Kompira Documentation

リリース ***1.5.5.post8***

Kompira development team

2020 年 **06** 月 **22** 日

目次

第 1 章

管理ガイド

著者　Kompira 開発チーム

1.1 はじめに

このマニュアルでは、Kompira を管理するための情報について記述します。

インストール、アップデート、Kompira 全体の設定、ログなどについて知りたい場合には本マニュアルを参照してください。

本マニュアルでは、Linux 上でのコマンド実行プロンプトとして一般ユーザーの場合は $、ルート権限ユーザーの場合は # を表記します。

```
$ echo '一般ユーザーでコマンド実行'
# echo 'ルート権限ユーザーでコマンド実行'
```

1.2 Kompira パッケージ管理

Kompira 関連パッケージのインストールとアップデートについて説明します。

参考:

本節では Kompira パッケージを単一のサーバ上で動作させる場合についてのみ説明をしています。Kompira を冗長構成で動作させたい場合は 冗長構成管理 を参照してください。

1.2.1 インストールパッケージの種類

Kompira には以下に示す種類のパッケージが存在しています。

パッケージ名	説明
Kompira パッケージ	Kompira の本体にあたるパッケージです。Kompira のコア機能群、ジョブマネージャ、イベント送信スクリプトを含みます。
ジョブマネージャパッケージ	ジョブマネージャとイベント送信スクリプトを含むパッケージです。
イベント送信パッケージ	イベント送信スクリプトを含むパッケージです。

Kompira を初めてお使いになる場合は、まず Kompira パッケージのインストールから始めてみましょう。

ジョブマネージャパッケージは Kompira パッケージをインストールしたサーバ以外にもジョブマネージャプロセスを起動させたい場合に使用します。

イベント送信パッケージは別サーバから Kompira に対してイベントを送信させたい場合に使用します。イベント送信を利用したシステム間の連携については 他システムとの連携 を参照してください。

1.2.2 インストールスクリプト

install.sh を用いることで、Kompira の各種パッケージのインストールを行うことができます。

```
install.sh [options]
```

インストール処理には Kompira で使用するミドルウェアのインストール、データベースの構築、プロセスの自動起動設定が含まれます。

install.sh はコマンドの成功・失敗に関わらず *install.<プロセス番号>.log* という名称のログファイルを作成します。

注釈: Red Hat にインストールする場合は、事前にサブスクリプションを登録しておく必要があります。

制限

install.sh は RHEL/CentOS でのインストールのみサポートしています。

install.sh では外部から Kompira で使用する各種ミドルウェアをダウンロードします。インターネットに接続可能な環境で実行してください。

プロキシ経由でインターネットに接続する環境の場合は以下のように –proxy オプションを付けて install.sh を実行してください。

```
# ./install.sh --proxy proxy:3128
```

注釈: オプションに渡す "proxy" や "3128" の部分は導入環境のプロキシサーバのホスト名（または IP アドレス）やポート番号に合わせてください。

認証付きプロキシサーバの場合は、以下のように "user" にユーザー名を、"password" にパスワードを指定して install.sh を実行してください。

```
# ./install.sh --proxy user:password@proxy:3128
```

コマンドラインオプション

install.sh に指定可能なオプションは以下の通りです。

オプション	説明
`--https`	Kompira サーバに対するアクセスを HTTPS のみに制限します（Kompira v1.5.0 からデフォルト）。HTTP でアクセスすると自動的に HTTPS にリダイレクトされるようになります。
`--no-https`	Kompira サーバに対する HTTP アクセスを許容します。
`--no-backup`	データベースのバックアップ取得およびリストア処理をスキップします。
`--initdata`	明示的にデータベースを初期化します。
`--initfile`	明示的に添付ファイルの保存先を初期化します。
`--force`	Kompira のメジャーバージョンが異なっていても強制的にインストールを試行します。また既存データベースがあるときに削除確認をせず、データベース初期化してインストールを試行します。
`--proxy <proxy>`	プロキシサーバの URL を指定してインストールを行います。ここで指定したプロキシサーバは Kompira サービスの環境変数として設定され、ジョブフローから外部 HTTP アクセスするときにも適用されます。
`--temp-proxy <proxy>`	インストール時だけ適用するプロキシサーバの URL を指定してインストールを行います。
`--locale-lang <LANG>`	ロケールを指定してインストールを行ないます。
`--locale-timezone <ZONENAME>`	タイムゾーンを指定してインストールを行ないます。
`--jobmngr <kompira_ip>`	ジョブマネージャパッケージのインストール、アップデートを行います。Kompira パッケージをインストールしたサーバを示すホスト名もしくは IP アドレスを指定する必要があります。
`--sendevt <kompira_ip>`	イベント送信パッケージのインストール、アップデートを行います。Kompira パッケージをインストールしたサーバを示すホスト名もしくは IP アドレスを指定する必要があります。

jobmngr, sendevt オプションは排他です。

1.2.3 Kompira パッケージ

Kompira の本体にあたるパッケージのインストール・アップデートについて記述します。

インストール

Kompira パッケージを展開し、install.sh を実行します。<version>は Kompira のバージョン番号に置き換えてください。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh
[2018-04-09 19:46:34] ****:␣
↪************************************************************
[2018-04-09 19:46:34] ****: Kompira-1.5.0:
[2018-04-09 19:46:34] ****: Start: Install the Kompira
[2018-04-09 19:46:34] ****:
[2018-04-09 19:46:34] INFO:     SYSTEM                  = CENT
[2018-04-09 19:46:34] INFO:     SYSTEM_RELEASE          = CentOS Linux release 7.4.
↪1708 (Core)
[2018-04-09 19:46:34] INFO:     PYTHON                  = /usr/bin/python2.7
[2018-04-09 19:46:34] INFO:     SYSTEMD                 = true
[2018-04-09 19:46:34] INFO:     TMPDIR                  = /root/kompira-1.5.0-bin/.
↪tmp.install-20180409-1946.rA4C

...

[2018-04-09 19:51:51] ****: ------------------------------------------------------------
↪-----
[2018-04-09 19:51:51] ****: Test access to kompira.
[2018-04-09 19:51:51] ****:
[2018-04-09 19:51:51] VERBOSE: run: curl -ksf https://localhost/.login
[2018-04-09 19:51:52] INFO: Access succeeded:   <div class="brand-version">1.5.0</div>
[2018-04-09 19:51:52] ****:
[2018-04-09 19:51:52] ****: Finish: Install the Kompira (status=0)
[2018-04-09 19:51:52] ****:␣
↪************************************************************
```

インストーラは自動で Kompira パッケージのインストール処理を行います。「Finish: Install the Kompira (status=0)」という表示があれば、正常に完了しています。

インストールが完了したら以下の URL に対して Web ブラウザから Kompira サーバにアクセスし、ログイン画面が表示されることを確認してください。

この際、サーバ証明書の警告が表示されます。この警告を表示されなくなるようにするためには、Kompira サーバ

用の SSL 証明書を取得し、Kompira サーバ上の Apache へのインストールを行ってください。

```
https://<Hostname or ipaddress of Kompira server>/
```

注釈: HTTP でアクセスする場合は –no-https オプションを付けてインストールする必要があります。

Web ブラウザでの操作方法については、操作ガイドマニュアルを参照してください。

アップデート

既に Kompira パッケージがインストールされている場合に、アップデートを行う方法について説明します。

Kompira のバージョン番号形式は次のように定められています。

```
1.<major-version>.<minor-version>
```

マイナーバージョン番号のみが変更されるアップデートをマイナーアップデート、メジャーバージョン番号が変更されるアップデートをメジャーアップデートと呼びます。

例えばバージョン 1.5.0 から 1.5.2 へのアップデートはマイナーアップデート、バージョン 1.4.10 から 1.5.0 へのアップデートはメジャーアップデートとなります。

メジャーアップデートはアーキテクチャ構成や DB スキーマの定義変更を含む可能性のあるアップデートであるため、マイナーアップデートとは異なる作業が必要となる場合があります。

アップデートの際は、現在使用している Kompira バージョンとアップデートする Kompira バージョンを確認の上で作業を行ってください。

マイナーアップデート

マイナーアップデートの場合は、オプション無しで install.sh を起動します。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh
[2018-04-09 21:36:13] ****:␣
↪************************************************************
[2018-04-09 21:36:13] ****: Kompira-1.5.0:
[2018-04-09 21:36:13] ****: Start: Install the Kompira
[2018-04-09 21:36:13] ****:
[2018-04-09 21:36:13] INFO:     SYSTEM                  = CENT
[2018-04-09 21:36:13] INFO:     SYSTEM_RELEASE          = CentOS Linux release 7.
↪4.1708 (Core)
[2018-04-09 21:36:13] INFO:     PYTHON                  = /usr/bin/python2.7
[2018-04-09 21:36:13] INFO:     SYSTEMD                 = true
```

(次のページに続く)

(前のページからの続き)

```
[2018-04-09 21:36:13] INFO:     TMPDIR                    = /root/kompira-1.5.0-
↪bin/.tmp.install-20180409-2136.RYDF

...

[2018-04-09 21:36:35] ****: ------------------------------------------------------
↪----------
[2018-04-09 21:36:35] ****: Check version of Kompira installed.
[2018-04-09 21:36:35] ****:
[2018-04-09 21:36:36] INFO: VERSION=1.5.0 [pip=/opt/kompira/bin/pip]
[2018-04-09 21:36:36] INFO: A compatible version is installed.

...

[2018-04-09 21:37:23] ****: ------------------------------------------------------
↪----------
[2018-04-09 21:37:23] ****: Test access to kompira.
[2018-04-09 21:37:23] ****:
[2018-04-09 21:37:23] VERBOSE: run: curl -ksf https://localhost/.login
[2018-04-09 21:37:24] INFO: Access succeeded:   <div class="brand-version">1.5.0</
↪div>
[2018-04-09 21:37:24] ****:
[2018-04-09 21:37:24] ****: Finish: Install the Kompira (status=0)
[2018-04-09 21:37:24] ****:␣
↪************************************************************
```

インストーラは自動で Kompira パッケージのアップデート処理を行います。「Finish: Install the Kompira (status=0)」という表示があれば、正常に完了しています。

アップデートが完了したら Web ブラウザから Kompira にログインし、Kompira のバージョン番号表記が更新されていることを確認してください。

メジャーアップデート

メジャーアップデートの場合は以下の手順でアップデートを行います。

- export_data コマンドで Kompira に保存されているデータを取り出す
- install.sh コマンドで –initdata オプションを付けてデータベース初期化モードで Kompira をインストールする
- import_data コマンドで最初に取り出したデータを Kompira に保存する

install.sh コマンドを実行する時点で、既存のデータベースが初期化されることに注意してください。

```
$ cd kompira-<version>-bin
$ /opt/kompira/bin/manage.py export_data --owner-mode --virtual-mode / > backup.
↪json
```

(次のページに続く)

(前のページからの続き)

```
# ./install.sh --force --initdata
$ /opt/kompira/bin/manage.py import_data --owner-mode --overwrite-mode backup.json
[2018-04-09 21:44:15,936:30953:MainProcess] INFO: import data: start...
[2018-04-09 21:44:16,010:30953:MainProcess] INFO: import object: imported "system/
↪types/TypeObject" to "/system/types/TypeObject" (updated)
[2018-04-09 21:44:16,022:30953:MainProcess] INFO: import object: imported "system/
↪types/Directory" to "/system/types/Directory" (updated)
[2018-04-09 21:44:16,033:30953:MainProcess] INFO: import object: imported "system"
 to "/system" (updated)

...

[2018-04-09 21:44:22,126:30953:MainProcess] INFO: import fields: /user/data: []
[2018-04-09 21:44:22,164:30953:MainProcess] INFO: import fields: /user/data/
↪nodes: []
[2018-04-09 21:44:22,202:30953:MainProcess] INFO: import fields: /user/data/
↪accounts: []
[2018-04-09 21:44:22,218:30953:MainProcess] INFO: import data: finished␣
↪(created=0, updated=59, skipped=0, error=0, warning=0)
```

install.sh を実行する際は、データベースを初期化するために –initdata オプションと –force オプションを指定します。

import_data の処理が完了したら Web ブラウザから Kompira にログインし、Kompira のバージョン番号表記が更新されていること、既存の Kompira 上で作成していた Kompira オブジェクトが存在することを確認してください。

1.2.4 ジョブマネージャパッケージ

ジョブマネージャとイベント送信スクリプトを含むパッケージのインストール・アップデートについて記述します。

インストール

Kompira パッケージを展開し、install.sh を実行します。ジョブマネージャは Kompira 本体と通信を行うため、Kompira パッケージをインストールしたサーバのホスト名、もしくは IP アドレスを install.sh の引数に指定する必要があります。

<version>は Kompira のバージョン番号に置き換えてください。

<kompira_ip>は Kompira サーバのホスト名、もしくは IP アドレスです。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
```

(次のページに続く)

(前のページからの続き)

```
# ./install.sh --jobmngr <kompira_ip>
[2018-08-09 14:35:52] ****:␣
↪************************************************************
[2018-08-09 14:35:52] ****: Kompira-1.5.0:
[2018-08-09 14:35:52] ****: Start: Install the Kompira
[2018-08-09 14:35:52] ****:
[2018-08-09 14:35:52] INFO:     SYSTEM                  = CENT
[2018-08-09 14:35:52] INFO:     SYSTEM_RELEASE          = CentOS Linux release 7.4.
↪1708 (Core)
[2018-08-09 14:35:52] INFO:     PYTHON                  = /bin/python2.7
[2018-08-09 14:35:52] INFO:     SYSTEMD                 = true
[2018-08-09 14:35:52] INFO:     TMPDIR                  = /root/kompira-1.5.0-bin/.
↪tmp.install-20180809-1435.zmWV
[2018-08-09 14:35:52] INFO:     PATH                    = /sbin:/bin:/usr/sbin:/usr/
↪bin

...

[2018-08-09 14:36:17] ****: ------------------------------------------------------------
↪-----
[2018-08-09 14:36:17] ****: Setup kompira-jobmngrd.
[2018-08-09 14:36:17] ****:
[2018-08-09 14:36:17] INFO: [systemd] Create kompira-jobmngrd configuration file: /etc/
↪sysconfig/kompira_jobmngrd
[2018-08-09 14:36:17] INFO: [systemd] Enable kompira-jobmngrd service.
[2018-08-09 14:36:17] VERBOSE: run: install -m 664 /root/kompira-1.5.0-bin/.tmp.
↪install-20180809-1435.zmWV/kompira_jobmngrd.service /usr/lib/systemd/system/kompira_
↪jobmngrd.service
[2018-08-09 14:36:17] VERBOSE: run: systemctl daemon-reload
[2018-08-09 14:36:17] INFO: Start kompira-jobmngrd service.
[2018-08-09 14:36:17] VERBOSE: run: systemctl enable kompira_jobmngrd.service
[2018-08-09 14:36:17] VERBOSE: run: systemctl restart kompira_jobmngrd
[2018-08-09 14:36:17] ****:
[2018-08-09 14:36:17] ****: Finish: Install the Kompira (status=0)
[2018-08-09 14:36:17] ****:␣
↪************************************************************
```

インストーラは自動でジョブマネージャパッケージの新規インストール処理を行います。「Finish: Install the Kompira (status=0)」という表示があれば、正常に完了しています。

ジョブマネージャプロセスが正しく起動しているかの確認方法については、*Kompira* ジョブマネージャの起動・停止・状態確認 を参照してください。

また、Kompira 本体がジョブマネージャと正しく通信できているかどうかについては、Kompira の「管理領域設定ページ」から確認することができます。Web ブラウザから Kompira にログインし、「設定」→「管理領域設定」→「default」ページを表示してください。「ジョブマネージャ状態」の欄にジョブマネージャパッケージをインストールしたサーバのホスト名が表示されていれば、正常に通信できています。

アップデート

インストールと同様の手順を実行することで、ジョブマネージャパッケージのアップデートを行うことができます。

1.2.5 イベント送信パッケージ

イベント送信スクリプトを含むパッケージのインストール・アップデートについて記述します。

イベント送信パッケージは、Linux および Windows へのインストールに対応しています。

RHEL/CentOS へのインストール

Kompira パッケージを展開し、install.sh を実行します。

イベント送信スクリプトは Kompira 本体にデータを送信するため、Kompira パッケージをインストールしたサーバのホスト名、もしくは IP アドレスを install.sh の引数に指定する必要があります。

<version>は Kompira のバージョン番号に置き換えてください。<kompira_ip>は Kompira サーバのホスト名、もしくは IP アドレスです。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh --sendevt <kompira_ip>
[2018-08-09 14:22:34] ****:␣
↪************************************************************
[2018-08-09 14:22:34] ****: Kompira-1.5.0:
[2018-08-09 14:22:34] ****: Start: Install the Kompira
[2018-08-09 14:22:34] ****:
[2018-08-09 14:22:34] INFO:     SYSTEM                  = CENT
[2018-08-09 14:22:34] INFO:     SYSTEM_RELEASE          = CentOS Linux release 7.4.
↪1708 (Core)
[2018-08-09 14:22:34] INFO:     PYTHON                  = /usr/bin/python2.7
[2018-08-09 14:22:34] INFO:     SYSTEMD                 = true
[2018-08-09 14:22:34] INFO:     TMPDIR                  = /root/kompira-1.5.0-bin/.
↪tmp.install-20180809-1422.oyzV

...

[2018-08-09 14:22:51] ****: ------------------------------------------------------------
↪-----
[2018-08-09 14:22:51] ****: Setup kompira common files.
[2018-08-09 14:22:51] ****:
[2018-08-09 14:22:51] INFO: Create log directory: /var/log/kompira
[2018-08-09 14:22:51] VERBOSE: run: install -g kompira -m 775 -d /var/log/kompira
[2018-08-09 14:22:51] VERBOSE: run: find /var/log/kompira -type f -user root -exec␣
↪chown kompira:kompira {} ;
```

(次のページに続く)

(前のページからの続き)

```
[2018-08-09 14:22:51] INFO: Create Kompira/Kompira-sendevt configuration file.
--- /opt/kompira/kompira.conf   2018-08-09 14:03:11.021994835 +0900
+++ /root/kompira-1.5.0-bin/.tmp.install-20180809-1422.oyzV/kompira.conf  2018-08-09␣
↪14:22:51.923983573 +0900
@@ -2,7 +2,7 @@
 site_id                = 1

 [amqp-connection]
-server          = localhost
+server          = <kompira_ip>
 port            = 5672
 user            = guest
 password        = guest
[2018-08-09 14:22:51] VERBOSE: run: install -S .old -b /root/kompira-1.5.0-bin/.tmp.
↪install-20180809-1422.oyzV/kompira.conf -m 644 /opt/kompira/kompira.conf
[2018-08-09 14:22:51] ****:
[2018-08-09 14:22:51] ****: Finish: Install the Kompira (status=0)
[2018-08-09 14:22:51] ****:␣
↪************************************************************
```

「Finish: Install the Kompira (status=0)」という表示があれば、正常に完了しています。

インストールが完了すると、/opt/kompira/bin 以下に kompira_sendevt コマンドが配置された状態となります。

```
$ /opt/kompira/bin/kompira_sendevt --version
kompira_sendevt (Kompira version 1.5.0)
```

Windows へのインストール

1. **Python** のインストール

   Python 2.7 を Windows にインストールします。

   - https://www.python.org/downloads/

   上記の公式ダウンロードサイトから最新の Python 2.7 のインストーラをダウンロードして、対象の Windows にインストールしてください。

   インストールが完了したら、コマンドラインから Python が起動できるように、環境変数のパスを追加しておきます。

パス	説明
C:\Python27	Python のコマンドが含まれるフォルダ
C:\Python27\Scripts	pip コマンドなどが格納されるフォルダ

2. **Kompira 用 Python 仮想環境 (virtualenv) の作成**

   C:\Kompira に Kompira 用の独立した Python 仮想環境 (virtuelenv) を作成します。

```
C:\> pip install virtualenv
C:\> python -m virtualenv C:\Kompira
```

3. **ログファイル出力先ディレクトリの作成**

   ログファイルの出力先としてディレクトリ C:\var\log\kompira を作成します。

```
C:\> mkdir C:\var\log\kompira
```

4. **kompira_sendevt パッケージのインストール**

   Kompira パッケージを Windows 上にダウンロードして展開したら、dist ディレクトリに含まれる Kompira_sendevt-<version>-py2-none-any.whl を仮想環境の pip コマンドでインストールします。

```
C:\> C:\Kompira\Scripts\pip.exe install Kompira_sendevt-1.5.0-py2-none-any.whl
Processing c:\users\kompira\documents\kompira-package\kompira_sendevt-1.5.0-
↪py2-none-any.whl
Collecting subprocess32~=3.2.7 (from Kompira-sendevt==1.5.0)

...

Installing collected packages: subprocess32, amqp, decorator, simplejson,␣
↪Kompira-sendevt
Successfully installed Kompira-sendevt-1.5.0 amqp-1.4.9 decorator-4.2.1␣
↪simplejson-3.13.2 subprocess32-3.2.7
```

   以上でイベント送信パッケージのインストールは完了です。kompira_sendevt コマンドは C:\Kompira\Scripts にインストールされますので、次のように kompira_sendevt コマンドを実行してみてください。

```
C:\> C:\Kompira\Scripts\kompira_sendevt.exe --version
kompira_sendevt (Kompira version 1.5.0)
```

   正しくインストールされていれば、バージョン番号が出力されます。

   C:\Kompira\Scripts も環境変数のパスに追加すれば、パスを省略して実行できます。

アップデート

インストールと同様の手順を実行することで、イベント送信パッケージのアップデートを行うことができます。

1.3 Kompira プロセス管理

Kompira システムは複数のプロセスが協調動作することにより成り立っています。以下では Kompira を構成するプロセスについての情報を記述します。

1.3.1 Kompira のプロセス構成

Kompira システムを構成するプロセスは、以下のようになっています。

Kompira デーモン (kompirad) Kompira ジョブフローを実行、管理するためのデーモンプロセスです。

Kompira ジョブマネージャに対してリモートコマンドの実行指示を出し、結果を受け取ります。

Kompira ジョブマネージャ (kompira_jobmngrd) Kompira デーモンから指示されたリモートコマンドを実行するためのデーモンプロセスです。

Kompira デーモンからリモートコマンドの指示を受信すると、Kompira ジョブマネージャは ssh または winrs でリモートホストへ接続し、コマンドを実行します。コマンドの実行結果は Kompira デーモンに送信されます。

その他、Kompira システムに必要なプロセスには、Apache(httpd)、PostgreSQL(postgresql)、RabbitMQ(rabbitmq-server)、lsyncd があります。

これらの各プロセスは、マシン起動時に自動的に起動するように install.sh によって設定されます。

1.3.2 Kompira デーモンの起動・停止・状態確認

Kompira デーモンの起動・停止は、ルート権限で行ってください。起動後の実効ユーザーは自動的に kompira に変更されます。

RHEL/CentOS 7 系の場合

RHEL/CentOS 7 系での Kompira デーモンの起動は以下のコマンドによって行います。

```
# systemctl start kompirad
```

停止には以下のコマンドを実行します。

```
# systemctl stop kompirad
```

systemctl status コマンドで、Kompira デーモンの状態を確認することができます。

```
$ systemctl status kompirad.service
* kompirad.service - Kompira-daemon
   Loaded: loaded (/usr/lib/systemd/system/kompirad.service; enabled; vendor preset:␣
↪disabled)
   Active: active (running) since Thu 2018-07-05 16:33:02 JST; 2h 20min ago
  Process: 5277 ExecStartPre=/opt/kompira/bin/prestart_kompirad.sh (code=exited,␣
↪status=0/SUCCESS)
 Main PID: 5368 (kompirad)
   CGroup: /system.slice/kompirad.service
           `-5368 /opt/kompira/bin/python2.7 /opt/kompira/bin/kompirad

 Jul 05 16:32:32 kompira-install-test3.dev.fixpoint.co.jp systemd[1]: Starting Kompira-
↪daemon...
 Jul 05 16:33:02 kompira-install-test3.dev.fixpoint.co.jp systemd[1]: Started Kompira-
↪daemon.
```

起動している時は Active: の欄が **active (running)**、停止している時は **inactive (dead)** と表示されます。

RHEL/CentOS 6 系の場合

RHEL/CentOS 6 系での Kompira デーモンの起動は以下のコマンドによって行います。

```
# start kompirad
kompirad start/running, process xxxx
```

停止には以下のコマンドを実行します。

```
# stop kompirad
kompirad stop/waiting
```

status コマンドで、Kompira デーモンの状態を確認することができます。

```
$ status kompirad
kompirad start/running, process xxxx
```

起動している時は **start/running**、停止している時は **stop/waiting** と表示されます。

1.3.3 Kompira ジョブマネージャの起動・停止・状態確認

Kompira ジョブマネージャの起動・停止も Kompira デーモンと同様にルート権限で行ってください。起動後の実効ユーザーは自動的に kompira に変更されます。

RHEL/CentOS 7 系の場合

RHEL/CentOS 7 系での Kompira ジョブマネージャの起動は以下のコマンドによって行います。

```
# systemctl start kompira_jobmngrd.service
```

停止には以下のコマンドを実行します。

```
# systemctl stop kompira_jobmngrd.service
```

status コマンドで Kompira ジョブマネージャの状態を確認することができます。

```
$ systemctl status kompira_jobmngrd.service
* kompira_jobmngrd.service - Kompira-jobmanager
   Loaded: loaded (/usr/lib/systemd/system/kompira_jobmngrd.service; enabled; vendor␣
↪preset: disabled)
   Active: active (running) since Thu 2018-07-05 16:32:22 JST; 2h 25min ago
 Main PID: 5164 (kompira_jobmngr)
   CGroup: /system.slice/kompira_jobmngrd.service
           |-5164 /opt/kompira/bin/python2.7 /opt/kompira/bin/kompira_jobmngrd
           `-5197 /opt/kompira/bin/python2.7 /opt/kompira/bin/kompira_jobmngrd

 Jul 05 16:32:22 kompira-install-test3.dev.fixpoint.co.jp systemd[1]: Started Kompira-
↪jobmanager.
 Jul 05 16:32:22 kompira-install-test3.dev.fixpoint.co.jp systemd[1]: Starting Kompira-
↪jobmanager...
```

起動している時は Active: の欄が **active (running)**、停止している時は **inactive (dead)** と表示されます。

RHEL/CentOS 6 系の場合

RHEL/CentOS 6 系での Kompira ジョブマネージャの起動は以下のコマンドによって行います。

```
# start kompira_jobmngrd
kompira_jobmngrd start/running, process xxxx
```

停止には以下のコマンドを実行します。

```
# stop kompira_jobmngrd
kompira_jobmngrd stop/waiting
```

status コマンドで Kompira ジョブマネージャの状態を確認することができます。

```
$ status kompira_jobmngrd
kompira_jobmngrd start/running, process xxxx
```

起動している時は **start/running**、停止している時は **stop/waiting** と表示されます。

1.3.4 Kompira 使用ポート一覧

Kompira パッケージをインストールしたサーバ上では、以下に挙げるポートに外部から接続可能である必要があります。

ポート番号	説明
80/TCP	HTTP (HTTPS からのみブラウザアクセスする場合は不要)
443/TCP	HTTPS (HTTP からのみブラウザアクセスする場合は不要)
5672/TCP	AMQP (ジョブマネージャパッケージおよびイベント送信パッケージ使用時のみ)
5405/UDP	corosync (冗長構成時のみ)
873/TCP	rsync (冗長構成時のみ)
5432/TCP,UDP	PostgreSQL (冗長構成時のみ)

1.4 ノードの設定

Kompira からリモートジョブを実行する対象ノードの制御方法は、'ssh' または 'winrs' がサポートされています。UNIX/Linux 機器に接続する場合は「ssh ノードの設定」、Windows 機器に接続する場合は「winrs ノードの設定」を行う必要があります。

1.4.1 ssh ノードの設定

Kompira から ssh ノードに対してコマンド実行を行う場合、ssh バージョン 2 でログインできる状態である必要があります。最近の Linux などであればインストールした時点で ssh ログインできる状態になっているため、特に設定する必要はありません。それ以外の対象ノードで ssh ログインを有効にする方法については、各システムのマニュアル等を参考にして設定してください。

注釈: サポートしている SSH のバージョンは v2 のみとなっています。v1 には対応しておりません。

1.4.2 winrs ノードの設定

Kompira から winrs ノードに対してコマンド実行を行う場合、事前に winrs ノード側にて WinRM の設定が必要となります。なお、対応している WinRM のバージョンは 1.1, 2.0, 3.0 になります。

1. **WinRM のリモート管理を有効にする**

   WinRM を有効にするため、Windows のコマンドプロンプトを「管理者として実行」して、`winrm quickconfig` （または `winrm qc`）と実行してください。「変更しますか [y/n]?」と表示されたら `y`

と答えてください。なおこの操作は最初に 1 回行っておけば、2 回目以降は不要です。

以下は Windows 7 で実行した場合の例ですが、Windows のバージョンやその時点での設定状況によって表示される内容の詳細は異なります。

```
C:\>winrm quickconfig
WinRM はこのコンピューター上で要求を受信するように設定されていません。
次の変更を行う必要があります:

WinRM サービスの種類を遅延自動開始に設定します。
WinRM サービスを開始します。
ローカル ユーザーにリモートで管理権限を付与するよう LocalAccountTokenFilterPolicy を構成してください。

変更しますか [y/n]? y

WinRM は要求を受信するように更新されました。

WinRM サービスの種類を正しく変更できました。
WinRM サービスが開始されました。
ローカル ユーザーにリモートで管理権限を付与するよう LocalAccountTokenFilterPolicy を構成しました。
WinRM は、管理用にこのコンピューターへのリモート アクセスを許可するように設定されていません。
次の変更を行う必要があります:

このコンピューター上のあらゆる IP への WS-Man 要求を受け付けるため、HTTP://* 上に WinRM リスナーを作成します。
WinRM ファイアウォールの例外を有効にします。

変更しますか [y/n]? y

WinRM はリモート管理用に更新されました。

このコンピューター上のあらゆる IP への WS-Man 要求を受け付けるため、HTTP://* 上に WinRM リスナーを作成しました。
WinRM ファイアウォールの例外を有効にしました。
```

2. **WinRM の接続設定を変更する**

   WinRM で非暗号化通信を許可するため、同様に（管理者として実行した）コマンドプロンプトから以下を実行します。

```
C:\> winrm set winrm/config/service @{AllowUnencrypted="true"}
```

注釈: Kompira Ver.1.4.10 以降では、デフォルトで NTLM 認証に対応したため、BASIC 認証の許可設定は不要となりました。

---

3. ジョブフローでの動作確認

Kompira で以下のようなジョブフローを作成・実行して winrs ノードへのコマンド制御ができるか確認してください。

```
[__host__ = '<Windows サーバのアドレス>',
 __user__ = '<Windows ユーザー名>',
 __password__ = '<Windows ユーザーパスワード>',
 __conntype__ = 'winrs']
-> ['ver']
-> print($RESULT)
```

ジョブフロープロセスのコンソールに Windows のバージョン番号が表示されれば成功です。

なお WinRM 2.0 以降ではデフォルトで TCP ポート 5985 番を利用しますが、Windows Server 2008 などの WinRM 1.1 では利用するポート番号が 80 となっていますので、その際はポート番号の設定 `__port__ = 80` を追加するようにしてください。

うまく接続できない場合は、ファイアウォールで TCP ポート 5985 番（や 80 番）が通過できない設定になっていないか、ログインアカウントの設定が正しいかなどを確認してください。

1.5 Kompira の設定とログファイル

Kompira が標準で使用するサーバ上のディレクトリおよび設定ファイルについて記述します。

1.5.1 Kompira 標準ディレクトリ

Kompira が標準で使用するサーバ上のディレクトリおよび設定ファイルの一覧を以下に示します。

パス		説明
/opt/kompira/		
	bin/	Kompira コマンド実行ファイル用ディレクトリ
	kompira.conf	Kompira 設定ファイル
/var/log/kompira/		エンジンやジョブマネージャのログファイル用ディレクトリ
/var/opt/kompira/		Kompira の各種可変ファイル用ディレクトリ
	kompira.lic	Kompira のライセンスファイル
	html/	オンラインヘルプの HTML ファイル群
	repository/	リポジトリ連携のための作業用ディレクトリ
	upload/	添付ファイル保存用ディレクトリ
/etc/httpd/conf.d/	kompira.conf	Apache 設定ファイル

1.5.2 Kompira ログ

Kompira のログファイルは、標準で以下のディレクトリ下に作成されます。

- /var/log/kompira/

作成されるログファイル名と内容は以下のようになっています。

ログファイル名	内容
kompira.log	リクエスト関連のログ出力
kompirad.log	Kompira デーモンのログ出力
process.log	Kompira ジョブフロープロセスのログ出力
kompira_jobmngrd.log	Kompira ジョブマネージャのログ出力
kompira_sendevt.log	イベント送信コマンドのログ出力

ログファイルは自動的にローテートされ、古いログファイルは日付がファイル名に追加された上で保存されます。

上記ログファイルのうち、kompira_jobmngrd.log、kompira_sendevt.log は/opt/kompira/kompira.conf によりローテートの間隔やバックアップする世代数の設定を変更することができます。

1.5.3 Kompira 設定ファイル

/opt/kompira/kompira.conf での設定項目は、次のとおりです。

セクション名	項目名	デフォルト値	内容
kompira	site_id	1	本バージョンでは未使用
logging	ログ出力関連の設定		
	loglevel	INFO	ログ出力レベルの設定 (DEBUG, INFO, WARNING, ERROR, CRITICAL)
	logdir	/var/log/kompira	ログファイルのディレクトリ
	logbackup	kompirad: 7 kompira_jobmngrd: 7 kompira_sendevt: 10	ログバックアップの世代数
	logmaxsz	kompirad: 0 kompira_jobmngrd: 0 kompira_sendevt: 1024*1024*1024	ログファイルの最大サイズ (単位は byte) 0 に設定すると日次ローテートとなる
amqp-connection	RabbitMQ の接続情報関連の設定		
	server	localhost	接続ホスト名
	port	5672	接続ポート番号
	user	guest	接続ユーザー名
	password	guest	接続パスワード
	ssl	false	SSL で接続するかどうか
	heartbeat_interval	10	ハートビートの送信間隔 (単位は秒)
	max_retry	3	接続断時に再接続する最大試行回数
	retry_interval	30	接続断時に再接続する間隔 (単位は秒)
agent	ジョブマネージャの動作に関する設定		
	name	default	ジョブマネージャの名称
	script_dir	/var/tmp/kompira_jobmngrd	スクリプトジョブ用の一時ディレクトリ
	remote_script_dir	~/.kompira_jobmngrd	スクリプトジョブ用のリモートの一時ディレクトリ
	ssh_port	未設定	デフォルトで使用する ssh 接続ポート番号
	ssh_keyfile	未設定	デフォルトで使用する

1.6 Kompira のデータバックアップ

Kompira 上に保存したデータのバックアップとリストアを行う方法について記述します。

Kompira 上で作成したジョブフローや機器情報の定義は、データベースに格納されます。これらのデータを json 形式でファイルにエクスポートしたり、インポートしたりすることが可能です。

1.6.1 Kompira オブジェクトのエクスポート

以下の形式の export_data コマンドを実行することで、標準出力に引数で指定した Kompira ファイルシステムのパス以下のデータが json 形式でダンプされます。

```
/opt/kompira/bin/manage.py export_data [options] <path>...
```

たとえば、Kompira で作成した/home/guest 以下の全てのデータをファイルにエクスポートするには、以下のコマンドを実行します。

```
$ /opt/kompira/bin/manage.py export_data /home/guest > guest.json
```

または、export_dir コマンドを実行することで、引数で指定した Kompira ファイルシステムのパス以下のデータを、オブジェクト単位にファイルとしてダンプすることができます。

```
/opt/kompira/bin/manage.py export_dir [options] <path>...
```

1.6.2 Kompira オブジェクトのインポート

import_data コマンドを用いて、エクスポートしたファイルからデータを取り込むことができます。import_data コマンドの形式は以下のとおりです。

```
/opt/kompira/bin/manage.py import_data [options] <filename>...
```

たとえば、/home/guest ディレクトリをエクスポートしたファイル guest.json を取り込むには、以下のコマンドを実行します。

```
$ /opt/kompira/bin/manage.py import_data guest.json
[2014-07-25 12:34:49,576] INFO: import data: start...
[2014-07-25 12:34:49,676] INFO: home/guest: import is skipped: "/home/guest" already␣
↪exists.
[2014-07-25 12:34:49,710] INFO: home/guest/a: import is skipped: "/home/guest/a"␣
↪already exists.
[2014-07-25 12:34:49,743] INFO: home/guest/b: import is skipped: "/home/guest/b"␣
↪already exists.
[2014-07-25 12:34:49,743] INFO: import data: finished (created=0, updated=0, skipped=3,
 error=0)
```

(次のページに続く)

(前のページからの続き)

インポートする json ファイル中に既に存在するパスのオブジェクトが含まれていた場合、そのオブジェクトのインポートはスキップされます。上記例の場合は、インポートしようとしたファイル 3 つが全てスキップされた状態です。

overwrite-mode オプションを指定すれば、上書きインポートをすることが可能です。

```
$ /opt/kompira/bin/manage.py import_data --overwrite-mode guest.json
[2014-07-25 12:39:15,685] INFO: import data: start...
[2014-07-25 12:39:15,821] INFO: import object: imported "home/guest" to "/home/guest"␣
↪(updated)
[2014-07-25 12:39:15,904] INFO: import object: imported "home/guest/a" to "/home/guest/
↪a" (updated)
[2014-07-25 12:39:15,971] INFO: import object: imported "home/guest/b" to "/home/guest/
↪b" (updated)
[2014-07-25 12:39:15,991] INFO: import fields: /home/guest: []
[2014-07-25 12:39:16,015] INFO: import fields: /home/guest/a: ['wikitext']
[2014-07-25 12:39:16,046] INFO: import fields: /home/guest/b: ['wikitext']
[2014-07-25 12:39:16,046] INFO: import data: finished (created=0, updated=3, skipped=0,
 error=0)
```

export_dir コマンドでダンプしたファイルについては、import_dir コマンドを用いて取り込むことが出来ます。

```
/opt/kompira/bin/manage.py import_dir [options] <dirname>...
```

1.6.3 バックアップ

Kompira の全てのデータをバックアップするための手順について説明します。

Kompira はデータベース上以外に、サーバ上の *Kompira 標準ディレクトリ* に挙げられているパスのデータを使用します。Kompira のデータをバックアップする際は、export_data コマンドによる Kompira オブジェクトのバックアップに加えて、必要に応じてサーバ上のファイルバックアップも行う必要があります。

特に、/var/opt/kompira/upload/は Kompira 上の添付ファイルフィールドからアップロードされたファイルの実体が保存されるディレクトリですので、バックアップしておくのがよいでしょう。

Kompira オブジェクトと添付ファイルディレクトリのバックアップを行う場合の例を以下に示します。

```
$ mkdir -p /tmp/kompira_backup
$ cd /tmp/kompira_backup
$ /opt/kompira/bin/manage.py export_data / --virtual-mode > backup.json
$ cp -r /var/opt/kompira/upload ./
$ cd /tmp
$ tar zcf kompira_backup.tar.gz ./kompira_backup
```

1.6.4 export_data のオプション

export_data コマンドには以下のオプションがあります。

オプション	説明
--directory=DIRECTORY	エクスポートするパスの起点となるディレクトリを指定します。(デフォルトは'/')
--virtual-mode	仮想ファイルシステムに含まれるデータも出力します。
--owner-mode	エクスポート対象となったオブジェクトの所有者のユーザオブジェクトおよびそのユーザの所属グループオブジェクトも出力します。
-h, --help	ヘルプメッセージを表示します。

1.6.5 export_dir のオプション

export_dir コマンドには以下のオプションがあります。

オプション	説明
--directory=DIRECTORY	エクスポートするパスの起点となるディレクトリを指定します。(デフォルトは'/')
--property-mode	表示名など属性も出力します。
--datetime-mode	作成日時と更新日時も出力します。
--current=CURRENT_DIR	出力先のディレクトリを指定します。
-h, --help	ヘルプメッセージを表示します。

1.6.6 import_data のオプション

import_data コマンドには以下のオプションがあります。

オプション	説明
--user=USER	インポートするデータの所有者を USER (ユーザー ID を指定) に設定します。
--directory=ORIGIN-DIR	インポート先の起点となるディレクトリを指定します。(デフォルトは'/')
--overwrite-mode	既存のファイルがある場合に上書きします。
--owner-mode	インポートするデータの所有者をエクスポート時の所有者情報に設定します。
--now-updated-mode	現在時刻をオブジェクトの更新日時に設定します。
-h, --help	ヘルプメッセージを表示します。

1.6.7 import_dir のオプション

import_dir コマンドには以下のオプションがあります。

オプション	説明
--user=USER	インポートするデータの所有者を USER (ユーザー ID を指定) に設定します。
--directory=ORIGIN-DIR	インポート先の起点となるディレクトリを指定します。(デフォルトは'/')
--overwrite-mode	既存のファイルがある場合に上書きします。
--owner-mode	インポートするデータの所有者をエクスポート時の所有者情報に設定します。
--now-updated-mode	現在時刻をオブジェクトの更新日時に設定します。
-h, --help	ヘルプメッセージを表示します。

1.7 Kompira ライセンス

license_info コマンドを用いて、Kompira のライセンス状態を確認することができます。license_info コマンドの形式は以下のとおりです。

```
/opt/kompira/bin/manage.py license_info
```

以下はライセンスが登録されている場合の実行例です。

```
$ /opt/kompira/bin/manage.py license_info
*** Kompira License Information ***
License ID:     KP-REGLM0-0000000001
Edition:        REGL
Hardware ID:    NODE:000C29FB949E
Expire date:    2015-12-31
The number of registered nodes: 0 / 100
The number of registered jobflows:      2 / 100
The number of registered scripts:       0 / 100
Licensee:       fixpoint,inc.
Signature:      dwyWvG9eKbnGxcpWfVr1H0wSybLkGL7UqB2E6d5f0jYapfTx/
↪AABJ66W3sRpK0byk+9Y724NuEZ9Rh90ySU8f2GRsIyujuVrgPloajokbdZrPFIqOlyvLkak8MAWcGJxiioPHPNd2Tv2BNOsq6bs
```

ライセンスが登録されていない場合は、仮ライセンス情報が表示されます。

```
$ /opt/kompira/bin/manage.py license_info
*** Kompira License Information ***
License ID:     KP-TEMP-0000000000
Edition:        temporary
Hardware ID:    NODE:000C29FB949E
```

(次のページに続く)

(前のページからの続き)

```
Expire date:    2015-01-22
The number of registered nodes: 0 / 100
The number of registered jobflows:      2 / 100
The number of registered scripts:       0 / 100
Licensee:
Signature:      None

Kompira is running with temporary license.
```

ライセンスファイルパスは/var/opt/kompira/kompira.lic です。

上記パスにライセンスファイルを配置した場合、license_info コマンドを使用してライセンスファイルが正しく Kompira から読み込まれているかどうかを確認して下さい。

例として、ライセンスが登録されていない Kompira に対して kompira_KP-REGLM100-0000000001.lic というライセンスファイルをコマンドラインで登録する手順を以下に示します。

```
$ cp kompira_KP-REGLM100-0000000001.lic /var/opt/kompira/kompira.lic
$ cd /var/opt/kompira
$ chown apache:apache kompira.lic
$ chmod 644 kompira.lic
$ /opt/kompira/bin/manage.py license_info
*** Kompira License Information ***
License ID:     KP-REGLM0-0000000001
Edition:        REGL
Hardware ID:    NODE:000C29FB949E
Expire date:    2017-12-31
The number of registered nodes: 0 / 100
The number of registered jobflows:      2 / 100
The number of registered scripts:       0 / 100
Licensee:       fixpoint,inc.
Signature:      dwyWvG9eKbnGxcpWfVr1H0wSybLkGL7UqB2E6d5f0jYapfTx/
↪AABJ66W3sRpK0byk+9Y724NuEZ9Rh90ySU8f2GRsIyujuVrgPloajokbdZrPFIqOlyvLkak8MAWcGJxiioPHPNd2Tv2BNOsq6bs
```

バージョン 1.4.2 で追加: license_info コマンド

参考:

ライセンス管理: ライセンスの確認と登録はブラウザ上からも行うことが可能です。

1.8 冗長構成管理

Kompira を 2 台のサーバで Pacemaker/corosync を用いた Active-Standby 型の冗長構成で動作させることができます。ここでは、そのインストール方法や状態確認方法、フェイルオーバー等について記述します。

1.8.1 はじめに

Pacemaker は Kompira が動作する上で必要なリソース (アプリケーション) を監視し、エラーを検知した場合はフェイルオーバーを行なうことで冗長性を実現します。

Pacemaker で監視するリソースの一覧を以下に示します。

httpd, kompirad, kompira_jobmngrd Kompira の動作に必要なプロセスです。アクティブ状態のサーバ上でのみ動作します。

RabbitMQ Kompira の動作に必要な RabbitMQ プロセスです。アクティブ状態のサーバのプロセスが Master、スタンバイ状態のサーバのプロセスが Slave として動作します。

IPaddr2 仮想 IP アドレスを管理するためのリソースです。

PostgreSQL PostgreSQL データベースプロセスです。アクティブ状態のサーバのプロセスが Master、スタンバイ状態のサーバのプロセスが Slave として動作します。PostgreSQL はレプリケーションの設定がされており、プライマリ機とセカンダリ機のデータが同期されます。

lsyncd ファイルのミラーリングを行うプロセスです。サーバのファイルシステム上に配置される添付ファイルのミラーリングを行います。アクティブ状態のサーバ上でのみ動作します。

1.8.2 インストール

Kompira の冗長構成を構築する場合、2 台のサーバにそれぞれ Kompira をインストールしてから、プライマリ機、セカンダリ機の順に冗長構成のセットアップを実施する必要があります。

2 台のサーバには 2 つのネットワークインタフェースが必要になります。ネットワークインターフェース名は OS のバージョンや環境によって、eth0, eth1,... であったり、ens192, ens224,... であったりします。

以降の説明では、それぞれのサーバをプライマリ機 (kompira-server1)、セカンダリ機 (kompira-server2) と呼び、ネットワークインターフェースとしては eth0, eth1 を持っていることとします。また、eth0 はサービス提供用のネットワークに接続され、eth1 はハートビート用インターフェースとして使用するため独立したネットワークで 2 台のサーバが接続されていることとします。

なお、冗長構成を構築した初期状態ではプライマリ機がアクティブ状態、セカンダリ機がスタンバイ状態となります。

Kompira の冗長構成を構築するにはパッケージに含まれる setup_cluster.sh を利用します。以下では OS インストール直後のサーバ 2 台に対して、冗長構成の Kompira をインストールする手順を記述します。

注釈: setup_cluster.sh は、install.sh と同様に外部から必要な各種ミドルウェアをダウンロードします。外部ネットワークに接続可能な環境で実行してください。

プライマリ機の設定

Kompira パッケージを通常インストール後、–primary オプションを指定して setup_cluster.sh を実行することでプライマリ機のセットアップを行います。

また setup_cluster.sh を実行するときに、以下の情報を引数で指定します。

- ハートビートネットワークデバイス名
- クラスタに割り当てる仮想 IP アドレス (VIP) とそのネットワークマスク長

例えば、ハートビートネットワークデバイスとして eth1 を、仮想 IP アドレスとして 192.168.0.100 とネットワークマスク長に 24 を指定する場合、以下のようにコマンドを実行します。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh
# ./setup_cluster.sh --primary --heartbeat-device eth1 192.168.0.100/24
```

注釈: setup_cluster.sh は、実行時にホスト名を変更します。プライマリの場合は kompira-server1、セカンダリの場合は kompira-server2 と設定されます。

セカンダリ機の設定

Kompira パッケージを通常インストール後、–secondary オプションを指定して setup_cluster.sh を実行することでセカンダリ機のセットアップを行ないます。

また setup_cluster.sh を実行するときに、以下の情報を引数で指定します。仮想 IP アドレスの指定は不要です。

- ハートビートネットワークデバイス名

例えば、ハートビートネットワークデバイスとして eth1 を指定する場合、以下のようにコマンドを実行します。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh
# ./setup_cluster.sh --secondary --heartbeat-device eth1
```

注釈: セカンダリ機での setup_cluster.sh の実行時、プライマリ機からデータベースのスナップショットを取得します。セットアップ済みのプライマリ機を立ち上げ、接続している状態で setup_cluster.sh を実行してください。

状態確認

プライマリ機、セカンダリ機でインストール処理が完了したら、以下の URL に対して Web ブラウザからアクセスし、ログイン画面が表示されることを確認してください。

```
http://192.168.0.100/
```

URL はプライマリ機インストール時に設定した仮想 IP アドレスです。この URL はプライマリ機に障害が発生し、フェイルオーバーが発生した場合でも維持されます。

また、冗長構成の各リソースの状態を確認するには、プライマリ機またはセカンダリ機で crm_mon コマンドを使用します。

```
$ crm_mon -A1
Last updated: Thu Jul 10 17:26:44 2014
Last change: Thu Jul 10 17:26:33 2014 via crm_attribute on kompira-server1
Stack: cman
Current DC: kompira-server1 - partition with quorum
Version: 1.1.10-14.el6_5.3-368c726
2 Nodes configured
9 Resources configured


Online: [ kompira-server1 kompira-server2 ]

 Resource Group: webserver
     res_vip    (ocf::heartbeat:IPaddr2):       Started kompira-server1
     res_httpd  (ocf::heartbeat:apache):        Started kompira-server1
     res_kompirad       (ocf::kompira:kompirad):        Started kompira-server1
     res_kompira_jobmngrd       (ocf::kompira:kompira_jobmngrd):        Started␣
↪kompira-server1
     res_lsyncd (lsb:lsyncd):   Started kompira-server1
 Master/Slave Set: ms_pgsql [res_pgsql]
     Masters: [ kompira-server1 ]
     Slaves: [ kompira-server2 ]
 Master/Slave Set: ms_rabbitmq [res_rabbitmq]
     Masters: [ kompira-server1 ]
     Slaves: [ kompira-server2 ]

Node Attributes:
* Node kompira-server1:
    + master-res_pgsql                  : 1000
    + master-res_rabbitmq               : 10
    + res_pgsql-data-status             : LATEST
    + res_pgsql-master-baseline         : 0000000007000090
    + res_pgsql-status                  : PRI
* Node kompira-server2:
    + master-res_pgsql                  : 100
```

(次のページに続く)

(前のページからの続き)

```
    + master-res_rabbitmq               : 5
    + res_pgsql-data-status             : STREAMING|ASYNC
    + res_pgsql-status                  : HS:async
```

crm_mon コマンドの出力において確認するべき点を示します。

- **Resource Group**

  アクティブ機のみで動作するリソースが表示されます。「Started <アクティブ機のホスト名>」と表示されていれば正常です。

- **Master/Slave Set**

  両方のサーバで動作するリソースが表示されます。Masters にアクティブ機側のホスト名、Slaves にスタンバイ機側のホスト名が表示されていれば正常です。

- **Node Attributes**

  PostgreSQL プロセスの詳細な状態が表示されています。正しくレプリケーションが行われている場合、res_pgsql-data-status の行にアクティブ機側には LATEST、スタンバイ機側には STREAMING|ASYNC と表示されます。

ライセンス登録

冗長構成の場合、アクティブ機とスタンバイ機、両方のサーバに対してそれぞれライセンスファイルを登録する必要があります。

Kompira ライセンス に記載された手順を実行し、各サーバに対してライセンスファイルを登録してください。

1.8.3 冗長構成の停止・起動

冗長構成で動作している Kompira の停止・起動方法について記述します。

はじめに、crm_mon コマンドを使用して 2 台のサーバのどちらがアクティブ状態として動作しているかを確認します。crm_mon コマンド中のリソース部分の表示で、「Started」および「Masters」と表示されているのがアクティブ状態のサーバとなります。

以降の説明では、kompira-server1 がアクティブ状態であると仮定して手順を記述します。

原則として、停止する場合はスタンバイ機を停止してからアクティブ機を停止、起動する場合はアクティブ機を起動してからスタンバイ機を起動という順序で行います。

これは、アクティブ機を先に停止するとスタンバイ機がアクティブ機に異常が起こったと判断してフェイルオーバー処理が行われてしまうためです。誤ってフェイルオーバーがされてしまった場合は、フェイルオーバー時の動作とフェイルバック の手順を参照してください。

冗長構成の停止

まず、セカンダリ (kompira-server2) の Pacemaker プロセスを停止します。

```
# pcs cluster stop
Stopping Cluster (pacemaker)...
Stopping Cluster (cman)...
```

サービスが停止したのを確認してから、プライマリ (kompira-server1) で同様の処理を行います。冗長構成の最後の一台を停止させるときは –force オプションが必要になります。

```
# pcs cluster stop --force
Stopping Cluster (pacemaker)...
Stopping Cluster (cman)...
```

これで Pacemaker/corosync によるリソースの監視が停止します。なお、pacemaker プロセスが停止している場合、crm_mon コマンドは実行できないことに注意してください。

プロセスだけでなくサーバー OS のシャットダウンを行う場合、上記処理は必要ありません。ただし、シャットダウンの場合もスタンバイ機のシャットダウンが完了してからアクティブ機のシャットダウンを行ってください。

冗長構成の起動

起動を行う場合は、停止とは反対の手順を実行します。はじめに、プライマリ (kompira-server1) の Pacemaker プロセスを起動します。

```
# pcs cluster start
Starting Cluster...
```

pacemaker プロセスが起動すると、pacemaker に登録されているリソースが順次起動します。crm_mon コマンドを実行し、全てのリソースが起動するまで待機してください。

リソースが起動したら、セカンダリ (kompira-server2) の Pacemaker プロセスを起動します。

```
# pcs cluster start
pcs cluster start
```

以上で冗長構成の起動が完了します。

プロセスだけでなくサーバー OS の起動から行う場合、上記処理は必要ありません。Pacemaker サービスは自動起動されるように設定されています。

アクティブ機を起動し、起動が完了したことを確認してからスタンバイ機の起動を行ってください。

1.8.4 フェイルオーバー時の動作とフェイルバック

アクティブ機で何らかの障害が発生すると、自動的にフェイルオーバーが実行されてスタンバイ機がアクティブ状態に昇格します。

以下はアクティブ状態であった kompira-server1 がシャットダウンした後に kompira-server2 で crm_mon コマンドを実行した場合の表示です。

```
$ crm_mon -A1
Last updated: Thu Jul 10 17:40:36 2014
Last change: Thu Jul 10 17:36:24 2014 via crm_attribute on kompira-server2
Stack: cman
Current DC: kompira-server2 - partition WITHOUT quorum
Version: 1.1.10-14.el6_5.3-368c726
2 Nodes configured
9 Resources configured


Online: [ kompira-server2 ]
OFFLINE: [ kompira-server1 ]

 Resource Group: webserver
     res_vip    (ocf::heartbeat:IPaddr2):       Started kompira-server2
     res_httpd  (ocf::heartbeat:apache):        Started kompira-server2
     res_kompirad       (ocf::kompira:kompirad):        Started kompira-server2
     res_kompira_jobmngrd       (ocf::kompira:kompira_jobmngrd):        Started␣
↪kompira-server2
     res_lsyncd (lsb:lsyncd):   Started kompira-server2
 Master/Slave Set: ms_pgsql [res_pgsql]
     Masters: [ kompira-server2 ]
     Stopped: [ kompira-server1 ]
 Master/Slave Set: ms_rabbitmq [res_rabbitmq]
     Masters: [ kompira-server2 ]
     Stopped: [ kompira-server1 ]

Node Attributes:
* Node kompira-server2:
    + master-res_pgsql                  : 1000
    + master-res_rabbitmq               : 10
    + res_pgsql-data-status             : LATEST
    + res_pgsql-master-baseline         : 000000000A000090
    + res_pgsql-status                  : PRI
```

kompira-server1 が OFFLINE と表示されており、各リソースは kompira-server2 上で動作していることが確認できます。

以降では、kompira-server1 が復旧可能だった場合、復旧不能だった場合に分けて手順を記述します。

サーバーが復旧可能だった場合

シャットダウンした kompira-server1 を正常に起動できた場合の手順です。

kompira-server1 を起動すると、状態は以下のようになります。

```
Last updated: Thu Jul 10 18:02:02 2014
Last change: Thu Jul 10 17:36:24 2014 via crm_attribute on kompira-server2
Stack: cman
Current DC: kompira-server2 - partition with quorum
Version: 1.1.10-14.el6_5.3-368c726
2 Nodes configured
9 Resources configured


Online: [ kompira-server1 kompira-server2 ]

 Resource Group: webserver
     res_vip    (ocf::heartbeat:IPaddr2):       Started kompira-server2
     res_httpd  (ocf::heartbeat:apache):        Started kompira-server2
     res_kompirad       (ocf::kompira:kompirad):    Started kompira-server2
     res_kompira_jobmngrd       (ocf::kompira:kompira_jobmngrd):    Started kompira-
↪server2
     res_lsyncd (lsb:lsyncd):   Started kompira-server2
 Master/Slave Set: ms_pgsql [res_pgsql]
     Masters: [ kompira-server2 ]
     Stopped: [ kompira-server1 ]
 Master/Slave Set: ms_rabbitmq [res_rabbitmq]
     Masters: [ kompira-server2 ]
     Slaves: [ kompira-server1 ]

Node Attributes:
* Node kompira-server1:
    + master-res_pgsql                  : -INFINITY
    + master-res_rabbitmq               : 5
    + res_pgsql-data-status             : DISCONNECT
    + res_pgsql-status                  : STOP
* Node kompira-server2:
    + master-res_pgsql                  : 1000
    + master-res_rabbitmq               : 10
    + res_pgsql-data-status             : LATEST
    + res_pgsql-master-baseline         : 000000000A000090
    + res_pgsql-status                  : PRI

Failed actions:
res_pgsql_start_0 on kompira-server1 'unknown error' (1): call=40, status=complete,␣
↪last-rc-change='Thu Jul 10 17:59:25 2014', queued=776ms, exec=0ms
```

kompira-server1 ではデータベースが起動しておらず、スタンバイ機として正常な状態にはなっていません。

スタンバイ機としてのセットアップを完了するためには、スタンバイ機で kompira パッケージに付属している sync_master.sh を使用します。sync_master.sh はアクティブ機のデータベースをスタンバイ機にコピーし、レプリケーションの設定を行った上でデータベースプロセスを起動します。

```
# /opt/kompira/bin/sync_master.sh
[2018-07-05 22:47:09] ****:␣
↪************************************************************
[2018-07-05 22:47:09] ****: Kompira-1.5.0:
[2018-07-05 22:47:09] ****: Start: Sync with the Master
...
[2018-07-05 22:47:15] INFO: Waiting for the resources to stabilize.
    RabbitMQ  | PostgreSQL                          | description
    #NA       | /DISCONNECT/#NA                     |
    #NA       | /DISCONNECT/#NA                     |
...
[2018-07-05 22:47:32] INFO: PostgreSQL on this node needs to be synchronized with the␣
↪Master.
[2018-07-05 22:47:32] INFO: RabbitMQ on this node is running as a Slave.
[2018-07-05 22:47:32] INFO: Enter the pacemaker in maintenance mode.
[2018-07-05 22:47:32] VERBOSE: run: pcs property set maintenance-mode=true
[2018-07-05 22:47:33] VERBOSE: run: rm -f /var/lib/pgsql/9.6/tmp/PGSQL.lock
[2018-07-05 22:47:33] VERBOSE: run: rm -rf /var/lib/pgsql/9.6/data.old
[2018-07-05 22:47:33] VERBOSE: run: mv -f /var/lib/pgsql/9.6/data /var/lib/pgsql/9.6/
↪data.old
[2018-07-05 22:47:33] VERBOSE: run: sudo -u postgres pg_basebackup -h ha-kompira-other␣
↪-U repl -D /var/lib/pgsql/9.6/data -X stream -P
33488/33488 kB (100%), 1/1 tablespace
[2018-07-05 22:47:35] INFO: Reset the failcount of RabbitMQ.
[2018-07-05 22:47:35] VERBOSE: run: crm_failcount -r res_rabbitmq -D
...
[2018-07-05 22:47:36] INFO: Waiting for the resources to stabilize.
    RabbitMQ  | PostgreSQL                          | description
    5         | STOP/DISCONNECT/-INFINITY           | RabbitMQ as demoted as a Slave.
    5         | STOP/DISCONNECT/-INFINITY           |
    5         | STOP/DISCONNECT/-INFINITY           |
    5         | STOP/DISCONNECT/-INFINITY           |
    5         | STOP/DISCONNECT/-INFINITY           |
    5         | HS:alone/DISCONNECT/-INFINITY       |
    5         | HS:alone/STREAMING|ASYNC/-INFINITY  |
    5         | HS:alone/STREAMING|ASYNC/100        |
    5         | HS:async/STREAMING|ASYNC/100        | PostgreSQL has demoted as a␣
↪Slave.
[2018-07-05 22:47:45] INFO: Resources stablized.
...
[2018-07-05 22:47:48] ****:
[2018-07-05 22:47:48] ****: Finish: Sync with the Master (status=0)
[2018-07-05 22:47:48] ****:␣
↪************************************************************
```

注釈: sync_master.sh は、実行時にデータベースに関連する全てのデータを /var/lib/pgsql/9.6/data.old に退避し、アクティブ機のデータを取得します。(ログファイルを含む)

sync_master.sh を実行後に crm_mon コマンドを呼び出すと、kompira-server1 の res_pgsql-data-status が STREAMING|ASYNC となったことが確認できます。

```
$ crm_mon -A1
Last updated: Thu Jul 10 19:03:25 2014
Last change: Thu Jul 10 19:03:16 2014 via crm_attribute on kompira-server2
Stack: cman
Current DC: kompira-server2 - partition with quorum
Version: 1.1.10-14.el6_5.3-368c726
2 Nodes configured
9 Resources configured


Online: [ kompira-server1 kompira-server2 ]

 Resource Group: webserver
     res_vip    (ocf::heartbeat:IPaddr2):   Started kompira-server2
     res_httpd  (ocf::heartbeat:apache):    Started kompira-server2
     res_kompirad       (ocf::kompira:kompirad):    Started kompira-server2
     res_kompira_jobmngrd       (ocf::kompira:kompira_jobmngrd):    Started kompira-
↪server2
     res_lsyncd (lsb:lsyncd):   Started kompira-server2
 Master/Slave Set: ms_pgsql [res_pgsql]
     Masters: [ kompira-server2 ]
     Slaves: [ kompira-server1 ]
 Master/Slave Set: ms_rabbitmq [res_rabbitmq]
     Masters: [ kompira-server2 ]
     Slaves: [ kompira-server1 ]

Node Attributes:
* Node kompira-server1:
    + master-res_pgsql                  : 100
    + master-res_rabbitmq               : 5
    + res_pgsql-data-status             : STREAMING|ASYNC
    + res_pgsql-status                  : HS:async
* Node kompira-server2:
    + master-res_pgsql                  : 1000
    + master-res_rabbitmq               : 10
    + res_pgsql-data-status             : LATEST
    + res_pgsql-master-baseline         : 0000000004000090
    + res_pgsql-status                  : PRI
```

サーバーが復旧不能だった場合

ハードウェア障害などによって、シャットダウンした機器を交換する必要がある場合の手順です。この場合、OS インストール直後のサーバを用意し、スタンバイ状態として導入します。

冗長構成環境ではアクティブ機とスタンバイ機それぞれにライセンスファイルを登録する必要があります。install.sh を実行後、license_info コマンドを使用してハードウェア ID の確認およびライセンスファイルの登録を行ってください。

```
$ tar zxf kompira-<version>-bin.tar.gz
$ cd kompira-<version>-bin
# ./install.sh

$ cp kompira_KP-EVALM100-0000000001.lic /var/opt/kompira/kompira.lic
$ cd /var/opt/kompira
$ chown apache:apache kompira.lic
$ /opt/kompira/bin/manage.py license_info

# ./setup_cluster.sh --primary --slave-mode
```

上記コマンドは、kompira-server1 をスタンバイ状態としてセットアップする場合の例です。

冗長構成環境ではアクティブ機とスタンバイ機それぞれにライセンスファイルを登録する必要があるため、setup_cluster.sh を実行する前にライセンスファイルの登録手順を行う必要があります。

setup_cluster.sh の実行において kompira-server1 ではなく kompira-server2 を追加する場合は、–primary オプションの代わりに–secondary オプションを使用します。

setup_cluster.sh の処理が完了したら、状態確認 を参考に動作確認を行ってください。

参考:

Kompira ライセンス

1.8.5 setup_cluster.sh のオプション

setup_cluster.sh の動作オプションを以下に示します。

オプション	デフォルト値	説明
--primary	true(指定あり)	プライマリとしてセットアップを開始します。
--secondary	false(指定なし)	セカンダリとしてセットアップを開始します。
--heartbeat-device=DEVICE		ハートビート用ネットワークデバイスを指定します。
--master-mode		アクティブ状態としてセットアップします。
--slave-mode		スタンバイ状態としてセットアップします。
--without-vip		VIP 無しの構成でセットアップします。(別途ロードバランサ等による ACT/SBY 監視とアクセス振り分けの設定が必要です)
--without-jobmanager		ジョブマネージャ無しの構成でセットアップします。
--hostname-prefix=PREFIX_NAME	kompira-server	設定するホスト名のプレフィックスを指定します。
--skip-hostname-setup		ホスト名の設定を行いません。(事前にホスト名を設定して名前解決可能な状態にしておく必要があります)
--heartbeat-netaddr=NETWORK_ADDRESS	192.168.99.0	ハートビート用インタフェースに設定するネットワークアドレスを指定します。
--manual-heartbeat		ハートビート用ネットワークを手動設定します。--heartbeat-primary および--heartbeat-secondary の指定が必要です。--heartbeat-netaddr の指定は無視します。ハートビートはユニキャストモードで動作します。
--heartbeat-primary=		プライマリの IP アドレスを指定します。

第 2 章

操作ガイド

著者　Kompira 開発チーム

2.1 はじめに

Kompira が提供する Web ユーザーインターフェースによって Kompira の機能を利用するための情報について記述します。

2.2 ログインとログアウト

以下の URL にアクセスすることで、Kompira のログイン画面にアクセスすることができます。

```
https://<Hostname or ipaddress of Kompira server>/
```

Kompira ログイン画面からユーザー名とパスワードを入力して、ログインしてください。

デフォルトで使用可能なユーザー一覧は ユーザー管理 を参照してください。

ログイン後、メニュー上部よりホーム、ファイルシステム、タスク一覧、インシデント一覧、プロセス管理 、スケジューラ 、設定 、ヘルプに移動することができます。

画面右上部には現在ログイン中のユーザー名が表示されており、ここからログアウトを行うことができます。

注釈: ログイン情報はブラウザの Cookie に保存されます。ログイン情報 Cookie の有効期限は 2 週間となっており、有効期限を過ぎた場合は再度ログイン処理を行っていただく必要があります。

2.3 Kompira ファイルシステム

ジョブフロー定義やノード情報など、Kompira で扱う情報定義は Kompira オブジェクトとして、Kompira ファイルシステム上で一元的に管理されます。

以下では、Kompira オブジェクトの種類によらない設定、値について説明します。

2.3.1 オブジェクトの名称

Kompira オブジェクトの名称は、以下の規則内で自由に命名することができます。

- アルファベット、数字、アンダーバー ("_")、日本語を使うことができる
- 先頭の文字は数字以外でなければいけない
- アルファベットの大文字と小文字は区別される
- オブジェクトの名称の長さは 128 文字以内でなければいけない
- 絶対パスの長さは 1024 文字以内でなければいけない

2.3.2 オブジェクトのプロパティ

全ての Kompira オブジェクトにはプロパティが存在しており、オブジェクトの所有者もしくはルート権限を持つユーザーはプロパティの各項目を編集することができます。

プロパティで設定できる項目の一覧を以下に示します。

フィールド	説明
表示名	オブジェクトの表示に使用する名称。(オブジェクトの名称とは異なる)
説明	オブジェクトについての説明。
所有者	オブジェクトの所有者。
ユーザーパーミッション	ユーザーに与えるアクセス許可リスト。
グループパーミッション	グループに与えるアクセス許可リスト。

ユーザーパーミッション、グループパーミッションの項ではユーザー、グループごとにアクセス許可を設定することができます。

全てのユーザーに対して共通のアクセス許可を設定したい場合、全てのユーザーが所属する other グループを利用するとよいでしょう。

注釈: プロパティを編集できるのはアクセス許可の設定に関わらず、常にオブジェクトの所有者もしくはルート権

限を持つユーザーのみです。書き込み権限を持つユーザーはオブジェクトの内容を編集することはできますが、プロパティは編集できないことに注意してください。

2.3.3 オブジェクトのアクセス許可

全ての Kompira オブジェクトにはアクセス許可の設定が存在しています。

以下に Kompira オブジェクトが持つアクセス許可種別の一覧を示します。

許可種別	説明
読み取り	オブジェクトの内容を読み取ることができます。 権限がないオブジェクトのパスに移動しようとするとエラーとなります。
書き込み	オブジェクトの内容を編集することができます。 ディレクトリまたはテーブルオブジェクトにおいて書き込み許可を持たない場合、新規にオブジェクトを追加することはできません。
実行	オブジェクトを実行することができます。 実行可能なオブジェクト (ジョブフロー、スクリプトジョブ) でのみ有効な権限種別です。

ルート権限を持つユーザーには、明示的に指定されていない場合でも全てのアクセスが許可されています。

オブジェクトのアクセス許可設定は オブジェクトのプロパティ から編集することができます。

注釈: ディレクトリまたはテーブルオブジェクトにオブジェクトを追加する場合、アクセス許可の設定は継承されません。

2.4 Kompira オブジェクト

Kompira ファイルシステム上で作成されるオブジェクトには、ジョブフローやノード情報など様々な種類のものがあります。これらは、やはり Kompira ファイルシステム上のオブジェクトである型オブジェクトによって規定されています。あらかじめ用意されている型オブジェクトは、/system/types で一覧を参照することができます。

現バージョンでは、標準で以下に示す型オブジェクトが定義されています。

型名	説明
TypeObject (型オブジェクト)	型オブジェクトを既定するためのオブジェクトです。型オブジェクトを作成すると、作成した型のオブジェクトを作成できるようになります。
Directory (ディレクトリ)	複数のオブジェクトを格納することができるオブジェクトです。
Table (テーブル)	同じ型の複数のオブジェクトを格納することができるオブジェクトです。
Jobflow (ジョブフロー)	ジョブフローの記述と実行をすることができるオブジェクトです。
ScriptJob (スクリプトジョブ)	スクリプトの記述と実行をすることができるオブジェクトです。
NodeInfo (ノード情報)	サーバの IP アドレスや SSH のポート番号など、ノードを特定するための情報を格納することができるオブジェクトです。
AccountInfo (アカウント情報)	リモートログインに必要なアカウント情報を格納することができるオブジェクトです。
Environment (環境変数)	key-value 形式の環境情報を格納することができるオブジェクトです。
Template (テンプレート)	タスクなどのメッセージで使用するテンプレートテキストを格納することができるオブジェクトです。
MailTemplate (メールテンプレート)	メール送信で使用するテンプレートテキストを格納することができるオブジェクトです。
Wiki (Wiki ページ)	Creole 形式の Wiki ページを作成をすることができるオブジェクトです。
AttachedFile (添付ファイル)	任意のファイルを保存することができるオブジェクトです。
Form (フォーム)	ユーザ入力フォームを作成することができるオブジェクトです。
Config (設定)	設定フォームを作成することができるオブジェクトです。
Channel (チャネル)	メッセージを格納することができるキューを持つオブジェクトです。メッセージの送受信に利用することができます。
MailChannel (メールチャネル)	IMAP サーバからのメールを取り込むことができるチャネルです。
Repository (リポジトリ)	バージョン管理システムと連携するための情報を定義するオブジェクトです。
Library (ライブラリ)	ジョブフローから呼び出し可能な Python のライブラリを定義するオブジェクトです。
Virtual (仮想オブジェクト)	仮想オブジェクトを既定するためのオブジェクトです。プロセス一覧 (/process)、タスク一覧 (/task) は仮想オブジェクトとして定義されています。一般に使用することはありません。
Realm (管理領域)	ジョブマネージャが管理する領域を既定するためのオブジェクトです。管理領域一覧オブジェクト以下で作成されることを前提としており、通常のディレクトリやテーブル以下に作成することはありません。
License (ライセ	ライセンスファイルを登録するためのオブジェクトです。システムで使用する特殊なオブ

各型オブジェクトはそれぞれのフィールドとその種別を定義しています。どのようなフィールドが定義されているかは、組み込み型オブジェクト を参照してください。

以下では一部の代表的な Kompira オブジェクトをとりあげ、その使用方法を記述します。

2.4.1 ディレクトリ

ディレクトリは、複数の異なる型のオブジェクトを格納できる Kompira オブジェクトです。

ディレクトリオブジェクトから、以下の操作を行うことができます。

操作	説明
新規作成	オブジェクトを作成します。作成する際は型オブジェクトを指定する必要があります。
閲覧	格納しているオブジェクトのページに移動します。
編集	オブジェクトの内容を編集します。
名前の変更	オブジェクトの名称を変更します。プロパティの表示名がオブジェクトの名称と等しい場合、表示名も同時に変更されます。
移動	オブジェクトを移動します。
コピー	オブジェクトを別のディレクトリにコピーします。
削除	オブジェクトを削除します。
エクスポート	選択したオブジェクトをエクスポートします。オブジェクト未選択時は、ディレクトリ以下のすべてのオブジェクトをエクスポートします。
インポート	ファイルから選択したディレクトリにオブジェクトをインポートします。ディレクトリ未選択時は現在のディレクト以下にオブジェクトをインポートします。
プロパティ	オブジェクトのプロパティを編集します。

新規作成・編集を行う場合、Kompira オブジェクトに対応した編集画面に移動します。

移動、コピー、削除は複数のオブジェクトを選択した上で一括で行うことができます。

オブジェクトのインポートとエクスポートは、当該ディレクトリの所有者、もしくはルート権限を持つユーザのみ可能です。

2.4.2 テーブル

テーブルはディレクトリと同様に複数のオブジェクトを格納できる Kompira オブジェクトです。ただし、格納できるオブジェクトの型が 1 つに限定されている点がディレクトリとは異なります。

テーブルオブジェクトを作成する場合、初めに型オブジェクトと、その型オブジェクト内のフィールドを選択しま

す。作成されたテーブルオブジェクトでは、ディレクトリオブジェクトで表示される情報に加えて作成時に選択したフィールド情報が表示されます。

テーブルオブジェクトを使用することで、格納されているオブジェクトのフィールドを一括で閲覧することができます。

2.4.3 ジョブフロー

ジョブフローオブジェクトから、ジョブフローの記述と実行ができます。ジョブフローの記述方法については、*Kompira* チュートリアル を参照してください。また、ジョブフロー言語の詳細については、*Kompira* ジョブフロー言語リファレンス が参考になります。

ジョブフローの実行

ジョブフローを記述し、保存すると、ジョブフローの実行ボタンが有効になります。実行ボタンを押すと、ジョブフローの実行を開始し、プロセス詳細画面に移ります。

注釈: 記述したジョブフローに文法エラーがある場合や、Kompira エンジンが停止している場合は、ジョブフローの実行ボタンが無効になります。

ジョブフロー実行には以下のオプションが選択できます。

オプション名	内容
ステップモード	ジョブフローをデバッグする際に使用するモードです。コマンドの実行前にジョブフローが一時停止し、実行コマンドの内容を確認できます。
チェックポイントモード	ジョブフローの実行状態を随時保存するモードです。ジョブフロー実行中に万が一 Kompira サーバが異常終了した場合、保存されたチェックポイントの状態から、ジョブフロープロセスを再開できるようになります。
監視モード	ジョブフローの実行監視モードを指定します。ジョブフローの完了時や異常終了時に、ジョブフローを実行したユーザのメールアドレス宛にメールを送信して通知します。

2.4.4 スクリプトジョブ

スクリプトジョブを作成すると Bash や Perl、Ruby、Python などの言語で記述されたスクリプトをリモートのサーバ上で実行することができます。

スクリプトの編集

編集ボタンを押すと、スクリプトの編集画面に移ります。ソースのテキストエリアに、実行させるスクリプトを記述してください。

Linux など、Unix 系 OS でスクリプトを実行させる場合、スクリプトの先頭行にシバン (shebang) を記述してください。

例

```
#!/bin/bash
echo hello
```

Windows 系 OS でスクリプトを実行させる場合、拡張子を指定する必要があります。スクリプトの種類に応じて、以下の拡張子を指定してください。

スクリプト	拡張子
バッチファイル	bat
VBScript	vbs
JScript	js
PowerShell スクリプト	ps1

スクリプトの実行方法

実行ボタンを押すと、スクリプトの実行が開始され、プロセス詳細画面に移ります。スクリプトの実行が完了すると、終了ステータス、標準出力、標準エラー出力の結果が、コンソールに表示されます。

実行ボタンの右側にあるテキストフィールドにて、コマンドライン引数を入力することができます。空白で区切ることで複数のコマンドライン引数を渡せます。

実行ノードは、スクリプトが実行されるリモートのサーバを指定します。指定しなかった場合は、ジョブマネージャが動作しているローカルサーバ上でスクリプトが実行されます。

実行アカウントは、リモートサーバにログインする場合のアカウント情報を指定します。

バージョン 1.4.0 で追加: スクリプトジョブをブラウザ上から直接実行できる機能を追加

2.4.5 リポジトリ

リポジトリオブジェクトを作成することで、Kompira のディレクトリと分散型バージョン管理システム (DVCS) 上のリポジトリを同期させることが可能です。リモートリポジトリから指定した Kompira ディレクトリにオブジェクトを取り込んだり、逆に、作成した Kompira オブジェクトをリモートリポジトリ上に保存したりすることができます。これによって、Kompira のジョブフローやスクリプトジョブなどのバージョン管理が可能になります。また、複数の Kompira 間でのジョブフローの共有なども簡単に行えるようになります。

リポジトリ設定方法

リポジトリオブジェクトの編集画面で、以下の各項目を設定し、保存してください。

設定項目	内容
URL	リモートリポジトリの URL を指定します。
リポジトリ種別	リモートリポジトリの種別を指定します。(現バージョンでは mercurial のみ選択可能)
ポート番号	リポジトリサーバのポート番号がデフォルトと異なる場合に指定します。
ユーザ名	リモートリポジトリにアクセスするアカウントのユーザ名を指定します。
パスワード	リモートリポジトリにアクセスするアカウントのパスワードを指定します。
ディレクトリ	同期対象となる Kompira のディレクトリオブジェクトを指定します。

注釈: リモートリポジトリは、あらかじめ作成しておく必要があります。

初期化

必要なリポジトリ設定項目が入力されると、リポジトリ画面から初期化ボタンが有効化されます。初期化ボタンを押すと、Kompira サーバ上のローカルリポジ トリが初期化され、リモートリポジトリの内容が指定した Kompira ディレクトリオブジェクトに取り込まれます。

プッシュ

プッシュボタンは、初期化後に有効になります。Kompira ディレクトリの更新情報をリモートリポジトリ側に送信する場合、プッシュボタンを押します。

プル

プルボタンは、初期化後に有効になります。リモートリポジトリ側の更新情報を Kompira 側に反映させる場合、プルボタンを押します。

2.5 プロセス管理

ジョブフローおよびスクリプトジョブの実行状態を管理するためのプロセスについて記述します。

プロセスが作成されるのは、ジョブフローおよびスクリプトジョブが実行されたときです。実行の開始については ジョブフロー および スクリプトジョブ を参照してください。

2.5.1 プロセス一覧

プロセス一覧画面では、実行中もしくは過去に実行されたプロセスの一覧を確認することができます。

デフォルトでは「実行中のプロセス」が選択された状態で、プロセスのステータスが NEW(新規)、READY(実行可能)、RUNNING(実行中)、WAITING(入力/コマンド完了待ち) のプロセスが表示されます。

プロセスのステータスが DONE(実行完了)、もしくは ABORTED(異常終了) のプロセスは既に終了したプロセスです。これらを確認したい場合は「全てのプロセス」を選択してください。

既に実行が終了しているプロセスについては、一覧画面より削除をすることができます。

注釈: 一般ユーザーは自身が実行したプロセスのみ閲覧することができます。ルート権限を持つユーザーは全てのプロセスを閲覧することができます。

2.5.2 プロセス詳細

プロセス詳細画面では、プロセスの実行状況の確認と制御を行うことができます。

プロセス詳細画面中のボタン、フォームについて以下に記述します。

中止 (Terminate)

プロセスの実行を中止します。中止が選択されたプロセスはステータスが ABORTED(異常終了) となり、再開することはできません。

実行中の子プロセスが存在する場合、子プロセスのステータスも ABORTED(異常終了) となります。

停止 (Suspend)

プロセスの実行を一時的に停止します。

実行中の子プロセスが存在する場合、子プロセスも停止状態となります。

続行 (Resume)

停止状態のプロセスを再開します。

停止中の子プロセスが存在する場合、子プロセスも再開します。

コンソール

プロセス実行時の出力です。

ジョブフローの場合、print 文のメッセージ、リモートコマンドの実行結果、エラー時のスタックトレース等が表示されます。

スクリプトジョブの場合、スクリプトの終了ステータス、標準出力、標準エラー出力が表示されます。

注釈: コンソールバッファの最大サイズは 64KB に制限されています。64KB 以上の出力があった場合、確認できるメッセージは最新の 64KB 分のみであることに注意してください。

ジョブフロー/スクリプト

実行されたジョブフローおよびスクリプトが表示されます。

ジョブフローの場合は現在実行中の行も合わせて表示されます。

子プロセス一覧

ジョブフローが実行された場合のみ表示される画面で、子プロセスの一覧が確認できます。

子プロセスは、fork や pfor 構文を使用して子プロセスを作成するジョブフローを実行した時に作成されます。

2.6 スケジューラ

Kompira 上に作成したジョブフローおよびスクリプトジョブをスケジューラに登録することで、定期的にジョブを実行させることができます。

スケジューラで設定できる項目の一覧を以下に示します。

フィールド	デフォルト値	説明
スケジュール名	なし	スケジュールの名称
説明	なし	スケジュールについての説明
ユーザー		ジョブの実行ユーザー
ジョブ		スケジューラにより実行されるジョブ
年	*	スケジュールを実行する年 (4 桁の数字で指定)
月	*	スケジュールを実行する月 (1-12 を指定)
日	*	スケジュールを実行する日 (1-31 を指定)
ISO 週番号	*	スケジュールを実行する週 (1-53 を指定) ISO8601 で定められた週番号
曜日名もしくは曜日番号	*	スケジュールを実行する曜日 (0(月曜日)-6(日曜日)、または mon,tue,wed,thu,fri,sat,sun を指定)
時	*	スケジュールを実行する時 (0-23 を指定)
分	*	スケジュールを実行する分 (0-59 を指定)
スケジュールの無効化	false (チェックなし)	true の場合 (チェックがつけられている場合)、ジョブを実行しない

2.6.1 日時設定フィールドの書式

日時設定のフィールドには、以下のように Unix の cron と同じ書式を用いることができます。

書式	フィールド	説明
*	全て	各値ごとに発火します。
*/a	全て	a 間隔ごとに発火します。
a-b	全て	値が a から b の範囲にあるときに毎回発火します。
a-b/c	全て	値が a から b の範囲にあるときに、c 間隔毎に発火します。
xth y	日 (Day)	その月の x 番目の y 曜日に発火します。
last x	日 (Day)	その月の最終の x 曜日に発火します。
last	日 (Day)	その月の最終日に発火します。
x,y,z	全て	条件 x か y か z に発火します（上記の書式の任意の組み合わせが可能）

注釈: 上記書式において、’,’, ’/’, ’-’ の隣に空白を入れないように注意してください。

例 1: 毎年 12 月の最初の月曜日と最終金曜日の 0 時 0 分に発火:

```
Month: 12
Day: 1st mon,last fri
```

例 2: 2012 年の 4 月と 8 月の 15-20 の 12 時 30 分に発火:

```
Year: 2012
Month: 4,8
Day: 15-20
Hour: 12
Minute: 30
```

例 3: 平日に 1 時間毎に発火:

```
Day of week: mon-fri
Hour: *
```

例 4: 毎年 1 月 1 日 0 時 0 分に発火:

```
Year: *
```

2.7 設定

Kompira 画面上部の「設定」より行える各種設定について記述します。

2.7.1 ユーザー管理

Kompira 上に登録されているユーザーの一覧を確認することができます。

初期状態で用意されているユーザーを以下に示します。

ユーザー名	パスワード	説明
guest	guest	ゲスト用ユーザーです。
root	root	ルートユーザーです。
admin	admin	管理用ユーザーです。アクセス許可設定とは無関係に全てのオブジェクトに対してアクセス可能です。デフォルトでは無効のユーザーとなっています。

ユーザーを新規作成すると/home/<ユーザー名>/ ディレクトリが作成され、自動的にホームとして設定されます。

一般ユーザーは自身のユーザー情報のみ編集可能です。全てのユーザー情報を編集できるのはルート権限を持つユーザーに限定されています。

ユーザーごとに設定できる項目の一覧を以下に示します。

フィールド	説明
ユーザー名	システムでユーザーの識別に使用する名称
姓	ユーザーの姓
名	ユーザーの名
E メール	ユーザーのメールアドレス
グループ	ユーザーが所属するグループ
有効	false(チェックなし) の場合、ユーザーのログインを許可しない
ホーム	ユーザーがログインしたときに最初に表示するページ
環境変数	ジョブフロー実行時、自動的に変数として読み込まれる環境変数オブジェクト

注釈: guest、root、admin ユーザーを削除することはできません。

2.7.2 グループ管理

Kompira 上に登録されているグループの一覧を確認することができます。

初期状態で用意されているグループを以下に示します。

グループ名	説明
other	Kompira 上の全てのユーザが属するグループ
wheel	ルート特権を持つユーザが属するグループ

グループ情報を編集できるのはルート権限を持つユーザーに限定されています。

注釈:

- other と wheel グループを削除することはできません。
- other グループに所属するユーザーの判定は設定にかかわらず、上記のとおりとなります。すなわち、other グループに所属するユーザ設定は無視されます。

2.7.3 管理領域設定

管理領域とは、ジョブマネージャごとに管理するネットワーク領域のことです。

Kompira で複数のジョブマネージャを使用する際、ジョブマネージャ A は 192.168.1.*へのアクセス、ジョブマ

ネージャ B は 192.168.2.*へのアクセスというように、各ジョブマネージャが管理する領域を定めることができます。

管理領域ごとに設定できる項目の一覧を以下に示します。

フィールド	説明
表示名	管理領域の表示名を指定します。
説明	管理領域の説明を記述します。
無効化	対象管理領域を一時的に無効にする場合にセットします。
範囲	管理領域の範囲を IP アドレスやホスト名で指定します。複数指定することができ、またワイルドカード (*) も使用可能です。

管理領域情報を編集できるのはルート権限を持つユーザーに限定されています。

デフォルトでは、default という管理領域があり、その範囲は'*' と設定されています。この場合、管理領域が default のジョブマネージャで全てのリモートコマンドを実行します。

ジョブマネージャが 1 台のみの構成で Kompira を使用する場合、もしくはジョブマネージャごとに管理領域を設定しなくてもよいという場合、管理領域設定を変更する必要はありません。

ジョブマネージャの状態確認

管理領域設定画面では、各管理領域に登録されているジョブマネージャの動作状況を確認することができます。

ジョブマネージャ状態として、以下の項目が表示されます。

値	説明
ホスト名	ジョブマネージャが動作しているホストの名称
プロセス ID	ジョブマネージャプロセス (kompira_jobmngrd) のプロセス ID
バージョン	ジョブマネージャの Kompira バージョン
ステータス	ジョブマネージャの動作状況 (' 動作中' もしくは' ダウン')

ステータスが「動作中」となっていれば、ジョブマネージャは Kompira と通信可能で、リモートコマンドが実行できる状態です。

2.7.4 システム設定

システム設定画面では、Kompira システム全体の設定を行います。

設定項目の一覧を以下に示します。

項目名 (キー名)	説明
サーバ URL (serverUrl)	Kompira サーバの URL
管理者メールアドレス (adminEmail)	Kompira サーバの管理者のメールアドレスを設定します。メール送信時に、from アドレスのデフォルト値として使われます。
正常終了時メールテンプレート (doneMailTemplate)	ジョブフローが正常に終了した場合に送信するメールのテンプレートを選択します。
異常終了時メールテンプレート (abortMailTemplate)	ジョブフローが異常終了した場合に送信するメールのテンプレートを選択します。
SMTP サーバ名 (SMTPServer)	メール送信時の SMTP サーバ名を設定します。省略した場合は、localhost の SMTP サーバが使われます。
SMTP ポート番号 (SMTPPort)	メール送信時の SMTP サーバの接続ポート番号を指定します。デフォルトは 25 です。
SMTP ユーザ名 (SMTPUser)	SMTP サーバの接続に認証が必要な場合、ユーザー名を設定します。
SMTP パスワード (SMTPPassword)	SMTP サーバの接続に認証が必要な場合、パスワードを設定します。
SMTP TLS 使用 (SMTPUseTLS)	SMTP サーバへ TLS を用いて接続する場合はチェックします。
SMTP SSL 使用 (SMTPUseSSL)	SMTP サーバへ SSL (SMTPS) を用いて接続する場合はチェックします。

注釈: システム設定 (/system/config) は設定型オブジェクト (設定 *(Config)*) ですので、ジョブフローから設定のデータ辞書に対して、キー名で参照することが可能です。

2.7.5 スタートアップジョブフロー

スタートアップディレクトリ (/system/startup) には、Kompira サーバ起動時に自動的に実行を開始するスタートアップジョブフローを登録することができます。複数のジョブフローが登録されている場合、名前順に実行されます。スタートアップディレクトリの下には、ジョブフロー以外のオブジェクトも作成することができます。スタートアップディレクトリの下にサブディレクトリを作成し、その中にジョブフローを登録した場合、それらのジョブフローは実行されません。スタートアップジョブフローは、引数を持ちません。また、root ユーザーで実行されます。

2.7.6 ライセンス管理

Kompira のライセンスを確認することができます。

ライセンス管理画面で確認できる項目の一覧を以下に示します。

フィールド	説明
ライセンス ID	ライセンスファイルの固有 ID
エディション	ライセンスの種類
ハードウェア ID	Kompira サーバのハードウェア固有 ID
有効期限	ライセンスの有効期限
登録済みノード数	ジョブフローから接続したことのあるノードの数 リセットを選択することで接続の履歴を削除する
ジョブフロー数	オブジェクトとして登録されているジョブフローの数
スクリプト数	オブジェクトとして登録されているスクリプトジョブの数
使用者	ライセンスの使用者
署名	ライセンスファイル署名

登録済みノード数、ジョブフロー数、スクリプト数の欄にはライセンスに応じた上限数が合わせて表示されます。

ライセンスファイルが未登録の場合、Kompira は仮ライセンスとして動作します。仮ライセンスは Kompira インストールから 1 週間の間自由にお使い頂けるライセンスです。

ライセンスの登録

ライセンス管理画面右側の編集ボタンを押すと、ライセンスファイルのアップロード画面に移動します。

「ファイルを選択」を押してライセンスファイルを登録後、保存ボタンを押すことでライセンス登録が完了します。

注釈: ライセンスファイルは/var/opt/kompira/kompira.lic に保存されます。上記パスにライセンスファイルを直接配置すれば、ライセンス管理画面にアクセスすることなくライセンスの登録をすることが可能です。

2.8 トラブルシューティング

Kompira をブラウザから操作してエラーが発生した場合の原因と対処方法を記述します。

2.8.1 「Jobflow の数が制限を超えました。」「ScriptJob の数が制限を超えました。」と表示される

Kompira で作成できるジョブフロー、スクリプトジョブオブジェクトの数は、ライセンスによる制限があります。

ライセンスが定める制限数を超えてオブジェクトを作成しようとすると、エラーメッセージが表示されて作成に失敗します。

ライセンス管理ページより、オブジェクトの数を確認してください。

2.8.2 「Kompira エンジンが停止しています」と表示される

kompirad プロセスが停止している場合に表示されるメッセージです。

/var/log/kompira 以下のログファイルを確認の上、kompirad プロセスの起動を行ってください。

参考:

Kompira デーモンの起動・停止・状態確認, *Kompira* ログ

2.8.3 「データベース接続エラーです」と表示される

データベースに接続できない場合に表示されるメッセージです。

下記コマンドにより、データベースプロセスの状態確認と再起動処理を行なうことができます。

RHEL/CentOS 7 系の場合

```
# systemctl status postgresql-9.6.service
# systemctl restart postgresql-9.6.service
```

RHEL/CentOS 6 系の場合

```
# /etc/init.d/postgresql-9.6 status
# /etc/init.d/postgresql-9.6 restart
```

注釈: Amazon Linux の場合、コマンド名が postgresql96 となります。

2.8.4 「内部エラーです」と表示される

Kompira 内部で予期しないエラーが発生した際に表示されます。

/var/log/kompira 以下のログファイルを確認の上、 support@kompira.jp までご連絡ください。

2.8.5 kompira_dump.sh による情報収集とサポートへの問合せ

Kompira 上の問題を解決するために各種ログファイルや設定ファイルなど、様々な情報の確認が必要になる場合があります。

Kompira サーバ上で /opt/kompira/bin/kompira_dump.sh を root 権限で実行すると、以下のように問題の解決に役立つ情報を自動的に収集します。なお、データベースのダンプを含むためサイズが大きくなる場合があります。スクリプトを実行する場所に十分に空き容量があることを確認してください。

```
$ sudo /opt/kompira/bin/kompira_dump.sh
2014-11-18 15:18:52 # mkdir /home/ec2-user/kompira_dump-20141118-151852
###
### kompira_dump ver 1.0.0
### dump started: 2014-11-18 15:18:52
###
========== system ==========
2014-11-18 15:18:52 # mkdir /home/ec2-user/kompira_dump-20141118-151852/system
2014-11-18 15:18:52 # cp -a /etc/os-release /etc/system-release ./
2014-11-18 15:18:52 # printenv
2014-11-18 15:18:52 # who -aH
    :
    :
    :
========== kompira ==========
2014-11-18 15:19:09 # mkdir /home/ec2-user/kompira_dump-20141118-151852/kompira
2014-11-18 15:19:09 # /opt/kompira/bin/kompirad --version
2014-11-18 15:19:09 # /opt/kompira/bin/manage.py license_info
2014-11-18 15:19:10 # /opt/kompira/bin/manage.py dumpdata -a
2014-11-18 15:19:16 # cp -a /opt/kompira/kompira.conf ./
2014-11-18 15:19:16 # cp -a /var/opt/kompira/kompira.lic ./
2014-11-18 15:19:16 # tar -cf - /var/log/kompira
tar: Removing leading `/' from member names
---------- kompira ----------
###
### dump finished: 2014-11-18 15:19:16
###
compressing...
/home/ec2-user/kompira_dump-20141118-151852.tar.gz
```

最後の行に収集結果をまとめたファイルが表示されます（上の例では kompira_dump-20141118-151852.tar.gz）ので、問題の状況と共にこのファイルを添付して support@kompira.jp までお問い合わせください。

なお、この .tar.gz ファイルは暗号化などの処理は行なっていないため、送付時にはお客様のセキュリティポリシーにしたがって必要な処理を行ってください。

収集しない項目

kompira_dump.sh では以下のような秘密情報は収集しません。

- Kompira サーバに設定されたアカウント・パスワード情報

収集する項目

kompira_dump.sh では以下のような情報を収集します。

- システム情報
  - プロセス情報 (ps ,top など)
  - サービス情報 (service, chkconfig など)
  - インストール済みパッケージ情報 (yum, rpm ,pip など)
  - カーネル情報 (sysctl, lsmod, /proc/{version,*info,*stat} など)
  - ログファイル (/var/log/{dmesg,messages} など)
- ネットワーク情報
  - インターフェース情報 (ip link, ip addr ,ip route など)
  - ファイアーウォール情報 (iptables -L など)
  - ネットワーク状態 (netstat, traceroute など)
- Apache に関する情報
  - サービス状態 (service httpd status など)
  - ログファイル (/var/log/httpd/)
  - 設定ファイル (/etc/httpd)
- RabbitMQ に関する情報
  - サービス状態 (service rabbitmq-server status)
  - ログファイル (/var/log/rabbitmq/)
- PostgreSQL に関する情報
  - サービス状態 (service postgresql-<pgver> status)
  - ログファイル (/var/lib/pgsql/<pgver>/data/{pg_log,pgstartup.log})
- Kompira に関する情報
  - バージョン (kompirad –version)

- ライセンス情報 (manage.py license_info など)
- データベースのダンプ (manage.py dumpdata -a)
- 設定ファイル (/opt/kompira/kompira.conf)
- ログファイル (/var/log/kompira/)

注釈: Kompira データベースのダンプを含むため **Kompira** オブジェクトやジョブフロー上に記載されたノード情報、アカウント・パスワード情報は含まれる ことに注意してください。

第 3 章

Kompira チュートリアル

著者　Kompira 開発チーム

3.1 はじめに

このチュートリアルでは、Kompira ジョブフロー言語の基本的な書き方や仕様についてざっと紹介します。実際に Kompira 上でジョブフローを作成しながら学ぶことで、リモートノードの制御方法や Kompira オブジェクトの利用方法が理解できることでしょう。

Kompira 標準オブジェクトの仕様については、*Kompira 標準ライブラリ* を参照してください。また *Kompira ジョブフロー言語リファレンス* にはより形式的な言語の定義が書いてあります。

このチュートリアルは Kompira の機能を網羅的に紹介しているわけではありません。しかしこのチュートリアルを読むことで、Kompira の特筆すべき機能や特徴を学び、簡単なジョブフローであれば読み書きできるようになるでしょう。

3.2 ジョブフローを動かす

3.2.1 Hello World

最初のジョブフローは、コンソールに "Hello World" を表示する単純なものです。

```
print("Hello World")
```

このジョブフローを実行するとコンソールに以下のように出力されるはずです。

```
Hello World
```

注釈: ジョブフローに文法エラーがあると、保存しても実行できない状態になります。実行ボタンが押せない場合は、ジョブフローのエラーを修正して再度保存してください。

Kompira のジョブフロー言語は、1 つの処理を表す **ジョブ** が基本的な実行の単位となります。

上の例では `print()` がジョブフローの **組み込みジョブ** のひとつで、丸括弧内に引数として与えられた文字列をコンソールに出力します。詳しくは *print* を参照してください。

3.2.2 コメントの書き方

ジョブフローではハッシュ文字 `#` から行末までがコメントとなります。コメントは行の先頭にも、ジョブの後にも書くことができます。ただし、文字列中に現れるハッシュ文字は対象外です。

```
# これはコメントです
print("# これはコメントではありません")  # これはコメント
```

3.2.3 コマンドを実行する

実行したいコマンドを文字列として **[** と **]** の中に記述することで、コマンドとして実行することができる **実行ジョブ** になります。

注釈: `[]` の中が文字列であれば実行すべきコマンドとして解釈しますので、コマンドラインの文字列を代入した変数を `[]` に記述して実行させることも可能です。(変数への代入については後ほど説明します)

コマンドの実行結果を表示する例を以下に示します。:

```
['whoami'] ->
print($RESULT)
```

このジョブフローを実行すると `whoami` コマンドを実行し、その結果(標準出力)が `print()` ジョブでコンソールに出力されます。通常はコンソールに以下のように表示されるはずです。:

```
[localhost] local: whoami

kompira
```

注釈: 特に指定しないとコマンドはジョブマネージャが動作しているホスト上で `kompira` アカウントで実行されるため、 `whoami` コマンドの実行結果として `kompira` と表示されています。

[localhost] local: で始まる行はどのノードでどんなコマンドを実行したかを示しています。リモートでコマンドを実行した場合は [<ホスト名>] run: <コマンド> または [<IP アドレス>] run: <コマンド> のように表示されます。

3.2.4 $RESULT

$RESULT は直前のジョブの実行結果が格納されている特殊な変数（状態変数）です。この場合は whoami コマンドの実行結果、すなわち "kompira" という文字列が格納されることになります。

注釈: $RESULT に格納される値の形式はジョブの種類によって異なります。コマンドジョブの場合は、標準出力が文字列として格納されていますが、ジョブによっては数値や辞書型の場合もあります。

3.2.5 ジョブの連結

ジョブとジョブの間の矢印 -> は、前のジョブが成功したら、後続のジョブを実行する、という意味です。したがって、ジョブを -> でつないでいくことで、順番にジョブを実行していくことができます。

ジョブが失敗した場合（コマンドの実行ステータスが 0 以外を返した場合）でも次の処理を継続したい場合には、二重矢印 => を使います。なお、直前のコマンドの実行ステータスは $STATUS 状態変数で参照することができます。

こうしたジョブを連結する矢印を 結合子 といい、ジョブフローでは 4 種類あります。

3.3 変数を使う

3.3.1 変数の定義

変数は { <変数定義> | <ジョブ> } という構文を用いて定義することができます。<変数定義> の部分は 変数名 = 値（または式）という形式で記述します。コンマで区切って複数の変数定義を記述することもできます。

```
{ x = 'what do you get if you multiply six by nine?', y = 6 * 9 |
  print(x) -> print(y) }
```

この場合、変数 x が 'what do you get if you multiply six by nine?' という文字列で初期化され、変数 y が 6 * 9 と式の計算結果で初期化されます。変数定義の後に縦棒 | で区切って、その変数を参照するジョブを記述することができます。

上のジョブフローを実行すると、コンソールには以下のように表示されます。

```
what do you get if you multiply six by nine?
54
```

3.3.2 識別子

変数名などに用いる識別子には、Unicode で単語文字として分類される文字が使用できます。これには日本語の漢字や平仮名、英数字やアンダースコアが含まれています（アンダースコア以外の記号は含まれません）。ただし、識別子の先頭に数字 [0-9] は使用できません。

したがって、以下のような文字列は識別子として使用可能です。

```
x, foo123, 結果, __reserved_variable__
```

次のような文字列は識別子として使用できません。

```
1st, foo-bar, @id, #hash
```

なお、以下は予約語またはキーワードとして扱われるため、変数名などに用いることはできません。

```
and             break           case            choice
continue        elif            else            false
for             fork            if              in
not             null            or              pfor
then            true            while
```

3.3.3 スコープ

変数の有効範囲（スコープ）は { と } で囲まれた範囲です。スコープ内で定義されていない変数は参照できないため、以下のようなジョブフローは実行時にエラーとなります。

```
{ x = 'hello' |       # 変数 x のスコープは
    print(x) }        # ここまで
-> print(x)           # ここはスコープ外
```

なお、以下のようにスコープを入れ子にすることは可能です。

```
{ x = 'outer', y = 999 |
    print(x) -> print(y)
    -> { x = 'inner' |
        print(x) -> print(y) }
    -> print(x) -> print(y)
}
```

このジョブフローを実行すると以下のようになり、 x = 'inner' のスコープが 3〜4 行目であり、5 行目では外側のスコープを参照していることが分かります。

```
outer
999
inner
999
outer
999
```

すなわち、ジョブフローでの変数のスコープ規則は、C や Java などと同様です。

3.3.4 変数の代入

定義された変数の値を変更するには、［**変数 = 値（または式）**］という形式の **代入ジョブ** を使います。

```
{ x = 'outer', y = 'foo' |
    print(x) -> print(y) ->
    { x = '1st' |
        print(x)
        -> [x = '2nd'] -> print(x)
        -> [x = '3rd'] -> print(x)
        -> [y = 'bar']
        -> [z = 'baz'] }
    -> print(x) -> print(y) }
-> print(z)
```

スコープが入れ子になっている場合、代入はその位置を含む外側のスコープのうち対象となる変数定義を含む **もっとも内側のスコープ** に対して行われます。上の例で言うと、5〜6 行目で値を代入している変数 x は 3 行目で定義したもの、7 行目で代入している変数 y は 1 行目で定義したものです。

未定義の変数に対して値を代入すると、 **最も外側のスコープ（ジョブフロースコープ）**で変数が新たに定義され、その値にセットされます。上の例で言うと、8 行目で値を代入している変数 z はその時点では未定義であるため、最も外側のスコープに新たに定義され、10 行目で表示できることになります。

最も外側のスコープというのは明示的に { } で囲まれてはいませんが、ジョブフロー全体を囲んでいるスコープがあると考えてください。

上記のジョブフローの実行結果は以下のようになります。

```
outer
foo
1st
2nd
3rd
outer
```

(次のページに続く)

(前のページからの続き)

```
bar
baz
```

注釈: $RESULT や $STATUS など状態変数は Kompira が内部的に値を設定するもので、ジョブフローで状態変数への値の代入はできません。

3.3.5 配列と辞書

配列

複数の値を一度に保持したい場合は配列や辞書を使います。配列は [式, . . .] と角括弧の中にコンマで複数の値や式を区切って記述します。配列要素へのアクセスは 0 始まりのインデックスを角括弧で指定することで可能です。また配列要素の書き換えは [値 >> 配列要素] で可能です。

```
[arr = [1, true, 'foo' ,['nested', 'array']]] ->
print(arr[1]) ->                # 配列要素の参照
[false >> arr[1]] ->            # 配列要素の書き換え
[arr = arr + ['added']] ->      # 配列要素の追加
print(arr[3][1]) ->             # 入れ子になった配列要素の参照
print(arr)                      # print() は配列自体の表示も可能
```

このジョブフローを実行すると、以下のようになります。

```
true
array
[1, false, 'foo', ['nested', 'array'], 'added']
```

また負の値をインデックスに指定すると、配列の後ろから要素にアクセスします。

```
[arr = [1, true, 'foo']] -> print(arr[-1])
```

このジョブフローの実行結果は次のようになります。

```
foo
```

辞書

辞書は { 識別子=式, . . . } と波括弧の中にコンマで複数の 識別子=値 を区切って記述します。辞書要素へのアクセスはドット記法または識別子を角括弧で指定することで可能です。また辞書要素の書き換えは [値 >> 辞書要素] で可能です。

```
[dic = {foo=1, bar=true, baz={a=123, b=456}}] ->
print(dic.foo) ->              # 辞書要素の参照（ドット記法）
[false >> dic.bar] ->          # 辞書要素の書き換え
print(dic['bar']) ->           # 辞書要素の参照（角括弧記法）
[[1,2,3] >> dic.arr] ->        # 辞書要素の追加
print(dic.baz.a) ->            # 入れ子になった辞書要素の参照
[777 >> dic.baz.a] ->          # 入れ子になった辞書要素の書き換え
[999 >> dic['baz']['b']] ->    # 入れ子になった辞書要素の書き換え
print(dic)                     # print() は辞書自体の表示も可能
```

このジョブフローを実行すると、以下のようになります。

```
1
false
123
{foo=1, bar=false, baz={a=777, b=999}, arr=[1, 2, 3]}
```

3.3.6 テンプレート文字列

ジョブフローでは文字列中に変数の値を展開することができます。以下のように文字列中に $ と識別子からなるプレースホルダがあると、その部分が識別子の示す変数値で置き換えられます。

```
[service = 'http', port = 80] ->
print('Port $port is used by $service')
```

このジョブフローを実行すると以下のようになります。

```
Port 80 is used by http
```

プレースホルダは $**識別子** のほかに ${**識別子**} という記法も可能ですので、文字列中で識別子の区切りがつかない場合に使用してください。

```
[w=640, h=480] ->
print("width=${w}px, height=${h}px")
```

また、文字列のあとに % を書くと続く辞書に含まれる値を展開することもできます。その場合は文字列中に % と識別子からなるプレースホルダを記述します。

```
print('Port %port is used by %service' % {service = 'http', port = 80})
```

% に続ける辞書はもちろん変数でもよいので以下のようにも書けます。

```
[ctx = {service = 'http', port = 80}] ->
print('Port %port is used by %service' % ctx)
```

いずれの記法でもプレースホルダが指定する変数や辞書要素が未定義である場合は、$ や % も含めてそのまま文字列中に残ります。

3.3.7 パラメータ

ジョブフローはその実行時にパラメータを受け取ることができます。

ジョブフローの先頭で変数名を縦棒で囲む |**変数名**| という記法で、その変数をパラメータとして定義することができます。また、 |**変数名** = **値（または式）**| と記述することで、パラメータのデフォルト値を定義することができます。デフォルト値が定義されていないパラメータは、ジョブフローの実行時に値を指定する必要がある（省略できない）ことに注意してください。

以下のジョブフローでは、 command と wait という 2 つのパラメータを定義していて、 wait にはデフォルト値として 10 が設定されています。

```
| command |
| wait = 10 |

print('Execute the command "$command" after $wait seconds.') ->
["sleep $wait"] ->
[command] ->
print($RESULT)
```

注釈: パラメータはジョブフローの実行開始時に上から順番に評価されます。そのため先に登場したパラメータの値を参照する式を使うこともできます。

3.4 リモートでコマンド実行する

次は、ジョブマネージャが動作しているホストとは別のホスト上でコマンドを実行させてみましょう。

3.4.1 制御変数による指定

まずは制御変数によるコマンドを実行するホストやアカウントの指定方法です。

```
[__host__ = '<ホスト名 もしくは IP アドレス>',
 __user__ = '<ユーザー名>',
 __password__ = '<パスワード>']
-> ['hostname'] -> print($RESULT)
-> ['whoami'] -> print($RESULT)
-> ['echo Hello World'] -> print($RESULT)
```

注釈: <ホスト名> や <ユーザー名> 、 <パスワード> は、自分の環境に合わせて書き変えてください。

__host__ , __user__ , __password__ は、Kompira で予約済みの制御変数 で、これらの変数にそれぞれ、ホスト名 (または IP アドレス)，ユーザー名，パスワードを設定しておくことで、以降のリモートを処理対象とするジョブを設定したホストとユーザー名で実行します。

成功すれば、実行結果は以下のように表示されるはずです。

```
<ホスト名>
<ユーザー名>
Hello World
```

もし、ホスト名が間違っていたり、ユーザー名やパスワードが間違っていると、ジョブフローが失敗し、処理が中止（abort）されます。

3.4.2 ノード情報とアカウント情報の指定

ノード情報オブジェクトとアカウント情報オブジェクトを Kompira ファイルシステム上に作成しておくと、それらをコマンド実行の対象サーバとしてジョブフローから指定することができます。

今、ノード情報オブジェクト test_node とアカウント情報オブジェクト test_account を作成し、ホスト名やユーザ名、パスワード情報が適切に設定されているとします。すると、同じディレクトリにあるジョブフローからのコマンド実行は、ノード情報オブジェクトの指定には制御変数 __node__ を、アカウント情報オブジェクトの指定には __account__ を使用することで、以下のように簡潔に記述することができます。

```
[__node__ = ./test_node, __account__ = ./test_account]
-> ['hostname'] -> print($RESULT)
-> ['whoami'] -> print($RESULT)
-> ['echo Hello World'] -> print($RESULT)
```

注釈: ジョブフローから Kompira オブジェクトの参照は、相対パスまたは絶対パスで記述することで行えます。上の例では ./ で始まる同じディレクトリにあるオブジェクトを指定していますが、 ../ や / で始まるパスで親ディレクトリやルートディレクトリを基準とした指定も可能です。

なおノード情報オブジェクト test_node にデフォルトアカウントを設定している場合は、 __account__ の指定を省略することも可能です。

```
[__node__ = ./test_node]
-> ['hostname'] -> print($RESULT)
-> ['whoami'] -> print($RESULT)
```

また、制御変数をジョブフローのパラメータとして指定することもできるので、実行時に制御対象ノードを指定するジョブフローを作ることもできます。

```
|__node__ = ./test_node|
-> ['hostname'] -> print($RESULT)
```

3.4.3 sudo による実行

コマンドの実行に root 権限が必要な場合、 `__sudo__` 制御変数に `true` をセットして sudo モードに移行します。

```
|__node__ = ./test_node|
-> ['whoami'] -> print($RESULT)
-> [__sudo__ = true]
-> ['whoami'] -> print($RESULT)
```

このジョブフローを実行するとコンソールには以下のように表示されます。

```
<ユーザー名>
root
```

警告: sudo モードでコマンドを正しく実行するためには、そのユーザーが sudoers ファイルに登録されている必要があります。そうでない場合には、sudo モードでのリモートコマンド実行時に処理が失敗 (abort) します。詳しくはマニュアル sudoers(5) を参照してください。

注釈: ホストを指定しないコマンド実行ジョブを sudo モードで実行する場合、ジョブマネージャを実行しているサーバ（通常は Kompira をインストールしたサーバ）の `kompira` ユーザーを sudoers ファイルに登録する必要があります。さらに、以下のように requiretty フラグを無効にする設定を sudoers ファイルに追加しておく必要もあります。

```
Defaults:kompira    !requiretty
```

3.5 制御構造でジョブを操る

3.5.1 条件分岐

直前のジョブの実行結果や変数の内容によって処理を分岐させるには、`if` ブロックもしくは `case` ブロックを利用します。

if ブロック

if ブロックを使うと条件式の結果によって、処理を分岐させることができます。

```
['echo $$RANDOM'] ->
[x = int($RESULT)] ->
{ if x % 2 == 0 |
    then: print('$x は偶数です')
    else: print('$x は奇数です')
}
```

上記では、変数 x の値を 2 で割った余りが 0 に等しければ then 節が実行され、それ以外の場合は else 節が実行されます。なお、 ['echo $$RANDOM'] は乱数を返す環境変数である RANDOM を表示しており、 [x = int($RESULT)] はその結果の文字列を整数化して変数 x に代入しています。

真・偽だけでなく、さらに処理を分岐させたい場合は elif 節を使います。

```
{ if x % 3 == 0 and x % 5 == 0 |
    then: print('FizzBuzz')            # x が 3 でも 5 でも割り切れれば 'FizzBuzz' と表示
    elif x % 3 == 0: print('Fizz')     # x が 3 で割り切れれば 'Fizz' と表示
    elif x % 5 == 0: print('Buzz')     # x が 5 で割り切れれば 'Buzz' と表示
    else: print(x)                     # それ以外の場合は x を表示
}
```

逆に else 節を省略することも可能です。さらにその場合は then キーワードを省略することもできます。

```
[command] =>
{ if $STATUS != 0 | print('エラーが発生しました: ' + $ERROR) }
```

上記の例では変数 command の内容が示すコマンドを実行し、その結果ステータス $STATUS の値が 0 でないとき、 print ジョブで標準エラー出力 ($ERROR) を表示します。

case ブロック

case ブロックによる条件分岐は以下のように書けます。

```
['cat /etc/redhat-release'] ->
{ case $RESULT |
    'CentOS*release 7.*': print("CentOS です")
    'Red Hat*release 7.*': print("Red Hat です")
    else: print("CentOS/Red Hat 7.x が必要です")
}
```

この例では、ファイル /etc/redhat-release の内容で OS の種別を判定するために条件分岐を行っています。パターン文字列には * や ? など Unix のワイルドカードが使用できます。

case ブロックでの文字列のマッチングは先頭のパターンから順次行われ、最初にマッチしたパターンに続くジョブフロー系列のみが実行されます。

どのパターンにもマッチしなかった場合は以下のようになります。

- else 節が含まれる場合はそのジョブフロー系列が実行されます。
- else 節が含まれない場合は case ブロック全体が失敗します（$STATUS に 1 がセットされます）。

注釈: if ブロックとは異なり、 case ブロックでは else 節を省略していてどの条件にもマッチしなかった場合はブロック全体が失敗することに注意が必要です。case ブロックでマッチしなかった場合に何もせずエラーにもしない場合は、else: [] とスキップジョブを else 節に書くようにしてください。

3.5.2 繰り返し

繰り返しは for ブロックや while ブロックを用います。

for ブロック

Kompira が扱えるオブジェクトには、配列や辞書といった複合データまたはディレクトリなど子要素を含むものがあります。あるオブジェクトに含まれる子要素（値やオブジェクト）に対して同じ処理を行ないたい場合 for ブロックを用います。for ブロックの構文は以下のようなものです。

```
{ for <ループ変数> in <子要素を含むオブジェクト> | ジョブ... }
```

たとえば in 節に <ディレクトリパス> を記述することで、そのディレクトリの中にあるオブジェクトのリストを参照することができます。

```
{ for t in /system/types | print(t) }
```

この例では /system/types ディレクトリの中にあるすべてのオブジェクトを、 1 つずつループ変数 t で参照して print() ジョブでコンソールに出力しています。なお Kompira オブジェクトを print() ジョブに渡すと、その絶対パスがコンソールに出力されるため、結果は以下のようになります。

```
/system/types/TypeObject
/system/types/Directory
/system/types/License
/system/types/Virtual
/system/types/Jobflow
/system/types/Channel
    :
```

in 節には以下のように直接配列を記述することも可能です。

```
{ sum = 0 |
    { for i in [1,2,3,4,5,6,7,8,9,10] |
        [sum = sum + i]
    } ->
    print('The total of 1 to 10 is ${sum}.')
}
```

このジョブフローは、1 から 10 までの合計値を計算して出力します。

```
The total of 1 to 10 is 55.
```

また辞書を in 節に続けて記述した場合は、その辞書に含まれる識別子のリストを順次参照することができます。

```
[dic = {a=10, b=20, c=30}] ->
{ for k in dic |
    print("$k = %{$k}" % dic)
}
```

このジョブフローを実行するとコンソールには以下のように表示されます。

```
a = 10
b = 20
c = 30
```

注釈: %{$k} となっている部分は % によるテンプレート展開の前に $k の部分が辞書の識別子によって置き換えられます。そのため、繰り返しのたびにまず %a, %b, %c と展開され、それが辞書 dic の各要素の値でテンプレート展開されて 10, 20, 30 と表示されます。

---

while ブロック

繰り返す対象が決まっているのではなく、ある条件を満たすあいだはジョブを繰り返し処理したいという場合は while ブロックを用います。 while ブロックの構文は以下のようなものです。

```
{ while <式> | ジョブ... }
```

例えば、与えられた 2 つの数の最大公約数を求める「ユークリッドの互除法」は剰余が 0 になるまで繰り返すアルゴリズムですが、これを while ブロックを用いて記述すると以下のようになります。

```
|x = 165|
|y = 105|
[m = x, n = y] ->
{ while n != 0 |
```

(次のページに続く)

(前のページからの続き)

```
    [r = m % n] ->
    print("$m と $n の剰余は $r です") ->
    [m = n, n = r]
} ->
print("$x と $y の最大公約数は $m です")
```

この while ブロックの部分では、 n が 0 ではない間、 m には n を、 n には m と n の剰余を代入する（および表示する）、というジョブを繰り返しています。実行すると以下のように表示されます。

```
165 と 105 の剰余は 60 です
105 と 60 の剰余は 45 です
60 と 45 の剰余は 15 です
45 と 15 の剰余は 0 です
165 と 105 の最大公約数は 15 です
```

3.5.3 ジョブの呼び出し

ジョブフローの呼び出し

あるジョブフローから別のジョブフローを呼び出すには以下の様な構文を用います。

```
[<ジョブフローオブジェクト>]
```

ここでは「サブジョブ」というジョブを作成して、それを呼び出す例を示します。まず、サブジョブを適当なディレクトリの下で、以下のように定義します。

```
print("サブジョブです") ->
return("成功しました")
```

return ジョブは、サブジョブを終了し、結果を呼び出し側のジョブに返します。

次に、このサブジョブを呼び出すメインジョブを同じディレクトリの下に作成します。

```
print("サブジョブを呼び出します")
-> [./サブジョブ]                    # サブジョブを呼び出す
-> print($RESULT)                    # サブジョブの実行結果を出力する
```

「サブジョブ」の呼び出しを指定するところで、文字列の先頭に「./」を追加していることに注意してください。これは、現在のジョブフローが定義されているディレクトリと同じディレクトリ内に「サブジョブ」が定義されていることを示しています。

サブジョブの実行結果は $RESULT で受け取ることができます。上記のメインジョブを実行すると、以下のように表示されます。

```
サブジョブを呼び出します
サブジョブです
成功しました
```

ジョブフローへのパラメータ渡し

ジョブフローを呼び出すときに、次のような構文を用いてパラメータを渡すこともできます。

```
[<ジョブフローオブジェクト> : <パラメータ列> ... ]
```

まず、サブジョブを拡張して、以下のようにパラメータを追加してみましょう。

```
|パラメータ 1 = 'Hello'|
|パラメータ 2 = 'World'|

print("サブジョブです")
-> print(パラメータ 1)
-> print(パラメータ 2)
-> return("成功しました")
```

この状態でさきほどのメインジョブをそのまま実行すると、以下のように表示されます。呼び出し時にパラメータを指定していないため、サブジョブ側で定義したデフォルトパラメータが使われていることがわかります。

```
サブジョブを呼び出します
サブジョブです
Hello
World
成功しました
```

このサブジョブにパラメータを渡して呼び出すには、メインジョブを以下のように拡張します。サブジョブ呼び出し時に、 : に続けて値を記述することで、それらをパラメータ値としてサブジョブに渡すことができます。

```
print("パラメータ付きでサブジョブを呼び出します")
-> [./サブジョブ: 'こんにちは', '世界']
-> print($RESULT)
```

これを実行すると、以下のような結果となります。

```
パラメータ付きでサブジョブを呼び出します
サブジョブです
こんにちは
世界
成功しました
```

呼び出されるジョブフロー側で定義したパラメータ名を指定して、パラメータ値を渡すこともできます。一部のパラメータだけ指定したい場合などに便利です。

```
[./サブジョブ: パラメータ 2='世界']
```

注釈: 呼び出される側で定義されていないパラメータ名を指定したり、定義されたより多くのパラメータ値を渡そうとしたりするとエラーになるので注意してください。

スクリプトジョブの実行

より複雑なジョブを作成したい場合、Kompira ジョブフロー言語よりも、bash, perl, ruby, python などの既存のスクリプト言語を組み合わせて使った方が良いでしょう。Kompira ファイルシステム上で、スクリプトジョブを作成することで、これらのスクリプト言語のプログラムをジョブフローから呼び出すことが可能になります。

ここでは、シェルスクリプトを用いてスクリプトジョブを記述し、それをジョブフローから呼び出す例をみてみます。まず、以下に示すような簡単なシェルスクリプトをスクリプトジョブとして保存します。

```
#! /bin/sh
echo Hello world from shell script
```

Unix 環境で実行させるスクリプトの場合は、1 行目の `#!` で始まる shebang 行を適切に記述してください。Windows 環境で実行するスクリプトの場合は拡張子 (bat/vbs/ps1 など) を適切に指定する必要があります。

このスクリプトジョブを「サンプルスクリプト」として保存したとすると、これを実行するためのジョブフローは、以下のようになります。

```
print('スクリプトジョブを実行します') ->
[./サンプルスクリプト] ->
print($RESULT)
```

__node__ や __host__ を指定しない場合、このスクリプトはジョブマネージャが動作しているマシン上に転送された上で実行されます。実行結果の出力は、リモートコマンドの実行結果と同じように `$RESULT` に格納されます。

注釈: スクリプトは実行時に指定したホストに一時ファイルとして転送され、実行後に削除されます。

スクリプトジョブにもパラメータを渡すことができます。スクリプトジョブ側では、コマンドライン引数としてパラメータを受け取ります。

スクリプトの呼び出し側では、以下のようにキーワード無しの引数としてパラメータを渡します。

```
[./サンプルスクリプト: 'パラメータ 1', 'パラメータ 2']
```

3.6 オブジェクトを操作する

3.6.1 オブジェクトの参照

ジョブフローや環境変数定義など、Kompira で扱う情報は **Kompira オブジェクト**として、Kompira ファイルシステム上で一元的に管理されています。そしてこれらのオブジェクトは Unix のファイルシステムのようなパス指定によってジョブフローからアクセスすることができます。

これまでの例では、Kompira オブジェクトの参照は相対パスによって指定していました。この場合、オブジェクトのパスは、実行中のジョブフローが定義されているディレクトリを基準にして特定されます。

たとえば、実行中のジョブが `/some/path/jobflow` であるときに、`./subdir/object` という相対パスによってオブジェクトを参照すると、`/some/path/subdir/object` がアクセスされることになります。

また、 `../object` という相対パスによって参照すると、 `/some/object` がアクセスされることになります。`../` で始まる相対パスは親ディレクトリにあるオブジェクトを参照することを意味します。

もちろん絶対パスで `/some/path/object` のように直接オブジェクトを参照することもできます。

> **警告:** Kompira オブジェクトの参照では先頭に、 `./` や `../` 、 `/` を付けるのを忘れないでください。Kompira は `./` や `../` 、 `/` から始まる文字列を **パス識別** と認識し、それ以外は変数の識別子と認識します。

パスを連結させてオブジェクトを参照したい場合は、 `path()` 組み込み関数を利用します。たとえばノードの種類毎に「リソース情報取得」を行なうジョブフローを用意しておき、ノードとノード種別を指定してそのジョブフローを実行したい場合、パスを動的に組み立ててジョブフローを参照することができます。

```
|node|
|node_type = 'Linux'|
|job_name = 'リソース情報取得'|
[job = path(./ノード別定義, node_type, job_name)] ->
[job: node]
```

ここでデフォルト引数がそのまま `path()` 関数に渡された場合、 **./ノード別定義**/Linux/**リソース情報取得**というジョブフローを変数 `job` で参照し、 `node` をパラメータとして渡して実行することになります。

3.6.2 プロパティの参照と更新

各 Kompira オブジェクトはシステムで定義された「プロパティ」を持っています。例えばオブジェクトの名称やパス、作成日時などがプロパティです。Kompira オブジェクトが持つプロパティの詳細については プロパティ を参照してください。

Kompira オブジェクトのプロパティを参照するにはドット記法 **オブジェクト . プロパティ名**を用います。以下の

ジョブフローではパラメータ `dir` でしたディレクトリにある Kompira オブジェクトを列挙し、そのプロパティである「所有者 (owner)」,「更新日時 (update)」,「型名 (type_name)」,「表示名 (display_name) 」をドット記法で参照して表示しています。

```
| dir = / |
{ for obj in dir |
    [attr = {
        owner = obj.owner,
        updated = obj.updated,
        type = obj.type_name,
        name = obj.display_name
    }] ->
    print("%owner %updated <%type> %name" % attr)
}
```

Kompira オブジェクトのプロパティ値を更新するには、出力ジョブ［**値 >> オブジェクト . プロパティ**］を用います。

```
["オブジェクトの説明文" >> obj.description]
```

注釈: ただし、プロパティの中にはジョブフローからは更新できない書き込み不可なものもありますので注意してください。詳細は プロパティ を参照してください。

3.6.3 フィールドの参照と更新

各 Kompira オブジェクトは型ごとに定義された「フィールド」を持っています。システムで定義された各型にどのようなフィールドが定義されているかは、`/system/types/` 下にある各型の定義情報を見るとわかります。

Kompira オブジェクトが持つフィールドは **オブジェクト**［**フィールド名**］または **オブジェクト . フィールド名** という記法で参照できます。

注釈: オブジェクトのプロパティにもドット記法でアクセスできることに注意してください。プロパティと同名のフィールドをユーザが定義することもできますが、ドット記法はプロパティ値を優先して参照します。

たとえば、ノード情報オブジェクトには「ホスト名 (hostname)」や「IP アドレス (ipaddr)」といったフィールドが定義されています。ジョブフローでこれらの値を参照するには以下のように記述します。

```
|node = ./node|
print(node['hostname'], node.ipaddr)
```

フィールドの値は辞書のように参照できるため、 `%` によるテンプレート展開もできます。

```
|node = ./node|
print('%hostname: %ipaddr' % node)
```

また、Kompira オブジェクトのフィールド値を更新するには、出力ジョブで［値 >> オブジェクト［フィールド名］］または［値 >> オブジェクト . フィールド名］という記法を用います。例えば Wiki ページ型の「Wiki テキスト」フィールド ('wikitext') を更新するには、以下のように記述します。

```
['= Sample Wiki\n' >> ./wiki['wikitext']]
```

式の結果を出力ジョブで書き込むこともできるので、以下のように参照したフィールド値を加工して再度書き込む、ということもできます。

```
|wiki = ./wiki|
|types = /system/types|
["= Type list\n" >> wiki.wikitext] ->
{ for type in types |
    [wiki.wikitext + "* $type: (" + type.description + ")\n" >> wiki.wikitext]
}
```

上の例では /system/types にあるシステム標準の型オブジェクトについて、そのパスと説明の一覧を記載した Wiki ページを作成しています。

3.6.4 メソッドの呼び出し

Kompira オブジェクトの中にはメソッドを備えているものがあります。オブジェクトのメソッドを呼び出すには、以下のような構文を用います。

```
[ <オブジェクト> . <メソッド名> : <パラメータ列> ... ]
```

例えば、オブジェクトの追加を行なうためにディレクトリ型のオブジェクトには add というメソッドがあります。add メソッドは name, type_obj, data という 3 つのパラメータを指定して呼び出します。以下の例ではジョブフローと同じディレクトリに 'ENV' という名前で環境変数型 (/system/types/Environment) オブジェクトを作成し、'environment' という名前の（辞書型）フィールドに {k1='value1', k2='value2'} という初期データを与えています。

```
[./.add: 'ENV', /system/types/Environment, {
    environment={k1='value1', k2='value2'}
}]
```

ここでは ./ という相対パス識別が、このジョブフローがあるディレクトリを示す Kompira オブジェクトを参照しています。オブジェクトの参照を変数で受け渡すこともできますので、以下のように書くこともできます。

```
[dir = ./, type=/system/types/Environment] ->
[dir.add: 'ENV', type, {environment={k1='value1', k2='value2'}}]
```

またパラメータ列ではパラメータ名を指定して値を渡すこともできます。

```
[dir = ./, type=/system/types/Environment] ->
[dir.add: 'ENV', type_obj=type, data={environment={k1='value1', k2='value2'}}]
```

3.7 イベントを待ち合わせる

チャネルを用いて、ジョブの同期やイベントの待ち合わせ処理をジョブフローで記述することができます。

3.7.1 メッセージの送受信

作成したチャネルに対して、メッセージを送信するには send メソッドを用います。「/home/guest/テストチャネル」に新しいチャネルを作成して試してみましょう。

チャネルに対してメッセージを送信するジョブフローは、以下のようになります。

```
[/home/guest/テストチャネル.send: 'こんにちは']
-> print('メッセージを送信しました')
```

次に、チャネルからメッセージを受信するジョブフローを以下のように定義します。

```
</home/guest/テストチャネル>
-> [mesg = $RESULT]
-> print('メッセージ「$mesg」を受信しました。')
```

上記の各ジョブフローを実行してみてください。受信側のジョブフロー実行のプロセスコンソールに以下のようにメッセージが出力されれば成功です。

```
メッセージ「こんにちは」を受信しました。
```

送信側ジョブフローを複数回実行すると、チャネルにその分だけメッセージが溜まります。受信側のジョブフローを 1 回実行するたびに、そのチャネルから 1 つ分のメッセージを取り出して出力します。もし、チャネルのメッセージが空の場合、受信側のジョブフローは新しいメッセージが到着するまで待ちます。

注釈: kompira_sendevt コマンドを用いることで、外部のシステムから任意の情報をチャネルに送信することができます。たとえば監視システムからアラート情報をチャネルに送信することで、障害発生時の手順をジョブフローで処理させることなども可能でしょう。kompira_sendevt コマンドの利用方法については 他システムと

の連携を参照してください。

3.7.2 イベントジョブについて

<と>で囲まれたジョブをイベントジョブと呼びます。イベントジョブは、他のジョブと同じようにジョブフローの中で組み合わせて使用することができます。

イベントジョブの形式は以下のとおりです。

```
< <オブジェクト名> : <パラメータ列> ... >
```

オブジェクト名にはチャネル型（およびそれに類する型：メールチャネル型など）のオブジェクトを指定します。それ以外のイベント待ち合わせ不可能なオブジェクトを指定すると実行時エラーとなります。

3.7.3 メッセージ受信のタイムアウト指定

チャネルからのメッセージの到着を待ち、一定時間内に来なかったらタイムアウトして処理を先に進めるにはイベントジョブにパラメータ timeout を指定します。

```
print('チャネルからメッセージを待ちます')
-> <./テスト用チャネル: timeout=10>
=> { if $STATUS==0 |
    then: [mesg=$RESULT]
          -> print('メッセージ「$mesg」を受信しました')
    else: print('タイムアウトしました') }
```

timeout で指定した秒数の間にメッセージが到着しなかった場合、イベントジョブは失敗しますので => で次のジョブと結合していることに注意してください。

注釈: メッセージからの到着を待っている時に、チャネルが削除されると、イベントジョブは失敗して $STATUS に -1 をセットします。

3.7.4 複数チャネルからの選択的受信

choice ブロックを使うことで、複数のチャネルからのメッセージの到着を待つことも可能です。この場合、先にメッセージ到着したチャネルについての処理が続行されます。

choice ブロックの使用例

```
print('チャネル 1 とチャネル 2 からメッセージの受信を待ちます')
-> { choice |
    <./チャネル 1> -> [mesg=$RESULT]
        -> print('チャネル 1 からメッセージ「$mesg」を受信しました')
    <./チャネル 2> -> [mesg=$RESULT]
        -> print('チャネル 2 からメッセージ「$mesg」を受信しました')
}
-> print('OK')
```

3.8 外部にアクセスする

3.8.1 メールを送信する

メールの送信には、組み込み mailto ジョブを用います。

```
[subject = 'テストメール',
 body  = 'テストメールを送信します。\n 受け取ったら破棄してください']
-> mailto(to='taro@example.com', from_user='hanako@example.com',
          subject=subject, body=body)
-> print('メールを送信しました')
```

mailto ジョブの引数には、 to (あて先メールアドレス)、 from_user (送信元メールアドレス)、 subject (メール表題)、 body (メール本文) を指定します。

複数のアドレスにメールを送信する場合、以下のように to 引数にメールアドレス文字列のリストを渡して下さい。

```
mailto(to=['taro@example.com', 'jiro@example.com'], from_user='hanako@example.com',
       subject=subject, body=body)
```

警告: メール送信には、sendmail コマンドを使用しています。メールがうまく送信できない場合、sendmail コマンドの設定を確認し、Kompira サーバから sendmail コマンドで正しくメールが送信可能かどうか確認してください。

3.8.2 HTTP アクセスする

Web サーバなどに HTTP アクセスするには、組み込み urlopen ジョブを用います。単純に urlopen() に URL だけを渡した場合は GET アクセスになります。

```
|url = 'http://www.kompira.jp'|
urlopen(url)
```

(次のページに続く)

(前のページからの続き)

```
=> [status = $STATUS, result = $RESULT]
-> { if status != 0 |
then:
    print('HTTP access failed.')
elif result.code != 200:
    print('HTTP status code is %code.' % result)
else:
    print(result.body)
}
```

urlopen() でアクセスに成功した場合の結果は辞書で返されます。code には HTTP のステータスコードが、body にはレスポンスの内容が格納されています。

なお、バージョン 1.5.0 時点での Kompira は HTML を解析する機能は持っていないので、以下のような簡単なスクリプトジョブを html_parse という名前で作成しておきます。このスクリプトは標準入力に渡した HTML から、パラメータで指定した箇所をテキストで抜き出します。

```
#! /usr/bin/python
import sys;
from lxml import html;
if __name__ == '__main__':
    doc = html.fromstring(sys.stdin.read().decode("utf-8"))
    for e in doc.xpath(sys.argv[1]):
        print html.tostring(e, method="text", encoding="utf-8")
```

さて、urlopen() はパラメータ data に辞書データを渡すことで POST アクセスさせることもできます。例として、ネットワークインターフェースに付与された MAC アドレスの前半部分 (OUI) から製品ベンダーを調べる、というジョブフローを考えます。OUI は IEEE という組織が管理しており、http://standards.ieee.org/develop/regauth/oui/public.html から検索することができます。このページにはフォームがあり、x という名前の入力欄に OUI を入力するようになっています。また、検索すると /cgi-bin/ouisearch という CGI が実行されるようになっているので、OUI を x という名前でデータとしてその CGI に渡す POST アクセスをすればよいことになります。

```
|oui = '00-00-00'|
urlopen('http://standards.ieee.org/cgi-bin/ouisearch', data={x=oui})
-> [./html_parse << $RESULT.body: '//pre']
-> print($RESULT)
```

検索結果のページでは <pre> タグに結果がありますので、そこを抜き出すために //pre というパラメータを html_parse スクリプトに渡しています。このパラメータは XPath という XML 文書の部分指定をするための構文で指定します。

このジョブフローを実行すると、以下のようにベンダー情報を外部の Web ページから取得できていることが分かります。

```
[localhost] local: (/tmp/tmpxaL7DG //pre) < /tmp/tmpktOTMU

  OUI/MA-L                      Organization
  company_id                    Organization
                                    Address


  00-00-00    (hex)             XEROX CORPORATION
  000000      (base 16)         XEROX CORPORATION
                                M/S 105-50C
                                800 PHILLIPS ROAD
                                WEBSTER NY 14580
                                UNITED STATES
```

3.9 プロセスを制御する

ジョブフローを実行すると、その終了の時まで Kompira 上ではプロセスという実行単位で管理され、プロセスはジョブフローに記述されたジョブを連続的に順次実行することになります。ここではプロセスを制御する方法について説明します。

3.9.1 プロセスの終了

ジョブフローの終端に到達するなどして継続すべきジョブが無くなった場合や、実行したコマンドが失敗した状態で -> を使ってジョブを結合した場合などは、プロセスは自動的に終了します。

そうした場合以外で、実行中のプロセスを明示的に終了させたいには `exit` ジョブあるいは `abort` ジョブを用います。

exit

実行中のプロセスを終了させるには、組み込み `exit` ジョブを用います。引数を指定せず `exit()` と呼出すと、プロセスをただちに正常終了します。

```
exit()
```

`exit` ジョブの引数で終了ステータスコードを指定することもできます。以下の例では、パラメータ `command` で指定されたコマンドを実行したのち、結果（成功・失敗）にかかわらず標準エラー出力と標準出力を表示してから、コマンド実行結果をステータスコードとしてプロセスを終了します。

```
|command|
[command]
=> [status=$STATUS, stderr=$ERROR, stdout=$RESULT]
```

(次のページに続く)

(前のページからの続き)

```
-> { if stderr | print(stderr) }
-> { if stdout | print(stdout) }
-> exit(status)
```

exit と return の違いに注意してください。例えば、メインジョブから呼び出されたサブジョブで exit ジョブを呼び出すと実行中のプロセスを終了させます（メインジョブに制御が戻らず即座に終了します）。一方、サブジョブで return ジョブを呼び出すと、プロセスを終了させるのではなく、メインジョブに制御が戻り、サブジョブを呼び出した実行ジョブの直後から処理が継続されます。

ただし、呼び出し元が存在しない場合、たとえば直接「実行ボタン」を押して実行したジョブフローから return ジョブを呼び出した場合は、その時点で継続するジョブが無くなりますのでプロセスが終了します。

abort

ジョブを継続できない状態になった場合などに、組み込み abort ジョブを呼出すことで、実行中のプロセスを異常終了させることができます。以下の例ではパラメータで指定した URL に urlopen でアクセスした時に、HTTP アクセスに失敗したか、HTTP ステータスコードが 200 以外の場合にプロセスを異常終了させます。

```
|url|
urlopen(url)
=> [result = $RESULT, status = $STATUS]
-> { if status != 0 | abort('HTTP access failed.') }
-> { if result.code != 200 | abort('HTTP status code is %code.' % result) }
-> return(result.body)
```

abort() ジョブは自動的に終了ステータスコードを 1 にセットしてプロセスを終了させるため、exit(status=1) とほぼ同じです。

3.9.2 子プロセスの起動

Kompira が持つ複数プロセスの並行動作という特徴は、ジョブフローでは「子プロセス」を起動するという方法で利用することができます。

子プロセスは起動した時点では親プロセス（子プロセスを起動したプロセスのこと）のコピーであり、ローカル変数や特殊変数は同じ値を持っていますが、プロセス間での共有や参照はできないので、子プロセスから親プロセスの変数を書き換えたりすることは出来ないことに注意してください（逆方向も同じ）。

fork

fork ブロックを用いて、複数の子プロセスを一度に起動することが可能です。以下では「処理Ａ」というサブジョブの実行結果を、「処理Ｂ」と「処理Ｃ」というサブジョブでそれぞれ並行して処理させるジョブフローの例になります。

```
[./処理A] -> [result = $RESULT] ->
{ fork |
    [./処理B: result] -> print('処理B終了')
    [./処理C: result] -> print('処理C終了')
} -> print('すべての子プロセスが終了しました')
```

fork ブロックの中でジョブ間を結合子で接続していない箇所がありますが、これが「ジョブフロー式」の区切りであり、上の例では fork ブロックに 2 つのジョブフロー式があることになります。この 2 つのジョブフロー式の部分がそれぞれ子プロセスとして並行に動作し、それらの実行が全て完了すると、親プロセスのジョブが継続して「すべての子プロセスが終了しました」とコンソールに出力されます。

なおジョブフローで子プロセスを起動したとき、そのプロセスのプロセス詳細画面の「子プロセス一覧」タブには起動した子プロセスが表示されます。逆に、子プロセスは「プロセス一覧」画面には表示されないことに注意してください。

pfor

for ブロックの代わりに pfor ブロックを使うことで、繰り返し処理が並列プロセスとして一度に実行することができます。

例えば管理対象のノードを「ノード一覧」で管理しており、管理対象すべてのノードに対して同じジョブ「構成情報収集」を実行したい場合、for ブロックを用いて以下のように書けます（構成情報収集はパラメータで処理対象のノードを指定するものとします）。

```
|job = ./構成情報収集|
{ for node in ./ノード一覧 |
    [job: node]
} -> print("すべてのノードに対する処理が終了しました")
```

もしこの「構成情報収集」ジョブがリモートノードに対して処理時間のかかるコマンドを投入しているような場合、このプロセスは「待ち状態」になっている割合が多くなります。結果的に、負荷は低いがすべてのノードに対する処理が終了するまでに長い時間がかかる、ということになります。

こうしたときに for の代わりに pfor を使うと、各ノードごとに子プロセスを起動してその子プロセスで「構成情報収集」ジョブを実行することになります。そうすると、あるノードに対する処理で「待ち状態」になっていても、別のノードの処理を並列に実行できるため、全体でのジョブの実行効率を上げて処理時間を短縮できるようになります。

```
|job = ./構成情報収集|
{ pfor node in ./ノード一覧 |
    [job: node]
} -> print("すべてのノードに対する処理が終了しました")
```

3.9.3 親プロセスからの切り離し

子プロセスを fork や pfor を使って起動した親プロセスはすべての子プロセスの終了を待つため、その間親プロセスは新たなジョブを動作させることはできません。しかし、子プロセスの終了を待たずに親プロセス側の処理を継続させたい、という場合もあります。そういう場合には detach() を用いて親プロセスから切り離すことで対応できます。

detach

例えばチャネルからメッセージを受信するたびに同じジョブフローを実行したい、ということがしばしばあります。以下ではメッセージを受信するたびに「メッセージ処理」というジョブフローに、メッセージをパラメータとして渡して呼び出しています。

```
|chan = /system/channels/Alert|
|proc = ./メッセージ処理|
{ while true |
    <chan>
    -> [msg = $RESULT]
    -> [proc: msg]
}
```

チャネルから受信する複数のメッセージ間に関連性が無いときは、メッセージ処理を同時に実行させるようにすることで、メッセージが連続して到着したときに全体での処理効率の向上につながることがあります。そのためには、メッセージを受信するジョブフローと「メッセージ処理」を別のプロセスとして動作させる必要があります。そこで fork を用いて「メッセージ処理」を子プロセスで動作させるようにしてみます。

```
|chan = /system/channels/Alert|
|proc = ./メッセージ処理|
{ while true |
    <chan>
    -> [msg = $RESULT]
    -> { fork | [proc: msg] }
}
```

しかし、これでは「メッセージ処理」のジョブが完了するまで親プロセスは待ってしまうため、メッセージ処理の最中に新しいメッセージが到着しても、同時に処理することはできません。そこで子プロセス側で detach() 組み込みジョブを用いて、子プロセスを親プロセスから切り離すようにします。

```
|chan = /system/channels/Alert|
|proc = ./メッセージ処理|
{ while true |
    <chan>
    -> [msg = $RESULT]
    -> { fork | detach() -> [proc: msg] }
}
```

親プロセス側は子プロセスの `detach()` により、処理完了を待つべき子プロセスがなくなるため、その時点で次のジョブを継続できることになります。すなわちチャネルから次のメッセージを受信を行ない、先に起動した「メッセージ処理」がまだ完了していなくても新しい「メッセージ処理」を起動できることになります。

子プロセス側は `detach()` により子プロセスではなく通常のプロセスとなり、親プロセスの「子プロセス一覧」ではなく「プロセス一覧」画面に表示されるようになります。

このように `fork` や `pfor` と `detach()` を組合せることで、多少複雑な並行処理でも簡単に記述することができます。

第 4 章

Kompira ジョブフロー言語リファレンス

著者　Kompira 開発チーム

4.1 イントロダクション

本ドキュメントは、ジョブフロー言語の字句と構文、および、意味について説明します。組み込み関数、組み込みジョブについての記述は *Kompira* 標準ライブラリにあります。

4.1.1 構文の表記法

本ドキュメントでは、拡張 BNF を用いて構文を示します。拡張 BNF は通常の BNF に加えて、0 回以上の繰り返しを表す「*」、1 回以上の繰り返しを表す「+」、省略可能な要素を表す「?」といった記号を用います。また、複数の要素をまとめるために丸カッコ「(」と「)」も使用します。

4.2 字句構造

この章では、ジョブフロー言語の字句構造について規定します。ジョブフロー言語のプログラムテキストは Unicode で記述します。テキストは、Kompira の字句解析器によって、トークンと呼ばれる語彙の単位に区切られます。

4.2.1 コメント

コメントは文字列リテラル内に含まれないハッシュ文字 (#) から始まり、行末までになります。コメントは字句解析器によって読み飛ばされます。

4.2.2 空白

改行文字、スペース、タブ、フォームフィードは空白として扱われます。空白は字句解析器によって読み飛ばされます。

4.2.3 識別子

識別子 (IDENTIFIER) は、以下の正規表現によって定義されます。:

```
IDENTIFIER = [^\W0-9]\w*
```

\w は任意の Unicode 単語文字にマッチします。これにはあらゆる言語で単語の一部になりうる文字、数字、およびアンダースコアが含まれます。\W は [^w] を意味します。識別子の長さに制限はありません。大小文字は区別されます。

キーワード

以下の文字の並びは、キーワードとして予約されているため識別子として用いることはできません。:

```
and             break           case            choice
continue        elif            else            false
for             fork            if              in
not             null            or              pfor
session         then            true            try
while
```

予約済みの識別子クラス

__*__ の形式の識別子は制御変数用にシステムで予約されていて特別な意味があります。ジョブフローの予期せぬ動作を引き起こす可能性があるため、ユーザーが識別子としてこれらの名前を用いることは避けた方が良いでしょう。

特殊識別子

以下で定義されるように $ で始まる識別子は特殊変数に用いられる特殊識別子です。:

```
SPECIAL_IDENTIFIER = "$" IDENTIFIER
```

4.2.4 オブジェクトパス

オブジェクトパスは、Kompira ファイルシステム上のオブジェクトの位置を指し示します。以下のように定義されています。:

```
      OBJECT_PATH = RELATIVE_PATH
                  | ("/" | RELATIVE_PATH) PATH_ELEMENT* LAST_PATH_ELEMENT
    RELATIVE_PATH = "./" | "../"
     PATH_ELEMENT = RELATIVE_PATH | IDENTIFIER "/"
LAST_PATH_ELEMENT = RELATIVE_PATH | IDENTIFIER
```

注釈: 単独の "/" もオブジェクトパスとして扱われますが、除算演算子"/" と字句解析上は区別がつかないため、字句としての OBJECT_PATH には含まれません。

4.2.5 リテラル

リテラルは、文字列 (String) 型、整数 (Integer) 型、ブール (Boolean) 型、ヌル (Null) 型、パターン (Pattern) 型の値のソースコード上の表記です。

文字列リテラル (STRING)

文字列リテラルは、一重引用符 (')、もしくは、二重引用符 (") で囲まれた 0 個以上の文字から構成されます。

```
""                      # 空文字列
'"'                     # "を保持した文字列
'\''                    # ' を保持した文字列
"This is a string"    # 16 文字を保有した文字列
' これは文字列です'     # 8 文字を保有した文字列
```

また、対応する 3 連の一重引用符や二重引用符で囲むこともできます。この場合には、エスケープされていない改行や引用符を書くことができます。

```
'''''''                 # ' を保持した文字列
"""改行を含む
文字列"""               # 改行コードを含む文字列
```

文字列リテラル中では、エスケープシーケンスを使用することによって、改行文字やタブ文字など、ある種の表現できない文字を表現することができるようになります。

エスケープシーケンスの一覧を以下に示します。

エスケープシーケンス	意味
\	バックスラッシュ ()
’	一重引用符 (’)
"	二重引用符 (")
a	ASCII 端末ベル (BEL)
b	ASCII バックスペース (BS)
f	ASCII フォームフィード (FF)
n	ASCII 行送り (LF)
r	ASCII 復帰 (CR)
t	ASCII 水平タブ (TAB)
v	ASCII 垂直タブ (VT)
ooo	8 進数値 *ooo* を持つ文字
xhh	16 進数値 *hh* を持つ文字

整数リテラル (INTEGER)

整数リテラルは 10 進数で表現することができます。以下の字句定義で記述されます。:

```
INTEGER       = NONZERO_DIGIT DIGIT* | "0"
DIGIT         = [0-9]
NONZERO_DIGIT = [1-9]
```

真偽リテラル (BOOLEAN)

真偽リテラルはブール型の真 (true) と偽 (false) の 2 つの値の表記があります。:

```
BOOLEAN = "true" | "false"
```

ヌルリテラル (NULL)

ヌルリテラルは値が無いことを示す値で、null と表記されます。:

```
NULL = "null"
```

パターンリテラル (PATTERN)

パターンリテラルは、パターンの種別を表す ’e’, ’g’, ’r’ のいずれかの文字に引き続き、一重引用符 (’)、もしくは、二重引用符 (") で囲まれた 0 個以上の文字（パターン文字列）からなります。最後にモードを示す ’i’ がオプションとして付加される場合もあります。

```
r"(From|Subject): "       # 正規表現パターン
g'*.txt'                  # glob パターン
e'kompira'i               # 大文字小文字を区別しない完全一致パターン
r"windows(95|nt|2000)"i   # 大文字小文字を区別しない正規表現パターン
```

パターン文字列内ではエスケープシーケンスは無効となり、そのままの文字として扱われます。${identifier} による文字列置換は有効です。

4.2.6 記号

記号は、演算子記号 (OPERATOR)、結合子記号 (COMBINATOR)、デリミタに分類されます。

演算子

以下のトークンは演算子です。:

```
+       -       *       /       %
<       >       <=      >=      ==      !=      =~      !~
```

結合子

以下のトークンは結合子です。:

```
->      =>      ->>     =>>
```

デリミタ

以下のトークンはデリミタです。:

```
(   )   {   }   [   ]   |   ,   .   =   >>   <<   ?   ??
```

4.3 値と型

Kompira のジョブフロー言語では、整数や文字列、日付など様々な値（データ）を扱うことができます。値はその種類毎にいくつかの型に分類されます。ある型の値は暗黙的に別の型に変換される場合があります。たとえば、整数型の値を文字列型のフィールドに書き込む場合、対応する文字列型の値に暗黙的に変換されてから書き込まれます。

4.3.1 プリミティブ型

プリミティブ型は、Kompira のジョブフロー言語が提供する基本的なデータの型の総称で、整数型、文字列型、ブール型、ヌル型の 4 つがあります。プリミティブ型の値は、他のプリミティブ型の値と状態が共有されることはありません。

整数型 (Integer)

整数型は 0 や 1, 1000, -9999 など、整数を表す値の型を取り扱います。Kompira の整数型は、（メモリの許す限り）範囲に限りはありません。

注釈: Kompira オブジェクトの Integer 型フィールドは範囲が制限されているため、ジョブフローから範囲外のデータを Integer 型フィールドに書き込むと実行時エラーとなります。

文字列型 (String)

文字列型は、"kompira" や "本日は晴天なり" のような文字列の値のための型です。文字列の各要素は文字です。Kompira のジョブフロー言語では文字型は存在しません。単一の文字は、要素が 1 つだけの文字列として表現されます。各文字は内部的には Unicode で表現されています。

"123" や "-999" など、整数を表す文字列の値を、整数型のフィールドに書き込む場合、対応する整数型の値に暗黙的に変換されます。

文字列型の値が、ブール型に変換される場合、空文字列 ("") が false、それ以外の文字列が true と対応します。したがって、文字列 "false" は ブール型の true に対応するので注意が必要です。

文字列型のデータは、以下のメソッドを備えています。

format (*args, **kwargs)　文字列の書式化操作を行います。このメソッドを呼び出す文字列は通常の文字、または、{} で区切られた置換フィールドを含みます。それぞれの置換フィールドは位置引数のインデックスナンバー、または、キーワード引数の名前を含みます。返り値は、それぞれの置換フィールドが対応する引数の文字列値で置換された文字列のコピーです。

join (list)　リスト中の文字列を結合した文字列オブジェクトを返します。要素間の区切り文字列は、このメソッドを提供する文字列です。

find (sub[, start[, end]])　文字列のスライス s[start:end] に部分文字列 sub が含まれる場合、その最小のインデックスを返します。オプション引数 start および end はスライス表記と同様に解釈されます。sub が見つからなかった場合 -1 を返します。

rfind (sub[, start[, end]])　文字列中の領域 s[start:end] に sub が含まれる場合、その最大のインデクスを返します。オプション引数 start および end はスライス表記と同様に解釈されます。sub が見つからなかった場合 -1 を返します。

startswith (prefix[, start[, end]])　文字列が指定された prefix で始まるなら true を、そうでなければ false を返します。prefix は見つけたい複数の接頭語のリストでも構いません。オプションの start があれば、その位置から判定を始めます。オプションの end があれば、その位置で比較を止めます。

endswith (prefix[, start[, end]])　文字列が指定された suffix で終わるなら true を、そうでなければ false を返します。suffix は見つけたい複数の接尾語のタプルでも構いません。オプションの start があれば、その位置から判定を始めます。オプションの end があれば、その位置で比較を止めます。

lower ()　全ての大小文字の区別のある文字が小文字に変換された、文字列のコピーを返します。

upper ()　全ての大小文字の区別のある文字が大文字に変換された、文字列のコピーを返します。

replace (old, new[, count])　文字列をコピーし、現れる部分文字列 old 全てを new に置換して返します。オプション引数 count が与えられている場合、先頭から count 個の old だけを置換します。

split ([sep[, maxsplit]])　文字列を sep をデリミタ文字列として区切った単語のリストを返します。maxsplit が与えられていれば、最大で maxsplit 回分割されます。

rsplit ([sep[, maxsplit]])　sep を区切り文字とした、文字列中の単語のリストを返します。maxsplit が与えられた場合、文字列の 右端 から最大 maxsplit 回分割を行います。

splitlines ([keepends])　文字列を改行部分で分解し、各行からなるリストを返します。

strip ([chars])　文字列の先頭および末尾部分を除去したコピーを返します。引数 chars は除去される文字集合を指定する文字列です。

ブール型 (Boolean)

ブール型は、真理値の真 (true) と偽 (false) という 2 つの値をとる型です。

ヌル型 (Null)

ヌル型は、null 値のみを持つ型です。

パターン型 (Pattern)

パターン型は、文字列とマッチングを行うためのパターンを表す値の型です。パターンの種別は、'r' (正規表現パターン)、'g' (glob パターン)、'e' (完全一致パターン) の 3 種類あります。また、パターンマッチングのモードとして、大文字、小文字を区別しないモード ('i') を組み合わせることもできます。

正規表現パターンは、プログラミング言語 Python の re モジュールの正規表現に準じています。

glob パターン では、Unix のシェル形式のワイルドカードを用いることができ、以下の特別な文字に対応しています。

パターン	意味
*	すべてにマッチします
?	任意の一文字にマッチします
[seq]	seq にある任意の文字にマッチします
[!seq]	seq にない任意の文字にマッチします

完全一致パターンは単に文字列の比較となります。

パターン型のデータは、以下のメソッドを備えています。

match (s)　文字列 s とパターンとのマッチングを試みます。マッチした場合は true、もしくはパターンが正規表現パターンの場合はマッチした情報を格納した辞書を返します。マッチしなかった場合は、false を返します。

正規表現パターンがマッチした際に返す辞書データには、以下のエントリが含まれます。

キー	意味
group	正規表現にマッチした文字列
groups	すべてのサブグループの文字列を含むリスト
groupdict	名前付きグループの辞書
start	マッチの開始位置
end	マッチの終了位置

4.3.2 複合データ型

複合データ型は、他の型の複数の要素を保持することができるデータの型の総称で、配列型と辞書型の 2 種類があります。

配列型 (Array)

配列型のデータは、要素を 1 次元に並べたデータ構造で、整数のインデックスで要素にアクセスすることができます。配列の長さが n の場合、インデックスは 0, 1, ..., n - 1 となります。配列 a の要素 i は a[i] で参照することができます。インデックス i が負の場合、 a[i] は 要素 n+i を参照します。

配列の範囲外の要素にアクセスした場合、実行時エラーとなります。また、配列は拡張することはできません。

配列型のデータは、以下のメソッドを備えています。

add_item (value)　配列 a の最後にデータ value を追加します。

del_item (index)　配列 a の要素 a[index] を削除します。

pop_item ([index])　配列 a の要素 a[index] を削除します。index が指定されなければ最後の要素を削除します。

辞書型 (Dictionary)

辞書型のデータは、要素を複合データ型を除く任意の型のキーで関連づけられた要素にアクセスすることができるデータ構造です。辞書 d のキー k で関連づけられた要素は、d[k] で参照することができます。キー k が文字列型の値で、かつ、識別子 (IDENTIFIER) の字句要件を満たす文字列の場合、d.k で参照することもできます。

辞書に含まれていないキーで要素を参照しようとした場合、実行時のエラーになります。書き込みジョブ（後述）によって新しいキーと要素を追加することができます。

辞書型のデータは、以下のメソッドを備えています。

del_item (key)　辞書 d の要素 d[key] を削除します。

get_item (key[, default])　辞書 d の要素 d[key] を取得します。要素 d[key] が存在しない場合、default を返します。default のデフォルト値は null です。

pop_item (key[, default])　辞書 d の要素 d[key] を削除します。要素 d[key] が存在しない場合、default を返します。default が与えられず、かつ、要素 d[key] が存在しない場合は、エラーとなります。

get_keys ()　辞書 d のキーのリストを返します。

4.3.3 不透明データ型

不透明データ型は、データの内部構造が隠ぺいされたデータ型の総称です。また、対応するデータコンストラクタを持たないため、複合データ型のようにジョブフロープログラムのソースコード中の表記によって直接データを生成することはできません。

オブジェクト型 (Object)

オブジェクト型の値は、Kompira ファイルシステム上のオブジェクトの参照を表現します。オブジェクト型の値の文字列表現はそのオブジェクトの絶対パスとなります。オブジェクト o のプロパティ p は、o.p で、フィールド f は o[f] という表記でアクセスすることができます。フィールド f と同名のプロパティ名やメソッド名が無ければ、o.f という表記でフィールドを参照できます。

Kompira のオブジェクトは、型オブジェクト (TypeObject) によって定義されるフィールドとメソッドを備えています。詳細は Kompira オブジェクトリファレンス（*Kompira 標準ライブラリ*）を参照してください。

ファイル型 (File)

ファイル型の値は、ファイル型フィールドを持つオブジェクトに添付されるファイルデータを表します。

ファイル型の値には以下のフィールドが定義されています。

フィールド名	
name	添付ファイル名
data	添付ファイルデータ
path	Kompira サーバ上のローカルなファイルパス (読み取り専用)
size	データサイズ (読み取り専用)
url	ダウンロード URL (読み取り専用)

日時型 (Datetime)

日時型の値は、日付と時刻の両方を含むデータを表します。

日時型の値は以下の読み取り専用のプロパティを備えています。

キー	意味
year	年
month	月 (1 から 12 までの値)
day	日 (1 から与えられた月と年における日数までの値)
hour	時 (0 から 23 までの値)
minute	分 (0 から 59 までの値)
second	秒 (0 から 59 までの値)
weekday	月曜日を 0、日曜日を 6 として、曜日を整数で表した値
date	日付型データ
time	時刻型データ

日時型のデータは、以下のメソッドを備えています。

format (dt_fmt)　dt_fmt で指定されたフォーマットで日時データを文字列に変換します。フォーマットの書式指定は、C 言語の strftime() 関数に準じています。

以下に例を示します。

```
[dt = now()] -> print(dt.format('%Y 年%m 月%d 日 %H 時%M 分%S 秒'))
```

isoformat ()　*YYYY-MM-DDTHH:mm:ssZ* の形式（ISO 8601 フォーマット）の文字列を返します。タイムゾーンは常に UTC となり、*Z* の接尾辞がつきます。

日付型 (Date)

日付型の値は、日付のデータを表します。

日付型の値は以下の読み取り専用のプロパティを備えています。

キー	意味
year	年
month	月 (1 から 12 までの値)
day	日 (1 から与えられた月と年における日数までの値)
weekday	月曜日を 0、日曜日を 6 として、曜日を整数で表した値

日付型のデータは、以下のメソッドを備えています。

format (dt_fmt)　dt_fmt で指定されたフォーマットで日付データを文字列に変換します。フォーマットの書式指定は、C 言語の strftime() 関数に準じています。

時刻型 (Time)

時刻型の値は、時刻のデータを表します。

時刻型の値は以下の読み取り専用のプロパティを備えています。

キー	意味
hour	時 (0 から 23 までの値)
minute	分 (0 から 59 までの値)
second	秒 (0 から 59 までの値)

時刻型のデータは、以下のメソッドを備えています。

format (dt_fmt)　dt_fmt で指定されたフォーマットで時刻データを文字列に変換します。フォーマットの書式指定は、C 言語の strftime() 関数に準じています。

経過時間型 (Timedelta)

経過時間型の値は、日時型の値の差分を表すデータです。日時型の値と経過時間型の値の間で、加算と減算が可能です。また、日時型の値同士の差分は経過時間型となります。

経過時間型の値は以下の読み取り専用のプロパティを備えています。

キー	意味
days	日数
seconds	秒数
microseconds	マイクロ秒数
total_seconds	経過時間のトータル秒数

4.4 変数

変数は値を保持するための記憶領域に付けた名前のことです。Kompira ジョブフロー言語における変数は、任意の型の値を保持することができます。

注釈: fork や pfor ブロックによって生成された子プロセス同士、あるいは親プロセスと子プロセスの間では、同じスコープであっても変数は共有されません。ただし、子プロセスから、子プロセスが生成された時点の親プロセスのスコープの変数を参照する（読み取る）ことは可能です。

4.4.1 ローカル変数

ローカル変数は、ジョブフローのパラメータ、代入ジョブによって導入されます。ローカル変数はその変数が導入されるソースコード上の位置によって異なるスコープを持ちます。

ジョブフロースコープ

ジョブフロースコープは、その変数が導入されたジョブの後続の任意のジョブから参照できるスコープです。ジョブフローパラメータはジョブフロースコープを持ちます。また代入ジョブによって未定義変数が新たに導入された場合には、その変数はジョブフロースコープを持ちます。

ジョブフロースコープの変数は、同名の変数が内側のブロックスコープで再定義された場合、隠されます。

ブロックスコープ

ブロックスコープは、そのブロックの内側からのみ参照できるスコープです。単純ブロックで定義される変数や、for や pfor ブロックによって導入されるループ変数は、ブロックスコープを持ちます。

4.4.2 環境変数

環境変数は、ユーザー毎に設定する環境変数型 (Environment) オブジェクトの環境変数フィールドの辞書にもとづいて、ジョブフロー実行時に読み込まれる読み取り専用の変数です。

環境変数は、同名の変数がジョブフロースコープやブロックスコープとして導入された場合、隠されます。

警告: 環境変数フィールドのキー名が識別子の形式に合致していない場合、その環境変数を参照することはできません。

4.4.3 特殊変数

特殊変数は、システムによってあからじめ定義された特別な意味を持つ変数です。

状態変数

状態変数はリモートジョブなどの実行結果やステータスコードなどを一時的に格納するための予約された変数であり、$で始まる変数です。これらの変数は、Kompira エンジンによって自動的に値がセットされるものであり、ジョブフローの中で明示的に代入を行うことはできません。

状態変数には、以下の種類があります。

変数名	意味
$RESULT	ジョブの実行結果 (標準出力)
$STATUS	ジョブの実行ステータス
$ERROR	ジョブ実行のエラーメッセージ (標準エラー出力)
$DEBUG	デバッグ情報

注釈: ジョブの実行結果の文字コードは自動判別され、適切な文字列に変換されます。文字列への変換に失敗した場合、ジョブの実行は失敗とみなされ、エラーを返します。

制御変数

制御変数は、リモートジョブを実行する際のホスト名やログイン名などを指定するための変数であり、__*__のように 2 つ続きのアンダースコア (_) が前後に付加された形式の変数です。

制御変数はローカル変数として定義することもできますし、環境変数として設定しておくことも可能です。

制御変数には、以下の種類があります。

変数名	意味
__realm__	リモートコマンドを実行する管理領域を指定する
__host__	リモートコマンドの実行ホスト名を指定する
__conntype__	リモートコマンドの実行ホストの接続タイプを指定する
__user__	リモートコマンドの実行ユーザー名を指定する
__password__	パスワードを指定する
__node__	リモートコマンドを実行するノード情報オブジェクトを指定する
__account__	リモートコマンドを実行するために必要なアカウント情報オブジェクトを指定する
__sudo__	sudo モードで実行する場合は、true にセットする (この時、PTY モードは true となる)
__dir__	リモートコマンドの実行ディレクトリを指定する
__port__	ssh ポート番号を指定する
__keyfile__	ssh 鍵ファイルパスを指定する
__passphrase__	ssh 鍵ファイルのパスフレーズを指定する (パスフレーズ無しの場合や__password__と同じなら省略可能)
__timeout__	リモートコマンドがタイムアウトするまでの秒数を指定する
__proxy__	実行ホストに接続する際の経由ホストを指定する
__shell__	リモードコマンド実行時に利用する shell を指定する (デフォルト: "/bin/bash -l -c")
__use_shell__	リモートコマンド実行時に shell を利用しない場合は false をセットする
__use_pty__	リモートコマンド実行時に PTY を利用する場合は true をセットする
__winrs_auth_type__	WinRS 接続の認証方式を"ntlm"(デフォルト), "credssp"から指定する。
__winrs_scheme__	WinRS 接続のスキームを"http"(デフォルト), "https"から指定する。
__winrs_use_tlsv10__	WinRS 接続にて CredSSP 認証を行う際に TLS1.0 を使用する場合は true をセットする。(WindowsServer2008 など、TLS1.2 を使用出来ない環境向け)

バージョン 1.4 で非推奨: 制御変数 __via__ はバージョン 1.4.0 で廃止されました。かわりに__proxy__を使用してください。

注釈: __dir__ で指定するディレクトリ文字列に (や)、" や ' など、シェルのメタ文字が含まれる場合は、以下のように適切にエスケープする必要があります。:

```
[__dir__ = 'somedir\\(foo\\)']
```

注釈: __sudo__ = true かつ __use_shell__ = false のとき、__dir__ は指定できません。

注釈: __timeout__ を指定しない場合、もしくは、0 の値を設定した場合、リモートコマンド実行時にタイムアウトはしません。__timeout__ に負の値を設定した場合の動作は未定義です。

バージョン 1.4.9 で変更.

winrs モードでは、実行しているリモートコマンドが出力を続けている限りはタイムアウトしません。すなわち、__timeout__ で指定した秒数の時間、出力が無い場合にタイムアウトします。

バージョン 1.5.4.post5 で変更.

winrs モードでコマンドの出力があっても __timeout__ で指定した秒数でタイムアウトするようになりました。

注釈: バージョン 1.4.8.post6 で変更.

winrs モードによるリモートコマンド実行では、 __timeout__ で指定した値と WinRM 側の MaxTimeoutms で設定した値のうち、より小さい値が適用されます。

バージョン 1.5.4.post5 で変更.

winrs モードでのコマンド実行において、__timeout__ によるタイムアウト指定が MaxTimeoutms より優先されるようになりました。

4.5 式

ジョブフロープログラム中の式は、ジョブフローの実行過程の中で評価され、結果として何らかの値を持ちます。

4.5.1 アトミック式

アトミックな式は、式を構成する基本単位となります。識別子やオブジェクトパス、リテラルはアトミックな式に含まれます。また、カッコで囲まれた形式も文法的にはアトミックな式に分類されます。

```
atomic_expression  ::=  IDENTIFIER | OBJECT_PATH | SPECIAL_IDENTIFIER
                        | literal
                        | parenth_form
                        | array_expression
                        | dict_expression
```

識別子 (IDENTIFIER)

アトミックな式としての識別子は、変数名を表します。変数名を評価すると、評価時の実行環境の下でその変数名に束縛された値を返します。

オブジェクトパス

オブジェクトパスは、そのパスが指し示す Kompira のオブジェクトを値として返します。オブジェクトパスが相対パス形式の場合は、相対パスの起点はそのジョブフローオブジェクトが所属するディレクトリとなります。たとえば、ジョブフロー /root/test_jobflow の中で、./test_object が指し示すオブジェクトは、 /root/test_object となります。

オブジェクトが存在しない場合、実行時エラーとなります。

特殊識別子

特殊識別子はジョブ実行後の状態変数を表します。ジョブフロー開始時点で、$STATUS は 0 に、$RESULT と $ERROR は空文字列 ("") にそれぞれ初期化されます。

リテラル

リテラルは、文字列、数値、真偽値、ヌルがあります。

```
literal  ::=   STRING | INTEGER | BOOLEAN | NULL
```

リテラルを評価すると、そのリテラルが示す値になります。

文字列リテラルの場合、その文字列に含まれる $で前置された変数が展開されます。以下の規則に従います。

- $identifier は置換プレースホルダの指定で、"identifier"というキーへの対応付けに相当します。デフォルトは、"identifier"の部分には Kompira の識別子が書かれていなければなりません。$の後に識別子に使えない文字が出現すると、そこでプレースホルダ名の指定が終わります。
- ${identifier} は$identifier と同じです。プレースホルダ名の後ろに識別子として使える文字列が続いていて、それをプレースホルダ名の一部として扱いたくない場合に必要な書き方です。

例えば、以下に示すジョブフローを実行すると、コンソールには "Hello Kompira" と出力されます。:

```
[name = 'Kompira']
-> print('Hello $name')
```

丸カッコ形式

丸カッコ形式は、囲まれている式を評価し、その値を返します。

```
parenth_form  ::=   "(" expression ")"
```

配列式

配列式は、角カッコで囲われた式のカンマ区切りの並びです。式の並びは省略することも可能です。

```
array_expression  ::=   "[" expression_list? "]"
expression_list   ::=   expression ("," expression_list)*
```

配列式を評価すると、新たに作成された配列型のデータを値として返します。配列の各要素は左から右へと順に評価されます。

辞書式

辞書式は、波カッコで囲われたキーと値のペアのカンマ区切りの並びです。ペアの並びは省略することも可能です。ペアが等号で結ばれる場合、キーは識別子でなくてはなりません。キーが重複している場合はコンパイル時にエラーを報告します。コロンで結ばれる場合は、キーは任意の式を記述することができます。この場合はキーの重複はチェックされません。

```
dict_expression  ::=   "{" ( binding_list | key_val_list )? "}"
binding_list     ::=   binding ("," binding_list)*
binding          ::=   IDENTIFIER "=" expression
key_val_list     ::=   key_val ("," key_val_list)*
key_val          ::=   expression ":" expression
```

辞書式を評価すると、新たに作成された辞書型のデータを値として返します。カンマ区切りの一連のキーと値のペアが与えられると、その要素は左から右へ評価され、辞書の項目を定義します。重複したキーを与えると文法エラーとなります。

4.5.2 後置式

後置式は、式の中で最も結合性が高くなります。

```
postfix_expression  ::=   attribute_reference
                        | subscript_reference
                        | function_call
                        | atomic_expression
```

属性参照

属性参照は、後置式の後にドットがあり、さらに識別子が続く形式です。

```
attribute_reference  ::=  postfix_expression "." IDENTIFIER
```

後置式の評価結果は、オブジェクト型でなければいけません。後置式の評価結果のオブジェクトの識別子で指定された属性が評価結果の値となります。指定された属性が存在しない場合は、識別子の文字列をキーとしてオブジェクトのフィールド値となります。そのようなフィールドも存在しない場合は、実行時エラーとなります。

添字参照

角カッコで囲まれた式が後に付いている後置式は、オブジェクトのフィールドや配列、辞書データから要素を取り出す式を表します。

```
subscript_reference  ::=  postfix_expression "[" expression "]"
```

後置式の評価結果は、オブジェクト型、辞書型、配列型のいずれかでなければいけません。後置式の評価結果がオブジェクト型か辞書型の場合、角カッコの中の式の評価結果は文字列型である必要があります。この場合、文字列をキーとしてオブジェクトのフィールドや辞書データの要素が選択されます。後置式が配列型の場合、角カッコの中の式は整数型である必要があります。この場合、整数値をインデックスとして該当する配列の要素が選択されます。

キーやインデックスに対応する要素が存在しない場合、実行時エラーとなります。

関数呼び出し

関数呼び出しは、組み込み関数やライブラリ型オブジェクトで定義された関数、オブジェクトのメソッドを引数のリストとともに呼び出します。引数リストは式リストとそれに続く束縛リスト（キーワード引数リスト）から構成され、それぞれ空でもかまいません。

```
function_call  ::=  postfix_expression "(" argument_list? ")"
argument_list  ::=  expression_list ("," "*" atomic_expression)? ( "," binding_list )?
                    ( "," "**" atomic_expression )?
                    | binding_list ("," "**" atomic_expression)?
                    | "*" atomic_expression ("," binding_list)? ("," "**" atomic_expressio
                    | "**" atomic_expression
```

関数呼び出しの際に構文 *atomic_expression が使われるなら、atomic_expression の評価は配列型でなくてはなり

ません。この配列型の要素は、追加の固定引数のように扱われます。

関数呼び出しで **atomic_expression 構文が使われた場合、atomic_expression の値評価結果は辞書型でなければなりません。辞書の内容は追加のキーワード引数として扱われます。

引数リストの各要素は、関数呼び出しの前に評価されます。

4.5.3 演算子式

単項演算子

単項演算子には、+ と - があります。単項演算子は右結合となるため、+-x は +(-x) と同じ意味になります。

```
unary_expression  ::=   postfix_expression
                      | ( "+" | "-" ) unary_expression
```

単項 - 演算子は、引数となる数値の符号を反転します。

単項 + 演算子は、数値引数を変更しません。

乗除演算子

乗除演算子には、 *, /, % があります。これらはどれも同じ優先順位で、左結合となります。

```
multiplicative_expression  ::=   unary_expression
                                 | multiplicative_expression "*" unary_expression
                                 | multiplicative_expression "/" unary_expression
                                 | multiplicative_expression "%" unary_expression
```

* 演算は、引数間の積になります。どちらかの引数が文字列や配列で、一方が整数の場合、文字列や配列がその数だけ繰り返された値となります。たとえば、式 'foo' * 3 は'foofoofoo' に評価されます。

/ 演算は、引数間の商になります。ゼロによる除算を行うとエラーとなります。

% 演算は、2 つの引数が整数の場合、第 1 引数を第 2 引数で除算したときの剰余になります。第 1 引数が文字列で、第 2 引数が辞書の場合、テンプレート文字列を置換した結果を返します。

加減演算子

加減演算子には、+, -があります。これらはどれも同じ優先順位で、左結合となります。

```
additive_expression  ::=   multiplicative_expression
                         | additive_expression "+" multiplicative_expression
                         | additive_expression "-" multiplicative_expression
```

+ 演算は、引数を加算した値を返します。両引数が文字列や配列の場合、連結された値を返します。

- 演算は、引数間で減算を行った値を返します。

比較演算子

比較演算子には、<, >, ==, >=, <=, !=, =~, !~ があります。これらはどれも同じ優先順位をもっており、左結合となります。

```
comparison_expression  ::=   additive_expression
                           | comparison_expression "<" additive_expression
                           | comparison_expression ">" additive_expression
                           | comparison_expression "==" additive_expression
                           | comparison_expression ">=" additive_expression
                           | comparison_expression "<=" additive_expression
                           | comparison_expression "!=" additive_expression
                           | comparison_expression "=~" additive_expression
                           | comparison_expression "!~" additive_expression
```

比較の結果はブール型の値 true または false となります。比較はいくらでも連鎖することができます。たとえば、x < y <= z は x < y and y <= z と等価になります。ただし、この場合、前者では y はただ一度だけ評価されます。また、x < y <= z と (x < y) <= z は意味が異なります。後者は、x < y 評価した結果のブール型の値を z と比較します。

同じ型の値同士の比較の意味は、型によって異なります。

- 整数同士の比較では、算術的な比較が行われます。
- 文字列同士の比較では、辞書的な比較が行われます。
- 配列同士の比較では、対応する各要素の比較結果を使って辞書的な比較が行われます。
- 辞書同士の比較は等価判定のみ定義されています。キーが同じ順序で並んでいて、かつ、キーと値の対応する各要素が等しいときのみ、等価となります。

x != y は not (x == y) と等価です。

=~ は類似比較を行います。意味は型によって異なります。

- パターンと文字列の比較では、パターンマッチによる比較が行われます。

- 文字列とそれ以外の型の比較では、文字列以外の値を文字列化して比較が行われます。
- 配列同士の比較では、対応する各要素を類似比較します。
- 辞書同士の比較は、キーの順序の違いを無視して、各キーに対応する値を類似比較します。
- 上記以外は通常の == による等価比較と同じ結果となります。

x !~ y は not (x ~= y) と等価です。

包含演算子

包含演算子は、in, not in があります。これらはどれも同じ優先順位を持っており、左結合となります。

```
membership_expression  ::=  comparison_expresion
                            | membership_expression "in" membership_expression
                            | membership_expression "not" "in" membership_expression
```

包含演算 x in y は値 x が y の要素として含まれている場合は true、含まれていなければ false を返します。x not in y は not (x in y) と同じです。

y が配列型以外の値の場合、要素かどうかの判定は以下のとおりになります。

- x と y がともに文字列の場合、x が y の部分文字列の場合に要素とみなされます。
- y が辞書型の値の場合、x が y のキー集合に含まれている場合に要素とみなされます。
- y がディレクトリ/テーブル型オブジェクトの場合、x が y の子オブジェクトの場合に要素とみなされます。

論理演算子

論理演算子には、not, and, or があります。論理演算のコンテキストや式の結果として真偽値を要求される場合、false, null, 0, 空の文字列 (""), 空の配列 ([]), 空の辞書 ({}) は全て偽と解釈されます。それ以外の値は真と解釈されます。

```
logical_not_expression  ::=  membership_expression
                             | "not" membership_expression
logical_and_expression  ::=  logical_not_expression
                             | logical_and_expression "and" logical_not_expression
logical_or_expression   ::=  logical_and_expression
                             | logical_or_expression "or" logical_and_expression
expression              ::=  logical_or_expression
```

演算子 not は、引数が偽である場合には true を、それ以外の場合は false になります。

式 x and y は、式 x と y をそれぞれ評価し、x が偽なら x の評価結果を返します。それ以外の場合は y の評価結果を返します。

式 x or y は、式 x と y をそれぞれ評価し、x が真なら x の評価結果を返します。それ以外の場合は y の評価結果を返します。

4.6 ジョブ

ジョブは、コマンドの実行やイベントの待ち合わせ、あるいは繰り返しや条件分岐などの制御を指示します。ジョブの構文は以下の通りです。

```
job  ::=   skip_job
         | execution_job
         | assignment_job
         | update_job
         | event_job
         | builtin_job
         | control_job
         | block_job
```

4.6.1 スキップジョブ

スキップジョブは、$STATUS を 0 にセットする以外には何も行いません。

```
skip_job  ::=   "[" "]"
```

4.6.2 実行ジョブ

実行ジョブは、式を評価した結果の値の型によって、異なる処理を実行します。

```
execution_job  ::=   "[" expression ("<<" expression)? (":" argument_list)? "]"
```

1 番目の式を評価した結果が文字列の場合、実行ジョブは制御文脈に応じて、リモートサーバもしくはローカルサーバで文字列をコマンドと解釈して実行します。記号 << に引き続いて 2 番目の式がある場合、その式の評価結

果を文字列とみなして、コマンドの標準入力に渡されます。

1 番目の式を評価した結果がジョブフローオブジェクトの場合、そのジョブフローオブジェクトを呼び出します。引数リストがある場合には、リスト中の引数の式を評価した値がジョブフローのパラメータとして渡されます。

1 番目の式を評価した結果がスクリプトオブジェクトの場合、そのスクリプトがリモート、もしくはローカルサーバで実行されます。記号 << に引き続いて 2 番目の式がある場合、その式の評価結果を文字列とみなして、スクリプト実行時の標準入力に渡されます。引数リストがある場合には、リスト中の引数の式を評価した値がスクリプトのコマンドライン引数として渡されます。

1 番目の式を評価した結果が、Kompira オブジェクトのメソッドの場合、引数リストをパラメータとして、そのメソッドを呼び出します。

1 番目の式を評価した結果が、ライブラリオブジェクトの関数の場合、引数リストをパラメータとして、その関数を呼び出します。

警告: コマンドの文字列の長さ、スクリプトとスクリプトのコマンドライン引数のサイズは、112KB に制限されています。この制限を超えた場合、ジョブの実行は失敗し、$STATUS に -1 をセットします。

4.6.3 代入ジョブ

代入ジョブは、変数に = の右辺式の評価結果を代入します。

```
assignment_job  ::=   "[" binding_list "]"
```

変数が未定義の場合、ジョブフロースコープを持つ変数が新たに定義され、評価した値で初期化されます。

4.6.4 更新ジョブ

更新ジョブは、最初の式を評価し、その結果の値で、ターゲット式を評価した結果の変数やオブジェクト、フィールドの内容を更新します。

```
update_job         ::=   "[" expression ">>" target_expression "]"
target_expression  ::=   IDENTIFIER | OBJECT_PATH
                         | target_expression "." IDENTIFIER
                         | target_expression "[" expression "]"
```

4.6.5 イベントジョブ

イベントジョブは、1 つ目の式を評価した結果がチャネルオブジェクト、もしくは、タスクオブジェクトの時、そのオブジェクトのイベントを待ち合わせします。$RESULT には受信したオブジェクトが格納されます。

プロセスオブジェクトを渡した場合は、そのプロセスが終了するまで待ち合わせします。この場合、$RESULT にはプロセスオブジェクトが格納されます。

チャネルやタスク、プロセスオブジェクトを要素とするリストを渡した場合は、いずれかのオブジェクトのイベントを待ち合わせします。$RESULT には、後述する 2 つの要素を持つリストが格納されます。リストの 1 つ目の要素はイベントが発生したオブジェクト、2 つ目の要素は受信したオブジェクトです。

```
event_job  ::=   "<" expression (("?" | "??") expression)? (":" argument_list)? ">"
               | "<" ">"
```

? の後に式（ガード式）が続く場合、ガード付きのイベントジョブとなります。この場合、ガード式が評価された結果の値とチャネルのメッセージキューの先頭にあるオブジェクトがマッチした場合のみ、メッセージを受信します。?? の場合も同様ですが、メッセージキューの先頭から順にオブジェクトがマッチするかどうかを調べ、マッチした場合には、それまでのオブジェクトを破棄して、マッチしたオブジェクトを受信します。プロセスオブジェクトを待つ場合、ガードの指定は単に無視されます。

引数リストがある場合、タイムアウト指定となります。最初の式の値が日時型の場合はタイムアウトする日時が指定され、整数型の場合、タイムアウトまでの秒数が指定されます。タイムアウトした場合、$STATUS は 1 にセットされます。

空式の場合、常に発火するイベントとなるため、ジョブフローの実行はただちに継続します。

4.6.6 組み込みジョブ

組み込みジョブは、Kompira の組み込みジョブを呼び出します。

```
builtin_job  ::=   IDENTIFIER "(" argument_list? ")"
```

引数リストがある場合、リストの先頭か順に式が評価され、その結果が組み込みジョブのパラメータとして渡されます。

Kompira が提供する組み込みジョブの一覧と詳細については、*Kompira* 標準ライブラリを参照してください。

4.6.7 制御ジョブ

制御ジョブは、 break と continue の 2 つがあります。

```
control_job  ::=  "continue" | "break"
```

制御ジョブは、while ブロックと for ブロックの内部でのみ使用することができます。それ以外の場所で使用するとコンパイル時エラーとなります。

continue

continue は while/for ブロックの次の繰り返しの先頭に制御を移します。

break

break は while/for ブロックの繰り返しを中止し、ブロックの後続に制御を移します。

4.6.8 ブロックジョブ

ブロックジョブはブロックスコープを新しく作成します。

```
block_job  ::=  simple_block
              | if_block
              | for_block
              | while_block
              | case_block
              | choice_block
              | fork_block
              | pfor_block
              | session_block
              | try_block
```

単純ブロック

単純ブロックは、変数宣言付きの場合は、そのブロックスコープを持つローカル変数を新規に定義し、そのもとでブロック内のジョブフロー式を実行します。変数宣言が省略された場合は、単にブロック内のジョブフロー式を実行します。

```
simple_block  ::=  "{" (binding_list "|")? jobflow_expression "}"
```

if ブロック

if ブロックは、最初の条件式を評価し、その結果によって処理を分岐します。条件式が省略された場合、直前のジョブの実行結果である $RESULT の値が使用されます。

```
if_block     ::=  "{" "if" expression? "|" jobflow_expression "}"
                | "{" "if" expression? "|" then_clause elif_clause* else_clause? "}"
then_clause  ::=  "then" ":" jobflow_expression
elif_clause  ::=  "elif" expression ":" jobflow_expression
else_clause  ::=  "else" ":" jobflow_expression
```

if ブロックの 1 番目の形式は、条件式が真の場合にのみブロック内のジョブフロー式が実行されます。

if ブロックの 2 番目の形式は、最初の条件式が真の場合に、then 節のジョブフロー式が実行されます。偽の場合には、次の elif 節の条件式が評価され、その値が真の場合に、elif 節のジョブフロー式が実行されます。どの条件式も偽の場合で、最後の else 節があれば else 節のジョブフロー式が実行されます。

case ブロック

case ブロックは最初の式を評価し、その値と各ケース節のパターン式を評価した値とのマッチングが試みられます。マッチングが成功すると、対応する case 節のジョブフローを実行します。case 節のパターン式がカンマ区切りで複数記述された場合、どれか一つのパターンとマッチするとマッチング成功とみなされ、その節のジョブフローが実行されます。

最初の式が省略された場合、直前のジョブの実行結果である $RESULT の値が使用されます。

```
case_block   ::=  "{" "case" expression? "|" case_clause+ else_clause? "}"
case_clause  ::=  expression_list ":" jobflow_expression
```

case 節は、パターン式の後に、区切り記号であるコロン (:) とそのパターンに合致したときに実行するジョブフロー式が続きます。パターン式を評価した結果がパターンオブジェクトの場合、そのパターンオブジェクトに応じたマッチングが試みられます。パターン式の評価結果が文字列の場合、大文字小文字を区別する Glob パターンとして扱われます。それ以外の場合は、単純な == 比較によるマッチングを行います。

パターンは case 節の先頭から順番にマッチングが試みられます。どのパターンにもマッチしない場合、else 節があればそのジョブフロー式が実行されます。else 節が無ければ、マッチングに失敗したとみなされ、 $STATUS に

1 がセットされます。

for ブロック

for ブロックはリストやディレクトリ、テーブルなどの複数の要素を含むオブジェクト内の要素にわたって反復処理を行うために使われます。

```
for_block  ::=   "{" "for" IDENTIFIER "in" expression "|" jobflow_expression "}"
```

式は for ブロックの実行時に最初の一度だけ評価されます。式の評価結果は反復可能なオブジェクトか、もしくは、整数値である必要があり、それ以外の場合は実行エラーとなります。オブジェクトの各要素は識別子 (IDENTIFIER) で示されるローカル変数に代入されます。式の評価結果が整数値 N の場合、ローカル変数が 0 から N-1 の範囲で反復します。ただし、N が 0 もしくは負の場合は反復しません。このローカル変数は for ブロックのスコープを持つため、for ブロックを抜けた後で参照することはできません。

ジョブフロー式の中で break ジョブが実行されると、ループを終了します。continue ジョブが実行されると、ジョブフロー式の後続の処理をスキップしてループを終了します。

for ブロック終了時の $STATUS は常に 0 にセットされます。

while ブロック

while ブロックは、式を繰り返し評価し、真であればジョブフロー式を実行します。式が偽であれば、while ブロックは繰り返しを終了します。

```
while_block  ::=   "{" "while" expression "|" jobflow_expression "}"
```

ジョブフロー式の中で break ジョブが実行されると、ループを終了します。continue ジョブが実行されると、ジョブフロー式の後続の処理をスキップして、式の評価に戻ります。

while ブロック終了時の $STATUS は常に 0 にセットされます。

choice ブロック

choice ブロックは複数のイベントジョブを待ち、いずれかが実行可能状態になると、そのイベントジョブに後続するジョブフロー式を実行します。

```
choice_block          ::=   "{" "choice" "|" eventflow_expression+ "}"
eventflow_expression  ::=   event_job ("->" | "=>" | "->>" | "=>>") jobflow_expression
```

複数のイベントジョブが同時に実行可能状態になった場合は、先頭に近いイベントジョブが優先されます。

fork ブロック

fork ブロックは、ジョブフロー式を子プロセスとして実行開始します。

```
fork_block  ::=   "{" "fork" "|" jobflow_expression+ "}"
```

fork ブロックは detach() していない全ての子プロセスが実行完了するまで待ちます。fork ブロック終了時の $RESULT には fork ブロックで生成した全ての子プロセスのリストがセットされ、$STATUS には異常終了した子プロセスの個数がセットされます。すべての子プロセスが正常終了すると $STATUS は 0 になります。

fork ブロックによって生成されるプロセスが、プロセス数の制限を超える場合、他のプロセスが実行完了し、プロセス数制限内に収まるまで fork ブロックは実行を待ちます。

pfor ブロック

pfor ブロックはリストやディレクトリ、テーブルなどの複数の要素を含むオブジェクト内の要素について、子プロセスを生成して並行処理を行います。

```
pfor_block  ::=   "{" "pfor" IDENTIFIER "in" expression "|" jobflow_expression "}"
```

式は pfor ブロックの実行時に最初の一度だけ評価されます。式の評価結果は反復可能なオブジェクトである必要があり、それ以外の場合は実行エラーとなります。オブジェクトの各要素について子プロセスが生成され、各子プロセス中で対応する要素が識別子 (IDENTIFIER) で示されるローカル変数に代入されて、子プロセスの実行が開始されます。式の評価結果が整数値 N の場合、ローカル変数が 0 から N-1 のそれぞれで子プロセスの実行が開始されます。ただし、N が 0 もしくは負の場合、子プロセスは実行されません。

pfor ブロックは detach() していない全ての子プロセスの実行が終了するまで待ちます。pfor ブロック終了時の $RESULT には pfor ブロックで生成した全ての子プロセスのリストがセットされ、$STATUS には異常終了した子プロセスの個数がセットされます。すべての子プロセスが正常終了すると $STATUS は 0 になります。

pfor ブロックによって生成されるプロセスが、プロセス数の制限を超える場合、他のプロセスが実行完了し、プロセス数制限内に収まるまで pfor ブロックは実行を待ちます。

session ブロック

session ブロックは、リモートサーバとのセッションを開始します。

```
session_block  ::=   "{" "session" IDENTIFIER "|" jobflow_expression "}"
```

セッションブロックが実行されると、まず、制御変数によって指定されたリモートサーバとのセッションを開始します。セッションでリモートサーバとのやりとりを行うためのセッションチャネルが、識別子 (IDENTIFIER) で示されるローカル変数に代入されます。このセッションチャネルに対して、文字列を送信 (send) すると、リモートサーバ側に文字列が送信されます。また、リモートサーバ側からの出力を取得するには、イベントジョブを用いてセッションチャネルからデータを取得します。リモートサーバからの出力は行単位のメッセージとしてセッションチャネルに格納されていきます。したがって、セッションチャネルからのメッセージの読み込みは 1 行ずつとなります。

セッションブロックを抜けるとセッションが終了し、とセッションチャネルがクローズされます。以後、セッションチャネルに対する送信はエラーとなります。また、セッションチャネルからのメッセージの読み込みもエラーとなります。(ただし、セッションクローズまでにリモートサーバから出力されたメッセージは読み込むことができます)

セッションブロック内で break を呼び出すと、セッションをクローズして、ブロックを終了します。

セッションブロックが正常に終了すると、$STATUS に 0 がセットされます。また、$RESULT には、セッションチャネルが格納されます。セッションチャネルの data 属性には、未読み込みのデータが格納されます。(メッセージの各行は連結され、1 つの文字列データとなります)

セッションの開始に失敗した場合は、セッションブロックの中は実行されずにセッションブロックは終了し、$STATUS には 非 0 がセットされます。また、$ERROR にはエラーの原因を示すメッセージが格納されます。

注釈: セッションブロック内で、コマンドジョブを実行することは可能ですが、別のセッションを新たに開始することはできません。

注釈: セッションブロックは、現時点で、ssh モードによる接続のみサポートされています。制御変数を winrs モードや local モードにセットしてのセッションブロック実行はエラーとなります。

以下では、su コマンドを実行して、対話的に処理を行うジョブフローのプログラム例を示します。

```
[__host__ = 'server.exmaple.com', __user__ = 'testuser', __password__ = 'password',
 __use_pty__ = true   # su コマンド実行には PTY が必要なため true にセット
] ->
# server.example.com にログインしてセッションを開始
{ session s |
    [s.send: 'LANG=C su\n'] ->                # su コマンドを実行
    <s ?? 'Password: '> ->                    # パスワードプロンプトを待つ
    [s.send: 'root_password\n'] ->            # root のパスワードを送信
    <s ?? g'*]# '> ->                         # root プロンプトを待つ
    [s.send: 'service httpd restart\n'] ->    # httpd サービスを再起動
```

(次のページに続く)

(前のページからの続き)

```
    <s ?? g'*]# '> ->                          # root プロンプトを待つ
    [s.send: 'exit\n']                         # root を 抜ける
} ->
print('OK')
```

try ブロック

try ブロックは、ブロック内のジョブフロー実行中に発生した異常終了をキャッチして、処理を続行します。

```
try_block  ::=   "{" "try" "|" jobflow_expression "}"
```

try ブロックで囲まれたジョブフローが正常に終了した場合は、try ブロックは $STATUS に 0 をセットし、 異常終了した場合は、$STATUS に 1 をセットします。また、$DEBUG にデバッグ情報を格納します。

try ブロック内のジョブフローを実行中に exit が呼び出された場合は、ジョブフローは常に終了します。また、try ブロック内のジョブフローを実行中に、ユーザーによってジョブフローの実行が中止された場合も、ジョブフローは実行を終了します。

4.7 ジョブフロー式

ジョブフロー式は、ジョブを結合子で結びつけた式です。

```
jobflow_expression  ::=   jobflow_expression "->" job
                        | jobflow_expression "=>" job
                        | jobflow_expression "->>" job
                        | jobflow_expression "=>>" job
```

4.7.1 結合子

結合子には複数の種類があり、ジョブが失敗した場合にジョブフローの処理を継続するかどうかが異なります。

以下に結合子の一覧と、ジョブが失敗した時の振る舞い、処理を継続する場合の状態変数の値を示します。

結合子	コマンド異常終了時	リモートログイン失敗時
->	強制終了	強制終了
=>	処理継続 $STATUS = 1-255 $RESULT = (stdout) $ERROR = (empty)	強制終了
->>	強制終了	処理継続 $STATUS = -1 $RESULT = (empty) $ERROR = (error message)
=>>	処理継続 $STATUS = 1-255 $RESULT = (stdout) $ERROR = (empty)	処理継続 $STATUS = -1 $RESULT = (empty) $ERROR = (error message)

リモートコマンドの実行ステータスが 0 以外の場合は、上表の「コマンド異常終了時」の動作となります。このとき、リモートコマンドの実行ステータスの値が$STATUS の値となります。

ssh がタイムアウトした場合や、ジョブフロー上で指定された IP アドレス、ユーザ名、パスワード等が正しくなかった場合は、上表の「リモートログイン失敗時」の動作となります。

バージョン 1.5.4.post5 で変更: リモートログイン失敗時の $ERROR にはエラーの原因を示すメッセージが格納されます。

4.8 ジョブフロープログラム

ジョブフロープログラムは、0 個以上のパラメータ宣言とそれに続くジョブフロー式から構成されます。ジョブフロー式が空の場合は、ジョブフロープログラムは実行省略することができます。

```
jobflow_program  ::=   (parameter_declaration)* jobflow_expression?
```

4.8.1 パラメータ宣言

パラメータ宣言は、以下の形式をとります。

```
parameter_declaration  ::=   "|" IDENTIFIER ("=" expression)? "|"
```

パラメータ宣言で、parameter_declaration = expresssion の形式がある場合、そのジョブフローはデフォルトのパラメータを持ちます。デフォルト値を持つパラメータに対しては、ジョブフロー呼び出しの際に対応するパラメータが省略されると、パラメータの値はデフォルト値で置き換えられます。デフォルトパラメータ式は、ジョブフロー呼び出しのたびに値評価されます。

第 5 章

Kompira 標準ライブラリ

著者　Kompira 開発チーム

このライブラリリファレンスマニュアルでは、Kompira に標準で付属しているライブラリについて説明します。

5.1 組み込み関数/ジョブ

Kompira のジョブには組み込みジョブ、組み込み関数としてあらかじめ定義されているものがあります。

組み込みジョブは、ジョブマネージャを介さないローカルな組み込みジョブと、ジョブマネージャによって実行されるリモートの組み込みジョブの 2 種類に分けられます。

5.1.1 ローカル組み込みジョブ

ローカル組み込みジョブは、ジョブマネージャが動作していなくても実行可能なジョブです。

self ()　自分自身のジョブフローを最初から再実行します。再実行のとき、そのジョブフローのパラメータは変更しません。また、多重度指定されているジョブフローの場合、ロックを保持したまま再実行します。

print ([*message*, [*args*, ...]])　コンソールに message 文字列を出力し、改行します。

複数の引数を与えた場合、複数のメッセージ文字列を空白文字で区切って出力します。引数を全て省略すると、改行だけします。

sleep (*timeout*)　プロセスを timeout で指定した秒数だけスリープします。timeout が日時型の場合は、その日時までスリープします。

exit ([*status=0*, [*result=''*, [*error=''*]]])　プロセスを終了します。status で終了ステータスコードを指定することもできます。result でプロセス終了時の実行結果を指定します。error で、エラーメッセージを指定することもできます。

return ([*result=''*, [*status=0*, [*error=''*]]]) ジョブフローの呼び出し元に制御を戻します。result で実行結果を指定します。status で終了ステータスコードを指定することもできます。error でエラーメッセージを指定することもできます。

abort ([*message*]) コンソールにメッセージを出力し、ジョブを異常終了させます。終了ステータスコードは 1 にセットされます。

assert (*value*, [*message*]) value が真であることを確かめ、そうでない場合には、コンソールに message を出力し異常終了します。

detach () 実行中の自身のプロセスを親プロセスから切り離します。これによって、親プロセス側は子プロセスの終了を待たずに処理を先に進めることができます。

suspend () 実行中のプロセスを一時停止状態にします。

urlopen (*url*, [*user*, *password*, *data*, *params*, *files*, *timeout*, *encode*, *http_method*, *verify*, *quiet*, *headers*, *cookies*, *charset*, *b*
引数で指定した url に対して、HTTP リクエストを送信し、結果を取得します。

user, password を指定すると、Basic 認証によるアクセスを行います。

data には POST リクエストで送信するデータを辞書型で指定することができます。送信データは encode 引数で指定した方式でエンコードされます。

params に辞書を渡すと URL のクエリ文字列として展開されます。例えば以下のように呼び出すと、実際にアクセスする URL は `http://example.com?key1=value1&key2=value2` となります。:

```
urlopen(url='http://example.com', params={key1='value1', key2='value2'})
```

files には、アップロードするファイルを渡すことができます。ファイルは name と data というフィールドを持つ辞書，ファイル名とコンテンツのリスト，Kompira サーバ上のファイル名，または添付ファイルフィールド，のいずれかの形式で指定することができます。:

```
files={file={name='filename', data='content'}}
files={file=['filename', 'content']}
files={file="/tmp/filename.xls"}
files={file=./attached_file.attached1}
```

なお、ここで files に渡す辞書のキー（上では file）は送信先 form のファイルフィールドの名前に合わせて指定してください。複数ファイルを受け付けるファイルフィールドに対しては、リスト形式でフィールド名とファイルを並べて渡すこともできます。:

```
files=[['file', {name='filename1', data='content1'}],
       ['file', {name='filename2', data='content2'}]]
```

この場合は、内側のフィールド名とファイルもリストで指定してください。files が指定された場合は multipart/form-data 形式でエンコードされます。

timeout には、タイムアウトするまでの時間を秒単位で指定します。

encode には、エンコードタイプとして "json"を指定することができます。data が指定され encode が "json" のとき、HTTP リクエストの Content-Type: ヘッダには自動的に application/json が設定されます。encode 引数を省略した場合、送信データは application/x-www-form-urlencoded 形式でエンコードされます。files が指定されている時に encode に "json" を指定するとエラーになります。

http_method には、HTTP リクエストのメソッドを 'GET', 'POST', 'PUT', 'DELETE', 'HEAD' のいずれかから指定します。http_method を省略したとき、data または files が指定されている場合は POST メソッド、指定されていない場合は GET メソッドとなります。

verify に true を指定すると、指定した URL が https アクセスである時に SSL 証明書のチェックを行います。不正な SSL 証明書を検出すると urlopen ジョブはエラーになります。verify のデフォルト値は false です。

quiet に true を指定すると、verify オプションが true 時に、https アクセスした際に表示される警告メッセージを抑止します。

headers には、HTTP リクエストに設定するヘッダ情報を辞書型の値で渡すことができます。

cookies には、サーバに渡すクッキーを辞書型の値で渡します。

charset には、レスポンスとして期待する文字コードを指定することができます。

プロキシサーバ経由で HTTP リクエストを送信する必要がある場合は、以下の例の用にプロキシサーバの URL の辞書を proxies パラメータに渡します。:

```
[proxies = {'http': 'http://10.10.1.10:3128', 'https': 'http://10.10.1.10:1080'}]␣
↪->
urlopen('http://www.kompira.jp', proxies=proxies)
```

取得したコンテンツがバイナリであるかどうかは Content-Type で判断します。Content-Type が image|audio|video で始まるとき、または octetl|binary を含む時はバイナリと判断します。ただし binary に true を指定すると、Content-Type によらずコンテンツをバイナリとして扱います

この組み込みジョブは、以下の要素を持つ辞書型の値を返します。

フィールド名	意味
url	レスポンスの URL
code	結果ステータスコード
version	HTTP バージョン（HTTP 1.1 なら 11 という数値になります）
text	レスポンスの内容（エンコード情報をもとにレスポンス本文をテキストにデコードしたもの。ただし、バイナリコンテンツの場合は空文字になります）
content	レスポンスの内容（バイナリのままのレスポンスの本文）
body	レスポンスの内容（コンテンツがバイナリであると判断したとき content と同じ値になり、テキストであると判断したときは text と同じ値になります）
encoding	エンコード情報
headers	レスポンスに含まれるヘッダ情報（辞書型）
cookies	サーバから渡されたクッキー値（辞書型）
history	リダイレクトがあった場合にその履歴情報（リスト型）
binary	バイナリコンテンツであるかの真偽値

mailto (*to*, *from*, *subject*, *body*, [*cc*, *bcc*, *reply_to*, *html_content*, *attach_files*, *parents*, *headers*, *charset*, *reply_to_all*, *inline_*

メールを送信します。

to には、送信先メールアドレスを文字列で指定します。複数アドレスに送信したい場合、送信先メールアドレスを要素に持つリストで指定します。from には、送信元のメールアドレスを指定します。subject には、メール表題の文字列を指定します。body には、メール本文の文字列を指定します。cc と bcc には、Cc/Bcc 先メールアドレスをそれぞれ指定します。複数アドレスを指定したい場合は、リストを渡します。reply_to には、返信先のメールアドレスを指定します。

html_content を指定すると HTML 形式 (text/html) のメールを送信します。html_content を省略または null を指定した場合、body をテキスト形式 (text/plain) で持ったメールを送信します。body および html_content の両方を null にすると、mailto ジョブはエラーになります。attach_files には、メールに添付するファイルオブジェクト、もしくはファイルオブジェクトの一覧を渡すことができます。

parents に親メッセージ（mail_parse した結果の辞書）を渡すと、そのメッセージに対する返信としてのメールを送信します。このときメールヘッダ In-Reply-To: および References: が適切に設定されます。また宛先として親メッセージの（設定されていれば） Reply-To: または From: に設定されたアドレスが設定されます。複数の親メッセージを参照する場合はリストで渡してください。parents でメールを返信する際に reply_to_all を true を指定すると、「全員に返信」扱いとして親メッセージの To: と Cc: に指定された宛先を引き継いで設定します。

headers に辞書を渡すと、辞書の各キーをヘッダ項目としてメールヘッダに追加します。

charset で送信時の文字コードを指定できます。省略時は UTF-8 です。

inline_content に true を指定すると添付ファイルをインライン展開します。このときメール本体の MIME mixed サブタイプは "related" となり、各添付ファイルの Content-Disposition ヘッダは "inline" となります。また、メール本文 (body, html_content) 中に "%{Content-ID#num}"（num 部分は attache_files で指定した添付ファイルのインデックス）または"%{Content-ID:filename}"（filename 部分は添付ファイルのファイル名）と指定されたプレースホルダがある場合、各添付ファイルに自動的に割り当てられた Content-ID（両端の'<' と'>' を取り除いたもの）に置換されます。例えば、attach_files で画像ファイルを 1 つ添付し、html_content に '' という記述を含んでおくと、対応したメールクライアントではその記述部分に添付した画像ファイルがインラインで表示されるようになります。

as_string に true を指定するとメールを実際に送信するかわりに、メールヘッダを含むメッセージ全体を文字列化します。結果の文字列は$RESULT で参照できます。

mailto ジョブで送信したメールの User-Agent ヘッダは "Kompira ver X.XX" となります。"X.XX" の部分は Kompira のバージョンが入ります。

注釈: from が省略された場合、次の優先順位で送信元のメールアドレスを決定します。

1. プロセスオーナーのメールアドレス
2. /system/config の管理者メールアドレス
3. webmaster@localhost

---

download (*from_file*, *to_path*)　添付ファイルフィールドのファイルを指定したパスにダウンロードします。from_file には、ダウンロード元の添付ファイルフィールドオブジェクトを指定します。

to_path には、ダウンロード先のファイルパスを指定します。ダウンロード先はジョブマネージャの動作しているサーバ上のファイルシステムになります。ダウンロード先のファイルパスがディレクトリを指している場合、ファイル名は添付ファイルのファイル名となります。

以下は Kompira オブジェクト/root/Package の attached フィールドに添付されたファイルをローカルの/tmp ディレクトリにダウンロードします。:

```
download(from_file=/root/Package.attached, to_path='/tmp/')
```

upload (*from_path*, *to_object*, *to_field*)　添付ファイルフィールドに指定したファイルをアップロードします。結果は添付ファイルのファイル情報を返します。from_path には、ダウンロード元のファイルパスを指定します。to_object には、添付先の Kompira オブジェクトを指定し、to_field で添付先 Kompira オブジェクトの添付ファイルフィールド名を指定します。

以下はローカルに置かれたファイル/tmp/foo.tar.gz を/root/Package オブジェクトの attached フィールドにアップロードします。:

```
upload(from_path='/tmp/foo.tar.gz', to_object=/root/Package, to_field='attached')
```

5.1.2 リモート組み込みジョブ

リモート組み込みジョブは、ジョブマネージャを介して動作する組み込みのジョブです。ジョブマネージャが動作していない場合は、ジョブマネージャが起動するまで実行が待ち状態となります。

また、リモート組み込みジョブでは、制御変数からリモートホストの接続情報を参照します。

put (*local_path*, *remote_path*)

ジョブマネージャが動作しているホストからリモートホストにファイルを転送します。結果は転送先のファイルパスのリストを返します。

local_path には、転送元のファイルパスを指定します。ワイルドカードを用いて複数ファイルを転送することも可能です。local_path を相対パス指定した場合、ジョブマネージャが動作しているディレクトリ (通常はルートディレクトリ) からの相対パスとなります。

remote_path には、転送先のディレクトリパスもしくはファイルパスを指定します。remote_path を相対パス指定した場合、ログインユーザーのホームディレクトリからの相対パス、もしくは、__dir__制御変数で指定したパスからの相対パスとなります。

get (*remote_path*, *local_path*)

リモートホストからジョブマネージャが動作しているホストにファイルを転送します。結果は転送先のファイルパスのリストを返します。

remote_path には、転送元のファイルパスを指定します。ワイルドカードを用いて複数ファイルを転送することも可能です。

local_path には、転送先 (ジョブマネージャ側) のファイルパスか、ディレクトリパスを指定します。

reboot ([*wait=120*])　リモートホストを再起動します。

wait には、リモートホストが再起動するまで待つ最大時間を指定 (単位は秒) します。

reboot ジョブは、sudo ジョブが実行できるユーザでなければ実行できません。

5.1.3 組み込み関数

組み込み関数は、Kompira の式として使用できる関数です。

式の中に記述する他に、組み込みジョブと同じように単独で使用することもできます。単独で使用した場合、結果は$RESULT に挿入されます。

now ()　現在時刻を取得します。

current ()　現在実行している自身のプロセスオブジェクトを取得します。

channel ()　データを複数プロセス間で送受信するためのオンメモリのチャネルオブジェクトを作成します。

datetime (*dt_str_or_date*, [*dt_fmt_or_time*, *zone*])　dt_str_or_date で指定された文字列から日時データに変換します。dt_fmt_or_time で、フォーマット文字列を指定することも可能です。dt_str_or_date に日付型データを渡し、dt_fmt_or_time に時刻型データを渡すことで、これらを合わせた日時型データを構成することができます。zone で、タイムゾーン ID を指定することが可能です。

フォーマットの書式指定は、C 言語の strftime() 関数に準じています。以下に例を示します。

```
[dt = datetime('2015年1月1日10時30分05秒', '%Y年%m月%d日%H時%M分%S秒', 'Asia/
↪Tokyo')] ->
print(dt) ->
[dt2 = datetime(dt.date, dt.time)] ->
print(dt2)
```

注釈: dt_fmt を省略した場合、日付文字列のフォーマットは以下のような ISO 8601 形式として変換を試みます。

YYYY-MM-DD[T]hh:mm(:ss(.mmmmmm)?)?([Z]|[+-]hh(:)?mm)?

日付けと時刻のセパレータは、T または空白が使えます。秒、マイクロ秒、タイムゾーンの指定は省略可能です。

zone を省略した場合、ローカルのタイムゾーンが指定されたとみなされます。

---

date (*date_str*, [*dt_fmt*])　date_str で指定された文字列から日付データに変換します。dt_fmt で、フォーマット文字列を指定することも可能です。

フォーマットの書式指定は、C 言語の strftime() 関数に準じています。

time (*time_str*, [*dt_fmt*])　time_str で指定された文字列から時刻データに変換します。dt_fmt で、フォーマット文字列を指定することも可能です。

フォーマットの書式指定は、C 言語の strftime() 関数に準じています。

timedelta (*days=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*)　経過時間を表すデータを作成します。timedelta 型の値と datetime 型の値は加算や減算が可能です。

int (*x=0*)　引数 x で与えられた文字列を整数型に変換します。

float (*x=0.0*)　数または文字列 x から生成された浮動小数点数を返します。

pattern (*pattern*, *typ='r'*, *mode=''*)　引数 pattern で与えられるパターンオブジェクトを生成します。typ はパターンの種別を表し、'r' (正規表現パターン)、'g' (glob パターン)、'e' (完全一致パターン) のいずれかを指定することができます。mode に'i' を指定すると大小文字を区別しないパターン照合となります。

path (*str_or_obj*, [*args*, ...]) パス名を表す文字列 str_or_obj から実際の Kompira オブジェクトを返します。str_or_obj には文字列の配列を指定することができます。配列、もしくは複数の引数を与えた場合、各要素を結合しパス名として解釈します。

以下の例はルートディレクトリ直下のオブジェクトを列挙します。使用例

```
{ for p in path('/') | print(p) }
```

str_or_obj に相対パスを指定した場合は、このジョブフローがあるディレクトリから相対位置にある Kompira オブジェクトを参照します。以下の例は、このジョブフローがあるディレクトリのパスを表示します。

使用例

```
print(path('.'))
```

また、str_or_obj には Kompira オブジェクトを指定することもできます。以下の例では、パラメータ 'dir' で指定された Kompira オブジェクトの親ディレクトリに含まれるオブジェクト、すなわち 'dir' で指定されたオブジェクトと同一階層にある Kompira オブジェクトを列挙します。

使用例

```
|dir = /home/guest|
{ for sibling in path(dir, '..') | print(sibling) }
```

user (*user*) ユーザー名 user を持つ User オブジェクトを返します。user に整数値を与えると、その値をユーザ ID として持つ User オブジェクトを返します。User オブジェクトを与えると、それをそのまま返します。

group (*group*) グループ名 group を持つ Group オブジェクトを返します。group に整数値を与えると、その値をグループ ID として持つ Group オブジェクトを返します。Group オブジェクトを与えると、それをそのまま返します。

string (*obj*) オブジェクト obj を文字列に変換します。

type (*obj*) オブジェクト obj の型名を返します。

decode (*data*, [*encoding='utf-8'*]) バイナリデータ data を encoding で指定された文字コード系で文字列にデコードします。

encode (*message*, [*encoding='utf-8'*]) 文字列 message を encoding で指定された文字コード系でバイナリデータにエンコードします。

length (*obj*) obj で渡された配列の長さを取得します。

has_key (*obj*, *key*) obj で渡された辞書データやオブジェクトに、指定されたキー key でフィールドアクセス可能かどうかをチェックします。

json_parse (*data*) JSON 形式で直列化された文字列を Kompira のオブジェクトに変換します。

使用例

```
[str = '[1,2,3,true,"foo","bar"]']
-> [obj = json_parse(str)]
-> { for elem in obj | print(elem) }
```

json_dump (*obj*)　Kompira のオブジェクトを JSON 形式で直列化された文字列に変換します。

mail_parse (*data*)　MIME 形式の文字列を Kompira の辞書オブジェクトに変換します。

メールのヘッダ情報に加え、'Body' キーでメールの本文に、'Filename' キーでファイル名にアクセスすることができます。(添付ファイルが存在しない場合は'Filename' キーは null となります)

メールの本文は、Content-Type が Text/plain で、かつ添付ファイルでない場合に限り utf-8 形式のエンコードされます。

Content-Type が multipart である場合は、'Is-Multipart' キーが true になり、'Body' キーの要素が Kompira 辞書オブジェクトの配列となります。

iprange (*address*)　CIDR 表記のネットワークアドレスを IP ネットワークオブジェクトに変換します。

使用例

```
{ for ip in iprange('192.168.0.1/24') |
    [__host__ = ip ] ->
    ['hostname'] ->> []
}
```

> **警告:** 組み込みジョブ iprange() は近い将来廃止される予定です。

5.2 Kompira オブジェクト

Kompira が扱う様々なデータは、ディレクトリ構造を備えた Kompira のファイルシステム上に Kompira オブジェクトとして格納されます。Kompira オブジェクトは、その型毎に固有のフィールドやメソッドを備えており、ジョブフロー上から操作することが可能です。

5.2.1 フィールド型

Kompira オブジェクトのフィールドで使用できる型は以下のとおりです。なお、コロン（:）の右側は、当該フィールドをジョブフローから参照した場合のデータの型を示しています。

String　[String] 文字列のフィールドを表します。

Integer [Integer] 整数のフィールドを表します。整数以外の値は入力できません。未入力の場合は、null 値となります。

Boolean [Boolean] 真偽値のフィールドを表します。フォーム上ではチェックボックスとして表示され、チェック時が true、未チェック時が false に対応します。

Enum [String] 選択肢フィールドを表します。選択肢の一覧はフィールド修飾子によって規定します。

Text [String] テキストフィールドを表します。

LargeText [String] 大きめのテキストフィールドを表します。

Password [String] パスワードフィールドを表します。フィールド表示時に文字列は隠されます。

File [File] 添付ファイルフィールドを表します。添付ファイルのアップロードやダウンロードができるようになります。

Object [Object] Kompira オブジェクトフィールドを表します。Kompira オブジェクトを選択肢の中から選べるようになります。フィールド修飾子を指定することで、特定の型を持つオブジェクトや、特定のディレクトリ下にあるオブジェクトに選択肢を制限することも可能です。

Datetime [Datetime] 日時フィールドを表します。入力する日時情報の形式は以下のとおりです。

フォーマット	例
%Y-%m-%d %H:%M:%S	2006-10-25 14:30:59
%Y-%m-%d %H:%M	2006-10-25 14:30
%Y-%m-%d	2006-10-25
%m/%d/%Y %H:%M:%S	10/25/2006 14:30:59
%m/%d/%Y %H:%M	10/25/2006 14:30
%m/%d/%Y	10/25/2006
%m/%d/%y %H:%M:%S	10/25/06 14:30:59
%m/%d/%y %H:%M	10/25/06 14:30
%m/%d/%y	10/25/06

Date [Date] 日付フィールドを表します。

Time [Time] 時刻フィールドを表します。

IPAddress [String] IP アドレスフィールドを表します。IPv4 アドレス形式の入力に対応しています。

EMail [String] メールアドレスフィールドを表します。

URL [String] URL フィールドを表します。

Array [Array] 配列フィールドを表します。複数の文字列要素を入力できます。

Dictionary　[Dictionary] 辞書フィールドを表します。複数のキーと値を入力できます。キーと値は文字列型のデータとなります。

5.2.2 フィールド修飾子

フィールド修飾子は、フィールド型に対してさらに細かいフィールド表示の制御や制約を加えます。フィールド修飾子は以下に示すような JSON オブジェクトの形式で記述します。

```
{ "<フィールド修飾子名 1>" : <値 1>, "<フィールド修飾子名 2>" : <値 2>, ... }
```

フィールド修飾子は以下の種類があります。

default: 全フィールド　フィールドのデフォルト値を指定します。

invisible: 全フィールド　フォームやビューからフィールドを隠します。

help_text: 全フィールド　フィールドについての説明を記述します。この修飾子を指定した場合、オブジェクトの編集時に指定したテキストが表示されます。

object: Object 型フィールド　オブジェクト型フィールドにおいて、選択肢を絞り込みます。型オブジェクトのパスを記述するとその型を持つオブジェクトが選択肢として表示されます。また、ディレクトリやテーブルのパスを指定すると、そのオブジェクトの子オブジェクトが選択肢として表示されます。

以下にジョブフロー型のオブジェクトを選択肢とする例を示します。

```
{ "object" : "/system/types/Jobflow" }
```

この修飾子を指定しなかった場合は、すべてのオブジェクトが選択肢として表示されるようになります。

directory: Object 型フィールド　オブジェクト型フィールドにおいて、選択肢を絞り込みます。ディレクトリやテーブルのパスを記述すると、その子孫オブジェクトが選択肢として表示されます。

先頭に "~" または "~(ユーザ名)" と記述したパスを指定すると、その部分がユーザのホームディレクトリに展開されます。ユーザ名を省略した場合はログイン中のユーザが対象になります。

修飾子 object と合わせて指定することで、あるディレクトリ配下にある特定の型のオブジェクトを選択させる、といったことが可能になります。

例

```
{ "object" : "/system/types/NodeInfo", "directory" : "~" }
```

ただし、存在しないまたは読み込み権限のないディレクトリを指定した場合は無視されます。

no_empty: Object 型フィールド　Object 型フィールドの入力フォームにおいて、空の選択肢を許さないようにします。

例

```
{ "object" : "/system/types/TypeObject", "no_empty" : true }
```

enum: Enum 型フィールド　Enum 型フィールドにおいて、選択肢となる文字列の一覧を指定します。

例

```
{ "enum" : ["サーバー", "スイッチ", "ルーター"] }
```

格納されるデータと表示名を別にする場合は、["<データ>", "<表示名>"] のペアを用いて、以下のように指定することもできます。

```
{ "enum" : [["SV", "サーバー"], ["SW", "スイッチ"], ["RT", "ルーター"]] }
```

pattern: String 型フィールド　文字列型フィールドの入力フォームにおいて、入力可能なパターンを正規表現で指定します。

min_length, max_length: String 型フィールド　文字列型フィールドの入力フォームにおいて、最小長および最大長を指定します。

min_value, max_value: Integer 型フィールド　整数型フィールドの入力フォームにおいて、最小値および最大値を指定します。

file_accept: File 型フィールド　ファイル型フィールドにおいて、選択可能なファイル種別を指定します。

例

```
{ "file_accept" : ".xls" }
```

バージョン 1.5.1 で追加: 新しいフィールド修飾子: pattern, min_length, max_length, min_value, max_value, file_accept が追加されました。

5.2.3 プロパティ

Kompira オブジェクトは、以下のプロパティを提供しています。

id　オブジェクトの ID の値です。オブジェクト ID は、オブジェクト生成時に自動で割り当てられる一意な整数値です。更新はできません。

abspath　オブジェクトの絶対パスの値です。更新はできません。

name　オブジェクト名の値です。オブジェクト名に使用できる文字列の形式は、Kompira ジョブフロー言語における識別子と同じです。同じディレクトリ中に同名のオブジェクトを作成することはできません。

description　オブジェクトを説明する文字列となります。

display_name　オブジェクトの表示名です。表示名の文字列にはオブジェクト名のような形式の制限はありません。

field_names　オブジェクトが備えるフィールド名の一覧です。リスト型の値となります。更新はできません。

owner　オブジェクトの所有ユーザーです。ユーザーオブジェクトとなります。

created　オブジェクトの作成日時です。日時型の値となります。更新はできません。

updated　オブジェクトの更新日時です。日時型の値となります。更新はできません。

parent_object　オブジェクトの親オブジェクト、すなわちディレクトリ（もしくはテーブル）オブジェクトです。更新はできません。

children　オブジェクトの子オブジェクト一覧です。オブジェクトが、テーブルかディレクトリ以外の場合など、子オブジェクトを持たない場合は、空のリストとなります。更新はできません。

type_object　オブジェクトの型オブジェクトです。更新はできません。

type_name　オブジェクトの型名です。更新はできません。

user_permissions　ユーザパーミッション情報です。writable, readable, executable をキーに持つ辞書型のオブジェクトとなります。

group_permissions　グループパーミッション情報です。writable, readable, executable, priority をキーに持つ辞書型のオブジェクトとなります。

5.2.4 メソッド

Kompira オブジェクトは、以下のメソッドを提供しています。

delete　オブジェクトを削除します。

update　[[*key1*=*val1*, *key2*=*val2*, ...]] オブジェクトのフィールド key1, key2, ... の値を val1, val2, ... に更新します。

rename　[*name*] オブジェクトの名前を name に変更します。

5.3 組み込み型オブジェクト

このセクションでは、Kompira にあらかじめ組み込まれている標準の型オブジェクトについて説明します。

Kompira のオブジェクトは、型オブジェクトで示される型を持ちます。例えば、ジョブフローオブジェクトは、ジョブフロー型を持ちますし、ディレクトリオブジェクトはディレクトリ型を持ちます。Kompira では、ジョブフロー型やディレクトリ型といった型もオブジェクトとして定義されているので、これらも型オブジェクトという型を持っています。ちなみに、型オブジェクトの型は型オブジェクトです。

Kompira のオブジェクトは、その型に特有のフィールドやメソッドを備えています。

5.3.1 型オブジェクト (TypeObject)

型オブジェクト型 はその型に属する Kompira オブジェクトのフィールドやメソッドを定義します。新しく型オブジェクトを定義することで、ユーザーは Kompira オブジェクトの型を自由に追加することができます。

注釈: 既存の型にフィールドを追加したり、不要なフィールドを削除したりなど、型オブジェクトを変更する場合、Kompira では、以下のような規則にしたがって処理されます。

- 変更後の型オブジェクトで削除されたフィールドは無視され、アクセス不可能となる。
- 変更後の型オブジェクトで新たに追加されたフィールドは自動的に null 値で初期化される。

フィールド

型オブジェクト型では、以下のフィールドを定義しています。

extend (拡張モジュール) [String] 型オブジェクトが参照する Python の拡張モジュールパスを指定します。デフォルトは、kompira.extends となります。

型オブジェクトの振る舞いやビューを拡張する場合、拡張モデルモジュール models.py と拡張ビューモジュール views.py を Python モジュールとして作成し、ここで指定したパスの下に配置します。

fieldNames (フィールド名の列) [Array] この型のオブジェクトが持つフィールド名の一覧を配列として指定します。フィールド名で使用できる文字列の規則は、ジョブフロー言語の識別子と同じです。

fieldDisplayNames (フィールド表示名の列) [Array] この型のオブジェクトが持つフィールド表示名の一覧を配列として指定します。フィールド表示名には任意の文字列を使用することができます。配列要素の順序は、フィールド名の列と対応させる必要があります。

fieldTypes (フィールド型の列) [Array] この型のオブジェクトが持つフィールド型の一覧を配列として指定します。配列要素の順序は、フィールド名の列と対応させる必要があります。

メソッド

型オブジェクト型に特有のメソッドは特に定義されていません。

5.3.2 ディレクトリ (Directory)

ディレクトリ型 は、ディレクトリオブジェクトの型を規定します。ディレクトリオブジェクトの下に複数の異なる型の Kompira オブジェクトを持つことができます。これによって、Unix のファイルシステムと同様に Kompira オブジェクトも階層構造を持つことができます。

フィールド

ディレクトリ型には固有のフィールドは定義されていません。

メソッド

ディレクトリ型には以下のメソッドが定義されています。

add [*name*, *type_obj*, [*data*, *overwrite*]] ディレクトリの下に、name で指定された名前を持つ type_obj 型の Kompira オブジェクトを追加します。data には辞書型のデータを渡すことができ、これによって、オブジェクトのフィールド値を初期化することができます。$RESULT には新規追加されたオブジェクトが格納されます。overwrite 引数に true を渡した場合、同名のオブジェクトがディレクトリの下に存在している場合でも、エラーにはならずオブジェクトを更新します。

move [*obj*, [*name*]] ディレクトリの下に obj で指定されたオブジェクトを移動します。name が指定された場合、移動対象のオブジェクトの名前が name に変更されます。

copy [*obj*, [*name*]] ディレクトリの下に obj で指定されたオブジェクトを複製します。name が指定された場合、複製されたオブブジェクトの名前は name に変更されます。obj がディレクトリやテーブルの場合、子オブジェクトは再帰的に複製されます。$RESULT には新規作成されたオブジェクトが格納されます。

has_child [*name*] ディレクトリの下に name で指定された子オブジェクトが存在すれば true を返し、存在しなければ、false を返します。

find [*args*] ディレクトリの下に args で指定した条件と一致するオブジェクトが存在すればそのオブジェクトの一覧をリストで返します。args にはフィルタリングを指定することができます。オブジェクトの属性でフィルタリングする場合は、 args に <**属性名**>=<**値**> を指定します。

例えば型オブジェクトの一覧を取得する場合は、以下のように指定します。

```
[result = /.find(type_object=/system/types/TypeObject)]
```

また、属性名にルックアップを付与することでより細かなフィルタリング条件を指定できます。ルックアップを付与する場合は args に <**属性名**>__<**ルックアップ**>=<**値**> を指定します。

例えばパスに kompira を含むオブジェクトを指定する場合は、以下のように指定します。

```
[result = /.find(abspath__contains='kompira')]
```

指定できるルックアップとフィルタリング方法は以下のとおりです。

ルックアップ	フィルタリング方式
exact, iexact	属性が指定した値に一致する。iexact では大小文字を区別しない。
contains, icontains	属性が指定した値を含む。icontains では大小文字を区別しない。
startswith, istartswith	属性が指定した値で始まる。istartswith では大小文字を区別しない。
endswith, iendswith	属性が指定した値で終わる。iendswith では大小文字を区別しない。
regex, iregex	属性が指定した正規表現にマッチする。iregex では大文字小文字区別しない。
gt, gte	属性が指定した値より大きい (gt)、または、属性が指定した値以上である (gte)。
lt, lte	属性が指定した値より小さい (lt)、または、属性が指定した値以下である (lte)。
in	属性が指定した値に含まれる。

仮想オブジェクト以外の一般のオブジェクトにおける属性値によるフィルタリングでは、属性によって指定できるルックアップが異なります。属性の一覧と指定できるルックアップは以下のとおりです。

属性	指定できるルックアップ
owner	exact, in
abspath	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
display_name	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
description	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
created	exact, gt, gte, lt, lte
updated	exact, gt, gte, lt, lte
type_object	exact, in

また、仮想オブジェクトでは、属性のデータ型によって指定できるルックアップが異なります。

属性の型	指定できるルックアップ
文字列型	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
整数型	exact, gt, gte, lt, lte
日時型	exact, gt, gte, lt, lte
オブジェクト型	exact
ユーザ型	exact
真偽型	exact

なお、ルックアップを指定していないときは exact が適用されます。属性は複数指定することができます。

また、type_object 属性フィルタリングによって型オブジェクトが指定されている場合、フィールド値によるフィルタリング条件も指定できます。フィールド値でフィルタリングする場合は、args に fields={<**属性名**>=<**値**>} 、もしくは fields={<**属性名**>__<**ルックアップ**>=<**値**>} で指定します。

型オブジェクトが指定されていない状況でフィールド値によるフィルタリングを指定するとエラーとなります。

例えばソースコードに urlopen を含むジョブフローを指定する場合は、以下のように指定します。

```
[result = /.find(type_object=/system/types/Jobflow, fields={source__contains=
↪'urlopen'})]
```

フィールド値によるフィルタリングで使用できるルックアップは以下のようになります。

フィールドの型	指定できるルックアップ
文字列型	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex, in, range
整数型	exact, isnull, gt, gte, lt, lte, in, range
真偽値型	exact
日時型	exact, isnull, gt, gte, lt, lte, range
オブジェクト型	exact, isnull
添付ファイル型	(文字列型と同じ、ファイル名がフィルタ対象になります)
配列型	(文字列型と同じ、配列の値がフィルタ対象になります)
辞書型	(文字列型と同じ、辞書の値がフィルタ対象になります)

フィールド値においても、ルックアップを指定していないときは exact が適用されます。またフィールド

値のフィルタリングも複数指定することができます。

glob [*pattern*] ディレクトリの下に pattern で指定した glob 条件と一致するオブジェクトが存在すればそのオブジェクトの一覧をリストで返します。パターンは次の形式で記述します。

```
"<オブジェクト名>"
```

オブジェクト名は glob パターンで指定します。例えば名前が kompira で始まるオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("kompira*")]
```

また、オブジェクト名に加えて次の要素を指定することが可能です。

- パス
- 型オブジェクト
- 所有者
- 属性値フィルタリング
- フィールド値フィルタリング

パスを指定した場合そのパスの下のオブジェクトを返します。パターンは次の形式で記述します。

```
"<パス>/<オブジェクト名>"
```

パスには /* と /** を指定できます。それぞれ任意の 1 段のディレクトリと任意の深さのディレクトリにマッチします。

例えば、パスに user が含まれ、名前が kompira で始まるオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("/**/user/**/kompira*")]
```

型オブジェクトを指定した場合その型オブジェクトを持つオブジェクトを返します。パターンは次の形式で記述します。

```
"<オブジェクト名>.<型オブジェクト名>"
```

例えば、ジョブフロー型のすべてのオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("*.Jobflow")]
```

所有者を指定した場合所有するオブジェクトを返します。パターンは次の形式で記述します。

```
"<オブジェクト名>@<所有者>"
```

例えば、root が所有するすべてのオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("*@root")]
```

属性値フィルタリングを指定した場合、マッチするオブジェクトを返します。パターンは次の形式で記述します。

```
"<オブジェクト名>(<属性値>=<値>)" もしくは "<オブジェクト名>(<属性値>_<ルックアップ>=<値>)"
```

属性値に指定できるルックアップの一覧は find メソッドをご参照下さい。

例えば、表示名に kompira が含まれるオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("*(diplay_name__contains='kompira')")]
```

フィールド値フィルタリングを指定した場合、マッチするオブジェクトを返します。パターンは次の形式で記述します。

```
"<オブジェクト名>[<フィールド値>=<値>]" もしくは "<オブジェクト名>[<フィールド値>_<ルックアップ>=
↪<値>]"
```

フィールド値に指定できるルックアップの一覧はは find メソッドをご参照下さい。

例えば、ソースコードに urlopen が含まれるオブジェクトを指定する場合は以下のようになります。

```
[result = /.glob("*[source__contains='urlopen']")]
```

またこれらは組み合わせて指定することができます。すべて指定した場合のパターンは次のようになります。

```
"<パス>/<オブジェクト名>.<型オブジェクト名>@<所有者>(<属性フィルタリング>)[<フィールドフィルタリング>
↪]"
```

例えば、下のようなオブジェクトを指定する場合は次のようになります。

- /user/app の下にあるオブジェクト
- root が所持
- 名前が kompira から始まる
- ジョブフロー
- 表示名にこんぴらが含まれる

- 多重度が 1 以下

```
[result =
/.glob("/user/app/**/kompira*.Jobflow
@root(display_name__contains='こんぴら')[multiplicity__lt=1]")]
```

5.3.3 ライセンス (License)

ライセンス型 は Kompira のライセンスファイルを管理するオブジェクトを定義します。

フィールド

ライセンス型には固有のフィールドは定義されていません。

メソッド

ライセンス型には固有のメソッドは定義されていません。

プロパティ

ライセンス型オブジェクトは、以下のプロパティを提供しています。

node_count　現在使用中のノード数を取得します。

5.3.4 仮想オブジェクト (Virtual)

仮想オブジェクト型 のオブジェクトは、プロセスやインシデントなど、Kompira の特殊オブジェクトを定義するための実装モジュールを指定します。

フィールド

仮想オブジェクト型では、以下のフィールドを定義しています。

virtual (実装モジュール)　[String] 特殊オブジェクトの Python の実装モジュールのパスを指定します。

メソッド

仮想オブジェクト型に特有のメソッドは特に定義されていません。

5.3.5 ジョブフロー (Jobflow)

ジョブフロー型 はジョブフローオブジェクトの型を規定します。

フィールド

ジョブフロー型では、以下のフィールドを定義しています。

source (ソース)　[LargeText] ジョブフローのソースコード文字列です。

code (コード)　[LargeText] ジョブフローのソースをコンパイルした結果の中間コード文字列が格納されます。ブラウザ上から編集することはできません。

parameters (パラメータ)　[Dictionary] ジョブフローのパラメータのデフォルト値をコンパイルした結果の中間コード文字列が、パラメータの辞書として格納されます。不可視設定されているため、ブラウザ上から編集することはできません。

executable (実行可能)　[Boolean] ジョブフローが実行可能な場合は true となります。コンパイルエラーなどで、ジョブフローが実行できない場合には false が格納されます。ブラウザ上から編集することはできません。

errors (エラー)　[Dictionary] コンパイル時のエラーメッセージが、対応するソースコードの行番号をキーとした辞書に格納されます。ブラウザ上から編集することはできません。

compilerVersion (コンパイラバージョン)　[String] ジョブフローのコンパイルに用いられたコンパイラのバージョン文字列が格納されます。ブラウザ上から編集することはできません。

multiplicity (多重度)　[Integer] ジョブフローの多重度を設定します。多重度の数を超えるジョブフロープロセスが同時にこのジョブフローを呼び出した場合、そのプロセスは他のプロセスがこのジョブフロー呼び出しを完了するまで待たされます。多重度を 0 以下の値に設定した場合、多重度は無制限と解釈されます。

defaultCheckpointMode (デフォルトチェックポイントモード)　[Boolean] ジョブフローのデフォルトのチェックポイントモードを指定します。

defaultMonitoringMode (デフォルト監視モード)　[Enum] ジョブフローのデフォルトの監視モードを指定します。

注釈: 多重度が指定されているジョブフローを呼び出すと、ジョブフロープロセスはロックを獲得します。ジョブフロー呼び出しから戻るか、もしくはジョブフローが終了すると、ロックが解放されます。ロックは再帰的に獲得可能です。したがって、多重度指定されているジョブフローを再帰的に呼び出した場合でも、そのプロセスの実行がブロックすることはありません。

多重度指定されたジョブフローの中で別のジョブフローを末尾で呼び出した場合、獲得したロックは解放されます。

メソッド

ジョブフロー型に特有のメソッドは特に定義されていません。

5.3.6 チャネル (Channel)

チャネル型 は、チャネルオブジェクトの型を規定します。チャネルオブジェクトを用いて、異なるジョブフロープロセス同士でメッセージの送受信を同期的に行うことができます。

フィールド

チャネル型では、以下のフィールドを定義しています。

message_queue (メッセージキュー)　[Array] チャネルに送信されたメッセージが格納されるキューです。ブラウザ上から編集することはできません。

event_queue (イベントキュー)　[Array] チャネルでメッセージの受信を待っているイベントが格納されるキューです。ブラウザ上から編集することはできません。

メソッド

チャネル型には以下のメソッドが定義されています。

send　[*message*] チャネルにメッセージ message を送信します。

プロパティ

チャネル型オブジェクトは、以下のプロパティを提供しています。

message_count　メッセージキューに溜まっているメッセージの数を示します。

event_count　イベントキューに溜まっているイベントの数を示します。

5.3.7 Wiki ページ (Wiki)

Wiki ページ型 は、Wiki ページオブジェクトの型を規定します。Kompira の Wiki ページオブジェクトは、Wiki Creole/Markdown/Textile 記法をサポートしています。

フィールド

Wiki ページ型では、以下のフィールドを定義しています。

wikitext (Wiki テキスト)　[LargeText] wiki ページのテキストを格納します。

style (スタイル)　[Enum] wiki ページの記法を Creole, Markdown, Textile の中から選択します。

メソッド

Wiki ページ型に特有のメソッドは特に定義されていません。

5.3.8 スクリプトジョブ (ScriptJob)

スクリプトジョブ型 は、スクリプトジョブオブジェクトの型を規定します。

フィールド

スクリプトジョブ型では、以下のフィールドを定義しています。

source (ソース)　[LargeText] スクリプトのソーステキストを格納します。

ext (拡張子)　[String] スクリプトの拡張子を設定します。Windows サーバ上でスクリプトを実行する場合、スクリプトの拡張子を適切に設定する必要があります。

multiplicity (多重度)　[Integer] スクリプトジョブの多重度を設定します。多重度の数を超えるジョブフロープロセスが同時にこのスクリプトジョブを呼び出した場合、そのプロセスは他のプロセスがこのスクリプトジョブ呼び出しを完了するまで待たされます。多重度を 0 以下の値に設定した場合、多重度は無制限と解釈されます。

メソッド

スクリプトジョブ型に特有のメソッドは特に定義されていません。

5.3.9 環境変数 (Environment)

環境変数型 は、環境変数オブジェクトの型を規定します。ユーザー設定の環境変数の項目に、環境変数オブジェクトを設定すると、そのユーザーがジョブフローを実行した場合、環境変数辞書で定義されているキーを変数名として、対応する値をジョブフローから参照することができるようになります。

フィールド

環境変数型では、以下のフィールドを定義しています。

environment (環境変数)　[Dictionary] 環境変数辞書を格納します。

メソッド

環境変数型に特有のメソッドは特に定義されていません。

5.3.10 テンプレート (Template)

テンプレート型 は、テンプレートオブジェクトの型を規定します。

フィールド

テンプレート型では、以下のフィールドを定義しています。

template (テンプレート)　[LargeText] テンプレート文字列を格納します。

メソッド

テンプレート型に特有のメソッドは特に定義されていません。

バージョン 1.4.7 で非推奨: 替わりにテキスト型オブジェクトを使用してください。

5.3.11 テーブル (Table)

テーブル型 は、テーブルオブジェクトの型を規定します。テーブルオブジェクトは、ディレクトリオブジェクトのように、複数の子オブジェクトを持つことができます。ただし、子オブジェクトの型は固定されます。

フィールド

テーブル型では、以下のフィールドを定義しています。

typeObject (オブジェクト型)　[Object] このテーブルに格納する子オブジェクトの型を指定します。

relatedObject (関連オブジェクト)　[Object (Jobflow or Form)] このテーブルのメニューから実行できるジョブフローやフォームを指定します。テーブル一覧から選択したオブジェクトに対して、ジョブフローやフォームを実行できるようになります。ジョブフロー実行の場合は、ジョブフローの最初のパラメータに選択されたオブジェクトリストが渡されます。フォーム実行の場合、objects パラメータに選択されたオブジェクトリストが渡されます。

displayList (表示フィールド一覧)　[Array] テーブルのビューで表示させる子オブジェクトのフィールド名を配列で指定します。

メソッド

テーブル型のメソッドはディレクトリ型が提供するメソッドと同じですが、add メソッドの type_obj パラメータは省略可能となります。

add [*name*, [*type_obj*, *data*, *overwrite*]] テーブルの下に、name で指定された名前を持つ テーブルのオブジェクト型フィールドで指定された型の Kompira オブジェクトを追加します。data には辞書型のデータを渡すことができ、これによって、オブジェクトのフィールド値を初期化することができます。overwrite 引数に true を渡した場合、同名のオブジェクトがテーブルの下に存在している場合でも、エラーにはならずオブジェクトを更新します。

5.3.12 管理領域 (Realm)

管理領域型 は、管理領域オブジェクトの型を規定します。管理領域オブジェクトを定義することにより、ジョブマネージャごとに、管理対象のネットワークを分割して管理できるようになります。

フィールド

管理領域型では、以下のフィールドを定義しています。

range (対象範囲) [Array] この管理領域が管轄するネットワークアドレスの対象範囲を指定します。

disabled (無効) [Boolean] この値を true に設定すると、管理領域の設定が無効化されます。

メソッド

管理領域型に特有のメソッドは特に定義されていません。

5.3.13 添付ファイル (AttachedFile)

添付ファイル型 は、添付ファイルオブジェクトの型を規定します。

フィールド

添付ファイル型では、以下のフィールドを定義しています。

attached1 (添付ファイル 1) [File] 1 番目の添付ファイルオブジェクトが格納されるフィールドです。

attached2 (添付ファイル 2) [File] 2 番目の添付ファイルオブジェクトが格納されるフィールドです。

attached3 (添付ファイル 3) [File] 3 番目の添付ファイルオブジェクトが格納されるフィールドです。

メソッド

添付ファイル型に特有のメソッドは特に定義されていません。

5.3.14 ノード情報 (NodeInfo)

ノード情報型 は、ノード情報オブジェクトの型を規定します。ノード情報オブジェクトをジョブフロー中の __node__ 制御変数に指定することで、コマンドを実行する対象ノードを指定することができます。

フィールド

ノード情報型では、以下のフィールドを定義しています。

hostname (ホスト名)　[String] ノードのホスト名を指定します。

ipaddr (IP アドレス)　[IPAddress] ノードの IP アドレスを指定します。

port (ポート番号)　[Integer] ノードのポート番号を指定します。指定しない場合、接続種別に応じたデフォルトのポート番号が使用されます。

conntype (接続種別)　[Enum] ノードの接続種別を 'ssh' か 'winrs' から選択します。

shell (シェル)　[String] リモート接続時に使用するシェルを指定します。指定しない場合、デフォルトとして '/bin/bash' が使用されます。

use_shell (シェル使用)　[Boolean] リモート接続時にシェルを使用しない場合、false に設定します。ネットワーク機器など、シェルを備えていない機器への接続時には、false を設定すると良いでしょう。 デフォルトは true です。

proxy (プロキシ)　[Object] 踏み台サーバ経由で SSH 接続する場合、踏み台サーバとなるノード情報オブジェクトを指定します。SSH 接続時のみ使用されます。

account (アカウント)　[Object] リモート接続時に使用するアカウント情報を指定します。ジョブフローの __account__ 制御変数を明示的に指定した場合は、そちらが優先されます。

メソッド

ノード情報型に特有のメソッドは特に定義されていません。

5.3.15 アカウント情報 (AccountInfo)

アカウント情報型 は、アカウント情報オブジェクトの型を規定します。アカウント情報オブジェクトをジョブフロー中の__account__ 制御変数にセットすることで、リモート接続時に使用するアカウント情報を指定することができます。

フィールド

アカウント情報型では、以下のフィールドを定義しています。

user (ユーザ名)　[String] アカウントのユーザ名を設定します。

password (パスワード)　[Password] アカウントのパスワードを設定します。パスフレーズ付きの SSH 鍵ファイルが設定されている場合、パスフレーズとしても使用されます。

keyfile (SSH 鍵ファイル)　[File] SSH 鍵ファイルを使ってログインする場合、鍵ファイルを添付します。

passphrase (SSH 鍵パスフレーズ)　[Password] パスフレーズ付きの SSH 鍵の場合に指定するパスフレーズです。パスフレーズ無しの場合やパスワードと同じ場合には指定を省略することができます。

メソッド

アカウント情報型に特有のメソッドは特に定義されていません。

5.3.16 リポジトリ (Repository)

リポジトリ型 は、リポジトリオブジェクトの型を規定します。リポジトリオブジェクトを用いて、外部の VCS リポジトリと連携することができ、Kompira のディレクトリのデータをリポジトリにプッシュしたり、逆にリポジトリ上のデータを Kompira ディレクトリにプルしたりといった、データの同期が可能となります。

フィールド

リポジトリ型では、以下のフィールドを定義しています。

URL (URL)　[URL] 同期対象リポジトリの URL を設定します。

repositoryType (リポジトリ種別)　[Enum] リポジトリ種別を指定します。現バージョンでは、'mercurial' のみサポートしています。

port (ポート番号)　[Integer] 外部リポジトリに接続するポート番号を指定します。未指定の場合、デフォルトのポート番号が使用されます。

username (ユーザー名)　[String] リポジトリ接続時のユーザー名を指定します。

password (パスワード)　[Password] リポジトリ接続時のパスワードを指定します。

directory (ディレクトリ)　[Object] 同期対象の Kompira ディレクトリを指定します。

log (ログ)　[LargeText] 同期実行時のログが格納されます。

メソッド

リポジトリ型に特有のメソッドは特に定義されていません。

5.3.17 メールチャネル (MailChannel)

メールチャネル型 は、IMAP4/POP3 サーバからのメールをチャネルに取り込むメールチャネルオブジェクトの型を規定します。

ジョブフローがメールチャネルからメール受信待ち状態になると、メールチャネルは設定されている IMAP4/POP3 サーバからメールを取り込みます。メールを受信した場合、ジョブフローにそのメールが渡され、ジョブフローは処理を続行します。受信したメールは、IMAP4/POP3 サーバのフォルダから削除されます。メールフォルダが空で受信できなかった場合は、checkInterval で設定さらた時間が経過した後に再度、IMAP4/POP3 サーバからメールを取り込みます。メールチャネルを待つジョブフローが存在しない場合、メール受信のポーリング処理は休止します。

フィールド

メールチャネル型では、以下のフィールドを定義しています。

message_queue (メッセージキュー)　[Array] メールチャネルに送信されたメッセージが格納されるキューです。ブラウザ上から編集することはできません。

event_queue (イベントキュー)　[Array] メールチャネルでメッセージの受信を待っているイベントが格納されるキューです。ブラウザ上から編集することはできません。

serverName (サーバ名)　[String] 接続する IMAP4/POP3 サーバのホスト名か IP アドレスを設定します。

protocol (プロトコル)　[Enum] メール受信のプロトコルとして、IMAP4 もしくは POP3 のいずれかを設定します。

SSL (SSL)　[Boolean] SSL による通信を行う場合 true にセットします。

port (ポート番号)　[Integer] IMAP サーバのポート番号を設定します。指定しない場合デフォルトのポート番号が使用されます。

mailbox (メールボックス)　[String] 受信するメールボックスを設定します。デフォルトは "INBOX" です。POP3 プロトコルの場合、メールボックスの設定は無視されます。

> **警告:** 日本語のメールボックス名を設定することはできません。

username (ユーザ名)　[String] IMAP4/POP3 サーバに接続するユーザー名を設定します。

password (パスワード)　[Password] IMAP4/POP3 サーバに接続する際のパスワードを設定します。

checkInterval (受信チェック間隔)　[Integer] IMAP4/POP3 サーバに対して新着メッセージを確認する間隔を分単位で指定します。デフォルトは 10 分です。0 を指定するとデフォルトの値となります。負の値を設定すると、新着メッセージの確認は行いません。

timeout (受信タイムアウト)　[Integer] IMAP4/POP3 サーバに対して受信タイムアウトを秒単位で指定します。空、もしくは、0 や負の値を設定すると、受信時にタイムアウトしません。

disabled (無効)　[Boolean] IMAP4/POP3 サーバへの接続を無効化します。

log (ログ)　[LargeText] IMAP4/POP3 サーバの接続ログが格納されます。

logSize (ログサイズ)　[Integer] ログの最大サイズを指定します。最大サイズを超えた場合、古いログメッセージから削除されます。

メソッド

メールチャネル型に特有のメソッドは特に定義されていません。

5.3.18 フォーム (Form)

フォーム型 は、ユーザ入力フォームのビューを提供するフォームオブジェクトの型を規定します。入力フォームの項目は、ユーザーが自由に定義することができます。

ユーザがフォームを提出するとフォームに入力された情報は、辞書型のデータとして、指定した提出オブジェクトに提出されます。提出オブジェクトがチャネル型の場合、データはそのチャネルオブジェクトのメッセージキューに置かれます。提出オブジェクトがジョブフローの場合、辞書データはそのジョブフローのパラメータに展開され、実行を開始します。

フィールド

submitObject (提出オブジェクト)　[Object (Channel or Jobflow)] フォームに入力されたデータの提出先オブジェクトを指定します。

fieldNames (フィールド名の列)　[Array] 入力フォームが持つフィールド名の一覧を配列として指定します。フィールド名で使用できる文字列の規則は、ジョブフロー言語の識別子と同じです。

fieldDisplayNames (フィールド表示名の列)　[Array] 入力フォームが持つフィールド表示名の一覧を配列として指定します。フィールド表示名には任意の文字列を使用することができます。配列要素の順序は、フィールド名の列と対応させる必要があります。

fieldTypes (フィールド型の列)　[Array] 入力フォームが持つフィールド型の一覧を配列として指定します。配列要素の順序は、フィールド名の列と対応させる必要があります。

メソッド

フォーム型に特有のメソッドは特に定義されていません。

5.3.19 設定 (Config)

設定型 は、設定フォームのビューを提供する設定オブジェクトの型を規定します。設定フォームの項目は、ユーザーが自由に定義することができます。

ユーザが設定フォームを保存するとフォームに入力された情報は、辞書型のデータとして、設定オブジェクトの data プロパティに保存されます。

フィールド

fieldNames (フィールド名の列)　[Array] 設定フォームが持つフィールド名の一覧を配列として指定します。フィールド名で使用できる文字列の規則は、ジョブフロー言語の識別子と同じです。

fieldDisplayNames (フィールド表示名の列)　[Array] 設定フォームが持つフィールド表示名の一覧を配列として指定します。フィールド表示名には任意の文字列を使用することができます。配列要素の順序は、フィールド名の列と対応させる必要があります。

fieldTypes (フィールド型の列)　[Array] 設定フォームが持つフィールド型の一覧を配列として指定します。配列要素の順序は、フィールド名の列と対応させる必要があります。

プロパティ

設定型オブジェクトは、以下のプロパティを提供しています。

data　設定フォームに入力されたデータ辞書の値です。

メソッド

設定型に特有のメソッドは特に定義されていません。

5.3.20 ライブラリ (Library)

ライブラリ型 は、ジョブフローから呼び出し可能な Python で実装されたライブラリを定義します。

フィールド

libraryType (ライブラリ種別)　[Enum] ライブラリの定義方法を指定します。'source' を選択すると、ソーステキストに格納された文字列が Python のモジュールプログラムとしてロードされます。'safe_source' を

選択すると、ソーステキストに格納された文字列が安全な Python のモジュールプログラムとしてロードされます。'module' を選択すると、モジュールパスに指定された文字列が、Kompira パッケージ内の kompira.library 以下のモジュールとしてロードされます。デフォルト値は 'source' です。このフィールドはブラウザ上から編集することはできません。

modulePath (モジュールパス) [String] ロードする Python ライブラリのモジュールパスを指定します。libraryType が 'module' の時に使用されるフィールドです。このフィールドはブラウザ上から編集することはできません。

sourceText (ソーステキスト) [LargeText] Python のソースコードを記述します。

document (ドキュメント) [LargeText] Python モジュールのドキュメント文字列が格納されます。ロードエラー時にはエラーメッセージが格納されます。このフィールドはブラウザ上から編集することはできません。

executable (実行可能) [Boolean] Python モジュールが正しくロードされ、ジョブフローから呼び出し可能な場合は true となります。ロードに失敗した場合は、false となります。このフィールドはブラウザ上から編集することはできません。

メソッド

設定型に特有のメソッドは特に定義されていません。

呼び出し例

ライブラリオブジェクトでは、定義した Python の関数をジョブフローから呼び出すことができます。たとえば、以下のような Python のプログラムをソーステキストとして test_lib オブジェクトを定義します。

Python プログラム

```
def split(s):
    return s.split():

def hello():
    print('Hello, world!')
```

このライブラリで定義された関数を呼び出すジョブフローは以下のようになります。:

```
[str = 'foo bar baz']
-> [result = ./test_lib.split(s)]
-> [./test_lib.hello]
```

上記のジョブフローを実行すると、result 変数には split の結果のリスト ['foo', 'bar', 'baz'] が格納されます。また、ジョブフロープロセスのコンソールには、"Hello, world!" が出力されます。

警告: ライブラリオブジェクトで定義する Python の関数名に、display_name や update や delete といった、Kompira オブジェクトにあらかじめ組み込まれているプロパティ名 (プロパティ) やメソッド名 (メソッド) を使った場合、名前が衝突するため、その関数は呼び出すことができません。

5.3.21 メールテンプレート (MailTemplate)

メールテンプレート型 は、メールテンプレートオブジェクトの型を規定します。

フィールド

メールテンプレート型では、以下のフィールドを定義しています。

subject (件名)　[String] メールの件名となるテンプレート文字列を格納します。

body (本文)　[LargeText] メールの本文となるテンプレート文字列を格納します。

メソッド

メールテンプレート型に特有のメソッドは特に定義されていません。

5.3.22 テキスト (Text)

テキスト型 は、プレインテキストや HTML テキストを保持するテキストオブジェクトの型を規定します。

テキストオブジェクトは、http://<Kompira サーバ>/<テキストオブジェクト>.render にブラウザからアクセスすることで、テンプレートエンジンによってレンダリングされたレンダービューを表示させることができます。

注釈: テンプレートエンジンには、 Jinja2 を使用しています。テンプレートの記法については、Jinja2 のドキュメント http://jinja.pocoo.org/docs/dev/templates/ を参照してください。

include や extends タグで、別のテキストオブジェクトのパス指定することで、テンプレートの取り込みや継承も可能です。

フィールド

テキスト型では、以下のフィールドを定義しています。

text (テキスト)　[LargeText] テキスト文字列を格納します。

ext (拡張子)　[String] レンダービューを表示するブラウザアクセス用の拡張子を指定します。たとえば、拡張子として "html" を指定すると、http://<Kompira サーバ>/< テキストオブジェクトパス>.html にアクセスするとレンダービューが表示され、http://<Kompira サーバ>/<テキストオブジェクトパス> にアクセスすると通常のビューが表示されます。

contentType (コンテントタイプ)　[String] テキストのコンテントタイプを指定します。コンテントタイプの指定を省略した場合は、拡張子からコンテントタイプが推測されます。また、拡張子の指定を省略し、コンテントタイプのみ指定した場合は、拡張子なしでブラウザアクセスした場合でも、通常ビューではなくレンダービューが表示されます。

context (コンテキスト): Object(Environment)　テンプレートに渡すコンテキストとして環境変数オブジェクトを指定します。テンプレート中から環境変数のキー値を変数として参照することができます。

メソッド

テキスト型には以下のメソッドが定義されています。

render :　テンプレートエンジンによってレンダリングされたテキストを取得します。

プロパティ

テキスト型オブジェクトは、以下のプロパティを提供しています。

content_type　推測されたコンテントタイプを示します。

バージョン 1.4.7 で追加: 新規にテキスト型が追加されました。

5.3.23 システム情報型

システム情報型 は Kompira のシステム情報をを提供するオブジェクトを定義します。

フィールド

システム情報型には固有のフィールドは定義されていません。

メソッド

システム情報型には固有のメソッドは定義されていません。

プロパティ

システム情報型オブジェクトは、以下のプロパティを提供しています。

engine_started　Kompira エンジンの起動日時を示します。

server_datetime　Kompira サーバの現在日時を示します。

version　Kompira のバージョン番号を示します。

バージョン 1.4.8.post2 で追加: 新規にシステム情報型が追加されました。

5.4 特殊オブジェクト

特殊オブジェクトは、通常のオブジェクトとは異なり、Kompira の型オブジェクトによって規定されていない組み込みのオブジェクトです。その種類毎に固有のプロパティとメソッドを備えています。フィールドは持ちません。

5.4.1 プロセス (Process)

ジョブフローの実行時のプロセス情報を表すオブジェクトです。

プロパティ

プロセスオブジェクトで定義されているフィールドは以下のとおりです。

checkpoint_mode　[Boolean] プロセスがチェックポイントモードで実行している場合は true、そうでない場合は false が設定されます。書き込み可能なプロパティです。

children　[Array of Process] 子プロセスの一覧が格納されます。

console　[String] コンソールに表示されている文字列です。

current_job　[Object] プロセスの現在実行しているジョブフローオブジェクト、もしくはスクリプトジョブオブジェクトが格納されます。ジョブフローの中から別のジョブフローを呼び出すと、current_job の値が変更されます。

elapsed_time: Timedelta　プロセスの実行経過時間を表します。

finished_time　[Datetime] プロセスの実行終了した日時です。

job　[Object] プロセスを開始したジョブフロー、もしくは、スクリプトジョブオブジェクトを表します。

monitoring_mode　[String] プロセスの監視モードを表す文字列です。書き込み可能なプロパティです。

文字列	監視モード
NOTHING	メール通知しない
MAIL	プロセス終了時にメール通知する
ABORT_MAIL	プロセス異常終了時にメール通知する

pid [Integer] プロセス ID です。

parent [Object] 親プロセスのジョブフローオブジェクトです。

schedule [Schedule] プロセスがスケジューラから起動された場合、該当するスケジュールオブジェクトが格納されます。

started_time [Datetime] プロセスを実行開始した日時です。

status [String] プロセスの実行状態を表す文字列です。

文字列	実行状態
NEW	新規（開始待ち）
READY	実行可能
RUNNING	実行中
WAITING	入力/コマンド完了待ち
ABORTED	異常終了
DONE	実行完了

step_mode [Boolean] プロセスがステップ実行モードで実行している場合は true、そうでない場合は false が設定されます。書き込み可能なプロパティです。

suspended [Boolean] プロセスが一時停止状態の場合は true、そうでない場合は false となります。

user [User] プロセスの実行ユーザーです。特権ユーザーに限り、実行ユーザーを変更することができます。

バージョン 1.5.0.post1 で追加: monitoring_mode が追加されました。

メソッド

delete プロセスオブジェクトを削除します。

5.4.2 プロセス一覧 (/process)

プロセス一覧 (/process) は、プロセスオブジェクトのリストを保持するオブジェクトであり、仮想オブジェクト (Virtual) として実装されています。

以下のように for や pfor ブロックにおいて、各プロセスオブジェクトに対して繰り返し処理を行うことが可能です。

```
{ for p in /process |
    print(p)
}
```

5.4.3 スケジュール (Schedule)

スケジュールオブジェクトは、Kompira のスケジューラに登録されたスケジュールを表します。

プロパティ

day: String　スケジュールを実行する日 (1 ～ 31) を表します。

day_of_week: String　スケジュールを実行する曜日名、もしくは、曜日番号を表します。0(月曜日)-6(日曜日)、または mon,tue,wed,thu,fri,sat,sun を指定します。

description: String　スケジュールの内容を説明する文字列が格納されます。

disabled: Boolean　スケジュールの無効化を示すフィールドです。スケジュールが無効な場合は true、有効な場合は false となります。

hour: String　スケジュールを実行する時 (0 ～ 23) を表します。

job: Object　スケジュールによって実行されるジョブフロー、もしくは、スクリプトジョブが格納されます。

minute: String　スケジュールを実行する分 (0 ～ 59) を表します。

month: String　スケジュールを実行する月 (1 ～ 12) を表します。

name: String　スケジュールの名称を表す文字列です。

next_run_time: Datetime　スケジュールが有効な場合、次に実行される日時が格納されます。(読み取り専用)

parameters: Array of String　ジョブフローやスクリプトに渡すパラメータの文字列が可能されます。

user: User　スケジュールのユーザーです。

week: String　スケジュールを実行する ISO 週番号 (1 ～ 53) を表します。

year: String　スケジュールを実行する年 (4 桁の数字) を表します。

注釈: 上記、プロパティのうち、実行日時を指定するフィールドについては、日時設定フィールドの書式が使用できます。

メソッド

delete　スケジュールオブジェクトを削除します。

5.4.4 スケジュール一覧 (/scheduler)

スケジュール一覧 (/scheduler) は、スケジュールオブジェクトのリストを保持するオブジェクトであり、仮想オブジェクト (Virtual) として実装されています。スケジュール一覧は、for や pfor ブロックで用いることで、各スケジュールオブジェクトを繰り返し処理することが可能です。

メソッド

スケジュール一覧 には以下のメソッドが定義されています。

add [*name*, *job*, [*parameters*, *datetime*]] スケジュール一覧に、name で指定された名前を持ち、job で指定されたジョブフローまたはスクリプトジョブを実行するスケジュールを追加します。parameters 引数にジョブフローやスクリプトジョブの実行時に与えるパラメータリストをオプションとして指定することができます。オプショナルな datetime 引数にジョブの実行日時を表す日時型の値を指定することができます。

5.4.5 ユーザー (User)

Kompira のユーザーを表すオブジェクトです。

プロパティ

ユーザーオブジェクトで定義されているプロパティは以下のとおりです。

username [String] ユーザー名です。

first_name [String] ユーザーの姓名の名の部分を表します。

last_name [String] ユーザーの姓名の姓の部分を表します。

full_name [String] ユーザーの姓名を表します。

mailbox [String] 以下の形式のアドレスを表します。

```
ユーザー名 <メールアドレス>
```

email [String] ユーザーのメールアドレスです。書き込み可能なプロパティです。

environment [Object (Environment)] 環境変数オブジェクトです。書き込み可能なプロパティです。

home_directory [Object (Directory)] ユーザーのホームです。書き込み可能なプロパティです。

groups [Array of Group] ユーザーが所属するグループ一覧です。

enable_restapi [Boolean] REST API を有効化するかどうかを示します。書き込み可能なプロパティです。

auth_token [String] ユーザーの認証トークンです。読み込み専用のプロパティです。REST API が無効の時は null となります。

メソッド

公開されているメソッドはありません。

5.4.6 グループ (Group)

Kompira のグループを表すオブジェクトです。

プロパティ

公開されているプロパティはありません。

メソッド

公開されているメソッドはありません。

第 6 章

他システムとの連携

著者　Kompira 開発チーム

6.1 はじめに

Kompira で他システムにデータを受け渡す、また他システムからデータを受け取るための方法、必要な設定等について説明します。

6.2 Kompira へのイベント送信

ジョブマネージャパッケージ および イベント送信パッケージ に含まれる kompira_sendevt コマンドを用いることで、Kompira に対してイベント情報を送信することができます。ここでは、kompira_sendevt を用いた Kompira へのイベント送信について説明します。

kompira_sendevt スクリプトは、引数で指定された <keyword>=<value> の組をメッセージに詰めて、Kompira サーバに送信します。

```
/opt/kompira/bin/kompira_sendevt [options] [<key1>=<value1> ...]
```

key1 と value1 を結ぶ '=' の両側にスペースを入れないように注意してください。ジョブフローは、受信したメッセージを辞書型のデータとして参照することができます。

引数が指定されなかった場合は、標準入力を 1 つの key として Kompira サーバに送信します。

6.2.1 Windows からのイベント送信

Windows へのインストール に従って Windows にイベント送信パッケージをインストールすることで、Linux の場合と同様に kompira_sendevt コマンドを用いて Windows からイベント送信を行うことができます。

注釈: 手順 *Windows へのインストール* に沿ってインストールした場合、kompira_sendevt コマンドは `C:\Kompira\Scripts\kompira_sendevt.exe` にインストールされます。

また、デフォルトのログディレクトリ `/var/log/kompira` が Windows 上には作成されないため、kompira_sendevt コマンドを実行する時に警告が表示され、標準出力上にログが出力されます。

これを回避するためには、例えば、以下のような設定ファイルを作成して、kompira.conf という名称で保存しておきます。

```
[logging]
logdir=.\
```

kompira_sendevt コマンドの `--conf` オプションで、上記のファイルを読み込むようにします。さらに、イベント送信先の Kompira サーバも `--server` オプションで指定するようにします。

```
$ kompira_sendevt --conf=kompira.conf --server=<kompira server> test_key=test_message
```

6.2.2 kompira_sendevt のオプション

kompira_sendevt コマンドには以下のオプションがあります。

オプション	説明
-c, --config=CONF	設定ファイルを指定します。(CONF は設定ファイルのパス) デフォルトでは/opt/kompira/kompira.conf が読み込まれます。
-s, --server=SERVER	送信先の Kompira サーバの IP アドレス、もしくはサーバ名を指定します。設定ファイルの指定より優先されます。
-p, --port=PORT	送信先 Kompira サーバのメッセージキューのポート番号を指定します。設定ファイルの指定より優先されます。
--user=USER	送信先 Kompira メッセージキューのユーザー名を指定します。設定ファイルの指定より優先されます。
--password=PASSWORD	送信先 Kompira メッセージキューのパスワードを指定します。設定ファイルの指定より優先されます。
--ssl	メッセージキューの接続に SSL を用います。
--channel=CHANNEL	メッセージを送信するチャネルの Kompira ファイルシステム上のパスを指定します。設定ファイルの指定より優先されます。
--site-id=SITE_ID	Kompira サイト ID を指定します。設定ファイルの指定より優先されます。
--max-retry=MAX_RETRY	イベントの送信を最大何回試みるかを指定します。
--retry-interval= RETRY_INTERVAL	イベントの送信間隔を指定します (単位は秒)。
--dry-run	実際にはデータを送信せずに、送信内容を標準出力に表示します。

6.3 Kompira でのメール受信

Kompira_sendevt を用いて、Kompira サーバが受信したメール内容をジョブフローで扱えるようにする方法を説明します。

注釈: IMAP サーバを使用している場合、以下に紹介する方法の他に、メールチャネルを使用することでメール内容をジョブフローで扱うことができます。詳しくは *メールチャネル (MailChannel)* を参照してください。

6.3.1 Linux の設定

Sendmail 用のエイリアスである /etc/aliases ファイルに設定を記述することで、Kompira サーバの特定のアカウント宛のメールに対して、任意のコマンドの実行を指定することができます。

以下は kompira サーバの kompira アカウント宛にメールが送られた際、メールを kompira_sendevt に送る場合の設定です。

```
kompira:    "|LANG=ja_JP.UTF-8 /opt/kompira/bin/kompira_sendevt --channel=/system/
↪channels/Mail"
```

/etc/aliases に上記を記述した後、下記のコマンドを実行することで設定が反映されます。

```
% newaliases
```

注釈: お使いのシステムによっては smrsh を使用する必要がある場合があります。その場合、kompira_sendevt コマンドのシンボリックリンクを smrsh のディレクトリに作成してください。

6.3.2 Kompira の設定

kompira_sendevt は任意のチャネルに値を送ることができます。ここでは/system/channels/Mail というメールを受信する専用のチャネルを作成しておきましょう。

以下はメール内容を受信し、内容を表示するジョブフロー例です。

```
</system/channels/Mail>
-> [mail = $RESULT]
-> mail_parse(mail)
-> [parsed_mail = $RESULT]
-> print(parsed_mail['Subject'])
-> print(parsed_mail['Body'])
```

Kompira の組み込みジョブである mail_parse を使うことで、MIME 形式のメールテキストをパースし、辞書形式で値を扱うことができるようになります。

6.4 監視システムとの連携

Kompira は、Zabbix や Nagios など外部の監視サーバと連携することが可能です。連携したい外部システムから、Kompira のメッセージキュー（RabbitMQ）に対してイベント情報を送信することで、ジョブフローからそのイベントを受信できるようになります。

ここでは、Zabbix を例にして、障害の発生を Kompira に通知する方法について説明します。

6.4.1 イベント送受信の確認

Zabbix が動作しているサーバに、Kompira に対してイベント情報を送信するためのスクリプトを準備します。ここでは、kompira_sendevt を用いた方法について説明します。

1. **Zabbix サーバに Kompira エージェントをインストールする。**

Kompira マニュアル (イベント送信パッケージ) にしたがって Kompira のイベント送信パッケージを Zabbix が動作しているサーバにインストールします。(ジョブマネージャを動作させない場合、ジョブマネージャの起動設定は不要です。)

2. 設定ファイルの変更

Zabbix サーバ側の/opt/kompira/kompira.conf ファイルの書き換えます。

具体的には、[amqp-connection] セクションの server 項目に、Kompira サーバの IP アドレス、もしくはホスト名を設定します。また、[event] セクションの channel 項目が/system/channels/Alert に設定されていることも確認してください。

3. メッセージの通知確認

この時点で、kompira_sendevt を実行して、Kompira サーバに対して正しくイベントが通知できることを確認します。Zabbix サーバ側で、以下のコマンドを実行してください。:

```
$ /opt/kompira/bin/kompira_sendevt test=hello
```

次に、Kompira にログインし、/system/channels/Alert のページを参照し、メッセージ数が増えていることを確認してください。

4. メッセージの受信方法

次に、/system/channels/Alert に到着したメッセージはジョブフローから読み出してみます。以下のようなジョブフローを定義して、実行してみてください。

```
</system/channels/Alert> -> [message = $RESULT] -> print(message.test)
```

コンソールに、hello と表示されたら成功です。

6.4.2 Zabbix の設定

次に、Zabbix の設定を行います。

Zabbix にログインし、「アクションの設定」メニューからアクションを新規に作成し、その中にアクションのオペレーションを新規に作成してください。オペレーションのタイプはリモートコマンドにします。

リモートコマンドの内容はたとえば以下のようにします。

```
Zabbix server:python /opt/kompira/bin/kompira_sendevt status="{TRIGGER.STATUS}"
  severity="{TRIGGER.NSEVERITY}" hostname="{HOSTNAME}"
  trigger_name="{TRIGGER.NAME}" trigger_key="{TRIGGER.KEY}"
  detail="{TRIGGER.KEY}: {ITEM.LASTVALUE}"
```

ここでは、以下のようなキーを含む辞書データを Kompira に送るように設定しています。

キー名	内容（値）
status	トリガーの状態
severity	深刻度
hostname	障害の発生したホスト名
trigger_name	トリガー名
trigger_key	トリガーキー
detail	イベント詳細情報 (トリガーキーとイベント値の組み)

あとは、ここで登録したアクションが、障害イベントをトリガーとしてキックされるように設定を行います。詳しくは、Zabbix のマニュアル等を参照してください。

6.5 Redmine との連携

外部のチケッティングシステムとの連携例として、Kompira のジョブフローから Redmine に対してチケットを発行する方法について説明します。

6.5.1 Redmine の設定

1. **REST API の有効化**

「管理」->「設定」->「認証」から、「REST による Web サービスを有効にする」チェックをつけて保存します。

2. **プロジェクトの作成**

「管理」->「プロジェクト」から「新しいプロジェクト」を選択し、プロジェクト「test」を作成します。

3. **優先度の設定**

「管理」->「列挙項目」でチケットの優先度に値を設定します。(例：「高」「中」「低」)

また、どれか 1 つを「デフォルト値」と設定します。

(※) デフォルト値を設定しない場合は、API 呼び出しの際に priority_id の値が必要となります。

4. ユーザの作成

「管理」->「ユーザ」から「新しいユーザー」を選択し、任意のユーザを作成します。

作成したユーザでログインし、「個人設定」ページにある API アクセスキーを控えておきます。

6.5.2 チケットを発行する

Redmine のチケットを発行するには、必要な情報を json 形式のデータに変換し、POST リクエストを Redmine の URL に送信します。

そのためには、Kompira の組み込みジョブである urlopen に辞書型のデータを渡して呼び出します。

具体的には以下のようなジョブフローを記述することで、Redmine に対してチケットを発行することができます。

```
|redmine_server = '192.168.0.1'|
|redmine_key = '1234567890abcdef1234567890abcdef12345678'|
|ticket_title = 'Task from Kompira'|
|project_name = 'test'|

[url = 'http://$redmine_server/issues.json?format=json&key=$redmine_key']
-> [ticket = {issue = {subject = ticket_title, project_id = project_name}}]
-> urlopen(url=url, data=ticket, timeout=60, encode='json')
```

「redmine_key」には「4. ユーザの作成」で確認した API アクセスキーを設定します。

上記に加えて、チケットの優先度、説明、担当者、カテゴリなどの情報を含めることもできます。

また、チケットの更新・削除、チケット情報の一覧の取得なども行うことができます。詳しくは Redmine API 仕様を参照してください。

6.6 SNMP トラップの受信

Linux コマンドの snmptrapd(8) と snmptrap(1) を用いて、Kompira のジョブフローで SNMP トラップを受信する方法を説明します。

6.6.1 環境

	IP Address	OS
Kompira サーバ	192.168.213.100	CentOS 6.5
SNMP エージェントサーバ	192.168.213.101	CentOS 6.5

6.6.2 Kompira サーバの設定

Kompira サーバには Kompira がインストール済みであるとします。

1. **snmptrapd** をインストール

```
$ yum install net-snmp
```

2. **/etc/snmp/snmptrapd.conf** を編集

SNMP トラップをハンドルするため、snmptrapd.conf を編集します。

```
authCommunity     log,execute,net default
traphandle default    /opt/kompira/bin/kompira_sendevt --channel=/system/channels/
↪snmptrap
```

ここで default は「全ての OID」を表します。

3. **Kompira** にジョブフローを追加

「/system/channels/snmptrap」チャネルを作成し、このチャネルへのデータを待ち受けるジョブフローを作成、実行します。

```
</system/channels/snmptrap> ->
print($RESULT)
```

4. **snmptrapd** を起動

```
$ service snmptrapd start
```

6.6.3 SNMP エージェントサーバの設定

snmptrap コマンドをインストール

```
$ yum install net-snmp-utils
```

6.6.4 SNMP トラップの送信

SNMP エージェントサーバ上で snmptrap コマンドを実行します。

```
$ snmptrap -v 2c -c default 192.168.213.100 '' netSnmp.99999 netSnmp.99999.1 s "hello␣
↪world"
```

Kompira サーバ側で正しく受信できた場合、/var/log/messages に以下のようなログが表示されます。

```
$ tail -f /var/log/messages
Dec 13 16:29:30 kompira-server snmptrapd[6110]: 2012-12-13 16:29:30 <UNKNOWN>
[UDP: [192.168.213.101]:56313->[192.168.213.100]]:#012DISMAN-EVENT-
↪MIB::sysUpTimeInstance = Timeticks: (590254) 1:38:22.54
#011SNMPv2-MIB::snmpTrapOID.0 = OID: NET-SNMP-MIB::netSnmp.99999#011NET-SNMP-
↪MIN::netSnmp.99999.1 = STRING: "hello world"
```

また、Kompira 上で実行していたジョブフロープロセスのコンソールには以下のような受信結果が表示されます。

```
<UNKNOWN>
UDP: [192.168.213.101]:56313->[192.168.213.100]
DISMAN-EVENT-MIB::sysUpTimeInstance 0:0:18:39.04
SNMPv2-MIB::snmpTrapOID.0 NET-SNMP-MIB::netSnmp.99999
NET-SNMP-MIB::netSnmp.99999.1 "hello world"
```

第 7 章

Kompira の監視

著者　Kompira 開発チーム

7.1 はじめに

このドキュメントでは、Zabbix などの監視システムを用いて Kompira の状態を監視する方法について説明します。

7.2 Zabbix での監視

Zabbix で Kompira の動作中のプロセス数や対応中インシデントの数などを取得する方法について説明します。

Zabbix の監視方法は様々なものがありますが、ここでは Zabbix Agent の「UserParameter」 機能を用いた監視と、「外部スクリプト」による監視の方法について説明します。

※ このドキュメントでは Zabbix 2.4 を用いた監視方法について紹介します。

7.2.1 準備

kompira_jq.sh

いずれの監視方法でも、Kompira が提供するスクリプト `kompira_jq.sh` を利用します。

なお、「外部スクリプト」による監視の場合は Zabbix サーバ上で、「UserParameter」による監視の場合は Zabbix Agent をインストールした Kompira サーバ上で、`kompira_jq.sh` を実行します。

`kompira_jq.sh` は内部で curl コマンドおよび jq コマンドを利用していますので、これらが利用できるよう監視方法に合わせて Zabbix サーバまたは Kompira サーバに、必要なパッケージをインストールしておいてください。

※ CentOS 環境では jq は EPEL リポジトリから、AWS 環境では amzn-main リポジトリからインストールできます。

注釈: Kompira 1.4.6 以降に付属する kompira_jq.sh は REST-API 対応版になったため、旧バージョンに付属するものとオプション指定方法などで互換性がありません。

Kompira サーバのホスト設定

いずれの監視方法でも Kompira サーバに対してアクセスを行なうため、Kompira サーバの URL と REST API トークンを Zabbix の「マクロ」として登録しておく必要があります。

Zabbix 上で Kompira サーバの「ホスト設定」→「マクロ」設定画面で、以下のマクロ名で REST API トークンを設定してください。

Macro	Value
{$KOMPIRA_URL}	Kompira サーバの URL
{$KOMPIRA_TOKEN}	REST API トークン

7.2.2 UserParameter による監視

こちらは Zabbix Agent が Zabbix サーバから指定された項目に対して事前に設定されたコマンドを実行することで、監視項目の値を収集する方法になります。

Zabbix Agent の設定

Zabbix Agent をインストールした Kompira サーバ上に、UserParameter の設定ファイルを準備する必要があります。/etc/zabbix/zabbix_agentd.d に userparameter_kompira.conf をコピーしてください。

設定ファイルが準備できたら Zabbix Agent を再起動してください。

```
$ sudo service zabbix-agent restart
Shutting down Zabbix agent:                                [  OK  ]
Starting Zabbix agent:                                     [  OK  ]
```

Zabbix Server の設定

Zabbix Server には UserParameter を利用した監視項目を設定する必要がありますが、zbx_kompira_basic_templates.xml をインポートすることで標準的な監視項目をすぐに利用できます。

Template Kompira Server という名前のテンプレートが作成されますので、監視したい Kompira サーバにこのテンプレートを適用してください。

監視項目

標準で以下の監視項目を利用可能です。

Name	概要
Kompira active incidents	アクティブなインシデント数
Kompira active processes	アクティブなプロセス数
Kompira active schedulers	アクティブなスケジュール数
Kompira active tasks	アクティブなタスク数
Kompira jobflows	ジョブフロー総数
Kompira license remain_days	ライセンスの残り日数
Kompira objects	Kompira オブジェクト総数
Memory usage of kompirad process	メモリ使用量 (kompirad)
Memory usage of kompira_jobmngrd process	メモリ使用量 (kompira_jobmngrd)
Number of kompirad process	プロセス数 (kompirad)
Number of kompira_jobmngrd process	プロセス数 (kompira_jobmngrd)

7.2.3 外部スクリプトによる監視

こちらは Zabbix サーバ上で外部のスクリプトを実行して、監視項目の値を収集する方法になります。

まず、Kompira が提供するスクリプト `/opt/kompira/bin/kompira_jq.sh` を、Zabbix サーバ上の外部スクリプトを配置するディレクトリにコピーしておいてください。デフォルトでは `/usr/lib/zabbix/externalscripts` になります。

プロセス数

外部スクリプト `kompira_jq.sh` を用いてプロセス数を監視する場合、以下のような設定で Item を作成してください。

Name	`Kompira processes`
Type	External check
Key	`kompira_jq.sh[-s,{$KOMPIRA_URL},-t,{$KOMPIRA_TOKEN},-ac,/process]`
Type of information	Numeric (unsigned)
Data type	Decimal

インシデント数

外部スクリプト `kompira_jq.sh` を用いてインシデント数を監視する場合、以下のような設定で Item を作成してください。

Name	`Kompira incidents`
Type	External check
Key	`kompira_jq.sh[-s,{$KOMPIRA_URL},-t,{$KOMPIRA_TOKEN},-ac,/incident]`
Type of information	Numeric (unsigned)
Data type	Decimal

第 8 章

Kompira REST API リファレンス

著者　Kompira 開発チーム

8.1 イントロダクション

本ドキュメントは、REST API の仕様について記述します。

8.2 共通

8.2.1 エンドポイント

REST API のエンドポイントは、通常の Kompira オブジェクトへのリソースパスと同様です。すなわちルートエンドポイントは

```
http[s]://<hostname>/
```

となります。

ブラウザからのアクセスと API リクエストを区別するために、HTTP リクエストの Accept ヘッダに

```
Accept: application/json
```

を含める必要があります。

もしくは、クエリ文字列に `format=json` を含める方法もあります。

8.2.2 ユーザ認証

認証はトークン認証と、セッション認証方式の 2 種類が許可されます。セッション認証方式はブラウザからのアクセスを想定したものなので、REST API のクライアントでは通常は使用しません。

トークン認証を使用する場合、以下のようにリクエストの Authorization ヘッダにトークン鍵を含めます。

```
Authorization: Token <トークン鍵>
```

または、HTTP リクエストのクエリ文字列に以下のようにトークン鍵を含める方法もあります。

```
token=<トークン鍵>
```

ユーザ設定ページにて、各ユーザの REST API を有効にすると、アクセストークンが生成されます。REST API を無効にし、再度有効にすると、トークンが再初期化されます。

8.2.3 フォーマット

データフォーマットは JSON 形式のみサポートしています。

日時型データ (Datetime)

日時型データは UTC であり、以下の形式 (ISO8601) が用いられます。

```
%Y-%m-%dT%H:%M:%S.%fZ もしくは %Y-%m-%dT%H:%M:%SZ
```

入力時にはマイクロ秒や秒を省略することが可能です。末尾の UTC 指示子である Z を省いた場合は、ローカル時刻（JST) とみなされ、内部で UTC に変換されます。

オブジェクト型データ (Object)

オブジェクト型データはオブジェクトの絶対パスにより表現されます。

ファイル型データ (File)

ファイル型データは出力時は、以下のように name, path をキーに持つ辞書型データとなります。

```
{ "name": "<ファイル名>", "path": "<ファイルパス>" }
```

入力時は、name と data をキーに持つ辞書型データとします。

```
{ "name": "<ファイル名>", "data": "<ファイルのデータ文字列>" }
```

8.2.4 エラー

エラー時は、エラーを示す HTTP ステータスコードを返します。この時、 HTTP レスポンスのコンテンツボディにはエラー理由を示すデータが含まれます。

多くのエラーデータは以下のように detail キーを含む辞書データとなります。

```
{ "detail": "<エラーの理由>" }
```

必須のフィールドがリクエストデータに含まれていないようなバリデーションエラーの場合、以下のような辞書データを返します。

```
{ "<フィールド名>": ["<エラーメッセージ>", ...],
  "<フィールド名>": ["<エラーメッセージ>", ...],
... }
```

8.2.5 ページネート

一覧取得の場合、以下の形式のページネートされたデータが返されます。

```
{
    "count": <オブジェクト総数>,
    "next": <次ページの URL>,
    "previous": <前ページの URL>,
    "results": <オブジェクトデータリスト>
}
```

ページを指定して取得する場合、クエリパスに page=<ページ番号> を含めます。最後のページを取得したい場合は、ページ番号に last を指定します。

ページサイズのデフォルトは 25 です。ページサイズを変更したい場合、

```
page_size=<ページサイズ>
```

をクエリ文字列に含めます。

8.2.6 フィルタリング

一覧取得において、オブジェクトの属性でフィルタリングするには、クエリパスに <属性名>=<値> を指定します。

たとえば、正常に完了したプロセスのみの一覧を取得する場合、以下のように指定します。

```
/process?status=DONE
```

また、以下のように複数の属性をクエリパスで指定した場合は、AND によるフィルタリングとなります。

```
/app.descendant?display_name=test&owner=root
```

上記は完全一致によるフィルタリングですが、以下のように属性名に続けてルックアップを記述することで、細かなフィルタリング条件の指定が可能です。

```
<属性名>__<ルックアップ>=<値>
```

たとえば、表示名に test を含むオブジェクトをフィルタリングできます。

```
/app.descendant?display_name__contains=test
```

ルックアップの種類ごとの値のフィルタリング方式は以下のとおりです。

ルックアップ	フィルタリング方式
exact, iexact	属性が指定した値に一致する。iexact では大小文字を区別しない。
contains, icontains	属性が指定した値を含む。icontains では大小文字を区別しない。
startswith, istartswith	属性が指定した値で始まる。istartswith では大小文字を区別しない。
endswith, iendswith	属性が指定した値で終わる。iendswith では大小文字を区別しない。
regex, iregex	属性が指定した正規表現にマッチする。iregex では大文字小文字区別しない。
gt, gte	属性が指定した値より大きい (gt)、または、属性が指定した値以上である (gte)。
lt, lte	属性が指定した値より小さい (lt)、または、属性が指定した値以下である (lte)。
in	属性が指定した値に含まれる。

仮想オブジェクト以外の一般のオブジェクトにおける属性値によるフィルタリングでは、属性によって指定できるルックアップが異なります。属性の一覧と指定できるルックアップは以下のとおりです。

属性	指定できるルックアップ
owner	exact, in
abspath	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
display_name	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
description	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
created	exact, gt, gte, lt, lte
updated	exact, gt, gte, lt, lte
type_object	exact, in

また、仮想オブジェクトでは、属性のデータ型によって指定できるルックアップが異なります。

属性の型	指定できるルックアップ
文字列型	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex
整数型	exact, gt, gte, lt, lte
日時型	exact, gt, gte, lt, lte
オブジェクト型	exact
ユーザ型	exact
真偽型	exact

なお、ルックアップを指定していないときは exact が適用されます。

オブジェクト型 (Object)

仮想オブジェクト以外の一般のオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
owner	所有者 (ユーザ型)
display_name	表示名 (文字列型)
description	説明 (文字列型)
created	作成日時 (日時型)
updated	更新日時 (日時型)
type_object	型オブジェクト (オブジェクト型)

さらに、型オブジェクトが特定される状況では、フィールド値によるフィルタリングも指定できます。

```
field:<フィールド名>__<ルックアップ>=<値>
```

たとえば、以下のように指定します。

```
/.descendant?type_object=/system/types/Jobflow&field:source__contains=urlopen&
↪field:defaultMonitoringMode=MAIL
```

型オブジェクトが特定される状況とは、以下のいずれかを言います。

- type_object 属性フィルタリングによって型オブジェクトが指定されている
- 起点となるオブジェクトがテーブル型であり、テーブルに型オブジェクトが設定されている

型オブジェクトが特定されていない状況で、フィールド値によるフィルタリングを指定するとエラーになります。

なお、フィールド値によるフィルタリングでは、フィールドのデータ型に応じて指定できるルックアップが異なります。

フィールドの型	指定できるルックアップ
文字列型	exact, iexact, contains, icontains, startswith, istartswith, endswith, iendswith, regex, iregex, in, range
整数型	exact, isnull, gt, gte, lt, lte, in, range
真偽値型	exact
日時型	exact, isnull, gt, gte, lt, lte, range
オブジェクト型	exact, isnull
添付ファイル型	(文字列型と同じ、ファイル名がフィルタ対象になります)
配列型	(文字列型と同じ、配列の値がフィルタ対象になります)
辞書型	(文字列型と同じ、辞書の値がフィルタ対象になります)

プロセス型 (Process)

プロセスオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
job	ジョブオブジェクト (オブジェクト型)
user	実行ユーザー (ユーザ型)
started_time	開始日時 (日時型)
finished_time	終了日時 (日時型)
status	ステータス [exact]
schedule	スケジュールオブジェクト (オブジェクト型)
parent	親プロセス (プロセス型)
current_job	実行中ジョブオブジェクト (オブジェクト型)
suspended	一時停止フラグ (真偽型)
lineno	実行中行番号 (整数型)
console	コンソール (文字列型)

スケジュール型 (Scheduler)

スケジュールオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
name	スケジュール名 (文字列型)
description	説明 (文字列型)
user	ユーザー (ユーザ型)
job	ジョブオブジェクト (オブジェクト型)
month	月 [exact, contains]
day	日 [exact, contains]
week	週 [exact, contains]
day_of_week	曜日 [exact, contains]
hour	時 [exact, contains]
minute	分 [exact, contains]
disabled	無効化 (真偽型)

インシデント型 (Incident)

インシデントオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
name	インシデント名 (文字列型)
device	デバイス名 (文字列型)
service	サービス名 (文字列型)
created_date	作成日時 (日時型)
closed_date	完了日時 (日時型)
status	ステータス [exact]
owner	所有者 (ユーザ型)

タスク型 (Task)

タスクオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
name	タスク名 (文字列型)
title	タイトル (文字列型)
message	メッセージ (文字列型)
status	ステータス [exact]
owner	所有者 (ユーザ型)
created_date	作成日時 (日時型)
closed_date	完了日時 (日時型)

ユーザー型 (User)

ユーザーオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
username	ユーザー名 (文字列型)
first_name	名 (文字列型)
last_name	姓 (文字列型)
email	E-mail (文字列型)
last_login	最終ログイン日時 (日時型)
is_active	有効化 (真偽型)
home_directory	ホームディレクトリ (オブジェクト型)
environment	環境変数 (オブジェクト型)

グループ型 (Group)

グループオブジェクトのフィルタリングに使用できる属性は以下のとおりです。

属性名	意味 (属性の型) または [指定できるルックアップ]
name	グループ名 (文字列型)

8.3 Kompira オブジェクトへのアクセス

8.3.1 オブジェクト情報の取得

リクエスト

- GET <オブジェクトパス>

応答

```
{
    "id": <オブジェクト ID>,
    "abspath": <オブジェクトパス>,
    "owner": <オブジェクト所有ユーザー名>,
    "fields": <フィールドデータ辞書>,
    "extra_properties": <オブジェクト拡張属性>,
    "user_permissions": <ユーザーパーミッション辞書>,
    "group_permissions": <グループパーミッション辞書>,
    "display_name": <表示名>,
    "description": <説明>,
    "created": <オブジェクト生成日時>,
    "updated": <オブジェクト最終更新日時>,
    "type_object": <型オブジェクトのパス>,
    "parent_object": <親オブジェクトのパス>
}
```

フィールドデータ辞書とオブジェクト拡張属性は、オブジェクトの型によって異なるキーを含む辞書データです。

8.3.2 オブジェクト情報の更新

リクエスト

- PUT <オブジェクトパス>
- PATCH <オブジェクトパス>

PUT はオブジェクトのデータ全体を置き換えます。オブジェクトの一部更新の場合は、PATCH リクエストを使用します。

リクエストデータ

```
{
    "owner": <オブジェクト所有ユーザー名>,
    "fields": <フィールドデータ辞書>,
    "user_permissions": <ユーザーパーミッション辞書>,
    "group_permissions": <グループパーミッション辞書>,
    "display_name": <表示名>,
    "description": <説明>                    # 省略可能
}
```

PATCH リクエストの場合、属性を省略するとそのキーに対応するオブジェクトの値は変更されません。

応答　更新されたオブジェクトのデータ

8.3.3 オブジェクトの新規追加

ディレクトリオブジェクト、もしくは、テーブルオブジェクトに対して POST リクエストを送信すると、オブジェクトが新規に作成されます。

リクエスト

- POST <ディレクトリ又はテーブルのオブジェクトパス>

リクエストデータ

```
{
    "owner": <オブジェクト所有ユーザー名>,# 省略可能（省略時はリクエストユーザーが所有者となる）
    "fields": <フィールドデータ辞書>,
    "name": <オブジェクト名>,
    "user_permissions": <ユーザーパーミッション辞書>,
    "group_permissions": <グループパーミッション辞書>,
    "display_name": <表示名>,# 省略可能
    "description": <説明>,# 省略可能
    "type_object": <型オブジェクトのパス>
}
```

応答　HTTP 201 Created 応答と新規追加されたオブジェクトのデータ

8.3.4 オブジェクトの削除

オブジェクトパスに対して DELETE リクエストを送信することで、そのオブジェクトを削除することができます。

プロセス、スケジュール、インシデント、タスク、の各オブジェクトも同様に削除することができる。

リクエスト

- DELETE <オブジェクトパス>

応答　成功すると HTTP 204 No Content が返されます。

8.3.5 子/子孫オブジェクトの一覧取得

ディレクトリオブジェクト、もしくは、テーブルオブジェクトの子や子孫オブジェクトの一覧を取得することができます。

リクエスト

- GET <オブジェクトパス>.children# 子オブジェクト一覧
- GET <オブジェクトパス>.descendant# 子孫オブジェクト一覧

応答　オブジェクトデータの一覧が返されます。

注釈: ディレクトリやテーブル以外のオブジェクトの場合、空の一覧が返されます。

8.3.6 ジョブフローの実行

リクエスト

- POST <ジョブフローパス>.execute

リクエストデータ

```
{
    "step_mode": <ステップモード>,                # true or false
    "checkpoint_mode": <チェックポイントモード>, # true or false
    "monitoring_mode": <監視モード>,              # NOTHING, MAIL, ABORT_MAIL
    "parameters": <ジョブフローパラメータの辞書>
}
```

応答　実行されたジョブフロープロセスのパスが返されます。

8.3.7 スクリプトジョブの実行

リクエスト

- POST <スクリプトジョブパス>.execute

リクエストデータ

```
{
    "node": <実行ノードオブジェクトのパス>,
    "account": <実行アカウントオブジェクトのパス>,
    "command_line": <コマンドライン文字列>
}
```

応答　実行されたジョブフロープロセスのパスが返されます。

8.3.8 メッセージ送信

チャネルオブジェクトに対してメッセージ送信が可能です。

リクエスト

- POST <チャネルオブジェクトパス>.send

リクエストデータ　任意の JSON 形式のデータ

応答　成功すると HTTP 200 OK が返されます。

8.3.9 メッセージ受信

チャネルオブジェクトからメッセージを受信します。

リクエスト

- POST <チャネルオブジェクトパス>.recv

リクエストデータ

```
{
    "timeout": <タイムアウト値（秒）>
}
```

応答　成功すると受信データが返されます。受信タイムアウトすると、ステータスコードとして HTTP 408 Request Timeout が返されます。

注釈: チャネルにデータが無い場合は、タイムアウトで指定された秒数だけ待ちます。タイムアウトのデフォルト値は 0 秒です。

8.4 プロセス

8.4.1 プロセス一覧の取得

リクエスト

- GET /process

子プロセス一覧を取得する場合は、以下のリクエストを投げます。

- GET /process/id_<プロセス ID>.children

応答　プロセス詳細データの一覧が返されます。

8.4.2 プロセス詳細の取得

リクエスト

- GET /process/id_<プロセス ID>

応答

```
{
    "id": <プロセス ID>,
    "abspath": <プロセスオブジェクトパス>,
    "user": <実行ユーザー名>,
    "elapsed_time": <プロセス経過時間>,
    "started_time": <実行開始日時>,
    "finished_time": <実行終了日時>,
    "status": <実行ステータス>,
    "exit_status": <終了時ステータス>,
    "result": <実行結果>,
    "error": <実行エラー>,
    "suspended": <停止中>,
    "lineno": <行番号>,
    "console": <コンソール文字列>,
    "job": <実行開始ジョブフローパス>,
    "schedule": <スケジュールパス>,
    "parent": <親プロセスパス>,
    "current_job": <実行中ジョブフローパス>
}
```

8.4.3 プロセスの操作

リクエスト

- POST /process/id_<プロセス ID>.terminate # プロセス実行を中止する
- POST /process/id_<プロセス ID>.suspend # プロセス実行を一時停止する
- POST /process/id_<プロセス ID>.resume # プロセス実行を再開する

リクエストデータ　resume において以下のリクエストデータを渡すことで、再開時の実行モードを指定することができます。

```
{
    "step_mode": <ステップモード>,                  # true/false を指定
    "checkpoint_mode": <チェックポイントモード>    # true/false を指定
}
```

応答　成功時は true、失敗時は false が返されます。

8.4.4 プロセスの実行完了を待つ

リクエスト

- POST /process/id_<プロセス ID>.wait

リクエストデータ

```
{
    "timeout": <タイムアウト値>                    # 0 以上の整数値を指定
}
```

応答　成功時はプロセスの詳細情報が返されます。タイムアウトすると、ステータスコードとして HTTP 408 Request Timeout が返されます。

8.5 スケジュール

8.5.1 スケジュール一覧の取得

リクエスト

- GET /scheduler

応答　スケジュール詳細データの一覧が返されます。

8.5.2 スケジュール詳細の取得

リクエスト

- GET /scheduler/id_<スケジュール ID>

応答

```
  {
      "id": <スケジュール ID>,
      "abspath": <スケジュールオブジェクトパス>,
      "user": <ユーザー名>,
      "scheduled_datetimes: <実行予定日時一覧>,
      "parameters": <パラメータリスト>,
      "name": <スケジュール名>,
      "description": <説明>,
      "year": <年>,
      "month": <月>,
      "day": <日>,
      "week": <週>,
      "day_of_week": <曜日>,
      "hour": <時>,
      "minute": <分>,
      "disabled": <無効化>,
      "job": <実行ジョブオブジェクトパス>
}
```

8.5.3 スケジュールの更新

リクエスト

- PUT /scheduler/id_<スケジュール ID>
- PATCH /scheduler/id_<スケジュール ID>

リクエストデータ

```
{
    "user": <ユーザー名>,
    "parameters": <パラメータリスト>,
    "name": <スケジュール名>,              # 必須パラメータ
    "description": <説明>,
    "year": <年>,
    "month": <月>,
    "day": <日>,
    "week": <週>,
    "day_of_week": <曜日>,
    "hour": <時>,
    "minute": <分>,
    "disabled": <無効化>,
    "job": <実行ジョブオブジェクトパス>   # 必須パラメータ
}
```

応答　更新されたオブジェクトのデータ

8.5.4 スケジュールの作成

リクエスト

- POST /scheduler

リクエストデータ　更新リクエストと同様

応答　作成されたオブジェクトのデータ

8.6 インシデント

8.6.1 インシデント一覧の取得

リクエスト

- GET /incident

応答　インシデント詳細データの一覧が返される。

8.6.2 インシデント詳細の取得

リクエスト

- GET /incident/id_<インシデント ID>

応答

```
{
    "id": <インシデント ID>,
    "abspath": <インシデントオブジェクトパス>,
    "owner": <所有者>,
    "worklogs": <作業ログ一覧>,
    "alerts": <アラート一覧>,
    "name": <インシデント名>,
    "device": <デバイス名>,
    "service": <サービス名>,
    "created_date": <作成日時>,
    "closed_date": <完了日時>,
    "status": <ステータス>                    # "OPENED", "WORKING", "CLOSED" のいずれか
}
```

8.6.3 インシデントの更新

リクエスト

- PUT /incident/id_<インシデント ID>
- PATCH /incident/id_<インシデント ID>

リクエストデータ

```
{
    "owner": <所有者>,
    "name": <インシデント名>,
    "device": <デバイス名>,
    "service": <サービス名>,
    "status": <ステータス>
}
```

応答　更新されたオブジェクトのデータ

8.6.4 作業ログの追加

リクエスト

- POST /incident/id_<インシデント ID>.worklogs

リクエストデータ

```
{
    "user": <ユーザー名>,
    "description": <作業ログ>
}
```

応答　追加された作業ログのデータ

8.6.5 インシデントの作成

リクエスト

- POST /incident

リクエストデータ　更新リクエストと同様

応答　作成されたオブジェクトのデータ

8.7 タスク

8.7.1 タスク一覧の取得

リクエスト

- GET /task

応答　タスク詳細データの一覧

8.7.2 タスク詳細の取得

リクエスト

- GET /task/id_<タスク ID>

応答

```
{
    "id": <タスク ID>,
    "abspath": <タスクオブジェクトパス>,
    "owner": <所有者>,
    "assigned_users": <宛先ユーザー一覧>,
    "assigned_groups": <宛先グループ一覧>,
    "name": <タスク名>,
    "title": <タスク件名>,
    "message": <タスクメッセージ>,
    "action_text": <アクション文字列>,
    "result": <結果>,
    "status": <ステータス>,       # "WAITING", "ONGOING", "DONE", "CANCELED" のいずれか
    "created_date": <作成日時>,
    "closed_date": <完了日時>
}
```

8.7.3 タスクのキャンセル

リクエスト

- POST /task/id_<インシデント ID>.cancel

応答 ::　成功すると HTTP 200 OK が返されます。

8.7.4 タスクの送信

タスクチャネルに対してメッセージを送信します。

リクエスト

- POST /task/id_<インシデント ID>.submit

リクエストデータ

```
{
    "result": <結果メッセージ>
}
```

リクエストデータを省略した場合は、タスクチャネルに "OK" が渡されます。

8.8 ユーザー/グループ管理

8.8.1 ユーザー一覧の取得

リクエスト

- GET /config/user

応答　ユーザー詳細データの一覧

8.8.2 ユーザー詳細の取得

リクエスト

- GET /config/user/id_<ユーザー ID>

応答

```
{
    "id": <ユーザー ID>,
    "abspath": <ユーザーオブジェクトパス>,
    "groups": <グループ一覧>,
    "last_login": <最終ログイン日時>,
    "username": <ユーザー名>,
    "first_name": <名>,
    "last_name": <姓>,
    "email": <E-mail アドレス>,
    "is_active": <有効フラグ>,
    "home_directory": <ホームディレクトリ>,
```

(次のページに続く)

(前のページからの続き)

```
    "environment": <環境オブジェクト>
}
```

8.8.3 ユーザーの更新

リクエスト

- PUT /config/user/id_<ユーザー ID>
- PATCH /config/user/id_<ユーザー ID>

リクエストデータ

```
{
    "groups": <グループ一覧>,
    "last_login": <最終ログイン日時>,
    "username": <ユーザー名>,            # 必須フィールド
    "password": <パスワード>,            # 必須フィールド
    "first_name": <名>,
    "last_name": <姓>,
    "email": <E-mail アドレス>,
    "is_active": <有効フラグ>,
    "home_directory": <ホームディレクトリ>,
    "environment": <環境オブジェクト>
}
```

応答　更新されたオブジェクトのデータ

8.8.4 ユーザーの新規作成

リクエスト

- POST /config/user

リクエストデータ　ユーザー更新と同じ

応答　作成されたオブジェクトのデータ

8.8.5 グループ一覧の取得

リクエスト

- GET /config/group

応答　グループ詳細データの一覧

8.8.6 グループ詳細の取得

リクエスト

- GET /config/group/id_<グループ ID>

応答

```
{
    "id": <グループ ID>,
    "abspath": <グループオブジェクトパス>,
    "name": <グループ名>
}
```

8.8.7 グループの更新

リクエスト

- PUT /config/group/id_<グループ ID>
- PATCH /config/group/id_<グループ ID>

リクエストデータ

```
{
    "name": <グループ名>
}
```

応答　更新されたオブジェクトのデータ

8.8.8 グループの新規追加

リクエスト

- POST /config/group

リクエストデータ　グループ更新と同じ

応答　更新されたオブジェクトのデータ

索引