker daemon --log-driver=gcplogs
```

ロギング・ドライバをコンテナに指定するには docker run のオプションで --log-driver を指定します。

```
docker run --log-driver=gcplogs ...
```

このログ・ドライバを実装すると、docker logs コマンドでログを参照できません。

Docker が Google Cloud プロジェクトを検出すると、インスタンス・メタデータ・サービス[*2] 上で設定を見つけられるようになります。あるいは、ユーザがプロジェクトのログを記録するには --gcp-project ログ・オプションを指定し、Docker が Google Application Default Credential[*3] から証明書を得る必要があります。 --gcp-project はメタデータ・サーバによって発見される情報よりも優先します。そのため、Google Cloud Project で動いている Docker デーモンのログは、 --gcp-project を使って異なったプロジェクトに出力できます。

gpclogs オプション

Google Cloud ロギング・ドライバのオプションは、--log-opt 名前=値 の形式で指定できます。

フィールド	必須	説明
gcp-project	オプション	どの GCP プロジェクトにログを記録するか指定。デフォルトは GCE メタデータ・サービスを経由して確認された値
gcp-log-cmd	オプション	コンテナ起動時、どこにログ記録コマンドがあるか指定。デフォルトは false
lables	オプション	ラベルをコンテナに指定する場合、メッセージを含むラベルのキー一覧をカンマ区切りで
env	オプション	環境変数をコンテナに指定する場合、メッセージに含める環境変数があれば、キーの一覧をカンマ区切りで指定

*1 https://cloud.google.com/logging/docs/

*2 https://cloud.google.com/compute/docs/metadata

*3 https://developers.google.com/identity/protocols/application-default-credentials

label と env キーの間で衝突があれば、env が優先されます。いずれのオプションもロギング・メッセージの追加フィールドの属性に追加します。

以下は、GCE メタデータ・サーバで発見されたデフォルトのログ送信先を使うために必要なオプション指定の例です。

```
docker run --log-driver=gcplogs \
    --log-opt labels=location
    --log-opt env=TEST
    --log-opt gcp-log-cmd=true
    --env "TEST=false"
    --label location=west
    your/application
```

また、この設定ではラベル location、環境変数 ENV、コンテナ起動時に使うコマンドの引数も指定しています。

7.11.7 Splunk ロギング・ドライバ

splunk ロギング・ドライバは、コンテナのログを Splunk Enterprise もしくは Splunk Cloud の HTTP Event Collector[*1] に送ります。

使い方

デフォルトのロギング・ドライバを変更するには、Docker デーモンに --log-driver オプションを付けます。

```
docker daemon --log-driver=splunk
```

特定のコンテナに対してロギング・ドライバを指定するには、docker run の実行時に --log-driver オプションを付けます。

```
docker run --log-driver=splunk ...
```

Splunk オプション

Splunnk ロギング・ドライバでは、--log-opt 名前=値 フラグを指定できます。

オプション	必須	説明
splunk-token	必須	Splunk HTTP Event Collector トークンを指定
splunk-url	必須	Splunk Enterprise または Splunk Cloud インスタンスに対するパス（HTTP Event Collector の使うポートとスキームを含む）https://your_splunk_instance:8088
splunk-source	オプション	イベント・ソース
splunk-sourcetype	オプション	イベント・ソースのタイプ
splunk-index	オプション	Event インデックス
splunk-capath	オプション	root 証明書のパス
splunk-caname	オプション	サーバ証明書に使う有効な名前。デフォルトでは splunk-url で指定したホスト名が使われる
splunk-insecureskipverify	オプション	サーバ証明書の認証を無効化
splunk-tag	オプション	メッセージに対するタグを指定。いくつかのマークアップを利用可能。デフォルトの値は {{.ID}}（コンテナ ID の 12 文字）。ログのタグ書式のカスタマイズ方法は、「ログ用のタグ」を参照
splunk-labels	オプション	メッセージに含めるラベルを、カンマ区切りのキーで指定できる。

*1 http://dev.splunk.com/view/event-collector/SP-CAAAE6M

		ラベルはコンテナで指定できる
splunk-env	オプション	メッセージに含める環境変数を、カンマ区切りのキーで指定できる。これらの変数はコンテナで指定できる

label と env のキーが重複する場合は、env が優先されます。どちらのオプションも、ロギング・メッセージの追加フィールドに追加できるものです。

以下はロギング・ドライバに Splunk Enterprise インスタンスを指定する例です。インスタンスは Docker デーモンを実行している同じマシン上のローカルにインストールされています。root 証明書とコモンネームの指定には HTTP スキームを使います。これが証明書の使い方です。SplunkServerDefaultCert は自動的に生成された Splunk 証明書です。

```
docker run --log-driver=splunk \
    --log-opt splunk-token=176FCEBF-4CF5-4EDF-91BC-703796522D20 \
    --log-opt splunk-url=https://splunkhost:8088 \
    --log-opt splunk-capath=/path/to/cert/cacert.pem \
    --log-opt splunk-caname=SplunkServerDefaultCert
    --log-opt tag="{{.Name}}/{{.FullID}}"
    --log-opt labels=location
    --log-opt env=TEST
    --env "TEST=false"
    --label location=west
    your/application
```

7.11.8 Journald ロギング・ドライバ

journald ロギング・ドライバは、コンテナログを systemd journal[*1] に送信します。ログエントリは、journalctl コマンドを使ってログエントリを確認できます。使うには journal API か docker logs コマンドを通します。

ログメッセージ自身の文字列に加え、journald ログドライバは各メッセージの journal に以下のメタデータを保管できます。

フィールド	説明
CONTAINER_ID	コンテナ ID の先頭から 12 文字
CONTAINER_ID_FULL	64 文字の完全コンテナ ID
CONTAINER_NAME	開始時のコンテナ名。docker rename でコンテナの名称を変えても、新しい名前は journal エントリに反映されない。
CONTAINER_TAG	コンテナのタグ

使い方

デフォルトのロギング・ドライバを設定するには、Docker デーモンに --log-driver オプションを使います。

```
docker daemon --log-driver=journald
```

特定のコンテナに対してロギング・ドライバを指定する場合は、docker run に --log-driver オプションを指定します。

```
docker run --log-driver=journald ...
```

*1 http://www.freedesktop.org/software/systemd/man/systemd-journald.service.html

オプション

--log-opt NAME=VALUE フラグで journald ロギング・ドライバのオプションを追加できます。

タグ

journald のログに CONTAINER_TAG 値でテンプレートを指定します。ログのタグ形式をカスタマイズするには「ログ用タグのオプション」についてのドキュメントをご覧ください。

label と env

label と env オプションは、どちらもカンマ区切りでキーを指定できます。label と env キーが重複する場合は、env の値が優先されます。どちらのオプションもロギング・メッセージの特別属性（extra attributes）に追加フィールドを加えます。

CONTAINER_NAME フィールドに記録される値は、起動時のコンテナ名が使われます。docker rename でコンテナの名前を変更しても、journal エントリの名前には反映されません。journal のエントリはオリジナルの名前を表示し続けます。

journalctl コマンドを使って、ログメッセージを表示できます。フィルタ表現を追加することで、特定のコンテナに関するメッセージしか表示しないようにできます。例えば、特定のコンテナ名に関する全てのメッセージを表示するには、次のようにします。

```
# journalctl CONTAINER_NAME=webserver
```

メッセージの制限だけでなく、他のフィルタも利用できます。例えば、システムが直近でリブートした以降のメッセージを生成するには、次のように実行します。

```
# journalctl -b CONTAINER_NAME=webserver
```

あるいは、全てのメタデータを含む JSON 形式でメッセージを表示するには、次のようにします。

```
# journalctl -o json CONTAINER_NAME=webserver
```

この例は systemd Python モジュールを使い、コンテナのログを取得しています。

```
import systemd.journal

reader = systemd.journal.Reader()
reader.add_match('CONTAINER_NAME=web')

for msg in reader:
  print '{CONTAINER_ID_FULL}: {MESSAGE}'.format(**msg)
```

8 章

セキュリティ

8.1 安全な Engine

このセクションは Docker Engine のセキュリティ機能や、インストール時の設定について扱います。

- トラステッド・イメージ（trusted image）を送受信できる機能を設定することで、Docker の信頼性を高めます。設定の詳細は「トラステッド・イメージを使う」をご覧ください。

- Docker デーモン・ソケットを守り、確かなものとするには信頼された Docker クライアント接続を使います。詳細は「Docker デーモンのソケットを守る」をご覧ください。

- 証明書をベースとするクライアント・サーバ認証を使えます。これは Docker デーモンがレジストリ上のイメージに対する適切なアクセス権があるかどうかを確認します。詳細は「証明書をリポジトリのクライアント認証に使用」をご覧ください。

- セキュア・コンピューティング・モード（Seccomp）ポリシーを設定することで、コンテナ内のシステムコールを安全にします。詳細な情報は「Docker 用の seccomp セキュリティ・プロフィール」をご覧ください。

- 公式の .deb パッケージは、Docker 用の AppArmor プロファイルがインストールされます。プロファイルや更新方法は「Docker 用の apparmor セキュリティ・プロフィール」をご覧ください。

8.2 Docker のセキュリティ

Docker のセキュリティには、主に 3 つの検討項目があります。

- カーネルに起因するセキュリティと、カーネルがサポートする名前空間と cgroups について
- Docker デーモン自身が直面する攻撃について
- コンテナ設定プロファイル（デフォルトでもユーザによってカスタマイズされた場合も含む）における抜け道について
- カーネルのセキュリティ「硬化」（hardening）機能と、コンテナへの対応。

Docker コンテナは LXC コンテナに非常に似ており、類似のセキュリティ機能を持っています。コンテナを `docker run` で起動する時、その背後で Docker がコンテナ向けの名前空間とコントロール・グループを作成します。

名前空間は、一流かつ最も簡単な方法で分離（isolation）を提供します。これによりコンテナの中で実行しているプロセスは、他のコンテナやホスト上のプロセスから見えなくなり、影響すら受けません。

各コンテナは自分自身のネットワーク・スタックを持ちます。つまり、コンテナはソケットや他のコンテナのインターフェースに対する特権（privileged）アクセスが得られません。もちろん、ホストシステムが適切に設定されている必要があります。そうしておけば、コンテナが相互に適切なネットワーク・インターフェースを通して通信できるようになります。ホストの外と通信できるのも同様です。コンテナに対して公開用のポートを指定するか、リンク機能を使うことで、コンテナ間での IP 通信が許可されます。お互いに ping できるようになり、UDP パケットの送受信や、TCP 接続が確立されます。しかし、必要があれば制限を設けられます。ネットワーク・アーキテクチャの視点から考えますと、全てのコンテナは特定のホスト上のブリッジ・インターフェースを備えています。つまりこれは、物理マシン上で共通のイーサネット・スイッチを使っているのと同じような状態を意味します。それ以上でも、それ以下でもありません。

カーネルの名前空間を提供するコードやプライベート・ネットワーキングの成熟度とは、どの程度でしょうか。カーネルの名前空間は カーネル 2.6.15 から 2.6.26 の間[1] に導入されました。これが意味するのは、2008 年 6 月にリリースされた（2.6.26 がリリースされたのは、今から 7 年前です）名前空間のコードは、多数のプロダクション・システム上で動作・精査されています。更にもう 1 つ。名前区間コードの設計と発想はやや古いものです。名前空間が効果的に実装された例としては OpenVZ[2] があり、カーネルのメインストリームとしてマージされたこともありました。OpenVZ の初期リリースは 2005 年であり、設計と実装は、多少成熟していると言えるでしょう。

8.2.2 コントロール・グループ

コントロール・グループは Linux コンテナにおけるもう 1 つの重要なコンポーネントです。これはリソースの計測と制限を実装しています。これらは多くの便利なメトリクス（監視上の指標）を提供するだけでなく、各コンテナが必要な共有リソース（メモリ、CPU、ディスク I/O）の割り当て保証にも役立ちます。更に重要なのは、単一のコンテナが膨大なリソースを消費しても、システムダウンを引き起こさない点です。

そのため、あるコンテナが他のコンテナからアクセスできませんし、データに対する何らかのアクセスや影響を及ぼすこともありません。これはあらゆるサービス拒否（denial-of-service）攻撃の本質です。特に重要なのはマルチテナントなプラットフォーム、例えばパブリックやプライベートな PaaS において、特定のアプリケーションが

*1 http://lxc.sourceforge.net/index.php/about/kernel-namespaces/

*2 http://ja.wikipedia.org/wiki/OpenVZ

誤った動作をしても、一定の稼働（とパフォーマンス）を保証します。

コントロール・グループも同様に、以前から存在していました。コードは 2006 年から書き始めら、カーネルに 2.6.24 で初めてマージされました。

Docker を使ったコンテナ（とアプリケーション）を実行するとは、Docker デーモンの稼働を意味します。このデーモンは現時点で root 特権が必要であり、それゆえ、いくつか重要な点に配慮が必要です。

まずはじめに、**信頼する利用者だけ、Docker デーモンに対するアクセスを許可するべき**です。これは Docker がもたらす強力な機能を直接扱うためです。特に、Docker は Docker ホストとゲストコンテナ間でディレクトリを共有できます。そして、それにより、コンテナ内に対する適切なアクセス権限が無くても、ディレクトリを使えるようになる可能性があります。つまりコンテナの **/host** ディレクトリは、ホスト上の / ディレクトリとしても実行可能です。それだけではありません。コンテナは何ら制限を受けずに、ホスト上のファイルシステム上に対する修正が可能になります。これは仮想化システムによるファイルシステム・リソースの共有に似ています。仮想マシン上における自分のルート・ファイルシステム（ルート・ブロック・デバイスも同様）の共有を阻止する方法はありません。

これはセキュリティに重大な影響を及ぼします。例えば、Docker の API を通してウェブ・サーバ用コンテナをプロビジョンしたいとします。通常通りパラメータの確認に注意を払うべきです。ここでは、悪意のあるユーザが手の込んだパラメータを使い、Docker が余分なコンテナを作成不可能にしてください。

この理由により、REST API エンドポイント（Docker CLI が Docker デーモンとの通信に使います）が Docker 0.5.2 で変更されました。現在は 127.0.0.1 上の TCP ソケットに代わり、UNIX ソケットを使います（最近はローカルのマシン上の Docker に対して、仮想マシンの外から直接クロスサイト・リクエスト・フォージェリ、CSRF を行う傾向があります)。伝統的な Unix パーミッションを確認し、ソケットに対するアクセスを制限するような管理が必要です。

明示的に HTTP 上で REST API を晒すことも可能です。しかし、そのように設定すべきではありません。上記で言及したセキュリティ実装のため、信頼できるネットワークや VPN、`stunnel` やクライアント SSL 証明が利用できる所でのみ使うべきです。より安全にするためには HTTPS と証明書を利用できます。

また、デーモンは入力に関する脆弱性を潜在的に持っています。これはディスク上で `docker load`、あるいはネットワーク上で `docker pull` を使いイメージを読み込む時です。これはコミュニティにおける改良に焦点がおかれており、特に安全に `pull` するためです。これまでの部分と重複しますが、`docker load` はバックアップや修復のための仕組みです。しかし、イメージの読み込みにあたっては、現時点で安全な仕組みではないと考えられていることに注意してください。Docker 1.3.2 からは、イメージは Linux/Unix プラットフォームの chroot サブ・プロセスとして展開されるようになりました。これは広範囲にわたる特権分離問題に対する第一歩です。

最終的には、Docker デーモンは制限された権限下で動作するようになるでしょう。それぞれが自身の（あるいは限定された）Linux **ケーパビリティ（capability；「能力」や「機能」の意味）**、仮想ネットワークのセットアップ、ファイルシステム管理といった、サブプロセスごとに委任したオペレーションを監査できるようになることを期待しています。

なお、Docker をサーバで動かす場合は、サーバ上で Docker 以外を動かさないことを推奨します。そして、他のサービスは Docker によって管理されるコンテナに移動しましょう。もちろん、好きな管理ツール（おそらく SSH サーバでしょう）や既存の監視・管理プロセス（例：NRPE、collectd、等）はそのままで構いません。

8.2.4 Linux カーネルのケーパビリティ

デフォルトでは Docker はケーパビリティ（capability；「能力」や「機能」の意味）を抑えた状態でコンテナを起動します。つまり、これはどのような意味でしょうか。

ケーパビリティとは、「root」か「root 以外か」といったバイナリの二分法によって分類する、きめ細かなアクセ

ス制御システムです。(ウェブサーバのような) プロセスがポート 1024 以下でポートをバインドする必要がある時、root 権限でなければ実行できません。そこで `net_bind_service` ケーパビリティを使い、権限を与えます。他にも多くのケーパビリティがあります。大部分は特定の条件下で root 特権を利用できるようにするものです。

つまり、コンテナのセキュリティを高めます。理由を見ていきましょう！

あなたの平均的なサーバ（ベアメタルでも、仮想マシンでも）が必要とするのは、root として実行される一連のプロセスです。典型的なものに SSH、cron、syslogd が含まれるでしょう。あるいは、ハードウェア管理ツール（例：load モジュール)、ネットワーク設定ツール（例：DHCP、WPA、VPN を取り扱うもの)、等々があります。ですが、コンテナは非常に異なります。なぜなら、これらのタスクのほぼ全てが、コンテナの中という基盤上で処理されるからです。

- SSH 接続は、 Docker ホストのサーバ上を管理する典型的な手法です。
- cron は、必要があればユーザ・プロセスとして実行可能です。プラットフォーム上のファシリティを広範囲に使うのではなく、専用、もしくはアプリケーションが個別に必要なサービスをスケジュールします。
- ログ管理もまた Docker の典型的な処理であり、あるいはサードパーティー製の Loggly や Splunk を使うでしょう。
- ハードウェア管理には適していません。これはコンテナ内で `udevd` や同等のデーモンを実行できないためです。
- ネットワーク管理はコンテナの外で行われので、懸念されうる事項を分離します。つまり、コンテナでは `ifconfig`、`route`、`ip` コマンドを実行する必要がありません（ただし、コンテナでルータやファイアウォール等の振る舞いを処理させる場合は、もちろん除きます)。

これらが意味するのは、大部分のケースにおいて、コンテナを「本当の」root 特権で動かす必要は全く無いということです。それゆえ、コンテナはケーパビリティの組み合わせを減らして実行できるのです。つまり、コンテナ内の「root」は、実際の「root」よりも権限が少ないことを意味します。例えば、次のような使い方があります。

- 全ての「mount」操作を拒否
- raw ソケットへのアクセスを拒否（パケット・スプーフィングを阻止）
- ファイルシステムに関するいくつかの操作を拒否（新しいデバイス・ノードの作成、ファイル所有者の変更、immutable フラグを含む属性の変更)
- モジュールの読み込みを禁止
- などなど

これが意味するのは、侵入者がコンテナ内で root に昇格しようとしても、深刻なダメージを与えるのが困難であり、ホストにも影響を与えられません。

通常のウェブ・アプリケーションには影響を与えません。しかし、悪意のあるユーザであれば、自分たちが自由に使える武器が減ったと分かるでしょう！ Docker は必要に応じて[*1] 全てのケーパビリティを除外し、ブラックリストからホワイトリストに除外する方法も使えます。利用可能なケーパビリティについては、Linux の man ページ[*2]

*1 https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/default_template.go

*2 http://man7.org/linux/man-pages/man7/capabilities.7.html

をご覧ください。

Docker コンテナ実行にあたり、最も重要なリスクというのは、デフォルトのケーパビリティのセットとコンテナに対するマウントにより、不完全な分離（独立性、あるいは、カーネルの脆弱性と組み合わせ）をもたらすかもしれない点です

Docker はケーパビリティの追加と削除をサポートしますので、デフォルトで何も無いプロファイルも扱えます。これにより、ケーパビリティが削除されても Docker は安全ですが、ケーパビリティを追加する時はセキュリティが低下します。利用にあたってのベストプラクティスは、各プロセスが明らかに必要なケーパビリティを除き、全て削除することです。

ケーパビリティは、最近の Linux カーネルで提供されている、様々なセキュリティ機能の 1 つです。他にも既存のよく知られている TOMOYO、AppArmor、SELinux、GRSEC のようなシステムが Docker で使えます。

現時点の Docker はケーパビリティの有効化しかできず、他のシステムには干渉できません。つまり、Docker ホストを堅牢にするには様々な異なった方法があります。以下は複数の例です。

- カーネルで GRSEC と PAX を実行できます。これにより、コンパイル時と実行時の安全チェック機能をもたらします。アドレスランダム化のような技術に頼る、多くの exploit を無効化します。Docker 固有の設定は不要です。コンテナとは独立して、システムの広範囲にわたるセキュリティ機能を提供します。

- ディストリビューションに Docker コンテナに対応したセキュリティ・モデル・テンプレートがあれば、それを利用可能です。例えば、私たちは AppArmor で動作するテンプレートを提供しています。また、Red hat は Docker 対応の SELinux ポリシーを提供しています。これらのテンプレートは外部のセーフティーネットを提供します（ケーパビリティと大いに重複する部分もありますが）。

- 好みのアクセス管理メカニズムを使って、自分自身でポリシーを制限できます。

Docker コンテナと連携する多くのサードパーティー製ツールが提供されています。例えば、特別なネットワーク・トポロジーや共有ファイルシステムです。これらは Docker のコアの影響を受けずに、既存の Docker コンテナを堅牢にするものです。

Docker 1.10 以降は Docker デーモンがユーザ名前空間（User Namespaces）を直接サポートしました。この機能により、コンテナ内の root ユーザをコンテナ外の uid 0 以外のユーザに割り当て（マッピング）できるようになります。コンテナからブレイクアウト（脱獄）する危険性を軽減する手助けとなるでしょう。この実装は利用可能ですが、デフォルトでは有効ではありません。

こちらの機能に関するより詳しい情報は daemon コマンド のリファレンスをご覧ください。Docker におけるユーザ名前空間の実装に関する詳細情報は、ブログ投稿記事[*1] をご覧ください。

8.2.6 まとめ

Docker コンテナはデフォルトでも安全ですが、とりわけコンテナ内でプロセスを権限の無いユーザ（例：root 以外のユーザ）で実行する時に、注意が必要です。

AppArmor、SELinux、GRSEC など任意の堅牢化ソリューションを有効化することで、更に安全なレイヤを追加できます。

*1 https://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/

最後ですが疎かにできないのは、他のコンテナ化システムのセキュリティ機能に興味があれば、それらは Docker と同じようにシンプルにカーネルの機能を実装しているのが分かるでしょう。私たちは皆さんからの問題報告、プルリクエスト、メーリングリストにおける議論を歓迎します。

8.3 Docker デーモンのソケットを守る

デフォルトでは、ネットワークを通さない Unix ソケットで Docker を操作します。オプションで HTTP ソケットを使った通信も可能です。

Docker をネットワーク上で安全な方法で使う必要があるなら、TLS を有効にすることもできます。そのためには、tlsverify フラグの指定と、tlscacert フラグで信頼できる CA 証明書を Docker に示す必要があります。

デーモン・モードでは、その CA で署名された証明書を使うクライアントのみ接続可能にします。クライアント・モードでは、その CA で署名されたサーバのみ接続可能にします。

【警告】TLS と CA の管理は高度なトピックです。プロダクションで使う前に、自分自身で OpenSSL、x509、TLS に慣れてください。

【警告】各 TLS コマンドは Linux 上で作成された証明書のセットのみ利用可能です。Mac OS X は Docker デーモンが必要な OpenSSL のバージョンと互換性がありません。

以下の例にある $HOST は、自分の Docker デーモンが動いている DNS ホスト名に置き換えてください。

まず、CA 秘密鍵と公開鍵を作成します。

```
$ openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................................
.....................................................................................++
........++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:Queensland
Locality Name (eg, city) []:Brisbane
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Docker Inc
Organizational Unit Name (eg, section) []:Sales
Common Name (e.g. server FQDN or YOUR name) []:$HOST
Email Address []:Sven@home.org.au
```

これで CA を手に入れました。サーバ鍵（server key）と証明書署名要求（CSR; certificate signing request）を

作成しましょう。接続先の Docker ホスト名が **Common Name**（例：サーバの FQDN や自分自身の名前）に一致しているのを確認します。

以下の例にある $HOST は、自分の Docker デーモンが動いている DNS ホスト名に置き換えてください。

```
$ openssl genrsa -out server-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................++
.....................................................................................++
e is 65537 (0x10001)
$ openssl req -subj "/CN=$HOST" -sha256 -new -key server-key.pem -out server.csr
```

次に公開鍵を CA で署名しましょう。

TLS 接続は DNS 名と同様に、IP アドレスでも通信可能にできます。その場合は、証明書に情報を追加する必要があります。例えば、 10.10.10.20 と 127.0.0.1 を使う場合は次のようにします。

```
$ echo subjectAltName = IP:10.10.10.20,IP:127.0.0.1 > extfile.cnf

$ openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem \
-CAcreateserial -out server-cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=your.host.com
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

クライアント認証用に、クライアント鍵と証明書署名要求を作成します。

```
$ openssl genrsa -out key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....................................................++
.............++
e is 65537 (0x10001)
$ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

クライアント認証用の鍵を実装するには、追加設定ファイルを作成します。

```
$ echo extendedKeyUsage = clientAuth > extfile.cnf
```

次は公開鍵に署名します。

```
$ openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem \
  -CAcreateserial -out cert.pem -extfile extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

cert.pem と server-cert.pem を生成したら、証明書署名要求を安全に削除できます。

```
$ rm -v client.csr server.csr
```

デフォルトの umask は 022 のため、秘密鍵は自分と同じグループから読み書き可能です。

何らかのアクシデントから自分の鍵を守るため、書き込みパーミッションを削除します。自分だけしか読み込めないようにするには、ファイルモードを次のように変更します。

```
$ chmod -v 0400 ca-key.pem key.pem server-key.pem
```

証明書は誰でも読み込めても問題ありませんが、予期しないアクシデントによる影響を避けるため、書き込み権限を削除します。

```
$ chmod -v 0444 ca.pem server-cert.pem cert.pem
```

あとは Docker デーモンを、自分たちの CA を使って署名した信頼できるクライアントしか接続できないようにします。

```
$ docker daemon --tlsverify --tlscacert=ca.pem --tlscert=server-cert.pem --tlskey=server-key.pem \
  -H=0.0.0.0:2376
```

これは Docker に接続する時、証明書の認証を必要とするものです。認証には先ほどのクライアント鍵、証明書、信頼できる CA を使います。

```
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --tlskey=key.pem \
  -H=$HOST:2376 version
```

【警告】上記の例では docker クライアントの実行に sudo が不要か、あるいは認証に使うユーザが docker グループに属しています。つまり、鍵を使って Docker デーモンにアクセス可能にするとは、デーモンを動かしているマシンの root 権限を与えるのを意味します。これらの鍵は root パスワード同様に保護してください！

8.3.2 デフォルトで安全に

Docker クライアントの接続をデフォルトで安全にしたい場合は、自分のホームディレクトリ直下の `.docker` ディレクトリにファイルを移動できます。そして、環境変数 `DOCKER_HOST` と `DOCKER_TLS_VERIFY` を使います（毎回 `-H=tcp://$HOST;2376` や `--tlsverify` を実行する代わりになります）。

```
$ mkdir -pv ~/.docker
$ cp -v {ca,cert,key}.pem ~/.docker
$ export DOCKER_HOST=tcp://$HOST:2376 DOCKER_TLS_VERIFY=1
```

こうしておけば、デフォルトで Docker は安全に接続します。

```
$ docker ps
```

8.3.3 他のモード

双方向認証を有効にしたくない場合、他のフラグと組み合わせて Docker を実行できます。

デーモン・モード

- `tlsverify`、`tlscacert`、`lscert`、`tlskey` をセット：クライアントを認証する
- `tls`、`tlscert`、`tlskey`：クライアントを認証しない

クライアント・モード

- tls：サーバをベースとした公開/デフォルト CA プールで認証
- tlsverify、tlscacert ：サーバをベースとした CA 認証
- tls、tlscert、tlskey ：クライアント認証を使い、サーバ側を指定した CA では認証しない
- tlsverify、tlscacert、tlscert、tlskey ：クライアント証明書と、サーバ側で指定した CA で認証する

クライアントがクライアント証明書を送信したら、自分の鍵を ~/.docker/{ca,cert,key}.pem に移動します。あるいは、別の場所に保管し、環境変数 DOCKER_CERT_PATH でも指定できます。

```
$ export DOCKER_CERT_PATH=~/.docker/zone1/
$ docker --tlsverify ps
```

curl を使って Docker ポートに安全に接続

curl を API リクエストのテストに使うには、コマンドラインで 3 つの追加フラグが必要です。

```
$ curl https://$HOST:2376/images/json \
  --cert ~/.docker/cert.pem \
  --key ~/.docker/key.pem \
  --cacert ~/.docker/ca.pem
```

これまで Docker を HTTPS で動かす 方法を学びました。デフォルトでは Docker はネットワークで使えない Unix ソケットについてと、Docker クライアントとデーモンが安全に通信できるように、HTTPS 上で TLS を有効化すべきという内容でした。TLS はレジストリのエンドポイントにおける認証を確実なものとし、かつ、レジストリからあるいはレジストリへの通信を暗号化します。

このセクションでは、Docker レジストリ（例：サーバ）と Docker デーモン（例：クライアント）間でのトラフィックを暗号化する方法と、証明書をベースとしたクライアント・サーバ認証の仕方を扱います。

認証局（CA; Certificate Authority）のルート証明書をレジストリに設定する方法と、クライアント側で TLS 証明書を使って認証する方法を紹介します。

任意の証明書を使うように設定するには、 `/etc/docker/certs.d` ディレクトリの下に、レジストリのホスト名と同じ名前のディレクトリ（例：`localhost`）を作成します。このディレクトリを CA ルートとして、全ての`*.crt` ファイルを追加します。

認証に関する root 証明書が足りなければ、Docker はシステムのデフォルトを使います（例：ホスト側のルート CA セット）。

任意のリポジトリにアクセスするには、Docker が 1 つ以上の `<ファイル名>.key/cert` のセットを認識する必要があります。

複数の証明書があれば、アルファベット順で処理を行います。もし、認証エラー（例： 403、404、5xx、等）があれば、次の証明書で処理を試みます。

以下は、複数の証明書がある場合の設定です。

```
/etc/docker/certs.d/        <-- 証明書のディレクトリ
 └──── localhost                <-- ホスト名
    ├──── client.cert           <-- クライアントの証明書（certificate）
    ├──── client.key            <-- クライアント鍵
    └──── localhost.crt         <-- 認証局によるレジストリ証明書への署名
```

なお、この例は典型的なオペレーティング・システム上で使う場合を想定したものです。OS が提供する証明書チェーン作成に関するドキュメントを確認すべきでしょう。

まず、OpenSSL の `genrsa` と `req` コマンドを使って RSA 鍵を生成し、それを証明書の作成に使います。

```
$ openssl genrsa -out client.key 4096
$ openssl req -new -x509 -text -key client.key -out client.cert
```

これらの TLS コマンドが生成する証明書は、Linux 上で動作するものです。Mac OS X に含まれている OpenSSL のバージョンでは、Docker が必要とする証明書のタイプと互換性はありません。

Docker デーモンは `.crt` ファイルを CA 証明書として認識し、 `.cert` ファイルをクライアント証明書として認識します。CA 証明書が正しい `.crt` 拡張子ではなく `.cert` 拡張子になれば、Docker デーモンは次のようなエラーメッセージをログに残します。

```
Missing key KEY_NAME for client certificate CERT_NAME. Note that CA certificates should use the
extension .crt.
```

9 章

Docker Engine の拡張

プラグインを追加することにより Docker を拡張できます。このセクションは以下の話題を扱います。

- Docker プラグインの理解
- Docker ボリューム・プラグインを書く
- Docker ネットワーク・プラグイン
- 認証
- Docker プラグイン API

サードパーティー製のプラグインを読み込むと、Docker Engine の機能を拡張できます。このページではプラグインの種類についてと、いくつかの Docker 向けボリュームとネットワークのプラグインのリンクを紹介します。

プラグインは Docker の機能性を拡張します。拡張機能には複数の種類があります。例えば、ボリューム・プラグインは、複数のホストを横断して存在する Docker ボリュームを有功にします。ネットワーク・プラグインはネットワークの管（plumbing）を提供するでしょう。

現時点の Docker はボリュームとネットワーク・ドライバのプラグインをサポートしています。近いうちに、他のプラグインの種類もサポートするでしょう。

9.1.2 プラグインのインストール

設定方法は、各プラグインのドキュメントをご覧ください。

9.1.3 プラグインを探す

次のようなプラグインがあります。

- Blockbridge plugin[*1] はボリューム・プラグインです。コンテナをベースとした持続型のストレージ向けオプション、その拡張セットへのアクセスを提供します。1 つまたは複数の Docker 環境で、テナントの分離、自動プロビジョニング、暗号化、安全な削除、スナップショット、QoS といった機能を提供します。

*1 https://github.com/blockbridge/blockbridge-docker-volume

- Convoy plugin[*1] はボリューム・プラグインです。device mapper や NSF を含む様々なストレージ・バックエンドに対応します。スタンドアローンのバイナリはシンプルな Go 言語で書かれています。スナップショット、バックアップ、リストアといったベンダ固有の拡張をサポートしたフレームワークを提供します。

- Flocker plugin[*2] は Docker 対応の複数ホストで、ボリュームをポータブルに持ち運ぶためのプラグインです。これにより、データベースや他のステートフル（状態を持たない）なコンテナを、クラスタ上のマシンにまたがって実行できるようにします。

- GlusterFS plugin[*3] は Docker が GlusterFS を使って複数ホストのボリュームを管理可能にするプラグインです。

- Kyewhiz plugin[*4] は Keywhiz を中央リポジトリとして、証明書やシークレット（秘密情報）の管理を提供するプラグインです。

- Netshare plugin[*5] は NFS v3/v4、AWS FEC、CIFS ファイルシステムでボリュームを管理するプラグインです。

- OpenStorage Plugin[*6] はクラスタ検出ボリューム・プラグインであり、ファイルやブロック・ストレージにおけるボリューム管理ソリューションを提供します。扱えるのは、ベンダ中立の拡張機能です。例えば CoS、暗号化、スナップショットです。サンプル・ドライバがベースにしているのは、FUSE、NFS、NBD、EBS などです。

- Pachyderm PFS plugin[*7] は Go 言語で書かれたボリューム・プラグインです。PFS（Pachyderm File System）リポジトリにマウントできる機能を提供します。Docker コンテナがなくても、ボリュームに対するコミットを行えるようにします。

- REX-Ray plugin[*8] は Go 言語で書かれたボリューム・プラグインです。ES2、OpenStack、XtreamIO、ScaleIO を含む多くのプラットフォームに対応した高度なストレージ機能を提供します。

- Contiv Volume Plugin[*9] はオープンソースのボリューム・プラグインです。ceph をベースとした技術により、マルチ・テナントで、永続型、分散したストレージを提供します。

- Contiv Networking[*10] はオープンソースの libnetwork プラグインであり、マルチ・テナントのマイクロサービスのデプロイにおけるインフラとセキュリティ・ポリシーを提供します。この環境では、コンテナに対

---

*1 https://github.com/rancher/convoy

*2 https://clusterhq.com/docker-plugin/

*3 https://github.com/calavera/docker-volume-glusterfs

*4 https://github.com/calavera/docker-volume-keywhiz

*5 https://github.com/gondor/docker-volume-netshare

*6 https://github.com/libopenstorage/openstorage

*7 https://github.com/pachyderm/pachyderm/tree/master/src/cmd/pfs-volume-driver

*8 https://github.com/emccode/rexraycli

*9 https://github.com/contiv/volplugin

*10 https://github.com/contiv/netplugin

する負担無しに、物理ネットワークの統合をもたらします。Contiv Networking は Docker 1.9 以降で利用可能な リモート・ドライバと IPAM API を実装しています。

- Weave Network Plugin[*1] は Docker コンテナを結ぶ仮想ネットワークを作成します。これは複数のホストやクラウドをまたがり、アプリケーションの自動的な発見を可能にします。Weave network は弾力性（resilient）があり、分散耐性（partition tolerant）があり、安全で、部分的なネットワークでも利用できます。他のツールによる環境と異なり、全ての設定が極めて単純です。

9.1.4 プラグインのトラブルシューティング

プラグインを読み込んだ後で Docker に問題が起こったら、プラグインの作者に助けを求めてください。Docker チームはあなたを助けられません（訳者注：Docker コミュニティ外のツールのため）。

9.1.5 プラグインを書くには

Docker プラグインを書くことに興味があれば、あるいは、水面下でどのような処理がされているかに興味があれば、docker プラグイン・リファレンスをご覧ください。

*1 https://github.com/weaveworks/docker-plugin

9.2 Docker ネットワーク・プラグイン

Docker ネットワーク・プラグインは、Docker が広範囲のネットワーク技術のサポートによりデプロイできるように拡張されています。XVLAN、IPVLAN、MACVLAN、あるいはこれらとも全く異なります。ネットワーク・ドライバ・プラグインは LibNetwork プロジェクトによってサポートされています。各ネットワーク・プラグインは LibNetwork の「リモート・ドライバ」であり、これは Docker とプラグイン基盤を共有するものです。効果的なのは、プラグイン基盤の共有によって、他のプラグインを同じように扱え、かつ同様のプロトコルを扱えることです。

9.2.1 ネットワーク・ドライバ・プラグインを使う

ネットワーク・ドライバ・プラグインのインストール実行は、個々のプラグインに依存します。そのため、プラグインをインストールするには、各プラグインの開発者から指示を得てください。

ネットワーク・ドライバを実行しても、内部ネットワーク・ドライバのように扱えない可能性があります。ネットワーク対応の Docker コマンドでドライバを操作します。例えば、次のようなコマンドです。

```
$ docker network create --driver weave mynet
```

ネットワーク・ドライバのプラグイン一覧は、 「9.1.3 プラグインを探す」をご覧ください。

`mynet` ネットワークは `weave` の所有となりましたので、以下に続くコマンドは、プラグインの管理するネットワークに対して送信されます。

```
$ docker run --net=mynet busybox top
```

9.2.2 ネットワーク・プラグインを書くには

ネットワーク・プラグインの実装は、Docker プラグイン API のネットワーク・プラグイン・プロトコルをご覧ください。

9.2.3 ネットワーク・プラグイン・プロトコル

ネットワーク・ドライバ・プロトコルとは、プラグイン・アクティベーション・コール（plugin activation call）の追加です。詳細については libnetwork の該当ドキュメント[*1] をご覧ください。

*1 https://github.com/docker/libnetwork/blob/master/docs/remote.md

9.3 Docker ボリューム・プラグインを書く

Docker ボリューム・プラグインとは、Amazon EBS のような外部のストレージ・システムと統合した環境に Docker をデプロイできるようにします。そして、単一の Docker ホスト上で、データ・ボリュームを使う間はその一貫性をもたらします。詳しい情報はプラグインのドキュメントをご覧ください。

9.3.1 コマンドラインの変更

ボリューム・プラグインを使うには docker run コマンドで -v と --volume-driver フラグを指定します。 -v フラグはボリューム名を受け付け、 --volume-driver フラグはドライバの種類を指定します。例えば、次のように実行します。

```
$ docker run -ti -v volumename:/data --volume-driver=flocker   busybox sh
```

このコマンドは、ユーザがボリュームで使う名前を volumename としてボリューム・プラグインに渡しています。volumename は / で始まってはいけません。

ユーザが volumename を指定したら、プラグインは 1 つのコンテナが稼働し続ける間、あるいはコンテナのホスト上における外部ボリュームをプラグインに関連づけます。これを使えば、例えばステートフルなコンテナを、あるサーバから別のサーバに移せます。

volumename と volumedriver を同時に使うよう指定したら、ユーザは Flocker[*1] のような外部プラグインで単一ホスト上のボリュームや EBS のようなボリュームを管理します。

9.3.2 ボリューム・ドライバの作成

コンテナが作成用エンドポイント（/containers/create）の volumeDriver フィールドにおいて、string タイプでドライバ名を指定します。デフォルトの値は "local" です（デフォルトのドライバは、local ボリュームです）。

9.3.3 ボリューム・プラグイン・プロトコル

プラグインは自身を VolumeDriver として登録した時に有効化されます。その後、Docker デーモンがファイルシステム上に、コンテナが使うための書き込み可能なパスを提供します。

Docker デーモンはユーザのコンテナが指定したパスに対し、マウントの拘束（バインド）を扱います。

/VolumeDriver.Create

リクエスト：

```
{
    "Name": "volume_name",
    "Opts": {}
}
```

プラグインはユーザが作成を望むボリュームを、ユーザが指定した名前で作成するよう命令します。プラグインは実際にファイルシステムのボリュームを明示する必要がありません（マウントがコールされるまで）。Opts はドライバ固有のオプションをユーザがリクエストする箇所です。

*1 https://clusterhq.com/docker-plugin/

応答 :

```
{
    "Err": null
}
```

エラーが発生した場合は、エラー文字列が表示されます。

/VolumeDriver.Remove

リクエスト :

```
{
    "Name": "volume_name"
}
```

ディスクから特定のボリュームを削除します。このリクエストはユーザから `docker rm -v` を呼び出された時、コンテナに関連するボリュームを削除します。

応答 :

```
{
     “Err” : null

}
```

エラーが発生した場合は、エラー文字列が表示されます。

/VolumeDriver.Mount

リクエスト :

```
{
    "Name": "volume_name"
}
```

Docker でプラグインがボリュームを必要とする場合は、ユーザがボリューム名を指定する必要があります。これは、コンテナが開始される度に必要です。既に作成されているボリューム名で呼び出されると、プラグインは既にマウントされている箇所に対して、新しいマウント・リクエストとプロビジョンが行われると、アンマウント・リクエストが呼び出され、プロビジョニングが取り消されるまで追跡します。

応答 :

```
{
    "Mountpoint": "/path/to/directory/on/host",
    "Err": null
}
```

ボリュームが利用可能になったり、あるいはエラーが発生したりする場合には、ホスト・ファイルシステム上のパスを返します。

/VolumeDriver.Path

リクエスト :

```
{
    "Name": "volume_name"
}
```

Docker はホスト上のボリュームのパスを覚えておく必要があります。

応答 :

```
{
    "Mountpoint": "/path/to/directory/on/host",
    "Err": null
}
```

ボリュームが利用可能になったり、あるいはエラーが発生したりする場合には、ホスト・ファイルシステム上のパスを返します。

リクエスト :

```
{
    "Name": "volume_name"
}
```

Docker ホストに指定した名前のボリュームを使わないことを指示します。これはコンテナが停止すると呼び出されます。その時点でプラグインはデプロビジョンが安全に行われているとみなします。

レスポンス

```
{
    "Err": null
}
```

エラーが発生したら、エラー文字列を返します。

10 章

その他のドキュメント

あなたの質問がここになければ、 docs@docker.com まで遠慮なくお送りください。あるいは、 リポジトリ[*1] をフォークし、ドキュメントのソースを自分自身で編集し、それらで貢献することもできます。

Docker は 100%自由に使えます。オープンソースであり、何も支払わなくても利用できます。

Apache License Version 2.0 を使っています。こちらをご覧ください：
https://github.com/docker/docker/blob/master/LICENSE

現時点の Docker は Linux 上でしか動きません。しかし VirtualBox を使えば、仮想マシン上で Docker を動かせるため、どちらの環境でも便利に扱えるでしょう。具体的な方法は Mac OS X と Microsoft Windows のインストールガイドをご覧ください。どちらの場合でも、OS 上に仮想マシン内で Docker Machine を実行するための、小さな Linux ディストリビューションをインストールします。

Docker Machine を通して仮想マシン上の Docker デーモンをリモート操作する場合は、ドキュメントのサンプルで、docker コマンドの前にある sudo を入力しないでください 。

相互補完します。仮想マシンはハードウェア・リソースの塊を割り当てるのに一番便利です。コンテナの操作はプロセス・レベルであり、ソフトウェアのデリバリをまとめることができるため、軽量かつパーフェクトです。

Docker 技術は LXC の置き換えではありません。「LXC」は Linux カーネルの機能を参照しており（特に名前空間とコントロール・グループ)、あるプロセスから別のプロセスに対するサンドボックスを可能とし、リソースの割り当てを制御するものです。これらはカーネル機能のローレベルを基礎としています。一方、Docker は複数の強力な機能を持つハイレベルなツールです。

*1 https://github.com/docker/docker

- **マシンをまたがるポータブルなデプロイ**。Docker はアプリケーションを構築するためのフォーマットを定義し、その全ての依存関係を 1 つのオブジェクトにすることで、Docker が利用可能なあらゆるマシン上に移動できるようにします。そして、アプリケーションが同じような環境で動作できるようにするのを保証します。LXC によって実装されるプロセスはサンドボックス（砂場）であり、ポータブルなデプロイの準備としては、重要な環境です。しかしながら、ポータブルなデプロイには十分とは言えません。もしあなたが私にカスタム LXC 設定を施したアプリケーションを送ってきたとしても、おそらく私のマシン上では動作しないでしょう。なぜなら、マシン固有の情報が紐付いているからです。例えばネットワーク、ストレージ、ログ保存、ディストリビューション等です。Docker は、これらマシン固有の情報を抽象化しますので、同じ Docker コンテナであれば間違いなく実行できます。マシンが異なり、環境が変わっていたとしても、コンテナに対して変更を加える必要がありません。

- **アプリケーション中心型**。Docker が最適化されているのはマシンに対してというよりも、アプリケーションのデプロイに対してです。これは API やユーザ・インターフェース、設計哲学やドキュメントにも反映されています。対照的に `lxc` の場合は、コンテナを軽量なマシンとして扱うための補助スクリプトに注力しています。ここで言うマシンというのは、基本的なサーバのことであり、より速く起動し、メモリを必要としない環境です。私たちは LXC よりもコンテナのほうが、より多くの利点があると考えています。

- **自動構築（Automatic Build）**。Docker には、 開発者向けにソース・コードからコンテナを自動的に構築する機能があります。これは構築ツールやパッケージングにあたるアプリケーションの依存性を完全に管理します。マシンの設定に関係無く、`make`、`maven`、`chef`、`puppet`、`salt`、`Debian` パッケージ、RPM、ソースの tar ボール等を自由に扱えます。

- **バージョン管理**。Docker には Git のようにコンテナのバージョン推移を追跡する機能があり、バージョン間の差分を調べ、新しいバージョンをコミットしたり、ロールバックしたり等ができます。また履歴を辿ることで、誰によって何が組み込まれたかを把握できます。そのため、開発元からプロダクションのサーバに至るまでの流れを完全に追跡できます。また Docker には `git pull` のようにアップロード回数とダウンロード回数を記録する機能があるため、コンテナの新しいバージョンを送信するとは、単に差分を送信するだけです。

- **再利用可能なコンポーネント**。コンテナは特別なコンポーネントを「ベース・イメージ」として利用できます。これは手動もしくは自動構築の一部で使えます。例えば、望ましい Python 環境を用意しておけば、10 以上もの異なるアプリケーションの基盤になります。あるいは、望ましい PostgreSQL をセットアップしておけば、自分の将来のプロジェクトで再利用可能になるでしょう。このような使い方ができます。

- **共有**。Docker は Docker Hub というパブリック・レジストリにアクセスします。そこでは数千人もの人たちが便利なイメージをアップロードしています。Redis、CouchDB、PostgreSQL といったイメージから、IRC バウンサーや Rails アプリケーション・サーバや、Hadoop 向けや、様々なディストリビューション向けのベース・イメージがあります。また、公式の「標準ライブラリ（standard library）」にはレジストリという名前の、Docker チームによって管理されている便利なコンテナがあります。レジストリ自身はオープンソースでアリ、誰もが自分自身でレジストリに対して、プライベートなコンテナの保管や転送が可能になります。例えば内部のサーバへデプロイすることも可能です。

- **ツールのエコシステム**。Docker はコンテナの作成と開発のために、自動化・カスタマイズ化の API を定義しています。Docker を互換性のある非常に多くのツールと連携できます。PaaS 風のデプロイ（Dokku、Deis、Flynn）、複数ノードのオーケストレーション（Maestro、Salt、Mesos、OpenStack Nova）、ダッシュボード管理（docker-ui、Openstack Horizon、Shipyard）、設定管理（Chef、Puppet）、継続的インテグレーション（Jenkins、Strider、travis）等です。コンテナを基盤としたツール標準として、Docker は自身を迅速に起動できます。

StackOverflow の回答に、違いについての素晴らしい説明[*1] があります。

コンテナのアプリケーションがディスクに書き込んだあらゆるデータは、コンテナを削除しない限りデータも削除されることはありません。コンテナを停止したとしても、コンテナのファイルシステムは一貫性を保ちます。

今日に世界中で大きなサーバ・ファームのいくつかは、コンテナを基盤としています。Google や Twitter のように大きなウェブのデプロイ環境や、Heroku や dotCloud のように全てをコンテナ技術上で実行します。これら数百から数千、もしくは数百万ものコンテナを並列で実行します。

現時点で推奨する方法は、Docker ネットワーク機能を通してコンテナに接続する方法です。詳細については「Docker ネットワークの働き」のセクションをご覧ください。
またサービスのポータビリティをフレキシブルにするには、アンバサダ・リンク・パターンも便利です。

http://supervisord.org/ のようなスーパーバイザや、runit、s6、daemontools によって実現できます。Docker はプロセス管理用デーモンを起動し、その後、追加プロセスをフォークして実行します。プロセス管理デーモンが動く限り、コンテナも同様に動き続けます。具体的な例については、supervisord の使い方 をご覧ください。

Linux:

- Ubuntu 12.04, 14.04 等
- Fedora 19/20+
- RHEL 6.5+
- CentOS 6+
- Gentoo
- ArchLinux
- openSUSE 12.3+
- CRUX 3.0+
- 等

Cloud:

- Amazon EC2
- Google Compute Engine

*1 http://stackoverflow.com/questions/16047306/how-is-docker-io-different-from-a-normal-virtual-machine

- Microsoft Azure
- Rackspace
- 等

プロジェクトのセキュリティ・ポリシーについてはサイト[*1] から確認できます。セキュリティ問題については、こちらのメールボックス[*2] までお知らせください。

DCO (Developer's Certificate of Origin) については、こちらのブログ投稿[*3] をご覧ください。

このディスカッションの詳細については docker-dev メーリングリストの議論[*4] をご覧ください。

全てのプログラムは擬似的に第三者のライブラリに依存しています。よくあるのは、動的なリンクや、ある種のパッケージ依存性です。そのため、複数のプログラムが同じライブラリを必要とするなら、インストールは一度で済みます。

しかしながら、いくつかのプログラムは、特定バージョンのライブラリに依存するため、自分自身でサード・パーティー製のライブラリを同梱しています。例えば、Node.js は OpenSSL を同梱していますし、MongoDB は V8 と Boost（他にも）を同梱しています。

Docker イメージの作成にあたり、ライブラリの同梱は使い易いものです。しかし、システム・ライブラリに含まれるデフォルトのものを使わず、自分自身でプログラムを構築すべきでしょうか？

システム・ライブラリに関する重要なポイントは、ディスクやメモリ使用量の節約のためではありません。セキュリティのためなのです。全ての主要なディストリビューションは深刻なセキュリティを抱えています。そのため、専用のセキュリティ・チームを持ち、脆弱性が発見されれば対処を行い、一般に情報を開示します（これら手順の具体例は Debian Security Information[*5] をご覧ください)。しかし、上流の開発者によっては、常に同じ手順が踏まれるわけではありません。

Docker イメージの構築時、ソースからプログラムを構築する前に、同梱したいライブラリがあるのであれば、上流の開発者がセキュリティの脆弱性に関する便利な情報を提供しているかどうか、彼らが適時ライブラリを更新するかどうか確認すべきです。彼らが対処しないならば、あなたが自分自身（そして、あなたのイメージの利用者）でセキュリティ脆弱性を対処することになります。

他人によって作成されたパッケージを使う場合も同様です。パッケージに関するセキュリティのベスト・プラクティスと同様に、チャンネルが情報を提供しているか確認すべきでしょう。「全てが中に入っている」(all-in-one) .deb や .rpm のダウンロードとインストールをする場合、OpenSSL ライブラリの脆弱性である Heartbleed バグを抱えているものをコピーされていないかどうか、それを確認する方法はありません。

---

*1 https://www.docker.com/security/

*2 security@docker.com

*3 http://blog.docker.com/2014/01/docker-code-contributions-require-developer-certificate-of-origin/

*4 https://groups.google.com/forum/#!topic/docker-dev/L2RBSPDu1L0

*5 https://www.debian.org/security/

Docker イメージを Debian と Ubuntu 上で構築する時、次のようなエラーがでることがあります。

```
unable to initialize frontend: Dialog
```

イメージの構築時、これらのエラーが出ても処理を中断しませんが、インストール時のプロセスでダイアログ・ボックスを表示しようとしても、実行できなかったという情報を表示しています。通常、これらのエラーは安全であり、無視して構いません。

Dockerfile の中で環境変数 DEBIAN_FRONTEND を変更して使い、これらエラーの回避のために使っている方がいます。

```
ENV DEBIAN_FRONTEND=noninteractive
```

これはインストール時にダイアログ・ボックスを開こうとして、エラーがあっても停止しないようにします。

これは良い考えかもしれませんが、一方で影響があるかもしれません。DEBIAN_FRONTEND 環境変数はイメージからコンテナを構築するにあたり、全てのイメージに対し変更設定が継承されます。対象のイメージを使おうとする人たちが、ソフトウェアをインタラクティブに設定する時に、インストーラは何らダイアログ・ボックスを表示しないため、問題が起こりうる場合があります。

このような状況のため、DEBIAN_FRONTEND を noninteractive に指定するのは「お飾り」の変更であるため、私たちはこのような変更を推奨しません。

本等にこの値を変更する必要がある場合は、あとでデフォルト値に差し戻してください。

このメッセージが表示される主な理由は、サービスは既にローカルホスト上に結びついているからです。その結果、コンテナの外から届いたリクエストは破棄されます。この問題を解決するには、ローカルホスト上のサービスの設定を変更し、サービスが全ての IP アドレスからのリクエストを受け付けるようにします。この設定の仕方が分からなければ、各 OS のドキュメントをご覧ください。

docker-machine 利用時に Cannot connect to ....というエラーが出ます

エラー「Cannot connect to the Docker daemon. Is the docker daemon running on this host」が表示されるのは、Docker クライアントが仮想マシンに接続できない時です。つまり、docker-machine 配下で動く仮想マシンが動作していないか、クライアントが操作時点でマシンを適切に参照できない場合を表します。

docker-machine ls コマンドを使って docker マシンが動作しているかどうかを各西、必要があれば docker-machine start コマンドで起動します。

```
$ docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL   SWARM                   DOCKER    ERRORS
default          -        virtualbox   Stopped                                  Unknown

$ docker-machine start default
```

Docker クライアントはマシンと通信する必要があります。これには docker-machine env コマンドを使います。実行例：

```
$ eval "$(docker-machine env default)"
$ docker ps
```

以下からも答えを探せます。

- Docker user mailinglist https://groups.google.com/d/forum/docker-user
- Docker developer mailinglist https://groups.google.com/d/forum/docker-dev
- IRC, docker on freenode irc://chat.freenode.net#docker
- GitHub https://github.com/docker/docker
- Ask questions on Stackoverflow http://stackoverflow.com/search?q=docker
- Join the conversation on Twitter http://twitter.com/docker

他にもお探しですか？ ユーザガイドをご覧ください。

以下は廃止された機能の一覧です。

docker login の -e および --email フラグ

廃止リリース：v1.11

削除目標リリース：v1.13

docker login コマンドから、ユーザ名が指定されなかった場合に、対象レジストリのアカウントに自動で関連づける機能が削除されます。この変更に伴い email フラグは必要ではなくなり、将来的に廃止します。

--security-opt フラグでキーと値の分割にコロン・セパレータ（ : ）を使えなくなります。これは --storage-opt のような他のフラグと同様、イコール記号（ = ）で指定することになります。

イベント API における不明確なフィールド

廃止リリース：v1.10

イベント API をより優れた構造にするため、ID、Status、From フィールドを廃止しました。新しいフォーマットに関してはイベント API のドキュメントをご覧ください。

docker tag の -f フラグ

廃止リリース：v1.10

削除目標リリース：v1.12

様々な docker コマンド間でタグの付け方を統一するため、docker tag コマンドの -f フラグを廃止しました。イメージのタグを別のものに変える時、 -f オプションの指定は不要です。また、対象のタグが既に利用中であれば、docker コマンドに -f フラグが無くてもエラーになりません。

廃止リリース：v1.10

削除目標リリース：v1.12

POST /containers/{name}/start における HostConfig の指定は、コンテナ作成時の定義（POST /containers/create ）に置き換えられました。

docker ps の「before」「since」オプション

廃止リリース： v1.10.0

削除目標リリース：v1.12

docker ps --before と docker ps --since オプションを廃止しました。代わりに docker ps --filter=before=... と docker ps --filter=since=...をお使いください。

コマンドラインの短縮オプション

廃止リリース：v1.9

削除目標リリース：v1.11

以下の短縮オプションを廃止するため、長いオプションを使うべきです。

```
docker run -c (--cpu-shares)
docker build -c (--cpu-shares)
docker create -c (--cpu-shares)
```

ログ用のドライバを指定するタグ

廃止リリース：v1.9

削除目標リリース：v1.11

異なったログ保存用（ロギング）ドライバを横断して使えるよう、標準化するためにログのタグ機能が生まれました。ドライバを指定するタグのオプションは標準的な tag オプションが望ましくなるため、syslog-tag、gelf-tag、fluentd-tag は廃止されました。

```
docker --log-driver=syslog --log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}"
```

廃止リリース：v1.8

削除目標リリース：v1.10

外部実装の内部（built-in）LXC 実行ドライバを廃止しました。lxc-conf フラグと API も削除予定です。

古いコマンドライン・オプション

廃止リリース：v1.8.0

削除目標リリース：v1.10

-d フラグと --daemon は daemon サブコマンドに移行するため、廃止されます。

```
docker daemon -H ...
```

コマンドライン・オプションのうち、以下のシングル・ダッシュ（-opt）派生を廃止し、新しいダブル・ダッシュ（--opt）に変わります。

```
docker attach -nostdin
docker attach -sig-proxy
docker build -no-cache
docker build -rm
docker commit -author
docker commit -run
docker events -since
docker history -notrunc
docker images -notrunc
docker inspect -format
docker ps -beforeId
docker ps -notrunc
docker ps -sinceId
docker rm -link
docker run -cidfile
docker run -cpuset
docker run -dns
docker run -entrypoint
docker run -expose
docker run -link
docker run -lxc-conf
docker run -n
docker run -privileged
docker run -volumes-from
docker search -notrunc
docker search -stars
```

```
docker search -t
docker search -trusted
docker tag -force
```

以下のダブル・ダッシュのオプションは、置き換えずに廃止です。

```
docker run --networking
docker ps --since-id
docker ps --before-id
docker search --trusted
```

バージョン 1.9 にフラグ（`--disable-legacy-registry=false`）を追加しました。これは docker デーモンが v1 レジストリと `pull`、`push`、`login` させないようにします。デフォルトは廃止された v1 プロトコルと通信しないよう無効化しています。

Docker Content Trust ENV パスフレーズの変数名を変更

廃止リリース：v1.9

削除目標リリース：v1.10

バージョン 1.9 における Docker Content Trust のオフライン鍵（Offline key）はルート鍵（Root key）に、タギング鍵（Tagging key）はリポジトリ鍵（Repository key）に名称変更されました。この名称変更により、関係する環境変数も変わります。

- `DOCKER_CONTENT_TRUST_OFFLINE_PASSPHRASE` を DOCKER_CONTENT_TRUST_ROOT_PASSPHRASE に変更します
- `DOCKER_CONTENT_TRUST_TAGGING_PASSPHRASE` を DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE に変更します。

各 Engine をリリースする度に、前のバージョンとの下位互換性を持つように努めています。全てのケースにおいて、機能を廃止するのは 2 つ先のバージョンというポリシーを持っており、廃止機能のページでドキュメント化しています。

残念ながら、Docker は非常に動いているプロジェクトであり、新しく導入した機能は変更や互換性を弱めてしまうかもしれません。このページでは各エンジンのバージョンごとに文書化しています。

10.3.1 Engine 1.10

1.10 のリリースにあたり、2 つの破壊的変更（breaking change）があります。

Registry

Registry 2.3 はイメージのマニフェストという改良を取り込んだため、破壊的変更をもたらしました。Engine 1.10 から Registry 2.3 にイメージを送信しても、古いバージョンの Engine で digest 値を計算したものは取得できません。`docker pull` を実行しても、次のようなエラーが表示されます。

```
Error response from daemon: unsupported schema version 2 for tag TAGNAME
```

Docker Content Trust は、この digest 値に強く依存しています。そのため、Engine 1.10 のコマンドラインで Registry 2.3 にイメージを送信したとしても、Docker Content Trust を有効化した古い Engine 用の CLI（バージョン 1.10 より小さい）では、イメージを取得（pull）できません。

Registry の古いバージョン（2.3 より小さい）を使っている場合は、どのエンジンの CLI を使って送受信（push、pull）しても問題ありません。ただし、Content Trust を使っていない場合に限ります。

Docker Content Trust

現在の Engine 1.10 よりも古いバージョンでは、key delegation（鍵の権限委譲）のためリポジトリからイメージを取得（pull）できません。key delegation 機能は手動で有効化する必要があります。

Docker Engine バージョン 1.10 以降は、ディスク上にイメージ・データを割り当てる方式が完全に変わります。従来は、各イメージとレイヤに対してランダムな UUID を割り当てていました。バージョン 1.10 からは、イメージとレイヤ・データの安全なハッシュ値を元にした ID を使い、中身を指定できる手法を実装しました。

新しい手法は、利用者を更に安全にします。組み込まれた方式は ID の衝突を防止し、pull・push・load・save を実行した後のデータを保証します。また、レイヤの共有を改善しました。たとえ同時に構築していなくても、多くのイメージ間でレイヤを自由に共有できるようになります。

イメージに対して内容に関する情報を割り当てることで、既にダウンロード済みのイメージがあるかどうかの検出も容易になります。これはイメージとレイヤが分離しているためであり、オリジナルの構築チェーンに含まれる各イメージを、それぞれ取得（pull）する必要はありません。また、構築命令のためにレイヤを作成する必用がなくなったため、ファイルシステムを変更しません。

連想機能（content addressability；コンテント・アドレッサビィティ）とは、新しい配布機能の基礎です。イメージの取得（pull）と送信（push）の手法は、ダウンロード／アップロード・マネージャの概念を扱うために調整されました。これにより、イメージの送受信がより安定し、並列リクエスト時の問題を軽減します。ダウンロード・マネージャはダウンロードの失敗時にリトライできるようになり、ダウンロードにおける適切な優先度付けも行えるようにもなりました。

更に、新しいマニフェスト・フォーマットも導入しました。これは連想機能をベースにしています。これは連想イメージ（content addressable image）の設定やレイヤのチェックサムを直接参照できます。この新しいマニフェスト・フォーマットにより、複数のアーキテクチャやプラットフォームのためにマニフェスト・リストが利用できるでしょう。新しいマニフェスト・フォーマットへの移行は、完全に透過的です。

10.4.1 アップグレードの準備

現在のイメージを新しいモデルで利用できるようにするには、連想ストレージ（content addressable storage）への移行が必要です。すなわち、現在のデータの安全なチェックサムを計算することを意味します。

Docker Engine 1.10 の初回起動時は、現時点における全てのイメージ、タグ、コンテナが自動的に新しい基盤上に移行します。そのため、コンテナを読み込む前に、デーモンは現時点のデータに対するチェックサムを全て計算する必要があります。計算が終わったら、全てのイメージとタグは新しい安全な ID に更新されます。

これは非常に単純な操作ですが、各ファイルに対する SHA256 チェックサムを計算するため、多くのイメージ・データがあれば計算に時間を消費します。データの移行プロセスにかかる時間は、およそ平均して 1 秒あたり 100MB と想定されます。この処理の間、Docker デーモンはリクエストに応答できません。

一度だけの処理が許容できるのであれば、Docker Engine のアップグレードと、デーモンの再起動により、イメージの移行が透過的に行われます。しかしながら、デーモンの停止時間を最小化したい場合は、古いデーモンを動かしたまま移行ツールを使えます。

このツールは現在のイメージを全て探し出し、それらのチェックサムを計算します。デーモンのアップグレードと再起動を行った後、移行するイメージに対するチェックサム・データが既に存在していれば、デーモンは計算処理を行いません。移行とアップグレード期間中に新しいイメージが追加されている場合は、1.10 へのアップグレードのプロセス中で処理されます。

移行ツールはこちらからダウンロードできます 。https://github.com/docker/v1.10-migrator/releases

移行ツールは Docker イメージとしても実行することができます。移行用ツールのイメージを実行している間、このコンテナに対して Docker のデータを直接公開する必要があります。

```
$ docker run --rm -v /var/lib/docker:/var/lib/docker docker/v1.10-migrator
```

devicemapper ストレージ・ドライバを使っている場合は、 --privileged フラグを指定することで、ツールがストレージ・デバイスにアクセス可能となります。