

Docker の監視・ジョブ運用検証報告書

第 1.0.0 版

作成：2015 年 9 月 15 日

更新：2015 年 9 月 15 日

株式会社 NTTデータ

技術革新統括本部

基盤システム事業本部システム方式技術事業部

第三技術統括部第三技術担当

技術情報

変更履歴

版	変更日	変更内容
1	2015/9/15	初版

目次

目次.....	2
1. はじめに	5
1.1. ライセンス	5
1.2. 目的	5
1.3. ドキュメント構成.....	5
1.4. 本ドキュメントの使い方.....	5
1.5. 対象システム構成.....	6
1.6. サンプルスクリプトの動作要件.....	7
1.7. サンプルスクリプトのセットアップ	8
1.8. 注意事項	9
2. 構成管理編.....	10
2.1. コンテナ登録.....	13
2.1.1. 概要	13
2.1.2. 仕様・アーキテクチャ	13
2.1.3. サンプルコードの使い方(CLI)	16
2.1.4. Hinemos での使い方.....	18
2.2. コンテナ削除.....	19
2.2.1. 概要	19
2.2.2. 仕様・アーキテクチャ	19
2.2.3. サンプルコードの使い方(CLI)	20
2.2.4. Hinemos での使い方.....	21
2.3. コンテナ更新.....	22
2.3.1. 概要	22
2.3.2. 仕様・アーキテクチャ	22
2.3.3. サンプルコードの使い方(CLI)	22
2.3.4. Hinemos での使い方.....	24
3. 監視編	25
3.1. Docker ホストの状態監視.....	27
3.1.1. 概要	27
3.1.2. 仕様・アーキテクチャ	27
3.1.3. スクリプトの使い方 (CLI 実行)	30
3.1.4. Hinemos での使い方.....	32

3.2.	Docker ホストのイベント監視	35
3.2.1.	概要	35
3.2.2.	仕様・アーキテクチャ	35
3.2.3.	スクリプトの使い方(CLI)	36
3.2.4.	スクリプトの使い方(Hinemos)	38
3.3.	Docker コンテナのリソース監視(CPU)	39
3.3.1.	概要	39
3.3.2.	仕様・アーキテクチャ	39
3.3.3.	サンプルコードの使い方(CLI)	41
3.3.4.	Hinemos での使い方	44
3.4.	Docker コンテナのリソース監視(メモリ)	47
3.4.1.	概要	47
3.4.2.	仕様・アーキテクチャ	47
3.4.3.	スクリプトの使い方(CLI)	50
3.4.4.	Hinemos での使い方	53
3.5.	Docker コンテナのリソース監視(ディスク)	55
3.5.1.	概要	55
3.5.2.	仕様・アーキテクチャ	55
3.5.3.	スクリプトの使い方(CLI)	56
3.5.4.	スクリプトの使い方(Hinemos)	58
3.6.	Docker コンテナのリソース監視(ネットワーク)	59
3.6.1.	概要	59
3.6.2.	仕様・アーキテクチャ	59
3.6.3.	スクリプトの使い方(CLI)	60
3.6.4.	Hinemos での使い方	63
3.7.	Docker コンテナ内のプロセス監視	64
3.7.1.	概要	64
3.7.2.	仕様・アーキテクチャ	64
3.7.3.	スクリプトの使い方(CLI)	65
3.7.4.	スクリプトの使い方(Hinemos)	67
3.8.	Docker コンテナの状態監視	68
3.8.1.	概要	68
3.8.2.	仕様・アーキテクチャ	68
3.8.3.	スクリプトの使い方(CLI)	69
3.8.4.	Hinemos での使い方	70
4.	ジョブ・操作編	73

4. 1.	Docker コンテナ内のコマンド実行	74
4. 1. 1.	概要	74
4. 1. 2.	仕様・アーキテクチャ	75
4. 1. 3.	スクリプトの使い方 (CLI)	76
4. 1. 4.	スクリプトの使い方 (Hinemos)	77
4. 2.	ファイルダウンロード	79
4. 2. 1.	概要	79
4. 2. 2.	仕様・アーキテクチャ	79
4. 2. 3.	スクリプトの使い方 (CLI)	79
4. 2. 4.	スクリプトの使い方 (Hinemos)	81
4. 3.	Docker コンテナの操作（起動）	83
4. 3. 1.	概要	83
4. 3. 2.	仕様・アーキテクチャ	83
4. 3. 3.	スクリプトの使い方 (CLI)	83
4. 3. 4.	スクリプトの使い方 (Hinemos)	85
4. 4.	Docker コンテナの操作（停止）	86
4. 4. 1.	概要	86
4. 4. 2.	仕様・アーキテクチャ	86
4. 4. 3.	スクリプトの使い方 (CLI)	87
4. 4. 4.	スクリプトの使い方 (Hinemos)	88
4. 5.	Docker コンテナの操作（再起動）	89
4. 5. 1.	概要	90
4. 5. 2.	仕様・アーキテクチャ	90
4. 5. 3.	スクリプトの使い方 (CLI)	90
4. 5. 4.	スクリプトの使い方 (Hinemos)	92
4. 6.	Docker コンテナの操作（削除）	93
4. 6. 1.	概要	93
4. 6. 2.	仕様・アーキテクチャ	93
4. 6. 3.	スクリプトの使い方 (CLI)	93
4. 6. 4.	スクリプトの使い方 (Hinemos)	95
5.	TIPS 集	97
5. 1.	nsenter を使ったコンテナ内のプロセス起動時の注意	97
5. 2.	Docker コンテナの Status と State	97
6.	おわりに	99

1. はじめに

1.1. ライセンス

Hinemos は GNU General Public License となります。各種ドキュメントは GNU General Public License ではありません。各種ドキュメントの無断複製・無断転載・無断再配布を禁止します。

1.2. 目的

Hinemos を使った Docker の監視・ジョブ運用検証を行い、そのノウハウを報告書として整理しました。

本マニュアルでの設定は一例であり、実際に使用される際はご利用の環境のセキュリティポリシーに沿って設定を変更して使用されることをお勧めします。本ソフトウェアの使用により生じたいかなる損害に対しても、弊社は一切の責任を負いません。

1.3. ドキュメント構成

本ドキュメントは、次の2章から4章にてこの Docker 環境の管理を行う背景とその仕組み、そしてサンプル実装しましたスクリプトの使い方を解説します。

2章 構成管理編

コンテナの状態遷移に合わせて Hinemos のリポジトリ構成を更新する方法を紹介します。

3章 監視編

Docker コンテナ、Docker ホストについての監視の必要性とその実現方法を紹介します。

4章 ジョブ・操作編

Docker コンテナを外部から操作する方法を紹介します。

また、上記2章から4章の内容を検証するにあたり、確認したノウハウを5章の TIPS 集に整理しました。

1.4. 本ドキュメントの使い方

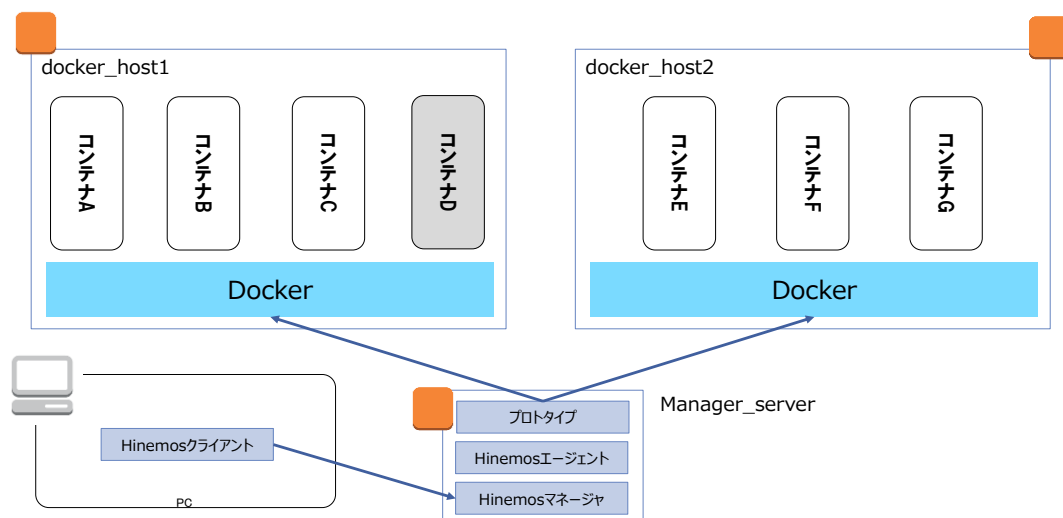
本ドキュメントでは、次節のシステム構成を想定して、Hinemos を使った Docker 環境の運用管理に必要な機能を洗い出し、各機能をサンプルスクリプトを用いて解説を行ったものです。Hinemos を使った Docker 環境の運用管理に必要な機能とは、Docker 環境を運用管

理のリポジトリ構成に自動反映（2章）させ、その環境の Docker 特有の監視（3章）や、Docker コンテナ操作をジョブフローフローに組み込むための機能になります。

サンプルスクリプトは本ドキュメントと同様に自由にダウンロードし、ご利用頂けるため、本ドキュメントの記載内容を、そのままご確認頂くことも可能です。本ドキュメントの記載内容をご確認頂く場合は、次節以降の対象システム構成や事前セットアップを実施頂いたのち、まず2章の構成管理編から開始してください。3章、4章は2章を実施頂いたのちに、どちらでも自由に進んでいただけるような構成になっています。

1.5. 対象システム構成

Hinemos マネージャの動作する Linux サーバ、Docker が動作する Linux サーバ(Docker ホスト)、Hinemos クライアントを表示する PC 端末が必要です。説明の都合上、Docker の動作する Linux サーバは2台で説明を進めますが、1台だけでも動作を確認できます。



○Linux サーバの環境

ホスト名(*1)	OS	用途
docker_host1	RHEL7.1 or CentOS7.1	Docker が動作するサーバ
docker_host2	RHEL7.1 or CentOS7.1	Docker が動作するサーバ
manager_server	RHEL7.1 or CentOS7.1	Hinemos マネージャ、Hinemos エージェント、及びサンプルスクリプトをインストール

(*1) 各サーバは名前解決によりホスト名でアクセス可能な環境とします。

○Hinemos をインストールする環境について

Docker の運用管理を行うにあたり、次の理由により Docker Remote API を使用することで、リモートから処理を実行する方式を検討しました。

- Docker ホストへの Hinemos エージェントインストール

SaaS 型で Docker を利用する形態が増えてきており、Docker ホストの Linux 環境に必ずしも追加のアプリケーションをインストールすることが可能かは分かりません。そのため、Docker ホストへの Hinemos エージェントインストールを行わない方式を検討しました。

- Docker コンテナへの Hinemos エージェントインストール

Docker コンテナは軽量であることがメリットになるため、Hinemos エージェントを導入することは、リソース使用効率の観点から好ましくありません。そのため、Docker コンテナへの Hinemos エージェントインストールを行わない方式を検討しました。

そのため、Hinemos をインストールする環境は Docker ホストとは別の Linux サーバ（通常は運用管理サーバと呼ばれるサーバ）に全てインストールします。

1.6. サンプルスクリプトの動作要件

本ドキュメントに付属するサンプルスクリプトは以下の環境で動作を確認しています。

○Docker ホスト

- Docker ホスト OS

Red Hat Enterprise Linux 7.1 / CentOS 7.1

- Docker バージョン

- docker-selinux-1.6.2-14.el7.x86_64
- docker-1.6.2-14.el7.x86_64

○Hinemos

- Hinemos バージョン

- Hinemos ver. 5.0.0

また Linux 環境は全て、SELinux は無効化、firewalld 無効化の設定を行っています。

1.7. サンプルスクリプトのセットアップ

本ドキュメントに付属するサンプルスクリプトのセットアップ方法を記載します。

○Hinemos のインストール

前節の「1.4. 対象システム構成」の内容の通り、Hinemos マネージャ、Hinemos エージェントを `manager_server` にインストールしてください。

Hinemos のインストールについては、「Hinemos ver5.0 インストールマニュアル 第 1 版」を参照してください。

○Hinemos のインストール後の設定

`manager_server` をノードとして登録し、Hinemos エージェントと疎通がとれているかをリポジトリ[エージェント]ビューから確認してください。

Docker ホストである `docker_host1` と `docker_host2` は、2 章で紹介する機能にてノード登録するため、ここでは手動でノード登録は行わないでください。

○サンプルスクリプトのインストール

`manager_server` にサンプルスクリプトをインストールします。インストール先は、`/opt/hinemos_agent/docker` です。

パッケージ名：`hinemos_docker_sample_v1.0.0.tar.gz`

インストール方法は、パッケージを展開したディレクトリ内にある `README.txt` を参照してください。

○Docker ホストの設定

サンプルスクリプトを動作させるためには、TCP 経由で Docker のリモート API を呼び出せるように、Docker デーモンを設定する必要があります。また、一部のサンプルスクリプトでは Docker ホストの SNMP に接続するものもあります。

これらの設定方法も、パッケージを展開したディレクトリ内にある `README.txt` に記載がありますので参照してください。

○Docker コンテナの用意

動作確認のため、Docker ホスト上に 1 つ以上のコンテナを用意する必要があります。特に Docker コンテナに制約はありませんが、例えば、2 章の Docker コンテナの自動検出の動作確認のための Docker コンテナの起動方法の例は、2 章の中にも解説があります。そのため、各章節を読み進める中で、適宜、Docker コンテナを起動したり削除したりしてくだ

さい。

1.8. 注意事項

本ドキュメントでは、Hinemos の基本機能に関する解説は行っておりません。Hinemos の使い方については、マニュアルや、Hinemos ポータルサイトの技術情報にあるスタートアップ記事を参考にしてください。

○Hinemos 技術情報

URL :

<http://www.hinemos.info/technology>

参考記事 :

[Hinemos ver.5.0 を使ってみる \(入門編\)](#)

2. 構成管理編

本章では、Docker コンテナの生成や削除に追従して、Hinemos のリポジトリ情報を自動的に更新する方式について記載します。

○背景と目的

Docker ではコンテナの生成・削除、起動・停止を素早く行えることが重要なメリットの 1 つです。そのため、Docker を活用するシステムでは、コンテナ数やコンテナの状態が激しく変動することが想定されます。

そのコンテナの状態を監視したり、コンテナ内のコマンドを定期的に行うなどの運用を行う場合に、生成・削除や起動・停止が頻繁に発生するコンテナを管理対象のリポジトリ情報に手で反映することはとても非効率です。

そこで、Docker 上のコンテナの状態変化に追従して、管理対象のリポジトリ情報を自動的に更新する機能が求められます。

○実現方法

Hinemos では、Docker コンテナや Docker ホストを Hinemos のリポジトリ情報として次のように扱います。

[Hinemos のリポジトリ情報]

• Docker コンテナ

Hinemos の「ノード」として表現します。

• Docker ホスト

Hinemos の「ノード」として表現します。

• Docker コンテナと Docker ホストのマッピング

Hinemos の「スコープ」として表現します。

Docker ホストに該当するスコープを用意し、それに割り当てられているノードを Docker コンテナとします。

これを Docker ホストから自動的に情報を取得し、Hinemos のリポジトリに反映する機能として次の 3 つをサンプルスクリプトで用意しました。

[機能一覧]

• コンテナ登録(RepositoryDockerNodeAdd. py)

指定した Docker ホスト上で動作する Docker コンテナの中で、追加されたコンテナをノードとして Hinemos のリポジトリに登録します。

• コンテナ削除(RepositoryDockerNodeDelete. py)

指定した Docker ホスト上で動作する Docker コンテナの中で、削除されたコンテナのノードを Hinemos のリポジトリから削除します。

• コンテナ更新(RepositoryDockerNodeUpdate. py)

指定した Docker ホスト上で動作する Docker コンテナの状態を確認し、コンテナの状態に合うように対象ノードの「管理対象フラグ」の ON/OFF を変更します。

その他にも、Docker v1.6 より導入された Label 機能により、コンテナに Label が指定されている場合は、その内容から任意のスコープにそのノードを Hinemos の割当てることができます。(Label は設定後、対象のコンテナに対して変更ができません。変更の際はコンテナの再作成が必要です)。これをうまく活用することで、Hinemos のスコープを使ってコンテナの分類をすることが出来、スコープ単位で監視設定やジョブの実行をすることで、分類されたコンテナ単位での運用が簡易に実現できます。

[使い方概要]

本機能は、コンテナ登録、削除、更新のスクリプトをセットで使用します。一番簡単な使い方は、本スクリプトをインストールした hinemos_manager サーバ上で cron で実行することです。

• 一括実行スクリプトの作成

```
(root) # cat /opt/hinemos_agent/docker/bin/RepositoryDockerNode. sh
#!/bin/bash

# RepositoryDockerNodeAdd. py
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeAdd. py -i 10.0.200.123 -f docker_host1
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeAdd. py -i 10.0.200.204 -f docker_host2

# RepositoryDockerNodeDelete. py
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeDelete. py -i 10.0.200.123 -f docker_host1
```

```
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeDelete.py -i 10.0.200.204 -f docker_host2  
s
```

```
# RepositoryDockerNodeUpdate.py
```

```
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeUpdate.py -i 10.0.200.123 -f docker_host1
```

```
/opt/hinemos_agent/docker/bin/RepositoryDockerNodeUpdate.py -i 10.0.200.204 -f docker_host2
```

- ・ cron 設定 (5 分毎の実行)

```
(root) # crontab -l
```

```
*/5 * * * * /opt/hinemos_agent/docker/bin/RepositoryDockerNode.sh
```

- ・ Docker コンテナ側の設定

Docker コンテナに次の Label を設定 (Docker v1.6 以降) をすると、その情報を参照して任意のスコープに割り当てることができます。

key : hinemosAssignScopeId

value : (スコープのファシリティ ID) [, (スコープのファシリティ ID), ...]

設定例)

```
docker run -d -it --name container_B -l hinemosAssignScopeId=web1,web2 httpd:2.4  
httpd -D FOREGROUND
```

この設定では、container_B コンテナに対応するノードが、Hinemos のスコープ web1 とスコープ web2 に割り当てられます。

- ・ 注意事項

〈接続先 Hinemos マネージャの設定〉

RepositoryDockerNodeAdd.py 、 RepositoryDockerNodeDelete.py 、
RepositoryDockerNodeUpdate.py の接続先 Hinemos マネージャへのログイン情報 (ユーザ/
パスワード) のデフォルト値は hinemos/hinemos です。必要に応じてスクリプト内の設定
を変更します。

```
# Hinemos へのログイン情報
```

```
args.hinemos_username = "hinemos"
```

```
args.hinemos_password = "hinemos"
```

個々の機能の詳細は、各々を解説している次節を参照してください。

○本章の構成

構成管理に関する 3 機能を順次説明します。

- ・ コンテナ登録
- ・ コンテナ削除
- ・ コンテナ更新

○注意事項

- ・ 構成管理に関する 3 つのサンプルスクリプトは 3 つセットで利用します。
- ・ Docker ホストを追加する際は、最初に対象 Docker ホストに対して、コンテナ削除 (RepositoryDockerNodeDelete.py) とコンテナ更新の前 (RepositoryDockerNodeUpdate.py) に、コンテナ登録 (RepositoryDockerNodeAdd.py) の実行が必要です。

2.1. コンテナ登録

2.1.1. 概要

指定した Docker ホスト上で動作する Docker コンテナの中で、追加されたコンテナをノードとして Hinemos のリポジトリに登録します。

Docker v1.6 より導入された Label 機能により、コンテナに Label が指定されている場合は、その内容を読み取り任意のそのノードを Hinemos のスコープに割当てることが出来ます。

2.1.2. 仕様・アーキテクチャ

指定した Docker ホスト上のコンテナ情報を取得し、Hinemos のリポジトリにノードとしてコンテナを新規に登録します。初回実行時は指定した Docker ホスト上の全てのコンテナを、2 回目以降実行時は追加されたコンテナのみを、ノードとして登録します。

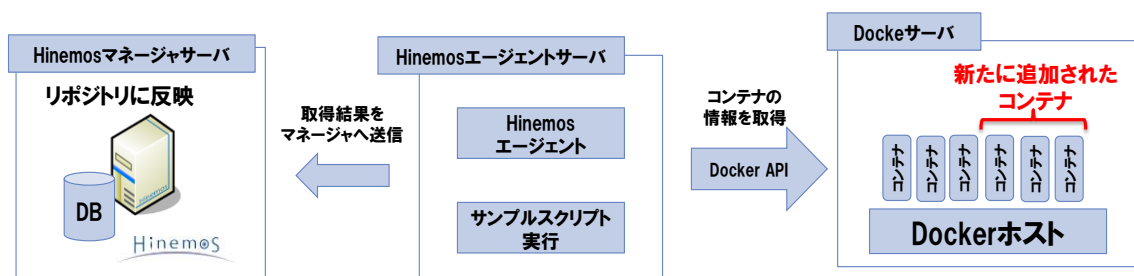


図 コンテナ登録

○ノード作成

・ホストノード

指定した Docker ホストをノードとして登録します。（登録済みの場合はスキップ）

・コンテナノード

Docker ホスト上のコンテナをノードとして登録します。（登録済みの場合はスキップ）

コンテナに `key=hinemosAssignScopeId` の Label が設定されている場合は、その `value=(ファシリティ ID,...)` に従い、指定のファシリティ ID のスコープに当該ノードを割り当てます。

○ノードプロパティ

自動的に以下の値がノードプロパティとして設定されます。

プロパティ	ホストノード	コンテナノード
ファシリティ ID	引数-f で指定した名前	(ホストのファシリティ ID)_コンテナ名
ファシリティ名	引数-f で指定した名前	コンテナ名
管理対象	有効	(管理対象フラグの説明を参照)
プラットフォーム	LINUX	LINUX
IP アドレスのバージョン	4	4
IPv4 のアドレス	引数-i で指定したアドレス	(ホストと同じアドレス)
ホスト名	引数-f で指定した名前	コンテナ ID(先頭 12 文字)
ノード名	引数-f で指定した名前	コンテナ ID(先頭 12 文字)
ノード変数	-	(ノード変数の説明を参照)

○管理対象フラグ

ノードプロパティの中の Docker コンテナの状態により、管理対象フラグを決定します。

Docker コンテナの状態とは、`docker ps -f` で指定する `state` を指します。

コンテナの状態	管理対象フラグ
running	有効
restarting, exited, paused	無効

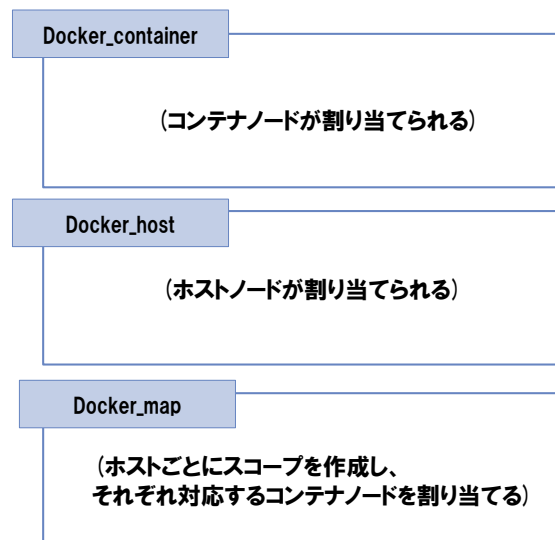
○ノード変数

コンテナノードでは、ノードプロパティの中のノード変数に、各種コンテナ情報が自動的に設定されます。

DOCKER_CONTAINER_ID	コンテナ ID (先頭 12 文字)
DOCKER_COMMAND	コンテナ起動時のコマンド
DOCKER_PORTS	コンテナ起動時に -v オプションでポートマップした場合のみ、ポートマップ設定を表示
DOCKER_IMAGE	コンテナ起動時に指定したイメージ
DOCKER_STATUS	コンテナの状態
DOCKER_HOST	コンテナが動作するホストノードのファシリティ ID

○スコープ作成とノードの割り当て

初回実行時に、以下のスコープを自動的に作成して、それぞれ対応する Docker コンテナノードを割り当てます。次節で説明するコンテナ更新の機能を使用すると、このスコープ構成が更新されます。



スコープ	割り当てスコープ/ノード
docker_container	Docker コンテナノード
docker_host	Docker ホストノード
docker_map	Docker ホスト毎のスコープ（自動作成） 本スコープ配下には、それぞれに対応する Docker コンテナノードが割当

2.1.3. サンプルコードの使い方 (CLI)

○スクリプト名

RepositoryDockerNodeAdd.py

○書式

RepositoryDockerNodeAdd.py [-h] [-i IP_ADDRESS] [-p DOCKER_PORT]
[-m HINEMOS_URL] [-f HOST_FACILITY_ID]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-m HINEMOS_URL	Hinemos マネージャの URL (デフォルト : http://127.0.0.1:8080/HinemosWS)	-
-f HOST_FACILITY_ID	Docker ホストの対応するノードのファシリティ ID	○
-h, --help	スクリプトの使用方法を表示	-

○戻り値

正常 : 0

異常 : 1

○注意事項

接続先 Hinemos マネージャのログイン情報の変更方法は、2 章の[使い方概要]を参照してください。

○実行例

Hinemos のリポジトリにスコープ web1 と web2 が作成済みであり、スコープ web3 は存在しない環境とします。

Docker ホスト上のコンテナは、各々次のようなラベルが設定されているとします。

```
(root) # docker run -d -it --name container_A -l hinemosAssignScopeId=web1  
httpd:2.4 httpd -D FOREGROUND
```

```
(root) # docker run -d -it --name container_B -l hinemosAssignScopeId=web1,web2  
httpd:2.4 httpd -D FOREGROUND
```

```
docker run -d -it --name container_C -l hinemosAssignScopeId=web1,web3 httpd:2.4
httpd -D (root) # FOREGROUND
```

この環境での RepositoryDockeNodeAdd.py の初回実行では、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/RepositoryDockeNodeAdd.py -i
100.74.26.24 -f docker_host1
```

開始しました。

Hinemos マネージャにログインしています。

```
endpoint: http://127.0.0.1:8080/HinemosWS/RepositoryEndpoint?wsdl
```

```
username: hinemos
```

Docker コンテナの情報を取得します。

port not bind

```
Id:16c155fe269b Name:container_C Status:'Up 4 minutes':running
```

port not bind

```
Id:f4bb5ade7641 Name:container_B Status:'Up 4 minutes':running
```

port not bind

```
Id:f0184f3feddc Name:container_A Status:'Up 4 minutes':running
```

スコープを作成します。

docker_container を作成します。

docker_host を作成します。

docker_map を作成します。

docker_host1_scope を作成します。

Docker ホストノードを作成します。

docker_host1 を作成します。

Docker ホストノードを docker_host スコープへ割当てます。

docker_host1 を割当てます。

Docker コンテナノードを作成します。

docker_host1-container_C を作成します。

docker_host1-container_B を作成します。

docker_host1-container_A を作成します。

Docker コンテナノードを docker_container スコープへ割当てます。

docker_host1-container_C を割当てます。

docker_host1-container_B を割当てます。

docker_host1-container_A を割当てます。

Docker コンテナノードを docker_host1_scope スコープへ割当てます。

docker_host1-container_C を割当てます。
 docker_host1-container_B を割当てます。
 docker_host1-container_A を割当てます。
 docker_host1-container_C をラベル設定からスコープへ割当てます。対象スコープ：
 [u'web1', u'web3']
 docker_host1-container_C を割当てます。
 [skip] スコープ web3 が存在していないため、割当てをスキップします。
 docker_host1-container_B をラベル設定からスコープへ割当てます。対象スコープ：
 [u'web1', u'web2']
 docker_host1-container_B を割当てます。
 docker_host1-container_B を割当てます。
 docker_host1-container_A をラベル設定からスコープへ割当てます。対象スコープ：
 [u'web1']
 docker_host1-container_A を割当てます。
 完了しました。

2.1.4. Hinemos での使い方

本機能は、cron で実行してその結果が Hinemos のリポジトリに反映されるため、ここでは Hinemos クライアントからどのように見えるかを紹介します。

次の環境の中の manager_sever 上で本スクリプトを実行します。

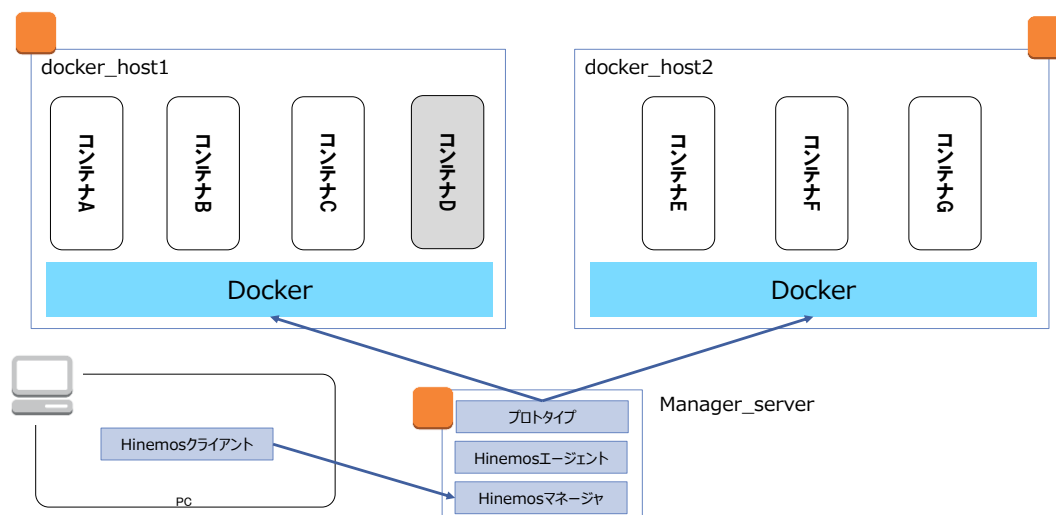


図 環境構成図

すると、リポジトリ[スコープ]ビューからは次のように見えます。

スコープ : docker_map>docker_host1_scope>

マネージャ	ファシリティID	ファシリティ名	説明	オーナーロールID
マネージャ1	docker_host1-container_A	container_A	Auto detect at 2015-08-31 16:	ALL_USERS
マネージャ1	docker_host1-container_B	container_B	Auto detect at 2015-08-31 16:	ALL_USERS
マネージャ1	docker_host1-container_C	container_C	Auto detect at 2015-08-31 16:	ALL_USERS
マネージャ1	docker_host1-container_D	container_D	Auto detect at 2015-08-31 16:	ALL_USERS

図 リポジトリ[スコープ]ビュー

このように Docker ホスト上にコンテナが作成されると、自動的に Hinemos のリポジトリ情報に反映されます。そのため、監視設定やジョブ定義を、コンテナが所属する「スコープ」に設定するだけでユーザは特に作業を行う必要がなく、追加されたコンテナへの関したジョブ実行が自動的に行えるようになります。

2.2. コンテナ削除

2.2.1. 概要

指定した Docker ホスト上で動作する Docker コンテナの中で、削除されたコンテナのノードを Hinemos のリポジトリから削除します。

2.2.2. 仕様・アーキテクチャ

指定した Docker ホスト上のコンテナ情報を取得し、Hinemos のリポジトリから既に削除されたコンテナのノードを削除します。ノードの識別はファシリティ ID (Docker ホストのファシリティ ID_コンテナ名) です。

ノードが削除されると、そのノードが割り当てられていたスコープからも自動的に割り当てが解除されます。

2.2.3. サンプルコードの使い方 (CLI)

○スクリプト名

RepositoryDockerNodeDelete.py

○書式

```
RepositoryDockerNodeDelete.py [-h] [-i IP_ADDRESS] [-p DOCKER_PORT]
                               [-m HINEMOS_URL] [-f HOST_FACILITY_ID]
```

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-m HINEMOS_URL	Hinemos マネージャの URL (デフォルト : http://127.0.0.1:8080/HinemosWS)	-
-f HOST_FACILITY_ID	Docker ホストの対応するノードのファシリティ ID	○
-h, --help	スクリプトの使用方法を表示	-

○戻り値

正常 : 0

異常 : 1

○注意事項

接続先 Hinemos マネージャのログイン情報の変更方法は、2 章の[使い方概要]を参照してください。

○実行例

docker_host1 から container_D を削除すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/RepositoryDockerNodeDelete.py -i
100.74.26.24 -f docker_host1
```

開始しました。

Hinemos マネージャにログインしています。

endpoint: http://127.0.0.1:8080/HinemosWS/RepositoryEndpoint?wsdl

username: hinemos

Docker コンテナの情報を取得します。

Id:0e0d8cdf3a57 Name:container_C

Id:1eae26f41927 Name:container_B

Id:b66e046560f3 Name:container_A

Docker コンテナノードを削除します。

docker_host1-container_C

存在します。[skip]

docker_host1-container_D

削除します。

docker_host1-container_A

存在します。[skip]

docker_host1-container_B

存在します。[skip]

完了しました。

2.2.4. Hinemos での使い方

本機能は、cron で実行してその結果が Hinemos のリポジトリに反映されるため、ここでは Hinemos クライアントからどのように見えるかを紹介します。

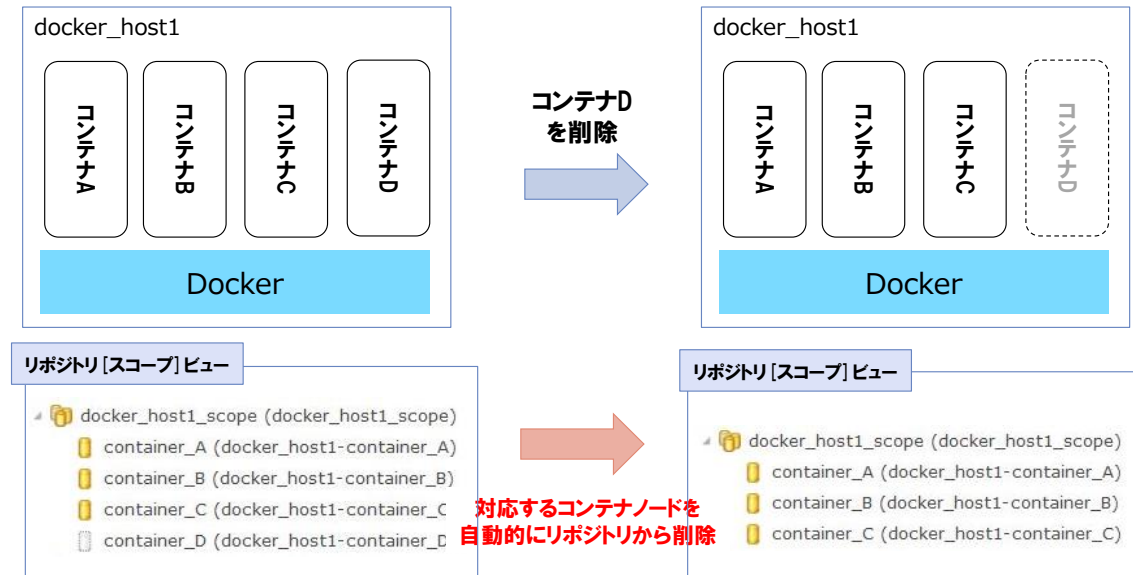


図 コンテナノードの削除

このように Docker ホスト上にコンテナが削除されると、自動的に Hinemos のリポジトリ情報に反映されます。そのため、削除されたコンテナへの関したジョブ実行の停止を自動的に行えるようになります。

2.3. コンテナ更新

2.3.1. 概要

指定した Docker ホスト上で動作する Docker コンテナの状態を確認し、コンテナの状態に合うように対象ノードの「管理対象フラグ」の ON/OFF を変更します。

Docker v1.6 より導入された Label 機能により、コンテナに Label が指定されている場合は、その内容を読み取り任意のそのノードを Hinemos のスコープに割り当てることが出来ます。

2.3.2. 仕様・アーキテクチャ

指定した Docker ホスト上のコンテナ情報を取得し、Hinemos のリポジトリ情報をアップデートします。

○更新内容

次の2つの情報をアップデートします。

- ・Label によるスコープ割当

コンテナに Label が追加された場合は、その情報を参照してスコープ割り当てを行います。割り当ててるスコープが減っていた場合でも、割り当て解除は行いません。

- ・コンテナ状態による管理対象フラグとノード変数 DOCKER_STATUS

コンテナ状態をノードプロパティの管理対象フラグとノード変数 DOCKER_STATUS に反映します。

2.3.3. サンプルコードの使い方(CLI)

○スクリプト名

RepositoryDockerNodeUpdate.py

○書式

```
RepositoryDockerNodeUpdate.py [-h] [-i IP_ADDRESS] [-p DOCKER_PORT]
                                [-m HINEMOS_URL] [-f HOST_FACILITY_ID]
```

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート	-

	(デフォルト : 2375)	
-m HINEMOS_URL	Hinemos マネージャの URL (デフォルト : http://127.0.0.1:8080/HinemosWS)	-
-f HOST_FACILITY_ID	Docker ホストの対応するノードのファシリティ ID	○
-h, --help	スクリプトの使用方法を表示	-

○戻り値

正常 : 0

異常 : 1

○注意事項

接続先 Hinemos マネージャのログイン情報の変更方法は、2 章の[使い方概要]を参照してください。

○実行例

docker_host1 から container_C を削除すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/RepositoryDockNodeUpdate.py -i
100.74.26.24 -f docker_host1
```

開始しました。

Hinemos マネージャにログインしています。

```
endpoint: http://127.0.0.1:8080/HinemosWS/RepositoryEndpoint?wsdl
```

```
username: hinemos
```

Docker コンテナの情報を取得します。

```
Id:0e0d8cdf3a57 Name:container_C Status:'Exited (0) 16 seconds ago':exited
```

```
Id:1eae26f41927 Name:container_B Status:'Up 35 minutes':running
```

```
Id:b66e046560f3 Name:container_A Status:'Up 35 minutes':running
```

Docker コンテナノードを更新します。

```
docker_host1-container_C
```

```
running -> exited
```

```
docker_host1-container_B
```

```
running -> running [skip]
```

```
docker_host1-container_A
```

```
running -> running [skip]
```

```
docker_host1-container_C のラベル設定の確認 hinemosAssignScopeId: [u'web1',
u'web3']
```


[skip] 既にスコープ web1 に割当てられています
 [skip] スコープ web3 が存在していないため、割当てをスキップします
 docker_host1-container_B のラベル設定の確認 hinemosAssignScopeId: [u'web1',
 u'web2']
 [skip] 既にスコープ web1 に割当てられています
 [skip] 既にスコープ web2 に割当てられています
 docker_host1-container_A のラベル設定の確認 hinemosAssignScopeId: [u'web1']
 [skip] 既にスコープ web1 に割当てられています
 完了しました。

2.3.4. Hinemos での使い方

本機能は、cron で実行してその結果が Hinemos のリポジトリに反映されるため、ここでは Hinemos クライアントからどのように見えるかを紹介します。

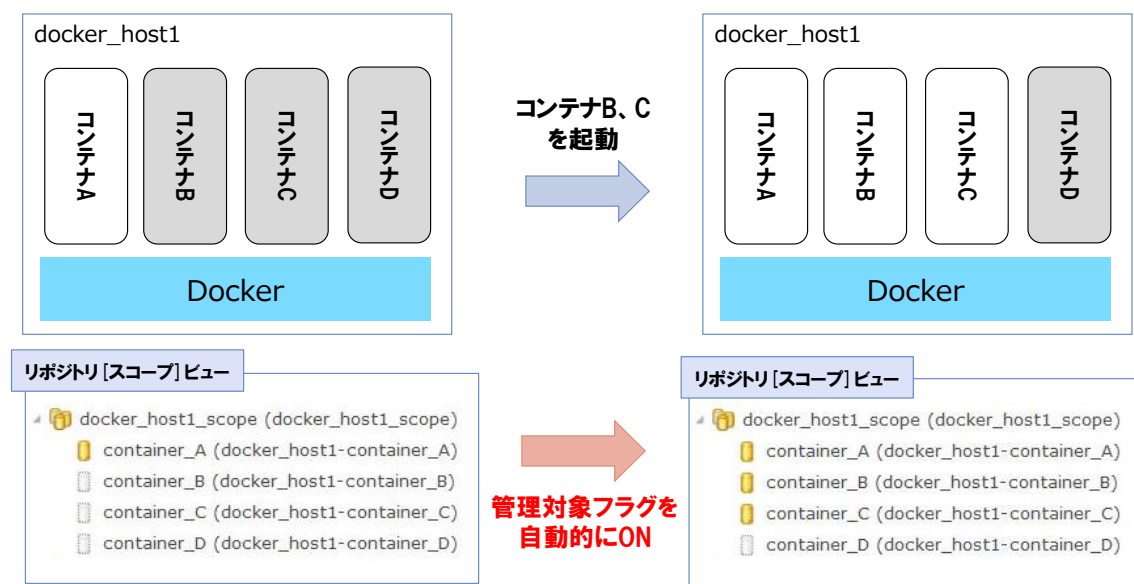


図 1-X 管理対象フラグの自動追従

このように常に状態が” running” であるコンテナのみ管理対象（監視やジョブの実行対象）とすることができます。

3. 監視編

本章では、Docker ホストと Docker コンテナの監視方式について記載します。

○背景と目的

Docker を使用したシステムでは、Docker ホストの監視と各 Docker コンテナの監視の必要があります。Docker ホスト監視は、そもそも Docker の動作に必須な Docker デーモンが正常に動作しているかや、Docker ホスト上のリソースが十分かなどの障害検知・サイジング目的などの情報収集があります。Docker コンテナの監視は、Docker コンテナのステータスに問題が無いかや、使用しているリソースに偏りが無いかの障害検知・サイジング目的などの情報収集があります。

○実現方法

Hinemos では、Docker ホストと Docker コンテナの監視を、Hinemos のカスタム監視やシステムログ監視などの監視機能と連携して実現するサンプルスクリプトを用意しました。

[Docker ホストの監視]

Docker ホストの監視には、次の 2 つの機能を用意しています。

・Docker ホストの状態監視

指定した Docker ホストに対し、Docker デーモンが正常に動作しているか、Docker コンテナ数や Docker が使用しているディスク容量を監視します。

・Docker ホストのイベント監視

指定した Docker ホストに対し Docker のイベントをシステムログとしてメッセージ監視をします。

[Docker コンテナの監視]

Docker コンテナの監視には、次の 6 つの機能を用意しています。

・Docker コンテナのリソース監視(CPU)

Docker コンテナが使用している CPU 使用率を監視します。

・Docker コンテナのリソース監視(メモリ)

Docker コンテナが使用しているメモリ使用率を監視します。

- **Docker コンテナのリソース監視(ディスク)**

Docker コンテナが使用しているディスク I/O 量を監視します。

- **Docker コンテナのリソース監視(ネットワーク)**

Docker コンテナが使用しているネットワーク転送量を監視します。

- **Docker コンテナ内のプロセス監視**

Docker コンテナ内で起動している(パターンパッチ表現にマッチする)プロセス数を監視します。

- **Docker コンテナの状態監視**

Docker コンテナの状態を監視します。

[使い方概要]

本機能は、本ドキュメントの他の機能と同様に、Docker ホストとは異なるリモートサーバ(本ドキュメントでは `manager_server`)から実行できるスクリプトとして用意しました。このスクリプトは、Hinemos のカスタム監視機能から実行して結果を収集・閾値監視するものと、システムログ監視を利用して `syslog` メッセージから異常検知するようなものがあります。

これにより、Hinemos 側で指定した業務カレンダーを適用し不要な時間帯の監視を抑制したり、監視の有効化・無効化を Hinemos の設定として切り替えることができます。

本監視機能は全ての監視を利用する必要はなく、要件に応じて選択し設定することを想定しています。そのため、各機能概要から必要有無を判断し、必要な場合は当該機能について詳細の説明のある各節を参照してください。

○本章の構成

Docker の監視に関する 8 機能を順次説明します。

- Docker ホストの状態監視
- Docker ホストのイベント監視
- Docker コンテナのリソース監視(CPU)
- Docker コンテナのリソース監視(メモリ)
- Docker コンテナのリソース監視(ディスク)
- Docker コンテナのリソース監視(ネットワーク)

- Docker コンテナ内のプロセス監視
- Docker コンテナの状態監視

○注意事項

特にありません。

3.1. Docker ホストの状態監視

3.1.1. 概要

指定した Docker ホストに対し、Docker デーモンが正常に動作しているか、Docker コンテナ数や Docker が使用しているディスク容量を監視します。

3.1.2. 仕様・アーキテクチャ

Docker ホストの監視には、Docker が動作する Linux サーバの死活状態やリソース使用状況の監視の他に、Docker 専用の監視として、次のような監視が必要になります。

監視項目	説明	頻度
Docker デーモンの正常性	Docker デーモンの正常性の監視により、Docker コンテナの障害か Docker 自体の障害かを即時に把握することが出来る。	1 分～5 分間隔程度
コンテナ数	停止後に消し忘れた Docker コンテナを把握するなど、動作状況を確認できる。	60 分間隔程度
ディスク使用容量	Docker が利用しているディスク容量（イメージ、コンテナ）を把握し、将来の見積もりや分析に使用できる。	60 分間隔程度

○Docker デーモンの正常性

Docker デーモンの正常性をリモートから監視するため、Remote API の Ping を使用します。

Remote API の HTTP リクエスト	HTTP レスポンス
/_ping	• Docker デーモンが正常に動作している場合 標準出力に” OK”、ステータスコード：200 • Docker デーモンに異常がある場合 ステータスコード：500

• Docker Remote API 実行例

[Docker デーモン起動時]

```
(root) # curl http://docker_host1:2375/_ping
```

OK

[Docker デーモン停止時]

```
(root) # curl http://docker_host1:2375/_ping
```

curl: (7) Failed connect to docker_host1:2375; Connection refused

○コンテナ数

Remote API の `containers/json` を実行することで、Docker コンテナの一覧を取得することができます（パラメータ `all=1` を付与することで全ての状態のコンテナの情報を取得できます）。このレスポンスの”Status”の情報を利用して、状態毎のコンテナ数をカウントすることが可能になります。

Remote API の HTTP リクエスト	HTTP レスポンス
<code>/containers/json?all=1</code>	全コンテナの情報を取得 Status にてコンテナの状態 (State) を識別

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/json?all=1 | jq '.'
```

```
[
  {
    "Id": "90b56eb899e74b5927d940d5dbd4f83dde7c1a1b5bfbcfb32cc1335cc8391e2a",
    "Names": [
      "/container_D"
    ],
    "Image": "httpd:2.4",
    "Command": "httpd -D FOREGROUND",
    "Created": 1441463410,
    "Ports": [],
    "Labels": {
      "hinemosAssignScopeId": "web1"
    },
    "Status": "Exited (0) 2 minutes ago"
```

```
},
(中略)
{
  "Id": "d7ee4d6ee44cd4fc60127765408f5384f215dcd18a62545a36b42e3f91104482",
  "Names": [
    "/container_A"
  ],
  "Image": "httpd:2.4",
  "Command": "httpd -D FOREGROUND",
  "Created": 1441463404,
  "Ports": [
    {
      "PrivatePort": 80,
      "Type": "tcp"
    }
  ],
  "Labels": {
    "hinemosAssignScopeId": "web1"
  },
  "Status": "Up 2 minutes"
}
]
```

Docker コンテナの状態 (State) と上記 Status のマッピングについては、5 章の TIPS を参照してください。

○ディスク使用容量

Remote API の containers/json のパラメータ size=1 を付与することで、各コンテナのディスク容量を取得可能になります。

Remote API の images/json より Docker ホスト上のイメージが使用しているサイズが確認できます。

Remote API の HTTP リクエスト	HTTP レスポンス
/containers/json?all=1&size=1	全コンテナの情報を取得 SizeRw にてコンテナのサイズを取得
/images/json?all=1	全イメージの情報を取得

Size にてコンテナのサイズを取得

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/images/json?all=1
```

```
(root) # curl http://docker_host1:2375/containers/json?all=1&size=1
```

※実行結果が非常に大きいため、実行結果は割愛します。

3.1.3. スクリプトの使い方 (CLI 実行)

○スクリプト名

MonitorDockerHostStatus.py

○書式

MonitorDockerHostStatus.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
--status	Docker デーモンの正常性(0:正常、1:異常)を監視 (デフォルト)	(*1)
--count	Docker ホスト上のコンテナ数を状態別に表示	(*1)
--size	Docker ホスト上のコンテナとイメージが使用するディスク容量を表示 コンテナのディスク容量は状態別に表示	(*1)
-h, --help	スクリプトの使用方法を表示	-

(*1)-status オプションまたは-count オプションまたは-size のいずれか 1 つ指定が必須。
何も指定しない場合は、--status となる。

○標準出力 (監視情報)

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準出力は key, value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、

「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。
出力される数字はいずれも瞬時値（実行した瞬間の情報）です。

[Docker デーモンの正常性]

項目	説明
Key	Status
Value	0:正常、1:異常

[コンテナ数]

項目	説明
Key	コンテナ状態 (running、restarting、paused、exited)
Value	コンテナ数 (単位: 個)

[ディスク使用容量]

項目	説明
Key	イメージ (image)、 コンテナ状態 (running、restarting、paused、exited)
Value	ディスク使用量 (物理サイズ、単位: byte)

○戻り値

正常: 0

異常: 1

○注意事項

- ・ディスク使用容量の負荷

Docker ホストの状態監視の中でも、ディスク使用容量の監視は、対象の Docker ホスト上から全コンテナと全イメージのディスク使用量に付随する多くの情報を取得する API を実行します。この API の実行負荷は大きく、Docker デーモンの CPU 使用率が高騰するようなケースがあります。そのため、短い間隔での実行は推奨しません。

○実行例

[Docker デーモンの正常性]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i
```



```
docker_host1 --status
```

```
Status, 0
```

[コンテナ数]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i
```

```
docker_host1 --count
```

```
running, 3
```

```
restarting, 0
```

```
paused, 0
```

```
exited, 0
```

[ディスク使用容量]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i
```

```
docker_host1 --size
```

```
image, 474361657
```

```
running, 6
```

```
restarting, 0
```

```
paused, 0
```

```
exited, 0
```

3.1.4. Hinemos での使い方

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

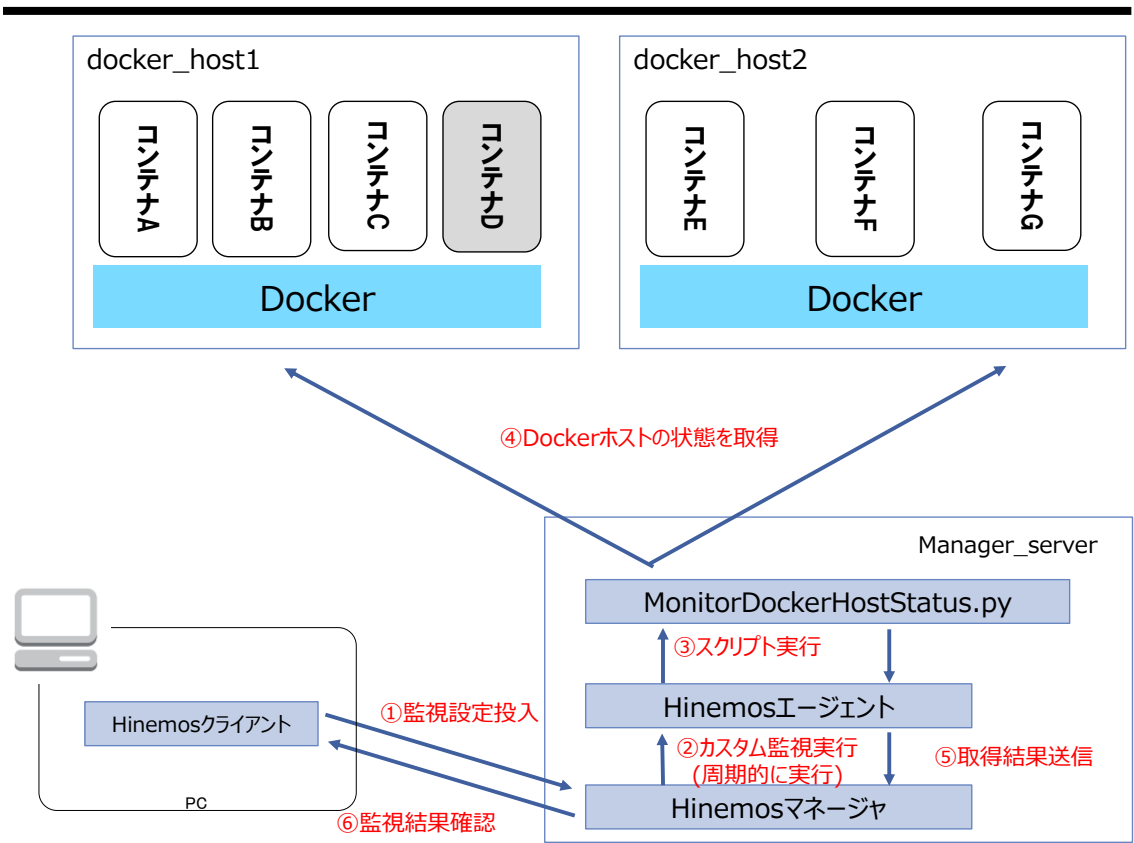


図 Docker コンテナの状態取得の流れ

○カスタム監視の設定例

全 Docker ホストの状態監視をするカスタム監視設定の例を以下に示します。

[Docker デーモンの正常性]

Docker デーモンの正常性の監視は非常に重要なので、1～5 分などの短い時間間隔で実行します。Docker デーモンの異常は、Docker 環境に大きく影響を与えますが、Docker ホストの負荷が非常に高く誤検知（瞬断）を検知しやすい可能性も考えられます。その場合は、通知設定の初回通知までの回数を調整することで対応します。

正常（0）か異常（1）の真偽値の監視のため、取得した結果を数値情報として蓄積することは通常は必要ありません。

表 カスタム監視の設定(DOC_HOST_STATUS)

監視項目 ID	DOC_HOST_STATUS
説明	Docker デーモンの正常性の監視
スコープ	docker_host（スコープ）
間隔	5 分

指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i #[IP_ADDRESS]
タイムアウト	15000
監視	チェック
情報の閾値	0（下限）～1（上限）
警告の閾値	0（下限）～1（上限）
通知 ID	EVENT_FOR_POLLING
アプリケーション	DOC_HOST_STATUS
収集	未チェック

[コンテナ数]

システム特性によりますが、コンテナ数の増減が分単位で発生するようなケースでない限り、コンテナ数は中長期的なサイジングに向けての指標に利用します。そのため、コンテナ数による閾値監視より、取得した結果を蓄積するケースが多いと想定れます。

閾値監視を行わず、取得した結果を蓄積するケースの設定例を示します。

表 カスタム監視の設定 (DOC_HOST_COUNT)

監視項目 ID	DOC_HOST_COUNT
説明	コンテナ数の監視
スコープ	docker_host（スコープ）
間隔	60 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i #[IP_ADDRESS] --count
タイムアウト	15000
監視	未チェック
収集	チェック

収集値表示名	コンテナ数
収集値単位	個

[ディスク使用容量]

コンテナ数の監視と同様に、ディスク使用容量は中長期的なサイジングに向けての指標に利用します。監視については、通常は、Docker ホストの動作する Linux サーバ上のディスク使用率の閾値監視で十分です。

閾値監視を行わず、取得した結果を蓄積するケースの設定例を示します。

表 カスタム監視の設定 (DOC_HOST_SIZE)

監視項目 ID	DOC_HOST_ SIZE
説明	コンテナのディスク使用容量の監視
スコープ	docker_host (スコープ)
間隔	60 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerHostStatus.py -i #[IP_ADDRESS] --size
タイムアウト	15000
監視	未チェック
収集	チェック
収集値表示名	ディスク容量
収集値単位	byte

3.2. Docker ホストのイベント監視

3.2.1. 概要

指定した Docker ホストに対して Docker のイベントをシステムログとしてメッセージ監視をします。

3.2.2. 仕様・アーキテクチャ

Docker コンテナは簡易に作成・削除が可能のため、例えば人為的な障害が発生した場合に、

対象の Docker コンテナに対してどのような操作が行われたかなど、時系列に追いかける必要があります。そのため、Docker イベントを履歴情報として蓄積する必要があります。

監視項目	説明	頻度
Docker イベント	Docker イベント (create、restart などの) を監視する	随時

Docker イベントは記の Docker Remote API より取得します。

Remote API の HTTP リクエスト	HTTP レスポンス
/events?since=時刻	指定した時刻以降の Docker イベント

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/events?since=1374067924
{"status":"destroy","id":"b427e1a79fb9f88060f90a4c6ca6301f5e83a75d4dcbae006a1c01b7229359e3","from":"httpd:2.4","time":1441463396}
{"status":"destroy","id":"6c51941883ba3ed92266cdce610e80cc8929de54eaf15b401890193a0a5f5280","from":"httpd:2.4","time":1441463399}
{"status":"destroy","id":"3c2eed193e26657073cf7ba612401f3485a7aead6f18a632bee8b9bf8dd46e09","from":"httpd:2.4","time":1441463402}
```

3.2.3. スクリプトの使い方(CLI)

○スクリプト名

MonitorDockerHostEvent.py

○書式

MonitorDockerHostEvent.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-k KEY_ID	監視設定毎のユニークな ID	○
-s SYSLOG_HOST:PORT	監視結果の syslog 送信先を指定	-(*)

-f LOGFILE	監視結果のファイル出力先を指定	-(*)
-h, --help	スクリプトの使用方法を表示	-

(*)-s オプションまたは-f オプションは何れか1つを指定する必要がある。

○標準出力（監視情報）

本スクリプトは、cron にて定期的に行われ、Hinemos のシステムログ監視による利用を想定しています。Hinemos のシステムログ監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.14 システムログ監視」を参照してください。

-s オプションまたは-f オプションは何れも標準出力には出力されません。

ファイル出力書式

[時刻] [-i で指定した文字列] [Docker イベント]

syslog 送信メッセージ書式

[時刻] [-i で指定した文字列] [Docker イベント]

※実行した環境のシステムログとして検知します。

○注意事項

特にありません

○実行例

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerHostEvent.py -i docker_host1
-k hoge_event -s 127.0.0.1:514
```

```
(root) # tail -f /var/log/messages
```

```
Sep  6 08:18:44 docker_host1 {u'status': u'restart', u'from': u'httpd:2.4',
u'id': u'd7ee4d6ee44cd4fc60127765408f5384f215dcd18a62545a36b42e3f91104482',
u'time': 1441527524}
```

```
Sep  6 08:24:06 docker_host1 {u'status': u'die', u'from': u'httpd:2.4', u'id':
u'd7ee4d6ee44cd4fc60127765408f5384f215dcd18a62545a36b42e3f91104482', u'time':
1441527846}
```

```
Sep  6 08:24:06 docker_host1 {u'status': u'start', u'from': u'httpd:2.4', u'id':
u'd7ee4d6ee44cd4fc60127765408f5384f215dcd18a62545a36b42e3f91104482', u'time':
1441527846}
```

```
Sep  6 08:24:06 docker_host1 {u'status': u'restart', u'from': u'httpd:2.4',
u'id': u'd7ee4d6ee44cd4fc60127765408f5384f215dcd18a62545a36b42e3f91104482',
```

```
u'time': 1441527846}
```

3.2.4. スクリプトの使い方(Hinemos)

本サンプルスクリプトは、cron により定期的の実行し、Hinemos のシステムログ監視機能と組み合わせて使用します。

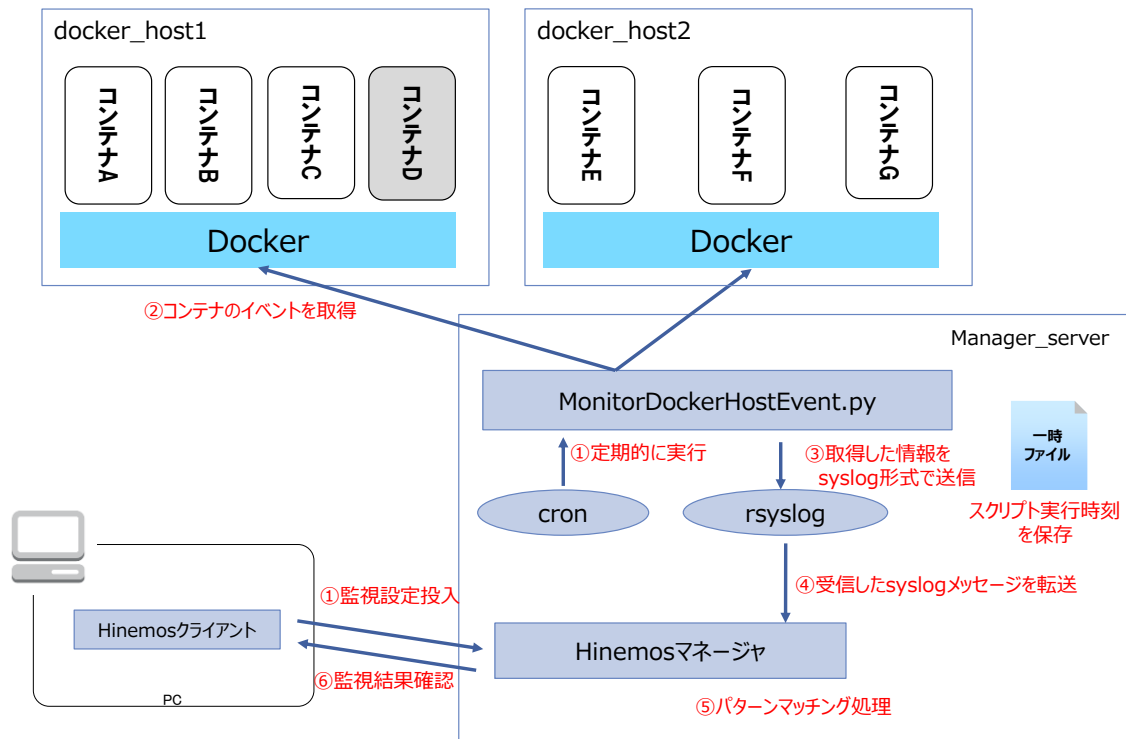


図 Docker イベント情報取得の流れ

○cron の設定例

`manager_server` 上の `cron` にて定期的の実行します。これにより Hinemos マネージャに `syslog` メッセージが定期的送信されるようになります。

```
(root) # crontab -l
*/5 * * * * /opt/hinemos_agent/docker/bin/MonitorDockerHostEvent.py -i
docker_host1 -k docker_host1_event -s 127.0.0.1:514 > /dev/null
*/5 * * * * /opt/hinemos_agent/docker/bin/MonitorDockerHostEvent.py -i
docker_host1 -k docker_host2_event -s 127.0.0.1:514 > /dev/null
```

○システムログ監視の設定例

`cron` 契機で定期的送信されてくる `syslog` を監視するシステムログ監視機能の設定例を

示します。通常時は直ちに Docker イベントを監視しアラートを上げることはないため、全てのイベントを単にイベント通知にて履歴として蓄積する例を示します。

Docker イベントはそのイベントが発生した Docker ホストのノードのイベントとして検知されます。

表 システムログ監視の設定 (CONTAINER_EVET)

監視項目 ID	CONTAINER_EVENT
説明	Docker イベントの監視
スコープ	docker_host

表 文字列パターンマッチングの設定

順序	条件	パターンマッチ表現	重要度	有効/無効
1	条件に一致したら処理する	.*	情報	有効

3.3. Docker コンテナのリソース監視 (CPU)

3.3.1. 概要

Docker コンテナが使用している CPU 使用率を監視します。

3.3.2. 仕様・アーキテクチャ

Docker ホストや Docker コンテナの CPU 使用率の監視には、Docker が動作する Linux サーバの死活状態やリソース使用状況の監視の他に、Docker 専用の監視として、次のような監視が必要になります。Docker コンテナは起動時に使用するコアを指定できるため (--cpu-set)、設定によってはコア別に使用率が偏る可能性があるため、全体及びコア別の 2 種類の監視が必要になります。

監視項目	説明	頻度
Docker コンテナの CPU 使用率	Docker コンテナが使用する CPU 使用率	1 分～10 分間隔程度
Docker コンテナのコア別 CPU 使用率	Docker コンテナが使用するコア別 CPU 使用率	1 分～10 分間隔程度

Docker のリソース情報は Stats API を利用することで取得できます。curl 等で取得すると、継続的に値が追加で送信され続けてきます。

Remote API の HTTP リクエスト	HTTP レスポンス
/containers/(id)/stats	指定のコンテナのリソース情報

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/container_B/stats
{"read": "2015-09-05T21:59:08.423347821Z", "network": {"rx_bytes": 1854, "rx_packets": 23, "rx_errors": 0, "rx_dropped": 0, "tx_bytes": 738, "tx_packets": 9, "tx_errors": 0, "tx_dropped": 0}, "pre
cpu_stats": {"cpu_usage": {"total_usage": 0, "percpu_usage": null, "usage_in_kernelmod
e": 0, "usage_in_usermode": 0}, "system_cpu_usage": 0, "throttling_data": {"periods": 0,
"throttled_periods": 0, "throttled_time": 0}}, "cpu_stats": {"cpu_usage": {"total_usag
e": 19377187586, "percpu_usage": [10738949512, 8638238074], "usage_in_kernelmode": 600
00000, "usage_in_usermode": 400000000}, "system_cpu_usage": 1473918600000000, "throttli
ng_data": {"periods": 0, "throttled_periods": 0, "throttled_time": 0}}, "memory_stats":
{"usage": 14479360, "max_usage": 14712832, "stats": {"active_anon": 8556544, "active_fi
le": 352256, "cache": 5922816, "hierarchical_memory_limit": 9223372036854775807, "hier
archical_mems_wlimit": 9223372036854775807, "inactive_anon": 258048, "inactive_file"
: 5312512, "mapped_file": 2314240, "pgfault": 2547, "pgmajfault": 25, "pgpgin": 2143, "pgp
gout": 141, "rss": 8556544, "rss_huge": 6291456, "swap": 0, "total_active_anon": 8556544,
"total_active_file": 352256, "total_cache": 5922816, "total_inactive_anon": 258048, "t
otal_inactive_file": 5312512, "total_mapped_file": 2314240, "total_pgfault": 2547, "to
tal_pgmajfault": 25, "total_pgpgin": 2143, "total_pgpgout": 141, "total_rss": 8556544, "
total_rss_huge": 6291456, "total_swap": 0, "total_unevictable": 0, "unevictable": 0}, "f
ailcnt": 0, "limit": 3609448448}, "blkio_stats": {"io_service_bytes_recursive": [{"maj
or": 253, "minor": 2, "op": "Read", "value": 5640192}, {"major": 253, "minor": 2, "op": "Writ
e", "value": 0}, {"major": 253, "minor": 2, "op": "Sync", "value": 0}, {"major": 253, "minor"
: 2, "op": "Async", "value": 5640192}, {"major": 253, "minor": 2, "op": "Total", "value": 564
0192}], "io_serviced_recursive": [{"major": 253, "minor": 2, "op": "Read", "value": 323},
{"major": 253, "minor": 2, "op": "Write", "value": 0}, {"major": 253, "minor": 2, "op": "Sync
", "value": 0}, {"major": 253, "minor": 2, "op": "Async", "value": 323}, {"major": 253, "mino
r": 2, "op": "Total", "value": 323}], "io_queue_recursive": [], "io_service_time_recursi
ve": [], "io_wait_time_recursive": [], "io_merged_recursive": [], "io_time_recursive":
[], "sectors_recursive": []}}
...
```

• CPU 使用率の計算方法

一般に CPU 使用率の計算方法は、2 点間の CPU のカウンタ値から差分を算出して、その期間の平均使用率となります。運用ツールから使用する場合は、一定の監視間隔でその計算を行います。

Docker コンテナの CPU 利用率は、Stats API の次の情報を使用して算出します。

取得項目	説明
read	Docker API にアクセスした時刻
total_usage	Docker コンテナの CPU 使用時間の累積値(*1) (ナノ秒単位)
percpu_usage	Docker コンテナのコア毎の CPU 使用時間の累計値(*1) (ナノ秒単位)

(*1)total_usage は全てのコアの CPU 使用率を含めた値で、percpu_usage の合計に等しい。

[Docker コンテナの CPU 使用率の計算式]

$$\text{CPU 使用率} = (T[n] - T[n-1]) / (r[n] - r[n-1]) [\%]$$

T[n]:n 回目の取得時の total_cpu の値

r[n]:n 回目取得時の時刻(ナノ秒単位に変換したもの)

[Docker コンテナのコア別 CPU 使用率の計算式]

$$\text{CPU 使用率} = (Ci[n] - Ci[n-1]) / (r[n] - r[n-1]) [\%]$$

Ci[n]:n 回目の取得時のコア i の percpu_ の値

r[n]:n 回目取得時の時刻(ナノ秒単位に変換したもの)

3.3.3. サンプルコードの使い方(CLI)

○スクリプト名

MonitorDockerContainerRepositoryCPU.py

○書式

MonitorDockerContainerRepositoryCPU.py [options]

オプション	動作	必須
-------	----	----

-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.01)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-sv SNMP_VER	SNMP のバージョン (default: '1')	-(*)1
-sc SNMP_COM	SNMP のコミュニティ (default: 'public')	-(*)1
-sp SNMP_PORT	SNMP のポート (default: '161')	-(*)1
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト : 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト : 全コンテナを対象)	-
-k KEY_ID	監視設定毎のユニークな ID	○
--all	CPU 全体の結果を表示 (--core とは排他)	○(*)2
--core	CPU コア別に結果を表示 (--all とは排他)	○(*)2
--with-host	ホストの CPU 使用率 (Linux サーバ全体の CPU 使用率から Σ コンテナの CPU 利用率を引いたもの) を合わせて表示 -c、-n、--core オプションと同時指定不可 (SNMP プロトコルによる通信発生)	-
-h, --help	スクリプトの使用方法を表示	-

(*)1) SNMP プロトコルは、--with-host オプション指定時に使用する。指定する値は、対象サーバの SNMP サービスに接続するための設定。

(*)2) -all オプションまたは-core オプションまたは-size のいずれか 1 つ指定が必須。何も指定しない場合は、--status となる。

○標準出力 (監視情報)

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準

出力は key, value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。

[Docker コンテナの CPU 使用率]

項目	説明
Key	cpu_usage:[コンテナ名 host]
Value	CPU 使用率

[Docker コンテナのコア別 CPU 使用率]

項目	説明
Key	cpu_usage#[コア番号]
Value	CPU 使用率

○戻り値

正常 : 0

異常 : 1

○注意事項

- ・-k KEY_ID の指定について

CPU 使用率の算出には、前回取得時のカウンタ値と時刻などの情報を記録しておく必要があります。本スクリプトでは、-k KEY_ID の指定により、/tmp/KEY_ID.txt に記録します。これにより、同一サーバ上で異なるユーザの利用または異なる Docker ホストへの監視が可能になります。Hinemos のカスタム監視から使用する場合は、本値が実行単位ごとに必ず異なるように注意してください。

○実行例

[Docker コンテナの CPU 使用率(--with-host なし)]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceCpu.py -i
docker_host1 -k hoge --all
cpu_usage:container_C,0.07
cpu_usage:container_B,0.08
cpu_usage:container_A,0.08
```

[Docker コンテナの CPU 使用率(--with-host あり)]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceCpu.py -i
docker_host1 -k hoge --all --with-host
cpu_usage:container_C,0.04
cpu_usage:container_B,0.04
cpu_usage:container_A,0.04
cpu_usage:host,0.65
```

[Docker コンテナのコア別 CPU 使用率]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceCpu.py -i
docker_host1 -k hoge --core
cpu_usage#0:container_C,0.05
cpu_usage#1:container_C,0.03
cpu_usage#0:container_B,0.00
cpu_usage#1:container_B,0.08
cpu_usage#0:container_A,0.03
cpu_usage#1:container_A,0.05
```

3.3.4. Hinemos での使い方

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

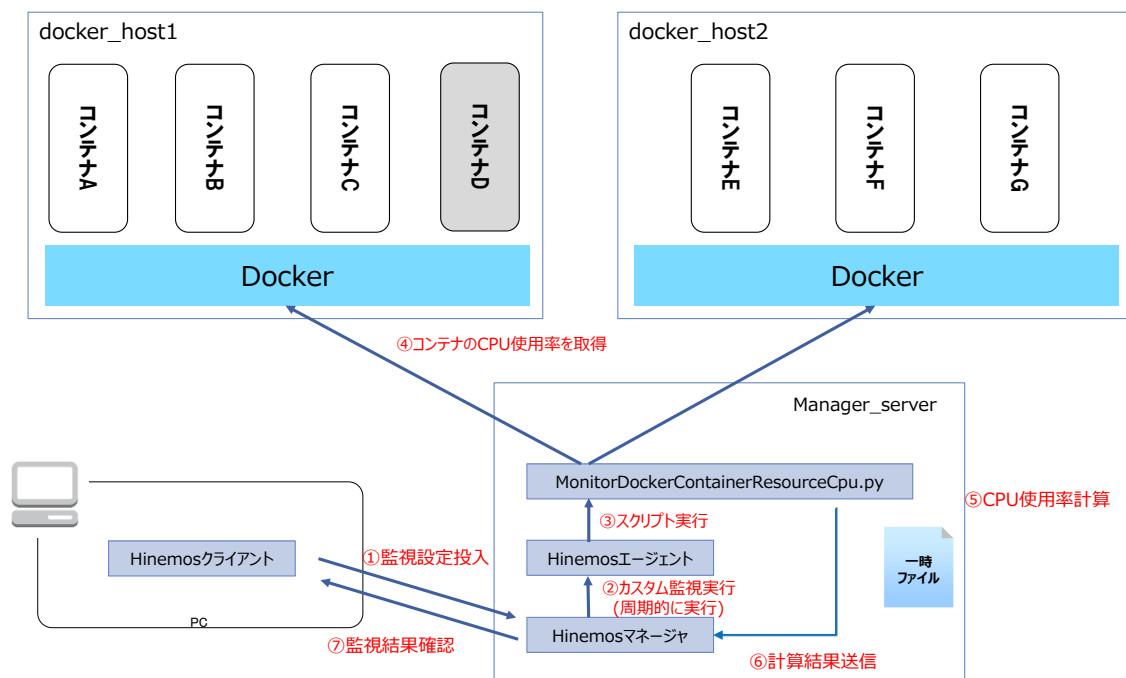


図 Docker コンテナの CPU 使用率取得の流れ

○カスタム監視の設定例

Docker コンテナの CPU 使用率の監視をするカスタム監視設定の例を以下に示します。

表 カスタム監視の設定 (CONTAINER_CPU)

監視項目 ID	CONTAINER_CPU
説明	Docker コンテナの CPU 使用率の監視
スコープ	docker_host (スコープ)
間隔	5 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerContainerResource Cpu.py -i #[IP_ADDRESS] -k #[NODE_NAME]_CONTAINER_CPU --all
タイムアウト	15000
監視	チェック
情報の閾値	0 (下限) ~80 (上限)
警告の閾値	80 (下限) ~90 (上限)
通知 ID	EVENT_FOR_POLLING
アプリケーション	CONTAINER_CPU
収集	チェック
収集値表示名	CPU 使用率
収集値単位	%

図 CPU 使用率取得のための監視設定

実行するコマンドには、ノード変数を使用します。以下のオプションを使用する場合は、次のように指定します。

オプション	値	意味
-i	#[IP_ADDRESS]	監視対象ノードの IP アドレス
-k	#[NODE_NAME]_(監視項目 ID)	CPU 使用率の計算時に作成される一時ファイル名を指定
-c	#[NODE_NAME]	監視対象ノードのノード名 監視対象に Docker ホストを指定した場合は必要ない

Docker コンテナのコア別 CPU 使用率も同様に指定できます。

○注意事項

- ・ 値取得失敗の不明

CPU 使用率の計算には、2 点間の情報が必要なため監視の初回実行や KEY_ID のファイル

が消されたような環境では値取得失敗の不明になる可能性があります。

3.4. Docker コンテナのリソース監視(メモリ)

3.4.1. 概要

Docker コンテナが使用しているメモリ使用率を監視します。

3.4.2. 仕様・アーキテクチャ

Docker コンテナのメモリ使用率に関する監視には、Docker が動作する Linux サーバの死活状態やリソース使用状況の監視の他に、次のような監視が必要になります。Docker コンテナは起動時に使用するメモリの上限を設定できるため、Linux サーバ全体でのメモリ利用率か上限値に対するメモリ使用率かで意味が変わります。メモリ使用率は、全て各時点での瞬時値になります。

監視項目	説明	頻度
Docker コンテナのホストに対するメモリ使用率	各 Docker コンテナが使用する CPU 使用率 使用率の母数は、Linux サーバの搭載物理メモリサイズになる	1 分～10 分間隔程度
Docker コンテナの上限値に対するメモリ使用率	各 Docker コンテナが使用する CPU 使用率 使用率の母数は、使用するメモリの上限設定がある場合はこの制限値を、ない場合は Linux サーバの搭載物理メモリサイズになる	1 分～10 分間隔程度
Docker コンテナのメモリ詳細情報	Docker コンテナが使用するメモリに関する詳細情報	1 分～10 分間隔程度

○Docker コンテナのホストに対するメモリ使用率

コンテナ毎のメモリ使用量は Stats API を利用することで取得できます。また、Docker ホストの動作する Linux サーバの物理メモリサイズなどのメモリ情報は、SNMP により取得可能です。

[コンテナ毎のメモリ使用率]

取得項目	説明
memorystats	Docker コンテナが使用している物理メモリの使用量

stats	(単位 : Byte)
rss	

[Docker ホストの動作する Linux サーバのメモリ情報 (SNMP)]

OID	項目名称	説明
. 1. 3. 6. 1. 4. 1. 2021. 4. 5. 0	memTotalReal	使用中メモリと使用可能メモリの総量 (単位 : KByte)
. 1. 3. 6. 1. 4. 1. 2021. 4. 6. 0	memAvailReal	使用可能メモリ (単位 : KByte)
. 1. 3. 6. 1. 4. 1. 2021. 4. 14. 0	memBuffer	
. 1. 3. 6. 1. 4. 1. 2021. 4. 15. 0	memCached	

[Docker コンテナのホストに対するメモリ使用率の計算方法]

$$\text{メモリ使用率} = \text{rss} / \text{host_rss} \times 100 [\%]$$

rss : コンテナの物理メモリ使用量

host_rss : host_rss : ホストのメモリ容量

○Docker コンテナの上限値に対するメモリ使用率

コンテナ毎のメモリ使用量は先述の通り、Stats API を利用することで取得できます。コンテナのメモリ上限値は Remote API の inspect により取得可能です。

[HTTP リクエスト: /containers/(id)/json]

取得項目	説明
Config Memory	Docker コンテナが使用している物理メモリの使用量(単位 : Byte)

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/container_B/json
{
  "AppArmorProfile":"","
  "Args": [],
  "Config": {
    "AttachStderr": false,
    ~ 中略 ~
```

"Memory":5242880 # コンテナのメモリ上限値

"MemorySwap":0,

~ 以下略 ~

[Docker コンテナの上限値に対するメモリ使用率]

Docker コンテナのメモリ使用率 = $M_container / L_container \times 100$ [%]

M_container : コンテナの物理メモリ使用量(= rss)

L_container : コンテナが使用可能なメモリの上限値(= Memory)

○Docker コンテナのメモリ詳細情報

Docker の Stats API には、個々のコンテナに対してメモリに関する詳細な情報が含まれています。リアルタイムでの監視は必要ありませんが、中長期目的でのサイジングや障害階席などで有用です。

・メモリに関する詳細な情報

active_anon

active_file

cache

hierarchical_memory_limit

hierarchical_memsw_limit

inactive_anon

inactive_file

mapped_file

pgfault

pgmajfault

pgpgin

pgpgout

rss

rss_huge

swap

total_active_anon

total_active_file

total_cache

total_inactive_anon

total_inactive_file

```
total_mapped_file
total_pgfault
total_pgmajfault
total_pgpgin
total_pgpgout
total_rss
total_rss_huge
total_swap
total_unevictable
unevictable
```

○Docker コンテナ以外の Linux サーバ上のメモリ使用率に関して

オプションによって、各 Docker コンテナのメモリ使用率と一緒に、Docker コンテナ以外のプロセスが使用しているメモリ使用率が出力できます。これは、Docker ホストのメモリ使用率から全 Docker コンテナのメモリ使用率の合計を差し引くことで求めます。

[Docker コンテナ以外の Linux サーバ上のメモリ使用率]

$$(M_host_used - M_container_sum) / M_host_total \times 100 [\%]$$

M_host_used : ホストのメモリ使用量

M_container_sum : 全コンテナのメモリ使用量の合計値

M_host_total : ホストのメモリ総量

3.4.3. スクリプトの使い方(CLI)

○スクリプト名

MonitorDockerContainerResourceMemory.py

○書式

MonitorDockerContainerResourceMemory.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.01)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-

-sv SNMP_VER	SNMP のバージョン (default: '1')	-(*)
-sc SNMP_COM	SNMP のコミュニティ (default: 'public')	-(*)
-sp SNMP_PORT	SNMP のポート (default: '161')	-(*)
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト: 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト: 全コンテナを対象)	-
--container	コンテナの設定上限に対するメモリ使用率を表示 (デフォルト: ホストに対するメモリ利用率)	-
--with-host	Docker コンテナ以外の Linux サーバ上のメモリ使用率を表示	-
--all	Docker コンテナのメモリ詳細情報を表示	-
-h, --help	スクリプトの使用方法を表示	-

(*)SNMP プロトコルは、--with-host オプション指定時に使用する。指定する値は、対象サーバの SNMP サービスに接続するための設定。

○標準出力（監視情報）

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準出力は key,value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。

[Docker コンテナのホストに対するメモリ使用率]

項目	説明
Key	host_mem_usage:[コンテナ ID host]
Value	Docker コンテナのホストに対するメモリ使用率[, Docker コンテナ以外の Linux サーバ上のメモリ使用率]

[Docker コンテナの上限値に対するメモリ使用率]

項目	説明
Key	host_mem_usage:[コンテナ ID]
Value	Docker コンテナの上限値に対するメモリ使用率

[Docker コンテナのメモリ詳細情報]

項目	説明
Key	Docker Remot API の Stats のメモリ情報
Value	対応する値

○戻り値

正常 : 0

異常 : 1

○注意事項

特にありません。

○実行例**[Docker コンテナのホストに対するメモリ使用率]**

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceMemory.py -  
i docker_host1  
host_mem_usage:4dfd0a12, 0.24  
host_mem_usage:c7a095b8, 0.24  
host_mem_usage:d7ee4d6e, 0.24
```

[Docker コンテナの上限値に対するメモリ使用率]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceMemory.py -  
i docker_host1 --container  
host_mem_usage:4dfd0a12, 0.24  
host_mem_usage:c7a095b8, 0.24  
host_mem_usage:d7ee4d6e, 0.24
```

[Docker コンテナのメモリ詳細情報]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceMemory.py -
```

```
i docker_host1 -n container_B --all
active_anon, 8556544
active_file, 352256
cache, 5922816
hierarchical_memory_limit, 9223372036854775807
hierarchical_memsw_limit, 9223372036854775807
inactive_anon, 258048
inactive_file, 5312512
mapped_file, 2314240
pgfault, 2547
pgmajfault, 25
pgpgin, 2143
pgpgout, 141
rss, 8556544
rss_huge, 6291456
swap, 0
total_active_anon, 8556544
total_active_file, 352256
total_cache, 5922816
total_inactive_anon, 258048
total_inactive_file, 5312512
total_mapped_file, 2314240
total_pgfault, 2547
total_pgmajfault, 25
total_pgpgin, 2143
total_pgpgout, 141
total_rss, 8556544
total_rss_huge, 6291456
total_swap, 0
total_unevictable, 0
unevictable, 0
```

3.4.4. Hinemos での使い方

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

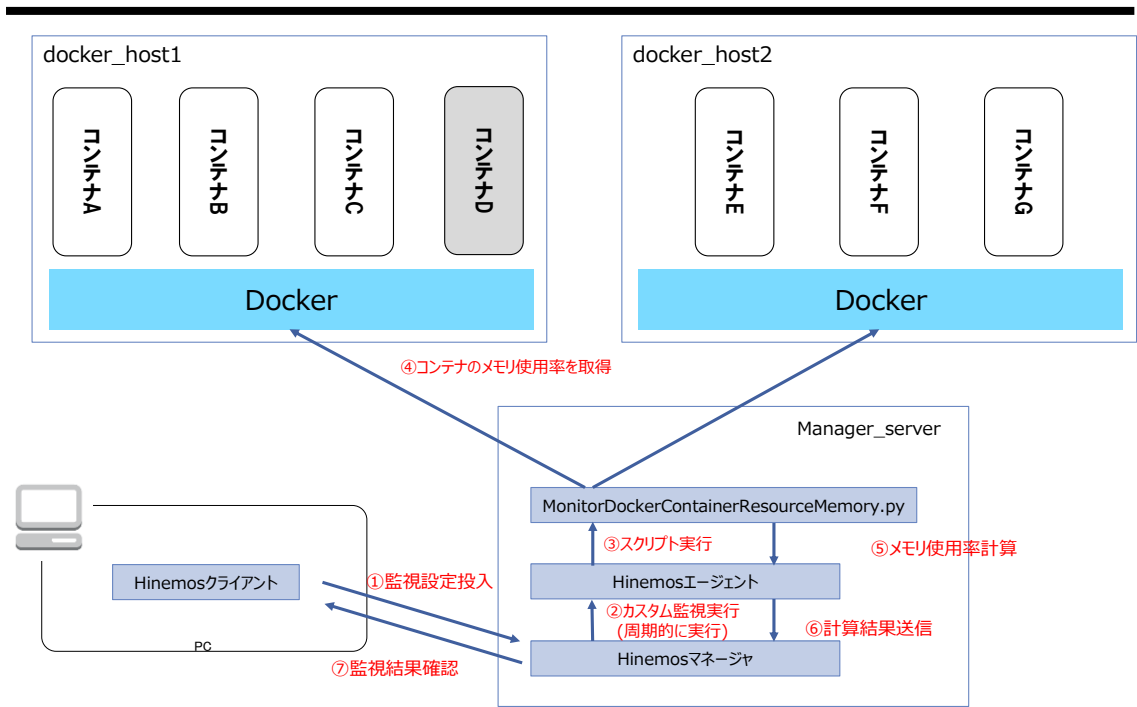


図 Docker コンテナのメモリ使用率取得の流れ

○カスタム監視の設定例

Docker コンテナのメモリ使用率の監視をするカスタム監視設定の例を以下に示します。

表 カスタム監視の設定 (CONTAINER_MEM)

監視項目 ID	CONTAINER_MEM
説明	Docker コンテナのメモリ使用率の監視
スコープ	docker_host (スコープ)
間隔	5 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceMemory.py -i #[IP_ADDRESS]
タイムアウト	15000
監視	チェック
情報の閾値	0 (下限) ～80 (上限)
警告の閾値	80 (下限) ～90 (上限)
通知 ID	EVENT_FOR_POLLING
アプリケーション	CONTAINER_MEM

収集	チェック
収集値表示名	メモリ使用率
収集値単位	%

3.5. Docker コンテナのリソース監視(ディスク)

3.5.1. 概要

Docker コンテナが使用しているディスク I/O 量を監視します。

3.5.2. 仕様・アーキテクチャ

Docker が動作する Linux サーバのディスク I/O の監視の他に、次のような Docker コンテナ単位のディスク I/O の監視が必要です。

監視項目	説明	頻度
Docker コンテナのディスク読込量の監視	Docker コンテナ単位の平均ディスク読込量 (単位 : Byte / 間隔)	5 分～10 分間隔程度
Docker コンテナのディスク書込量の監視	Docker コンテナ単位の平均ディスク書込量 (単位 : Byte / 間隔)	5 分～10 分間隔程度

コンテナ毎のディスク I/O 量は Stats API を利用することで取得できます。ディスク I/O の監視は 2 点間のディスク I/O のカウンタ値から差分を算出して、その期間の平均となります。

[コンテナ毎のディスク I/O 量]

取得項目	説明
“blkio_stats” { “io_service_bytes_recursive”	Docker コンテナのディスク I/O 量(累積値) “op” の値により Read 値や Write 値を区別して取得ことができる。

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/container_B/stats
~ 略 ~
blkio_stats":{
    "io_service_bytes_recursive":
```



```
[
{
    "major":7,
    "minor":0,
    "op":"Read",          # ディスク読み取りである
    "value":2039808       # ディスク読み取りの累積値
},
{
    "major":7,
    "minor":0,
    "op":"Write",         # ディスク書き込みである
    "value":0             # ディスク書き込みの累積値
},
~ 略 ~
```

3.5.3. スクリプトの使い方(CLI)

○スクリプト名

MonitorDockerContainerResourceDisk.py

○書式

MonitorDockerContainerResourceDisk.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.01)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト : 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト : 全コンテナを対象)	-

-k KEY_ID	監視設定毎のユニークな ID	○
--read	平均ディスク読込量を表示	-
--write	平均ディスク書込量を表示	-
-h, --help	スクリプトの使用方法を表示	-

○標準出力（監視情報）

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準出力は key, value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。

[Docker コンテナのディスク読込量の監視]

項目	説明
Key	disk_read: コンテナ ID
Value	平均ディスク IO 量（単位 byte）

[Docker コンテナのディスク書込量の監視]

項目	説明
Key	disk_write: コンテナ ID
Value	平均ディスク IO 量（単位 byte）

○戻り値

正常：0

異常：1

○注意事項

- ・ -k KEY_ID の指定について

平均ディスク IO 量の算出には、前回取得時のカウンタ値と時刻などの情報を記録しておく必要があります。本スクリプトでは、-k KEY_ID の指定により、/tmp/KEY_ID.txt に記録します。これにより、同一サーバ上で異なるユーザの利用または異なる Docker ホストへの監視が可能になります。Hinemos のカスタム監視から使用する場合は、本値が実行単位ごとに必ず異なるように注意してください。

○実行例

[Docker コンテナのディスク読込量の監視]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceDisk.py -i
docker_host1 -k hoge_disk --read
disk_read:4dfd0a12, 0.00
disk_read:c7a095b8, 0.00
disk_read:d7ee4d6e, 0.00
```

[Docker コンテナのディスク書込量の監視]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceDisk.py -i
docker_host1 -k hoge_disk --write
disk_write:4dfd0a12, 0.00
disk_write:c7a095b8, 0.00
disk_write:d7ee4d6e, 0.00
```

3.5.4. スクリプトの使い方 (Hinemos)

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

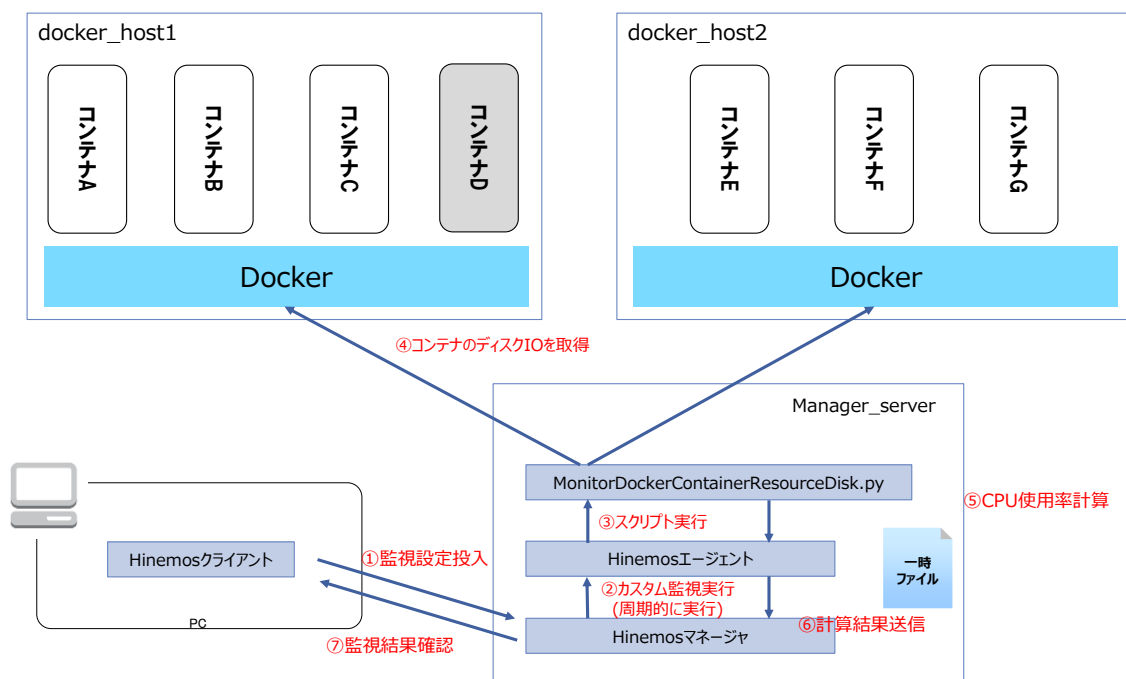


図 Docker コンテナのディスク I/O 量取得の流れ

○カスタム監視の設定例

Docker コンテナのディスク I/O 量の監視をするカスタム監視設定の例を以下に示します。ディスク I/O 量の監視では、閾値監視を行い直ちにアラートを上げるような用途は少なく、

中長期向けのサイジングや障害解析などでの利用が多いため、当該内容の設定例を示します。

表 カスタム監視の設定 (CONTAINER_DISK)

監視項目 ID	CONTAINER_DISK
説明	Docker コンテナのディスク I/O 量の監視
スコープ	docker_host (スコープ)
間隔	5 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceDisk.py -i #[IP_ADDRESS] -k #[NODE_NAME]_CONTAINER_DISK
タイムアウト	15000
監視	未チェック
収集	チェック
収集値表示名	ディスク I/O 量
収集値単位	Byte/5 分

3.6. Docker コンテナのリソース監視(ネットワーク)

3.6.1. 概要

Docker コンテナが使用しているネットワーク転送量を監視します。

3.6.2. 仕様・アーキテクチャ

Docker が動作する Linux サーバのネットワーク I/O の監視の他に、次のような Docker コンテナ単位のネットワーク I/O の監視が必要です。

監視項目	説明	頻度
Docker コンテナのネットワーク転送量(受信)	Docker コンテナ単位の平均ネットワーク転送量(受信) (単位 : Byte / 間隔)	5 分～10 分間隔程度
Docker コンテナのネットワーク転送量(送信)	Docker コンテナ単位の平均ネットワーク転送量(送信) (単位 : Byte / 間隔)	5 分～10 分間隔程度

Docker コンテナのネットワークパケット数(受信)	Docker コンテナ単位の平均ネットワークパケット数(受信) (単位: 個 / 間隔)	5 分～10 分間隔程度
Docker コンテナのネットワークパケット数(送信)	Docker コンテナ単位の平均ネットワークパケット数(送信) (単位: 個 / 間隔)	5 分～10 分間隔程度

コンテナ毎のネットワーク I/O 量は Stats API を利用することで取得できます。ネットワーク I/O の監視は 2 点間のネットワーク I/O のカウンタ値から差分を算出して、その期間の平均となります。

[コンテナ毎のネットワーク I/O 量]

取得項目	説明
“network” { “rx_” または “tx_”	Docker コンテナのネットワーク I/O 量(累積値)

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/container_B/stats
```

```
{
  "read": "2015-08-25T07:10:20.848951084-04:00",
  "network":
  {
    "rx_bytes": 4787112,
    "rx_packets": 3367,
    "rx_errors": 0,
    "rx_dropped": 0,
    "tx_bytes": 116440,
    "tx_packets": 1686,
    "tx_errors": 0,
    "tx_dropped": 0
  },

```

(以下、省略)

3.6.3. スクリプトの使い方(CLI)

○スクリプト名

MonitorDockerContainerResourceNetwork.py

○書式

MonitorDockerContainerResourceNetwork.py

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト : 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト : 全コンテナを対象)	-
-k KEY_ID	監視設定毎のユニークな ID	○
--kb	転送量を表示	-
--pck	パケット数を表示	-
--rx	受信情報のみを表示	-
--tx	送信情報のみを表示	-
-h, --help	スクリプトの使用方法を表示	-

○標準出力（監視情報）

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準出力は key,value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。

[Docker コンテナのネットワーク転送量の監視]

項目	説明
Key	[nw_rxkb nw_txkb]: コンテナ ID
Value	平均転送量 (単位 byte)

[Docker コンテナのネットワークパケットの監視]

項目	説明
Key	[nw_rxpck nw_txpck]: コンテナ ID
Value	平均パケット数 (単位個)

○戻り値

正常 : 0

異常 : 1

○注意事項

- ・ -k KEY_ID の指定について

平均ネットワーク I/O 量の算出には、前回取得時のカウンタ値と時刻などの情報を記録しておく必要があります。本スクリプトでは、-k KEY_ID の指定により、/tmp/KEY_ID.txt に記録します。これにより、同一サーバ上で異なるユーザの利用または異なる Docker ホストへの監視が可能になります。Hinemos のカスタム監視から使用する場合は、本値が実行単位ごとに必ず異なるように注意してください。

○実行例**[Docker コンテナのネットワーク転送量の監視]**

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceNetwork.py
-i docker_host1 -k hoge_nw --tx --rx --kb
nw_rxkb:4dfd0a12, 0.00
nw_txkb:4dfd0a12, 0.00
nw_rxkb:c7a095b8, 0.00
nw_txkb:c7a095b8, 0.00
nw_rxkb:d7ee4d6e, 0.00
nw_txkb:d7ee4d6e, 0.00
```

[Docker コンテナのネットワークパケットの監視]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerResourceNetwork.py
-i docker_host1 -k hoge_nw --tx --rx --pck
nw_rxpck:4dfd0a12, 0.00
nw_txpck:4dfd0a12, 0.00
nw_rxpck:c7a095b8, 0.00
```

```
nw_txpck:c7a095b8, 0.00
nw_rxpck:d7ee4d6e, 0.00
nw_txpck:d7ee4d6e, 0.00
```

3.6.4. Hinemos での使い方

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

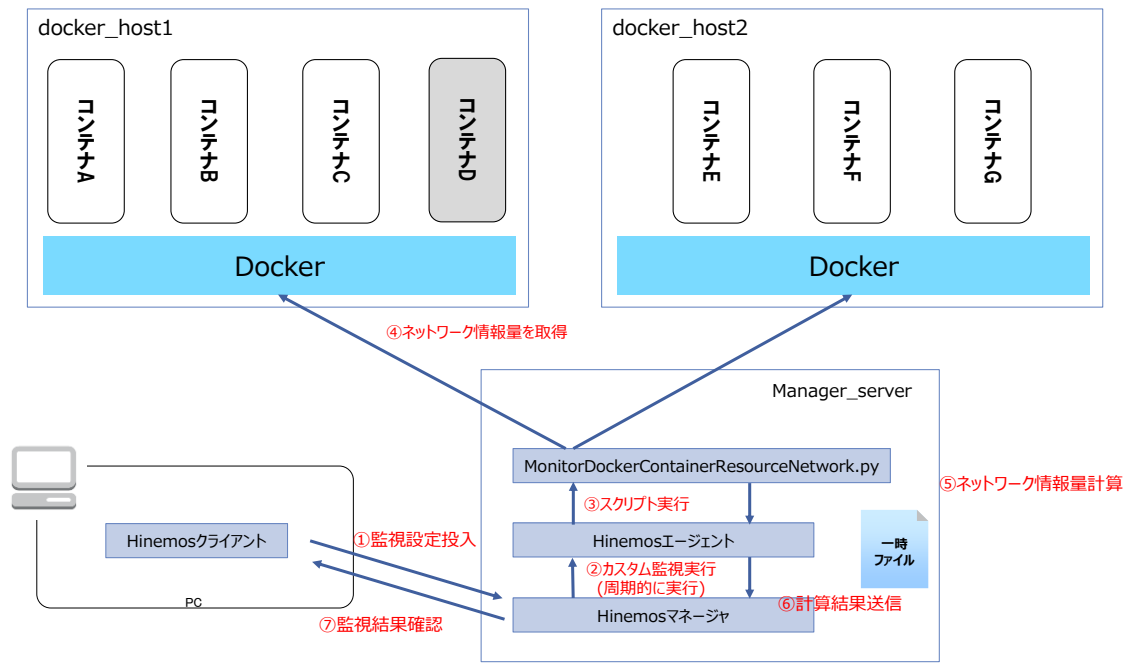


図 Docker コンテナのネットワーク情報量取得の流れ

○カスタム監視の設定例

Docker コンテナのネットワーク I/O 量の監視をするカスタム監視設定の例を以下に示します。ネットワーク I/O 量の監視では、閾値監視を行い直ちにアラートを上げるような用途は少なく、中長期向けのサイジングや障害解析などでの利用が多いため、当該内容の設定例を示します。

表 カスタム監視の設定 (CONTAINER_NETWORK)

監視項目 ID	CONTAINER_NETWORK
説明	Docker コンテナのネットワーク I/O 量の監視
スコープ	docker_host (スコープ)
間隔	5 分
指定したノード上でまとめてコマンド実行	チェック

参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerContainerResource Network.py -i #[IP_ADDRESS] -k #[NODE_NAME]_CONTAINER_NETWORK --tx --rx --kb
タイムアウト	15000
監視	未チェック
収集	チェック
収集値表示名	ネットワーク転送量
収集値単位	Byte/5 分

3.7. Docker コンテナ内のプロセス監視

3.7.1. 概要

Docker コンテナ内で起動している（パターンパッチ表現にマッチする）プロセス数を監視します。

3.7.2. 仕様・アーキテクチャ

Docker コンテナ内にマルチプロセスのアプリケーションを配置した時、Docker コンテナ内の対象アプリケーションの起動数が適切かを監視する必要があります。

監視項目	説明	頻度
Docker コンテナ内の起動プロセス数	Docker コンテナで起動しているプロセス数を監視する（起動コマンドの文字列パターンマッチングに該当するプロセスの特定も可能） （単位：個）	1 分～5 分間隔程度

Docker コンテナ内で起動しているプロセス情報は Remote API の top を使用して取得します。

Remoto API	HTTP リクエスト	HTTP レスポンス
top	/container/(id)/top	指定した Docker コンテナ内で起動しているプロセスの情報 (ps コマンドの出力結果に相当)

• Docker Remote API 実行例

```
(root) # curl http://docker_host1:2375/containers/container_B/top
{"Processes":[[{"root","22592","22287","0","Sep05","pts/2","00:00:01","httpd      -D
FOREGROUND"}, {"bin","22617","22592","0","Sep05","pts/2","00:00:12","httpd      -D
FOREGROUND"}, {"bin","22618","22592","0","Sep05","pts/2","00:00:12","httpd      -D
FOREGROUND"}, {"bin","22619","22592","0","Sep05","pts/2","00:00:12","httpd      -D
FOREGROUND"}], "Titles":["UID","PID","PPID","C","STIME","TTY","TIME","CMD"]}
```

Titles:” CMD” に起動コマンドが引数（オプション）と共に表示されます。

3.7.3. スクリプトの使い方 (CLI)

○スクリプト名

MonitorDockerContainerProcess.py

○書式

MonitorDockerContainerProcess.py

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト : 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト : 全コンテナを対象)	-
-prog PROG	対象とするプロセス名のパターン スペースがある場合は” ” で括る (default: 全てのプロセスが対象)	-
-args ARGS	プロセスに渡されたオプションのパターン (処理ないでは、上記 PROG と連結して使用する)	-

	ため、PROG に集約可)	
-h, --help	スクリプトの使用方法を表示	-

○標準出力（監視情報）

本スクリプトは、Hinemos のカスタム監視による利用を想定しています。そのため、標準出力は key, value の CSV 形式になっています。Hinemos のカスタム監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.13 カスタム監視」を参照してください。

[Docker コンテナ内の起動プロセス数]

項目	説明
Key	PROGS ARGS: コンテナ ID
Value	PROGS ARGS にマッチしたプロセス数（個）

○戻り値

正常：0

異常：1

○注意事項

特にありません。

○実行例

[Docker コンテナ内の起動プロセス数（全プロセス数）]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerProcess.py -i
docker_host1
.*:4dfd0a12b9de,4
.*:c7a095b8ec29,4
.*:d7ee4d6ee44c,4
```

[Docker コンテナ内の起動プロセス数（コマンド指定）]

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerProcess.py -i
docker_host1 -prog httpd
httpd:4dfd0a12b9de,4
httpd:c7a095b8ec29,4
httpd:d7ee4d6ee44c,4
```

3.7.4. スクリプトの使い方 (Hinemos)

本サンプルスクリプトは、Hinemos のカスタム監視機能に組み込んで使用します。

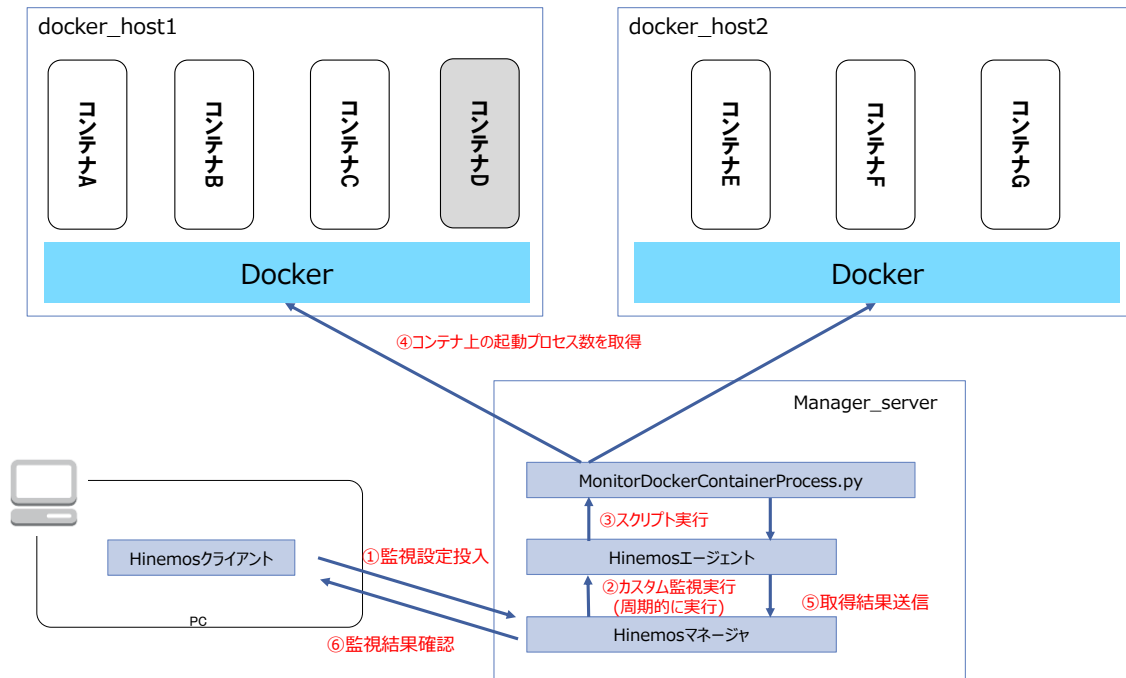


図 Docker コンテナ内の起動プロセス数 取得の流れ

○カスタム監視の設定例

Docker コンテナのプロセス数の監視をするカスタム監視設定の例を以下に示します。プロセス数の監視では、閾値監視を行い直ちにアラートを上げるようなケースと、中長期向けのサイジングや障害解析などでの利用の両方のケースが多いため、当該内容の設定例を示します。

表 カスタム監視の設定 (CONTAINER_PROCESS)

監視項目 ID	CONTAINER_PROCESS
説明	Docker コンテナのプロセス数の監視
スコープ	docker_host (スコープ)
間隔	5 分
指定したノード上でまとめてコマンド実行	チェック
参照	manager_server
実行ユーザ	エージェント起動ユーザ
コマンド	/opt/hinemos_agent/docker/bin/MonitorDockerContainerProcess.

	py -i #[IP_ADDRESS] -prog httpd
タイムアウト	15000
監視	チェック
情報の閾値	0（下限）～10（上限）
警告の閾値	10（下限）～20（上限）
通知 ID	EVENT_FOR_POLLING
アプリケーション	CONTAINER_MEM
収集	チェック
収集値表示名	httpd プロセス数
収集値単位	個

3.8. Docker コンテナの状態監視

3.8.1. 概要

Docker コンテナの状態を監視します。

3.8.2. 仕様・アーキテクチャ

Docker コンテナの状態（State）が通常変化がない（running）システムの場合、そこから状態が変化したときにアラートを上げる必要があります。

監視項目	説明	頻度
Docker コンテナの状態 (State)	Docker コンテナの状態を監視する	1 分～10 分間隔程度

Docker コンテナの状態に関する情報は下記の Docker Remote API より取得します。

Remote API の HTTP リクエスト	HTTP レスポンス
/containers/(id)/json	指定のコンテナ情報 State

• Docker Remote API 実行例

```
(root) # curl http://docker\_host1:2375/containers/container\_B/json
```

(中略)

```
"State": {
  "Running": true,
  "Paused": false,
```

```

    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 22592,
    "ExitCode": 0,
    "Error": "",
    "StartedAt": "2015-09-05T14:30:08.450307527Z",
    "FinishedAt": "0001-01-01T00:00:00Z"
  },
  (中略)

```

3.8.3. スクリプトの使い方(CLI)

○スクリプト名

MonitorDockerContainerStatus.py

○書式

MonitorDockerContainerStatus.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	○
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可 (デフォルト : 全コンテナを対象)	-
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可 (デフォルト : 全コンテナを対象)	-
-s SYSLOG_HOST:PORT	監視結果の syslog 送信先を指定	○
-h, --help	スクリプトの使用方法を表示	

○標準出力 (監視情報)

本スクリプトは、cron にて定期的に実行し、Hinemos のシステムログ監視による利用を想定しています。Hinemos のシステムログ監視の仕様については、「Hinemos ver5.0 ユーザマニュアル 7.14 システムログ監視」を参照してください。

標準出力書式

[時刻] [コンテナ ID] [コンテナ状態]

syslog 送信メッセージ書式

[コンテナ ID] [コンテナ状態]

※時刻情報は、メッセージを受け取った syslog サーバ側で設定

○戻り値

正常 : 0

異常 : 1

○注意事項

- ・ syslog メッセージの HOSTNAME 部

syslog メッセージの HOSTNAME 部はコンテナ ID となるため、Hinemos のシステムログ監視ではコンテナ ID がホスト名に設定されたノードのイベントとして検知されます。

○実行例

manager_server で事項すると、ローカルの rsyslog 経由で Hinemos マネージャに送信され、システムログ監視の対象になります。

```
(root) # /opt/hinemos_agent/docker/bin/MonitorDockerContainerStatus.py -i
docker_host1 -s 127.0.0.1:514
2015-09-06 07:15:16 90b56eb899e7 stopped
2015-09-06 07:15:16 4dfd0a12b9de running
2015-09-06 07:15:16 c7a095b8ec29 running
2015-09-06 07:15:16 d7ee4d6ee44c running
```

3.8.4. Hinemos での使い方

本サンプルスクリプトは、cron により定期的に実行し、Hinemos のシステムログ監視機能と組み合わせて使用します。

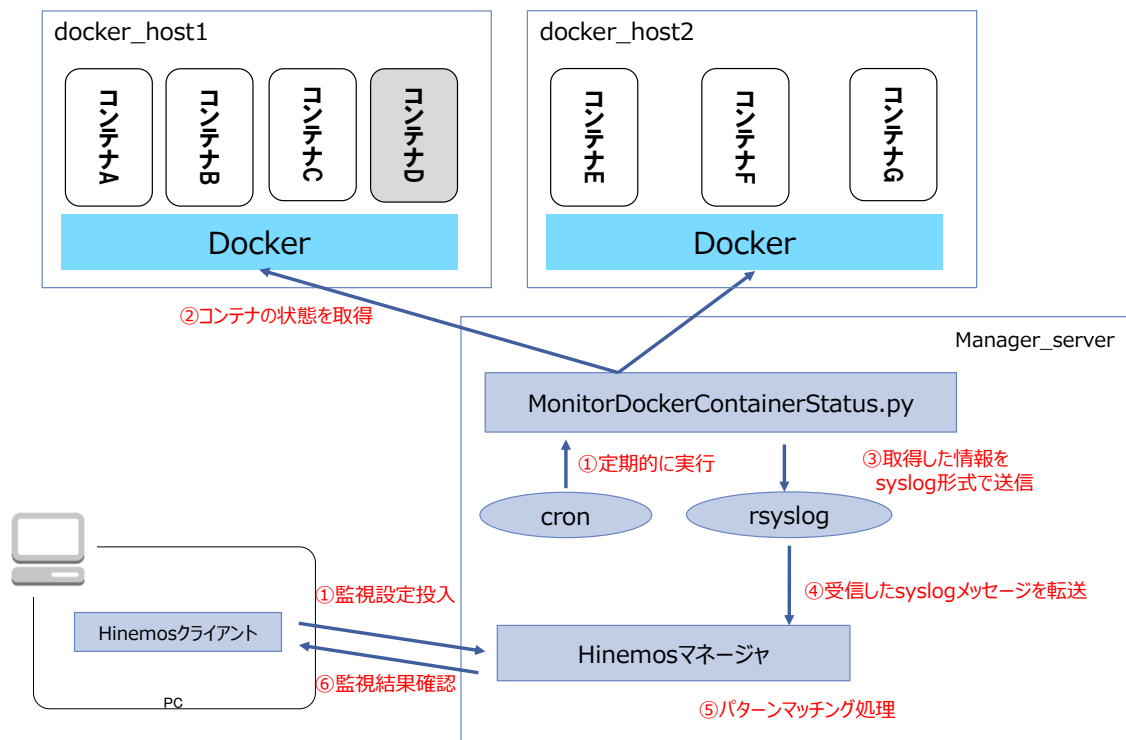


図 Docker コンテナの状態取得の流れ

○cron の設定例

manager_server 上の cron にて定期的に実行します。これにより Hinemos マネージャに syslog メッセージが定期的に送信されるようになります。

```
(root) # crontab -l
*/5 * * * * /opt/hinemos_agent/docker/bin/MonitorDockerContainerStatus.py -i
docker_host1 -s 127.0.0.1:514 > /dev/null
*/5 * * * * /opt/hinemos_agent/docker/bin/MonitorDockerContainerStatus.py -i
docker_host2 -s 127.0.0.1:514 > /dev/null
```

○システムログ監視の設定例

cron 契機で定期的に送信されてくる syslog を監視するシステムログ監視機能の設定例を示します。常時起動が正常の状態のシステムでは、running のみを正常と、それ以外を警告または危険と設定します。

検知されたイベントは、Docker コンテナに対応するノードのイベントとして通知されます。

表 システムログ監視の設定 (CONTAINER_STATUS)

監視項目 ID	CONTAINER_STATUS
説明	Docker コンテナの状態監視
スコープ	docker_container

表 文字列パターンマッチングの設定

順序	条件	パターンマッチ表現	重要度	有効/無効
1	条件に一致したら処理する	.*running.*	情報	有効
2	条件に一致したら処理する	.*paused.*	危険	有効
3	条件に一致したら処理する	.*restarting.*	警告	有効
4	条件に一致したら処理する	.*exited.*	危険	有効

4. ジョブ・操作編

本章では、Docker コンテナに対するジョブ管理とコンテナ操作に関する方式について記載します。

○背景と目的

Docker コンテナで動作するアプリケーションには様々あり、Web アプリケーションやバッチ処理のアプリケーションなどがあります。また Docker コンテナを使ったエンタープライズシステムを考えると、コンテナ化が可能なアプリケーション部分と、それ以外のオンプレ・仮想化・クラウド環境との組み合わせたハイブリッド環境での利用が考えられます。

その場合は、既存のサーバ機器/仮想マシンと同様に(連携して)、業務カレンダーに合わせて Docker コンテナを対象にジョブ監視や起動・停止などのコントロールが行える必要があります。

○実現方法

Hinemos では、ジョブ機能の範囲内で、Docker コンテナのジョブ管理とコンテナ操作を実現するサンプルスクリプトを用意しました。

[ジョブ管理]

・Docker コンテナ内のコマンド実行

指定の Docker コンテナ上の任意のコマンドを実行し、その戻り値と標準出力・標準エラー出力を管理します。

・ファイルダウンロード

指定の Docker コンテナ上の任意のファイルを、ローカルにダウンロードします。

[コンテナ操作]

・Docker コンテナの操作（起動）

指定の Docker コンテナを起動します。

・Docker コンテナの操作（停止）

指定の Docker コンテナを停止します。

・Docker コンテナの操作（再起動）

指定の Docker コンテナを再起動します。

- **Docker コンテナの操作（削除）**

指定の Docker コンテナを削除します。

[使い方概要]

本機能は、本ドキュメントの他の機能と同様に、Docker ホストとは異なるリモートサーバ（本ドキュメントでは `manager_server`）から実行できるスクリプトとして用意しました。このスクリプトは、Hinemos のジョブ機能により実行するコマンドをしての利用を想定しています。

これにより、監視機能にて異常を検知した際に実行するジョブフローや、スケジュール契機で実行するジョブフローの中の 1 要素として組み込むことができます。

実行するコンテナ ID またはコンテナ名にはワイルドカード(*)が指定できます。これより、リモートサーバから指定の名前を含むコンテナに対して一括して操作を実行することができます。

○本章の構成

Docker の監視に関する 6 機能を順次説明します。

- Docker コンテナ内のコマンド実行
- ファイルダウンロード
- Docker コンテナの操作（起動）
- Docker コンテナの操作（停止）
- Docker コンテナの操作（再起動）
- Docker コンテナの操作（削除）

○注意事項

- 本章の機能範囲に、Docker コンテナの作成は含まれておりません。

Docker コンテナの作成には、非常に多くのパラメタ(引数)が必要であり、またデプロイ管理の範囲に含まれるものとして、対象外と設定しています。

4. 1. Docker コンテナ内のコマンド実行

4. 1. 1. 概要

指定の Docker コンテナ上の任意のコマンドを実行し、そのコマンド実行の戻り値と標準

出力・標準エラー出力を返却します。

4.1.2. 仕様・アーキテクチャ

指定の Docker コンテナ上の任意のコマンドを実行します。コンテナ ID またはコンテナ名の指定では前方一致で複数の Docker コンテナを一括して指定します。前方一致の指定は、最後にワイルドカード(*)を指定します。

○Hinemos エージェント

通常、サーバ上でジョブのコマンドを事項する場合は、Hinemos エージェントをインストールが必要です。しかし、コンテナへ Hinemos エージェントをインストールすることなく、Docker Remote API を使用してリモートからコマンドを実行します。

これは、コマンド実行のためにコンテナ毎に Hinemos エージェントを導入することは、リソース使用効率の観点から好ましくないためです。

○標準出力・標準エラー出力について

実行した複数の Docker コンテナを一括して指定を、サンプルスクリプトの標準出力・標準エラー出力とします。複数の Docker コンテナを一括して指定した場合は、順次コマンドを実行し、複数の Docker コンテナ上の実行結果も全て出力します。

○戻り値について

実行したコマンドの戻り値をサンプルスクリプトの戻り値にします。複数の Docker コンテナを一括して指定した場合は、（暫定として）一番最初に実行したコマンドの戻り値を返します。

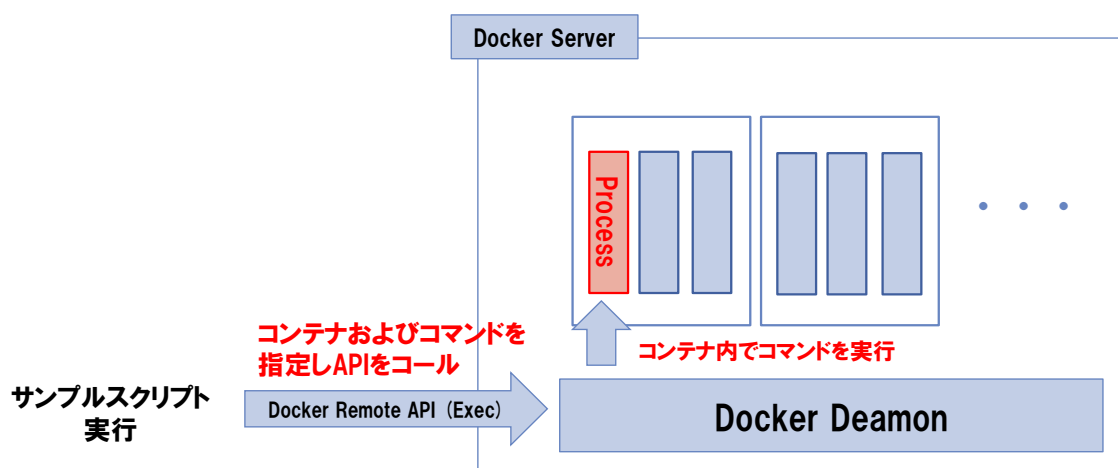


図 コンテナ内でのコマンド実行

4.1.3. スクリプトの使い方 (CLI)

○スクリプト名

JobDockerContainerCommand.py

○書式

JobDockerContainerCommand.py [options] COMMAND

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.01)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)(*2)
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)(*2)
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

(*2) 本バージョンでは、複数コンテナ指定は推奨しません。

○戻り値

- ・実行したコマンドの戻り値
- ・ただし、複数の Docker コンテナを一括して指定した場合は、一番最初に実行したコマンドの戻り値

○注意事項

非常に応答が長いコマンドを実行すると、Read Timeout が発生する可能性があります。

○実行例

docker_host1 上のコンテナ名 container_A に find コマンドを実行すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/JobDockerContainerCommand.py -i
docker_host1 -n container_A "find /root"
/root
/root/.bashrc
/root/.gnupg
/root/.gnupg/pubring.gpg~
/root/.gnupg/secring.gpg
/root/.gnupg/trustdb.gpg
/root/.gnupg/pubring.gpg
/root/.gnupg/gpg.conf
/root/.profile
```

4.1.4. スクリプトの使い方(Hinemos)

4章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

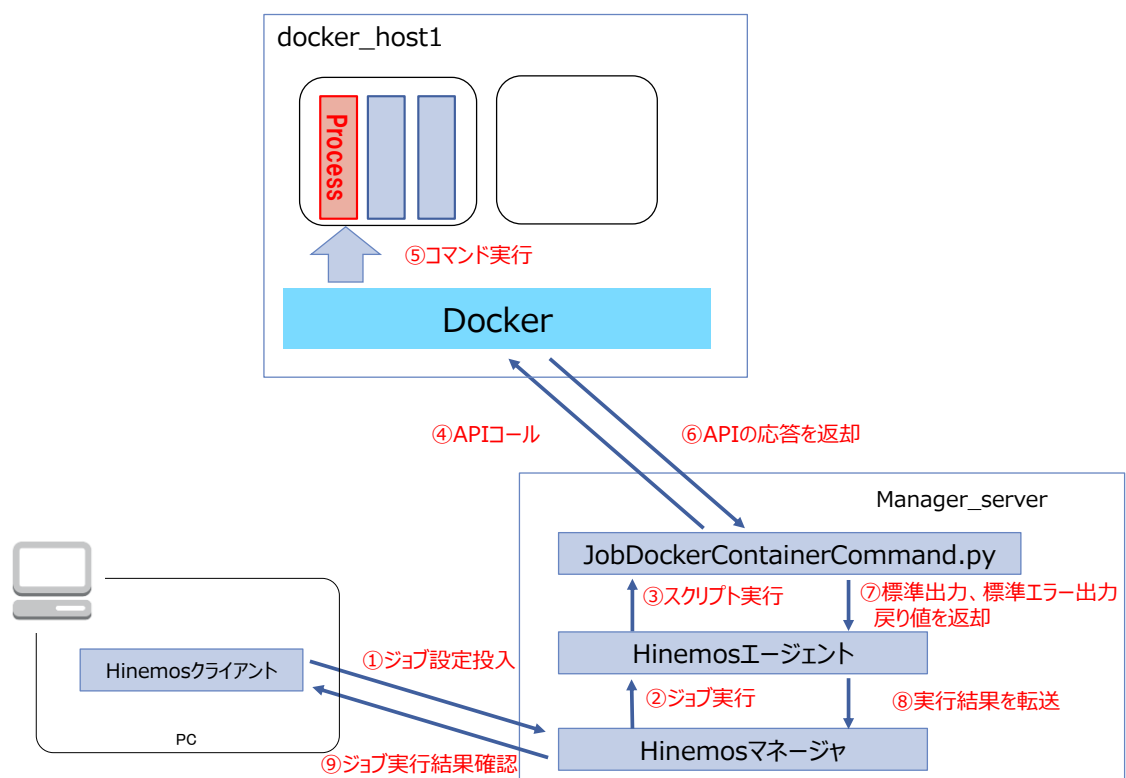


図 Docker コンテナ内でコマンドを実行するまでの流れ

○ジョブの設定例

docker_host1 上の停止中のコンテナ” container_A” 上で find コマンドを起動するジョブ定義の例を以下に示します。

表 コンテナ内でコマンド実行ジョブの設定例

ジョブ ID	COMMAND_CONTAINER_D
ジョブ名	コマンド実行ジョブ
説明	Docker コンテナ内で find コマンドを実行します。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/JobDockerContainerCommand.py -i docker_host1 -n container_A "find /root/"

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

The screenshot shows the Hinemos Job History interface. The top section displays a list of jobs with columns: マネージャ, 実行状態, 終了状態, 終了値, セッションID, ジョブID, ジョブ名, ジョブユニットID, 種別, ファシリティID, スコープ, オーナーロールID, 開始予定日時. Below this, the 'ジョブ履歴[ノード詳細]' view is selected, showing details for a specific job with columns: 実行状態, 終了状態, 終了値, ジョブID, ジョブ名, ジョブユニットID, 種別, ファシリティID, スコープ, 時刻, 開始・再実行日時, 終了・中断日時, 実行時間, メッセージ. The message column shows the command: 'stdoutr=/root/ /root/anac'.

図 ジョブ履歴パースペクティブ

ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。



図 メッセージダイアログ

4.2. ファイルダウンロード

4.2.1. 概要

指定の Docker コンテナ上の任意のファイルを、ローカルにダウンロードします。

4.2.2. 仕様・アーキテクチャ

指定の Docker コンテナ上の任意のファイルまたはディレクトリを、指定のローカルディレクトリにダウンロードします。ユーザ/グループ/permission はダウンロード元の情報を保持されます。また、ダウンロードファイルはファイル/ディレクトリに関わらず、tar 形式にまとめて出力されます。

4.2.3. スクリプトの使い方 (CLI)

○スクリプト名

FileDockerContainerDownload.py

○書式

.FileDockerContainerDownload.py [options] SOURCE DESTINATION

• SOURCE

コンテナからダウンロードするファイル/ディレクトリパスを指定します。

• DESTINATION

ダウンロード先のローカルディレクトリパスを指定します。

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)(*2))
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)(*2))
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

(*2)本バージョンでは、複数コンテナ指定は推奨しません。

○戻り値

正常 : 0

異常 : 1

○注意事項

特にありません。

○実行例

docker_host1 上のコンテナ container_A の /root ディレクトリをローカルディレクトリにダウンロードすると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/FileDockerContainerDownload.py -i
docker_host1 -n container_A /root/ /tmp/20150905/
/tmp/20150905//ames_root--.tar
drwx----- 0/0          0 2015-08-24 18:28:36 root/
-rw-r--r-- 0/0          570 2010-01-31 11:52:26 root/.bashrc
drwx----- 0/0          0 2015-08-24 18:28:36 root/.gnupg/
```

```
-rw----- 0/0          9188 2015-08-24 18:27:19 root/.gnupg/gpg.conf
-rw----- 0/0        67755 2015-08-24 18:27:27 root/.gnupg/pubring.gpg
-rw----- 0/0        67755 2015-08-24 18:27:27 root/.gnupg/pubring.gpg~
-rw----- 0/0           0 2015-08-24 18:27:19 root/.gnupg/secring.gpg
-rw----- 0/0        1200 2015-08-24 18:27:27 root/.gnupg/trustdb.gpg
-rw-r--r-- 0/0         140 2007-11-19 17:57:23 root/.profile
```

```
(root) # ls -l /tmp/20150905/
```

```
total 152
```

```
-rw-r--r-- 1 root root 154112 Sep  5 10:29 ames_root--.tar
```

```
(root) # tar -xvf /tmp/20150905/ames_root--.tar
```

```
root/
```

```
root/.bashrc
```

```
root/.gnupg/
```

```
root/.gnupg/gpg.conf
```

```
root/.gnupg/pubring.gpg
```

```
root/.gnupg/pubring.gpg~
```

```
root/.gnupg/secring.gpg
```

```
root/.gnupg/trustdb.gpg
```

```
root/.profile
```

4.2.4. スクリプトの使い方(Hinemos)

4 章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

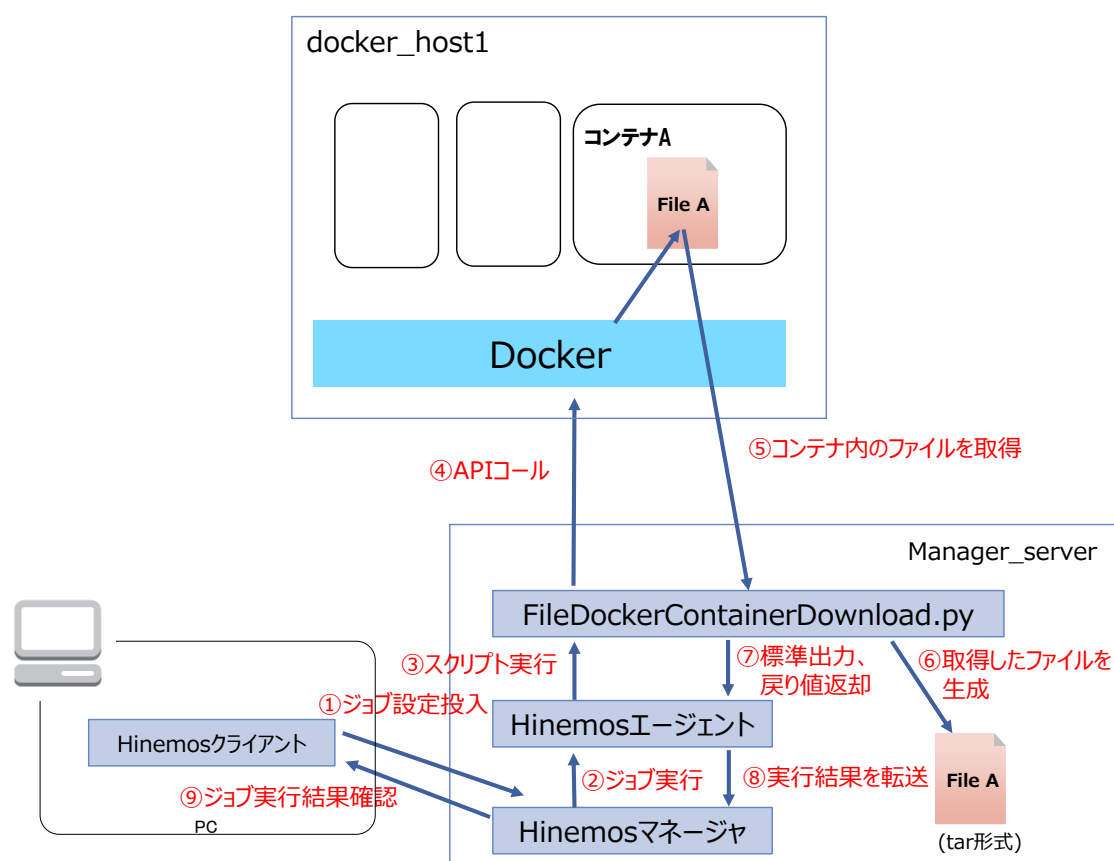


図 Docker コンテナからファイルをダウンロードするまでの流れ

○ジョブの設定例

docker_host1 上のコンテナ” container_A” からファイルをダウンロードするジョブ定義の例を以下に示します。

表 コンテナからのファイルダウンロードジョブの設定例

ジョブ ID	FILE_DOWNLOAD_FROM_CONTAINER_D
ジョブ名	ファイルダウンロードジョブ
説明	コンテナ内からファイルをダウンロードします。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/FileDockerContainerDownload.py -i docker_host1 -n container_D /root/ /tmp/20150905/

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

4.3. Docker コンテナの操作（起動）

4.3.1. 概要

指定の Docker コンテナを起動します。

4.3.2. 仕様・アーキテクチャ

停止している(コンテナの状態が `exited`)の Docker コンテナを起動します。コンテナ ID またはコンテナ名の指定では前方一致で複数の Docker コンテナを一括して指定します。前方一致の指定は、最後にワイルドカード(*)を指定します。コンテナの状態が `exited` 以外のものについては、何もしません。

○処理対象のコンテナの状態

Docker コンテナの状態	処理内容
<code>exited</code>	コンテナを起動する
<code>stopped</code>	何もしない
<code>paused</code>	何もしない
<code>running</code>	何もしない

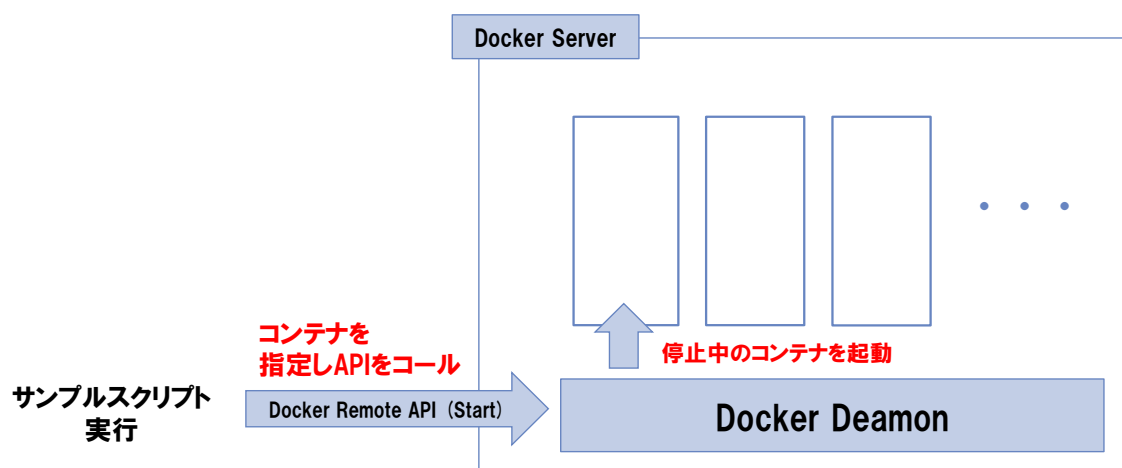


図 コンテナの起動

4.3.3. スクリプトの使い方 (CLI)

○スクリプト名

JobDockerContainerStart.py

○書式

JobDockerContainerStart.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.01)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

○戻り値

正常 : 0

1 つ以上のコンテナで処理失敗 : 9

異常 : 1

○注意事項

特にありません。

○実行例

docker_host1 上の先頭から container という文字のコンテナを一括して起動すると、以下のような標準出力が出力されます。

```
(root)      # /opt/hinemos_agent/docker/bin/JobDockerContainerStart.py -i
docker_host1 -n container*
0e0d8cdf3a57 container_C Success
1eae26f41927 container_B Success
```

b66e046560f3 container_A Success

4.3.4. スクリプトの使い方(Hinemos)

4 章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

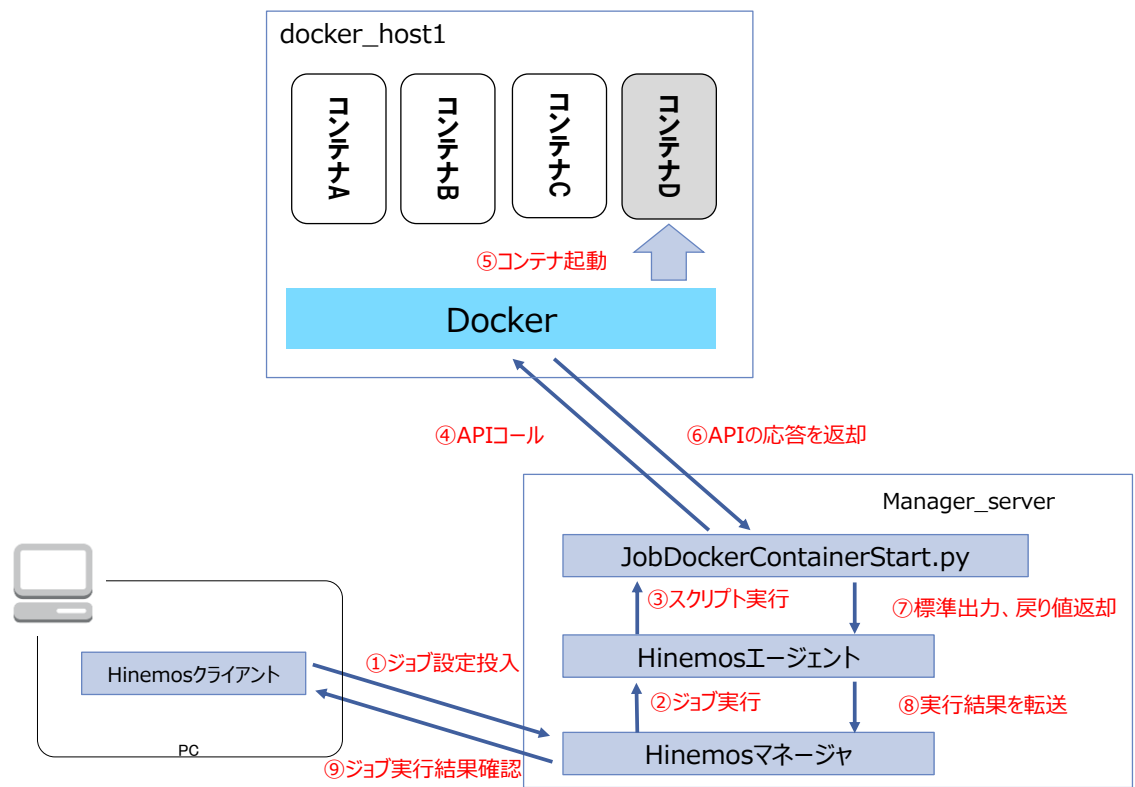


図 Docker コンテナを起動するまでの流れ

○ジョブの設定例

docker_host1 上の停止中のコンテナ” container_D” を起動させるジョブ定義の例を以下に示します。

表 コンテナ起動ジョブの設定例

ジョブ ID	START_CONTEINER_D
ジョブ名	コンテナ起動ジョブ
説明	停止中の Docker コンテナを起動します。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/JobDockerContainerStart.py -i docker_host1 -n container_D

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

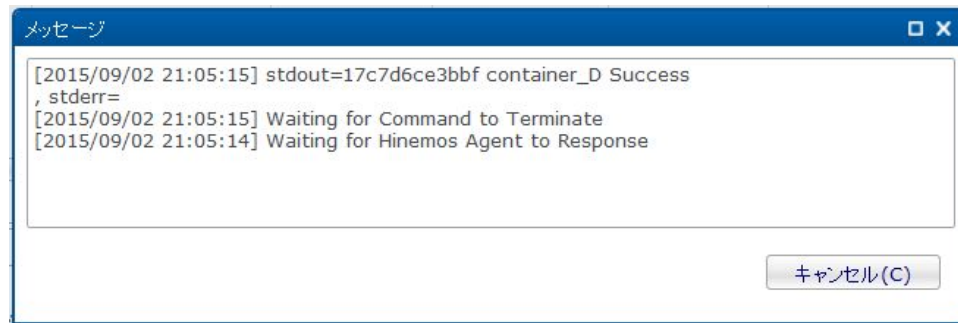


図 コンテナ起動ジョブの実行結果確認

4. 4. Docker コンテナの操作（停止）

4. 4. 1. 概要

指定の Docker コンテナを停止します。

4. 4. 2. 仕様・アーキテクチャ

起動している(コンテナの状態が running)の Docker コンテナを停止します。コンテナ ID またはコンテナ名の指定では前方一致で複数の Docker コンテナを一括して指定します。前方一致の指定は、最後にワイルドカード(*)を指定します。コンテナの状態が running 以外のものについては、何もしません。

○処理対象のコンテナの状態

Docker コンテナの状態	動作
running	コンテナを停止する
stopped	何もしない
paused	何もしない
restarting	何もしない

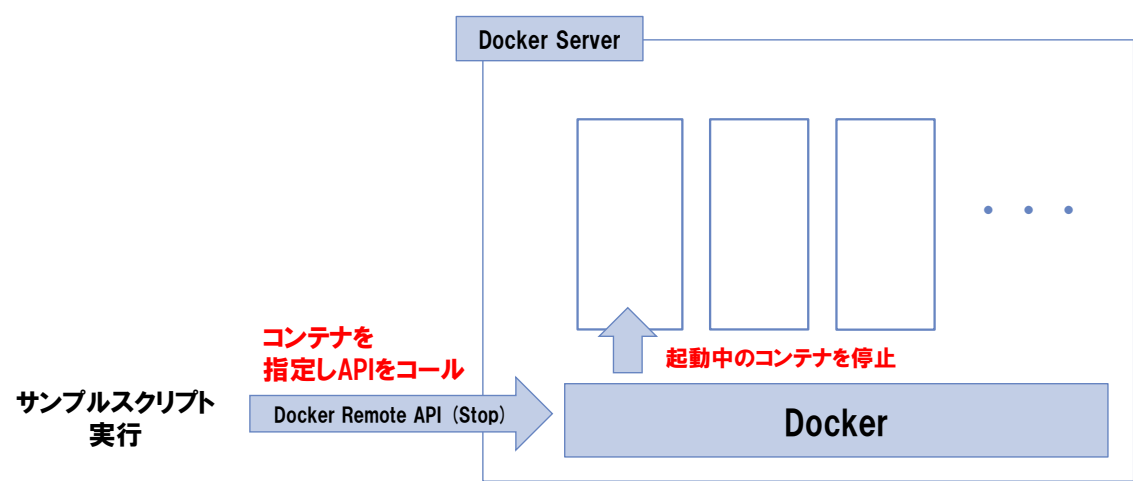


図 コンテナの停止

4. 4. 3. スクリプトの使い方 (CLI)

○スクリプト名

JobDockerContainerStop. py

○書式

JobDockerContainerStop. py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127. 0. 01)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)
-t WAIT	停止処理のタイムアウト(秒)	-
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

○戻り値

正常 : 0

1 つ以上のコンテナで処理失敗 : 9

異常 : 1

○注意事項

特にありません。

○実行例

docker_host1 上の先頭から container という文字のコンテナを一括して停止すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/JobDockerContainerStop.py -i docker_host1  
-n container*  
0e0d8cdf3a57 container_C Success  
1eae26f41927 container_B Success  
b66e046560f3 container_A Success
```

4.4.4. スクリプトの使い方 (Hinemos)

4 章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

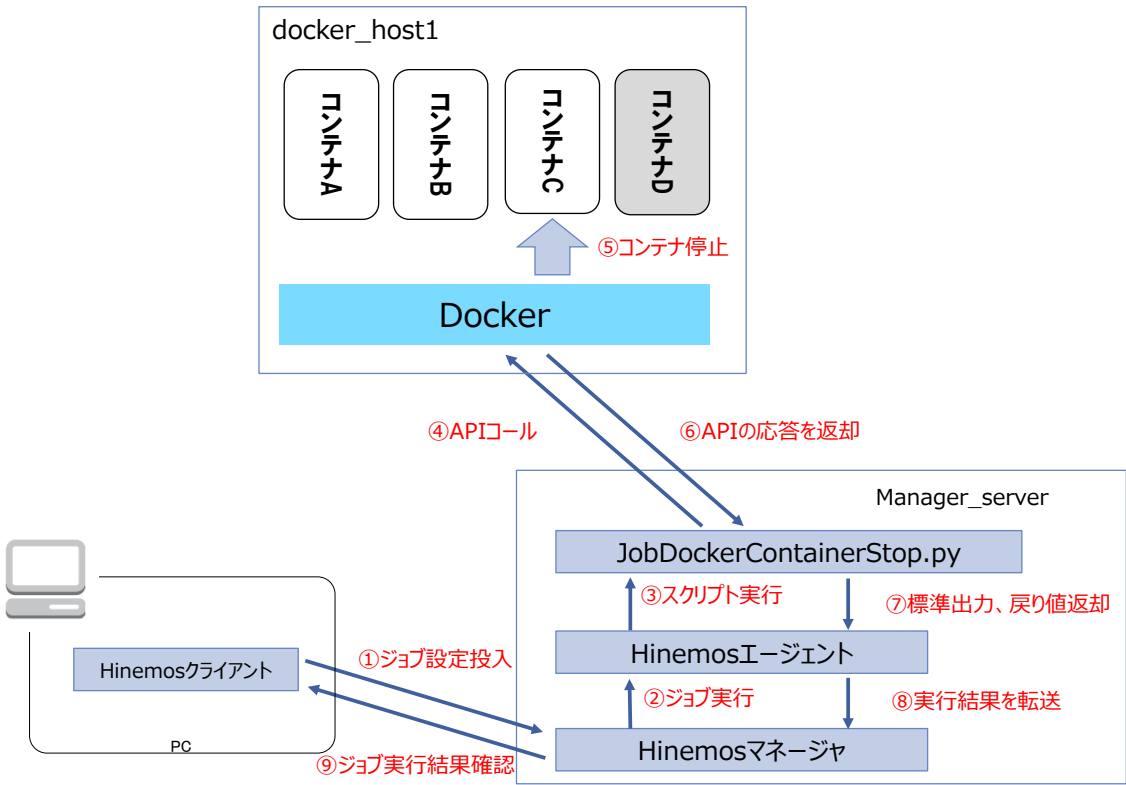


図 Docker コンテナを停止するまでの流れ

○ジョブの設定例

docker_host1 上の起動中のコンテナ” container_D” を停止させるジョブ定義の例を以下に示します。

表 コンテナ停止ジョブの設定例

ジョブ ID	STOP_CONTEINER_D
ジョブ名	コンテナ起動ジョブ
説明	停止中の Docker コンテナを起動します。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/JobDockerContainerStop.py -i docker_host1 -n container_D

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

4. 5. Docker コンテナの操作（再起動）

4.5.1. 概要

指定の Docker コンテナを再起動します。

4.5.2. 仕様・アーキテクチャ

起動している(コンテナの状態が running)の Docker コンテナを再起動します。コンテナ ID またはコンテナ名の指定では前方一致で複数の Docker コンテナを一括して指定します。前方一致の指定は、最後にワイルドカード(*)を指定します。コンテナの状態が running 以外のものについては、何もしません。

○処理対象のコンテナの状態

Docker コンテナの状態	動作
running	コンテナを再起動する
stopped	何もしない
paused	何もしない
restarting	何もしない

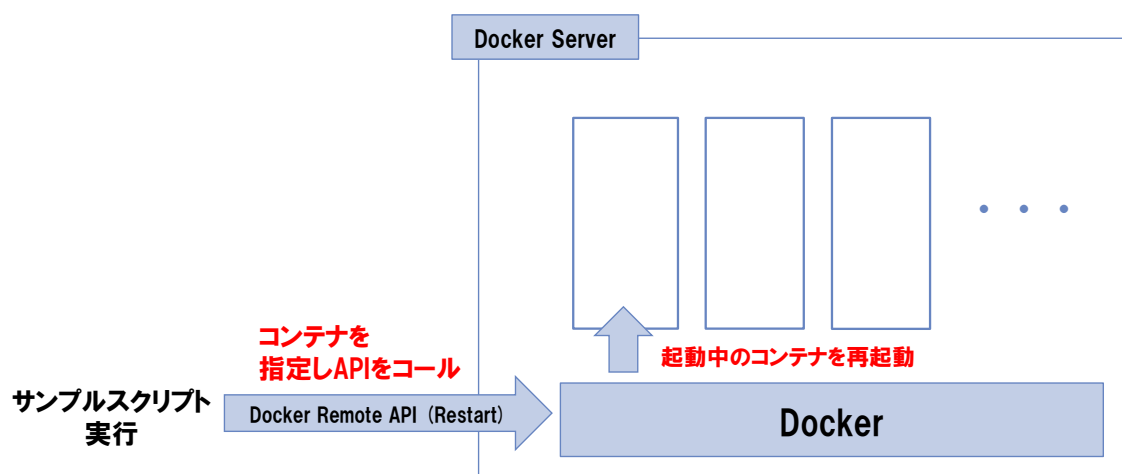


図 コンテナの再起動

4.5.3. スクリプトの使い方(CLI)

○スクリプト名

JobDockerContainerRestart.py

○書式

JobDockerContainerRestart.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)
-t WAIT	停止処理のタイムアウト(秒)	-
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

○戻り値

正常 : 0

1 つ以上のコンテナで処理失敗 : 9

異常 : 1

○注意事項

特にありません。

○実行例

docker_host1 上の先頭から container という文字のコンテナを一括して再起動すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/JobDockerContainerRestart.py -i
docker_host1 -n container*
0e0d8cdf3a57 container_C Success
1eae26f41927 container_B Success
b66e046560f3 container_A Success
```

4.5.4. スクリプトの使い方(Hinemos)

4 章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

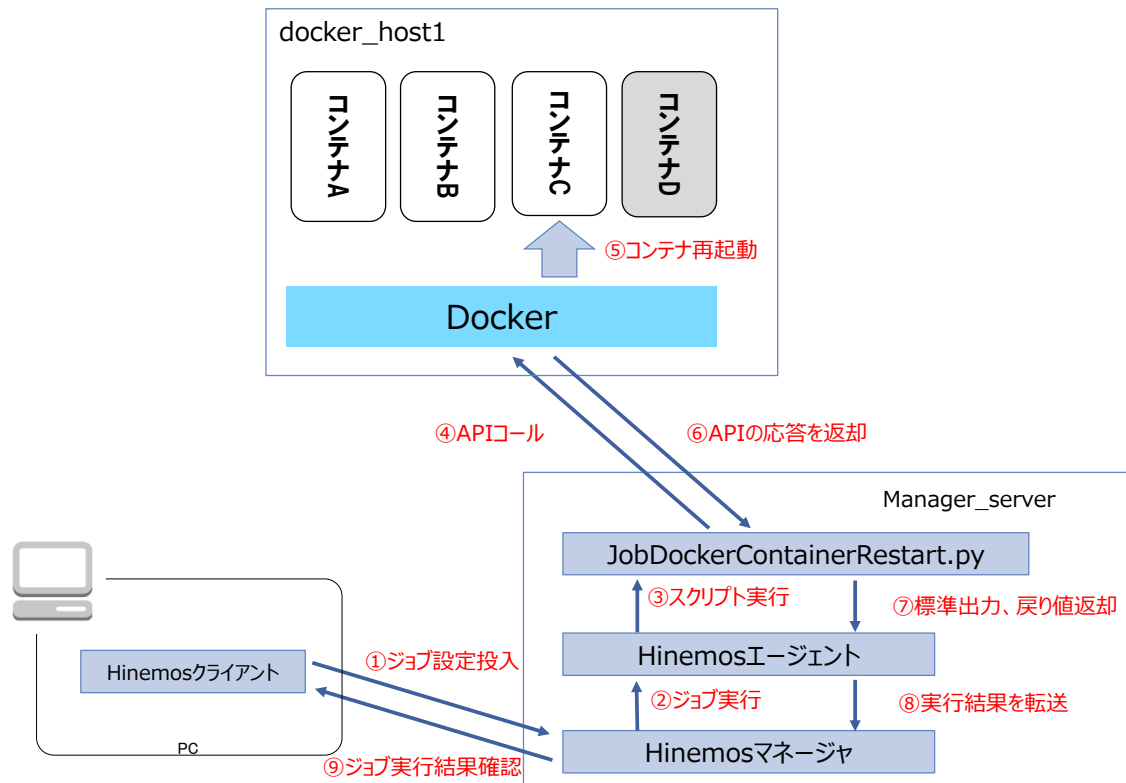


図 Docker コンテナを再起動するまでの流れ

○ジョブの設定例

docker_host1 上の起動中のコンテナ” container_D” を再起動させるジョブ定義の例を以下に示します。

表 コンテナ再起動ジョブの設定例

ジョブ ID	RESTART_CONTAINER_D
ジョブ名	コンテナ再起動ジョブ
説明	停止中の Docker コンテナを起動します。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/JobDockerContainerRestart.py -i docker_host1 -n container_D

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

4.6. Docker コンテナの操作（削除）

4.6.1. 概要

指定の Docker コンテナを削除します。

4.6.2. 仕様・アーキテクチャ

指定の Docker コンテナを起動します。コンテナ ID またはコンテナ名の指定では前方一致で複数の Docker コンテナを一括して指定します。前方一致の指定は、最後にワイルドカード(*)を指定します。デフォルトでは、コンテナの状態が `exited` のもののみ削除し、`exited` でなければエラーが出ます。コンテナの状態が `running` の場合でも強制的に削除するには、`-f` オプションを指定するとエラーが出力されることがなく削除されます。

○処理対象のコンテナの状態

Docker コンテナの状態	処理内容
<code>exited</code>	コンテナを削除する
<code>stopped</code>	何もしない(*1)
<code>paused</code>	何もしない(*1)
<code>running</code>	何もしない(*1)

(*1)-f オプションを指定することで、強制的に削除することができます。

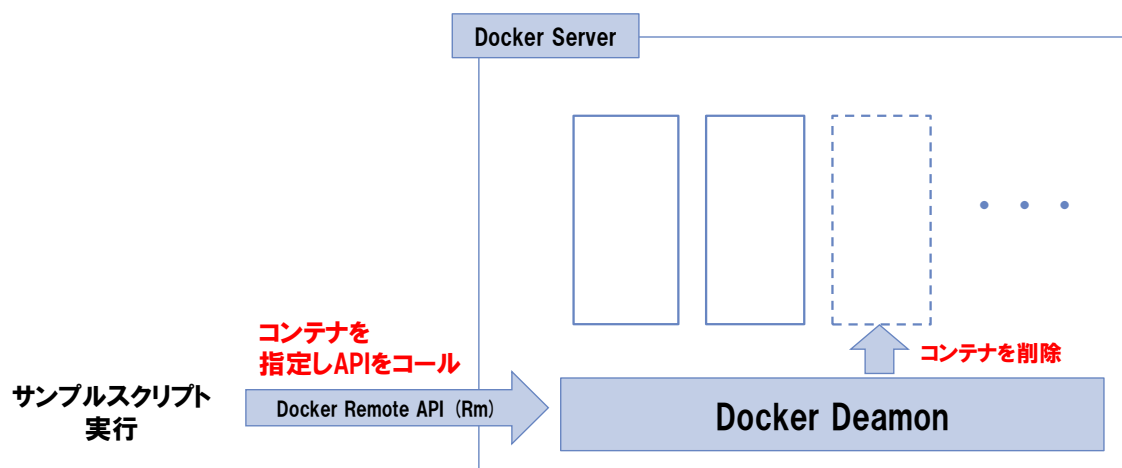


図 コンテナの削除

4.6.3. スクリプトの使い方(CLI)

○スクリプト名

JobDockerContainerRm.py

○書式

JobDockerContainerRm.py [options]

オプション	動作	必須
-i IP_ADDRESS	Docker Remote API の接続先 IP アドレス (デフォルト : 127.0.0.1)	-
-p DOCKER_PORT	Docker Remote API の接続先ポート (デフォルト : 2375)	-
-c CONTAINER_ID	操作対象のコンテナ ID 前方一致で指定する場合は、末尾に*を指定 -n と同時指定不可	○(*1)
-n CONTAINER_NAME	操作対象のコンテナ名 前方一致で指定する場合は、末尾に*を指定 -c と同時指定不可	○(*1)
-f,	全ての状態のコンテナを強制削除	-
--force=false	停止しているコンテナのみ削除	-
-l	リンク先のコンテナを削除	-
--link=false	リンク先を削除しない	-
-v	コンテナに関連付けられているボリュームを削除	-
--volumes=false	コンテナに関連するボリュームを削除しない	-
-h, --help	スクリプトの使用方法を表示	-

(*1)-c オプションまたは-n オプションの指定が必須。

○戻り値

正常 : 0

1 つ以上のコンテナで処理失敗 : 9

異常 : 1

○注意事項

特にありません。

○実行例

docker_host1 上の先頭から container という文字のコンテナを一括して強制削除すると、以下のような標準出力が出力されます。

```
(root) # /opt/hinemos_agent/docker/bin/JobDockerContainerRm.py -f -i
docker_host1 -n container*
e065ef8063d3 container_C Success
6f397ec1c4fa container_B Success
965e5c081a75 container_A Success
```

4.6.4. スクリプトの使い方(Hinemos)

4 章の[使い方概要]に記載しました通り、本機能はジョブ機能で実行するコマンドとして使用します。

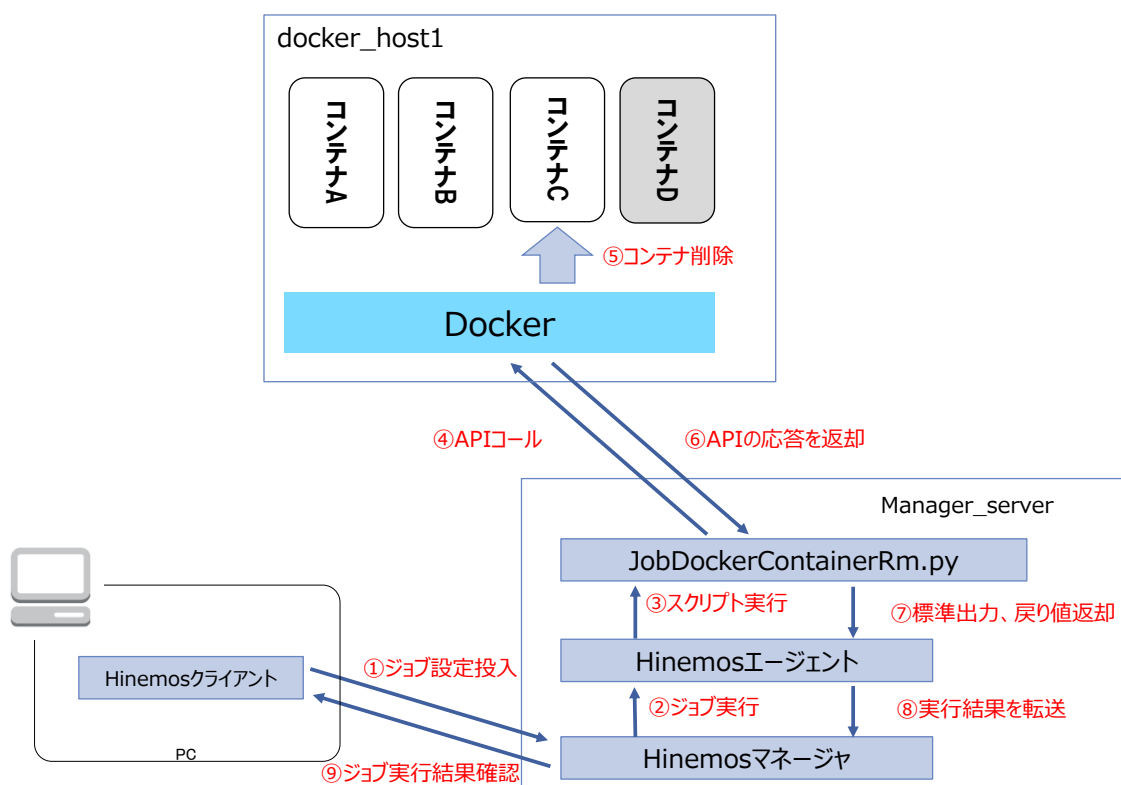


図 Docker コンテナを削除するまでの流れ

○ジョブの設定例

docker_host1 上の起動中のコンテナ” container_D” を削除するジョブ定義の例を以下に示します。

表 コンテナ削除ジョブの設定例

ジョブ ID	RM_CONTAINER_D
ジョブ名	コンテナ削除ジョブ
説明	停止中の Docker コンテナを削除します。
スコープ	manager_server
起動コマンド	/opt/hinemos_agent/docker/bin/JobDockerContainerRm.py -i docker_host1 -n container_D

ジョブを実行すると、ジョブ履歴[ノード詳細]ビューの「メッセージ」欄より、コマンドの実行結果(標準出力、標準エラー出力)を確認することができます。

5. TIPS 集

5.1. nsenter を使ったコンテナ内のプロセス起動時の注意

nsenter コマンドを使うと Docker コンテナ内に入ることができ、Docker コンテナ内でコマンドを実行することができます。しかし、Docker コンテナ単位のリソース管理の観点では、nsenter の利用は推奨されません。

nsenter で実行したコマンドは、Docker コンテナとは別のプロセスグループのプロセスとして扱われます。そのため Docker の Stats API 経由で取得するコンテナ単位のリソース地には、この nsenter で実行したコマンドのリソースは含まれません (systemd-cgls コマンドを使用して確認できます)。本ドキュメントでは、例えば次の機能で大きく影響がでます。

- ・ コンテナ別 CPU 使用率の監視 (3.3 節. MonitorDockerContainerResourceCpu. py)
- ・ コンテナ内プロセス監視 (3.7 節. MonitorDockerContainerProcess. py)

よって、Docker コンテナ内でコマンドを実行し、そのリソースは当該コンテナのものとするためには、Remote API の Exec を使用する方式 (4.1 節) を推奨します。

5.2. Docker コンテナの Status と State

Docker には Status と State の 2 つの「状態」を表現する方法があります。

○Docker コンテナの State

docker ps コマンドのオプションで -f(--filter) で指定するもので、次 4 種類があります。

- running
- restarting
- exited
- paused

○Docker コンテナの Status

`docker ps` コマンドで表示される、何分起動している、等が表示される列です。

本ドキュメントでは、Docker コンテナの状態は全て `state` で統一しており、次のような変換ロジックで Status to State の定義を設定しています。

```
def conv_docker_status(status):  
  
    # 起動エラーによりステータスが空欄となることがある  
  
    if status == "": return "exited"  
  
    w = status.split(" ")  
  
    if w[0] == "Exited": return "exited"  
  
    if w[0] == "Restarting": return "restarting"  
  
    if w[0] == "Up":  
        if w[-1] == "(Paused)": return "paused"  
        return "running"  
  
    raise LocalException("変換できないステータス:%s" % status)
```

6. おわりに

Docker はまだ発展途上ですが、非常に有用な技術です。今までのインフラエンジニアではないレイヤのエンジニアがインフラ部分を簡易に活用できることで、アプリケーションとインフラをセットで開発・構築・運用管理を行ったり、軽量である特徴を活かして並列分散処理基盤や簡易テスト環境利用など、幅広い分野での利用が期待されます。

Docker の利用は、クラウドサービスベンダによる SaaS 型提供と、Docker 及びサードベンダが提供をしているツール群を利用したクラウド外での利用、の大きく二極化になっています。

SaaS 型提供の場合は、デプロイから運用後までのサービスを抱え込み戦略という形でクラウドサービス側が一式を用意していますが、利用用途は Web アプリケーションに特化したような形態です。代わって、クラウド外での利用を考えるとユーザが各自でツール類を活用し、導入を進めることになりましたが、いまだエンタープライズ利用を意識したシステム導入後の「運用管理」に意識を向けた製品はまだ少ないです。

本ドキュメントが、この課題を解決し、Docker の導入のお役に立てば幸いです。

Hinemos ポータルサイト お問い合わせフォーム

<https://www.hinemos.info/contact>