

serverspec: 宣言的記述でサーバの状態をテスト可能な汎用性の高いテストフレームワーク

宮下 剛輔^{†1,†2,a)} 栗林 健太郎^{†1} 松本 亮介^{†3}

概要：システムの大規模・複雑化に伴い，サーバの構築・運用を効率化するために，サーバの状態をコードで記述する手法が数多く提案されている．それらの手法を効率良く扱うプロセスとして，テスト駆動開発の手法をサーバ構築に応用した Test-Driven Infrastructure が提案されている．このプロセスを支援するテストフレームワークもいくつか登場しているが，あるものは特定の構成管理ツールに依存，またあるものは OS 毎の違いを自ら吸収しなければならないなど，汎用性に難がある．そこで，本論文では，特定の構成管理ツールや OS に依存することなく，サーバの状態を汎用的かつ可読性の高いコードでテスト可能なテストフレームワークを提案する．提案手法では，汎用性を高めるために，これまでの OS や構成管理ツール固有の振る舞いを整理して一般化し，汎用コマンド実行フレームワークとして定義する．これにより，運用業務で発生するコマンド群，特に確認作業に関して体系化・抽象化する．続いて，テストコード記述の抽象度を高め可読性を上げるために宣言的な記法で汎用コマンド実行フレームワークを操作できる制御テストフレームワークを定義する．これにより，管理者が OS や構成管理ツールの違いを気にすることなくサーバの状態を容易にテストできるようになり，サーバの運用・管理コストを低減できる．また，フレームワークを用途別に分離して定義することにより，制御テストフレームワークを独自の記述に変更する事も容易である．提案するテストフレームワークを `serverspec` と名付けた．

serverspec: A Versatile Test Framework for Testing States of Servers by Declarative Description

MIYASHITA GOSUKE^{†1,†2,a)} KURIBAYASHI KENTARO^{†1} MATSUMOTO RYOSUKE^{†3}

Abstract: システムの大規模・複雑化に伴い，サーバの構築・運用を効率化するために，サーバの状態をコードで記述する手法が数多く提案されている．それらの手法を効率良く扱うプロセスとして，テスト駆動開発の手法をサーバ構築に応用した Test-Driven Infrastructure が提案されている．このプロセスを支援するテストフレームワークもいくつか登場しているが，あるものは特定の構成管理ツールに依存，またあるものは OS 毎の違いを自ら吸収しなければならないなど，汎用性に難がある．そこで，本論文では，特定の構成管理ツールや OS に依存することなく，サーバの状態を汎用的かつ可読性の高いコードでテスト可能なテストフレームワークを提案する．提案手法では，汎用性を高めるために，これまでの OS や構成管理ツール固有の振る舞いを整理して一般化し，汎用コマンド実行フレームワークとして定義する．続いて，テストコード記述の抽象度を高め可読性を上げるために宣言的な記法で汎用コマンド実行フレームワークを操作できる制御テストフレームワークを定義する．これにより，管理者が OS や構成管理ツールの違いを気にすることなくサーバの状態を容易にテストできるようになり，サーバの運用・管理コストを低減できる．また，フレームワークを用途別に分離して定義することにより，制御テストフレームワークを独自の記述に変更する事も容易である．提案するテストフレームワークを `serverspec` と名付けた．

^{†1} 現在，株式会社 paperboy&co.
Presently with paperboy&co., Inc.
^{†2} 現在，帝京大学 理工学部 情報科学科 通信教育課程
Presently with Department of Information Science Correspondence Course, Faculty of SCIENCE and ENGINEER-

ING, Teikyo University
^{†3} 現在，京都大学 情報科学研究科，京都市
Presently with Graduate School of Informatics, Kyoto University
a) gosukenator@gmail.com

1. はじめに

安価な UNIX ライク OS を搭載したサーバが普及し、TCP/IP により異なる OS を搭載したサーバ同士がネットワーク接続可能になった。これによりシステムが大規模化・複雑化し、UNIX シェルにより書かれたプログラムに代わり、サーバの設定を宣言的なコードで扱う構成管理手法が提案された。その実装として CFEngine[1] が登場した。その後様々な構成管理ツールが生み出されているが [2]、2005 年の Puppet の登場 [3] と 2006 年の Amazon EC2 の登場 [4] をきっかけに “Infrastructure as Code” という概念が台頭した。この概念における “Infrastructure” はアプリケーションを載せるためのインフラを意味し、OS やミドルウェアといったソフトウェアレイヤーを含む。インフラをコードで扱うことから、アジャイルソフトウェア開発 [5] と同様の手法がサーバ構築・運用にも適用できるのでは、という発想が生まれ “Agile infrastructure and operations” [6] という流れが生じている。

“Agile infrastructures and operations” を効率よく実践するためのプロセスとして、テスト駆動開発 [7] の手法をサーバ構築・運用に応用した “Test-Driven Infrastructure” [8] というプロセスが提案されており、このプロセスを支援するテストフレームワークがいくつも登場している [9][10][11][12][13][14]。これらのうち、ChefSpec[9]、rspec-puppet[10] は、構成管理ツール固有の言語で書かれたコードの内容をテストするのみで、実際にコードをサーバに適用した結果はテストしない。そのため、単体テストとしては利用できるが結合テスト用途には利用できない。Cucumber-chef[11]、minitest-chef-handler[12] は Chef という特定の構成管理ツールに依存している。そのため、Chef 以外の構成管理ツールでは利用することができない。Test Kitchen[13] や rspec-system[14] は、テスト用 VM の作成、テスト用 VM への構成管理ツールの適用、テストの実行をトータルで行う統合テストスイートであるが、組み込みのテスト機構は汎用性に乏しく、特定の構成管理ツールに依存していたり、OS やディストリビューション毎の違いを意識したテストコードを書く必要がある。

我々は、テストフレームワークの汎用性を高めるために、構成管理ツール特有の振る舞い、例えば、パッケージのインストールやシステムユーザの作成などを抽出・一般化し、それらをテストするためのコマンドを OS やディストリビューション毎に分離、その上で OS や実行形式の違いを吸収するレイヤーを設けることにより、汎用コマンド実行フレームワークを定義した。続いて、テストコードの記述の抽象度を高め可読性を上げるために、宣言的な記法で汎用コマンド実行フレームワークを操作できる制御テストフレームワークを定義した。これにより、テストコードのメンテナンス性を高め、サーバの運用・管理コストを低

減することができる。また、フレームワークを用途別に分離して定義することにより、制御テストフレームワークを独自の記法に変更することも容易である。例えば、本論文で提案するテストフレームワークでは、テスト記法として RSpec[15] を採用しているが、minitest? に差し替えたり、あるいはまったく独自の記法に差し替えたりすることも可能である。このテストフレームワークを serverspec? と名付けた。serverspec を採用している企業も既に存在する??。

本論文の構成について述べる。2 章ではサーバの構成管理とテスト手法について更に詳しく述べる。3 章では提案するサーバのテスト手法を詳細に述べ、4 章では提案手法の評価について論じ、5 章で結びとする。

2. サーバの構成管理とテスト手法

本論文で提案するテストフレームワークは、構成管理ツールによりサーバ構成を記述したコードのテストをいかに効率よく行うか、という観点から出発している。そこで代表的な構成管理ツールを例に、ツールと言語の特徴、テスト手法ならびに従来手法の問題点について言及する。

2.1 CFEngine から Chef へ

第 1 章で触れた CFEngine は 1993 年に登場した。2005 年には CFEngine に影響を受け、より抽象度と拡張性を高める形で発展させた Puppet[3] が登場、2009 年には Puppet から影響を受け、コードのプログラマブルな側面を強めた Chef? が登場している。CFEngine、Puppet は独自のドメイン固有言語でサーバの構成を記述するが、Chef は実装言語と同じ言語を利用したドメイン固有言語、すなわち Ruby というプログラミング言語そのもので記述する。その特性ゆえ、Chef はシステム管理者よりも開発者に強く訴求し、Amazon Elastic Compute Cloud[4] の様な開発者自身がサーバ構築・運用を行える環境の普及とともに広まりを見せている。

2.2 Chef から Test-Driven Infrastructure へ

CFEngine や Puppet では、記述できることが独自言語の範囲内に収まるため、コードは比較的簡易なものとなる。しかし Chef の構成管理コードは Ruby? というプログラミング言語で記述する。そのためシステムが複雑になるにともない、それを記述するコードも複雑になる。そこでサーバ構成を記述したコードに対し、アジャイル開発におけるテスト駆動開発のプロセスを適用する事で、効率よくコードを記述することができる、といった考えが生まれる。Test-Driven Infrastructure という概念は Chef コミュニティ周辺で発生したものであるが、それは偶然ではなく、このような Chef が持つ言語特性ゆえである。

2.3 Test-Driven Infrastructure におけるテスト手法の分類

Test-Driven Infrastructure におけるテストの種類は、テスト駆動開発における以下の 3 つに分類できる。

- (1) 単体テスト
- (2) 結合テスト
- (3) 受け入れテスト

単体テストは構成管理ツールにおける“モジュール”に対するテストで、実際にコードをサーバに適用する前の段階で行うテストである。結合テストはコードをサーバに適用した後に行うテストで、コードが期待通りにサーバの設定を行ったかどうかをテストする。受け入れテストもコードをサーバに適用した後に行うテストだが、結合テストがサーバ内部の状態をテストするホワイトボックステストであるのに対し、受け入れテストはサーバの外から見た振る舞いをテストするブラックボックステストであるという違いがある。

2.4 従来テスト手法の問題点

単体テストツールとしては Chef には ChefSpec, Puppet には rspec-puppet というツールが存在する。その名が示すとおり、それぞれ Chef と Puppet 専用のツールであり、単体テストツールとしては十分である。しかし、実際にコードをサーバに適用した結果をテストすることはできないため、単体テストツールだけでは不十分である。

結合テストツールとしては、minitest-chef-handler, Test Kitchen, rspec-system が存在する。この内 minitest-chef-handler と Test Kitchen は Chef に依存したツールであり、他の構成管理ツールとともに利用することができない。Test Kitchen や rspec-system は、テスト用 VM の作成、テスト用 VM への構成管理ツールの適用、テストの実行をトータルで行う統合テストスイートであるが、ツール標準のテスト機構は汎用性に乏しく、特定の構成管理ツールに依存していたり、OS やディストリビューション毎の違いを意識したテストコードを書く必要がある。

受け入れテストツールとしては cucumber-chef, leibniz が存在するが、これらは Chef に依存したツールであり、他の構成管理ツールとともに利用することができない。表 1 に各テストツールの比較を示す。

これにより、従来テスト手法では構成管理ツールからの独立性と OS・ディストリビューションの汎用性双方を満たすものが存在しないことがわかる。

3. 提案するサーバテスト手法

3.1 要件の考察

構成管理ツール独立性と OS・ディストリビューション汎用性をいかに満たすかを考察する。

特定の構成管理ツールからの独立性を満たせない理由は

表 1 テストツール比較表

ツール名	テスト種別	ツール独立性	OS 汎用性
ChefSpec	単体	×	
rspec-puppet	単体	×	
minitest-chef-handler	結合	×	
Test Kitchen	結合	×	×
rspec-system	結合		×
Cucumber-Chef	受入	×	
leibniz	受入	×	

2 つある。ひとつは構成管理ツールがもつ OS・ディストリビューション汎用性を利用するために、テストの実装が特定の構成管理ツールに依存していることである。

もうひとつはテストスイートでのテスト用 VM 構築フェーズが、特定の構成管理ツールのみ対応していることである。

OS・ディストリビューション汎用性が満たせないのは、テストがシェルコマンドを直接記述する実装になっており、OS・ディストリビューションの違いをテストコードを書く者自らが意識しないといけなからである。

この考察から、提案するテスト手法に必要な要件は以下の通りとなる。

- (1) テストの実装を特定の構成管理ツールに依存しない
- (2) テストスイートではなくテストのみに特化する
- (3) OS・ディストリビューションの違いを利用者に意識させない

3.2 要件を満たすための手法の提案

考察した要件を満たすために、まずは OS・ディストリビューション毎にコマンドを分離し、統一的な API でコマンドを呼び出すことができる汎用コマンド実行フレームワークを定義する。このフレームワークではまず構成管理ツール固有の振る舞い（パッケージインストール等）を抽出する。そして振る舞いをテストするための API を定義する。更に API から呼び出されるコマンドを OS・ディストリビューション毎に定義する。API とコマンド群の間には OS・ディストリビューションを判別して自動で適したコマンドを返すレイヤーを設ける。これにより運用業務で発生するコマンド群、特に確認作業に必要なコマンド群の体系化・抽象化を行う。

次に、テストコードの記述の抽象性を高め可読性を上げるために、宣言的な記法で汎用コマンド実行フレームワークを操作できる制御テストフレームワークを定義する。このフレームワークではまず記法の定義を行う。次に記法内の各命令と実際に呼び出す汎用コマンド実行フレームワークの API メソッドをひもづける。

汎用コマンド実行フレームワークと制御テストフレームワークの仕組みおよびその関係を図 1 に示す。

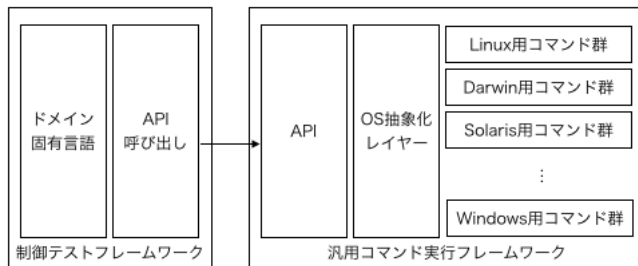


図 1 汎用コマンド実行フレームワークと制御テストフレームワークの仕組みと関係

```
describe file("/etc/password") do
  it { should be_file }
end

describe file("/tmp") do
  it { should be_directory }
end

describe file("/var/run/unicorn.sock") do
  it { should be_socket }
end

describe file("/etc/httpd/conf/httpd.conf") do
  its(:content) do
    should match /ServerName www.example.jp/
  end
end
```

図 2 serverspec によりファイルに対するテストを実行するためのコード

```
describe user("root") do
  it { should exist }
  it { should have_uid 0 }
  it { should belong_to_group "root" }
  it { should have_home_directory "/root" }
  it { should have_login_shell "/bin/bash" }
end
```

図 3 serverspec によりシステムユーザに対するテストを実行するためのコード

```
describe group("root") do
  it { should exist }
  it { should have_gid 0 }
end
```

図 4 serverspec によりシステムグループに対するテストを実行するためのコード

3.3 提案手法の実装

提案手法に基づき実装した汎用コマンド実行フレームワークを SpecInfra?, 制御テストフレームワークを serverspec? と名付けた。実装は図 1 に基づいて行っているが、ここでは実装内部の詳細には触れず、実装の結果としてどのようにテストコードが記述できるようになったかを、図 2, 図 3, 図 4, 図 5, 図 6, 図 7, 図 8, 図 9 に示す。

```
describe package("httpd") do
  it { should be_installed }
end

describe package("serverspec") do
  it { should be_installed.by("gem") }
end

describe package("serverspec") do
  it do
    should be_installed.by("gem").with_version("0.14.3")
  end
end
```

図 5 serverspec によりパッケージに対するテストを実行するためのコード

```
describe service("ntpd") do
  it { should be_enabled }
  it { should be_running }
  it { should be_monitored_by("monit") }
end
```

図 6 serverspec によりサービスに対するテストを実行するためのコード

```
describe port(80) do
  it { should be_listening }
end

describe port(80) do
  it { should be_listening.with("tcp") }
end

describe port(53) do
  it { should be_listening.with("udp") }
end
```

図 7 serverspec によりポートに対するテストを実行するためのコード

```
describe interface('eth0') do
  it { should have_ipv4_address("192.168.10.10") }
  its(:speed) { should eq 1000 }
end
```

図 8 serverspec によりネットワークインターフェースに対するテストを実行するためのコード

```
describe zfs('rpool') do
  it { should exist }
  it do
    should have_property {
      'mountpoint' => '/rpool',
      'compression' => 'off'
    }
  end
end
```

図 9 serverspec により ZFS に対するテストを実行するためのコード

4. 提案手法の評価

提案手法を実装した serverspec を採用している企業がい

```
@test "The package apache2 is installed" {
    dpkg-query -f '${Status}' -W apache2 \
    | grep '^install ok installed$'
}

@test "The apache2 service is running" {
    service apache2 status
}

@test "Port 80 is listening" {
    netstat -tunl | grep ":80 "
}
```

図 10 bats による Ubuntu 上でのテストコード

くつか見受けられる??。また、同種ツールとしてとりあげた Test Kitchen や rspec-system には、ツール標準のテスト機構を serverspec で置き換える busser-serverspec?や rspec-system-serverspec?が存在する。他にも、Vagrant?と連携して VM のテストを行う vagrant-serverspec?というツールが存在する。このように、serverspec は単体利用だけではなく、同種ツールの一機能として取り込まれたり、他種ツールと連携する形で利用が広がっている。

serverspec がなぜ同種のツールと比べて広く使われているのかを考察する。まずひとつは特定の構成管理ツールに依存していないことが挙げられる。そのためどの構成管理ツールを利用していても serverspec は利用できる。それだけにとどまらず、構成管理ツールを利用していない場合でも serverspec を利用することができる。そのため特定の構成管理ツール依存のテストツールと比べて利用の間口が広い。

特定の構成管理ツールに依存していないということは、テスト対象のサーバに特定の構成管理ツールを入れる必要がないということでもある。serverspec はテスト対象サーバで sshd が動いてさえいれば、Ruby すら入れる必要がない。そのため特定の構成管理ツール依存のテストツールと比較して利用の障壁が低い。

広く使われている理由の二つ目は記法の汎用性と抽象度の高さである。汎用性を高めたため、OS・ディストリビューションの違いを気にすることなくテストを容易に書くことができる。また抽象度が高いためテストコードの可読性が高く、メンテナンス性が高い。例として、Test Kitchen では標準で bats?を利用してテストを書くが、bats による Ubuntu?上でのテストコードを図 10 に示す。また、同じ内容のテストを Solaris?向けに書く場合の例を図 11 に示す。serverspec によるテストコードは、OS が何であっても図 12 で示すようなコードになる。このように、提案手法では OS の違いを意識することなく、テストコードを記述することができる。

また別の例として、/etc/sudoers が他人から読めないことをテストするコードの例を示す。bats を利用する場合は

```
@test "The package apache2 is installed" {
    pkg list -H apache2
}

@test "The apache2 service is running" {
    svcs -l apache2 | egrep '^status *online$'
}

@test "Port 80 is listening" {
    netstat -an | grep LISTEN | grep ".80 "
}
```

図 11 bats による Solaris 上でのテストコード

```
describe package("apache2") do
  it { should be_installed }
end

describe service("apache2") do
  it { should be_running }
end

describe port(80) do
  it { should be_listning }
end
```

図 12 serverspec によるテストコード

```
@test "/etc/sudoers is not readable by others" {
  ls -l /etc/sudoers | egrep '^.....-..'
}
```

図 13 bats による/etc/sudoers が他人から読めないことをテストするコード

```
describe file("/etc/sudoers") do
  it { should_not be_readable.by("others") }
end
```

図 14 serverspec による/etc/sudoers が他人から読めないことをテストするコード

図 13 に示すコードとなる。serverspec を利用する場合は図 14 に示すコードとなる。このように、bats の場合はテストコードだけでは何をテストしているのか判別しにくい。ため、説明用のテキストが必要となる。一方 serverspec はテストコードだけでテスト内容が理解できるため、別途説明用のテキストを必要としない。

構成管理ツール依存を排除し、汎用的であるのに加え、単機能でもある。ゆえに他ツールとも組み合わせやすく、従来からあるテストツールや多種ツールに取り込まれる形での利用も広がっている。

弊社では serverspec を採用することによって、古くからある Puppet マニフェスト、すなわち Puppet の記法で書かれた構成管理コードをリファクタリングしようという動きが現場エンジニアの間で活発になっている。

とは言え、利用の広がりについては、きちんとした調査の結果ではなく、定量的な評価は今後の課題である。

5. まとめ

本論文では、Test-Driven Infrastructure を支援するための従来テストフレームワーク手法の問題点について指摘し、解決するために必要な要件として、構成管理ツール独立性と OS・ディストリビューション汎用性を提案した。また、これらの要件を満たすために、汎用コマンド実行フレームワークと制御テストフレームワークに分離して定義する手法についても提案した。これらふたつの実装例として SpecInfra と serverspec を紹介した。

serverspec の登場によって Test-Driven Infrastructure というプロセスが国内でも徐々に認知され、この分野の今後の発展が期待される。また SpecInfra を基盤とした、serverspec よりも優れたテストフレームワーク実装の登場や、確認作業以外の運用業務に必要なコマンド群を体系化することによるテスト以外への SpecInfra の応用、例えば構成管理ツールへの応用なども今後期待される。

今後の課題として、提案手法による運用効率の向上の定量的評価ができていないので、評価手法について検討するとともに、手法に関する助言を募りたい。

参考文献

- [1] Burgess, M.: CFEngine: a system configuration engine, , available from http://cfengine.com/markburgess/papers/cfengine_history.pdf (accessed 2014-01-20).
- [2] Wikipedia: Comparison of open-source configuration management software, , available from http://en.wikipedia.org/wiki/Comparison_of_open-source_configuration_management_software (accessed 2014-01-20).
- [3] Kanies, L.: Puppet: Next-Generation Configuration Management, *USNIX; login*, Vol. 31, No. 1 (2006).
- [4] Amazon Web Services, I.: Release: Amazon EC2 on 2006-08-23, , available from <http://aws.amazon.com/releasenotes/Amazon-EC2/353> (accessed 2014-01-20).
- [5] Cunningham, W.: アジャイルソフトウェア開発宣言, , 入手先 <http://agilemanifesto.org/iso/ja/> (参照 2014-01-20)
- [6] Debois, P.: Agile infrastructure and operations: how infra-gile are you?, *Agile, 2008. AGILE '08. Conference*, pp. 202–207 (2009).
- [7] Beck, K.: *Test Driven Development: By Example*, Addison-Wesley Professional (2002).
- [8] Nelson-Smith, S.: *Test-Driven Infrastructure with Chef, 2nd Edition*, O'Reilly Media (2013).
- [9] Vargo, S.: ChefSpec, , available from <http://code.sethvargo.com/chefspec/> (accessed 2014-01-20).
- [10] Sharpe, T.: rspec-puppet, , available from <http://rspec-puppet.com/> (accessed 2014-01-20).
- [11] Nelson-Smith, S.: Cucumber-Chef, , available from <http://www.cucumber-chef.org/> (accessed 2014-01-20).
- [12] Calavera, D.: minitest-chef-handler, , available from <https://github.com/calavera/minitest-chef-handler> (accessed 2014-01-20).
- [13] Heavy Warter Opertaions, L.: Test Kitchen, , available from <http://kitchen.ci/> (accessed 2014-01-20).
- [14] Barber, K.: rspec-system, Puppet Labs (online), available from <https://github.com/puppetlabs/rspec-system> (accessed 2014-01-20).
- [15] RSpec: RSpec, , available from <http://rspec.info/> (accessed 2014-01-20).