入門Ansible

*2014*年*8*月版 日経様

若山 史郎

2014年8月1日

目次

クラウドの利用が大きく広がり、昨今では大量のサーバーを少人数で扱う必要が増えています。それにつれて、省力化・自動化の要望が大きくなっています。

そのような背景があり、複数台のホストを扱える構成管理ツールやデプロイを行うツールが注目を集めています。そのうちの一つが Ansible です。

この本は、

- 「Ansible とはなんだろう」
- 「Ansible を使ってみたいがどこから手を付けていいか分からない」

という方に対して、Ansible の説明とその使い方について説明します。この本を読み終える頃には Ansible について理解し、複数のホストを管理できていると思います。また、付録や知っていると便利なことなども随所に記載しておりますので、なにか分からないことが出てきたらこの本を再び読むと解決できるかもしれません！

この本では Ansible について次の構成で説明します。

1 章 「**はじめに**」では Ansible の特徴を概説します。

2 章 「*Ansible* **を使ってみよう**」では Ansible のインストールおよび ansible コマンドを実行して複数台のホスト上で任意のコマンドを実行できることを説明します。

3 章 「*playbook* **を作ってみよう**」では複数のコマンドを実行する基礎的な playbook を作成し、ansible-playbook コマンドを使って多くの作業を自動化していきます。

4 章 「**複雑な** *playbook* **を作ってみよう**」では 3 章で説明しきれなかった多彩な機能について述べ、柔軟な処理ができることを説明します。

5 章 「**大規模な** *playbook* **を構築してみよう**」ではさらに大規模な構成になった場合について説明します。また、重要な再利用方法である role についても説明します。

6 章 「**コマンドラインオプションを使ってみよう**」ではコマンドラインオプションについて説明します。コマンドラインからでも多くの指示ができます。

7 章 「**変数ファイルの暗号化** *– ansible-vault*」では変数を暗号化するコマンドである ansible-vault について説明します。

8 章 「**公開されている** *role* **を使ってみよう** *– Ansible Galaxy*」では role を全世界で共有する Ansible Galaxy とその使い方について説明します。

この本は Ansible について聞いたことがあるという方以外に、次のような方々を対象としています。

- 今手作業で行っていることを自動化したい方
- 管理するサーバーの数がどんどん増えて、手が回らなくなってきた方
- 構成管理ツールを使っていたが、複雑になりすぎてしまった方

また本書は、奥野 晃裕さん、tboffice さん、中村 弘輝さん、futoase さんにレビューして頂きました。皆さまからさまざまな視点での指摘を頂き、大変参考になりました。ありがとうございます。

第1章　はじめに

最近では一人が数十台、数百台ものサーバーを管理することも珍しくありません。また、クラウド上にものの数分でサーバーを作成、あるいは削除することも日常茶飯事です。

その場合、

- 新規作成したサーバーに多数のソフトウェアをインストールする
- 設定ファイルを適切に管理・維持する

といったことが必要になってきます。これを手動で多数のサーバーに対して実行することは困難ですし、間違いが入り込む余地は非常に大きいです。

そこで重要な位置づけとなってきたツールが**構成管理ツール**です。構成管理ツールとは、ソフトウェアや設定ファイルなどの対象とするサーバーの構成を適切に管理し、維持するツールです。新規作成したサーバーなどは指定した構成になっておりませんので、構成管理ツールはそのサーバーを自動的に適切な構成へと変更します。

Ansible は構成管理ツールの一つです。ただ、それだけにとどまらない、任意のコマンドをリモートで実行したり、結果を取得したりする**オーケストレーションツール**でもあります。

オーケストレーションツールの正確な定義は定まっておりませんが、ここでは以下の操作ができるものとします。

- 構成管理が可能
- ソースコードをサーバーにコピーする、再起動する、などのデプロイが可能
- アドホックコマンドにより、任意のコマンドをいつでも複数のサーバーに実行可能
- 他のシステムとの連携をし、複雑な業務のワークフローを支えるシステムを構築可能

Ansible は、これらのことがすべてできます。単なる構成管理ではなく、デプロイなどの定形作業、障害時の非定形作業、あるいは他のシステムとの連携し、通知や自動実行まで Ansible 一つで行えます。

個々の操作はそれぞれ別々のツールがあると思います。しかし、これらを一つで統一的にこなせるツールは現状では Ansible が最も有力でしょう。

コラム: ライセンスと開発状況

Ansible は GitHub 上でオープンソース (GPL v3) で開発されているソフトウェアです。Ansible 社 (本社アメリカ) が中心となって開発されていますが、コミュニティが活発で、プルリクエスト申請の数も常時 300 を超え、取り込まれる早さを上回っています。GitHub で注目度が高いプロジェクトに付けられるスターは 6000 を超え、GitHub でも有数の注目プロジェクトです。

1.1 Ansible の特徴

構成管理ツールおよびオーケストレーションツールについて説明したところで、ここでは Ansible の特徴を述べ、どのようなものなのか解説していきます。

1.1.1 ssh だけがあればいい

Chef や Puppet など、多くの構成管理ツールは、エージェントと呼ばれるソフトウェアを操作対象のホストにインストールする必要があります。

しかし、Ansible は “ssh” が使えればそれだけで使い始められます (実際には Python が必要ですが、ほぼすべての OS に標準で付属しています)。ssh は UNIX 系 OS のホストを管理するためには必須のツールであり、極めて一般的です。

サーバーが他社によって管理されており、自由にソフトウェアをインストールすることが出来ない場合もあります。その場合にも Ansible は使えます。さらに言うと、ssh を適切に設定することにより、踏み台サーバー経由で管理することも出来ます。これらは Ansible の適用場所を広げてくれます。

1.1.2 必要な準備が最小限

Ansible を使い始めるのに必要なファイルはたった 1 つ、**inventory** と呼ばれるファイルだけです。

しかもその内容は

```
192.168.0.1
192.168.0.2
```

と、管理したいホストの IP アドレスを列挙するだけです。これで複数台のホストに対してコマンドを発行することができるようになります。

例えば以下のようなコマンドを打つことで

```
$ ansible all -m command -a "uptime"
```

intenvory ファイルに書かれているすべてのサーバーで “uptime” コマンドを実行できます。

実際には後ほど説明する playbook ファイルを記述し、より多くのことを行いますが、使いはじめるために準備する設定は、この inventory ファイルだけです。

1.1.3 動作順序が分かりやすい

Ansible は、ほぼすべての動作が、設定ファイルに書かれた順に上から下へと動作します。

他の構成管理システムでは宣言的な設定ファイルを使用し、動作順序をシステムが決定する、というものもあります。この方式は、あるべき姿にシステムを収束させていくという観点で見ると非常に正しいものだと言えます。

しかし、現実のシステムはなかなかそううまくいきません。この設定をしてからこのユーザーを追加して、など順序が必要な処理が多々あります。

Ansible ではユーザーが記述した順番そのままで動くので、分かりやすく一目瞭然となります。

1.1.4 べき等性がある

べき等性 (Idempotency) とは、操作を何度も行っても同じ結果になることを指します。

例えば、べき等性がない場合「“spam” という行をファイルに追加する」という操作を 2 回行うと spam という行が二行できます。べき等性がある場合は一行だけとなります。

べき等性は構成管理を行う上で重要な性質です。なぜなら、先ほどの例のように設定ファイルが変わってしまうと問題が起きるというからです。さらに、すでに変更がなされている場合はなにもしない、と実装することで実行速度が早くできます。

Ansible には基本的にべき等性があります。従って、複数回実行しても同じ結果となります。

ただし、Ansible のべき等性はモジュールと呼ばれる個々の機能ごとで実現されています。べき等性があるモジュールが多いですが、例えばメールを送るなど、べき等性がないモジュールもあります。メールの場合、実行した回数だけメールを送ります。

1.1.5 既存資産を活かせる

多くの人がすでに shell script を用いていろいろな管理をしているかと思います。

Ansible には `script` というモジュールがあり、これは手元にある shell script を対象のホストに送りこみ、そのホストで実行してくれます。つまり、既存の shell script 資産をそのまま複数台に適用できるのです。

1.1.6 プログラマでなくても使いやすい

Ansible は Python で実装されていますが、使う上では Python を書くことはまったくと言っていいほどありません。

設定ファイルは 2 種類あり、一つは先ほど述べた inventory ファイルです。一般的に記述する回数が多い playbook ファイルは YAML 形式と呼ばれる形式で記載します。以下にその例を書きます。

```
---
- hosts: all
  tasks:
    - name: 必要なソフトウェアを yum でインストールする
      yum: name={{ item }} state=present
      with_items:
         - python
         - ruby

    - name: 手元にある foo.sh を対象ホストに送って実行する
      script: foo.sh
```

詳しくは後ほど説明しますが、–で項目を並べるということが分かれば、ある程度理解できるのではないでしょうか。

また、name はなにを実行するかという説明です。UTF-8 で保存すれば、ここに日本語も記載でき、コメントとして使えます。プログラマ以外にも分かりやすくなりますし、後から見直すときにも便利です。複数人で管理をしている時にも便利でしょう。

1.1.7 追加インストールが不要

Ansible は “Batteries included(直訳: 電池付属)” というポリシーを持っており、非常に多くのことが最初からできるようになっています。

少しだけ例をを上げると、

- htpasswd 設定
- Google Computing Engine のインスタンスを起動
- Amazon Route53 設定
- nagios の設定 (アラートの無効化/有効化など)
- arista (ネットワーク機器) の設定
- apache2 のモジュールを有効化/無効化

などなど多くのことが出来ます。手作業で行っても時間がかかることを数行記述するだけで実行でき、しかもそれが何台に対しても実行できるのです。

Ansible を使わない場合、それぞれの設定は HTTP だったり設定ファイルだったりと千差万別な方式で行われますので、変更するにはまず仕様を調べなければいけません。しかし Ansible を使うと、Ansible の設定ファイルという共通の形式で設定できます。

1.1.8 どんなコマンドでもその場で実行 – アドホックコマンド

Ansible は構成管理だけを行うツールではありません。Ansible が他の構成管理ツールと違うところは、一つのツールで構成管理とコマンド実行の両方ができることです。

つまり、従来は

- ホストの構成管理は Chef や Puppet
- 新しいソフトウェアのデプロイは Capistrano や Fabric

というように、複数種類のツールを組み合わせる必要がありましたが、これが Ansible だけ覚えれば良い、ということになります。

Ansible は、構成管理だけでなくデプロイにも使えます。また、複数台のホストに対して一つのコマンドを実行して、結果をファイルに書き出すこともできます。障害時には非常に役立つでしょう。

1.2 Ansible はシンプル

ここまで Ansible の特徴を並べてきました。多くの特徴がありますが、Ansible の基本理念として、**シンプル**であることが貫かれています。

Ansible はデザイン指針として下の 5 つを設定しています。

- Simply Clear (分かりやすい)
- Simply Fast (習得もインストールも早い)
- Simply Complete (全部を備えているからすぐに使える)
- Simply Efficient (ssh は効率的)
- Simply Secure (ssh は安全)

すべてに Simply と付いていることからシンプルを重視していることが分かると思います。

1.3 Chef や Puppet との違い

構成管理ツールとして、CFEngine から始まり、Puppet・Chef など多くのツールがすでに存在しています。ではなぜ Ansible が出てきたのでしょうか。

以下の表で比べてみます。

表 1.1: Chef や Puppet との違い

	対応に必要な要素	記述方式	UI
Chef	エージェントとサーバー	Ruby	付属
Puppet	エージェントとサーバー	独自 DSL	付属
Ansible	ssh のみ	YAML	有償 (Ansible Tower)

特に大きな違いは既存システムへの変更が必要かどうかです。Ansible は ssh が使えればすぐに始められます。一方、他のツールでは管理用サーバーと、各対象ホストに管理サーバーから指示を受け取るエージェントをインストールする必要があり、最初に使用し始めるまでの敷居が高いです。そのため、Chef で言えば chef-solo を使うなどして、管理用ホストを使わない方式も広く使われています。Ansible は当初から ssh のみで管理するというポリシーが貫かれていますので、使いやすくなっています。

もちろんこれには利点と欠点があり、Chef サーバーはノードの検索や変数の管理などが得意です。大規模になり複雑化すればするほど Ansible ではない管理ツールの方が便利になってきます。

Ansible でも dynamic inventory やグルーピング、対象を正規表現で指定するなどができますので、複雑な構成にも対応できます。しかし、Chef サーバーの機能や UI にはどうしても見劣りがしてしまいます。

また、Amazon AWS CloudFormation は Chef のレシピを使うことができます。Amazon EC2 を利用している方は Chef の方が統合されて使い勝手が良いです。

Ansible だけが正解ではありませんので、よく吟味して選択する事が必要です。

以下の場合には Ansible を選ぶと良いと思います。

- 管轄などにより、管理対象にエージェントなどをインストール出来ない場合
- Shell Script などの既存資産を使い回したい場合
- Ruby を書けない、あるいは書きたくない場合
- 手軽に構成管理をしたい場合

1.4 Ansible は “Better Shell Script”

筆者は Ansible を **Better Shell Script** だと感じています。Chef や Puppet は「構成管理」として、すべてのホストはこういう状態に収まるべき、ということを記述します。つまり、最終的な状態は一つになります。

しかし、Ansible はやりたいことの数だけ playbook を記述しても構いません。インストール用、構築用、デプロイ用、障害調査用、従来 shell script を書いていたように、それぞれに対して playbook を書いても構いません。もちろん、playbook を一つだけにしてホストの状態を記述しても構いません。

Ansible は shell script のように、やりたいことをそのまま記述していける手軽さこそが特徴だと考えています。

本章では Ansible の概要について述べました。では、次章から実際に Ansible を使い始めたいと思います。

第2章　Ansibleを使ってみよう

この章では、

1. Ansible 本体のインストール
2. 実行先ホストを設定する inventory ファイル
3. 実行コマンドを示す module

の 3 つについて説明します。

2.1 インストール

これから

- Ansible を実行する**管理ホスト**
- Ansible によって管理される**対象ホスト**

のインストールについてそれぞれについて順に説明していきます。まずは管理ホスト側のインストール方法です。

2.1.1 前提条件

Ansible そのものを実行するには Python 2.6 以上が必要です。現在ほぼすべての OS、ディストリビューションがこの条件をインストール時から満たしていますが、古いバージョンをお使いの方は満たしていない場合があります。その場合は、別途 Python をインストールしてください。

ヒント: 残念ながら Ansible は Python3 に対応していません。また、対応する予定も今のところありません。

なお、本書では **Ansible 1.6.6** を前提としています。

2.1.2 パッケージ管理ツールでのインストール

Ansible は多くのシステム標準のパッケージ管理ツールに含まれています。お使いの OS やディストリビューションに応じて適切な箇所をご覧ください。

Ansible をインストールすると、以下のコマンドが使えるようになります。

ansible ansible のモジュールを実行する

ansible-playbook モジュールをまとめた Playbook を実行する

ansible-vault ファイルの暗号化を行う

ansible-galaxy 全世界のユーザーが投稿している playbook を使う

このうち、主に使うのは `ansible-playbook` コマンドです。その他のコマンドについては必要になった時に説明します。

yum 経由でインストールする場合

EPEL に RPM が入っていますので、EPEL を設定した後に以下のコマンドを実行します。

```
$ sudo yum install ansible
```

あるいは、リポジトリから自分で RPM を作ることも出来ます。

```
$ git clone git://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ~/rpmbuild/ansible-*.noarch.rpm
```

実行には `rpm-build` と `python2-devel` がインストールされている必要があります。

apt でインストールする場合

PPA を設定し通常のようにインストールします。

```
$ sudo apt-add-repository ppa:rquillo/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

リポジトリから deb パッケージを自分で作ることも出来ます。

```
$ make deb
```

pkg でインストールする場合

FreeBSD で使用されているパッケージ管理ツールを使ってインストールする場合です。

```
$ sudo pkg install ansible
```

Homebrew でインストール場合

Mac OS X の Homebrew でのインストールは以下のとおりです。

```
$ brew update
$ brew install ansible
```

pip でインストールする場合

Ansible は Python で作成されており、Python ソフトウェアは pip というツールでインストールします。

```
$ pip install ansible
```

ヒント： OS X Mavericks では以下のようにしないとエラーが出るという報告があります

```
$ sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments \
           pip install ansible
```

2.1.3 対象ホストへのインストール

Ansible では対象ホストにも Python がインストールされている必要があります。

管理ホストは 2.6 以上が必要ですが、対象ホストには 2.4 以上あれば動作します。ただし、2.4 の場合、以下のように依存ライブラリを一つ入れる必要があります。

```
# CentOS など
$ sudo yum install python-simplejson

# Debian など
$ sudo apt-get install python-simplejson

# FreeBSD
$ sudo pkg install py27-simplejson
```

すでに Ansible を使われている方なら、raw モジュールを使って Ansible でインストールすることもできます。この場合、複数台のホストに対して同時にインストールできます。

```
$ ansible all -m raw -a 'yum -y install python-simplejson'
```

これで Ansible を使う準備が出来ました。

Ansible には三種類の設定ファイルがあります。

1. inventory ファイル

   実行対象のホストを設定するファイル

2. playbook ファイル

   task と呼ばれる一つ一つの動作をまとめて記述したファイル

3. ansible.cfg ファイル

   Ansible 自体の設定を記述した設定ファイル

Ansible を使うために最低限必要なファイルは inventory ファイルです。まずは inventory ファイルについて説明します。

2.2 inventory ファイル

inventory ファイルは ini 形式 (正確には ini 形式を拡張した形式) で記述します。例として以下のような内容のファイルを作成します。`/etc/ansible/hosts` に作成すると、コマンドラインオプションで指定しなくても Ansible が自動的に読み込んでくれます。

```
mail.example.com

[web]
web01.example.com
web02.example.com
web03.example.com

[db]
db01.example.com
db02.example.com
```

`[webservers]` や `[db]` は**グループ**を示します。また、`web01`、`db01` などは**ホスト**を示します。つまり、この例では web というグループに web01.example.com、 web02.example.com、web03.example.com という 3 つのホストが含まれていることを示します。また、どこのグループにも属していない mail.example.com というホストがあることも示しています。

このように設定しておくと ansible コマンドを実行できます (事前に ssh で各ホストに入れることを確認する必要があります)。

```
$ ansible all -m command -a "uptime"

mail.example.com | success | rc=0 >>
23:51:40 up 33 days,  1:55,  2 users,  load average: 1.17, 1.21, 1.20

web01.example.com | success | rc=0 >>
23:51:41  up 29 days, 12:22, 2 users, load averages: 0.36, 0.28, 0.18
(以下略)
```

この例では全サーバーで “uptime” コマンドを実行して、その結果が表示されています。

次にグループを指定してみます。以下のように “db” を指定して実行してみてください。

```
$ ansible db -m command -a "uptime"

db01.example.com | success | rc=0 >>
23:55:11 up 33 days,  1:55,  2 users, load averages: 0.41, 0.30, 0.20

db01.example.com | success | rc=0 >>
23:55:11  up 29 days, 12:22, 2 users, load averages: 1.11, 1.07, 1.09
```

今度は db グループに属しているホストだけで実行されました。

もちろん、ホストを指定しても構いません。

```
$ ansible web02.example.com -m command -a "uptime"

web02.example.com | success | rc=0 >>
23:58:32 up 29 days,  1:55,  2 users, load averages: 0.13, 0.20, 0.17
```

コラム: 複雑なホスト指定

ホスト名指定には完全一致以外に、正規表現や複雑な指定が可能です。

www*.example.com www のホストすべて

www[01:50].example.com www01 から www50 まで

web:db web か db (or 指定)

web!ng web のうち、ng 以外 (not 指定)

web:&prod web かつ prod (and 指定)

web:!{{excluded}}:&{{required}} 後述する変数を使い、かつ、複数指定を組み合わせる

2.2.1 ホストの設定

各ホストに対してさまざまな設定出来ます。次のように書くと、

```
web01.example.com ansible_ssh_port=2222 ansible_ssh_host=192.168.1.50
```

web01 に対しては 192.168.1.50 のポート 2222 で繋ぎに行く、という設定です。

他にも一例として以下のような設定ができます。

ansible_ssh_user ssh で接続するユーザー名

ansible_ssh_private_key_file 秘密鍵鍵ファイル

ansible_connection ssh 以外の接続方法を指定

ヒント: 対象ホストが Linux 以外の場合の設定

Ansible は対象ホストの Python が “/usr/bin/python” であることを前提としています。FreeBSD などは “/usr/local/bin/python” にあることが多いため、以下のように `ansible_python_interpreter` を設定する必要があります。

```
freebsd_host  ansible_python_interpreter=/usr/local/bin/python
```

2.2.2 グループの設定

ansible_ssh_port などで各ホストごとに設定をしていくと、ホストが多いと大変になります。せっかくグループ分けをしているのですから、グループ単位で設定をしたくなります。

その場合、以下のように`:vars` を書くとグループでの設定をまとめて記載できます。

```
[web]
web01.example.com

[web:vars]
ansible_ssh_port=2222
ansible_ssh_user=admin
```

この例では web グループに属しているすべてのホストの設定をしています。

2.2.3 グループの入れ子

グループは `children` を付けることで入れ子構造にすることができます。

```
[eggservice:children]
eggweb
eggdb

[eggweb]
web01.example.com
web02.example.com

[eggdb]
db01.example.com
db02.example.com
```

上記例では

- eggservice
  - eggweb
    - web01
    - web02

  - eggdb
    * db01
    * db02

という構成になります。上位のグループを指定すると含まれているホストすべてが対象になります。

2.3 モジュール (module)

inventory ファイルで実行対象のホストを指定した後は、そのホストの上でなにを実行するかを指定します。そのホストの上で実行するものを**モジュール (module)** と呼びます。

例として setup モジュールを実行してみます。setup モジュールは実行したホストの IP アドレスなどの状態を得るモジュールです。setup モジュールは通常は自動的に呼ばれますが、明示的に指定することもできます。

inventory ファイルを用意した状態で、以下のように実行します。

```
$ ansible all -m setup
```

すると、以下のように表示されます。これは省略しており、実際にはかなりの量の情報が表示されます。

```
web02.example.com | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.122.1"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fe99:ebc3",
        ],
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "12/01/2006",
        "ansible_bios_version": "VirtualBox",
```

表示中の `ansible_all_ipv4_addresses` を見ると、実行対象ホストの IP アドレスであることが分かります。

このように

- inventory ファイルで実行対象を指定し、
- module を実行する

ことで望みの動作を行うことができます。

次の章からはもっと実践的な `ansible-playbook` について解説します。

第3章　playbookを作ってみよう

全章で説明したansibleコマンドは一つのモジュールを実行するだけでした。

この章では、複数のモジュールを実行する以下の方法を説明します。

- **task**と呼ばれる、モジュールとその引数をまとめた設定
- taskを複数集めて一連の動作を記す**playbook**の書き方
- playbookを実行する**ansible-playbook**コマンド

3.1 YAMLの文法

playbookはYAMLという形式で記述します。YAMLは入れ子になった構造や、シーケンス(配列)・マッピング(辞書)のデータ構造を読みやすく記述できます。

コラム: なぜYAML?

プログラム言語ではなくYAML形式を使うことで、書き方が制限され、柔軟な制御はできなくなります。しかし、そのために書き方が統一され、また、プログラマ以外にも書きやすくなります。

また、playbookファイルはYAMLなのにinventoryファイルはini形式で書かれていることに違和感を持たれるかもしれません。Ansibleの開発者によると、inventoryファイルには複雑な階層構造が必要ではないため、より単純なini形式を選択したとのことです。

3.1.1 基本的な書き方

まずYAMLの基本的な書き方を説明します。より詳細はプログラマーのためのYAML入門(初級編)などを参考にしてください。

シーケンス

シーケンスとはリストや配列などと呼ばれているデータ構造です。行頭に-をつけるとシーケンスとなります。

```
- A
- B
- C
```

また、以下のようにすると一行に書くことが出来ます。

```
[A, B, C]
```

階層構造

YAML ではインデントでデータの階層構造を表します。

```
- A
- B
  - C
  - D
- E
```

は、以下のような階層構造となります。

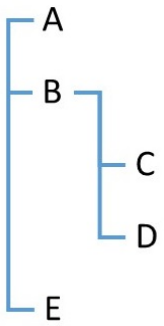

図 3.1: YAML での階層構造表現

インデントする文字数は決められていませんが、2 文字の場合が多いです。また、タブ文字 (ハードタブ) は使えません。

3.1.2 マッピング

連想配列や辞書、マップとも呼ばれているマッピングは以下のように:の後に半角スペースを一つ以上入れて書きます。

```
A: aaa
B: bbb
C: ccc
```

マッピングもシーケンスと同じく半角スペースでインデントすることで階層構造を表せます。

```
A: aaa
B:
  B1: bbb1
  B2: bbb2
C: ccc
```

3.1.3 シーケンスとマッピングの組み合わせ

シーケンスとマッピングは組み合わせて階層構造に出来ます。

```
- hosts: web
  sudo: yes
  vars:
    - required: ['docker', 'pyyaml']  # シーケンスを指定
  tasks:
  - name: docker image を作成
    docker_image: name="my/app" state=present
```

3.1.4 間違えやすいところ

- マッピングの場合:のあとには空白が必要です。ダメな例:

```
    value:somevalue
```

- マッピングで次のように書くことは出来ません。

```
    a: aaa
      a1: aaa1
      a2: aaa2
```

    a: aaa の次の行にインデントをつけてマッピングの要素は書けません。書く場合にはこのようになります。

```
    a:
      a1: aaa1
      a2: aaa2
```

3.1.5 コメント

#をつけることでコメントを記述できます。

```
# ここにコメントを書けます
a:  # ここもコメントです
  a1: aaa1
```

3.1.6 複数行にまたがって書く

たくさんの引数を渡したい時があります。その場合、改行すると見やすくなります。改行後にはインデントが必要になります。

```
- glance_image:
     login_username=admin
     login_password=passme
     login_tenant_name=admin
```

```
    name=cirros
    state=present
```

コラム: YAML の確認

YAML の形式がおかしいと、Ansible でコマンドを実行したときに指摘してくれます。

```
ERROR: Syntax Error while loading YAML script, test.yml
Note: The error may actually appear before this position: line 5, column 1

  roles:testrole

  ^
```

行数が表示されるので分かりやすいと思います。

3.2 playbook を書いてみる

YAML の書き方を一通り学んだところで、実際に playbook を書いてみましょう。

この章では以下の inventory ファイルがあるとします。この inventory ファイルを**/etc/ansible/hosts** に保存します。

```
[web]
web01.example.com
web02.example.com
```

ではこの inventory ファイルにある web グループに newuser というユーザーを追加する playbook を書いてみましょう。UTF-8 形式で以下の内容を記述します。

```
---
- hosts: all                # 対象を指定。all は全ホスト
  sudo: yes                 # sudo を行う
  remote_user: hawk         # 実行ユーザー名
  vars:                     # 変数指定
    username: newuser
  tasks:                    # 実行する task の指定を開始
    - name: ユーザーを追加  # task の名前
      user: name={{ username }} group=admin shell=/bin/bash
```

これを **web.yml** という名前で保存します。インデントに注意してください。

保存しましたら、以下のように実行します。

```
$ ansible-playbook web.yml
```

そうすると、以下のように表示されます。

```
PLAY [all] ********************************************

GATHERING FACTS ***************************************
ok: [web01.example.com]
```

```
ok: [web02.example.com]

TASK: [ユーザーを追加] *****************************
changed: [web01.example.com]
changed: [web01.example.com]

PLAY RECAP ********************************************
all                        : ok=2    changed=1    unreachable=0    failed=0
```

changed が黄色く表示されると思います。これで、二台のホストの両方で newuser というユーザーが追加されました。

もう一度実行してみましょう。

```
PLAY [all] ********************************************

GATHERING FACTS ***************************************
ok: [web01.example.com]
ok: [web02.example.com]

TASK: [ユーザーを追加] *****************************
ok: [web01.example.com]
ok: [web01.example.com]

PLAY RECAP ********************************************
all                        : ok=2    changed=0    unreachable=0    failed=0
```

今度は ok となりました。つまり、もう既にユーザーが追加されているため、二回目は変化が起きることなく、終了したことが分かります。この playbook は何度実行しても **newuser というユーザがいる状態**になります。これは、つまり**べき等性がある**ということになります。

3.3 playbook の解説

では、さきほどの playbook を解説します。playbook は以下の 3 つのセクションに分けられます。

target セクション 実行対象の設定

vars セクション 変数の設定

tasks セクション 実行する task の設定

これからこれらのセクションについて説明します。

3.3.1 target セクション

実行する対象ホストを指定する部分を target セクションと呼びます。先ほどの例では以下の部分です。

```
---
- hosts: all                # 対象を指定。all は全ホスト
  remote_user: hawk         # 実行ユーザー名
  sudo: yes                 # sudo を行う
```

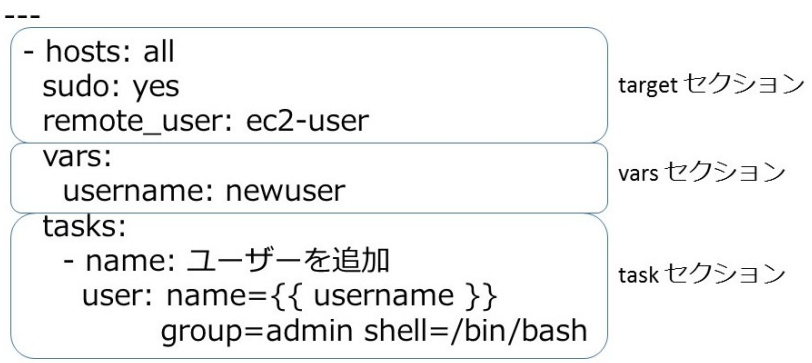

図 3.2: playbook のセクション構造

hosts で対象を指定しました。ここでは all とすべてのホストを指定しています。

先程の例ではインストールに root 権限が必要なため、**sudo** を実行するように指定しています。また、**remote_user** でログインユーザーを指定しています。ansible-playbook を実行するユーザーと同じであれば指定の必要はありません。

これにより、

- コマンドラインで指定したすべてのホストに対して
- ユーザー hawk でログインし
- sudo を使って実行する

ということが指定されていることが分かります。

playbook の実行には最低限 `hosts` が必要です。

3.3.2 vars セクション

`vars` セクションでは使用する変数を設定します。具体的には以下の部分です。

```
vars:                       # 変数指定
  username: newuser
```

ここでは `username` という変数に `newuser` という値を設定しています。これを task の中で使用しています。

```
user: name={{ username }} group=admin shell=/bin/bash
```

変数を使用するには**{{ username }}**というように`{{ }}`で囲みます。

ヒント: 従来は`${tempdir.stdout}`という形式が使えましたが、1.6 から使えなくなりました。古い記述のブログ等がありますので気をつけてください。

---

vars は複数指定可能ですし、入れ子構造も可能です。入れ子構造の子の内容を指定する場合は. で区切ります。また、後述のループで使うためにシーケンスも指定できます。

```
vars:     # 変数指定
  username: newuser
  group: admin
  shell:
    bash: /bin/bash
    zsh: /bin/zsh
tasks:
  - user: name={{ username }} group={{ group }} shell={{ shell.zsh }}
```

なお、変数を文の先頭に配置する場合、""で囲む必要があります。さもないと、以下のようなエラーが表示されます。

```
ERROR: Syntax Error while loading YAML script, test.yml
Note: The error may actually appear before this position: line 7, column 28

  tasks:
    - command: {{ shell }} wrong.sh
                         ^
```

この問題を修正するには、以下のようにします。

```
tasks:
  - command: "{{ shell }} wrong.sh"
```

エラーメッセージの中に問題がある箇所と例が出ますので分かりやすいかと思います。

ファイルからの読み込み

vars は playbook で指定する他に、vars_files を使って別ファイルから読み込むことも出来ます。

```
---
- hosts: all
  vars_files:
    vars/user.yml
    vars/apache_settings.yml
  tasks:
    - name: 以降は省略します
```

この例では playbook が置かれている場所からの相対パスで vars/names.yml というファイルと vars/apache_settings.yml というファイルから変数を読み込んでいます。

それぞれのファイルは以下のような形式になっています。

```
---
username: alice
group: admin
```

変数をファイルに分割することで、管理しやすくなります。

3.3.3 tasks セクション

tasks 以下でその playbook で実行する task を指定します。task は、一つのモジュールとそれに対する引数で構成される、一つの動作を行う最小の単位です。

```
tasks:                          # 実行する task の指定を開始
  - name: ユーザーを追加   # task の名前
    user: name={{ username }} group=admin shell=/bin/bash
```

この例では **name** という task に対する名前と **user** というモジュールおよび user モジュールに対する引数からなる一つの task が記述されています。

それぞれは以下の意味となります。

name その task の名前です。必須ではありませんが、つけておくとあとから分かりやすくなります。

user 実行するモジュールです。name、group、shell は user モジュールの引数です。引数はモジュールごとに違います。

一つの playbook 内で複数の task を実行できます。インデントは揃える必要があります。

```
tasks:
  - name: ユーザーを追加
    user: name={{ username }} group=admin shell=/bin/bash
  - name: authorized_key を追加
    authorized_key: user={{ username }} key=keys/hawk.pub
```

この例では、ユーザーを追加した後に authorized_key に hawk.pub を追加しています。Ansible では記述した順番に task が実行されます。また、一つの playbook 内に書ける task の数に制限はありません。

3.4 task

ここまで基本的な playbook の書き方を説明してきました。この節では、task についてもう少し詳しく述べます。

3.4.1 task の実行順序

複数のホストに対して playbook を実行すると、その複数のホストすべてが task が終わるまで待ってから、次の task を実行します。

例えば task A と B が定義されている場合、

1. 全ホストで task A を実行
2. 全ホストで task A が終わるまで待つ
3. 全ホストで task B を実行

となります。

そのため、極端に動作が遅いホストがあると、playbook 全体の実行速度はその遅いホストに合わさることになります。

また、あるホストだけがなにかしらの原因で失敗した場合、そのホスト以外は次の task から実行します。

3.4.2 task ごとの設定

先程の例では、一番上位の階層に sudo などを設定しました。task ごとに sudo や他の設定もできます。この task ごとに設定した値が優先されます。

```
---
- hosts: all
  tasks:
    - name: apache をインストール
      yum: name=httpd state=latest
      sudo: yes
      sudo_user: admin
```

新しく出てきた `sudo_user` は task を実行するユーザーを sudo で変更する場合に設定します。

これにより、基本的には sudo は行わないが、ある task だけは sudo を行う、などが設定できます。

3.5 handler

設定ファイルを書き換えた場合など、再起動が必要な場合があります。その場合、複数の設定ファイルを書き換えた後に再起動したくなります。

このような場合に **handler** を使います。handler は書き方は通常の task と同じですが、`handlers` 以下に書き、**notify** が指定された task が実行されると、指定された name が付いている handler が実行されます。

```
---
- hosts: all
  tasks:
    - name: 設定ファイルをコピー
      copy: src=httpd.conf dest=/etc/httpd/conf/httpd.conf
      notify: apache の再起動
    - name: 設定ファイルの owner を設定
      file: dest=/etc/httpd/conf/httpd.conf owner=httpd state=file
      notify: apache の再起動
  handlers:
    - name: apache の再起動
      service: name=httpd state=restarted
```

この例では設定ファイルをコピーと設定ファイルの `owner` を設定の両方で `apache` の再起動という handler を指定しています。これにより、指定したタスクで変化があった場合に、apache の再起動が**一度だけ**実行されます。複数回 notify が呼ばれていますが、実行されるのは一度だけです。

また、notify が呼ばれるのは変化があった場合だけですので、この例では設定ファイルがコピー済みで owner も設定されていた場合は再起動されません。

handler の書き方は通常の task と同じです。ただし、notify は name で指名しますので、handler は name が必須となります。

また、notify を以下のように書くことで複数の handler を起動できます。

```
- name: 設定ファイルの owner を設定
  file: dest=/etc/nginx/nginx.conf owner=nginx state=file
  notify:
    - nginx の再起動
    - 設定の通知
```

複数の handler が呼ばれた場合、書かれた順番にそれぞれ一度だけ実行されます。

3.6 よく使うモジュール

Ansible は 200 以上のモジュールが最初から使えるようになっており、実に多くの用途に対応できます。しかし、一般的にはそこまで多くの種類のモジュールを使う必要はありません。ここではよく使うモジュールについて簡単に説明します。

ヒント: ここで解説する引数はごく一部です。実際には多くの有用な引数がありますので、 詳しくは公式のドキュメントをご覧ください。

3.6.1 script

管理ホストに置かれているスクリプトを対象ホストに転送し、対象ホスト上でそのスクリプトを実行します。既存の shell script をそのまま流用できるので、移行に便利です。

```
tasks:
  - name: command.sh を実行する
    script: command.sh
  - name: files/other.sh を実行する。/tmp/done.txt があれば実行しない
    script: files/other.sh  creates=/tmp/done.txt
```

この command.sh というファイルは、playbook と同じ場所からの相対パス指定、あるいは絶対パスでの指定となります。

また、creates という引数を指定すると、そのファイルがあった場合には実行せずにスキップします。スクリプト内部でこのファイルを最後に作るようにすれば、何度実行しても良いためべき等性を備えることも出来ます。

3.6.2 command

command は任意のコマンドを対象ホストで実行します。

```
tasks:
  - name: シャットダウンします
    command: /sbin/shutdown -t now
  - name: something.sh を実行します
    command: /opt/bin/something.sh creates=/var/db/database
  - name: /home/admin/bin に移動し、exec.sh を実行します
    command: exec.sh chdir=/home/admin/bin
```

creates 引数を設定すると、そのファイルがある場合は実行されません。また、chdir 引数を設定すると、そのディレクトリに移動した後にコマンドを実行します。

なお、command モジュールでは>(リダイレクト) や | (パイプ) などの shell の機能は使えません。その場合は、shell モジュールを使用してください。

3.6.3 shell

shell モジュールは任意のコマンドを実行します。shell モジュールでは command モジュールとは異なり、>や | が使えます。

```
tasks:
  - name: target という行を抜き出します
    shell: cat /tmp/list | grep "target" >> /tmp/list.out
```

creates や chdir など、command モジュールで使用できる引数も使えます。

3.6.4 file

ファイルやディレクトリの作成、owner や group の変更、symbolic link の作成などを行います。

```
tasks:
  - name: /etc/example.conf を設定
    file: path=/etc/example.conf owner=admin group=admin mode=0644
  - name: シンボリックリンクを作成
    file: src=/etc/link/to dest=/etc/symlink state=link
  - name: ディレクトリ作成
    file: path=/etc/deep/dir/ex state=directory
  - name: /home/admin 以下すべてのファイルの owner を設定
    file: path=/home/admin owner=admin recurse=yes
```

すでに指定した内容になっていればスキップされます。

state には以下の種類があります。

file ファイルを設定します。もしファイルが存在しない場合エラーになります

link シンボリックリンクを作成します

directory ディレクトリを指定した場合、深い階層でも途中のディレクトリをすべて作成します。また、**recurse=yes** を指定すると再帰的に指定したディレクトリ全てを変更できます

hard ハードリンクを作成します

touch ファイルを設定します。ファイルが存在しない場合、作成します

absent ファイルやディレクトリを削除します。シンボリックリンクやハードリンクの場合、そのリンクを削除します

3.6.5 copy

管理ホストに置かれているファイルを対象ホストにコピーします。

```
tasks:
  - name: admin.conf をコピー
    copy: src=admin.conf dest=/etc/admin.conf owner=admin mode=0644
  - name: バックアップを作成
    copy: src=../admin.conf dest=/etc/admin.conf backup=yes
```

src となるファイルは、playbook からの相対パスあるいは絶対パスで指定します。

3.6.6 fetch

copy は管理ホストから対象ホストにファイルを転送していました。`fetch` は逆に対象ホストから管理ホスト側にファイルを転送します。ログの収集などに使うと便利です。

```
tasks
  - name: dmesg を取得する
    fetch: src=/var/log/dmesg dest=/tmp/fetched/dmesg
  - name: ログファイルを取得。ないならばエラーになる
    fetch: src=/var/log/app.log dest=applog fail_on_missing=yes
```

取得したファイルは、自動的にホスト名がついたディレクトリと対象ホストにおける絶対パスが結合されたパスにディレクトリが作成されます。例えば、

```
fetch: src=/var/log/app.log dest=/var/log/xbackup
```

のように`/var/log/app.log` ファイルを `example.com` ホストから取得し、dest が `/var/log/backup` というディレクトリ名という場合、

```
/var/log/backup/example.com/var/log/app.log
```

というディレクトリおよびファイル名が playbook ファイルが置かれている場所以下に保存されます。

自動的にホストごとに分類されるので、書き潰したりということがありません。ホストごとの分類が必要ない場合は `flat=yes` を付けるとホストごとの分類もディレクトリの展開もされず、ファイルがそのまま置かれます。また、対象となるホストが一つだけの場合はホスト毎の分類はされません。

3.6.7 template

Python でよく使われている Jinja2 というテンプレート言語を使ってファイルのコンテンツを記述しておいたテンプレートに対して変数を埋め込み、対象ホストでファイルを生成します。

例として、以下のテンプレートを `template.j2` というファイル名で作成します。

```
user {{ user }};
worker_processes  {{ worker_process_num }};

error_log  {{ error_log_path }};
pid        /var/run/nginx.pid;
```

その後、vars を設定し、template モジュールを使います。

```
vars:
  user: www-data
  worker_process_num: 1
  error_log_path: /var/log/nginx/error.log
tasks:
  - name: nginx.conf を設定します
    template: src=template.j2 dest=/etc/nginx/nginx.conf
```

結果として、以下のファイルが /etc/nginx/nginx.conf に作成されます。

```
user www-data;
worker_processes  1;

error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;
```

Jinja2 は繰り返しや条件式など高度な機能が搭載されており、Python における標準的なテンプレート言語の一つです。

3.6.8 get_url

指定した URL からファイルをダウンロードします。

```
tasks:
  - name: file.conf をダウンロードします
    get_url: url=http://example.com/path/file.conf
             dest=/etc/file.conf owner=root
```

`dest` で指定した場所にファイルをダウンロードします。この時、dest がファイルかディレクトリかで挙動が変わります。

dest がファイルの場合 指定したファイル名で作成されます。ファイルが存在していた場合、ダウンロードされません。

dest がディレクトリの場合 web サーバーが指定したファイル名で dest のディレクトリ以下に作成されます。ファイルは常にダウンロードされます。

べき等性を考えると、dest をファイルに指定する方が良いと考えられます。また、ファイルがあってもダウンロードしたいという場合は、`force=yes` をつけると dest がファイルの場合でも常にダウンロードされます。

3.6.9 lineinfile

ファイル中の任意の一行を設定あるいは削除します。その行は正規表現で指定します。

```
tasks:
  - name: regexp がないので最終行の後に一行付け加える
    lineinfile:
      dest=/etc/snmp/snmpd.conf
      line='dontLogTCPWrappersConnects yes'
  - name: regexp で指定した行を置き換える
    lineinfile:
      dest=/etc/selinux/config
      regexp=^SELINUX=
      line=SELINUX=disabled
```

regexp がない、あるいはマッチしない場合は最終行の後に一行付け加えられます。

一行付け加えるだけや、一行置き換えるだけであれば lineinfile は簡単に使えるので便利です。ただし、正規表現が複雑になると分かりにくくなります。

複数行にマッチする置換を一回の lineinfile で実行しても、最後の一つしか置換されません。複数行をまとめて置換したい場合は replace モジュールを使います。

```
tasks:
  - name: /etc/hosts の*.example.com を*.new.example.com に置き換える
    replace:
      dest=/etc/hosts
      regexp='^(.*).example.com$'
      replace='\1.new.example.com'
      backup=yes
```

3.6.10 unarchive

管理ホストにある tar.gz や zip ファイルを対象ホストに転送し、展開します。拡張子などで判断して unzip や tar コマンドを使い分けます。

```
tasks:
  - name: something.tgz を送り、/var/lib/some 以下に展開します
    unarchive: src=something.tgz dest=/var/lib/some
```

command モジュールなどと同じく、`creates` 引数を指定すると指定したファイルがあれば実行しません。

3.6.11 yum

yum コマンドを使ってパッケージをインストールします。

```
tasks:
  - name: 最新の httpd をインストールした状態にします
    yum: name=httpd state=latest
  - name: httpd をインストールしていない状態にします
    yum: name=httpd state=removed
  - name: repo を指定します
    yum: name=httpd enablerepo=testing  state=installed

  - name: httpd と python-dev をインストールした状態にします
    yum: name={{ item }} state=installed
    with_items:
      - httpd
      - python-dev
```

最後の例は with_items を使ったループの指定です。これについては次章でご説明します。

なお、yum モジュールでは、name は以下のパラメータを指定できます。

バージョン付きのパッケージ名 something-1.2.2

rpm の URL http://example.com/something.rpm

rpm のファイル名 /tmp/something.rpm

グループ名 “@Development tools”

正規表現 python*

Ansible には yum だけではなく apt や pkg、pacman など、各種 OS・ディストリビューションのパッケージ管理ツールがあり、ほぼ同じ書き方で使えます。その他、キャッシュの更新など、そのツール独自の機能も指定できます。詳しくは Ansible のドキュメントをご覧ください。

3.6.12 service

service コマンドを使って、サービスを管理します。

```
tasks:
  - name: httpd を起動した状態にします
    service: name=httpd state=started
  - name: httpd を停止した状態にします
    service: name=httpd state=stopped
  - name: httpd を再起動します。stop と start の間に 10 秒待ちます
    service: name=httpd state=restarted sleep=10
  - name: eth0 という引数を与えて network サービスを再起動します
    service: name=network state=restarted args=eth0
  - name: httpd をホスト起動時に自動的に起動するように設定します
    service: name=httpd enabled=yes
```

sleep という引数を設定すると、stop と start の間に指定した秒数だけ時間を待ちます。args という引数を設定すると、サービスに引数を渡せます。

enabled を yes にすると、ホスト起動時に自動的に指定したサービスが起動するようになります。state か enabled のどちらかは必ず指定する必要があります。

3.6.13 user

ユーザーを追加、あるいは削除します。

```
tasks:
  - name: ユーザー hawk を追加します
    user: name=hawk comment="South Reach" uid=1040 group=admin
  - name: zsh を使うユーザー ogion を追加します
    user: name=ogion
          shell=/bin/zsh groups=admins,developers
          password=$1$SomeSalt$Drh7s/vUcl5XnIZ/Neglz1
```

password を指定することでパスワードを設定できます。このパスワードはハッシュ化されている必要があります。

mkpasswd が使用可能であれば以下のコマンドで実行できます。

```
mkpasswd --method=SHA-512
```

あるいは、passlib を python でインストールすると以下のコマンドで得られます。

```
# passlib をインストール
pip install passlib
# <your password>部分を書き換え、ハッシュを取得
python -c "from passlib.hash import sha512_crypt; \
           print sha512_crypt.encrypt('<your password>')"
```

3.6.14 authorized_key

SSH の authorized key を追加、あるいは削除します。初期設定時などに役に立ちます。

```
tasks:
  - name: ユーザー ogion に対して、管理ホストにある鍵を追加する
    authorized_key: user=ogion
                    key="{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
```

この例では、次の章で説明する lookup を使用し、追加する鍵を指定しています。前述の user モジュールと組み合わせることで、root でログインして通常作業用ユーザーと必要な鍵を追加し、以降はそのユーザーで実行する、ということができます。

3.6.15 wait_for

このモジュールは指定した時間を待つのではなく、port が開くまで、あるいはファイルが作られるまで待つ、という動作を行います。これは、ssh が使えるようになるまでや、インストールログを監視し “complete” が表示されるまでなどに使えます。

単純に指定した時間だけ待ちたい場合は、 command モジュールで sleep を指定するのが簡単です。

```
tasks:
  - name: 8000 番ポートが開くまで 10 秒間隔でチェックして
    wait_for: port=8000 delay=10
  - name: /var/log/foo.log が作成されるまで
    wait_for: path=/var/log/foo.log
  - name: lock ファイルが消えるまで
    wait_for: path=/var/lock/file.lock state=absent
  - name: install.log 中に "completed" という文字が現れるまで
    wait_for: path=/var/foo/install.log search_regex=completed
```

3.6.16 irc

irc モジュールは IRC に任意のメッセージを通知します。

次の例では、一連の task の最後に指定することで、playbook 終了時に IRC に通知します。

```
tasks:
  - name: playbook 終了時に IRC に通知する
    irc:
      server=irc.example.net
      channel="#t1"
      msg="playbook が終了しました"
```

Ansible にはその他にメールなど、外部へと通知するモジュールがあります。

3.6.17 debug

debug モジュールは任意の内容を表示します。変数の確認などに使います。

```
vars:
  user:
    name: tomato
  host: pizza
tasks:
  - name: 変数を表示します
    debug: msg="create {{ user.name }} user on {{ host }}"
```

これを実行すると、以下のように表示されます。

```
TASK: [変数を表示します] ***********************************
ok: [54.178.145.110] => {
   "msg": "create tomato user on pizza"
}
```

残念ながら msg を日本語にすると文字化けします。

本章では基礎的な playbook の作成方法について述べました。ここまで説明したことで以下のことができるようになります。

- 対象となるホストを指定する
- 変数を宣言し、利用する
- 複数の task を実行する
- よく使われるモジュールを利用する

これだけで多くの場面で Ansible を使えるようになるかと思います。

次章では、繰り返しや条件分岐など、もう少し複雑なことをしたい場合にどうすればよいのかについて述べていきます。

第4章　複雑なplaybookを作ってみよう

ここまでで playbook の書き方を一通り説明してきました。本章ではより複雑で実践的な使い方を説明します。

4.1 繰り返し – with_items

一つのモジュールを何度も繰り返したい場合、`with_items` を使います。当てはめたいところには**{{ item }}**のように item という変数を使います。

```
vars:
  softwares:
    - httpd
    - python-dev
tasks:
 - name: /opt/foo 以下に bin, conf, log ディレクトリ作成
   file: path=/opt/foo/{{ item }} state=directory
   with_items:
     - bin
     - conf
     - log
 - name: yum でインストール
   yum: name={{ item }} state=installed
   with_items: softwares
```

with_items にシーケンスを指定すると、そのシーケンスの数だけ指定したモジュールを繰り返します。二番目の例のように、with_items に vars で宣言した変数を与えることも出来ます。

ヒント: yum モジュールなどに with_items を使うと yum コマンド実行を一回にまとめてくれます。

基本的なループには with_items で対応できます。しかし、もっと複雑なループをしたい場合があります。残念ながら YAML 形式はプログラミング言語ほどの表現力を持たないため、柔軟なループは表現できません。とはいえ、ある程度の柔軟性を可能にする、ループをこれからいくつか紹介します。

4.1.1 入れ子のループ

`with_nested` を使うと、入れ子構造を表現できます。

```
tasks:
 - name: PostgreSQL の各 DB にユーザーを設定する
   postgresql_user: db={{ item[0] }} name={{ item[1] }}
   with_nested:
     - [ 'customerdb', 'somedb', 'productdb']
     - [ 'ged', 'tenar']
```

この例では、ユーザー名のシーケンスと対象 DB のシーケンスをまとめたシーケンスを利用しています。これにより、すべての DB にユーザーが作られます。

ヒント: with_nested による入れ子構造は、一段階だけしかできません。

4.1.2 マッピングをループ

with_dict を使うと、マッピングを key と value でループ出来ます。

```
vars:
  users:
    ged:
      name: Hawk
    arha:
      name: Tenar
tasks:
  - name: Print phone records
    user: name={{ item.key }} comment={{ item.value.name }}
    with_dict: users
```

4.1.3 ファイルリストをループ

with_fileglob を使うと、管理ホスト側のファイルを正規表現で指定してループできます。

task であるため、対象ホストに置かれているファイルに対してループするように思う人がいるかもしれませんが、対象のファイルは対象ホストではないことに注意が必要です。with_fileglob で指定したファイルが存在しない場合、task は一度も実行されずにスキップされます。エラーが表示も出ないため、問題になかなか気が付きません。

```
tasks:
  - copy: src={{ item }} dest=/opt/app/conf/ owner=root mode=600
    with_fileglob:
      - /opt/app/conf/*
```

なお、role として fileglob を使っている場合、相対パスは roles/<role 名>/files ディレクトリからの相対パスとして展開されます。

fileglob で得られる得られるファイルパスは絶対パスへと変換されています。ファイル名を別途使いたい場合は後述する filter を使い、パス名をファイル名へと変換するなどの処理が必要です。

4.2 出力を保存してあとで使う – register

register を使うとモジュールの出力を保存しておいて、あとで使えます。

```
tasks:
  - name: /tmp の情報を得る
    file: dest=/tmp state=directory
    register: tmp
  - name: /var/tmp を設定する
    file: dest=/var/tmp mode={{ tmp.mode }} state=directory
  - name: 日付を得る
    command: date +"%Y%m%d"
    register: date
  - name: 日付のファイルを作成する
    file: path=/tmp/log.{{ date.stdout }} state=touch
```

playbook を記述している時など、register の内容を知るには **debug** モジュールを使うと便利です。

```
- debug: var=date
```

結果は以下のようになります。これにより、date.stdout を設定すればよいことが分かります。

```
"date": {
        "changed": true,
        "cmd": [
            "date",
            "+%Y%m%d"
        ],
        "delta": "0:00:00.012852",
        "end": "2014-06-09 23:46:41.268260",
        "invocation": {
            "module_args": "date +\"%Y%m%d\"",
            "module_name": "command"
        },
        "rc": 0,
        "start": "2014-06-09 23:46:41.255408",
        "stderr": "",
        "stdout": "20140609",
        "stdout_lines": [
            "20140609"
        ]
    }
```

この例で実行した file モジュールであれば stat があるなど、register に登録される内容はモジュールによって異なります。ただし、**stdout** は必ずありますので、標準出力を得たい時は **date.stdout** とします。

4.3 条件付き実行 – when

task に when を付けることで、その task を実行する条件を設定できます。

```
tasks:
 - name: apache を入れる (debian 系)
   apt: name=apache2 state=installed
   when: ansible_os_family == "Debian"
 - name: apache を入れる (redhat 系)
   yum: name=httpd state=installed
   when: ansible_os_family == "RedHat"
```

この例では、Debian 系であれば apt で apache2 を、RedHat 系であれば yum で httpd をインストールします。

ansible_os_familty は setup という ansible が自動で実行するホストの情報を収集するモジュールによって自動で設定される変数です。もちろん、vars で設定した変数を使うこともできます。

複雑な条件も設定できます。

```
vars:
  - word: "hello"
tasks:
  - name: 変数 word が定義されていたら
    command: echo "{{ word }}"
    when: word is defined
  - name: 変数 word が定義されて、かつ、ansible でなかったら
    command: echo "ansible"
    when: word is defined and word is not "ansible"
```

and や not は Jinja2 の文法です。他の言語で使用される&&などではありませんのでご注意ください。

先ほどの register と組み合わせると以下のような使い方が出来ます。

```
- template: src=test.j2 dest=/tmp/test.j2
  register: test_out
- debug: msg="test が変わりました"
  when: test_out|changed
- debug: msg="test は変わっていません"
  when: test_out|skipped
- debug: msg="test_out 中に spam がありました"
  when: test_out.stdout.find('spam') != -1
```

この例では、template モジュールの結果が test_out に格納されます。その後 test_out をチェックし、/tmp/test.j2 が書き換わっていれば"test が変わりました"と出力され、変わっていなければ"test は変わっていません"と出力されます。

|のあとは次の四つが使えます。

- failed
- success
- skipped

- changed

また、最後に示した例のように **stdout.find()** を使うことで、標準出力に指定の文字列が含まれているかどうかなどを判定できます。

4.4 成功するまで繰り返す – until

register と **until** を使うと task が終了する、あるいは任意の状態になるまでリトライできます。

```
tasks:
  - shell: /usr/bin/check_system
    register: result
    until: result.stdout.find ("all system green") != -1
    retries: 5
    delay: 10
```

この例では、 /usr/bin/check_system を実行し、標準出力中に “all system green” という文字列が出てくるまで 10 秒間隔で /usr/bin/check_system を実行します。5 回繰り返して “all system green” という文字列が出なかったら失敗となります。

成功した場合、最後に実行した内容が register で示された値に格納されます。また、その場合 “attempts” という key が追加され、その値に何回リトライしたかという値が格納されます。

4.5 外部情報の参照 – lookup

今まで変数を参照するために**{{ }}**で変数を囲ってきました。この囲みの中では単に変数を入れるだけでなく、多くの事ができます。その一つが **lookup** です。

以下にいくつかの例を挙げます。

```
vars:
  # redis から fookey という key で値を取ってきます
  foovalue: "{{ lookup('redis_kv', 'redis://redishost:6379,fookey') }}"
tasks:
  - name: 環境変数 HOME を表示する
    debug: msg="{{ lookup('env', 'HOME') }}"
  - name: 管理ホストの公開鍵を使って対象ホストの authorized_key を設定する
    authorized_key: user=foo
                    key="{{ lookup('file', '/home/foo/.ssh/id_rsa.pub') }}"
```

このように、**{{ lookup(‘動作’, ‘引数’) }}**のように書くことで、lookup を呼び出し、その結果を {{}} が置かれている箇所に挿入できます。例えば AWS_SECRET_KEY といった環境変数を使用するサービスを活用できます。

注意していただきたいのは、lookup が実行するのは管理ホスト側だということです。例えば env は対象ホストの環境変数ではなく、管理ホストで ansible-playbook コマンドを実行したユーザーの環境変数ということです。

なお、lookup は lookup_plugin という仕組みでユーザー自身が拡張できます。これについては付録で述べます。

4.6 変数を処理する – filter

{{ }} の中は Jinja2 という Python のテンプレートエンジンで処理されており、Jinja2 の機能としてこの中で渡した変数に対していろいろな処理ができます。これを **filter** と言います。Ansible は Jinja2 の filter に加えて独自の filter も定義しています。

```
vars:
  path: /usr/bin/python
  sample:
    - complex:
       - a
       - b
tasks:
  - name: ファイル名を取り出す
    debug: msg="{{ path | basename }}"  # -> python
  - name: ディレクトリ名を取り出す
    debug: msg="{{ path | dirname }}"  # -> /usr/bin
  - name: md5 を計算
    debug: msg="{{ path | basename | md5 }}"
  - name: JSON 形式に変換
    debug: msg="{{ sample | to_nice_json }}"
  - name: デフォルト値を設定
    debug: msg='{{ novalue | default(1000) }}'
  - name: バージョン文字列を比較
    debug: msg={{ ansible_distribution_version |
    version_compare('12.04', '>=') }}
```

さらに言うと、前述の**条件付き実行** – *when* で述べた **test_out|changed** も filter です。

ここで紹介したのはごく一部の filter のみです。これ以外にも多くの filter が用意されています。また、自分で plugin として filter を作成することもできます。

4.7 キーボードから入力する – vars_prompt

変数を予め指定するのではなく、実行時にユーザーに入力してもらいたい場合があります。その場合、**vars_prompt** を使います。

```
hosts: all
connection: local
vars_prompt:
  yourname: "名前を入力してください"
```

これにより、変数 yourname に入力した値が入ります。

ヒント: 日本語プロンプト

Ansible 1.6.6 ではプロンプトに日本語を表示できません。現在修正する Pull Request が提案されていますので、Ansible 開発者によって取り込まれましたら問題が解決すると思います。

変数のデフォルト値を設定したい場合は `default` を設定します。一段階深い階層が必要な点と、格納先変数名を `name` で設定することに注意してください。

```
vars_prompt:
   - name: yourname
     default: "Ursula"
     prompt: "名前を入力してください"
```

標準では入力した文字がそのまま表示されます。パスワードなど、表示されては困る場合は `private` を設定します。

```
vars_prompt:
   - name: yourname
     prompt: "名前を入力してください"
     private: yes
```

PassLib という Python のライブラリを管理ホストにインストールしている場合、パスワード生成も同時に行えます。

```
vars_prompt:
  - name: md5_password
    prompt: "Enter password"
    private: yes
    encrypt: "md5_crypt"
    confirm: yes
    salt_size: 7
```

encrypt にはアルゴリズムを記載でき、md5 の他 sha1、sha256、DES といったアルゴリズムを指定できます。

また、`salt_size` を指定すると自動生成した salt を使い、`salt` を指定するとその値を salt として使います。

4.8 管理ホストで実行する – local_action

通常の task は対象ホストで実行されます。しかし、例えば事前に送るファイルを tar で圧縮しておくなど、管理ホスト側で実行したい task もあります。

その場合、`local_action` を使うとその task は管理ホスト側で実行されることとなります。

```
tasks:
  - name: あとで送るように、send ディレクトリを固めておく
    local_action: command tar cvfz send.tar.gz send chdir=/tmp/
```

local_action では、最初の引数に使用するモジュール名を書きます。

また、local_action は次の章で説明する、EC2 との連携などにも使用します。

4.9 実行するモジュールを変数で変更する – action

通常、実行するモジュールを定義するにはこう書きます。

```
- name: nginx を apt で入れる
  apt: name=nginx
```

一方、action を使って指定することもできます。

```
- name: nginx を apt で入れる
  action: apt name=nginx
```

この形式は、以前のバージョンで使われていたもので、冗長ですので現在はあまり使いません。

しかし、apt や yum など、ディストリビューションが使っているパッケージマネージャが格納されている ansible_pkg_mgr という変数を使うと便利になります。

```
- name: nginx を環境のパッケージマネージャで入れる
  action: "{{ ansible_pkg_mgr }} name=nginx"
```

これにより、CentOS では yum で、Debian では apt で、といったように、環境に合わせたパッケージマネージャを実行してくれます。ただし、name で指定したパッケージ名が異なっていると使えませんので、注意が必要です。

4.10 環境変数を設定する - environment

HTTP_PROXY など実行時に環境変数を設定したい場合があります。**environment** を設定することで、task 実行時の環境変数を設定できます。

```
tasks:
   - apt: name=nginx state=installed
     environment:
       http_proxy: http://proxy.example.com:8080
```

4.11 失敗しても無視する - ignore_errors

task が失敗すると、通常はそこで処理が終了します。失敗しても処理を続行して欲しい場合は、**ignore_errors** を使います。

```
tasks:
  - name: 必ず失敗します
    command: /bin/false
    ignore_errors: yes
```

Ansible はデフォルトでは終了コードが 0 以外の場合を失敗とみなしますので、正常時でも終了コードが 0 以外を返すようなコマンドを利用する場合にも便利です。

4.11.1 失敗条件を定義 – failed_when

ignore_errors はどのようなエラーでもすべて無視します。しかし、通常使用する場合、想定しているエラーの時だけ無視し、それ以外の本当に問題が起きた場合はきちんと報告してほしいこと

が多くあります。**failed_when** を使うと、失敗条件を定義できます。

```
tasks:
  - name: exit 1 を実行する。0 か 1 以外の場合はエラーとする
    command: shell exit 1
    register: exit
    failed_when: exit.rc not in [0, 1]
```

register と組み合わせ、exit.rc のように終了ステータスを得ることが多いでしょう。この例では、終了ステータスが 0 か 1 以外の場合はエラーとなります。

あるいは stdout を使い、標準出力中に指定の文字がなかったら、という使い方もあります。

4.11.2 変更条件を定義 – changed_when

task を実行すると、終了コードなどによって変更されたとみなされます。しかし、変更されたとみなされると handler が起動するなどが行われますので、変更されたとみなしてほしくない場合もあります。**changed_when** はそのような場合に使います。

```
tasks:
  - name: 終了コードが 2 の場合は変更とみなさない
    shell: /usr/bin/spell --mode="with true name"
    register: result
    changed_when: "result.rc != 2"
```

4.12 非同期で task を実行する – async

通常 Ansible は playbook を実行している間 ssh 接続を継続します。しかし、長時間かかる task を実行させていると、ssh のコネクションがタイムアウトしてしまうことなどの不具合が起きる場合があります。

そのような場合 **async** を使い、いったん接続を切って非同期に動作させることができます。

```
tasks:
  - name: とても長い時間かかるコマンドを実行
    command: /bin/sleep 15
    async: 45
    poll: 5
```

実行結果には以下のようにかかっている時間が表示されます。

```
TASK: [command /bin/sleep 15]  ***************************

<job 930224669966.14266> polling, 40s remaining
<job 930224669966.14266> polling, 35s remaining
```

引数は

poll task 終了をチェックする間隔 (秒)。指定がない場合は 10 秒

async 終了を待ち受ける最大時間 (秒)

です。

他の task を待ち受ける必要がなく実行するだけでいい task に関しては、以下のように poll を 0 にすると、task の終了を待たずに次の task を実行します。その場合、async で設定した時間を超えてもエラーにはなりませんが、async の設定は非同期にするために必要です。

```
tasks:
  - name: 実行完了を待たないで次の task を実行します
    command: /bin/sleep 15
    async: 45
    poll: 0
```

もしもそのまま playbook 自体が終了した場合でも、プロセスは対象ホスト上で動き続けます。メモリを大量に使うプロセスなどの場合は特に注意してください。また、例えば yum モジュールなど、ロックを獲得するような動作を行った場合、その後の task で同じロックを獲得することはできなくなり、失敗します。

また、後述するコマンドラインオプション--forks で並列数を上げることも検討してください。非同期にするよりも並列数を上げるほうが高速になる場合もあります。

本章では Ansible が提供する複雑な機能について述べました。このすべてを一度に使う必要はありません。必要となった時にこの章を参照してください。また、ここで紹介した以外にも多くの機能がありますし、紹介した機能の中にも実はこういう使い方ができる、というものもあります。それらについては Ansible の公式ドキュメントをご覧ください。

次章では、playbook の構成が大規模になってきた時にどうすればよいかについて述べていきます。

第5章　大規模なplaybookを構築してみよう

今まで説明してきたplaybookの書き方は、一つのplaybookファイルにすべてを書くものでした。

しかし、taskが増えたり、使いまわしたりしていくと、一つのファイルでは見通しが悪くなります。本章では、そのようにplaybookが大規模になってきた場合について述べます。

5.1 他のplaybookを読み込む – include

includeを使うと、他のファイルを読み込めます。例えば以下のようなplaybookがあったとします。

```
---
- name: すべてのことを行うplaybook
  hosts: all
  remote_user: root
  tasks:
    - name: echo doall
      tags: doall
      dmesg: msg="すべてのことを行います"

- include: load_balancers.yml
- include: webservers.yml
- include: dbservers.yml
```

includeは書かれたインデント階層に対して読み込まれます。従って、この例ではhostsと同じ階層に読み込まれることになります。

load_balancers.ymlは以下のように書かれているとします。

```
---
- name: load balancer設定
  hosts: lb
  remote_user: lbuser
  tasks:
    - dmesg: msg="load balancerを設定します"
```

このplaybookを実行すると、「すべてのことを行います」というメッセージを表示した後に、load_balancers.yml、webservers.yml、dbservers.ymlとうい３つのplaybookを実行します。

もちろん、load_balancers.ymlだけを実行することも可能です。

最初からincludeを前提にしていくと、どのような粒度で分割していけばいいか悩むことになります。まずは、一つだけのplaybookファイルに記述しておき、大きくなってきたら分割するという進め方で問題ありません。

5.1.1 task ファイルを分ける

先ほどの例では、別々の playbook を一つにつなげた例でした。従って、例えば hosts や remote_user の設定は個々の playbook の設定が使われます。

task だけを共有し、hosts などは設定したい、ということもあります。その場合、task ファイルだけを分割し、それを読み込むことも出来ます。

例として、まず最初に webservers.yml というファイルとを作ります (“tasks” というディレクトリを作り、その中に置くと良いでしょう)。

```
---
- name: nginx の使用準備
  apt: name=nginx
- dmsg: msg="{{ user }}さん、nginx を設定します"
```

通常の playbook には hosts 項目が必要ですが、このファイルでは設定していないことに注意してください。このファイルには通常 tasks 以下に書く、task のみを書きます。

こうしておいて、以下のような playbook を書きます。

```
---
- hosts: all
  tasks:
    - include: tasks/webservers.yml  user=ansible
```

こうすると、tasks/webservers.yml で設定した task が実行されます。また、その時に変数 user を渡すことも出来ます。

5.1.2 変数を設定する

include で読み込んだ playbook に対して変数を設定することも出来ます。

```
tasks:
  - include: earthsea.yml user=onyx
  - include: earthsea.yml user=hawk
```

もっと複雑な変数を設定したい場合はこう書くこともできます。

```
- include: earthsea.yml
  vars:
      remote_user: hound
      usename:
        - ogion
        - rose
        - crow
```

5.2 推奨ディレクトリ構成 – ベストプラクティス

Ansible では、以下のようなディレクトリ構成でファイルを配置することが推奨されています。ただし、これはあくまで推奨であって必須ではありません。このベストプラクティスにとらわれず

に、使いやすい構成にするほうが良い結果となります。

```
.
|-- group_vars        # 各グループごとの変数設定用ディレクトリ
|   |-- dbservers     # dbservers 用変数ファイル
|   `-- webservers    # webserers 用変数ファイル
|
|-- host_vars         # 各ホストごとの変数設定用ディレクトリ
|   |-- db01          # db01 用変数ファイル
|   |-- web01         # web01 用変数ファイル
|   `-- web02         # web02 用変数ファイル
|
|-- production        # 本番環境用 inventory ファイル
|-- staging           # staging 環境用 inventory ファイル
|-- site.yml          # site 全体用 playbook ファイル
|-- dbservers.yml     # db 用 playbook ファイル
|-- webservers.yml    # web 用 playbook ファイル
|
`-- roles             # ロール用ディレクトリ （後述します）
```

5.2.1 group_vars、 host_vars

inventory **ファイル**で group と host に対して設定できることを紹介しました。group_vars や host_vars という名前のディレクトリを作成し、その中にグループ名と同じ名前のファイルを作成しておくと、自動的に読み込み、変数としてくれます。

例えば、以下の構成でファイルが置かれていたとします。

```
.
|-- group_vars
|   `-- dbservers
|-- host_vars
|   `-- db01
```

ここで、dbservers というグループに属している db01 というホストに対して操作を行う場合は、

- dbservers
- db01

の二つのファイルから変数が読み込まれ、適用されます。

5.3 まとめて再利用 – role

role は一連の task を共有するための仕組みです。include は必要な部分だけ共有するという仕組みに対して、role は task や template で使うファイルなど、その role の中で必要な情報をすべてその role で使うディレクトリ内に格納しています。従って、role を使う指定をするだけで期待している動作を行えます。

role はこのように独立しているため、他者との共有がしやすいです。後述する ansible-galaxy コマンドを使うと、世界中から投稿されている role を使えます。

5.3.1 role のディレクトリ構造

以下に role を使う際のディレクトリ構造を示します。トップの roles はこのディレクトリ名でなければいけません。また、tasks、files などもこのディレクトリ名である必要がありますし、main.yml というファイル名である必要があります。しかし、全部を作る必要はなく、最低限 tasks ディレクトリだけあれば role として機能します。

```
roles  # roles という名前である必要あり
|-- common   # 共通 role 用ディレクトリ
|   |-- defaults       # 変数のデフォルト設定ディレクトリ
|   |   `-- main.yml
|   |-- files          # ファイル。file モジュールなどで使われる
|   |   |-- init.conf
|   |   `-- someconf.txt
|   |-- handlers       # ハンドラー task 用ディレクトリ
|   |   `-- main.yml
|   |-- meta           # メタ情報用ディレクトリ
|   |   `-- main.yml
|   |-- tasks          # task 用ディレクトリ
|   |   `-- main.yml
|   |-- templates      # テンプレートファイル用ディレクトリ
|   |   `-- var.js
|   `-- vars           # 変数用ディレクトリ
|       `-- main.yml
|-- app         # app role 用ディレクトリ
|-- mysql       # mysql role 用ディレクトリ
`-- nginx       # nginx role 用ディレクトリ
```

この階層構造を見ていただくと分かりますが、role は再利用を想定していますので、一つの role の中には動作に必要なすべての情報が含まれています。その中で変更可能な部分だけを変数として切り出して、外部から設定する、という使い方になります。

role の置き場所は ansible.cfg という設定ファイルで設定できますので、別リポジトリに role だけを集めておき、その中から必要な role だけを個別のリポジトリから使う、ということも可能です。

5.3.2 role の作成方法

例えば、 roles/common/tasks/main.yml は以下のように記述します。

```
---
- name: 共通ユーザーを作成
  user: name={{ common_user_name }} state=present
- name: 鍵登録
  authorized_key: user={{ common_user_name }}
                  key=common_user_key.pub

- include: other_common.yml
```

ここで、

- common_user_name という変数は **vars/main.yml** から
- common_user_key.pub というファイルは **files** 以下から

検索されます。そのため、files/などをつける必要はありません。

また、最初は必ず main.yml が読み出されますが、上記例にあるように **include** を使って分割することも出来ます。

5.3.3 role の使い方

このように roles 以下を作成しておけば、

```
---
- hosts: webservers
  roles:
     - common
     - nginx
     - app
```

のように使えます。

それぞれの role 内の vars で設定した変数は以下のようにすることで上書きできます。

```
roles:
   - common
   - { role: app, dir: '/opt/a',  port: 5000}
```

また、when を指定することで、条件に基づいて読み込むことや、後述するタグを追加することもできます。

```
roles:
   - common
   - { role: centos, when: "ansible_os_family == 'RedHat'"}
   - { role: app, tags: ["devel", "db"]}
```

5.3.4 role と task の実行順序

もし、tasks と roles の両方が playbook にあった場合、

1. role
2. task

の順番で実行されます。もしも role の前や後に task を実行したい場合、**pre_tasks** や **post_tasks** を使います。

```
---
- hosts: all

  pre_tasks:
    - shell: echo 'run_1'

  roles:
    - run_2
```

```
  tasks:
    - shell: echo 'run_3'

  post_tasks:
    - shell: echo 'run_4'
```

この例の場合、 run_1, run_2, run_3, run_4 の順番で標示されます。

5.3.5 role 内で使う変数名

role 内で使用する変数のうち、vars セクションなどの外部から与えることを想定している変数の名前は**{role 名}_{変数名}**のように role 名を接頭辞として使うと他の role や task と重複してしまう可能性が低くなります。

例えば、 nginx という role であれば以下のような変数名です。

```
nginx_worker_connections = 1024
nginx_keepalive_timeout = 65
```

role は再利用されるものですので、変数名に少し気を使うと使い勝手が良くなります。

5.4 並列実行 – fork

対象のホストが多くなってくると、Ansible の実行に時間がかかってきます。特に 6 台以上となるとかなりの時間がかかります。これは、標準では Ansible は 5 台までのサーバーしか並列に動作しないためです。

`-f`、`--forks` オプションをつけて実行するか、あるいは ansible.cfg 設定ファイルの `forks` を設定することで並列実行数を変えられます。

```
$ ansible-playbook -f 50 large_hosts.yml
```

並列実行数が増えると、同時に実行できる ssh の数が増えます。ssh に必要な CPU やメモリは比較的少ないため、この数字を 20 から 50 ぐらいにしても多くの場合問題ありません。

5.5 順々に実行する – serial

複数台のサーバーがあり、それぞれの設定を更新して再起動していく場合などでは、並列実行をしてしまうと、すべてのサーバーを一度に落としてしまいます。

`serial` を付けることにより、task を実行するホストは、設定した台数分だけとなります。

```
- hosts: webservers
  user: root
  serial: 1
```

serial を適切に設定することで、サービス自体は落とさないまま順々に更新していく、いわゆる rolling upgrade が実現できます。

5.6 AWS EC2 との連携

AWS や Digital Ocean などのクラウドでインスタンスを立ち上げて必要な設定を行いたい場合があります。Ansible は多くのクラウドに対応しています。

例として、AWS EC2 を立ち上げ、role を実行する playbook を示します。AWS などを操作するのは管理ホストから行いますので、**local_action** を使います。また、動的にホスト情報を追加するため、**add_host** モジュールを使う必要があります。

```
---
- hosts: localhost
  gather_facts: no
  tasks:
    - name: EC2 インスタンス起動
      local_action:
        module: ec2
        instance_type: m1.small
        image: ami-c9562fc8
        group: your-group-name
        region: ap-northeast-1
        key_name: your-key-name
        wait: yes
        count: 1
      register: ec2
    - name: host グループに新しいインスタンスを追加
      local_action:
        module: add_host
        hostname: "{{ item.public_ip }}"
        ansible_ssh_private_key_file: your-key-name
        groupname: launched
      with_items: ec2.instances
    - name: SSH が通じるまで待つ
      local_action:
        module: wait_for
        host: "{{ item.public_dns_name }}"
        port: 22
        delay: 60
        timeout: 320
        state: started
      with_items: ec2.instances

- hosts: launched
  user: ec2-user
  roles:
    - somerole
```

インデントに注意してください。つまり、この playbook では インスタンス起動と somerole という role の実行という二つのことを行っています。

そのため、例えば後半部分を別のファイルに記述し、以下のように書いても構いません。

```
(省略)
        state: started
      with_items: ec2.instances

- include: do_somerole.yml
```

なお、インスタンスを立ち上げた場合、ssh での host_key チェックが入りますので、後述の ansible.cfg にて `host_key_checking=False` としておくと完全に自動化できます。

5.7 ホストのリストを動的に作成 – dynamic inventory

対象ホストが多くなってくると一つの inventory ファイルで管理することが困難になってきます。また、AWS などのクラウドを使用している場合、ホストを作ったり破棄したりが多くなり、inventory ファイルの管理が大変になってきます。

inventory ファイルとしてスクリプトを指定すると、Ansible はそのスクリプトを実行してその結果をホストやグループの情報として使います。

例えば、GitHub 上の Ansible のリポジトリには ec2.py というファイルがあります。

ec2.py と ec2.ini をダウンロードし、playbook と同じ場所に置き、ec2.ini を適宜編集します。また、環境変数 **AWS_ACCESS_KEY_ID** などを設定します

その後、以下のように実行すると、ec2 上のホストを ansible から扱えます。

```
$ ansible-playbook -i ec2.py -l security_group_default deploy.yml
```

この例では対象ホストを制限する `-l` オプションで security group を指定しています。-l オプションについては次章で述べます。

ec2.py では

- instance id
- region
- availability zone
- security group
- tag

などで自動的にグループ分けが行われていますので、任意のグループに対して playbook を実行できます。

どういうグループ分けがされているかは `python ec2.py` と実行すれば JSON 形式で標示されますので参照してください。

EC2 の他に Google Compute Engine や Docker、あるいは Zabbix などからもホストリストを取得するスクリプトが GitHub 上の Ansible リポジトリにて提供されています。また、自分で作成することも可能です。

ヒント: inventory ファイルとしてディレクトリを指定すると、そのディレクトリ以下すべてのファイルを inventory ファイルとして実行、あるいは単に読み込み、結合したリストを inventory として扱います。これにより、静的な inventory と dynamic inventory とを組み合わせることが出来ます。

本章までで playbook に関連する説明は終わりです。ここまで理解できればどんな構成になったとしても問題はなくなります。

次章ではコマンドラインから指定できるさまざまな設定について述べます。

第6章　コマンドラインオプションを使ってみよう

ansible-playbook コマンドには多くのコマンドラインオプションがあり、指定することでより便利な使い方が出来ます。

本章ではそのコマンドラインオプションを説明します。

6.1 ssh 認証

下記のオプションを指定することで、使用する認証情報をコマンドラインから設定できます。

-k 、 --ask-pass ssh の接続用パスワードを最初に設定できます

--private-key=key ssh 接続で使用する秘密鍵を指定します

--ask-su-pass su のパスワードを最初に設定できます

-K 、 --ask-sudo-pass sudo パスワードを最初に設定できます

-S 、 --su su で実行するよう設定します

-s 、 --sudo sudo で実行するよう指定します

-R SU_USER 、 --su-user=SU_USER su 実行時にユーザーを設定できます。デフォルトは root です

-U SUDO_USER 、 --sudo-user=SUDO_USER sudo 実行時のユーザーを指定できます。デフォルトは root です

-u REMOTE_USER 、 --user=REMOTE_USER ssh 接続時のユーザー名を設定できます。デフォルトは実行ユーザーです

`--K` は playbook 中の `sudo:  yes` と同じ意味になります。ただし、コマンドラインオプションで指定した方が優先されます。

`-k` や`-K` でパスワードを入力することにより、公開鍵が置かれていないホストに対しても ansible-playbook を実行できます。また、テストなどにも便利です。

ヒント: Ansible では sudo ではなく su を使うこともできます。

6.2 対象ホストを制限する – limit

実行対象ノードを選択するために、通常は hosts: all というように playbook 内で設定します。しかし、-l オプションを使うと、これを上書きできます。

```
# web グループに
$ ansible-playbook -l web deploy.yml

# web グループのうち web02 以外
# !の前にエスケープが必要
$ ansible-playbook -l "web:\!web02" deploy.yml

# web グループか db グループに属しているホスト
$ ansible-playbook -l "web:db" deploy.yml

# web グループかつ db グループに属しているホスト
$ ansible-playbook -l "web:&db" deploy.yml
```

このように、実行対象について複雑な指定ができます。playbook 内では検証環境を設定しておき、本番環境に対してデプロイを行うときなどの時にコマンドラインで指定する、などの利用方法が考えられます。

6.3 実行する task を制限する – tag

task に tags を付けておき、コマンドラインオプションで-t を指定すると、一つの playbook 中で任意の task だけを実行できます。

例えば、以下のように tag をつけたとします。

```
tasks:
  - yum: name=httpd state=installed
    tags:
      - install
      - config
  - template: src=httpd.conf.j2 dest=/etc/httpd/httpd.conf
    tags:
      - config
```

tags を指定しておき、-t あるいは--tags をコマンドラインで指定します。

```
$ ansible-playbook site.yml --tags "install"
```

とすると、install が設定されている task のみが実行されます。

tag は複数設定できます。

```
# 複数のタグを実行
$ ansible-playbook site.yml --tags "install,config"
```

--skip-tags で指定した tag を飛ばすことも出来ます。

```
# --skip-tags で指定した tag を実行しない
$ ansible-playbook site.yml --skip-tags "install"
```

6.4 dry-run 実行 – check

--check をつけて実行すると、task を実際に実行はせずに、変化が起きるかどうかだけを報告してくれます。いわゆる **dry-run** 機能です。check mode に対応しているモジュールであれば、さらに、どのような変化が起こるかを報告してくれます。

```
$ ansible-playbook something.yml --check
```

なお、always_run をつけた task は check モードを指定した場合でも必ず実行されます。

```
- name: この task は checkmode でも実行されます
  command: /something/to/run --even-in-check-mode
  always_run: yes
```

6.5 task を確認しながら実行 – step

playbook 中で実行される task を一つ一つ確認しながら実行していきたい場合があります。その時には--step を指定します。このオプションを実行すると

```
Perform task: command uname (y/n/c):
```

のように、すべての task について実行するかを聞いてきます。

y 実行する

n その task をスキップして次の task を実行する

c (continue) その task 以降すべての task を実行する

c を入力すると残りすべての task が実行されますので注意してください。

6.6 差分表示 – diff

--diff をつけて実行すると、変化した内容の差分表示が出来ます。--check と合わせて使うことで、これからどのような変化が起きるのかを実際に見てから動作させることが出来ます。

```
$ ansible-playbook example.yml --diff

# check との組み合わせ
$ ansible-playbook example.yml --check --diff
```

ただし、diff を使うと変更点が多い場合、膨大な量の表示がされる場合があります。複数のホストに対して実行するとその分だけ増えますので、diff を使う際は-l で対象ホストを一つに絞り込んだほうが良いです。

本章ではコマンドラインオプションについて述べました。playbook に書くのではなく、コマンドラインオプションで指定することで簡単に書き換えられます。

次章では、ansible-vault という変数を暗号化する機能について述べます。

第7章　変数ファイルの暗号化 – ansible-vault

変数を設定するとき、変数が記述されているファイルに直接変数の内容を記述していました。しかし、例えばパスワードやアクセストークンなどはそのまま記述すると、セキュリティ上問題になります。仮にそのまま GitHub など誰もが見られる場所で公開してしまった場合には、大変な影響が出ます。

公開したくない情報を記述するとき、`ansible-vault` コマンドを使うと vars_file で指定しているファイルを暗号化できます。ansible-vault は AES256 形式で暗号化しています。かなり強力な暗号化方式であり、現時点においては解読される可能性はほぼありません。

7.1 ansible-vault の使い方

ansible-vault コマンドは Ansible をインストールすると ansible-playbook コマンドと同じようにインストールされます。ここでは ansible-vault コマンドの使い方を説明します。

7.1.1 暗号化ファイルを作成

まず最初に、以下のように `create` を指定し暗号化ファイルを作成します。

```
$ ansible-vault create vars.yml
Vault password:
Confirm Vault password:
(ここで環境変数 EDITOR で定義されたエディタが開くので入力)
```

この例では `vars.yml` というファイルが作成されます。エディタで入力する内容は通常の vars と同じく

```
---
some_key: value1
other_key: other_value
```

という形式で変数を記述していきます。

入力が終わったらエディタを閉じると自動的に ansible-vault コマンドも終了し、指定したファイル名で暗号化されたファイルが作成されます。

暗号化されたファイルは以下のような内容となっており、変数自体が分からなくなっています。

```
$ANSIBLE_VAULT;1.0;AES
53616c7465645f5f7a4860a465a200b34ec42466ca2d6c8da01ad09ca5a
a781247f75f3faee603e2203118abcd25f8d62a80c096e46f9969e6967f
2ae98842b119a8143e8a5251a32c9988
```

7.1.2 暗号化

create ではファイルを作成するコマンドでした。encrypt を指定すると、指定した平文のファイルを暗号化します。

```
$ ansible-vault encrypt vars.yml
Vault password: <パスワードを入力します>
Confirm Vault password: <確認パスワードを入力します>
Encryption successful
```

この例では vars.yml というファイルを暗号化します。vars.yml ファイルそのものが置き換わり、バックアップも作成されませんので、パスワードは忘れないようにしてください。忘れてしまうと二度と変数を取り出すことができなくなります。

7.1.3 復号

decrypt を指定すると暗号ファイルを復号できます。

```
$ ansible-vault decrypt vars.yml
Vault password:  <パスワードを入力します>
Decryption successful
```

encrypt と同じく、指定したファイルそのものが置き換わる点に注意してください。

7.1.4 編集

edit を指定すると、暗号ファイルを直接エディタで編集できます。

```
$ ansible-vault edit vars.yml
 Vault password:
 (ここで環境変数 EDITOR で定義されたエディタが開くので入力)
```

encrypt や decrypt と同じく、指定されたファイルそのものが置き換わる点に注意してください。

7.1.5 パスワードの変更

rekey を指定すると、一度設定したパスワードを変更できます。

```
$ ansible-vault rekey vars.yml
Vault password: <現在のパスワードを入力します>
New Vault password: <新パスワードを入力します>
Confirm New Vault password: <新パスワードを確認します>
Rekey successful
```

7.2 暗号化されたファイルの使い方

上記のコマンドで暗号化しておいた変数ファイルは、通常の変数を読み込む vars_files で指定します。

```
---
- hosts: localhost
  vars_files:
      - vars.yml
  tasks:
    - debug: msg="{{ spam }}"
```

ansible-playbook 実行時に--ask-vault-pass をつけると、パスワードを聞いてきます。

```
$ ansible-playbook -i ansible_hosts  vault.yml --ask-vault-pass
Vault password:
```

あるいは、以下のように--vault-password-file オプションでパスワードファイルを指定することもできます。

```
$ ansible-playbook -i ansible_hosts vault.yml \
      --vault-password-file ~/.ssh/pass.txt
```

暗号化されたファイルを使用しているのにパスワードを指定しなかった場合、「パスワードの指定が必要です」というエラーが出ます。

```
ERROR: A vault password must be specified to decrypt data
```

ansible vault はファイル全体を暗号化するため、どのような変数が入っているかも分からなくなります。そのため、例えば変数名で grep したり、ということができなくなります。

将来的には改善されていく可能性がありますが、現状ではこういう問題があるということを覚えておくといいでしょう。

本章では ansible-vault という暗号化機能について述べました。構成管理ツールは鍵やパスワードなどセキュリティ上重要なファイルを扱うことが必要なため、ansible-vault は特に重要となります。

次章では Ansible Galaxy という全世界の人で role を共有し、簡単に扱えるようにするツールについて説明します。

第8章　公開されているroleを使ってみよう – Ansible Galaxy

本章では**まとめて再利用** – *role* で説明した role を全世界で共有する **Ansible Galaxy** という仕組みについて説明します。

8.1 Ansible Galaxy とは

Ansible Galaxy とは https://galaxy.ansible.com/にて公開されている、Ansible, Inc. によって開発・運営されている Web サービスです。

Ansible Galaxy には世界中の人が開発している role が登録されており、自由にダウンロードして使うことが出来ます。

role には必要な task に加えてテンプレートや変数が含まれていますので、Ansible Galaxy から role をダウンロードして、変数を上書きしてあげれば望みの動作ができます。

Chef では Community Cookbook があり、それと同じ位置づけです。しかし、Community Cookbook との違いは、Ansible Galaxy は投稿されている role がユーザーに結びついており、同じことを行う role でも複数の人が投稿することができる、という点が異なります。これにより、Community Cookbook ではなかなか受け入れてくれなかった変更を受け入れてくれる余地が大きくなります。

例えば、Chef の Community Cookbook には、nginx 用の Cookbook は一つだけしかありません。この一つの Cookbook は非常に汎用的に作られており、たくさんの変数を設定してほとんどの人が望みの動作を実行できるようになっています。しかし、そのために多くの場合分け処理が読みにくくなっています。また、変更も影響する人が多いため慎重にならざるを得ません。

しかし、Ansible Galaxy では少なくとも三つの nginx role が存在しています。これらは違う人が作成しており、作者の目的を満たすために作られています。従って自分の用途に合わないかもしれません。しかし、それをコピーして自分用に変更しても良いのです。汎用的ではない分シンプルで分かりやすくなっています。

8.2 roleの検索方法

Ansible Galaxy を使うには、まず Ansible Galaxy の Web ページを開き、右上の “Browse Roles” をクリックします。

“Search role name” から関連するキーワードを入力すると、検索結果が表示されます。

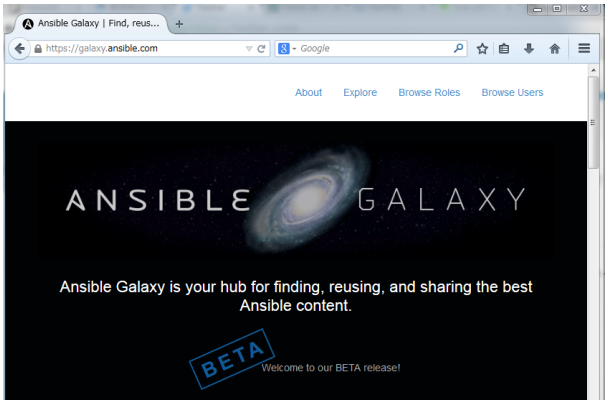

図 8.1: Ansible Galaxy トップページ

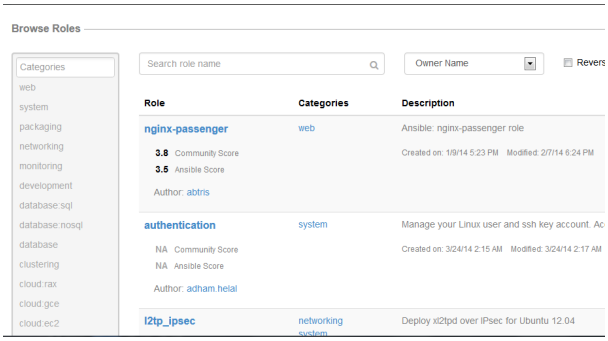

図 8.2: role の検索

ユーザーが投稿した role には **Community Score** と **Ansible Score** という二つのスコアが付けられています。

Community Score とはコミュニティの各ユーザーがランク付けしたスコアで、Ansible Score は Ansible, Inc. の従業員がランク付けしたスコアです。Ansible Score が付いていると安心です。

スコアを見て気になる role がありましたら、その role をクリックしてください。role の詳細が出てきます。その詳細ページで表示されるうち、重要な項目としては以下の項目があります。

Supported Platforms サポートされているプラットフォーム (Ubuntu natty など)

Role Variables role で使用可能な変数

License ライセンス

これらの情報を見て、使いたい role を選んでください。

8.3 role を手に入れる

使いたい role が決まったら、role の詳細ページの真ん中に表示されている **Installation** を見てください。以下のように書かれているはずです。

```
$ ansible-galaxy install <ユーザー名.role 名>
```

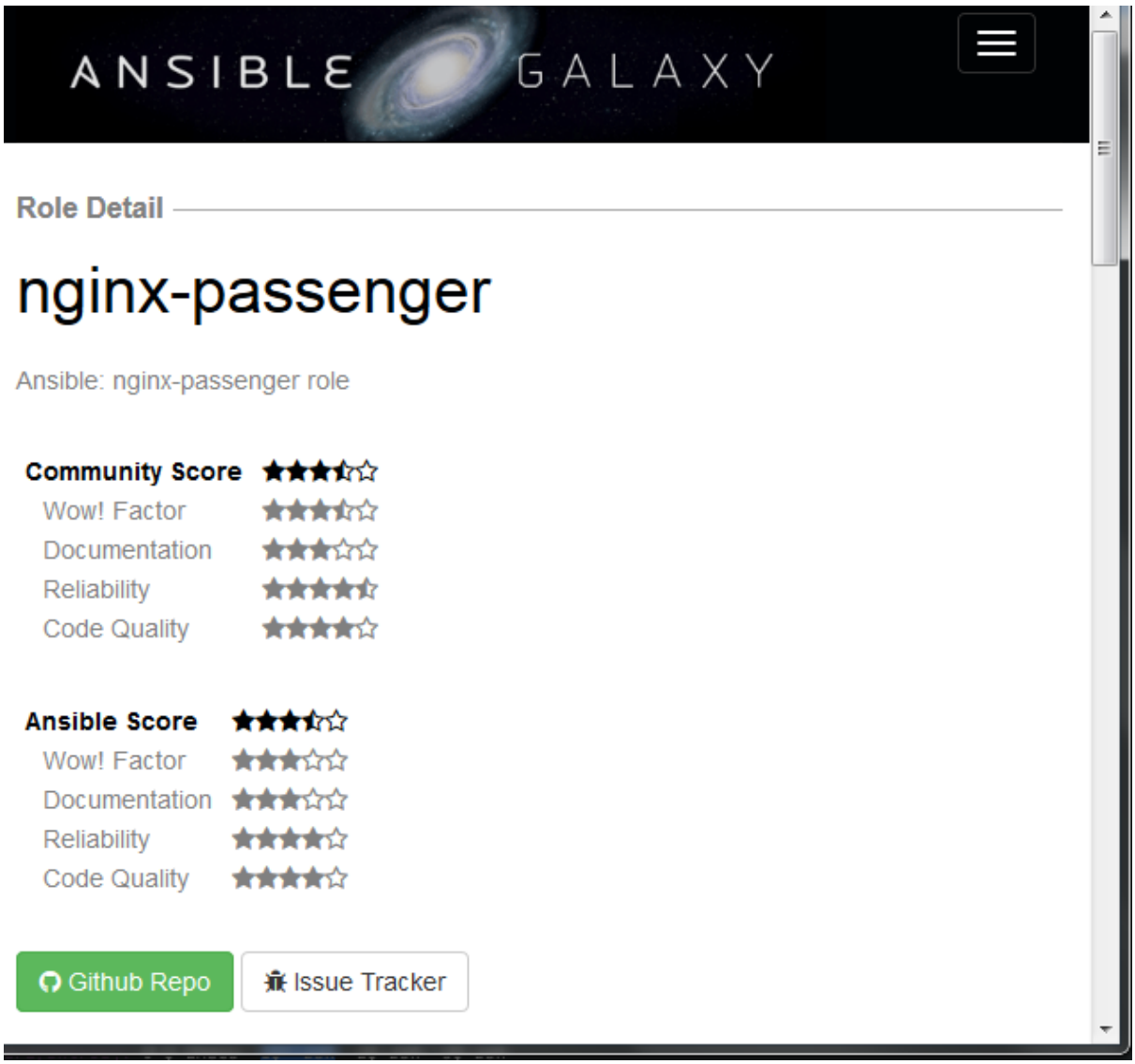

図 8.3: role の詳細画面

```
例:
$ ansible-galaxy install bennojoy.nginx
```

これをそのままコピーし、任意の場所で入力します。そうすると

```
downloading role 'nginx', owned by bennojoy
no version specified, installing master
- downloading role from
  https://github.com/bennojoy/nginx/archive/master.tar.gz
- extracting bennojoy.nginx to /etc/ansible/roles/bennojoy.nginx
bennojoy.nginx was installed successfully
```

と出てきて、/etc/ansible/roles/以下にインストールされます。

なお、/etc/ansible 以下などではない場所にインストールしたい場合は**-p** オプションでインストール先を指定できます。playbook がある場所にインストールしておくのも一つの考えです。

8.4 role の使い方

これで準備が出来ました。あとは通常通り role を指定するだけです。

```
---
- hosts: all
  roles:
    - {role: nginx,
       nginx_http_params: { sendfile: "on",
                            access_log:
                            "/var/log/nginx/access.log"},
                            nginx_sites: none }
```

role の中で使用する変数を上書きしています。多くの変数を指定する場合があるので、上記例のように改行すると見やすくなるでしょう。

本章では Ansible Galaxy の使い方について説明しました。ここでは説明しませんでしたが、自分で作成した role を Ansible Galxy に投稿することもできます。自分で作成した role を Ansible Galaxy に投稿し、みんなに使ってもらうことで、さらに洗練された role にしていくことができます。

第9章　よくあるご質問

本章では、よくあるご質問を紹介します。

9.1 接続できない

コマンドラインから ssh で接続できるかどうかをご確認ください。

`-vvvv` をつけて実行すると、ssh 接続の様子が分かりますので、問題の解決に役立ちます。

.ssh/config ファイルでトンネル設定などをしている場合、paramiko 接続では.ssh/config を読みこまないため、接続できません。inventory ファイルにて以下の設定をしてください。これにより、paramiko 接続ではなく OpenSSH による接続になります。

```
target_host ansible_connection=ssh
```

なお、CentOS 6 などの RHEL 6 系では OpenSSH が古く、そのままでは以下のエラーで ssh 接続ができません。

```
using -c ssh on certain older ssh versions may not support ControlPersist,
```

この問題を解決するためには、環境変数 `ANSIBLE_SSH_ARGS` を以下のように設定します。

```
export ANSIBLE_SSH_ARGS=""
```

あるいは、 ansible.cfg にて以下の設定を記述します。

```
[ssh_connection]
ssh_args =
```

9.2 ControlPath too long というエラーが出る

接続できず、`-vvvv` をつけて実行すると、以下のようなエラーが出ていることが分かる場合があります。

```
ControlPath "/Users/gedo/.ansible/cp/ansi" too long for Unix domain socket
subsystem request failed on channel 0
Connection closed
```

MacOSX をお使いの場合や、Amazon EC2 など長いホスト名に対して実行した場合などにこのエラーが出ることがあります。

これは、ssh の ControlPath の保存パスが長すぎる場合に起きる問題です。解決するには ansible.cfg で以下のように設定してください。

```
[ssh_connection]
control_path = %(directory)s/%%h-%%r
```

9.3 実行しても途中で止まる

`gathering fact` と表示されて止まった場合、gathering fact は数分程度時間がかかる場合がありますので、しばらくお待ちください。

5 分以上止まっている場合は、sudo の prompt が表示されて止まっているかもしれません。

`-K` や`--ask-sudo-pass` で入力したパスワードが間違っていないか、ssh でログインしたユーザーで sudo ができるかなどを確認してください。

9.4 inventory ファイルがなくても接続したい

以下のようにコマンドラインで入力することで、inventory ファイルがなくてもホスト名や IP アドレスの指定ができます。

```
$ ansible-playbook -i "127.0.0.1," test.yml
```

最後に , をつける必要があります。

9.5 一つの playbook が複雑になってしまった

いろいろな条件分岐の task を一つにまとめたり、構成管理とアドホックコマンドを一つにまとめたりしていくと、playbook が大きく複雑になる場合があります。

複雑な条件分岐が必要な場合、変数を切り出し、role に分割していくことをまず検討してください。

また、構成管理とアドホックコマンドは別々の playbook にしたほうが良い場合があります。

例えば、EC2 インスタンスの起動をする playbook とインストールをする playbook を別々に分けても問題ありません。むしろその方がすっきりする場合もあります。

9.6 python not found というエラーが出る

なにも設定せずに FreeBSD や NetBSD の対象ホストに対して ansible を実行すると、以下のエラーが出ます。

```
invalid output was: /usr/bin/python: not found
```

これは host 側で実行に必要な Python が/usr/bin/python となっているためで、inventory ファイルで以下の設定を行う必要があります。

```
freebsd_host  ansible_python_interpreter=/usr/local/bin/python
```

9.7 Windows で使いたい

残念ながら現時点では Windows を管理ホストとして Ansible を動かすことは出来ません。Cygwin をインストールし、Cygwin 上の Python を使用して Ansible を動かすことは出来ます。

なお、執筆段階ではまだ開発途中ですが、次期バージョンの 1.7 からは、Windows を対象ホストとして動作させることができるようになります。具体的には WinRM を利用して通信し、PowerShell を使うモジュールを動かせるようになります。

9.8 ansible-playbook に変な絵が表示される

もしもその絵が牛の場合、cowsay というコマンドがインストールされていませんか。

```
export ANSIBLE_NOCOWS=1
```

とするか、 ansible.cfg ファイルで nocows を 1 に設定してください。

9.9 ansible が収集する変数を知りたい

ansible_os_family など、ansible が収集し、変数として多くの情報が使えます。

どのような情報を使えるかは以下のように、ansible コマンドで setup モジュールを実行します。

```
ansible -m setup <hostname>
```

使用可能な情報が JSON として表示されます。かなりの量がありますので、適宜スクロールが必要になります。

9.10 invalid type <type 'list'>と出る

以下の様に shell モジュールを使用した場合、

```
tasks:
  - name: ファイル存在確認
    shell: [ -f /etc/already ]
    register: exists
```

このようなエラーが出ます (改行しています)。

```
ERROR: action specified for task ファイル存在確認 has \
          invalid type <type 'list'>
```

これは [ ] がリストとみなされるために起こります。"で囲むことで修正できます。

```
shell: "[ -f /etc/already ]"
```

9.11 変数を使った時に Syntax Error が出る

文の途中に変数を埋め込む場合は、"で囲む必要があります。具体的には

```
with_items:
   - apache-{{ version }}.tar.gz
```

ではなく、

```
with_items:
   - "apache-{{ version }}.tar.gz"
```

とする必要があります。

9.12 --- はどういう意味？

この本で述べた playbook や Ansible の公式ドキュメントでは、多くの場合---が先頭行についています。YAML の仕様では、---で区切ることにより、複数の YAML をひとつのファイルに含めることができます。

```
---
name: foo
email: foo@mail.com
---
name: bar
email: bar@mail.org
```

ただし、複数の YAML を扱うには Stream という手法を使う必要があり、Ansible では使えません。また、---がなくても問題ありません。

9.13 Ansible という名前の由来は？

代表的な作品として「ゲド戦記」がある、アーシュラ・K・ル＝グウィンという作家がいます。「ハイニッシュ・サイクル」という一連の SF 作品の中に「アンシブル」という超光速通信を行う機械が登場しており、これが由来です。

アンシブルはマイクとスピーカー、キーボードとディスプレイを備えた弁当箱サイズの機械だという描写があります。

第10章　おわりに

本書では Ansible のインストールから playbook の作り方、大規模な playbook の構築方法などについて述べてきました。**ssh だけで動作するという手軽さ**と、**YAML で書くことにより非技術者でも扱いやすいこと**が Ansible を手軽に使えるツールとしています。

また、単なる構成管理ではなく、**アドホックコマンド**によって多くの日常的あるいは非日常的な作業を簡単に多数のサーバーに対して行うことができることを述べました。

Ansible は YAML を採用しているため、通常のプログラム言語に比べて表現力は劣っています。しかし、**ループ**や**条件付き実行**、*register* **による実行結果の保存**など、多くの機能を搭載することにより、YAML の書きやすさ、読みやすさを保ちつつ高い表現力を実現しています。

Ansible は拡張ポイントが随所に存在しています。YAML では表現できない機能についてはユーザーがどんどん拡張していけます。これから続く付録には自分で望みの拡張を実装できる方法を紹介しています。

さらに、**Ansible Galaxy** についても紹介しました。Ansible の role は再利用しやすい形式となっており、Ansible Galaxy は全世界の Ansible ユーザーが作成した role を検索し、ダウンロードし、実行できます。うまく使えばほとんど playbook を書く必要がなくなります。

ここで本書は終了となりますが、ここまでの内容でみなさまのサーバ管理業務にかかる時間が短縮されることを願っております。

付録A　モジュールを自作する

Ansible には 200 以上のモジュールがすでに用意されていますが、さらにモジュールを自分で作成することが出来ます。

- 使えるモジュールがない
- 既存のモジュールの動作を拡張・改変したい
- 既存のモジュールにバグがある
- シェルスクリプトなどの過去の資産を有効に使いたい

このような場合は、自分でモジュールを作成することをお勧めします。

Ansible 本体は Python 言語で実装されていますが、モジュールはどんなスクリプト言語で実装されていても構いません。実際、Shell Script がよく利用されています。

ヒント: モジュールは確かに気軽に作成できますが、既存モジュールの組み合わせで実現できることであれば、モジュールを作成しないほうが良いです。

A.1 モジュールの動作

モジュールは ansible-playbook コマンドを実行する管理ホストから、ssh を用いて実行対象ホストにモジュール自信が転送され、そこで実行されます。従って、モジュールが別のソフトウェアに依存している場合、管理ホストではなく対象ホストにその依存ソフトウェアが入っている必要があります。

A.1.1 モジュールを置く場所

ansible-playbook コマンドは playbook 内で指定したモジュールを以下の順番で検索します。

1. ansible.cfg の library
2. 環境変数 `ANSIBLE_LIBRARY`
3. コマンドオプション`--M` あるいは`--module-path` で指定
4. 実行したディレクトリの直下にある library という名前のディレクトリ以下
5. role 内の library という名前のディレクトリ以下

あるリポジトリに対して特別なモジュールが必要であれば、そのリポジトリ内に library というディレクトリを作成すると、共有が容易にできるようになります。

A.2 モジュールの形式

モジュールは入力と出力に対してそれぞれ規約があり、それに沿うように実装する必要があります。逆に言えば、沿ってさえいればどのような言語で実装されていても構いません。

モジュール名とファイル名とは同一にする必要があります。従って、.py などの拡張子をつけてはいけません。また、一行目の shebang には

```
#!/usr/bin/python
```

のように、`#!/usr/bin/`以下を指定してください。env などを使用してはいけません。これは Ansible がこの行を置き換えるからです。

A.2.1 入力

モジュールには入力として引数が以下のように渡されます。

```
モジュールのファイルパス 引数ファイルパス
```

引数ファイルパスを元に、そのファイルを開くと、以下のような形式で引数が保存されています。

```
引数名 1=値 1 引数名 2=値 2 ...
```

モジュールへの引数ではなく、ファイルを開く必要があるので少し実装が多くなります。

A.2.2 出力

出力は JSON 形式が基本です。

```
{"changed": true, "time": "2014-06-17T23:44:54.83543"}
```

以下の特殊なプロパティがあります。

changed 変更があった場合に true にセットします。セットしない場合は ok となります

failed 失敗した時に true にセットします

msg 失敗した時にセットしておくと、表示されます

ansible_facts マップ形式で値を入れて返すと、その値が変数として代入されます

rc 終了コードを指定できます

また、以下のフォーマットで出力することもできます。これは主に JSON を簡単には扱えない shell script 用です。

```
key=value rc=0 changed=true favcolor=green
```

A.3 モジュールのサンプル

以下に ruby で書いたサンプルを示します。 “first” と “second” という引数を受け取り、合計値を返します。(require ‘json’ を使用しているので、対象ホストに ruby 1.9 以上が必要です)

```
#!/usr/bin/ruby

require 'json'

File.open(ARGV[0]) do |fh|
   # 空白を区切る
   args = fh.read().split(" ")
   # =で区切る
   args.map! {|arg| arg.split("=")}
   # ハッシュに代入する
   data = {}
   args.each do |arg|
     key = arg[0]
     value = arg[1]
     data[key] = value
   end

   begin
      a = data['first'].to_i()
      b = data['second'].to_i()
   rescue
      # 失敗を返す
      print JSON.dump({
          'failed' => True,
          'msg'    => 'failed to parse args'
      })
      exit(1)
   end

   result = {
      'first'   => a,
      'seconds' => b,
      'sum'     => a + b,
   }

   print JSON.dump(result)
end
```

このように、Ansible のモジュールは Ruby や Shell Script などで実装出来ます。

A.4 Python での便利関数

モジュールはどんな言語でも実装できますが、Python であれば Ansible が提供している便利な関数を使用できます。

```
from ansible.module_utils.basic import *
main()
```

と書いておくと、便利な関数が使えるようになります。

例えば、引数の読み込みは以下のように記述できます。

```
module = AnsibleModule(
    argument_spec = dict(
        state     = dict(default='present',
                         choices=['present', 'absent']),
        name      = dict(required=True),
        enabled   = dict(required=True, choices=BOOLEANS),
        etc       = dict(aliases=['something'])
    )
)
```

choice や aliases が使用でき、引数のバリデーションも同時にできます。

また、返り値も以下のように記述できます。

```
# 成功した場合
module.exit_json(changed=True, result=3)

# 失敗した場合
module.fail_json(msg="module failed")
```

そのほか AnsibleModule クラスには多くの関数があるので使用することにより、さらに簡単にモジュールを作成出来ます。

A.5 モジュールのデバッグ

モジュールは対象ホスト側で実行されるため、標準出力などに途中経過を出しても画面上に表示されません。また、モジュールは実行のたびに削除されます。

モジュールをテストするには、GitHub から Ansible のリポジトリを取得し、その中の hacking/test-module を使います。まず下のように git clone でリポジトリを取得し、test-module に実行権限を付与します。

```
git clone git@github.com:ansible/ansible.git
chmod +x ansible/hacking/test-module
```

実行は-m でモジュールを、-a で引数を指定します。結果は標準出力に書きだされます。

```
$ ansible/hacking/test-module -m ./sum -a "first=1 second=2"

********************************
RAW OUTPUT
{"first": 1, "second": 2, "sum": 3, "changed": true}
```

また、ANSIBLE_KEEP_REMOTE_FILES 環境変数を 1 にすると、通常は実行後に自動的に削除されるモジュールが対象ユーザーの$HOME/.ansible 以下に残ったままになります。実際のファイル名は実行するたびに変わりますので-vvv とつけてファイル名を確認してください。

付録B　pluginを自作する

Ansible は plugin 構造になっています。自分で plugin を書くことにより、Ansible の動作を拡張できます。

ここでは Ansible の plugin について説明します。

B.1 pluginの種類

Ansible には以下の 5 種類の plugin があります。

lookup plugin lookup 関数を用いて変数を動的に設定するときに使います

filter plugin Jinja2 で使う filter を拡張します

callback plugin playbook 開始時や task 終了時、エラー発生時などのタイミングで呼ばれる callback を設定します

action plugin モジュールと組み合わせて使われる plugin です

connection type plugin 操作対象に対して接続する際に使います

vars plugin group_vars や host_vars の変数を設定します

これらは指定のディレクトリに置くだけで自動的に Ansible が認識します。

なお、モジュールはどんな言語でも実装できますが、残念ながら plugin は Ansible が直接実行するので Python で実装するしかありません。

B.2 lookup plugin

標準で使える plugin は GitHub の `lib/ansible/runner/lookup_plugins/`にあります。

lookup plugin の使い方の例を示します。一つ目の引数 `file` は実行する lookup plugin の種類を示し、二つ目の引数には plugin に渡す引数を設定します。

```
vars:
  # /etc/foo.txt の内容を contents 変数に入れます
  contents: "{{ lookup('file', '/etc/foo.txt') }}"

  # 環境変数 HOME を home 変数に入れます
  home: "{{ lookup('env','HOME') }}"
```

二つ目の引数は、, で区切ることで複数の引数を渡せます。

```
tasks:

  - name: redis から'somekey' というキーの値を get して表示します
    debug: msg="{{ lookup('redis_kv', 'redis://localhost:6379,somekey') }}
```

lookup plugin を自作する場合は、以下の二種類の方法のどちらかで指定したディレクトリに.py をつけたファイルを置くと、実行してくれます。

- playbook ファイルがあるディレクトリに lookup_plugins というディレクトリを作る
- ansible.cfg で lookup_plugins=<パス>で指定する

コラム: lookup と with

yum モジュールなどでは with_items を使って複数のアイテムを task に渡すという使い方をよくします。この with_items は実は items lookup plugin を呼び出しています。従って、lookup 関数は必ずリストを返します。

B.3 filter plugin

template モジュールなどで{{ foo | to_json }}というようにフィルターを指定できます。標準で使えるフィルターは GitHub の lib/ansible/inventory/filter_plugins/core.py にあります。

例としていくつかを挙げてみます。

b64encode, b64decode base64 エンコード/デコードをします。

failed, success, changed 必ず指定した状態になります。テストなどに使います。

search, regex, regex_replace 正規表現を実行します

basename, dirname ファイルパスを変換します

random ランダムな数字を返します。

これらは例えば以下のように使います。

```
# 'encoded' という変数を base64 でエンコードした値となります
{{ encoded | b64decode }}

# 0 から 59 までのランダムな値となります
{{ 59 |random }} * * * * root /script/from/cron

# 'ansible' を'able' に変換します
{{ 'ansible' | regex_replace('^a.*i(.*)$', 'a\\1') }}
```

filter plugin はこのフィルターを拡張するものです。

B.3.1 filter plugin のサンプル

以下の 2 種類の方法のどちらかで指定したディレクトリに.py をつけたファイルを置くと、実行してくれます。どのようなファイル名であっても.py がついていれば実行されます。

- playbook ファイルがあるディレクトリに `filter_plugins` というディレクトリを作る
- ansible.cfg で `filter_plugins=<パス>`で指定する

filter plugin は `FilterModule` というという名前のクラスを定義し、`filters` というメソッドで任意の名前と関数を含んだ辞書を返すだけです。

```
def first_func(a):
    return a[0]

class FilterModule(object):
    def filters(self):
        return {
            'first': first_func,
        }
```

これにより、`{{ var | first }}`というフィルターが使えるようになります。なお、この first という filter は与えられた文字列の先頭だけを取り出す filter となります。

B.4 callback plugin

callback plugin は標準では提供されていません。GitHub の `plugins/callbacks` 以下にサンプルがありますので、そちらをダウンロードする必要があります。

いくつかサンプルがありますが、その中の 3 つを紹介します。

hipchat task の実行が進むにつれて、HipChat というチャットサービスに状況を送ってくれる

mail 結果をメールしてくれる

osx_say osx の say コマンドを利用して失敗した時に音声合成で喋ってくれる

B.4.1 callback plugin のサンプル

callback plugin のサンプルです。`CallbackModule` という名前のクラスを定義し、そのクラスで所定のメソッドを定義します。

```
class CallbackModule(object):
    def __init__(self):
        pass
    def playbook_on_task_start(self, name, is_conditional):
        pass
    def runner_on_ok(self, host, res):
        pass
```

定義されたタイミングでこのメソッドが呼ばれます。定義しない場合は単純に無視されます。多くのメソッドが定義できますが、ここでその一部を紹介します。

playbook_on_play_start playbook 開始時に呼び出されます

playbook_on_task_start task 開始時に呼び出されます

runner_on_ok task が成功した時に呼び出されます

runner_on_failed task が失敗した時に呼び出されます

runner_on_skipped task がスキップされた時に呼び出されます

playbook_on_stats playbook 終了時に呼びだされます

callback plugin は以下の 2 種類の方法のどちらかで指定したディレクトリに.py をつけたファイルを置くと、実行してくれます。

- playbook ファイルがあるディレクトリに `callback_plugins` というディレクトリを作る
- ansible.cfg で `callback_plugins=`<パス>で指定する

このディレクトリに複数の callback plugin を置いた場合は、すべて実行されます。

B.5 action plugin

モジュールの説明で述べたように、モジュールは対象ホストで実行されます。しかし、例えば管理ホスト側からファイルを送るなど、管理ホスト側で実行したいモジュールもあります。

action plugin は、そのような場合にモジュールと組み合わせて使われる plugin です。action plugin にて管理ホスト側での前準備を実装し、モジュールにて対象ホスト側で転送したあとの操作を実装する、という具合です。

管理ホスト側の操作だけで完結する場合は、モジュールを作る必要はありません。

action plugin は以下の 2 種類の方法のどちらかで指定したディレクトリに.py をつけたファイルを置くと実行してくれます。

- playbook ファイルがあるディレクトリに `action_plugins` というディレクトリを作る
- ansible.cfg で `action_plugins=`<パス>で指定する

ファイル名はそのままモジュール名となります。つまり、`wizard.py` というファイル名をつけた場合には `wizard` モジュールとして呼び出します。

B.5.1 action plugin のサンプル

action_plugin は以下のように `ActionModule` というクラス名を定義し、その中で `run` というメソッドを定義します。

```
from ansible.runner.return_data import ReturnData
from ansible.callbacks import vv, vvv

class ActionModule(object):
    def __init__(self, runner):
        self.runner = runner

    def run(self, conn, tmp, module_name, module_args,
            inject, complex_args=None, **kwargs):

        module_return = self.runner._execute_module(conn, tmp,
            'wizard',module_args, inject=inject,
            complex_args=complex_args)
        return ReturnData(conn=conn, comm_ok=True,
                          result=module_return.result)
```

`self.runner._execute_module` でモジュール側を呼び出しています。モジュール側は通常のモジュールと変わらず、管理ホスト側で実行されます。この例では同じ wizard というモジュールを呼び出していますが、任意のモジュールを実行できます。

ActionModule で重要なのが `conn` です。これは次に述べる connectiontype plugin となっており、put_file や fetch_file でファイルの送信・受信が出来ます。

B.6 connection type plugin

ssh や paramiko などの対象に接続する方式を実装した plugin です。実際にはこの plugin を自作することはないと思います。

GitHub の `lib/ansible/runner/connection_plugins/`以下に置かれています。

connection type plugin は以下の 5 つのメソッドを実装したクラスを定義します。

- connect
- put_file
- fetch_file
- exec_command
- close

ただし、connection type plugin はユーザーが自分で作成することはほぼありませんので、本書では割愛します。

B.7 vars plugin

`group_vars` や `host_vars` といった変数を扱います。ただし、vars_plugin の作成は推奨されておらず、inventory スクリプトで変数を扱う方がより容易に扱えるため、inventory スクリプトが推奨されています。そのため、本書では割愛します。

付録C　ansible configファイル

Ansible では、ansible.cfg というファイルに ansible 全体における設定を記述します。

Ansible は実行時に ansible.cfg を以下の順番で検索し、最初に見つかったものを使用します。

1. 現在のディレクトリにある ansible.cfg
2. ANSIBLE_CONFIG 環境変数
3. ホームディレクトリの.ansible.cfg
4. /etc/ansible/ansible.cfg

ここでは ansible config ファイルを解説します。 () 内は初期設定の値です。

C.1 defaultセクション

ansible 全体に関する設定です。

```
[default]
hostfile       = /etc/ansible/hosts
library        = /usr/share/ansible
```

hostfile (/etc/ansible/hosts) inventory ファイルの場所を指定します

library (/usr/share/ansible) module の場所を指定します

remote_tmp ($HOME/.ansible/tmp) 対象ホストで使用する一時ファイル置き場を指定します

pattern (*) 対象ホストのグループが指定されなかった場合のデフォルトを指定します。ansible-playbook コマンドでしか使われません

forks (5) 並列実行する数を指定します

poll_interval (15) async 時にチェックするデフォルトの間隔を設定します

sudo_user (root) デフォルトの sudo をするユーザーを指定します

ask_sudo_pass (False) True にすると実行時に最初に sudo パスワードを聞いてきます

ask_pass (False) True にすると ssh 接続時にパスワードを聞いてきます

transport (smart) デフォルトの接続方式を設定します。smart の場合、ControlPersist が使えれば ssh を、使えなければ paramiko を使います

remote_port (22) デフォルトの ssh 接続ポートを設定します

module_lang (C) module 実行時の lang 設定です

gathering (implicit) 対象ホストで実行する fact 収集のポリシーに関する設定です。“implicit” の場合、“gather_facts: False” と明示的に設定しない限り収集されます。“explicit” の場合、明示的に設定しない限り収集されません。“smart” の場合、同じホストに対して複数回の playbook 実行をした場合には収集がスキップされます

roles_path (/etc/ansible/roles) Ansible が role を読み込む場所を設定します。なお、roles_path で読み込む場所をしなくても、Ansible は playbook があるディレクトリからも role を読み込みます

host_key_checking (True) ssh の host_key チェックを設定します。チェックをしない場合は False に設定してください。ただし、セキュリティには十分気をつけてください

sudo_exe (sudo) 管理ホストで sudo 以外の実装を使用する場合、この設定によって実行コマンドを設定できます

sudo_flags (-H) sudo に付け加えるオプションフラグを設定します

timeout (10) SSH のタイムアウト時間を設定します

remote_user (root) ansible-playbook で接続に使用するユーザー名を指定します。ansible コマンドでは常に現在のユーザーで接続します

log_path (/var/log/ansible.log) この設定がされた場合、指定された場所に ansible はログを書き出します。デフォルトでは ansible は syslog にログを渡します

module_name (command) ansible コマンドで-m で指定しない場合の module です

executable (/bin/bash) sudo 以下で使用する shell を指定します

hash_behaviour (replace) ansible は標準では変数を上書きします。変数が辞書 (ハッシュ) だった場合、通常では上書きしますが、“merge” を指定した場合、完全な上書きではなく、辞書中にすでにある値のみが上書きされる merge となります

jinja2_extensions (jinja2.ext.do,jinja2.ext.i18n) Jinja2 の拡張子を追加できます。これは ansible の開発者用ですので、変える必要はありません

private_key_file (/path/to/file) デフォルトで使用する秘密鍵を設定します。`--ansible-private-keyfile` オプションで上書きできます

ansible_managed (Ansible managed: {file} modified on %Y-%m-%d %H:%M:%S by {uid} on {host}) template モジュールによって操作するファイルに挿入する文字列。この文字列があると ansible によって管理されていることが分かります

display_skipped_hosts (True) False の場合、skip された task を表示しません

error_on_undefined_vars (True) True の場合、未定義の変数を参照した場合にエラー終了します

system_warnings (True) 管理ホストにおける ansible 自体の問題について警告を出すかどうかを設定します

deprecation_warnings (True) deprecation(廃止予定) の警告を表示するかどうかを設定します

action_plugins (/usr/share/ansible_plugins/action_plugins) action plugin の置き場所を設定します

callback_plugins (/usr/share/ansible_plugins/callback_plugins) callback plugin の置き場所を設定します

connection_plugins (/usr/share/ansible_plugins/connection_plugins) connection plugin の置き場所を設定します

lookup_plugins (/usr/share/ansible_plugins/lookup_plugins) lookup plugin の置き場所を設定します

vars_plugins (/usr/share/ansible_plugins/vars_plugins) vars plugin の置き場所を設定します

filter_plugins (/usr/share/ansible_plugins/filter_plugins) filter plugin の置き場所を設定します

nocows (0) `cowsay` というソフトウェアをインストールしていると、ansible-playbook の出力がとても楽しくなります (詳細は実行してからのお楽しみです。ただし、必ず最初はテストで使ってくださいね！)。nocows を 1 にすると表示されなくなります

nocolor (0) 標準では ansible の出力には色がついていますが、nocolor を 1 にすると色がつかなくなります

C.2 paramiko セクション

ControlPersist が使えない場合、明示的に指定しない限り paramiko が使われます。そのため、RedHat Enterprise Linux 6 や CentOS 6 ではデフォルトでは paramiko が使われます。

`[paramiko]` と記述してセクションを作成した後に、python の ssh 実装である、paramiko に関する設定ができます。以下のように記述します。

record_host_keys (True) record_host_keys が True でかつ、host_key_checking が True の場合、ユーザーの hostfile をチェックし、鍵を追加します。

コラム: record_host_keys と処理速度

record_host_keys は通常 True になっています。しかし、この設定では多くのホストに対して実行すると処理速度が遅くなります。OpenSSH を使うことが推奨されていますが、なんらかの事情で OpenSSH が使えない場合で処理速度が問題になる場合は、record_host_keys を False にすると改善します。

C.3 ssh_connection セクション

`[ssh_connection]` と記述してセクションを作成した後に、OpenSSH の設定ができます。以下のように記述します。

```
[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

ssh_args (-o ControlMaster=auto -o ControlPersist=60s) ssh の引数を設定します

control_path (%(directory)s/ansible-ssh-%%h-%%p-%%r) ControlPath ソケットを保持する場所を設定します

pipelining (False) Ansible はまずモジュールのファイルを対象ホストに転送していますが、pipelining を True にすることで、ssh が多重化され、転送が早くなります。ただし、sudo を実行する場合、対象ホストの /etc/sudoers で **requiretty を無効化**する必要があります。標準では互換性のためにこの設定は False となっています

scp_if_ssh (True) SFTP が使えない対象ホストの場合、この値を True にすることで scp ではなく ssh でファイルを送るようになります

C.4 accelerate セクション

target セクションで `accelerate: true` と設定することで Accelerate モードを使用します。これは、ssh ではなく Ansible 独自の daemon を通じてデータを転送することで速度を向上するモードです。

しかし、前述の ssh pipelining を設定したほうが有用なこともあり、accelerate は非推奨となっています。

`[accelerate]` と記述してセクションを作成した後に accelerate の設定ができます。以下のように記述します。

```
[accelerate]
accelerate_port=5099
```

accelerate_port (5099) 使用するポート番号を設定します

accelerate_timeout (30) タイムアウト時間を設定します

accelerate_connect_timeout (5.0) 接続タイムアウト時間を設定します

accelerate_daemon_timeout (30) daemon の生存期間を分で指定します

accelerate_multi_key (yes) daemon に複数の鍵を登録できるようにします。必ず yes のままにしておいてください

索引