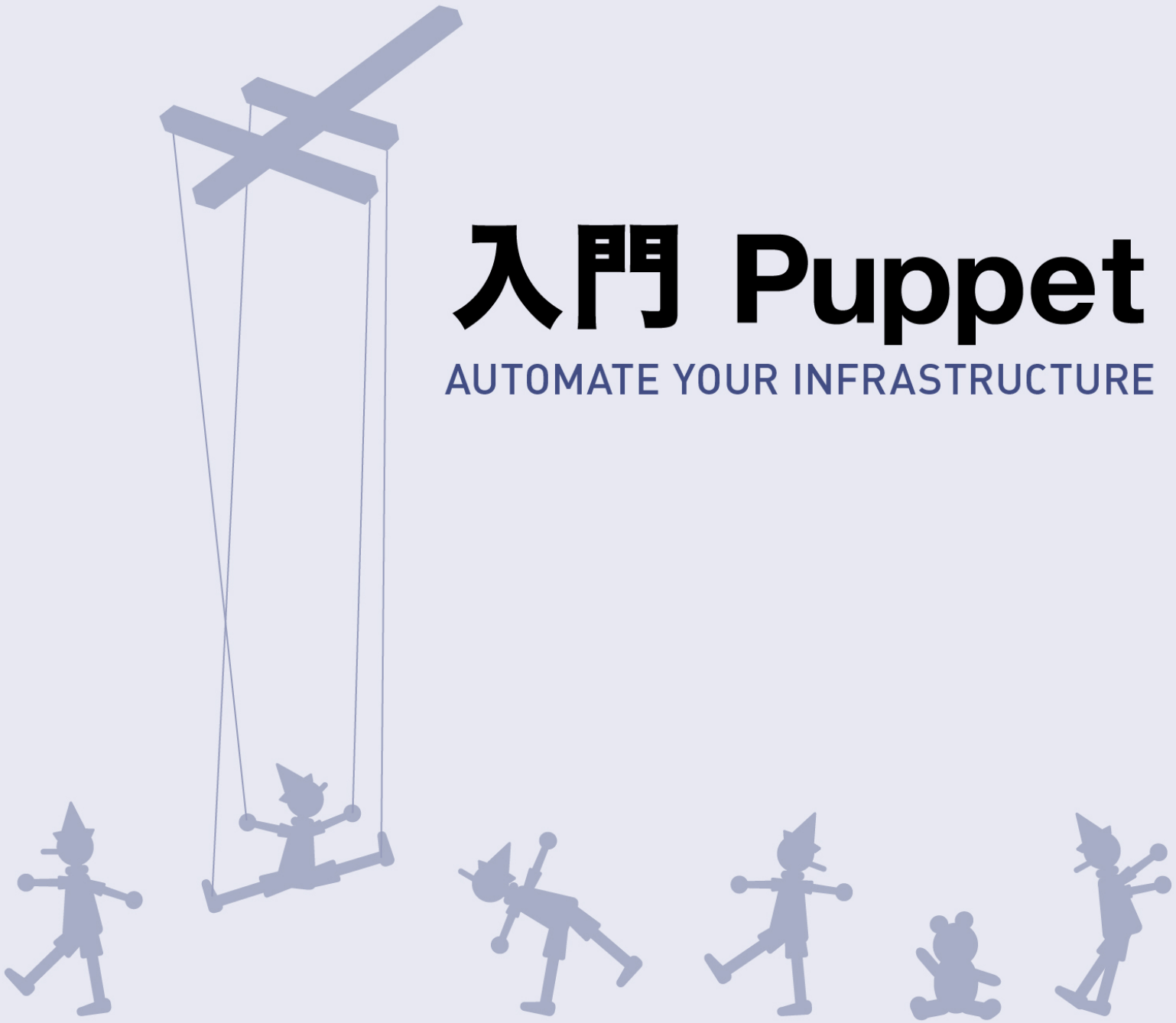

入門 Puppet

AUTOMATE YOUR INFRASTRUCTURE

入門 Puppet

栗林健太郎 著

2013-05-02 版　　達人出版会　発行

はじめに

クラウドが一般的になってきた昨今、サーバ構成管理の自動化は、もはやそれなしでは考えられないほど当たり前のものになっています。Puppet は、そのためのフレームワークのひとつです。

Puppet は 2005 年のリリース以来、後発の Chef とともに、サーバ構成管理の自動化に欠かせないフレームワークとして広く利用されてきました。とはいえ、ドキュメントが非常に充実してはいるもののその機能は膨大で、初心者にとって決してとっつきやすいものでないことは確かでしょう。現に、筆者の周りでも「Puppet を学習してみたいけど、どこから手をつけたらいいのか……」という声をよくききます。

クラウドの一般化によって、物理的な制約から離れ、サーバをあたかもプログラム上のオブジェクトであるかのように扱えるようになった現在、エンジニアにとって、Puppet のような自動化ツールを使いこなせるようになることは、技術スキルの向上に大きく寄与するでしょう。この本は、既に Puppet などの自動化ツールを使いこなしているオペレーションエンジニアよりもむしろ、技術向上への意欲を燃やすアプリケーション開発者への入門となることを目指しています。

本書の目標は、この本を読んだ読者が Puppet の基本についてひととおり知り、オペレーションエンジニアの書いた manifest(サーバのあるべき状態を記述した設定ファイルのようなもの。後述) に変更を加えたり、ある程度の規模のものなら自力でいちから書けるようになったりすることです。そのため、本書はあえてリファレンスとしての網羅性を目指しません。実際の学習段階で必要となる知識にしぼって説明します。

是非、本書を読みながら自分でも手を動かしてみて、一歩先行くエンジニアになってみませんか。

システム環境

本書執筆時の、筆者のシステム環境は以下の通りです。

- 作業環境: Mac OSX 10.8.2 + ruby 2.0.0
- 本番環境: Amazon Linux AMI 2013.03 + ruby 1.8.7

- 開発環境: Vagrant 1.1.5 + CentOS 6.4 + ruby 1.8.7
- Puppet: 3.1.1

現状、最も広く使われている Puppet のバージョンは 2.7 系だと思われますが、本書における説明の範囲内では、3.1 系でもたいした違いはありませんので、ご安心ください。

フィードバック

本書に関するフィードバックは以下におよせください。

- GitHub: http://github.com/kentaro/puppet-book-support/issues
- メール: kentarok+puppet-book@gmail.com

なお、Puppet その他にまつわるテクニカルサポートは行っておりません。その類のお問い合せに関しましては一切おこたえしかねますので、あらかじめご了承ください。

本書の書誌情報

パブー版

- puboo-v1.0.0: 2013 年 4 月 30 日
- puboo-v1.0.1: 2013 年 4 月 30 日
- puboo-v1.0.2: 2013 年 4 月 30 日
- puboo-v1.0.3: 2013 年 5 月 2 日
- puboo-v1.0.4: 2013 年 5 月 2 日
- puboo-v1.0.5: 2013 年 5 月 2 日
- puboo-v1.0.6: 2013 年 5 月 2 日

Kindle 版

- kindle-v1.0.1: 2013 年 5 月 2 日

筆者について

　栗林健太郎。ネット上では「antipop/kentaro/あんちぽ」として知られる。市役所職員、株式会社はてな勤務を経て、現在は株式会社 paperboy&co. で技術基盤整備エンジニアとして勤務。Perl Monger で Rubyist、時々ぺちぱー。http://kentarok.org/

目次

第1章

なぜ Puppet が必要なのか？

Puppet は、サーバの構成管理を自動化するためのフレームワークです。2005 年のリリース以来、Chef と並んで、その分野におけるスタンダードなフレームワークとして、広く使われています。

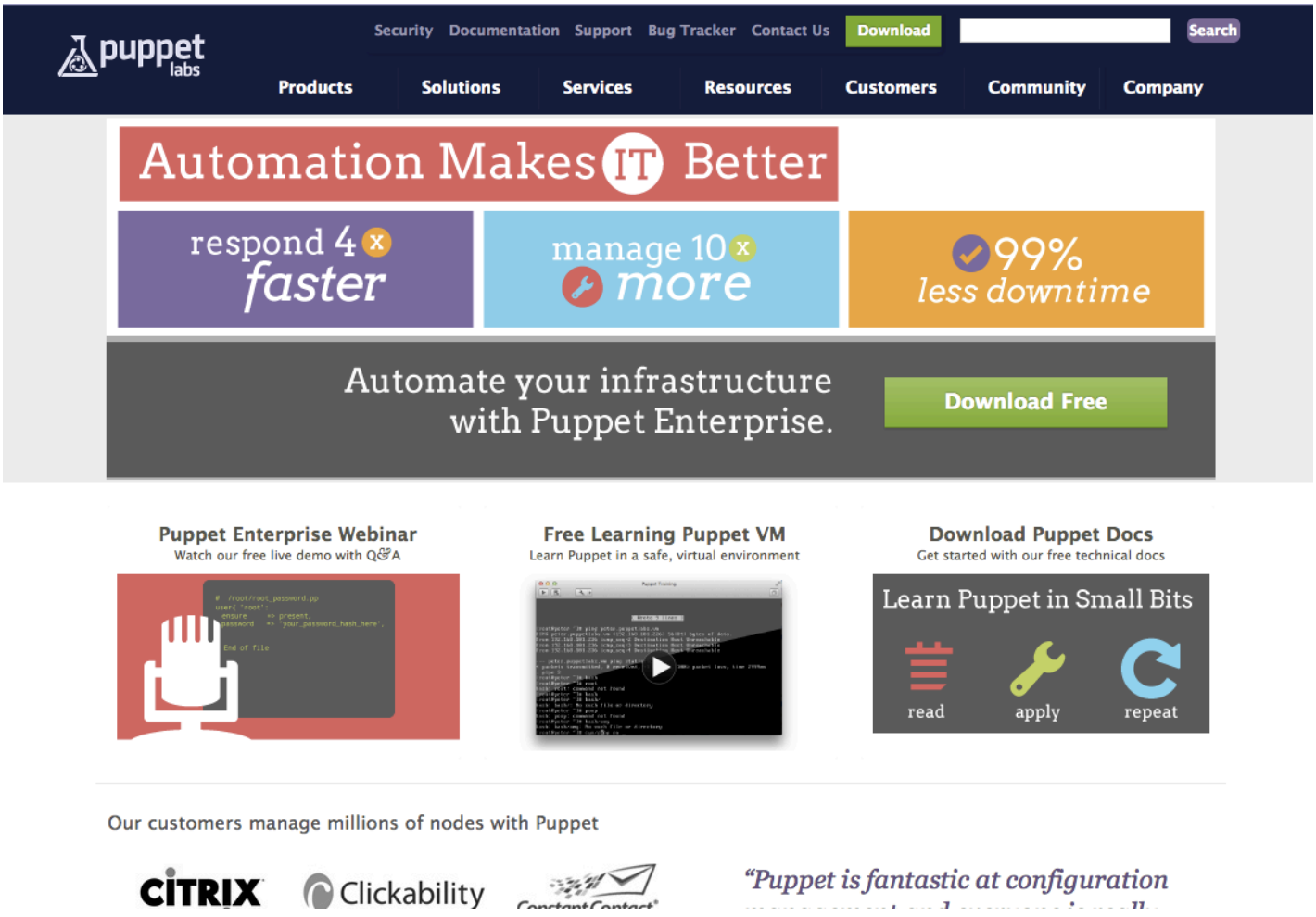

図 1.1　Puppet Labs

なぜ Puppet が必要なのか

Puppet のようなフレームワークを使わずに構成管理をする場合、主に採られる手法として、以下のふたつが挙げられるでしょう。

- 手順書による管理
- スクリプトによる自動化

いずれの手法であっても、サーバの構築当初には十分に機能するでしょう。しかし、システムのあるべき姿は、現在のような変化の激しい時代には、どんどん移り変わっていきます。手順書やスクリプトによる構築では、そのような変化に十分に対応することができません。

変化に柔軟、かつ、継続的に対応するために、Puppet のような構成管理専用のフレームワークが必要とされるのです。

手順書の問題点

手順書による構成管理の問題点として、以下が挙げられます。

1. 手作業による時間のロス
2. オペレーションミスの発生可能性
3. 時間の経過にともなう手順の変更が、往々にして手順書に反映されないことによる作業漏れ

手順書という、コンピュータによる実行が不可能な手段に頼って構成管理をする以上、上記のような問題から逃がれることはできません。手順書を捨て、本書を参考にして、すぐにでも Puppet を使い始めましょう。

スクリプトの問題点

スクリプトによる構成管理の問題点として、以下が挙げられます。

1. 書くひとによってスクリプトの品質にバラつきが出る

2. よほど注意して書かない限り、すぐにスクリプトが肥大化し破綻する。そもそもシェルスクリプト自体の保守性に問題がある
3. 構成内容に変化があった場合、これまでの状態を保ちつつ変更を加えることが難しい

コンピュータによって実行が可能という意味で、スクリプトによる構成管理は、手順書によるそれに比べるといくぶんかマシではあります。ただ、それでも上記のような問題点はつきまとってきます。ここでは特に、3 番目について見ていきましょう。

冪等性

Puppet のような構成管理フレームワークの肝となる考え方に「冪等性」というものがあります。これは、構築手順を何度実行しても、そのたびにシステムが同じ状態に構築されるということを意味します。このことは、構成管理において、非常に重要です。

前述の通り変化の激しい現在、スクリプトでやるにせよ Puppet を使うにせよ、同じ内容の手順を何度も実行することになります。スクリプトで構成管理をしていると、たとえば「あるファイルがなかったら作成する、あったら何もしない」という操作を記述するのに、いちいちファイルの存在確認を行う条件分岐を、そのたびごとに何度も書かなければなりません。

Puppet が提供する resource type(後述) のほとんどは、あらかじめ冪等性を担保することが前提に作られているので、そのようなことに気を揉むことなく、常に安全に実行できます。構成管理において冪等性を担保することが、Puppet のようなフレームワークの本質です。

manifest

Puppet の世界では、これからみなさんが書いていく設定ファイルのことを"manifest"と呼びます。正確には「設定ファイル」というよりは「システムのあるべき状態を記述したもの」というべきものです。その manifest を、Puppet 独自の宣言的言語を使って書いていきます。

なぜ XML や YAML などの設定ファイルによく用いられるようなフォーマットや、あるいは、Ruby のようなプログラミング言語を使わないのでしょうか。manifest は、先に述

べたように「システムのあるべき状態」を記述するためのものなので、XML や YAML のような汎用データ記述フォーマットも、Ruby のような汎用プログラミング言語も、その役を果たすにはあまりにも一般的に過ぎるから、という判断によるようです。

Chef との比較

昨今、Puppet と並んでよく使われている Chef の人気が高まっています。後発だけあって、比較的洗練されている感じもしますし、周辺ツールの開発も活発なようです。本書では Puppet について解説しますが、筆者としては正直なところ、Puppet でも Chef でも、構成管理がしっかりできさえすれば、使うのはどちらでもかまわないと考えます。

Chef は、Ruby によるいわゆる内部 DSL によってレシピを書くという仕様により、記述の自由度が高いというイメージを持たれているようです。ただし、以下の表の通り、ディレクトリ構成は Puppet に比べて厳格です。Puppet では、後述していく通り、ディレクトリ構成のベストプラクティスはあるものの、実際にはほとんど制限がありません。

	言語	ディレクトリ構成
Puppet	外部 DSL	比較的自由
Chef	内部 DSL(Ruby)	決まっている

このように、ひとくちに「自由度」といってもどこを見るかによって違ってきます。また、自由度が高いということは、うまくハンドリングできなかった場合、それだけ「カオス」を招く結果に陥りやすいということでもあります。読者のおかれた環境により適したフレームワークを選ぶのがよいでしょう。どちらを選んでも、結果としてやれることは同じです。

Puppet の言語

Puppet は Chef とは異なり、manifest の記述に独自の宣言的言語を用いると述べました。とはいえ、基本的な記述を行う分には、その文法は非常にシンプルで、特に難しいところはなにもありません。むしろ、Ruby が得意ではないひとからしたら、Puppet の言語の方が圧倒的に簡単でしょう。

zsh のパッケージをインストールするための記述を、Puppet と Chef と、併記してみます。

Puppet:

```
package { 'zsh':
  ensure => installed
}
```

Chef:

```
package 'zsh' do
  action :install
end
```

たいして違いませんね。Puppet は外部 DSL を採用しているため、あらたにそれを習得しなければならないのは事実です。しかし、仕様の隅々まで知るとなれば話は別ですが、実際に必要になってくる記法はそう多くはありません。

また、もし読者が Ruby に親しんでいないとすれば、Ruby 自体の記法に加えて Chef の DSL も憶えなくてはならないのですから、どちらの方が習得コストが高いかどうかは学習者の状況次第です。

本書は、manifest を書きシステムに適用しながら、最初の一歩からより実践的な設定まで、手を動かしながら学んでいくというスタイルを採っています。本書を読み進め、手を動かしているうちに、どのように書けばいいのか、自然におぼえてしまっていることでしょう。

まとめ

本章では以下のことを学びました。

- 手順書やスクリプトによる構成管理には問題がある
- 変化に対応するために、Puppet のようなツールが提供する「冪等性の担保」が必要
- Puppet と Chef とは、やれることは同じ。状況に応じて選び分ければよい

Puppet がなぜ必要なのか、その理由として「変化に対応するため」ということを明らかにしました。キーポイントになる「冪等性」については、本書を通じて何度も言及され

ることでしょう。それだけ大切だということです。

　絶えまない変化に対応していくためにこそ、Puppet のようなフレームワークが必要とされるのです。

第2章

本書の方針

本書は、Puppet 入門書としてはやや変わった構成を採っています。本章では、なぜそのような方針で構成したのかについて説明します。

本書の方針

「はじめに」で述べた通り、本書の目標は、この本を読んだ読者が Puppet の基本についてひととおり知り、オペレーションエンジニアの書いた manifest に変更を加えたり、ある程度の規模のものなら自力でいちから書けるようになったりすることです。

そのため、膨大な Puppet の機能のうち、本質的な知識にのみ焦点をあてて説明します。といっても、実践を軽んじているということではまったくなく、むしろその逆で、読了と同時、あるいは、本書を読みながら、読者が実際に Puppet を始められることを目指しています。

本書の方針は、上述の「すぐに実践できる知識についてのみ説明する」の他、以下の通り挙げられます。

1. ハンズオンを行っているかのように説明する
2. agent/master 構成を採らない
3. node 情報を管理しない
4. 必ずしも標準的なディレクトリ構成を用いない

それぞれについて説明します。

ハンズオンを行っているかのように説明する

本書は実践的な Puppet 入門を目指していますので、この本を読むみなさんには、是非とも実際に手を動かし、manifest を書きながら読み進めていっていただきたいと思います。Puppet に限らず、それが新しい技術的知識を習得する上で、一番の近道だからです。

本書は「〜というファイルを以下の内容で作成してください」「〜というコマンドを実行してください」と、順番に説明していく構成を採っていますので、その説明に従い、実際に手を動かしていくとよいでしょう。また、他のエンジニアに対して Puppet についての研修を行うといった場合の、ハンズオン資料としても使えるでしょう。

時間的な関係で実際に手を動かすのが難しい場合でも、本書で作成していく manifest を、以下のサポートページに完全な形で掲載していますので、すくなくともそれらを実際に適用しながら読み進めていくことをおすすめします。

- 本書のサポートページ: https://github.com/kentaro/puppet-book-support

manifest をどこに置き、どこで適用するか

本書は、基本的には次章で紹介する Vagrant という仮想化ツール (のサポートツール) を利用し、manifest を作成・適用していくことを前提としています。しかし、Vagrant を使わずに、既存の Linux その他の環境 (自宅サーバや VPS、クラウド環境など) で直接試したいという読者もいるでしょう。ここでは、Vagrant を使う場合と、そうでない場合とに分けて、説明します。

Vagrant を使う場合

Vagrant を使う場合は、manifest その他のファイルの作成・編集はホスト OS 上で行います (本書の作業環境の場合は Mac 上)。manifest の適用時のみ仮想ホストに SSH ログインして、`puppet` コマンドを実行します。

上記サポートページで提供されているファイル群を利用する場合は、ホスト OS 上の適当なディレクトリで、以下のように `git clone` コマンドによってダウンロードしてください。

```
$ git clone git://github.com/kentaro/puppet-book-support.git
```

次章で説明する通り、Vagrant はホスト OS と仮想ホストとの間でディレクトリを共有する機能がありますので、上記で `git clone` したホスト OS 上のファイルが、仮想ホスト上からも見える状態になります。そのため、ファイルを仮想ホストへデプロイする手間

なしに、ファイルの作成・編集はホスト OS 上で、その適用は仮想ホスト上で、といったことが可能です。

Vagrant を使わない場合

Vagrant を使わずに、ご自身の Linux その他の環境で Puppet を実行する場合は、以下のふたつがあり得ます。

1. manifest その他のファイルの作成・編集も、それらのシステムへの適用も同じ環境で行う
2. manifest その他のファイルの作成・編集はホスト OS 上で、それらの適用はホスト OS とは別の Linux その他のホストへ

1 の場合は「Vagrant を使う場合」で説明したのと同様に、適当なディレクトリに `git clone` すれば準備完了です。2 の場合は、ホスト OS 上で `git clone` してきたファイルを、`rsnyc` や `scp` コマンドなどを利用してデプロイした上で適用することになるでしょう。

いずれにせよ、Vagrant を使うほうが便利なので、筆者としては Vagrant を使うことを強くおすすめします。また、リモートホストへの manifest のデプロイ・適用については、「第 20 章 リモートホストに対して manifest を適用する」で解説します。

agent/master 構成を採らない

Puppet には「agent/master モード」と呼ばれる構成があります。agent とは、構成管理の対象となるサーバ (node と呼ばれます)、正確には、その node 上で動くクライアントプログラムです。一方 master とは、node とは独立に存在し、agent からのリクエストに応えてコンパイルされた manifest(catalog と呼ばれます) を配布する役割を担います。

Puppet Labs の公式ドキュメントに掲載されている以下の図を見ると、容易にイメージできるでしょう。

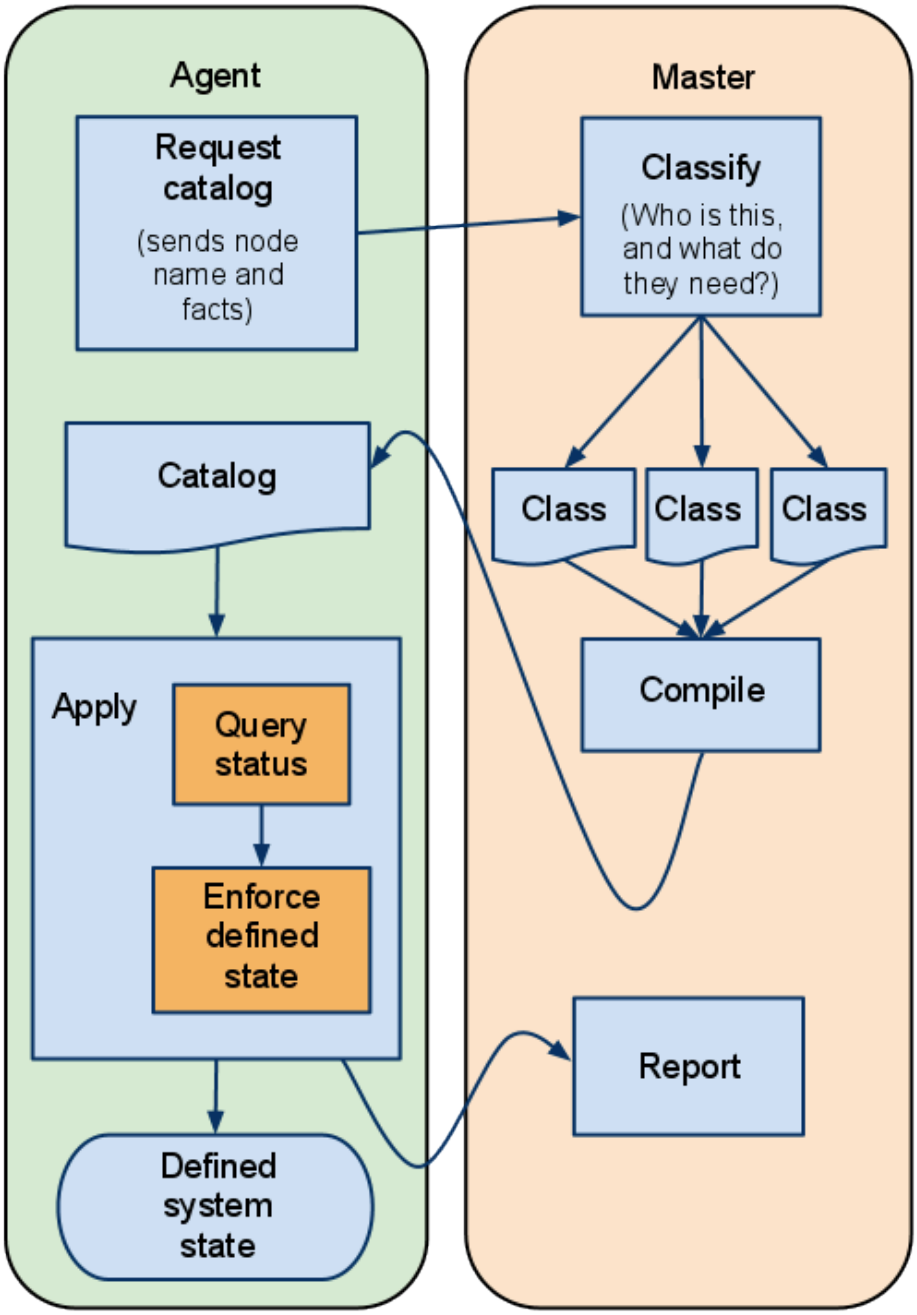

図 2.1　Puppet は agent/master からなる

本書では、この agent/master 構成を採用しません。理由は以下の通りです。

1. agent/master 構成は大規模なクラスタには有用だが、小規模なサービスの場合はなくても事足りる
2. master の設定や運用はそれなりに難しい
3. 将来、本当に必要になった時に agent/master 構成へスイッチすればいい

本書は、Puppet の manifest をそれぞれの node 上で個別に適用する方式、具体的には `puppet apply` コマンドのみを使う方法を採ります (Chef をご存知の方は、Chef Solo を想起するとわかりやすいでしょう)。まず小さな規模から実践したいという本書の目的からいって、じゅうぶん理に叶った方針であると考えます。

node 情報を管理しない

Puppet の agent/master 構成では、node をホスト名に基づいて識別し、それぞれについてどのような状態が適用されるべきかという記述を行います。具体的には、以下のような記述があったとして、

```
node 'app001.example.com' {
  # app001 用の設定
}

node 'db001.example.com' {
  # db001 用の設定
}
```

`app001.example.com` の agent と、`db001.example.com` の agent とで設定を分けることで、特定の node に対して特定の manifest を適用できます。

これは便利な機能ではある一方、以下の理由により本書では採りません。

1. agent/master 構成を採用しないので使えない
2. node 情報は別の場所で管理したい

2 については、具体的には「第 20 章 リモートホストに対して manifest を適用する」で説明する通り、capistrano で実現します。本書の説明に従って manifest を書いていけば、capistrano + `puppet apply` コマンドで、小規模環境なら十分に機能する道具立てがそろいます。

必ずしも標準的なディレクトリ構成を用いない

第 1 章で Chef との比較において述べた通り、Puppet はディレクトリ構成において、Chef より比較的自由です。そのため、manifest を気をつけて構成していかないと、保守性に乏しいものになってしまいがちです。

本書では、Puppet のベストプラクティスを適宜参照することでそうした混沌を避けつつ、保守性の高いかっちりした構成と、上記で述べてきたような実践の容易さとを両立する方法を紹介します。

まとめ

本章では「本書の方針」として以下のことを説明しました。

1. ハンズオンを行っているかのように説明する
2. agent/master 構成を採らない
3. node 情報を管理しない
4. 必ずしも標準的なディレクトリ構成を用いない

構成管理というと、大規模なクラスタのためのものというイメージが強く、Puppet ももちろんそのような要件に対応するべく開発が進められてきたのですが、それだけではもったいないというのが筆者の考えです。

本書の方針は、Puppet による構成管理の自動化と、その学習・運用コストをてんびんにかけ、両方のいいとこ取りが可能な、適切なバランスであると考えます。

第3章

Vagrant で開発環境を用意する

本書では、「はじめに」で述べたように、Puppet の実行環境として Linux(Amazon Linux と CentOS) を使用します。といっても、いますぐ手元に、自分の自由になる Linux 環境を用意できないという読者も多いでしょう。

そこで、Puppet の実践に入る前に、まずは Vagrant というツールを使って、開発環境を用意してみましょう。

Vagrant とは？

Vagrant は、VirtualBox や VMware、Amazon EC2 といった仮想化ツールを、簡単にコントロールするためのラッパツールです。元々は VirtualBox 専用に開発されたものでしたが、本書で使用するバージョン 1.1 以降は、Providers という仕組みを導入したことで、プラグインによって VirtualBox 以外の仮想化ツールにも対応するようになりました。

本書では、これまで Vagrant での利用実績が豊富な VirtualBox を使っていくことにします。

ちなみに、Vagrant には Puppet と連携できる機能があり、仮想ホストの起動時や起動後に、指定した設定をもとに manifest を適用する仕組みがあります。しかし、本書は現場で使える Puppet 入門を目指しており、より実践的な方法を採るため、その機能は使いません。

Vagrant のインストール

まずは VirtualBox をインストールしましょう。VirtualBox のダウンロードページからお使いの環境にあったパッケージをダウンロードし、インストーラの指示に従ってインストールしてください。

VirtualBox のインストールが終わったら、今度は Vagrant をインストールします。Vagrant のダウンロードページからお使いの環境にあったパッケージをダウンロードして

ください。その際、本書の環境と合わせるため、1.1.0 以上のバージョンをダウンロードするようにしてください。その後、インストーラの指示に従ってインストールします。

仮想ホストの起動

Vagrant で利用できる仮想ホストのひな形 (box といいます) は、有志により様々なディストリビューションのものが用意されています (http://www.vagrantbox.es/)。また、Puppet 提供元の Puppet Labs からも、様々なディストリビューションの box が提供されています (http://puppet-vagrant-boxes.puppetlabs.com/)。

本書では、Puppet Labs が提供する CentOS 6.4 の box を利用します。

http://puppet-vagrant-boxes.puppetlabs.com/centos-64-x64-vbox4210.box

仮想ホストを起動するための設定は、非常に簡単です。適当なディレクトリで、以下のコマンドを実行します。

```
$ vagrant init
```

すると、コマンドを実行したディレクトリに `Vagrantfile` というファイルが作成されます。中を見てみるといろいろ書かれていますが、本書の範囲内で必要とする設定はわずかです。設定の詳細については Vagrant のドキュメントを見ていただくとして、ここではまず、以下の内容に変更してください。

```
Vagrant.configure("2") do |config|
  config.vm.box      = "centos-6.4-puppet"
  config.vm.box_url  = "http://puppet-vagrant-boxes.puppetlabs.com/centos-64 ⏎
-x64-vbox4210.box"
  config.vm.hostname = "puppet-book.local"
end
```

このファイルでは、以下の設定を行っています。

- 仮想ホストに使用する box 名の指定
- box が存在しなかった場合に取得する先の URL
- 仮想ホストの hostname の指定

ファイルを作成したのと同じディレクトリで、vagrant up コマンドを実行すると、仮想ホストが起動します。

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Box 'centos-6.4-puppet' was not found. Fetching box from specified ⏎
 URL for
the provider 'virtualbox'. Note that if the URL does not have
a box for this provider, you should interrupt Vagrant now and add
the box yourself. Otherwise Vagrant will attempt to download the
full box prior to discovering this error.
Downloading with Vagrant::Downloaders::HTTP...
Downloading box: http://puppet-vagrant-boxes.puppetlabs.com/centos-64-x64-vb ⏎
ox4210.box
Extracting box...
Cleaning up downloaded box...
Successfully added box 'centos-6.4-puppet' with provider 'virtualbox'!
[default] Importing base box 'centos-6.4-puppet'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Setting hostname...
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

以下、本書の説明を通じて、vagrant コマンドは、Vagrantfile のあるディレクトリで実行してください。

初回実行時には、box をダウンロードするために時間がかかりますが、次回からは既に

ダウンロードした box を使用するため、すぐに起動します。

仮想ホストに SSH ログインする

以下のコマンドを実行すると、仮想ホストに SSH でログインできます。

```
$ vagrant ssh
```

また、のちの章では通常の ssh コマンドによるログインが必要になりますので、以下のようにコマンドを実行して、準備しておいてください。

```
$ vagrant ssh-config --host puppet-book.local >> ~/.ssh/config
```

これで、いつものように ssh コマンドで仮想ホスト (ここでは puppet-book.local というホスト名を指定) にログインできます。

```
$ ssh puppet-book.local
```

ログインしたら、適当にコマンドを実行してみたりして、触ってみてください。ふだん使っている Linux と変わらない環境が簡単にできてしまったことに、驚くことでしょう。

また、あれこれといじってみた結果、たとえ壊してしまったとしても、後述の通り簡単にリセットして元通りにできますので、安心です。

仮想ホストの停止・破棄

仮想ホストを停止するには `halt` サブコマンドを、破棄 (リセット) するには `destroy` サブコマンドを使います。

```
$ vagrant halt
```

を実行すると一時停止、

```
$ vagrant destroy
```

で破棄 (リセット) されます。再度、仮想ホストを起動したい場合は、最初と同じく

```
$ vagrant up
```

を実行してください。

Vagrant には、上記で紹介したものの他にもたくさんの便利なサブコマンドがありますが、本書の範囲内では、以上で十分です。より詳しく知りたい方は、Vagrant のドキュメントを参照するとよいでしょう。

まとめ

本章では以下のことを学びました。

- Vagrant を使って開発環境を用意する方法
- Vagrant の基本的な操作

Vagrant を使うと、仮想ホストを `vagrant up` で起動し、あれこれいじった後に気に入らなくなってきたら `vagrant destroy` でまっさらなんてことも、簡単にできてしまいます。これから Puppet でこの仮想ホストをいじり倒していくには、もってこいの機能です。

manifest を書いていく途上で、何度も仮想ホストを作り直していくことになるでしょう。言葉を変えていえば、何度作り直しても manifest さえあればすぐに元通りになるという状態を作っていくことが「あるべき状態を記述する」ということになるのです。

第4章

Hello, Puppet!

開発環境も整ったところで、さっそく Puppet を始めましょう。本章では、定番の"Hello, World!"を Puppet で実行してみます。さらに、もうすこし実践的な例として、Puppet を使ってパッケージをインストールする方法を紹介します。

まずはざっくりと、Puppet を使う最初の一歩を簡単な例から始めることにしましょう。

Puppet のインストール

実は、前章で準備した Vagrant の仮想ホストには、最初から Puppet がインストールされています。そのため、Puppet を特別にインストールする必要はありません。`vagrant ssh` で仮想ホストにログインして、確かめてみましょう。

```
$ vagrant ssh
Welcome to your Vagrant-built virtual machine.
[vagrant@puppet-book ~]$ puppet --version
3.1.1
```

本書執筆時点で最新のバージョン、3.1.1 が既に入っているのを確認できます。

以下、本書全体を通して、次の表記によりホスト OS 上と仮想ホスト上とでのコマンド実行を区別します (Vagrant を使わずに本書を読み進める場合は、適宜読み替えてください)。

- `$`のみで始まる行はホスト OS でのコマンド実行
- `[vagrant@puppet-book ~]$`のように、`[vagrant@`ホスト名 ディレクトリ名`]$`から始まる行は仮想ホストでのコマンド実行

本書の前提とする以外の、Puppet がインストールされていない環境では、以下のように `gem` コマンドによってインストールできます。

```
[vagrant@puppet-book ~]$ sudo gem install puppet --no-rdoc --no-ri
Successfully installed facter-1.6.18
Successfully installed json_pure-1.7.7
Successfully installed hiera-1.2.0
Successfully installed puppet-3.1.1
4 gems installed
```

ちなみに、Vagrant で起動した仮想ホストでは、デフォルトのログインユーザである vagrant ユーザが、パスワードの入力なしに sudo コマンドを実行できるよう設定されています。

manifest を置く場所

さて、さっそく manifest を書いていきましょう。とはいっても、どこにファイルを置けばいいのでしょうか。ここでも Vagrant がその便利さを発揮します。

仮想ホストにログインした状態で、/vagrant ディレクトリに移動し、ls コマンドを実行してみてください。

```
[vagrant@puppet-book ~]$ cd /vagrant
[vagrant@puppet-book vagrant]$ ls
Vagrantfile  puppet
```

Vagrantfile が見えますね。Vagrant は、特に指定しなくても、仮想ホストの/vagrant ディレクトリに、ホスト OS 上の Vagrantfile のあるディレクトリをマウントしてくれます。また、サンプルコードを git clone した場合は、puppet ディレクトリも見えているはずです。

これから manifest を書いていく際は、ホスト OS 上でファイルの作成・編集を行い、Puppet の実行のみを仮想ホスト上で行うことにします。そうすることで、ファイルの編集をホスト OS 上でいつも使っているエディタで行いつつ、Puppet の実行は仮想ホストで、といったことが可能になり、とても便利です。

manifest を書いてみる

まずはホスト OS 上で、manifest を置くためのディレクトリを作成しましょう。なお、このディレクトリの作成は必須ではなく、単純に、のちの章で作成するものとわけて置くほうが整理されていいだろうというだけの理由です。

```
$ mkdir -p puppet/hello_puppet
$ cd puppet/hello_puppet/
```

次に、以下の内容で、`hello_world.pp` というファイルを作成します。

```
notice("Hello, World!")
```

このように、Puppet の manifest ファイルは、`.pp` という拡張子をつけることになっています。

manifest を適用する

この manifest を適用してみましょう。今度は、仮想ホスト上で manifest ファイルのあるディレクトリに移動してから、`puppet apply` コマンドを実行します。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/hello_puppet/
[vagrant@puppet-book hello_puppet]$ puppet apply hello_world.pp
Notice: Scope(Class[main]): Hello, World!
Notice: Finished catalog run in 0.03 seconds
```

"Hello, World!"と表示されています。簡単ですね。

パッケージをインストールする

さて、"Hello, World!"の表示からもう一歩すすんで、今度は実際にシステムの状態を変更する操作を行ってみましょう。ここでは zsh を、Puppet を使ってインストールし

ます。

さきほどの hello_world.pp と同階層に、以下の内容で zsh.pp というファイルを作成してください。

```
package { 'zsh':
  ensure => installed,
}
```

今度は、新しく作成したファイルを引数にして puppet apply を実行します。今度は、ログの表示だけではなく、システムへの変更 (yum コマンドによるパッケージのインストール) も行うので、sudo 権限で実行する必要があります。

```
[vagrant@puppet-book hello_puppet]$ sudo puppet apply zsh.pp
Notice: /Stage[main]//Package[zsh]/ensure: created
Notice: Finished catalog run in 9.76 seconds
```

意図した通り、zsh パッケージがインストールされたようですね。

```
[vagrant@puppet-book hello_puppet]$ which zsh
/bin/zsh
```

ここで注目したいのは、CentOS 上でパッケージをインストールするに際して、yum コマンドのような、プラットフォーム固有のコマンドを指定していないということです。

Puppet には、RAL(Resouce Abstraction Layer) という仕組みがあり、プラットフォーム固有の事情を抽象化することで、差異を吸収してくれます。この仕組みのおかげで、どのプラットフォームに manifest を適用するかを気にすることなく、システム構成の記述という本質に注力することができるのです。

manifest の作成・適用の流れ

このように、Puppet で manifest を作成・適用する流れは、

1. 適当な場所に manifest を書く

2. manifest をシステムに適用する

という流れを繰り返していくことに他なりません。のちのちの説明では、さらに規模の大きな manifest を書いていくために様々な記述方法を説明していきますが、本質的には上記の流れをくりかえしていくだけということに変わりありません。

どんどん書いて、どんどん `apply` していきましょう。

Function

さて、今回書いてみた manifest について、その中身をもうすこしくわしく見ていきましょう。"Hello, World!"を出力する箇所は、こういう内容でしたね。

```
notice("Hello, World!")
```

この `notice` は、一般的な言語でいうところの「関数」と同じ働きをするもので、Puppet では function と呼ばれています。この `notice` のように、Puppet はあらかじめいくつかの function を提供しています。どのようなものがあるのか、その一覧については、以下のドキュメントをご参照ください。

http://docs.puppetlabs.com/references/latest/function.html

Resouce Type

zsh パッケージをインストールする箇所は、こうでした。

```
package { 'zsh':
  ensure => installed,
}
```

Puppet では、manifest によって記述する「システムのあるべき状態」を構成するひとつひとつを、resource と呼びます。ここでは `package` とそれに続く記述によって、zsh パッケージのインストールを行ってます。この場合の `package` を、resource の具体的な種類という意味で、resource type と呼びます。

resource type には、`package` 以外にも、たとえば `file` や `user` など、システムの構成要素の多様性に応じて、たくさんのものが用意されています。どのようなものがあるのか、その一覧については、以下のドキュメントをご参照ください。

http://docs.puppetlabs.com/references/latest/type.html

また、ここで`'zsh':`と記述されている箇所を title、`ensure => installed` を attributes といいます。title は、resource type を一意に決定するための名前を、attributes はどういう状態であるべきかを記述するために使われます。

まとめ

本章では以下のことを学びました。

- Puppet のインストール、バージョンの確認方法
- manifest 作成の流れ
- プログラミングにおける関数のような働きをする function
- 「システムのあるべき状態」の構成要素としての resource type

実際に Puppet の manifest を書いて、"Hello, World!"を表示してみたり、パッケージをインストールしてみたりすることで、ひととおりの manifest の作成から、システムへの適用の流れまでを体験してみました。

また、今回書いた manifest には、function や resource type という、Puppet 言語の構成要素が使われていることも学びました。このように本書では、Puppet 言語の構成要素をまとめて網羅的に解説するというよりもむしろ、必要に応じて解説していきます。その方が、記憶への定着が行われやすいだろうからです。

第5章

nginx の manifest を書く

前章では、簡単な manifest を作成し、システムへ適用することを通して、Puppet での作業の流れを体験してみました。本章では、さらに実践的な manifest の作成に挑戦してみましょう。

今回取り上げるのは、HTTP サーバの nginx です。

「nginx が使える状態」とは？

CentOS に nginx をインストールし、実際に HTTP サーバとしてユーザからのリクエストに応答できるようにするまでのステップを、まずはあらためて考えてみましょう。

1. 必要に応じて、yum パッケージを提供するリポジトリをシステムに登録する
2. `yum` コマンドを使って nginx をインストールする
3. 設定ファイルを記述する
4. nginx を起動する
5. nginx がシステム起動時に自動的に起動するよう設定する

「nginx が使える状態」であるといえるためには、最低限これぐらいのことをする必要があります。実際、みなさんがこれまで nginx をシステムにインストールしてきた際には、同様のことを行ってきたことでしょう。

manifest に落とす

次に、上記の 1〜5 を、manifest に落としていきます。まずは、本章で作成する manifest を置くためのディレクトリを作成しましょう。

```
$ cd puppet/
$ mkdir nginx
```

```
$ cd nginx/
```

以下の内容で、nginx.pp というファイルを作成してください。

```
yumrepo { 'nginx':
  descr    => 'nginx yum repository',
  baseurl  => 'http://nginx.org/packages/centos/6/$basearch/',
  enabled  => 1,
  gpgcheck => 0,
}

package { 'nginx':
  ensure  => installed,
  require => Yumrepo['nginx'],
}

$port = 80

file { '/etc/nginx/conf.d/my.conf':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('my.conf'),
  require => Package['nginx'],
  notify  => Service['nginx'],
}

$target = 'Puppet'

file { '/usr/share/nginx/html/index.html':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('index.html'),
  require => Package['nginx'],
```

```
}

service { 'nginx':
  enable     => true,
  ensure     => running,
  hasrestart => true,
  require    => File['/etc/nginx/conf.d/my.conf'],
}
```

manifest の内容

作成した manifest を、上から見ていきましょう。package については前章で紹介したので、省略します。今回はあらたに yumrepo、file、そして service という resource type が使われています。これら resource type についてはあとの章でより詳しく見ていきますので、ここでは簡単な説明にとどめます。

yumrepo は、システムへの yum リポジトリの登録状態を記述するための resource type です。ここでは nginx のインストールマニュアルに掲載されている公式の yum リポジトリを登録し、使用可能な状態にしています。

ちなみに、ここでは gpgcheck を 0 とし、チェックしないよう設定していますが、これは nginx のリポジトリが GPG キーを提供していないためです。EPEL のように GPG キーを合わせて提供しているリポジトリを利用する場合は、極力 GPG キーのチェックも行うほうがよいでしょう。詳しくは「第 7 章 yum リポジトリを登録する - yumrepo」で解説します。

file は、ファイルやディレクトリに関する resource type です。ファイルやディレクトリが、指定された attribute 通りに存在しする (あるいは存在しない) という状態を記述するために使います。owner や group などの各 attribute がどのような意味を持つかは、想像がつくでしょう。template()、および require と notify の各 Attribute については、後述します。

$port = 80 や$name = "Puppet"という箇所は、見ての通り、変数への代入を行っています。Puppet ではこのように、manifest の中で変数を使えます。後述の template() といっしょに説明します。

service は、今回の例の nginx のような、サービスを提供するデーモンのあるべき状態

を記述するための resource type です。enable は、システム起動時にサービスとして起動するかどうかを、ensure は、常に起動した状態を保っているべきかどうかを記述するのに使います。nginx のようなデーモンは常に起動しておきたいのがふつうでしょうから、このように記述します。

テンプレートを用意する

yum コマンドによってインストールされる nginx は、/etc/nginx/conf.d/*.conf をロードするよう設定されています。そこで、自分用にカスタマイズした設定を /etc/nginx/conf.d/my.conf として配置し、自動的にロードされるよう manifest を記述しています。その箇所をもう一度見てみましょう。

```
file { '/etc/nginx/conf.d/my.conf':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('my.conf'),
  require => Package['nginx'],
  notify  => Service['nginx'],
}
```

ここでは template() という function を用いて、設定ファイルの内容を生成しています。ここであえて「生成」と述べたのにはわけがあります。template() は、引数に渡されたファイルの内容を、Ruby のテンプレートエンジン erb の形式で書かれたものとして評価し、その結果として生成された文字列を返す関数だからです。

以下の内容で、my.conf というファイルを nginx.pp と同じディレクトリに作成してください。

```
server {
  listen       <%= port %>;
  server_name  localhost;

  location / {
```

```
    root  /usr/share/nginx/html;
    index index.html;
  }
}
```

この<%= port %>の箇所が、erb の変数展開の記法として評価され、manifest で$port = 80 と記述されている内容がうめこまれます。

同様に、index.html というファイルを、以下の通り作成してください。

```
Hello, <%= target %>!
```

このファイル内の<%= target %>も、同名変数$target の値に展開されます。

manifest を適用する

さて、manifest を作成したら、システムに適用してみましょう。今回は前回と違ってテンプレートも使うので、puppet apply コマンド実行時に、--templatedir オプションでカレントディレクトリ (.) を指定します。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/nginx
[vagrant@puppet-book nginx]$ sudo puppet apply --templatedir=. nginx.pp
Notice: /Stage[main]//Yumrepo[nginx]/descr: descr changed '' to 'nginx yum r⏎
epository'
Notice: /Stage[main]//Yumrepo[nginx]/baseurl: baseurl changed '' to 'http://⏎
nginx.org/packages/centos/6/$basearch/'
Notice: /Stage[main]//Yumrepo[nginx]/enabled: enabled changed '' to '1'
Notice: /Stage[main]//Yumrepo[nginx]/gpgcheck: gpgcheck changed '' to '0'Not⏎
ice: /Stage[main]//Package[nginx]/ensure: created
Notice: /Stage[main]//File[/usr/share/nginx/html/index.html]/content: conten⏎
t changed '{md5}e3eb0a1df437f3f97a64aca5952c8ea0' to '{md5}1db16ebfb21d376e5⏎
b2ae9d1eaf5b3c8'
Notice: /Stage[main]//File[/etc/nginx/conf.d/my.conf]/ensure: created
Notice: /Stage[main]//Service[nginx]/ensure: ensure changed 'stopped' to 'ru⏎
nning'
```

```
Notice: /Stage[main]//Service[nginx]: Triggered 'refresh' from 1 events
Notice: Finished catalog run in 33.69 seconds
```

エラーなく終了したら、nginx が実際に起動しているかどうか、確認してみましょう。

```
[vagrant@puppet-book nginx]$ curl http://localhost:80/
Hello, Puppet!
```

変数に代入した値がきちんと展開された文字列が、nginx によって返されました。

Relationship

さて、nginx の manifest を無事に適用できたところで、説明を先送りにした `require` と `notify` について、見ていきましょう。

今回は、`yumrepo`、`package`、`file`、`service` という、4 つの resource type を使って manifest を記述しました。それぞれの resource type によって記述された resource は、任意の順番で個別に存在するのはなく、次の順番で依存関係を持っています。

1. yum リポジトリを登録する
2. nginx パッケージをインストールする
3. 設定ファイルを配置する
4. サービスを起動する

1 から 4 までが順番通りに実行されないと、正しく nginx を起動できません。このような依存関係のことを dependency relationship と呼び、`require` によって記述します。

```
package { 'nginx':
  ensure  => installed,
  require => Yumrepo['nginx'],
}
```

これは、nginx パッケージをインストールする前に、nginx の yum リポジトリを登録する必要がある、ということを意味します。

「関係」にはもうひとつ種類があります。nginx のインストール時のみならず、nginx の設定ファイルを変更した場合にも、nginx にその設定を再読み込みさせる必要があります。このような、ある resource の変更にともなって再読み込みのようなアクションが必要となる関係を refresh relationship と呼び、`notify` によって記述します。

```
file { '/etc/nginx/conf.d/my.conf':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('my.conf'),
  require => Package['nginx'],
  notify  => Service['nginx'],
}
```

Puppet の manifest は、ファイル内の記述順で処理が実行されるわけではありません。Puppet 言語は、これまで何度も述べている通り「システムのあるべき状態」を記述するためのものですから、ファイル内での記述の順序によって実行順が変わるのは好ましくありません。今回行ったように、明示的に relationship を記述してやる必要があります。

Resource Reference

上記で、`package` を宣言する際にはすべて小文字で書いていた一方で、`require` によって依存関係を記述する際には `Package['nginx']` と、最初の 1 文字を大文字で書いていたことに気付いたでしょうか？

このように、resource の宣言時にはすべて小文字を使い、relationship のある resource への参照時には、最初の文字を大文字にした名前を使います。他の resource への参照を、resource reference と呼びます。宣言時はすべて小文字で、それ以外は最初の 1 文字を大文字にする、と憶えておいてください。

まとめ

本章では、nginx の manifest を作成することを通して、多くのことを学びました。

- `yumrepo`、`file`、`service` という、あらたな resource type
- `template()` によって erb 形式のテンプレートが評価・展開されること
- resource 間の relationship を記述すること

すこし大変だったかもしれませんが、ここで学んだことをきちんと理解しておけば、Puppet の本質の、多くの部分を把握したことになります。本章をよく読み返してみて、いまいちど理解を確かめてみてください。

第6章

パッケージをインストールする - package

前章までで manifest 作成・適用の流れをひと通りおさえたところで、本章からはしばらく、よく使う resource type についてざっと見ていきましょう。

まず最初に、パッケージのインストール状態を記述するのに使う package について説明します。

http://docs.puppetlabs.com/references/latest/type.html#package

Package

第 4 章で、zsh をインストールするのに以下のように書きました。

```
package { 'zsh':
  ensure => installed,
}
```

これだけの記述で、プラットフォームが RHEL 系なら yum コマンドが、Debian 系なら apt コマンドが自動的に選択され、パッケージのインストールが行なわれます。環境の差異にあわせ、resource の状態を実際に担保する仕組みを、provider といいます。

ensure で状態を記述する

ここでひとつ念頭に留めておきたいのは、package はパッケージをインストールするためだけに使われるものではないということです。resource type は、あくまでも resource の「状態」を記述するものであって、その「状態」には、インストールされていることの他に、「インストールされていないこと」という場合もあり得ます。

ここで、どういう状態であるべきかを指定しているのが ensure という attribute です。この場合は、installed に指定されています。他には、

- `present`: `installed` と同じ
- `absent`: インストールされていない
- `latest`: 常に最新のものがインストールされている

という状態を記述できます。

パッケージをまとめて記述したい

インストールしたいパッケージが複数ある際、そのいちいちについて前述のように書くのは面倒です。Puppet の resource type は、その名前として配列を受け取ることができるので、この場合、以下のように書けます。

```
package {
  [
    'gcc',
    'gcc-c++',
    'kernel-devel',
    'make',
  ]:
  ensure => installed,
}
```

この「配列」は、Puppet 言語でいう Array のことです。変数を使って、以下のようにも書けます。

```
$packages = [
  'gcc',
  'gcc-c++',
  'kernel-devel',
  'make',
]

package { $packages:
  ensure => installed,
}
```

また、上記のような複数のパッケージがインストールされている状態に依存する他の resource type がある場合、上記のように変数を使っておくと以下のように require を書けて、依存関係の記述が楽になります。

```
require => Package[$packages]
```

このような書き方を、condensed form と呼びます。package に限らず、全ての resource type において、同様の記法を使えます。詳しくは、以下のドキュメントを参照してください。

http://docs.puppetlabs.com/puppet/latest/reference/lang_resources.html#condensed-forms

ファイルからインストールしたい

以下のように、source でファイル名を、provider で rpm を指定します。

```
package { 'supervisor':
  ensure   => installed,
  source   => '/path/to/supervisor-3.0a12-2.el6.noarch.rpm',
  provider => 'rpm',
}
```

通常は yum でパッケージをインストールし、一部、独自のパッケージを使いたいという場合などに使えるでしょう。

まとめ

本章では以下のことを学びました。

- パッケージの状態記述のための resource type である package の使い方
- ensure で状態を保証する
- 同じような resource type をまとめて記述する condensed form
- ファイルからインストールする方法

パッケージのインストールは、システム構築の基本中の基本ですね。package は頻繁に使う resource type です。

第7章

yum リポジトリを登録する - yumrepo

今回は、OS 標準のリポジトリ以外の yum リポジトリを利用したい場合に使う yumrepo について紹介します。

http://docs.puppetlabs.com/references/latest/type.html#yumrepo

Package

第 5 章で、nginx の公式サイトで提供されている yum リポジトリを指定するため、以下のように書きました。

```
yumrepo { 'nginx':
  descr    => 'nginx yum repository',
  baseurl  => 'http://nginx.org/packages/centos/6/$basearch/',
  enabled  => 1,
  gpgcheck => 0,
}
```

この記述により、/etc/yum.repos.d/nginx.repo に、以下の内容でリポジトリが追加されます。

```
[nginx]
name=nginx yum repository
baseurl=http://nginx.org/packages/centos/6/$basearch/
enabled=1
gpgcheck=0
```

EPEL を登録する

さて今度は、RHEL の標準に含まれない追加のパッケージを提供するプロジェクト EPEL のリポジトリを登録してみましょう。

```
yumrepo { 'epel':
  descr      => 'epel repo',
  mirrorlist => 'http://mirrors.fedoraproject.org/mirrorlist?repo=epel-6&arc⏎
h=$basearch',
  enabled    => 1,
  gpgcheck   => 1,
  gpgkey     => 'https://fedoraproject.org/static/0608B895.txt',
}
```

EPEL のように、複数のミラーサーバのリストを提供しているリポジトリの場合は、nginx のリポジトリを登録した時と異なり、`mirrorlist` でそのリストの URL を指定します。

また、`gpgcheck` を 1 にし、`gpgkey` を指定することで、万が一、ミラーサーバでパッケージが書き換えられてしまうようなことがあった場合に、危険なパッケージがインストールされることのないよう、設定しておきましょう。

GPG キーのリストは、以下の URL から取得できます。

https://fedoraproject.org/keys

まとめ

本章では以下のことを学びました。

- OS 標準でない yum リポジトリを登録する方法

通常の利用には、本章の説明で十分です。より細かい制御を必要とする場合は、`man yum.conf` および yumrepo のドキュメントをご参照ください。

第8章

サービスを起動する - service

ウェブサーバやデータベースサーバなどは、インストール後「サービス」として常に起動し、クライアントからのリクエストを受け付けるという使い方が普通でしょう。またその時、OS 起動時にそれらサービスも自動的に起動するよう設定するでしょう。

くわえて、設定ファイルが変更されたり、あらたなプラグインをインストールした時など、それと同時にサービスをリスタートする必要があることもあるでしょう。

`service` はそのような、サービスに関するシステム状態を記述するための resource type です。

http://docs.puppetlabs.com/references/latest/type.html#service

Service

第 5 章で、nginx のサービス起動状態を記述するのに、以下のように書きました。

```
service { 'nginx':
  enable     => true,
  ensure     => running,
  hasrestart => true,
  require    => File['/etc/nginx/conf.d/my.conf'],
}
```

このように書くことで、

- OS 起動時に、nginx も自動的に起動する (`enable`)
- nginx が常に起動している状態を保つよう保証する (`ensure`)

という、ふたつの状態を記述したのでした。

RHEL 系 OS の場合、

- enable の状態は chkconfig コマンドで
- ensure の状態は service コマンドで

実行されます。いずれも、みなさんが手動でサービスの起動状態を変更する時に慣れ親しんだコマンドでしょう。

hasrestart について補足しておきましょう。サービスをリスタートするための方法が、サービス (この場合は nginx) の init スクリプトに用意されている場合は、上記のように hasrestart => true を指定しておきます。さもないと、Puppet はサービスの再起動を、stop と start で代用してしまいます。

もし、init スクリプト以外のなんらかのコマンドやスクリプトなどで restart したい場合は、restart => "..."として、そのコマンド/スクリプトの文字列を指定します。

notify と subscribe

いちど Puppet で構築したシステムも、時間が経つに連れ、その時々の要件に合わせて変化していきます。たとえば、今回の nginx の例でいうと、必要に応じて設定を変更するような場合です。

nginx の設定に変更を行った場合、その変更を反映させるため、再起動などが必要になるでしょう。第 4 章で書いた設定ファイルに関する記述は、以下のようになっていました。

```
file { '/etc/nginx/conf.d/my.conf':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('my.conf'),
  require => Package['nginx'],
  notify  => Service['nginx'],
}
```

ここで重要なのが notify の箇所です。もし、この設定ファイル /etc/nginx/conf.d/my.conf の状態に変更があった場合、Service['nginx'] へ re-

fresh event が通知されます。そのイベントを受け取った Service['nginx'] は、この場合、/sbin/service nginx restart を実行して、nginx を再起動します。

今回は notify を使いましたが、subscribe を使って、逆方向に関係を記述することもできます。以下のように、通知をする側ではなく通知を受け取りたい側で、subscribe により対象の resource を指定します。

```
service { 'nginx':
  enable     => true,
  ensure     => running,
  hasrestart => true,
  require    => File['/etc/nginx/conf.d/my.conf'],
  subscribe  => File['/etc/nginx/conf.d/my.conf'], #=> ここを追加
}
```

notify でも subscribe でも、結果はいずれも同じです。その時々に応じて、わかりやすい方を適宜使うとよいでしょう。

まとめ

本章では以下のことを学びました。

- サービスの起動状態を記述する resource type である service
- relationship を notify とは逆方向から指定する subscribe

ちなみに、refresh relationship は、service だけに適用されるものではありません。後述の、任意のコマンドを実行する exec にも適用されます。

第9章

ファイルやディレクトリを作成する - file

システム状態の記述に欠かせないファイルやディレクトリの操作について、本章では見ていきましょう。file という resource type を使います。

http://docs.puppetlabs.com/references/latest/type.html#file

File

第 5 章で、nginx の設定ファイルを配置するのに、以下のように書きました。

```
file { '/etc/nginx/conf.d/my.conf':
  ensure  => present,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('my.conf'),
  require => Package['nginx'],
  notify  => Service['nginx'],
}
```

この例では、nginx の大元の設定ファイルである/etc/nginx/nginx.conf からロードされることを前提に、/etc/nginx/conf.d/my.conf に自分用にカスタマイズした設定を記述しました。

owner、group、mode は見ての通りの意味ですので、詳述するまでもないでしょう。ここでは content => template('my.conf') という箇所を見ていきます。

ファイルの内容を文字列で指定する

content は、ファイルの内容を指定するために使われる attribute です。その値として、ファイルの内容を文字列で記述します。まずは、本章で作成する manifest を置くための

ディレクトリを作成しましょう。

```
$ cd puppet/
$ mkdir file
$ cd file/
```

このディレクトリで、以下の内容のファイルを、content_string.pp というファイル名で作成してください。

```
file { '/tmp/hello_puppet.txt':
  content => "Hello, Puppet!\n",
}
```

さっそく実行してみましょう。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/file
[vagrant@puppet-book file]$ sudo puppet apply content_string.pp
Notice: /Stage[main]//File[/tmp/hello_puppet.txt]/ensure: defined content as ⏎
 '{md5}570be4af90660458e6c37633d5676ec2'Notice: Finished catalog run in 0.04 ⏎
 seconds
```

cat コマンドで中身を表示してみましょう。

```
[vagrant@puppet-book file]$ cat /tmp/hello_puppet.txt
Hello, Puppet!
```

意図通りに作成されたようです。

テンプレートを使う

このように、content に文字列をわたしてやれば、任意の文字列をその内容に持つファイルを作成できるのですが、manifest にベタ書きしてしまうのは、保守性を考えるとお

すすめできるやり方ではありません。そこで、別のファイルを読み込んで content にわたす方法を見ていきましょう。

先の nginx の設定では content => template('my.conf') という指定があり、puppet apply 実行時に--templatedir=. というオプションをわたしてやることで、同ディレクトリにある my.conf というテンプレートファイルを読み込んでいました。

また、この読み込まれたファイルは、単なる文字列としてではなく、template() により erb のテンプレートとして評価されるのでしたね。

以下の内容のファイルを、content_template.pp というファイル名で作成してください。

```
$content = "Hello, Puppet!"

file { '/tmp/hello_puppet_template.txt':
  content => template('hello_puppet_template.erb'),
}
```

次に、hello_puppet_template.erb という名前で、以下の内容のテンプレートファイルを作成します。

```
<%= content %>
```

今回は、puppet apply 時に、テンプレートファイルを格納したディレクトリ (ここではカレントディレクトリ) を指定するため、--templatedir=. として、オプション引数をわたして実行します。

```
[vagrant@puppet-book file]$ sudo puppet apply --templatedir=.content_templat ⏎
e.pp
Notice: /Stage[main]//File[/tmp/hello_puppet_template.txt]/ensure: defined c ⏎
ontent as '{md5}6f459c8c8efb17c22040ffd76a4335d6'
Notice: Finished catalog run in 0.04 seconds
```

cat コマンドで中身を表示してみましょう。

```
[vagrant@puppet-book file]$ cat /tmp/hello_puppet_template.txt
Hello, Puppet!
```

template() によって erb として評価された文字列が content の値としてわたされ、意図通りにファイルが作成されました。

変数展開

template() は、引数としてわたされたファイル名を erb のテンプレートファイルとして評価すると述べました。その際、どのような変数が erb のコンテキストで参照できるのか、すこし見ていきましょう。以下の 3 つがあります。

1. トップスコープの変数
2. 現在のローカルスコープ内の変数
3. 他のローカルスコープ内の変数

今回、ファイルの中身となる文字列を格納した$content という変数は、1 のトップスコープの変数です。

1 と 2 の変数については、テンプレート内で以下のように$なしで参照できます。

```
<%= content %>
```

変数のスコープについては、「第 18 章 最低限必要な Puppet 言語の構文を学ぶ」で説明します。ここでは、トップスコープの変数は erb テンプレートの中からも参照できるということだけ憶えておいてください。

erb の詳細については、以下の記事が簡便です。

http://jp.rubyist.net/magazine/?0017-BundledLibraries

ただし、Puppet で使う分には複雑なロジックを書くことはないでしょうので、<%= . . . %>による変数展開の記法程度を憶えておくところから始めるので十分です。むしろ、テンプレートで複雑なロジックを書くのは、保守性を損うことになりますので、おすすめできません。

source

本章では、content と、その中身を template() で生成するという説明しかしてきませんでしたが、ファイルの中身を別ファイルから取得し、配置する方法は他にもあります。source という attribute を使う方法です。

template() のように、erb を使って変数展開するまでもない、どのような状況でも内容が一定で変わらないファイルもあり得ます。そこで source を使うと、その値として指定した URI(puppet:///path/to/file のような) から、そのままファイルをコピーしてくることができます。

しかし、本書では source を用いることはしません。

1. テンプレートは、puppet apply 時に--templatedir オプションで格納場所を指定できるため、取り扱いが容易である
2. 変数展開をするかしないかに関わらず、content => template('...') で統一的に扱う方が見通しがいい

という理由によります。特に 2 について、最初の構築時には source で用が足りると判断しても、あとになって変数展開が必要になることはあり得ることです。最初から content => template('...') にしておくと、そのような場合も容易に対応可能です。

ディレクトリとシンボリックリンク

ensure の値を変更することで、ディレクトリやシンボリックリンクの状態を記述することが可能です。

ディレクトリの場合、ensure に directory を指定します。

```
file { '/etc/nginx/site-available':
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}
```

```
file { '/etc/nginx/site-enabled':
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}
```

シンボリックリンクの場合、ensure に link を、target にリンク元を指定します。

```
file { '/etc/nginx/site-available/mysite.conf':
  ensure => present,
  owner  => 'root',
  group  => 'root',
  mode   => '0644',
}

file { '/etc/nginx/site-enabled/mysite.conf':
  ensure => link,
  target => '/etc/nginx/site-available/mysite.conf',
  owner  => 'root',
  group  => 'root',
  mode   => '0644',
  require => File['/etc/nginx/site-available/mysite.conf'],
}
```

以下のように、シンボリックリンクが作成できました。

```
[vagrant@puppet-book file]$ ls -la /etc/nginx/site-enabled/
total 8
drwxr-xr-x 2 root root 4096 Apr  7 07:18 .
drwxr-xr-x 5 root root 4096 Apr  7 07:16 ..
lrwxrwxrwx 1 root root   37 Apr  7 07:18 mysite.conf -> /etc/nginx/site-avai ⏎
lable/mysite.conf
```

まとめ

本章では以下のことを学びました。

- `file` でファイルの状態を記述する
- `template()` で erb テンプレートを評価し、変数を埋め込む
- 通常のファイルの他、ディレクトリやシンボリックリンクの状態を記述する

`file` は、数ある Puppet の resource type でも、最も活用頻度の高いものでしょう。きっちり押さえておきましょう。

第10章

ユーザやグループを作成する - user/group

ユーザやグループの状態を記述するには、それぞれ user と group という resource type を使います。

- http://docs.puppetlabs.com/references/latest/type.html#user
- http://docs.puppetlabs.com/references/latest/type.html#group

User

さっそくユーザを作成してみましょう。例によって、今回の manifest 用のディレクトリを用意します。

```
$ cd puppet/
$ mkdir user_group
$ cd user_group/
```

以下の内容で、user.pp というファイルを作成してください。

```
user { 'kentaro':
  ensure     => present,
  comment    => 'kentaro',
  home       => '/home/kentaro',
  managehome => true,
  shell      => '/bin/bash',
}
```

pupply appet で適用してみましょう。

```
[vagrant@puppet-book user_group]$ sudo puppet apply user.pp
Notice: /Stage[main]//User[kentaro]/ensure: created
Notice: Finished catalog run in 0.11 seconds
```

以下の通りユーザが作成されたのが確認できます。

```
[vagrant@puppet-book user_group]$ cat /etc/passwd | grep kentaro
kentaro:x:502:503:kentaro:/home/kentaro:/bin/bash
```

managehome

ここでは attribute のうち managehome について補足することにします。他の attribute については、見ての通りの意味で、自明でしょう。詳細はドキュメントをご覧ください。

home では、単にそのユーザのホームディレクトリのパスを指定するだけですが、それに加えて managehome => true としておくと、ユーザの作成時にはそのディレクトリも同時に作成し、ユーザの削除時 (ensure => absent) にはそのホームディレクトリも同時に削除します。ホームディレクトリが実際に必要な場合は、managehome => true も指定しておくとよいでしょう。

Group

グループの状態記述には group を使います。以下の内容で、group.pp というファイルを作成してください。

```
group { 'developers':
  ensure => present,
  gid    => 999,
}
```

適用してみましょう。

```
[vagrant@puppet-book user_group]$ sudo puppet apply group.pp
Notice: /Stage[main]//Group[developers]/ensure: created
Notice: Finished catalog run in 0.07 seconds
```

以下の通りグループが作成されたのが確認できます。

```
[vagrant@puppet-book user_group]$ cat /etc/group | grep developers
developers:x:999:
```

ユーザをグループに追加する

ユーザを作成し、かつ、あるグループに所属させたいとします。その場合、gid にグループ名、あるいは、グループ ID を指定することになるのですが、Puppet はそのグループを定義した resource を、自動的に user の定義への依存関係にあるものとして扱います (autorequire と呼ばれる機能です)。

以下の内容で、add_user_to_group.pp というファイルを作成してください。

```
user { 'antipop':
  ensure     => present,
  gid        => 'guest',
  comment    => 'antipop',
  home       => '/home/antipop',
  managehome => true,
  shell      => '/bin/bash',
}

group { 'guest':
  ensure => present,
  gid    => 1000,
}
```

上記では、antipop というユーザを作成し、かつ、guest というグループに追加された状態を記述しています。manifest を適用してみましょう。

```
[vagrant@puppet-book user_group]$ sudo puppet apply add_user_to_group.pp
Notice: /Stage[main]//Group[guest]/ensure: created
Notice: /Stage[main]//User[antipop]/ensure: created
Notice: Finished catalog run in 0.11 seconds
```

確認してみましょう。

```
[vagrant@puppet-book user_group]$ cat /etc/passwd | grep antipop
antipop:x:502:1000:antipop:/home/antipop:/bin/bash
```

グループ ID が 1000 で、ちゃんと追加されていますね。

まとめ

本章では以下のことを学びました。

- user を使ってユーザの状態を記述する
- group を使ってグループの状態を定義する
- ユーザをグループに追加する方法

ユーザやグループもまた、システム状態の記述に欠かせない resource です。本章で学んだ知識は、頻繁に活用することになるでしょう。

第11章

任意のコマンドを実行する - exec

Puppet はドキュメントにある通り、たくさんの resource type を標準で用意していますが、時にはそれでも足りないこともあります。exec を使うと、それら標準の resource type だけではできないことが、任意のコマンドを実行することで可能となります。

http://docs.puppetlabs.com/references/latest/type.html#exec

Exec

ここでは、xbuild という、様々な言語のビルドスクリプトをラップし、それらの言語を簡単にビルドできるようにしたツールを使うことを想定して、manifest を書いてみましょう。

今回の manifest 用のディレクトリを用意します。

```
$ cd puppet/
$ mkdir exec
$ cd exec/
```

以下の内容で、xbuild.pp というファイルを作成してください。xbuild を git clone した上で、Ruby のバージョン 2.0.0-p0 をビルドします。

```
package { 'git': }

exec { 'xbuild':
  user    => 'vagrant',
  cwd     => '/home/vagrant',
  path    => ['/usr/bin'],
  command => 'git clone git://github.com/tagomoris/xbuild.git local/xbuild',
  creates => '/home/vagrant/local/xbuild',
  require => Package['git'],
```

```
}

exec { 'xbuild ruby':
  user        => 'vagrant',
  cwd         => '/home/vagrant',
  environment => ['USER=vagrant'],
  path        => ['/bin', '/usr/bin', '/home/vagrant/local/xbuild'],
  command     => 'ruby-install 2.0.0-p0 /home/vagrant/local/ruby-2.0.0-p0',
  creates     => '/home/vagrant/local/ruby-2.0.0-p0',
  timeout     => 0,
  require     => Exec['xbuild'],
}
```

適用してみましょう。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/exec/
[vagrant@puppet-book exec]$ sudo puppet apply xbuild.pp
Notice: /Stage[main]//Exec[xbuild]/returns: executed successfully
Notice: /Stage[main]//Exec[xbuild ruby]/returns: executed successfully
Notice: Finished catalog run in 800.58 seconds
```

以下の通り、指定したバージョンの Ruby がインストールされたことが確認できます。

```
[vagrant@puppet-book exec]$ /home/vagrant/local/ruby-2.0.0-p0/bin/ruby -v
ruby 2.0.0p0 (2013-02-24 revision 39474) [x86_64-linux]
```

コマンドの実行ユーザ・グループ

コマンドは、user で指定したユーザ、あるいは、group で指定したグループの権限で実行されます。いずれも指定されていない場合は、root ユーザの権限で実行されます。状況に応じて、適切に指定するようにしてください。

また、cwd で指定したディレクトリをカレントディレクトリとしてコマンドが実行されます。

environment と path

exec では、環境変数やコマンドのサーチパスが空の状態でコマンドが実行されます。そのため、environment で適切な環境変数を指定し、path でサーチパスを指定する、あるいは、コマンドをフルパスで書く必要があります。

environment は、以下のように配列で指定します。

```
environment => ['USER=vagrant'],
```

path は、以下のように配列で指定しても、

```
path => ['/bin', '/usr/bin', '/home/vagrant/local/xbuild'],
```

:で区切った文字列として指定しても、同じように扱われます。

```
path => '/bin:/usr/bin:/home/vagrant/local/xbuild',
```

environment で環境変数 PATH を指定した場合、path で指定した内容が上書きされます。注意してください。

冪等性を担保する

exec はなんでもできる反面、冪等性を自分で担保しなければなりません。すなわち、上記の manifest を再度実行した場合、

```
[vagrant@puppet-book exec]$ sudo puppet apply xbuild.pp
Notice: Finished catalog run in 0.54 seconds
```

と、再度実行されない状態にしなければなりません。下記の方法のいずれかによって、必ず冪等性を担保するようにしましょう。

実行するコマンドが、なんらかのファイルやディレクトリを作成するようなものである場合、create でそのファイル/ディレクトリを指定することで、もしそれらが存在する場合は実行しないことにより、冪等性を担保できます。上記の xbuild の例では、以下のように書きました。

```
creates => '/home/vagrant/local/xbuild',
```

その他、onlyif や unless によって、コマンドを実行し、その終了ステータスを見ることで、冪等性を担保する方法もあります。

- onlyif: 指定したコマンドが終了ステータス 0 を返した場合に限って実行する
- unless: 指定したコマンドが終了ステータス 0 以外を返した場合に限って実行する

上記の create は、以下のように unless を使って置きかえられます。

```
unless => 'test -d /home/vagrant/local/xbuild'
```

まとめ

本章では以下のことを学びました。

- 任意のコマンドを実行することのできる exec の使い方
- creates、onlyif、unless を使って冪等性を担保する方法

exec は便利な反面、冪等性を自分で担保しなければならない、扱いの難しい resource type です。

manifest を書いていて、exec を多用しているなと感じたら、たいていは manifest の書き方に問題があります。exec を使わざるを得ない場合でも、「第 17 章 manifest の共通部分をくくりだす」で解説する defined type でラップして、できるだけ生で exec を使わないようにする方がよいでしょう。

第12章

td-agent の manifest を書く

前章まででひと通りの resource type を見てきたところで、本章からはいくつかの章にまたがって、さらに実践的な manifest を書いていきます。取り上げる題材は、td-agent です。

td-agent は、Treasure Data の提供するログ解析基盤へと、ログを収集・集約するためのツールです。そのコアをなす fluentd に、Treasure Data のクライアントとしての機能などをパッケージングし、提供するものです。

なぜ td-agent なのか

なぜ td-agent を題材に取り上げるのでしょうか。以下の理由によります。

1. 構成が比較的シンプルであること
2. 複数台のホストをまたがった構成が通常であること

我々はこれまで、Puppet の基本的な要素を見てきただけですので、あまりにも複雑な manifest に取り組むのはまだ早いでしょう。そこで、比較的構成がシンプルな td-agent を題材に選びました。とはいえ、今後、manifest をどんどん書いていけるようになるためには、実戦でそのまま使えるレベルのものを書いてみなければなりません。

td-agent は、ログ収集をする agent と、ログ集約をする agent を分けて管理するのが一般的な構成です。具体的には下図のような構成となります。

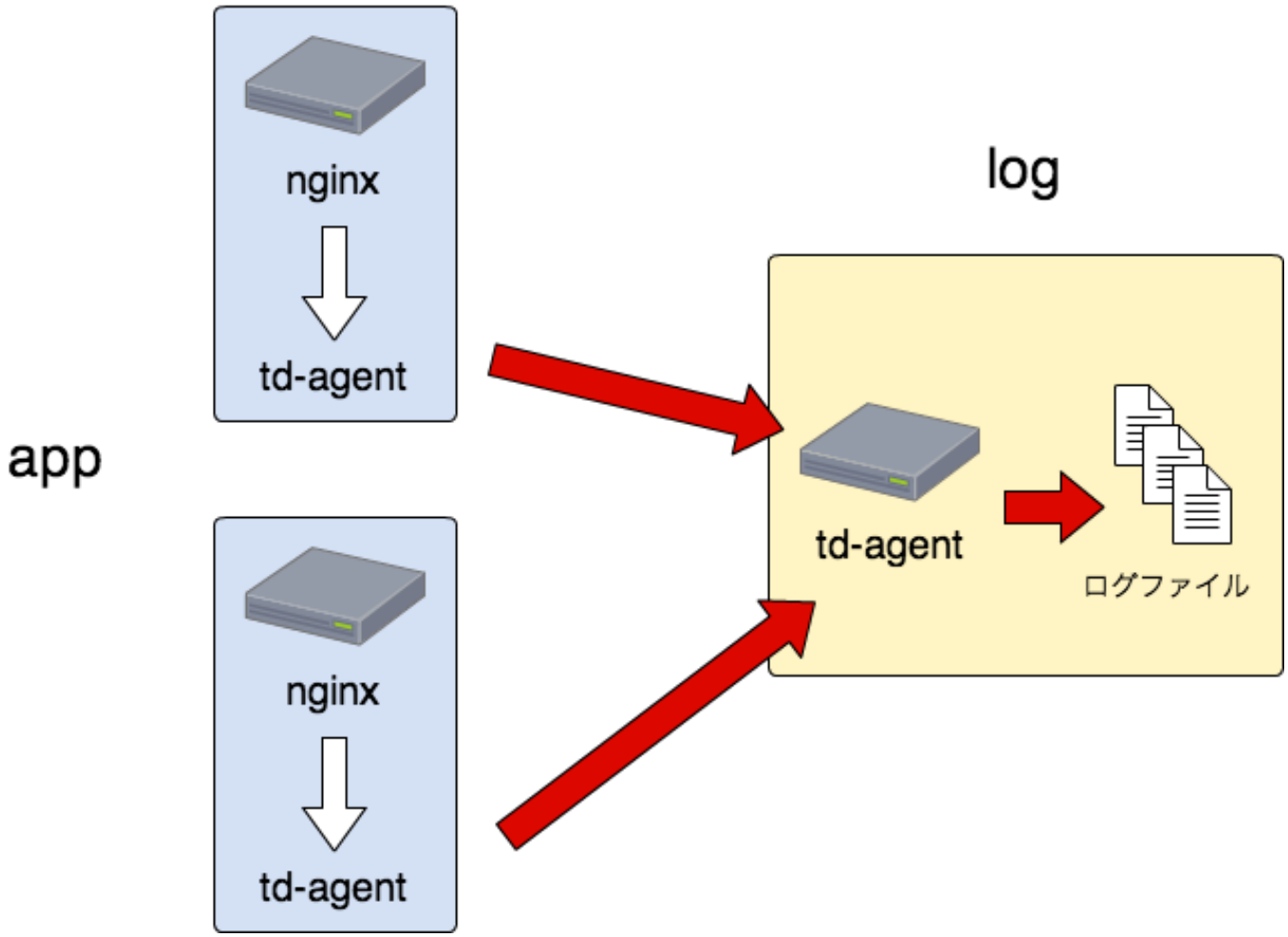

図 12.1　td-agent の構成図

あるひとつのサービスを提供するに際して、ただひとつのサーバのみを構築するというよりも、複数のサーバを構築し、システムを構成することがほとんどでしょう。その意味でも、今回の td-agent による例は、実践的です。

なにを学ぶのか

実践的な manifest を書くには、これまでに学んできた簡単な文法、resource type だけではすこし足りません。システム全体の状態を記述する大規模な manifest を書くには、システムを要素に分解し、わかりやすく構造化して記述していく必要があります。そのために、Puppet には `class` と `module` という仕組みがあります。

システム構成が複雑になり、manifest が大規模化してくると、変更を正しく加えることが難しくなってきます。ある箇所を変更するとそれに依存する別の箇所にも影響が及び、

そのことで正しくシステムが構成されないということが起こり得ます。そこで、システム記述にもプログラミングにおける「テスト」の概念を持ち込み、安全に manifest を書いていく方法を学びます。

先述の通り、あるシステムは複数のホストからなります。また、それらのホストは、様々な役割 (ロール) を持つものとしてグルーピングして管理されることが一般的です。Puppet においてそのような役割による管理を実現する方法や、また、複数のホストに manifest を適用する方法についても、td-agent の manifest を通して学んでいきましょう。

まとめ

Puppet についての基本的な知識を学習し終えたところで、より実践的な manifest を書くべく、本章では、今後数章をかけて学んでいくことになる内容について、ざっと紹介しました。

第13章

resource type のグルーピング - class

これまでの説明では、例として取り上げる題材が小さかったこともあって、manifest を上から下へと、ただそのまま羅列して書いてきました。小さな例ではそれでよいのですが、なにも策を講じなかった場合、manifest が肥大化し、保守性に問題が生じることは容易に想像がつくことでしょう。

そこで、本章では resource type を任意の単位にグルーピングするのに使われる class について見ていくことにしましょう。

Puppet における class とは

「クラス」と聞くと、プログラミング言語に親しんでいる読者は、OOP におけるクラスのようなものを想像することでしょう。Puppet の言語はプログラミング言語ではなく、これまで何度も述べている通り「システムのあるべき状態」を記述するためのものですから、そのような意味でのクラスではありません。

Puppet における class は、システムを構成する各種 resource をグルーピングし、より構造化された記述のために使われるものです。そのことにより、manifest の規模が大きくなってきても、保守性を損なうことなくシステム記述が可能となります。

Puppet の class はプログラミング言語のクラスとは違うと述べましたが、このように、分類・整理のために使われるという意味では、似ているともいえるかもしれません。

class 化の粒度

我々は現在、td-agent の manifest に取り組んでいるのでした。ここで、システムにおける td-agent のあるべき状態を考えてみましょう。たとえば、以下のように考えられます。

1. システムに td-agent をインストールする
2. 設定ファイルを準備する

3. td-agent が確実に起動するよう保証する

class 化の粒度は、上記の項目それぞれに対応させておく程度でよいでしょう。

プログラミング言語におけるクラスの粒度にこれといった決まりがなく、設計者の考え次第であるのと同様に、Puppet における class の粒度にも特に決まりはありません。だからといって好きにやってよいというものでないのも、プログラミング言語同様です。

このように、あるべき状態を上述のように分割して考え、その項目それぞれを class に対応させていくと、うまく設計できるでしょう。

td-agent の class を定義する

さっそく class を定義していきましょう。上記では 3 種類の項目を挙げましたが、その前に、それらの関係性を整理するための元締めとでもいうべき class を定義することにします。まずは、td-agent という class を定義します。さっそく書いていきましょう。

今回の manifest 用のディレクトリを用意します。

```
$ cd puppet/
$ mkdir class
$ cd class/
```

以下の内容で、`td-agent.pp` というファイルを作成してください。

```
class td-agent {
  include td-agent::install
  include td-agent::config
  include td-agent::service

     Class['td-agent::install']
  -> Class['td-agent::config']
  ~> Class['td-agent::service']
}

class td-agent::install {
  yumrepo { 'treasuredata':
```

```
    name     => 'treasuredata',
    descr    => 'treasuredata repo',
    baseurl  => 'http://packages.treasure-data.com/redhat/$basearch/',
    enabled  => 1,
    gpgcheck => 0,
  }

  package { 'td-agent':
    ensure  => installed,
    require => Yumrepo['treasuredata'],
  }
}

class td-agent::config {
  file { '/etc/td-agent/td-agent.conf':
    content => template("td-agent.conf"),
  }
}

class td-agent::service {
  service { 'td-agent':
    enable     => true,
    ensure     => running,
    hasrestart => true,
  }
}

include td-agent
```

また、以下の内容で、同じディレクトリに td-agent.conf というファイルを作成してください。

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

<match debug.**>
```

```
  type stdout
</match>
```

見慣れない記法がいくつか登場しています。順番に見ていきましょう。

class 定義の方法

最初に、td-agent の class を定義している箇所について説明しましょう。class を定義するには、以下のように `class` キーワードを使います。

```
class td-agent {
  # ここに内容を書く
}

class td-agent::install {
  # ここに内容を書く
}
```

簡単ですね。また、`td-agent::install` のように、class 名に`::`を用いて階層化して表現できます。

Puppet の class には「継承」の機能もあるのですが、本書では扱いません。以下の理由によります。

- すくなくとも、本書の範囲では必要ない
- より一般的にいって、継承よりも後述の `include` による「合成」の方が class の拡張として適切

継承よりも合成を、というのは、プログラミング言語でもよく言及される、よりよいプラクティスでもあります。

class を `include` する

次に目につくのは `include` です。これは、先に挙げた td-agent の状態記述に必要な 3 つの項目のそれぞれに対応する class を読み込むための記述です。

```
include td-agent::install
include td-agent::config
include td-agent::service
```

後述の依存関係の記述のために、td-agent というトップレベルの class 内であらかじめ全て読み込んでおきます。

class 間の依存関係

先に挙げた td-agent の状態記述に必要な 3 つの項目には、上から順番に関係があります。その順番にシステム状態が遷移していかなければ、td-agent が正しく起動する状態にはなり得ないわけです。

1. システムに td-agent をインストールする
2. 設定ファイルを準備する
3. td-agent が確実に起動するよう保証する

このような依存関係を、これまでは require や notify/subscribe によって記述してきました。今回は、->(dependency relationship) や~>(refresh relationship) といった記法を用いて、関係を記述しています。

class を使って構造化を進めていくと、個々の resource type 間の関係というよりもむしろ、class 間の関係といったものを記述する必要がでてきます。そうした場合に、あらかじめ必要な class を include した上で、->や~>を用いて関係を記述すると、一箇所でまとめて関係を記述できて便利です。

```
   Class['td-agent::install']
-> Class['td-agent::config']
~> Class['td-agent::service']
```

class 内部の resource type 間の関係はこれまで通り require や notify/subscribe で、class 間の関係は一箇所で class をまとめて include した上で->や~>で記述していくと、見通しのよい記述になるでしょう。

manifest を適用する

以上のように class を定義した上で、td-agent.pp の最後の行で include td-agent として大元の class を読み込み、適用するようにしています。このようにして、td-agent の各種状態に関する class 定義ができたところで、この manifest を適用してみましょう。

--templatedir=. を忘れずに引数にわたして、例によって puppet apply を実行します。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/class/
[vagrant@puppet-book class]$ sudo puppet apply --templatedir=. td-agent.pp
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/descr: descr ch ⏎
anged '' to 'treasuredata repo'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/baseurl: baseur ⏎
l changed '' to 'http://packages.treasure-data.com/redhat/$basearch/'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/enabled: enable ⏎
d changed '' to '1'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/gpgcheck: gpgch ⏎
eck changed '' to '0'
Notice: /Stage[main]/Td-agent::Install/Package[td-agent]/ensure: created
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/td-agent.conf]/cont ⏎
ent: content changed '{md5}c61a851e347734f4500a9f7f373eed7f' to '{md5}344838 ⏎
bf3c7825824ab99ded78dff244'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]/ensure: ensure chan ⏎
ged 'stopped' to 'running'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh ⏎
' from 1 events
Notice: Finished catalog run in 99.17 seconds
```

td-agent のインストール、設定ファイルの配置、サービス起動まで、うまくいっているようですね。

```
[vagrant@puppet-book class]$ sudo service td-agent status
td-agent (pid  6651) is running...
```

動作確認をする

ここでもう一歩ふみこんで、本当に意図通りに動いているかどうか、確かめてみましょう。fluent-cat コマンドを使うと、任意のメッセージを fluentd に対して送信できます。以下の例では、{"hello":"puppet"}という JSON 文字列を、debug.test というタグをつけて、fluentd に送信します。

```
[vagrant@puppet-book class]$ echo '{"hello":"puppet"}' | /usr/lib64/fluent/r ⏎
uby/bin/fluent-cat debug.test
```

ログファイルをのぞいてみましょう。

```
[vagrant@puppet-book class]$ cat /var/log/td-agent/td-agent.log
2013-04-14 06:06:42 +0000 [info]: starting fluentd-0.10.33
2013-04-14 06:06:42 +0000 [info]: reading config file path="/etc/td-agent/td ⏎
-agent.conf"
2013-04-14 06:06:42 +0000 [info]: using configuration file: <ROOT>
  <source>
    type forward
  </source>
  <match debug.**>
    type stdout
  </match>
</ROOT>
2013-04-14 06:06:42 +0000 [info]: adding source type="tcp"
2013-04-14 06:06:42 +0000 [info]: adding match pattern="debug.**" type="stdo ⏎
ut"
2013-04-14 06:06:42 +0000 [info]: listening fluent socket on 0.0.0.0:24224
2013-04-14 06:06:47 +0000 debug.test: {"hello":"puppet"}
```

さきほどの manifest 適用時に fluentd が起動したログとあわせて、一番下に、メッセージを受信し、ログに記録している様子が見られます。manifest がきちんと適用され、fluentd が意図通りに動いている様子も確認できました。

まとめ

本章では以下のことを学びました。

- class を使って、システムを構成する各種 resource をグルーピングし構造化する
- `include` で class をロードできる
- class 間の relationship の記述方法

これまで manifest を書く際にしてきたように、システム状態を適切に分割し、それらの関係を考えるところまでは同じです。今回は、分割した各パーツをそれぞれ class にマッピングするというところだけが異なる点でした。

第14章

manifest に関連するファイルをまとめる - module

前章では、manifest の内容を構造化する方法としての class について学びました。本章では、内容に加えて、ファイルとしても manifest を分割し、さらにわかりやすく構造化する方法としての module について見ていきます。

Puppet における module とは

module とは、class の含まれる様々なファイルや、それらの class が必要とするテンプレートなどをひとまとめに管理するための仕組みです。class が manifest の内容を分類・整理するためのものであった一方で、module は上記のファイル群を分類・整理します。

module は、再利用が可能な単位でまとめるのが一般的です。そのような、特定のシステムに限らず、広く一般的に使える module が、有志により Puppet Forge で公開されています。本章を読み、module の作成方法を学んだら、興味をひいた module を眺めてみてください。今後 module を作成する際に、おおいに参考になるでしょう。

td-agent の module を作成する

module の詳細に入る前に、実際に内容を見る方が理解がはやいでしょうから、まずはさっそく作成していきましょう。

今回の td-agent の例では、td-agent を利用するに際して、最低限これだけは必要になるだろうという範囲で、ひとつの module として構成します。前回作成した `td-agent.pp` の内容は、ほぼそれに相当するといえるでしょう。

今回作成する module 用のディレクトリを用意します。

```
$ cd puppet/
$ mkdir modules
$ cd modules/
```

まずは、module 名と同名のディレクトリを作成しましょう。

```
$ mkdir td-agent
```

次に、manifest とテンプレートを格納するディレクトリを作成します。それぞれ、`manifests` と `templates` という名前にする必要があります。

```
$ mkdir td-agent/manifests
$ mkdir td-agent/templates
```

前回作成した `td-agent.pp` には、4 つの class が定義されていました。これをそれぞれ、下記のようにファイルに分割していきます。4 つの class のうち `td-agent` だけは特別で、`init.pp` というファイル名にしてください。

td-agent/manifests/init.pp:

```
class td-agent {
  include td-agent::install
  include td-agent::config
  include td-agent::service

     Class['td-agent::install']
  -> Class['td-agent::config']
  ~> Class['td-agent::service']
}
```

td-agent/manifests/install.pp:

```
class td-agent::install {
  yumrepo { 'treasuredata':
    name     => 'treasuredata',
    descr    => 'treasuredata repo',
    baseurl  => 'http://packages.treasure-data.com/redhat/$basearch/',
    enabled  => 1,
    gpgcheck => 0,
  }

  package { 'td-agent':
    ensure  => installed,
    require => Yumrepo['treasuredata'],
  }
}
```

td-agent/manifests/config.pp:

```
class td-agent::config {
  file { '/etc/td-agent/td-agent.conf':
    content => template("td-agent/td-agent.conf"),
  }



  file { '/etc/td-agent/conf.d':
    ensure => directory,
  }
}
```

td-agent/manifests/service.pp:

```
class td-agent::service {
  service { 'td-agent':
    enable     => true,
    ensure     => running,
    hasrestart => true,
  }
```

```
}
```

また、td-agent.conf は templates ディレクトリ以下に配置します。内容は前回とは異なり、以下のようにします。サーバの用途ごとに異なる設定を conf.d/以下に配置することを前提に、用途を問わず共通化しておくべき設定を提供します。

```
# conf.d 以下に用途ごとに設定を分けて置く
include conf.d/*.conf

# デバッグ用ログ
<match debug.**>
  type stdout
</match>

# fluentd の内部イベントログ
<match fluent.**>
  type file
  path /var/log/td-agent/fluent.log
</match>

# どのタグにもマッチしなかったログ
<match **>
  type file
  path /var/log/td-agent/no_match.log
</match>

# drb 経由で接続してデバッグするための設定
<source>
  type debug_agent
  port 24230
</source>
```

以上の作業を終えたら、以下の通りのディレクトリ/ファイル構成になっているはずです (tree コマンドは、MacOSX の場合、homebrew によってインストールできます)。

```
$ tree puppet/modules/puppet/modules/
└── td-agent
    ├── manifests
    │   ├── config.pp
    │   ├── init.pp
    │   ├── install.pp
    │   └── service.pp
    └── templates
        └── td-agent.conf
```

これで、td-agent の module が作成できました。

module のレイアウト

module は、前述の通り、ファイル群を分類・整理するための仕組みです。そのため、ディレクトリ名やファイル名に、決まったルールがあります。

まずはディレクトリ名のルールについて説明します。

- module のディレクトリ名を module と同名にする
- manifest を配置するディレクトリ名は `manifests` にする
- template を配置するディレクトリ名は `templates` にする

他にも module に含められる種類のディレクトリはあるのですが、本書の範囲を超えますので、説明しません。興味のある読者は、以下のドキュメントを参照してください。

http://docs.puppetlabs.com/puppet/latest/reference/modules_fundamentals.html

次に、manifest について見ていきましょう。

上記で、`td-agent.pp` に含まれている 4 つの class をファイルに分割しましたが、`td-agent` だけは `init.pp` というファイル名にした他は、それぞれ class 名の::以降の部分をファイル名とし、`manifests` ディレクトリ以下に配置しました。

- module と同名の class については `init.pp` というファイル名にする
- その他の class についてはモジュール名::以降をファイル名と対応させる
- 各ファイルにはただひとつだけ class を含めることができる

このように class 名とファイル名を対応させておくことで、Puppet が class 名

からファイル名を決定し、自動的にロードすることができるようになっています。init.pp では、include td-agent::install などとしていますが、このルールにより、td-agent/manifests/install.pp が自動的にロードできるのです。

module を適用する

さて、作成した td-agent の module を適用してみましょう。今回は、あらたなコマンドライン引数--modulepath と--execute を使います。

--modulepath により、module の格納されているパスを指定します。そのパス以下に含まれるディレクトリの名前が、module 名として認識されます。今回の例では、puppet/modules/ディレクトリ以下にある td-agent ディレクトリが、module 名として認識されます。

--execute オプションに文字列をわたすと、manifest ファイルに書かれたもの同様に実行することができます。今回は、これまでのような通常の manifest を書くことなく、module のみ作成したので、--execute オプションによって class の include を行います。

```
[vagrant@puppet-book ~]$ cd /vagrant/puppet/modules/
[vagrant@puppet-book modules]$ sudo puppet apply --modulepath=. --execute 'i⏎
nclude td-agent'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/descr: descr ch⏎
anged '' to 'treasuredata repo'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/baseurl: baseur⏎
l changed '' to 'http://packages.treasure-data.com/redhat/$basearch/'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/enabled: enable⏎
d changed '' to '1'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/gpgcheck: gpgch⏎
eck changed '' to '0'
Notice: /Stage[main]/Td-agent::Install/Package[td-agent]/ensure: created
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/td-agent.conf]/cont⏎
ent: content changed '{md5}c61a851e347734f4500a9f7f373eed7f' to '{md5}183b11⏎
dffec86747a3b31ddda02384ab'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]/ensure: ensure chan⏎
ged 'stopped' to 'running'
```

```
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh
' from 1 events
Notice: Finished catalog run in 338.01 seconds
```

前回同様に、td-agent のインストールから起動まで、ちゃんと適用が成功したようですね。

まとめ

本章では以下のことを学びました。

- ファイル群を分類・整理するための仕組みとしての module
- module のレイアウトと、`include` によるファイルロードの仕組み
- `--modulepath` と`--execute` オプションを使った module の適用

前章で学んだ class も、本章で学んだ module も、これから本格的な manifest を書いていくためには必須の仕組みですので、よく復習して、内容を完全に把握するようにしてください。

ルールがあることを最初は複雑に感じるかもしれませんが、その分、便利なものですので、慣れさえしたら簡単です。

第15章

サーバの役割を定義する Part.1

前章では、td-agent を使う上でこれだけは最低限必要だろうという状態を、module として記述しました。本章では、その module を使ってより具体的な状態、すなわちサーバの「役割 (ロール)」を定義する manifest を書いていきましょう。

なぜそのような、一見するとまわりくどく思えるようなことをするのでしょうか。それは、module の再利用性を担保するためです。その意味において Puppet の manifest 記述は、一般のプログラミングと同じように、適切な設計を要します。さっそく具体例を見ていきましょう。

td-agent クラスタの構成

td-agent は前述の通り、様々なログを収集・集約するためのツールです。td-agent 間で通信してログをリレー送信できる機能を用い、クラスタを構成して利用するのが一般的です。

単純なクラスタ構成の場合、

- アプリケーションサーバと同居し、そのログを収集して集約サーバに送信する
- 各サーバから送信されてきたログを集約する

という構成になるでしょう。つまりここでは、サーバの役割として

1. なんらかのサービスがログを記録し、そのログを収集・送信する
2. 各サーバから送信されてくるログを集約する

のふたつがあるということになります。第 12 章で掲載した図を、確認のため再度掲載しておきます。

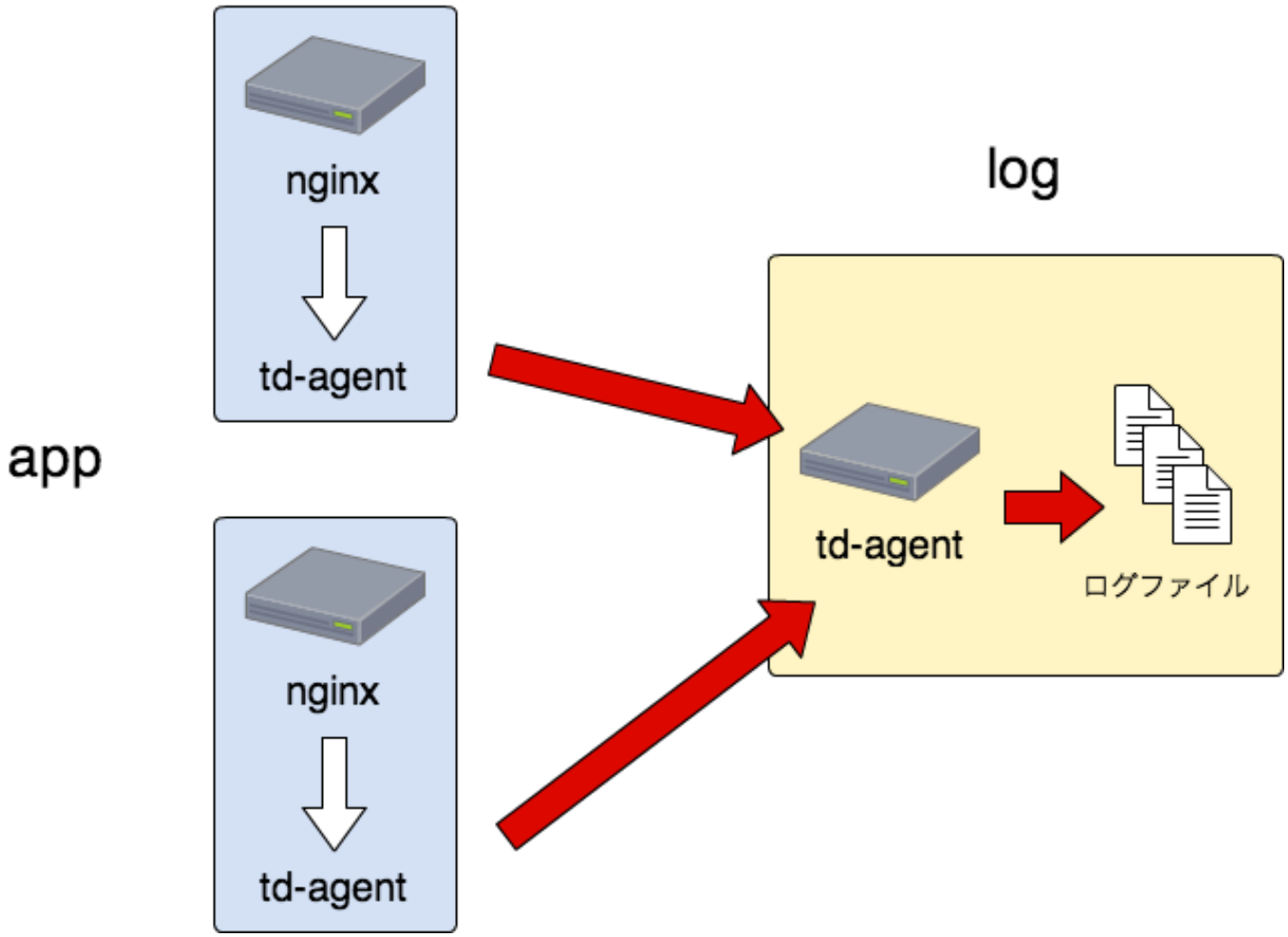

図 15.1　td-agent の構成図

前章で作成した module を拡張するかたちで、これら具体的な役割を記述していきます。

ディレクトリを準備する

まず、今回の manifest を配置するディレクトリを用意します。

```
$ cd puppet/
$ mkdir cluster
$ cd cluster/
```

次に、前回作成した td-agent の module をコピーしてください (実際に module を再利用する際はコピーする必要はありませんが、説明の都合上、前回のディレクトリとわけ

たいのでそうしてもらっています)。

```
$ cp -R ../modules modules
```

上記したふたつの役割を記述する manifest を配置するために、roles というディレクトリを作成しましょう。

```
$  mkdir roles
```

前述の「なんらかのサービスがログを記録し、そのログを収集・送信する」という役割のために app、「各サーバから送信されてくるログを集約する」という役割のために log というディレクトリを作成しましょう。同時に、それぞれのディレクトリの下に manifests と templates も作成します。

```
$ mkdir roles/{app,log}
$ mkdir roles/app/{manifests,templates}
$ mkdir roles/log/{manifests,templates}
```

最後に、roles 以下に定義した manifest を実際に include するファイルを置くために、manifests というディレクトリを作成しましょう。

```
$ mkdir manifests
```

以上の作業により、最終的に以下のようなディレクトリ/ファイル構成ができあがっているはずです。

```
$ tree ../cluster
../cluster
├── manifests
├── modules
│   └── td-agent
│       ├── manifests
```

```
│       │   ├── config.pp
│       │   ├── init.pp
│       │   ├── install.pp
│       │   └── service.pp
│       └── templates
│           └── td-agent.conf
└── roles
    ├── app
    │   ├── manifests
    │   └── templates
    └── log
        ├── manifests
        └── templates
```

あらたに Vagrantfile を準備する

これまでとは異なり、今回はクラスタ構成を採るため仮想ホストを複数台必要としますが、ここでも Vagrant が活躍します。

以下の内容で、puppet/cluster(上で作成した cluster ディレクトリ) の直下に、あらたに Vagrantfile を作成してください (「第 3 章 Vagrant で開発環境を用意する」で説明したように、vagrant init コマンドにより Vagrantfie を作成してから、内容を編集するとよいでしょう)。

```
Vagrant.configure("2") do |config|
  config.vm.box     = "centos-6.4-puppet"
  config.vm.box_url = "http://puppet-vagrant-boxes.puppetlabs.com/centos-64-⏎
x64-vbox4210.box"

  config.vm.define :app do |app_config|
    app_config.vm.hostname = "app.puppet-book.local"
    app_config.vm.network :private_network, ip: "192.168.0.100"
  end

  config.vm.define :log do |log_config|
```

```
    log_config.vm.hostname = "log.puppet-book.local"
    log_config.vm.network :private_network, ip: "192.168.0.101"
    log_config.vm.network :forwarded_port, guest: 24224, host: 24224
  end
end
```

今回は、仮想ホスト間の通信が必要となるため、スタティックなプライベート IP アドレスを割り当てています。

いつものように vagrant up するだけで、Vagrantfile で指定した通り、app と log という名前で 2 台の仮想ホストが起動している様子が確認できます (便利ですね!)。

```
$ vagrant up
Bringing machine 'app' up with 'virtualbox' provider...
[app] Importing base box 'centos-6.4-puppet'...
[app] Matching MAC address for NAT networking...
[app] Setting the name of the VM...
[app] Clearing any previously set forwarded ports...
[app] Fixed port collision for 22 => 2222. Now on port 2205.
[app] Creating shared folders metadata...
[app] Clearing any previously set network interfaces...
[app] Preparing network interfaces based on configuration...
[app] Forwarding ports...
[app] -- 22 => 2205 (adapter 1)
[app] Booting VM...
[app] Waiting for VM to boot. This can take a few minutes.
[app] VM booted and ready for use!
[app] Setting hostname...
[app] Configuring and enabling network interfaces...
[app] Mounting shared folders...
[app] -- /vagrant
Bringing machine 'log' up with 'virtualbox' provider...
[log] Importing base box 'centos-6.4-puppet'...
[log] Matching MAC address for NAT networking...
[log] Setting the name of the VM...
[log] Clearing any previously set forwarded ports...
[log] Fixed port collision for 22 => 2205. Now on port 2206.
```

```
[log] Creating shared folders metadata...
[log] Clearing any previously set network interfaces...
[log] Preparing network interfaces based on configuration...
[log] Forwarding ports...
[log] -- 22 => 2206 (adapter 1)
[log] -- 24224 => 24224 (adapter 1)
[log] Booting VM...
[log] Waiting for VM to boot. This can take a few minutes.
[log] VM booted and ready for use!
[log] Setting hostname...
[log] Configuring and enabling network interfaces...
[log] Mounting shared folders...
[log] -- /vagrant
```

ちなみに、app と log の仮想ホストをそれぞれ別々に vagrant コマンドから扱いたい場合、

```
$ vagrant up app
```

というように、その名前を引数にわたすことができます。

nginx モジュール

さて「なんらかのサービスがログを記録し、そのログを収集・送信する」役割のサーバを定義していくわけですが、あまり人工的な例でもつまらないので、より実践的な題材と取り上げます。その「なんらかのサービス」の具体例として、nginx のログを収集してみることにしましょう。

そのために、まずは第 5 章で作成した nginx の manifest を module 化します。

```
$ mkdir modules/nginx
$ mkdir modules/nginx/manifests
```

内容はほとんど第 5 章と同じですので、駆け足でいきます。nginx へのアクセスを LTSV フォーマットでログに記録するだけの、簡単な設定です。

```
[vagrant@app vagrant]$ curl http://app.puppet-book.local/
```

とアクセスすると、/var/log/nginx/app.access.log にアクセスログが、LTSV フォーマットで書き出されていきます。

以下の内容でそれぞれファイルを作成してください。

modules/nginx/manifests/init.pp:

```
class nginx {
  include nginx::install
  include nginx::config
  include nginx::service

     Class['nginx::install']
  -> Class['nginx::config']
  ~> Class['nginx::service']
}
```

modules/nginx/manifests/install.pp:

```
class nginx::install {
  yumrepo { 'nginx':
    descr    => 'nginx yum repository',
    baseurl  => 'http://nginx.org/packages/centos/6/$basearch/',
    enabled  => 1,
    gpgcheck => 0,
  }

  package { 'nginx':
    require => Yumrepo['nginx'],
  }

  file { '/var/log/nginx':
    ensure  => directory,
    owner   => 'nginx',
    group   => 'nginx',
```

```
    mode    => '0755',
    require => Package['nginx'],
  }
}
```

modules/nginx/manifests/config.pp:

```
class nginx::config {
  file { '/etc/nginx/conf.d/my.conf':
    content => template('nginx/my.conf'),
  }

  file { '/usr/share/nginx/html/index.html':
    content => template('nginx/index.html'),
  }
}
```

modules/nginx/manifests/service.pp:

```
class nginx::service {
  service { 'nginx':
    enable     => true,
    ensure     => running,
    hasrestart => true,
  }
}
```

modules/nginx/templates/my.conf:

```
log_format ltsv "time:$time_local\t"
                "host:$remote_addr\t"
                "method:$request_method\t"
                "path:$request_uri\t"
                "version:$server_protocol\t"
                "status:$status\t"
```

```
                  "size:$body_bytes_sent\t"
                  "referer:$http_referer\t"
                  "ua:$http_user_agent\t"
                  "restime:$request_time\t"
                  "ustime:$upstream_response_time";

server {
  listen       80;
  server_name  app.puppet-book.local;

  access_log /var/log/nginx/app.access.log ltsv;

  location / {
    root  /usr/share/nginx/html;
    index index.html;
  }
}
```

modules/nginx/templates/index.html:

```
Hello, Puppet!
```

ログを収集・送信する

さて、今度はログを収集・送信する td-agent の manifest を書いていきます。前章で作成した module で大枠はできているので、今回は

- nginx のログを取得する
- それを集約サーバに送信する

というふたつの設定をほどこすだけです。

その前に、まずは app という役割を manifest として定義します。といっても特に変わったことをするわけではなく、app という class を定義するだけです。

以下の内容で、roles/app/manifests/init.pp というファイルを作成してください。

```
class app {
  include app::nginx
  include app::td-agent

     Class['app::nginx']
  -> Class['app::td-agent']
}
```

app ロールでは、nginx と td-agent を起動させるのでした。言葉を変えていうと、そのふたつのサービスからなる役割のことを app と呼ぶことにしたのでした。このように、module を組合せることで、役割を定義していきます。

ここでいったん整理してみると、システム状態を記述する manifest の全体は、

1. システムの状態を表す最小単位としての各種 resource type
2. resource type をグルーピングする class
3. class などの集まりとしての module
4. module の組合せとしての役割 (ロール)

という階層関係を持ちます。

さて、上で名前だけ出てきた app::nginx の中身を作成しましょう。以下の内容で、roles/app/manifests/nginx.pp というファイルを作成してください。

```
class app::nginx {
  include ::nginx
}
```

今回、nginx は module の設定をそのまま用いるので、ここでは単に、先に定義した module に含まれる class を include するだけです。ちなみに、::nginx と::と先頭につけているのは、名前空間のトップレベルから名前解決を行うという意味です。Ruby を知っている読者なら、Ruby における module の名前解決と同様の仕組みだといえば、わかりやすいでしょう。詳細については、以下のドキュメントを参照してください。

http://docs.puppetlabs.com/puppet/latest/reference/lang_namespaces.html

このように、いまいるコンテキストとは違う階層にある別の class を include する際は、このように::を先頭につけて、明示的にトップレベルから指定すると、思わぬハマり

どころを避けられるので、お勧めです。

次は、td-agent です。以下の内容で、roles/app/manifests/td-agent.pp というファイルを作成してください。

```
class app::td-agent {
  include ::td-agent
  include app::td-agent::config

     Class['::td-agent::install']
  -> Class['app::td-agent::config']
  ~> Class['::td-agent::service']
}
```

今回、app ロールのために専用の設定を追加するので、app::td-agent::config という class を作成します。また、module に含まれる各 class との関係も、ここで定義しておきます。

app::td-agent::config の内容は以下の通りです。

roles/app/manifests/td-agent/config.pp:

```
class app::td-agent::config {
  file { '/etc/td-agent/conf.d/app.conf':
    content => template('app/td-agent/app.conf'),
  }
}
```

テンプレートも用意しましょう。

roles/app/templates/td-agent/app.conf:

```
<source>
  type     tail
  path     /var/log/nginx/app.access.log
  tag      forward.app.access
  format   ltsv
</source>
```

```
<match forward.**>
  type forward

  <server>
    host 192.168.0.101
    port 24224
  </server>

  buffer_type file
  buffer_path /var/log/td-agent/buffer/forward

  # すぐに結果を確認できるよう、一時的に短かくしておく
  flush_interval 1s
</match>
```

大元の td-agent の module では、/etc/td-agent/conf.d/*.conf をロードするよう設定されているので、ここでは roles/app/templates/td-agent/app.conf として、app ロールに特化したファイルを配置するだけで、設定が完了します。

app ロールを include する

上記で app ロール自体は定義できましたが、それを適用するには、app ロールを定義した class をどこかで include しなければなりません。ここでは、manifests ディレクトリ以下に、app ロールへの、いわばエントリーポイントとでもいうべきファイルを用意し、そのファイルを puppet apply 時に引数としてわたすようにしましょう。

以下の内容で、manifests/app.pp というファイルを作成してください。

```
include app
```

中身は、単に app ロールを定義した class を include しているだけです。

manifest を適用する

ここでまず、完成した app ロールの manifest を適用してみましょう。

vagrant ssh に app という引数をわたしてログインします。また、Vagrantfile の場所がこれまでとは違うので、マウントされているディレクトリも異なります。注意してください。

```
$ vagrant ssh app
Welcome to your Vagrant-built virtual machine.
[vagrant@app ~]$ cd /vagrant
[vagrant@app vagrant]$ ls
Vagrantfile  modules  roles
```

適用は、いつもの通り puppet apply コマンドを使います。今回は、--modulepath の引数に roles ディレクトリを追加しています。役割を定義する manifest も、実際には module を組合せた module として構成しているからです。

```
[vagrant@app vagrant]$ sudo puppet apply --modulepath=modules:roles manifest ⏎
s/app.pp
Notice: /Stage[main]/Nginx::Install/Yumrepo[nginx]/descr: descr changed '' t ⏎
o 'nginx yum repository'
Notice: /Stage[main]/Nginx::Install/Yumrepo[nginx]/baseurl: baseurl changed ⏎
'' to 'http://nginx.org/packages/centos/6/$basearch/'
Notice: /Stage[main]/Nginx::Install/Yumrepo[nginx]/enabled: enabled changed ⏎
'' to '1'
Notice: /Stage[main]/Nginx::Install/Yumrepo[nginx]/gpgcheck: gpgcheck change ⏎
d '' to '0'Notice: /Stage[main]/Nginx::Install/Package[nginx]/ensure: create ⏎
d
Notice: /Stage[main]/Nginx::Install/File[/var/log/nginx]/owner: owner change ⏎
d 'root' to 'nginx'
Notice: /Stage[main]/Nginx::Install/File[/var/log/nginx]/group: group change ⏎
d 'root' to 'nginx'
Notice: /Stage[main]/Nginx::Config/File[/usr/share/nginx/html/index.html]/co ⏎
ntent: content changed '{md5}e3eb0a1df437f3f97a64aca5952c8ea0' to '{md5}1db1 ⏎
```

```
6ebfb21d376e5b2ae9d1eaf5b3c8'
Notice: /Stage[main]/Nginx::Config/File[/etc/nginx/conf.d/my.conf]/ensure: d
efined content as '{md5}0f2ddfb71fadb5571cdb578235054a99'
Notice: /Stage[main]/Nginx::Service/Service[nginx]/ensure: ensure changed 's
topped' to 'running'
Notice: /Stage[main]/Nginx::Service/Service[nginx]: Triggered 'refresh' from
 1 events
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/descr: descr ch
anged '' to 'treasuredata repo'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/baseurl: baseur
l changed '' to 'http://packages.treasure-data.com/redhat/$basearch/'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/enabled: enable
d changed '' to '1'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/gpgcheck: gpgch
eck changed '' to '0'
Notice: /Stage[main]/Td-agent::Install/Package[td-agent]/ensure: created
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/conf.d]/ensure: cre
ated
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/td-agent.conf]/cont
ent: content changed '{md5}c61a851e347734f4500a9f7f373eed7f' to '{md5}f3d4e2
ffaec6ef9b67bd01171844fa60'
Notice: /Stage[main]/App::Td-agent::Config/File[/etc/td-agent/conf.d/app.con
f]/ensure: defined content as '{md5}498cd8e61d1fb468975524222092 59b9'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]/ensure: ensure chan
ged 'stopped' to 'running'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh
' from 1 events
Notice: Finished catalog run in 98.93 seconds
```

実際に nginx と td-agent が起動しているか、確認してみましょう。

```
[vagrant@app vagrant]$ sudo service nginx status
nginx (pid  5886) is running...
[vagrant@app vagrant]$ sudo service td-agent status
td-agent (pid  6118) is running...
```

ちゃんと起動しているようです。

まとめ

本章では以下のことを学びました。

- システムの具体的な役割をロールとして記述していくこと
- その際、既存のモジュールを拡張する形で manifest を書いていくこと
- Vagrant で複数の仮想ホストを起動する

仮想ホストを 2 台使うようになって、構成が急速に複雑化しましたが、やっていること自体はこれまでと変わるところはありません。じっくり取り組んでみてください。

第16章

サーバの役割を定義する Part.2

前章では、td-agent クラスタのうち「nginx のログを収集・送信する」役割 (ロール) のサーバについて、manifest を記述しました。本章では「各サーバから送信されてくるログを集約するサーバ」について、manifest を書いていきましょう。

集約サーバの役割

今回採った構成では、nginx と同じサーバで起動する td-agent がそのログを読み取り、本章で manifest を書いていく中間 td-agent に対してログを送信するのでした。そのようにして集約したログに対して、中間 td-agent で、

- 必要な加工をほどこす
- ログファイルに書き出す
- MongoDB や S3 など、別のストレージに送信する

といったことを行うことで、ログの収集・集約の仕組みを簡単に作れてしまうのが、td-agent の素晴しいところです。

本章ではその一端を体験してみるために、集約したログをファイルに書き出してみるということをしてみましょう。

ディレクトリ構成のおさらい

まずは前回作成したディレクトリを確認してみましょう。以下のようになっているはずです。

```
$ cd puppet/cluster/
$ tree
.
├── Vagrantfile
```

```
├── manifests
│   └── app.pp
├── modules
│   ├── nginx
│   │   ├── manifests
│   │   │   ├── config.pp
│   │   │   ├── init.pp
│   │   │   ├── install.pp
│   │   │   └── service.pp
│   │   └── templates
│   │       ├── index.html
│   │       └── my.conf
│   └── td-agent
│       ├── manifests
│       │   ├── config.pp
│       │   ├── init.pp
│       │   ├── install.pp
│       │   └── service.pp
│       └── templates
│           └── td-agent.conf
└── roles
    ├── app
    │   ├── manifests
    │   │   ├── init.pp
    │   │   ├── nginx.pp
    │   │   ├── td-agent
    │   │   │   └── config.pp
    │   │   └── td-agent.pp
    │   └── templates
    │       └── td-agent
    │           └── app.conf
    └── log
        ├── manifests
        └── templates
```

今回は、roles/log 以下にファイルを配置していきます。さっそく、大元の class を書いていきましょう。以下の内容で、roles/log/manifests/init.pp というファイル

を作成してください。

```
class log {
  include ::iptables
  include log::td-agent
}
```

今回は log という役割を、その具体的な中身として集約 td-agent が動くものとして定義していきます。そのため、これから log::td-agent という class を作成します。

iptables についての補足

その前に、include ::iptables という箇所について説明しておきましょう。

提供されている Vagrant の OS イメージによっては、iptables により、外部からの接続が制限されている場合があります。本書の検証環境では、80 番ポートや集約 td-agent で利用する 24224 番ポートなどは、外部からの接続が禁止された状態になっていました。

今回は、manifest を書き、td-agent の動作を検証することが目的ですので、いったん iptables を無効にしておきましょう。「iptables が無効になっている」という状態を定義するため、以下の通り modules を作成します。

```
$ mkdir modules/iptables
$ mkdir modules/iptables/manifests
```

以下の内容で、それぞれファイルを作成します。

modules/iptables/manifests/init.pp:

```
class iptables {
  include iptables::service
}
```

modules/iptables/manifests/service.pp:

```
class iptables::service {
  service { 'iptables':
    enable => false,
    ensure => stopped,
  }
}
```

これを、前述の通り incelude ::iptables として読み込んでおき、iptables を無効にしておきます。今回は、あくまでも検証なのでこのような方法を採りましたが、本番環境でこのようなことをすることのないよう、念のため申し添えておきます。

log::td-agent

さて、次は log::td-agent です。以下の内容で、td-agent.pp というファイルを作成してください。

```
class log::td-agent {
  include ::td-agent
  include log::td-agent::config

     Class['::td-agent::install']
  -> Class['log::td-agent::config']
  ~> Class['::td-agent::service']
}
```

ここでなにをやっているかについてはもう、説明するまでもないでしょう。大元の module の設定に、集約 td-agent 専用の設定ファイルを追加して、サービスを refresh するという状態を定義しています。

次に、以下の内容で log::td-agent::config を作成します。

```
class log::td-agent::config {
  file { '/etc/td-agent/conf.d/log.conf':
    content => template('log/td-agent/log.conf'),
  }
```

```
  file { '/var/log/td-agent/app':
      ensure => directory,
      owner  => 'td-agent',
      group  => 'td-agent',
  }
}
```

今回、app ロールのサーバから td-agent を通じて送信されてくる nginx のアクセスログを、log ロールの/var/log/td-agent/app 以下にファイルとして集約することにします。そのため、あらかじめログを配置するディレクトリを作成しています。

次に設定ファイルです。以下の内容で、roles/log/templates/td-agent/log.conf というファイルを作成してください。

```
<source>
  type forward
  port 24224
</source>

<match forward.**>
  type file
  path /var/log/td-agent/app/access
  time_slice_format %Y%m%d
</match>
```

この設定により集約 td-agent は、

1. 24224 番ポートで接続を待ち受ける
2. 送信されてきたログを一定期間バッファリングする (上記の設定だと、time_slice_format が %Y%m%d なので、1 日ごとにログファイルに書き出す)
3. path、time_slice_format で指定されたファイル名、フラッシュ時に自動採番される数字により構成されるファイル名に、ログを書き出します。

out_file プラグインの仕様については、以下のブログエントリが詳しいのでご参照ください。

http://blog.tnmt.info/2012/10/19/about-fluentd-out-file-plugin/

log ロールを include する

app ロールと同様に、log ロールのためのエントリーポイントとなるファイルを作成しましょう。

以下の内容で、manifests/log.pp というファイルを作成してください。

```
include log
```

最終的なディレクトリ構成は、以下のようになります。

```
$ tree ../cluster
../cluster/
├── Vagrantfile
├── manifests
│   ├── app.pp
│   └── log.pp
├── modules
│   ├── iptables
│   │   └── manifests
│   │       ├── init.pp
│   │       └── service.pp
│   ├── nginx
│   │   ├── manifests
│   │   │   ├── config.pp
│   │   │   ├── init.pp
│   │   │   ├── install.pp
│   │   │   └── service.pp
│   │   └── templates
│   │       ├── index.html
│   │       └── my.conf
│   └── td-agent
│       ├── manifests
│       │   ├── config.pp
│       │   ├── init.pp
```

```
│       │   ├── install.pp
│       │   └── service.pp
│       └── templates
│           └── td-agent.conf
└── roles
    ├── app
    │   ├── manifests
    │   │   ├── init.pp
    │   │   ├── nginx.pp
    │   │   ├── td-agent
    │   │   │   └── config.pp
    │   │   └── td-agent.pp
    │   └── templates
    │       └── td-agent
    │           └── app.conf
    └── log
        ├── manifests
        │   ├── init.pp
        │   ├── td-agent
        │   │   └── config.pp
        │   └── td-agent.pp
        └── templates
            └── td-agent
                └── log.conf
```

manifest を適用する

まず、log ロールの仮想ホストにログインします (仮想ホストが 2 台あってまぎらわしいので、間違えないよう気をつけてください)。

```
$ vagrant ssh log
```

puppet apply で manifest を適用しましょう。--modulepath=modules:roles と、roles ディレクトリも指定します。

```
[vagrant@log ~]$ cd /vagrant
[vagrant@log vagrant]$ sudo puppet apply --modulepath=modules:roles manifest ⏎
s/log.pp
Notice: /Stage[main]/Iptables::Service/Service[iptables]/ensure: ensure chan ⏎
ged 'running' to 'stopped'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/descr: descr ch ⏎
anged '' to 'treasuredata repo'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/baseurl: baseur ⏎
l changed '' to 'http://packages.treasure-data.com/redhat/$basearch/'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/enabled: enable ⏎
d changed '' to '1'
Notice: /Stage[main]/Td-agent::Install/Yumrepo[treasuredata]/gpgcheck: gpgch ⏎
eck changed '' to '0'
Notice: /Stage[main]/Td-agent::Install/Package[td-agent]/ensure: created
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/conf.d]/ensure: cre ⏎
ated
Notice: /Stage[main]/Log::Td-agent::Config/File[/var/log/td-agent/app]/ensur ⏎
e: created
Notice: /Stage[main]/Log::Td-agent::Config/File[/etc/td-agent/conf.d/log.con ⏎
f]/ensure: defined content as '{md5}b5f1e6a793038fe449196c52aa544168'
Notice: /Stage[main]/Td-agent::Config/File[/etc/td-agent/td-agent.conf]/cont ⏎
ent: content changed '{md5}c61a851e347734f4500a9f7f373eed7f' to '{md5}f3d4e2 ⏎
ffaec6ef9b67bd01171844fa60'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]/ensure: ensure chan ⏎
ged 'stopped' to 'running'
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh ⏎
' from 1 events
Notice: Finished catalog run in 152.16 seconds
```

td-agent が起動しているか確かめてみましょう。

```
[vagrant@log vagrant]$ sudo service td-agent status
td-agent (pid  6162) is running...
```

ちゃんと動いているようですね。

仮想ホスト同士の連携を動作確認する

なにを「動作確認」するべきなのでしょうか。あらためて整理してみましょう。app で http://app.puppet-book.local/にアクセスすると:

1. app ロールの/var/log/nginx/app.access.log に、アクセスログが記録される
2. 1 のログが、log ロールの td-agent に送信され、/var/log/td-agent/app 以下に記録される

というものでした。実際にみていきましょう。

まず、app ロールの仮想ホストで、http://app.puppet-book.local/にアクセスしてみます。

```
[vagrant@app vagrant]$ curl http://app.puppet-book.local/
Hello, Puppet!
```

ログを見てみましょう。

```
[vagrant@app vagrant]$ cat /var/log/nginx/app.access.log
time:20/Apr/2013:10:09:55 +0000 host:127.0.0.1  method:GET      path:/  vers⏎
ion:HTTP/1.1        status:200    size:15 referer:-       ua:curl/7.19.7 (x8⏎
6_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.13.6.0 zlib/1.2.3 libidn/1.18 li⏎
bssh2/1.4.2     restime:0.000   ustime:-
```

ちゃんと記録されているようですね。

次に、今度は log ホストにログインして、ログファイルのディレクトリをのぞいてみましょう。

```
[vagrant@log vagrant]$ ls -la /var/log/td-agent/app
total 12
drwxr-xr-x 2 td-agent td-agent 4096 Apr 20 10:09 .
drwxr-xr-x 3 td-agent td-agent 4096 Apr 20 10:05 ..
-rw-rw-rw- 1 td-agent td-agent  331 Apr 20 10:09 access.20130420.b4dac809922⏎
```

```
779494
```

なにやらファイルができています。

```
[vagrant@log vagrant]$ cat /var/log/td-agent/app/access.20130420.b4dac809922 ⏎
779494
2013-04-20T10:09:55+00:00       forward.app.access      {"time":"20/Apr/2013 ⏎
:10:09:55 +0000","host":"127.0.0.1","method":"GET","path":"/","version":"HTT ⏎
P/1.1","status":"200","size":"15","referer":"-","ua":"curl/7.19.7 (x86_64-re ⏎
dhat-linux-gnu) libcurl/7.19.7 NSS/3.13.6.0 zlib/1.2.3 libidn/1.18 libssh2/1 ⏎
.4.2","restime":"0.000","ustime":"-"}
```

app の仮想ホストに記録されていたログが、きちんと転送されてきているようですね。

ファイル名が access.20130420.b4dac809922779494 となっており、末尾になにやらハッシュ値のようなものがついています。これは現在バッファリング中の状態であることを示します。

先に説明した通り、out_file プラグインは time_slice_format で指定した期間の単位でログファイルを作成するので、このファイルは翌日 (正確には、この場合 4/21 の 0:00 を過ぎた後に)、access.20130420_0 というファイルに書き出されます。

以上で、Puppet で td-agent のクラスタ構成を記述し、動作確認することができました。

まとめ

本章では以下のことを学びました。

- 集約 td-agent の役割、manifest の書き方
- iptables について
- out_file プラグインについて

また、前章までに書いてきた module やロール定義と合わせて、Puppet により td-agent のクラスタ構成を定義し、nginx のログを取得・集約できることを確認しました。

現実のシステムは、様々な役割を持った複数台のサーバにより構成されます。今回はごく単純な td-agent によるクラスタ構成を試してみましたが、現実のシステム構築も本質

的には、今回と同じようなことを必要な分だけくりかえしていくことに他なりません。

本書をここまで読んだあなたは、Puppet を使ってシステムを構築する知識を既に習得済みです。自信を持って、現実のシステムに適用していってください。

第17章

manifest の共通部分をくくりだす

manifest を書いていると、同じような記述を何度も書いていることに気付くことがあるでしょう。そのような場合、プログラミング言語では共通の処理を関数としてくくりだし、差異を引数として渡されるパラメタにより調整するということが行われます。

本章では、あたかもプログラミング言語における関数のように manifest の共通部分をくくりだし、再利用可能にする方法について説明します。

Defined Type

先に「関数のように」と述べましたが、すこし不正確な表現です。Puppet における共通部分のくくりだしは、実際には、defined type(ユーザ定義 resource type) と呼ばれる、あらたな resource type の定義に他なりません。

http://docs.puppetlabs.com/puppet/latest/reference/lang_defined_types.html

とはいえ、ここでは言葉の定義には深入りしません。関数のようなものだと思って使っても、特に問題はないからです。

td-agent のプラグイン

defined type がどういうものなのか、どう役立つのかを知るためには、まずは実際にそれを定義し、使ってみるのがはやいでしょう。ここでは、前章までに記述した manifest に、td-agent のプラグインをインストールする記述を追加することで、具体例を見ていきます。

今回は、fluent-plugin-extract_query_params というプラグインを使ってみましょう。これは、指定したキーの指す値を URL としてパースし、クエリパラメタを抽出して、それらを元のログの key/value として追加するためのプラグインです。

具体的には、下記のようなログがあったとして、

```
{
  "url" : "http://example.com/?foo=bar&baz=qux
}
```

url というキーの指す値を URL としてパースし、クエリパラメタを抽出し、ログを以下のように加工するというものです。

```
{
  "url" : "http://example.com/?foo=bar&baz=qux"
  "foo" : "bar",
  "baz" : "qux"
}
```

exec を生で使うことの問題点

td-agent のプラグインをインストールするには、td-agent に同梱されている fluentd-gem コマンドを使います。CentOS の x86_64 アーキテクチャでは、/usr/lib64/fluent/ruby/bin 以下にインストールされているはずです。

さて、fluent-gem コマンドを使ってプラグインをインストールする manifest を書いていくわけですが、これまでの知識を使って書くなら、exec を使ってこのように書くことになるでしょう。

```
exec { "fluent-gem install ${plugin_name}":
  path    => '/bin:/usr/lib64/fluent/ruby/bin',
  command => "fluent-gem install ${plugin_name}",
  unless  => "fluent-gem list | grep ${plugin_name} 2>/dev/null",
}
```

単にプラグインをインストールする分には、これだけでも十分に用をなしますが、よりよい manifest を書いていくには以下の点が問題となります。

1. 「インストールされていないこと」という状態を記述したい場合にどうするか
2. 冪等性をいちいち自分で担保するのは面倒

3. プラグインのインストール状態自体をあらたな resource type として定義する方が再利用性が高い

これらの問題を、defined type を使って解決していきましょう。

defined type を定義する

上記に挙げた問題は、主に exec を生で使っていることに起因します。第 11 章で以下のように書かれていたことをおぼえているでしょうか。

> manifest を書いていて、exec を多用しているなと感じたら、たいていは manifest の書き方に問題があります。exec を使わざるを得ない場合でも、「第 17 章 manifest の共通部分をくくりだす」で解説する defined type でラップして、できるだけ生で exec を使わないようにする方がよいでしょう。

exec は、便利な反面、自分で冪等性を担保する必要があり、その取り扱いが難しい resource type です。また、exec が多用されている manifest は、ぱっと見で何をやっているのかを判別するのが困難です。define type を用いて適切に名前づけしてやることで、manifest の可読性を向上させ、再利用性を高めることができます。

さっそく、td-agent プラグインのインストール状態を記述するための defined type を書いていきましょう。これはどのロールからも利用され得るものですので、module に追加します。以下の内容で、modules/td-agent/manifests/plugin.pp というファイルを作成してください。

```
define td-agent::plugin (
  $ensure = 'present'
) {
  $plugin_name = "fluent-plugin-${title}"

  Exec {
    path => '/bin:/usr/lib64/fluent/ruby/bin',
  }

  if $ensure == 'present' {
    exec { "fluent-gem install ${plugin_name}":
```

```
      command => "fluent-gem install ${plugin_name}",
      unless  => "fluent-gem list | grep ${plugin_name} 2>/dev/null",
    }
  }
  elsif $ensure == 'absent' {
    exec { "fluent-gem uninstall ${plugin_name}":
      command => "fluent-gem uninstall ${plugin_name}",
      onlyif  => "fluent-gem list | grep ${plugin_name} 2>/dev/null",
    }
  }
  else {
    fail "${ensure} for `ensure` is not supported!"
  }
}
```

見慣れない記法がたくさんでてきましたね。このうちのいくつかは次章で説明しますので、いまは深く考えないようにしてください。とはいえ、その意味はだいたい想像がつくでしょう。ここでは、おさえておきたいふたつのことがらについてのみ説明します。

まずは、冒頭の記述について。

```
define td-agent::plugin (
  $ensure = 'present'
) {
  ...
}
```

ここではふたつのことが行われています。まず、`td-agent::plugin` という名前であらたな resource type が定義されています。また、この名前とファイル名とを一致させる必要があるため、`modules/td-agent/manifests/plugin.pp` というファイルに、この defined type を記述したのでした。

また`$ensure = 'present'`という記述によって、この defined type を利用する時に使える attribute を宣言しています。`= 'present'`の箇所は、もし `ensure` が指定されていなかった場合の、デフォルト値を示します。

ふたつめは、$title という変数に resource を宣言した時の名前が自動的に格納されるということです。具体的には、

```
td-agent::plugin { 'extract_query_params': }
```

と書いた時、$title には extract_query_params という文字列が自動的に格納されます。あとの記述ではそれを用いて、コマンド文字列などを構築しています。

td-agent::plugin が行っていること

今回作成した td-agent::plugin は、以下のことを行っています。

1. プラグインのインストール状態に対応する、td-agent::plugin という resource type を定義する
2. $ensure の値によって、インストール/アンインストールという状態を条件分岐する
3. それぞれについて、exec を用いて、冪等性を担保しつつ状態を実現する
4. もし、$ensure の値として不適当なものがわたってきた場合、メッセージを表示してエラーとする

これらにより、先に挙げた 3 つの問題を全て解決します。つまり、

1. $ensure の値で条件分岐することで、インストール/アンインストール両方の状態に対応する
2. defined type の中に冪等性の担保を隠蔽することで、使う側がそれを意識する必要をなくす
3. td-agent::plugin というあらたな resource type を定義することで、manifest の可読性と再利用性を高める

このように、一度 defined type として定義しておけば、使う側は exec の羅列に悩まされることなく、明確な名前に基づく簡単な記述のみで済むのです。

defined type を使う

さっそく td-agent::plugin を使ってみましょう。log ロールの td-agent に、プラグインがインストールされた状態の記述を追加します。

```
$ cd puppet/cluster
```

以下の内容で、roles/log/manifests/td-agent/plugin.pp というファイルを作成してください。

```
class log::td-agent::plugin {
  td-agent::plugin { 'extract_query_params':
    ensure => present,
  }
}
```

大元の class で log::td-agent::plugin を include するため、roles/log/manifests/td-agent.pp を以下のように変更します。

```
class log::td-agent {
  include ::td-agent
  include log::td-agent::config
  include log::td-agent::plugin

     Class['::td-agent::install']
  -> Class['log::td-agent::config']
  ~> Class['::td-agent::service']

     Class['::td-agent::install']
  -> Class['log::td-agent::plugin']
  ~> Class['::td-agent::service']
}
```

プラグインのインストール状態の変更にともなって td-agent を再起動する必要があり

ますので、refresh relationship も定義しておきます。

manifest を適用する

さて、今度は manifest を適用してみましょう。前回同様、log ロールで puppet apply を実行します。

```
[vagrant@log vagrant]$ sudo puppet apply --modulepath=modules:roles manifest ⏎
s/log.pp
Notice: /Stage[main]/Log::Td-agent::Plugin/Td-agent::Plugin[extract_query_pa ⏎
rams]/Exec[fluent-gem install fluent-plugin-extract_query_params]/returns: e ⏎
xecuted successfully
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh ⏎
' from 1 events
Notice: Finished catalog run in 28.99 seconds
```

プラグインがインストールされているかどうか、fluent-gem list コマンドで確認してみましょう。

```
[vagrant@log vagrant]$ /usr/lib64/fluent/ruby/bin/fluent-gem list fluent-plu ⏎
gin-extract_query_params
*** LOCAL GEMS ***

fluent-plugin-extract_query_params (0.0.2)
```

冪等性が正しく担保されているかどうか、もう一度 puppet apply を実行することで確かめてみましょう。

```
[vagrant@log vagrant]$ sudo puppet apply --modulepath=modules:roles manifest ⏎
s/log.pp
Notice: Finished catalog run in 0.56 seconds
```

既にプラグインがインストールされているので、今度はなにも出力されることなく、適用が終了しました。

さらに、アンインストールされている状態も正しく適用されるかどうか、確かめてみましょう。ensure => present と書いた箇所を ensure => absent に変更した上で、puppet apply を実行します。

```
[vagrant@log vagrant]$ sudo puppet apply --modulepath=modules:roles manifest ⏎
s/log.pp
Notice: /Stage[main]/Log::Td-agent::Plugin/Td-agent::Plugin[extract_query_pa ⏎
rams]/Exec[fluent-gem uninstall fluent-plugin-extract_query_params]/returns: ⏎
 executed successfully
Notice: /Stage[main]/Td-agent::Service/Service[td-agent]: Triggered 'refresh ⏎
' from 1 events
Notice: Finished catalog run in 1.41 seconds
```

Exec[fluent-gem uninstall fluent-plugin-extract_query_params] が実行されていることから、意図通りアンインストールが実行されたようです。

プラグインの設定を追加する

せっかく fluent-plugin-extract_query_params プラグインをインストールしたので、defined type とは関係ありませんが、そのための設定も manifest に追加してみましょう。

roles/log/templates/td-agent/log.conf を、以下のように変更します。

```
<source>
  type forward
  port 24224
</source>

<match forward.**>
  type   extract_query_params
  key    path
  except time, host, method, path, version, status, size, referer, ua, resti ⏎
me, ustime
  add_tag_prefix with_queries.
</match>
```

```
<match with_queries.**>
  type file
  path /var/log/td-agent/app/access
  time_slice_format %Y%m%d
</match>
```

これまでは、送信されてきたログをそのままファイルに書き出していましたが、今回はその前に、extract_query_params プラグインで加工する処理を挟んでいます。再度 sudo puppet apply --modulepath=modules:roles manifests/log.pp を実行し、変更を適用してください。

前回同様に、動作確認をしてみましょう。今回は、アクセスする URL にクエリパラメタを付与します。app ロールの仮想ホストから、以下の通りコマンドを実行します。

```
[vagrant@app vagrant]$ curl 'http://app.puppet-book.local/?foo=bar&baz=qux'
Hello, Puppet!
```

今度は log ロールの仮想ホストにログインします。

```
[vagrant@log vagrant]$ ls /var/log/td-agent/app/
access.20130421.b4dad900a93dc7767
```

ログファイルが作成されています。中身を見てみましょう。

```
[vagrant@log vagrant]$ cat /var/log/td-agent/app/access.20130421.b4dad900a93 ⏎
dc7767
2013-04-21T06:24:19+00:00       with_queries.forward.app.access {"time":"21/⏎
Apr/2013:06:24:19 +0000","host":"127.0.0.1","method":"GET","path":"/?foo=bar ⏎
&baz=qux","version":"HTTP/1.1","status":"200","size":"15","referer":"-","ua"⏎
:"curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.13.6.0 zlib/1.2 ⏎
.3 libidn/1.18 libssh2/1.4.2","restime":"0.000","ustime":"-","foo":"bar","ba ⏎
z":"qux"}
```

ログの末尾に"foo":"bar","baz":"qux"と、クエリパラメタがあらたな key/value として追加されているのが確認できますね。

これで、td-agent プラグインも自由に使えるようになりました。

まとめ

本章では以下のことを学びました。

- `exec` を生で使うことの問題
- defined type による解決
- td-agent プラグインのインストール、設定方法

defined type を積極的に使っていくことで、manifest の可読性を向上させ、再利用性を高めることができます。みなさんがこれから manifest を書いていく上で同様の問題に直面したら、defined type を使ってみることをおすすめします。

第18章

最低限必要な Puppet 言語の構文を学ぶ

前章で記述した td-agent::plugin には、これまで説明してこなかったいくつかの言語要素が含まれていました。本章では、その内容に沿って、manifest を書く上で最低限必要となるだろう構文について見ていきます。

前章のおさらい

前章で作成した td-agent::plugin は、以下の通りでした。

```
define td-agent::plugin (
  $ensure = 'present'
) {
  $plugin_name = "fluent-plugin-${title}"       #=> (1), (2), (3)

  Exec {                                        #=> (4)
    path => '/bin:/usr/lib64/fluent/ruby/bin',
  }

  if $ensure == 'present' {                     #=> (5)
    exec { "fluent-gem install ${plugin_name}":
      command => "fluent-gem install ${plugin_name}",
      unless  => "fluent-gem list | grep ${plugin_name} 2>/dev/null",
    }
  }
  elsif $ensure == 'absent' {
    exec { "fluent-gem uninstall ${plugin_name}":
      command => "fluent-gem uninstall ${plugin_name}",
      onlyif  => "fluent-gem list | grep ${plugin_name} 2>/dev/null",
    }
  }
  else {
```

```
    fail "${ensure} for `ensure` is not supported!"
  }
}
```

ここに現れる言語要素のうち、これまでに説明してこなかったものとして、

1. 変数とデータ型
2. 文字列中の変数展開
3. 変数のスコープ
4. resource のデフォルト値
5. 条件分岐

が挙げられます。以下、それぞれについて簡単に見ていきましょう。

変数とデータ型

変数については、第 5 章の nginx の manifest で、以下のような記述がありました。

```
$port = 80
```

変数はこのように、なんらかの値を代入することで宣言されます。変数に代入可能な値には、以下のものがあります。

- 真偽値
- 未定義値
- 文字列
- 数値
- 配列
- ハッシュ

このうち、文字列、数値、配列については、既に使いました。実際、それらのデータ型を変数に代入することがよく行われます。データ型の詳細については、以下のドキュメントを参照してください。

http://docs.puppetlabs.com/puppet/latest/reference/lang_datatypes.html

文字列中の変数展開

Puppet の言語では、文字列リテラル中の変数展開 (interpolation) が可能です。スクリプト言語によく見られるのと同様の機能です。先の例だと、以下のようになっていました。

```
$plugin_name = "fluent-plugin-${title}"
```

この場合、$title に extract_query_params が格納された状態で、$plugin_name に$title が展開された値、すなわち、fluent-plugin-extract_query_params が代入されたのでした。

ちなみに、この箇所は以下のように書いても同じです。

```
$plugin_name = "fluent-plugin-$title"
```

ただし、変数展開する際は{と}で必ず囲むことが、Style Guide で推奨されています。
http://docs.puppetlabs.com/guides/style_guide.html#quoting

変数のスコープ

変数のスコープ (変数が参照できる範囲) には、以下の 3 つがあります。

1. トップスコープの変数
2. 現在のローカルスコープ内の変数
3. 他のローカルスコープ内の変数

スコープについてのドキュメントに掲げられている、以下の図の通りです。

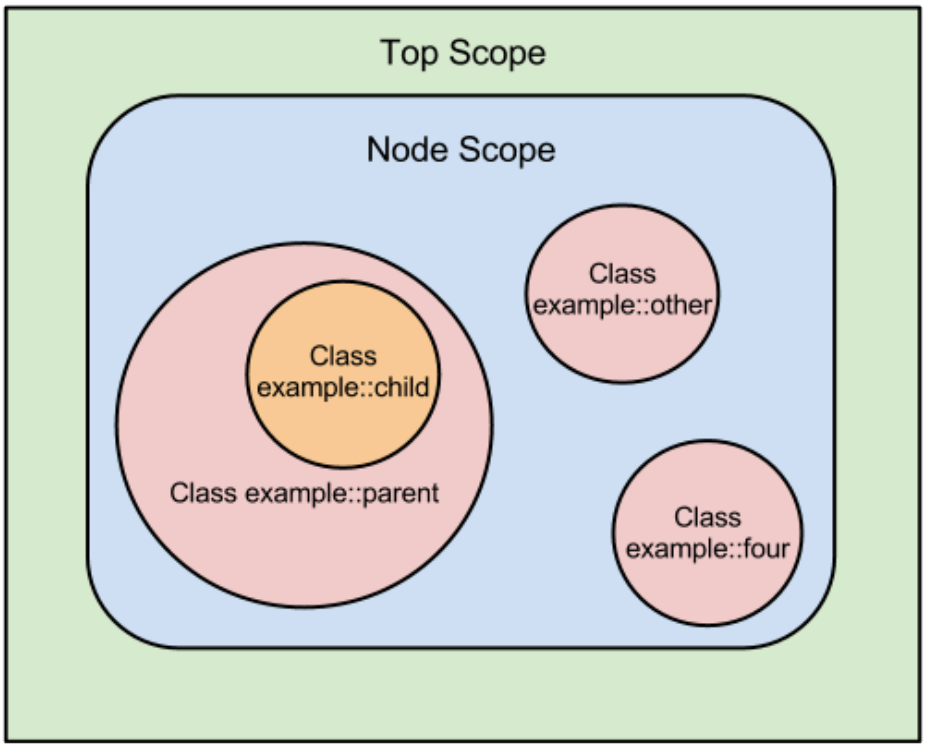

図 18.1　スコープの概略図

図中には Node Scope についても描かれていますが、node は本書の範囲外なので、ここではスコープの種類を 3 つとしています。

以下のような `scope.pp` というファイルがあったとします。これを `puppet apply scope.pp` として実行する場合、`$top` はトップスコープに属することになります。

```
$top = "Hello, Puppet!"

class foo {
  notice($top) #=> "Hello, Puppet!"
}

include foo
```

トップスコープとは、その名の通り、manifest 中のどこからでも参照できる変数です。第 9 章で、template に文字列を埋め込む際にその名前が出てきたのを憶えているでしょうか。

```
$to_be_overwritten = "top"

class bar {
  $local = "Hello, Puppet!"
  notice($local)               #=> "Hello, Puppet!"

  $to_be_overwritten = "local"
  notice($to_be_overwritten)   #=> "local"
}

include bar

notice($local)                 #=> undef
notice($to_be_overwritten)     #=> "top"
```

現在のローカルスコープとは、上記における bar という class の中のような、現在の文脈におけるスコープのことです。bar の中で定義された$local は、トップレベルのスコープからは参照できません (未定義の変数となる)。

また、$to_be_overwritten という変数に見られるように、トップスコープの変数を現在のスコープで上書きすると、現在のスコープの値が優先されます。しかし、トップスコープでは元の値のままです。

```
class baz {
  $local = "baz"
}

class hoge {
  include baz
  notice($baz::local) #=> "baz"
}

include hoge
```

上記のように、他のクラス内で定義された変数も、::で修飾して参照することができます。ただし、代入することはできません。

resource のデフォルト値

td-agent::plugin には、以下のような記述がありました。

```
Exec {
  path => '/bin:/usr/lib64/fluent/ruby/bin',
}
```

td-agent::plugin には、exec が 2 回出てきます。そのそれぞれについて、同じ内容の attribute を指定するのは DRY ではありません。

そこで、第 5 章で紹介した resource reference を用い、resource type 名の先頭を大文字にして Exec { ... }と書くことで、それ以降に現れる exec に対して attribute のデフォルト値を指定し、記述を省略できます。

「それ以降に現れる」とは、正確には「同じスコープに属する」という意味です。そのスコープは、前述の、変数のスコープと同じです。

条件分岐

Puppet には、いくつかの条件分岐のための構文が用意されています。

```
if $ensure == 'present' {
  ...
}
elsif $ensure == 'absent' {
  ...
}
else {
  ...
}
```

最もよく使うのは、td-agent::plugin でも使った上記のような if/elsif/else 文でしょう。これは、case 文を使って以下のようにも書き換えられます。

```
case $ensure {
  'present': { ... }
  'absent' : { ... }
  default  : { ... }
}
```

条件が多くなってくると、elsif が増えて可読性を損うので、case 文を使うのがよいでしょう。

その他、詳細についてはドキュメントを参照してください。

http://docs.puppetlabs.com/puppet/latest/reference/lang_conditional.html

まとめ

本章では以下のことを学びました。

- 変数とデータ型
- 文字列中の変数展開
- 変数のスコープ
- resource のデフォルト値
- 条件分岐

Puppet の言語は、複雑なシステムの記述に耐えうるよう、たくさんの機能を持った言語です。しかし、これまでの章、および、本章で学んだ程度の知識があれば、だいたいの必要はまかなえるはずです。manifest を書いていく上で、必要に応じてドキュメントにあたりながら憶えていけば、十分事足ります。

http://docs.puppetlabs.com/puppet/latest/reference/lang_visual_index.html

第19章

システム状態をテストする - serverspec

現実のシステムは複雑です。また、manifest はいちど書いたら終わりではありません。現実のシステムはたえず変化していきます。我々は、そのような複雑さと変化にうまく対応していかなければなりません。

本章では、manifest により記述・構築されるシステム状態を、効果的・効率的にテストする方法を紹介します。

serverspec

プログラミングにおけるテストが、現実のアプリケーションの複雑さと変化に対応するためのものであるように、システム状態についても同様のアプローチが可能なのではないか？ そんな着想に基づいて、宮下剛輔氏により開発されているのが serverspec です。

http://serverspec.org/

serverspec は、以下の特徴を持つシステム状態のテストツールです。

1. RSpec のカスタムマッチャによる宣言的な記法でシステム状態をテストする
2. manifest のシンタクスレベルではなく、Puppet などを適用した実際のサーバに対するテストを行う
3. どの構成管理ツールを使っているかに関わらず、テストを行える

RSpec の強力な語彙とカスタマイズ性をフルに活かして、Puppet がそうであるような宣言的なボキャブラリを提供することで、ちょっとしたルールとマッチャを憶えさえすれば自然に使えるようになり、習得は非常に容易です。

また、Puppet の manifest レベルでのテストではなく、manifest が適用された後の、実際の状態をテストするため、より現実の状態に近いテストが可能です。また、そのことにより、Puppet のみならず、Chef やその他のツールによって構築したサーバのテストにも使える、汎用的なツールとなっています。

SSH ログインの設定

さっそく、これまで書いてきた manifest を適用したサーバに対するテストを書いていきましょう。

serverspec は、対象ホストに対してテストを実行する方法を複数用意しています。今回は、テスト対象のホスト (ここでは Vagrant の仮想ホスト) に SSH ログインし、コマンドを実行することでサーバ状態をテストする方法を採ります。まずは、Vagrant の仮想ホストに `ssh` コマンドでログインできるよう設定しましょう。

今回のテスト対象は、前章までに引き続き、td-agent のクラスタです。`Vagrantfile` のあるディレクトリに移動します。

```
$ cd puppet/cluster
```

Vagrantfile では、以下の通り設定がされています。

- `app` ロールのホスト名は app.puppet-book.local
- `log` ロールのホスト名は log.puppet-book.local

ホスト OS 側から、それらのホスト名で仮想ホストにアクセスできるようにします。まずは `app` ロールから設定します。

```
$ vagrant ssh-config app --host app.puppet-book.local >> ~/.ssh/config
```

上記コマンドの実行により、`~/.ssh/config` に、以下の通り設定が追加されているはずです。

```
Host app.puppet-book.local
  HostName 127.0.0.1
  User vagrant
  Port 2222
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
```

```
  PasswordAuthentication no
  IdentityFile "/Users/kentaro/.vagrant.d/insecure_private_key"
  IdentitiesOnly yes
  LogLevel FATAL
```

ログインできるか確かめてみましょう。

```
$ ssh app.puppet-book.local
Last login: Mon Apr 29 10:59:02 2013 from 10.0.2.2
Welcome to your Vagrant-built virtual machine.
[vagrant@app ~]$
```

ちゃんとログインできたようです。log ロールの仮想ホストに対しても、同じように設定してください。

serverspec のセットアップ

まずはホスト OS に、gem コマンドで serverspec をインストールします。

```
$ gem install serverspec
```

serverspec-init コマンドで、初期設定ファイルを作成します。プロンプトで環境についてたずねられるので、適宜解答してください。

```
$ serverspec-init
Select a backend type:

  1) SSH
  2) Exec (local)
  3) Puppet providers (local)

Select number: 1
```

```
Input target host name: app.puppet-book.local

Select OS type of target host:

  1) Auto Detect
  2) Red Hat
  3) Debian
  4) Gentoo
  5) Solaris

Select number: 2

 + spec/
 + spec/app.puppet-book.local/
 + spec/app.puppet-book.local/httpd_spec.rb
 + spec/spec_helper.rb
 + Rakefile
```

コマンドの実行が終わると、上記のようにいくつかのファイルが作成されます。

nginx の serverspec を書く

それでは、さっそく serverspec を書いていきましょう。今回は、nginx の状態をテストしてみます。

我々は先に、nginx の manifest を書いたわけですが、そこではどういう状態が記述されていたでしょうか。リストアップしてみましょう。

- `nginx` というパッケージがインストールされていること
- システム起動時に、nginx も起動するよう設定されていること
- nginx が起動した状態にあること
- nginx が 80 番ポートでリクエストを待ち受けていること
- `/etc/nginx/conf.d/my.conf` というファイルが存在すること
- `/usr/share/nginx/html/index.html` というファイルが存在すること

ざっと思い出してみるだけでも、これぐらいは挙げられます。これらの状態それぞれに

ついて、テストを書いていきます。

serverspec-init コマンドは、便利なことに、spec/app.puppet-book.local/httpd_spec.rb という、テストファイルのひな形も作成してくれています。これを参考に、nginx のテストを書いていきましょう。

マッチャ

serverspec では、マッチャ (matcher) という、サーバの個々の状態をテストするためのメソッドのようなものを使って、テストを記述していきます。File/Directory、Package、Service など、Puppet の resource type に似た分類がなされた上で、下記ドキュメントに整理されています。

http://serverspec.org/matchers.html

まずは「nginx というパッケージがインストールされていること」という状態をテストします。以降のテストコードは、spec/app.puppet-book.local/nginx_spec.rb に書いていきます。

```
require 'spec_helper'

describe 'nginx' do
  it { should be_installed }
end
```

ご覧の通り、直感的な記法で状態をテストできます。さっそく実行してみましょう (テストを実行する前に、仮想ホスト側で puppet apply コマンドを実行しておいてください)。

```
$ rspec spec/app.puppet-book.local/nginx_spec.rb
.

Finished in 0.1286 seconds
1 example, 0 failures
```

テストが通りました。続いて、

- システム起動時に、nginx も起動するよう設定されていること
- nginx が起動した状態にあること

という状態をテストしましょう。先の記述につけたします。

```
describe 'nginx' do
  it { should be_installed }
  it { should be_enabled   } # 追加する
  it { should be_running   } # 追加する
end
```

再度、テストを実行します。

```
$ rspec spec/app.puppet-book.local/nginx_spec.rb
...

Finished in 0.17448 seconds
3 examples, 0 failures
```

このようにして、先にリストアップした「システムのあるべき状態」を、serverspec に落としていきます。その結果は、以下の通りです。

```
require 'spec_helper'

describe 'nginx' do
  it { should be_installed }
  it { should be_enabled   }
  it { should be_running   }
end

describe 'port 80' do
  it { should be_listening }
end

describe '/etc/nginx/conf.d/my.conf' do
```

```
  it { should be_file }
  it { should contain "server_name  app.puppet-book.local;" }
end

describe '/usr/share/nginx/html/index.html' do
  it { should be_file }
  it { should contain "Hello, Puppet!" }
end
```

あらためて、テストを実行してみます。

```
$ rspec spec/app.puppet-book.local/nginx_spec.rb
........

Finished in 0.2695 seconds
8 examples, 0 failures
```

ちゃんと通ったようですね。これで、我々が先に puppet apply によって適用した状態が、serverspec による外部からの状態テストによっても正しいことが確認できました。

テストが失敗したら？

テストが失敗した時、どういうことが起きるのでしょうか。確かめてみましょう。システムが誤った状態にあることを、故意に nginx を落とすことでエミュレートしてみます。

```
$ vagrant ssh app
[vagrant@app ~]$ sudo service nginx stop
Stopping nginx:                                            [  OK  ]
```

この状態でテストを実行するとどうなるでしょうか。

```
$ rspec spec/app.puppet-book.local/nginx_spec.rb
..FF....
```

```
Failures:

  1) nginx
     Failure/Error: it { should be_running   }
       expected "nginx" to be running
     # ./spec/app.puppet-book.local/nginx_spec.rb:6:in `block (2 levels) in ⏎
<top (required)>'

  2) port 80
     Failure/Error: it { should be_listening }
       expected "port 80" to be listening
     # ./spec/app.puppet-book.local/nginx_spec.rb:10:in `block (2 levels) in⏎
 <top (required)>'

Finished in 0.29048 seconds
8 examples, 2 failures

Failed examples:

rspec ./spec/app.puppet-book.local/nginx_spec.rb:6 # nginx
rspec ./spec/app.puppet-book.local/nginx_spec.rb:10 # port 80
```

なにやらいろいろといわれていますね。

- nginx が起動した状態にあること
- nginx が 80 番ポートでリクエストを待ち受けていること

の 2 点について、システム状態が正常でないようです。nginx を故意に止めたのだから当然です。このように、なんらかの原因でシステム状態が正しくない状態に陥ってしまった場合、テストを実行することで検知することができます。

システムのテストがなぜ必要か

筆者の勤務先では、大量にあるサーバの状態を、膨大な manifest によって記述・管理しています。アプリケーションエンジニアとオペレーションエンジニアを問わず、必要であれば誰もが manifest を変更し、システムに反映します。本章の冒頭で述べた通り、そ

の状態は複雑で、また、アプリケーションへの新機能の追加、社会の変化への対応にともない、その内実は刻々と変化していきます。

そうした、多人数による大規模なシステム構築は、プログラミング言語によるアプリケーション開発と、複雑さと変化への対応が必要であるという意味において、まったく変わるところはありません。そのため、serverspec のような、システムの状態をテストするための仕組みが生み出されたのです。

manifest が複雑なものになってくると、module やロール間の関係が入り組んできて、おもむろに変更を加えたところ、既存の状態を壊してしまうことがよくあります。

筆者の勤務先では、単に開発者の手元でテストを実行するというところから一歩すすんで、serverspec を継続的に実行する環境を構築し、運用しています。筆者もしょっちゅう、継続テストに助けられています。

システム状態の継続テストについては、前述の serverspec 作者の宮下剛輔氏の以下のブログエントリをご覧ください。

Ukigumo と serverspec で Puppet の継続的インテグレーション

前章までに作成したクラスタについて、残りの状態についてもぜひ、読者自身の手でテストを書いてみてください。

まとめ

本章では以下のことを学びました。

- serverspec は、複雑さと変化に対応するための仕組み
- serverspec の使い方
- アプリケーション同様、システム状態についてもテストが必要であること

システム状態をプログラマティカルにテストする serverspec は、始まったばかりのまだまだ新しい取り組みです。是非みなさんの開発の現場で取り入れて、確実なシステム構築に役立てるとともに、プロジェクトにフィードバックを送っていきましょう。

第20章

リモートホストに対して manifest を適用する

前章までは、Vagrant による仮想ホストに 1 台ずつ SSH ログインして `puppet apply` コマンドを実行することで manifest を適用してきました。開発時はそれでも事足りるでしょうが、本番環境にはたくさんのサーバがあり得ます。いちいちログインしてまわるわけにもいきません。

本章ではより実践的に、複数台の Amazon EC2 のホストに対していっせいに manifest を適用することを通して、本番環境へ効果的に manifest を適用する方法を見ていきましょう。

capistrano

複数のリモートホストに対して、一箇所から一気にコマンドを実行する方法や、それに特化したツールにはたくさんのものがありますが、ここでは定番の capistrano を使うことにしましょう。

capistrano は Ruby 製のデプロイツールで、Ruby on Rails アプリケーションのデプロイに便利な機能を持っています。とはいえ、Rails アプリケーション以外のデプロイにも使えますし、単純に、複数ホストへコマンドをいっせいに実行するという目的のためにも便利に使えます。

単に複数のホストに対してコマンドを実行するだけであれば、ホスト OS 側でシェルスクリプトを書くなどして対応することもできますが、ここでは以下の理由により capistrano を採用しています。

- capistrano には採用実績が多数あり、ドキュメントも豊富であること
- 今後の変更可能性にそなえて、より保守性の高いツールを利用する方がよいだろうこと
- manifest を適用するホストを設定ファイル内に決め打ちせず、EC2 の API から取

得するため、Ruby のような言語で書く方が楽であること

capistrano のセットアップ

まずは gem コマンドにより、capistrano をインストールします。

```
$ gem install capistrano
```

また、capistrano の設定ファイルの中で使う aws-sdk ライブラリも、ここでインストールしておきましょう。

```
$ gem install aws-sdk
```

次に、capistrano の設定ファイルを準備します。manifest のルートディレクトリで、capify コマンドを実行してください (capify . と、カレントディレクトリを指定するのを忘れないように)。

```
$ capify .
[add] writing './Capfile'
[add] making directory './config'
[add] writing './config/deploy.rb'
[done] capified!
```

上記の操作で作成された config/deploy.rb を、後述の通り編集します。

Amazon EC2 に本番ホストを用意する

次に、本番環境を想定し、Amazon EC2 にホストを用意しましょう。今回は、app ロールのホストを 2 台用意します。

今回、capistrano により manifest を適用できるようにするための前提として、以下があります。

1. Puppet がインストール済みであること
2. `role` というキーで、`app` または `log` というタグが付与されていること
3. `~/.ssh/puppet-book.pem` を使って、ec2-user で SSH ログインできること

Amazon EC2 の詳しい使い方については、よいドキュメントや書籍が多数ありますので、そちらを参照してください。ここでは、上記の条件を満たす方法についてのみ説明します。

1 の「Puppet がインストール済みであること」について。まずは、なにはともあれ Puppet がインストールされていないことにははじまりません。User Data という仕組みを使うと、インスタンスの作成時に任意のスクリプトを実行できますので、それを用いて Puppet をインストールするのが簡単です。インスタンス作成ウィザードで、以下の画像の通り指定してください。

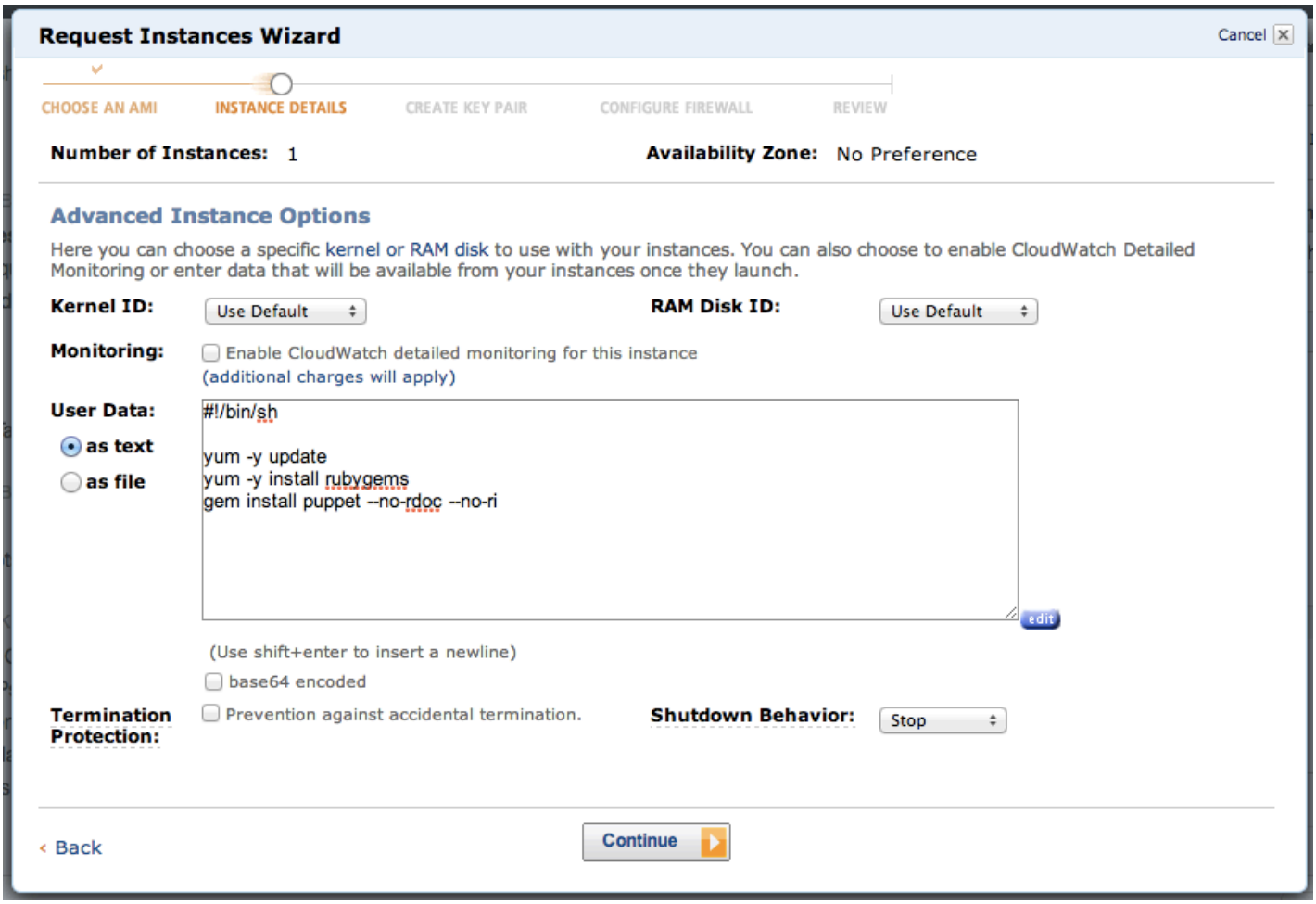

図 20.1　User Data の設定

2 の「`role` というキーで、`app` または `log` というタグが付与されていること」につい

て。後述の capistrano の設定で、タグによって適用すべきロールを判別する設定をしますので、あらかじめ適切なタグを付与しておきます。app ロールの場合、以下の画像のように設定してください。

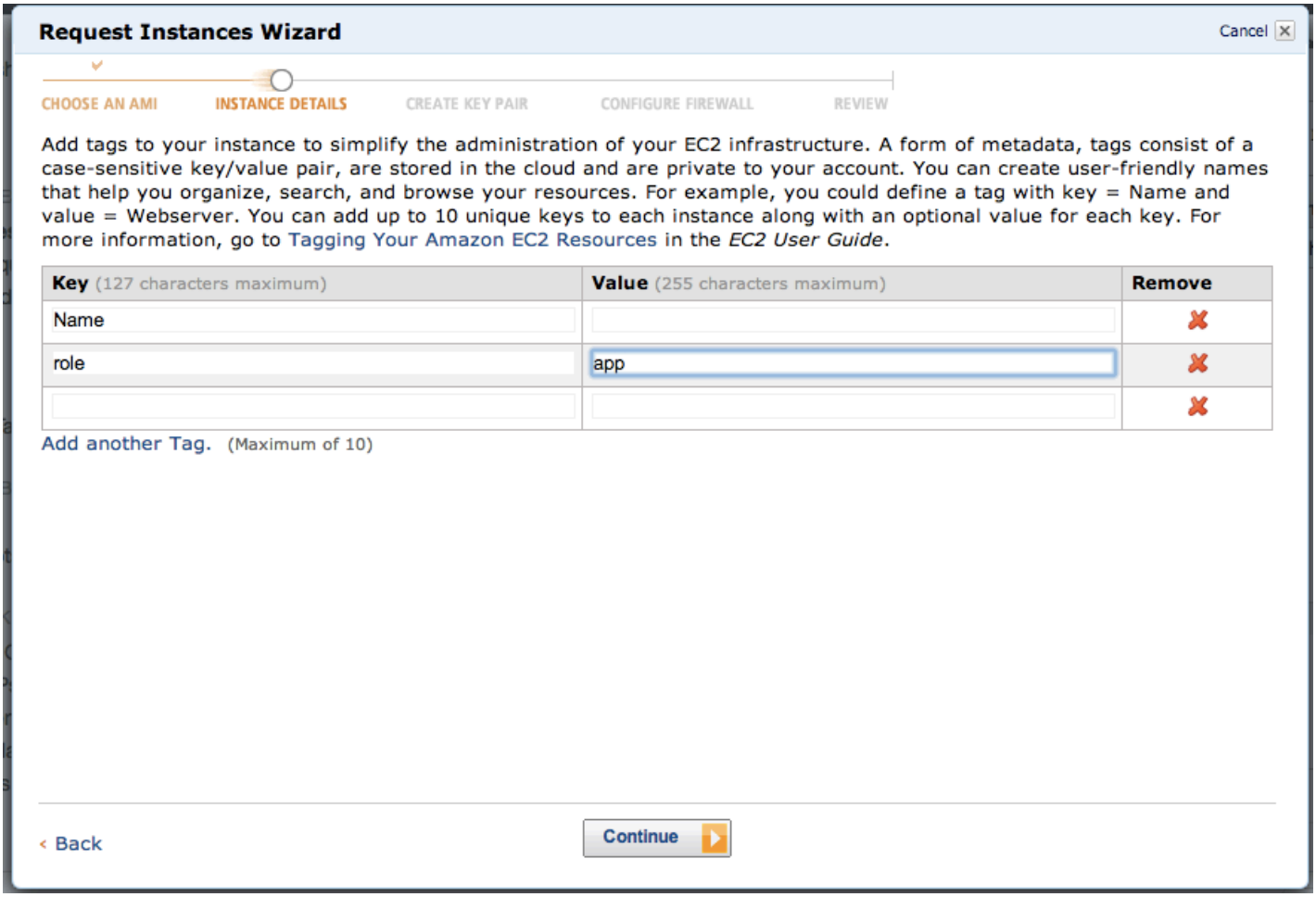

図 20.2　ロールの設定

最後に「~/.ssh/puppet-book.pem を使って、ec2-user で SSH ログインできること」について。EC2 のホストに SSH ログインするのに必要な秘密鍵を、~/.ssh/puppet-book.pem としてコピーしておいてください。後述の capistrano の設定で使います。また、作成したホストに SSH ログインできるよう、セキュリティグループで 22 番ポートへのアクセスを許可するよう、あらかじめ設定しておいてください。

capistrano の設定

準備が整ったところで、capistrano の設定ファイルを準備しましょう。config/deploy.rb を、以下の内容に修正します。

```
require 'aws-sdk'
require 'capistrano/ext/multistage'

set :application, "puppet-book"
set :repository,  "."
set :deploy_to,   "/tmp/#{application}"
set :deploy_via,  :copy
set :puppet_role, nil

set :access_key_id,     ENV['AWS_ACCESS_KEY_ID']
set :secret_access_key, ENV['AWS_SECRET_ACCESS_KEY']
set :region,            ENV['AWS_REGION']

set :use_sudo, false
default_run_options[:pty] = true
ssh_options[:keys] = [File.join(ENV["HOME"], ".ssh", "puppet-book.pem")]

namespace :puppet do
  namespace :apply do
    task :app do
      set :puppet_role, "app"
      deploy.update
      apply_manifest("app")
    end

    task :log do
      set :puppet_role, "log"
      deploy.update
      apply_manifest("log")
    end
  end
```

```
end

def apply_manifest(puppet_role)
  sudo "puppet apply --modulepath=#{current_path}/puppet/cluster/modules:#{c ⏎
urrent_path}/puppet/cluster/roles #{current_path}/puppet/cluster/manifests/#⏎
{puppet_role}.pp"
end

def ec2_instances(puppet_role)
  ec2 = AWS::EC2.new(
    access_key_id:     access_key_id,
    secret_access_key: secret_access_key,
    region:            region,
  )

  instances = ec2.instances.select do |instance|
    # tag は ["key", "value"] という構造になっている
    name_tag = instance.tags.find { |tag| tag[0] == "role" }
    name_tag && instance.status == :running && (
      !puppet_role || name_tag[1] == puppet_role
    )
  end

  instances.map { |instance| instance.dns_name }
end
```

また、以下の内容で、config/deploy/production.rb というファイルを作成してください。

```
set :user, "ec2-user"

role :production do
  ec2_instances(puppet_role)
end
```

この設定では、以下のことを行います。

- cap production puppet:apply:app とした場合は app ロールの、cap production puppet:apply:log とした場合は log ロールのホストに対して、それぞれに対応する manifest を適用する
- aws-sdk ライブラリを使い、各ロールに対応するホスト情報を、AWS の API 経由で自動的に取得する
- ec2-user でログインし、puppet apply を実行して manifest を適用する

manifest を適用する

まず初めに、AWS の API キーを以下の URL からあらかじめ取得しておいてください。
https://portal.aws.amazon.com/gp/aws/securityCredentials

上記の設定では、それらのキーを環境変数を通して capistrano にわたすよう設定しているのですが、コマンドの実行時に、いちいち環境変数を指定するのは面倒ですので、以下のように、ファイルに記述しておくとよいでしょう (このファイルをリポジトリにコミットすることがないよう、.gitignore に追加しておきましょう)。

aws_environment.sh:

```
export AWS_ACCESS_KEY_ID='あなたの access_key_id' \
export AWS_SECRET_ACCESS_KEY='あなたの secret_access_key' \
export AWS_REGION='ホストを作成したリージョン'
```

上記のファイルを source コマンドによってあらかじめ読み込んでおきます。

```
$ source aws_environment.sh
```

その上で、以下の通り deploy:setup コマンドにより、リモートホストのセットアップを行います。ちなみに、このコマンドは capistrano がデフォルトで提供するものですので、上記の設定には記述されていません。

この操作は、インスタンス作成直後にいちど行うだけでけっこうです (複数回実行しても問題ありません)。

```
$ cap production deploy:setup
    triggering load callbacks
  * 2013-04-24 01:29:18 executing `production'
    triggering start callbacks for `deploy:setup'
  * 2013-04-24 01:29:18 executing `multistage:ensure'
  * 2013-04-24 01:29:18 executing `deploy:setup'
  * executing "mkdir -p /tmp/puppet-book /tmp/puppet-book/releases /tmp/pupp ⏎
et-book/shared /tmp/puppet-book/shared/system /tmp/puppet-book/shared/log /t ⏎
mp/puppet-book/shared/pids"
    servers: ["ec2-175-41-216-81.ap-northeast-1.compute.amazonaws.com", "ec2 ⏎
-54-249-198-208.ap-northeast-1.compute.amazonaws.com"]
    [ec2-54-249-198-208.ap-northeast-1.compute.amazonaws.com] executing comm ⏎
and
    [ec2-175-41-216-81.ap-northeast-1.compute.amazonaws.com] executing comma ⏎
nd
    command finished in 99ms
  * executing "chmod g+w /tmp/puppet-book /tmp/puppet-book/releases /tmp/pup ⏎
pet-book/shared /tmp/puppet-book/shared/system /tmp/puppet-book/shared/log /⏎
tmp/puppet-book/shared/pids"
    servers: ["ec2-175-41-216-81.ap-northeast-1.compute.amazonaws.com", "ec2 ⏎
-54-249-198-208.ap-northeast-1.compute.amazonaws.com"]
    [ec2-175-41-216-81.ap-northeast-1.compute.amazonaws.com] executing comma ⏎
nd
    [ec2-54-249-198-208.ap-northeast-1.compute.amazonaws.com] executing comm ⏎
and
    command finished in 131ms
```

次に、以下の通り puppet:apply:app コマンドにより、リモートホスト (この場合は app ロールのホストのみ) に対して manifest を適用します。

```
$ cap production puppet:apply:app
```

(コマンド出力は長くなるので省略します。実行して確かめてください)

以上で、複数台のリモートホストに対して、そのいちいちに SSH ログインすることなしに、手元から manifest を適用することができました。今回は app ロールのみに適用しましたが、同様の手順で、log ロールに対しても manifest を適用できます。各自で実際

に試してみてください。

まとめ

本章では以下のことを学びました。

- EC2 インスタンスの作成
- capistrano の設定
- capistrano を用いて、複数台のリモートホストに manifest 適用する方法

以上で、manifest の書き方とその適用方法については終了です。おつかれさまでした。ここまで学んだことを活用すると、複雑なシステム状態を manifest で記述することから本番環境への適用まで、全て自力でできることになります。是非、ご自分のシステムに対して、学習したことを適用してみてください。

第21章

再び、なぜ Puppet が必要なのか？

"Hello, World!"から始まり、より実践的なクラスタ構成まで、ひと通り manifest を書いてシステム記述・構築を行ってきました。本章では、それらの実践を経たいま、「なぜ Puppet が必要なのか？」についてあらためて考えてみましょう。

プログラマの三大美徳

プログラミング言語 Perl の作者ラリー・ウォール氏は、その著書『プログラミング言語 Perl』で「プログラマの三大美徳」として、以下の 3 つを挙げています。

1. 無精 (Laziness)
2. 短気 (Impatience)
3. 傲慢 (Hubris)

読んで字のごとくという表現ではなく、いずれも反語的な意味を持っています。

「無精」とは、何度も同じことをくりかえすようなことを面倒だと思うあまりに、便利なプログラムを書いたり、ドキュメントを書いたりするような性質。つまり「面倒なことを避けるためなら、いくら面倒なことでもやる」という、一見矛盾するような意味です。

「短気」とは、コンピュータが自分の思い通りにならないことをよしとしないがために、「傲慢」とは、他人に自らのプログラムを悪しざまにいわれるのを受け入れられないがために、よいプログラムを書かざるを得ない、そういう性質のことです。

その中でも「無精」が、プログラマにとっての第 1 の美徳として顕彰されています。このことについて、もう少し掘り下げていきましょう。

システム構築は面倒

あらためて考えてみると、システム構築は、まさに「面倒」の固まりです。そう思いませんか？

やるべきことは山のようにあります。タスクを洗い出し、実行し、さらにはあとに続く人々 (その中には未来の自分も含まれるでしょう) のために作業のリストを手順書に残したところで、今度はその手順書のメンテナンスといった、あらたな面倒が増えるばかりです。「第 1 章 なぜ Puppet が必要なのか？」で述べた問題点は、あらかたそれらが「面倒だから」ということに尽きます。

- 手順書を作成するのが面倒
- 手順書を更新するのが面倒
- スクリプト化するのが面倒
- スクリプトが肥大化して保守するのが面倒
- スクリプトの冪等性を確保するのが面倒

また、システムは一度構築すればそれで終わりではありません。一般に以下の 2 軸において、変化への圧力を受けます。

- システム上に構築されるアプリケーション (Web アプリケーションなど) の要件の変化
- OS やライブラリ、パッケージの更新

みなさんが構築するシステムは、普通はその上でなにかしらのアプリケーションを実行するためのものでしょう。であるからには、アプリケーションの要件の変化によって、システム状態へも変化の圧力がかかってきます。また、そもそもそうしたアプリケーション要件の変化は、社会情勢の変化と密接に結びついたものです。アプリケーションが世の中に価値を提供し続けるためには、情勢の変化に適応し続ける必要があるためです。

また、Linux を始めとするオープンソースのプロダクトを、その利用やサポートにかかる料金という意味おいて「無償」で使っている場合には、それらのプロダクトの更新に常に対応していく必要もでてきいます。いつまでも古い、パッチもリリースされないようなバージョンのプロダクトを使い続けるわけにもいきません (もちろん自力で対応できる、あるいは、サポートにお金を払えるならそれでもよいかもしれませんが)。

つまりシステム構築とは、ある一時点での営みではなく、そこからどれだけ将来にわたるか不明な時間軸における「面倒」の固まりなのです。

システム構築に「無精」を導入する

"Infrastructure as code"というスローガンを目にしたことがある方も多いでしょう。伊藤直也氏による『入門 Chef Solo』のサブタイトルにも採用されている、有名なフレーズです。この言葉の含意には様々なものがあると思いますが、本章の文脈に引き寄せ「システム構築にプログラマの美徳を導入しよう」という意味であると解釈したとしても、大きく外してはいないように思います。

上述した通り、多大な「面倒の固まり」であるシステム構築を「コード」の力によって解決していこうという試み。その一助となるだろうのが Puppet や Chef といったフレームワークなのです。

筆者の勤務先では、アプリケーション開発者も Puppet の manifest を書きます。とはいえ、いちからシステム状態のすべてを記述するというわけではありません。基本的には、最初のシステム構築時にオペレーションエンジニアが基盤を作った上で、アプリケーション要件の変化に対応していく部分については、アプリケーション開発者も積極的に manifest を書いていくというスタイルを採っています。

多人数によるシステム構築においては、構築時のいちいちが手順書のような実行不可能なものに頼って行われていては、不便でしかたありません。それが正しいのか正しくないのか、実行できないことには「面倒」は解決しないからです。スクリプトによる手順化にしても、変化していく環境の中で、状態を壊さないよう保ちつつ、大規模なシステムを記述していくのは至難の業でしょう。

Puppet のようなシステム記述に特化したフレームワークは、それらの問題を、冪等性の確保などの「面倒」をフレームワーク内で隠蔽しつつ「システム状態をコード化する」というアプローチにより、解決します。一見すると、あらたな記法を憶えなければならないのは、またひとつ「面倒」が増えるだけのようにも思えますが、「無精」の定義を思い出してみると明らかなように、それは「面倒なことを避けるためなら、いくら面倒なことでもやる」という性質の発露に他ならないのです。

「無精」を発揮する

上述の通り、筆者の勤務先ではアプリケーション開発者も manifest を書くわけですが、その際、GitHub/GitHub Enterprise の pull request 機能を活用し、職種に関わらず相互

レビューを行いながら、manifest を育てています。これは、アプリケーション開発者が、アプリケーションコードについて日常的にやっていることとまったく同じです。

Web アプリケーションを開発し、サービスとして提供するには、それを動かすためのサーバが必要です。また、ふだんの機能開発においても、新機能のために新しいパッケージやミドルウェアを使いたいのでインストールするといったことはよくあることでしょう。そのような場合に、いちいちオペレーションエンジニア (多くの場合、山積みのタスクを前に忙しくしている) にお願いして導入してもらうのは「面倒」ではありませんか？

それが「面倒」であるのは、システム構築が、開発/オペレーションを問わず、エンジニアの共通言語である「コード」になっていない場合です。Puppet により manifest 化されていれば、ただそれを編集して pull request を投げれば済む話です。筆者は、主にアプリケーション開発者として仕事をしてきたので、システム構築に関しては (開発に関することと比較すると) それほど詳しくはありません。そのため、pull request を通してレビューをしてもらいながら、必要な機能開発のためにシステム状態を変更していっています。

そのようにして「面倒」を「コード」で解消していくうちに、あらたな面倒にぶちあたるようになりました (かように「面倒」の種は尽きないものです……)。

1. manifest の適用が特定の環境 (開発サーバ等) でしかできないのが面倒
2. 修正箇所が他に悪影響をもたらしていないか検証するのが面倒

いずれも後述の通り、勤務先の同僚らといっしょになって、現在解決しつつある問題です。

モジュール化とローカル開発

1 については、agent/master 構成の便利さを認めつつも、環境セットアップの「面倒」さを避けるため、本書でこれまで説明してきたような、`puppet apply` のみを使って manifest をシステムに適用するという方法で、日々の開発を行っています。また、そのためには manifest の各部分が高度にモジュール化されていないとなりませんので、いわゆる Puppet における module 化を進めると同時に、これまでに説明してきたロールという概念を導入しました。

ふだん、Ruby on Rails などのモダンな開発環境でアプリケーション開発をしている読

者には共感していただけると思うのですが、筆者はどこか別の場所にあるサーバにログインして開発するということを、非常に「面倒」に感じます。できるだけ手元のマシンのみで済ませたい。そのため、本書では Vagrant を使って、すべてを手元のマシン 1 台のみで完結できるスタイルで説明してきました。

前章で示した通り、そのような手法を採ったとしても本番環境への適用も十分に行えますし、現に筆者が勤務先で開発しているサービスも、同様にして日々の開発を行っています。

インテグレーション時の問題を解決する

実際の開発では、td-agent クラスタの箇所で述べたように、おのおののエンジニアがその時点で主に担当する各ロールに注力して、manifest を書いていくことになります。

そうすると、最終的にモジュールやロールが統合され、ひとつのシステムとして適用される際に、自分の変更が他に悪影響を与えてしまっていないかを確認・検証する必要がでてきます。これは、アプリケーション開発におけるインテグレーション時の問題と同様です。モジュール化を進めるには、統合時における問題解決が避けては通れません。それが 2 に述べた「面倒」の内実です。

この問題を解決するのが第 19 章で説明した serverspec です。

Web アプリケーションを開発する際に、インテグレーションテストをまったく行わずに、手動でブラウザをポチポチと操作して、膨大なテスト項目を非常な労力を使って検証していくなんてことはもはやあるまいとは思いますが、これまでのシステム構築は、まさにアプリケーションにおける統合テストの不在そのものの状態だったわけです。

各モジュールやロールがおりなすシステムは、アプリケーションがそうであるのと同様に、複雑なものです。システム記述がコード化されたいま、インテグレーション時の検証についてもコード化することは、いまとなっては自然な発想であるように思えます。これでまた、ひとつ「面倒」が解消されたのでした。

「無精」なエンジニアのための Puppet

面倒を省みず、なにごとにも一生懸命に取り組むというような勤勉さが美徳であることは、いうまでもないことです。しかし我々は、職業としてエンジニアを選択し、社会的な価値を提供するべき存在であるからには、単に勤勉であるだけではなく、結果を残さなけ

ればなりません。そのためには、本章で述べた「無精」であることとという、プログラマ的な美徳が役に立つでしょう。

システム構築におけるあらゆる状況を、可能な限り、エンジニアにとっての共通言語「コード」によって可視化していくこと。システム構築、ひいては、サービスの提供には、技術的・組織的な面でいろいろな「面倒」がついてまわるものですが、それら「面倒」をものともしない強い「無精」によって、よりよい価値を提供できるエンジニアになるために、Puppet はおおいに役立つものと確信しています。

おわりに

筆者の勤務先は、大規模なホスティング事業を、その収益の柱のうちのひとつとして行っています。大量のサーバをいかに効率よく扱えるかは、事業の成否に直結する重要な技術的前提となります。そのため、早い段階から Puppet や Chef を高度に使いこなし、サービス運営を行ってきました。

ホスティング事業にとっては、ユーザに提供するサーバそのものが、Web サービスにおけるアプリケーションのようなものであるともいえます。となると、Web アプリケーションをプログラムするのと同様の感覚でもって、ユーザに直接価値を提供するサーバの構築を「コード」化したいという欲求は、自然なものであるといえるでしょう。

筆者は、現在の勤務先に移ってくるまで、Puppet や Chef について、その存在はもちろん知ってはいたものの、正直いってほとんど興味がありませんでした。以前の勤務先でも Puppet や Chef は高度に使われてはいたのですが、オペレーションエンジニアまかせにして、自分で触るということはほとんどありませんでした。

しかし、maglica(前述 serverspec の開発者・宮下剛輔氏による) という、パーソナルクラウドツールとでもいうべきツールを前提とした開発環境や社内ツールの整備を行っていったり、AWS 等のクラウドを積極的に使っているうちに、これは自分のいままでのやり方を変えてなんとかしないとおいつかないなと自覚し、Puppet や Chef の学習を始めたのでした。

本書は、そのような環境に飛び込んでくる新人エンジニアのための研修資料として企画・作成されたものです。Puppet に関するドキュメントは公式サイトに充実していますが、当然すべて英語で書かれたものですし、その分量の多さゆえに、初学者にとってはどこから手をつけたらいいのか悩ましいでしょう。また、「第 2 章 本書の方針」で述べたような、類書にはない新しい観点から Puppet の入門を書いてみたいという気持ちもありました。

本書は「はじめに」で、以下の目標を掲げました。

> 本書の目標は、この本を読んだ読者が Puppet の基本についてひととおり知り、オペレーションエンジニアの書いた manifest(サーバのあるべき状態を記述した設定ファイルのようなもの。後述) に変更を加えたり、ある程度の規模のものなら自力

でいちから書けるようになったりすることです。

本書をここまで読んでこられた読者が、実際に知識を身につけることができ、今後さらに進んだ実践に取り組んでいく一助になれたとしたら幸いです。

最後になりましたが、本書執筆においてお世話になった方々のお名前を挙げることで、謝辞にかえたいと思います。

まずは、本書に結実した内容について、日々の業務を通して知見を与えてくださったり、レビュワーをつとめてくださったりした同僚の宮下剛輔、柴田博志、黒田良、伊藤洋也、常松伸哉、長谷川浩平、吉田真世登、黒瀧悠太の各氏。

野田達也、内藤建の各氏には、たくさんの誤字・脱字などを指摘、修正していただきました。

本書のカバーは、「いろいろデザイン」の長山武史氏にデザインしていただきました。

伊藤直也氏は、「『入門 Chef Solo』の姉妹本といった体裁で Puppet についての本を書きたいのですが、どうでしょう」という不躾なお願いを快く許可してくださった上に、貴重なレビューもたくさんいただきました。

2013 年 5 月 栗林健太郎 (http://kentarok.org/)

入門 Puppet

2013 年 5 月 2 日 v1.0.0 版発行

著　者　栗林健太郎
発行所　達人出版会