1K-02

ChefScript: 運用ワークフロー記述を可能とする構成管理記述言語

青山　真也†　　廣津　登志夫†

法政大学情報科学部†

1. はじめに

現在，データセンタなどの大規模なサービス基盤では，ネットワーク機器やハイパーバイザ上に展開した複数の仮想マシンなどで複雑に構成されている．こうした環境では，サービス基盤を構成するノードの手動構築が困難となるため，サービス基盤の状態を宣言的なコードで記述しておき，自動的に構築する枠組みである Infrastructure as Code の概念が広まってきている [1]．現状では，Infrastructure as Code の概念の元となった構成管理ツールでシステム構成を記述して運用している．しかし，Chef・Puppet・Ansible に代表される構成管理ツール [2]では，サーバの『状態』をコード化することは可能であるが，サーバの『状態推移』をコード化することができておらず，完全には Infrastructure as Code を実現できていないという問題が存在する．そこで本研究では，運用業務に関するコード化管理に着目し，運用業務のモデル化及び記述性の設計・実装を行う．

2. Chef のアーキテクチャ

Chef では，設定されるべきサーバ状態の詳細定義は Recipe(図 1 左)と呼ばれており，通常はある程度汎用的に記述された Recipe に，JSON 形式で保存された各ノード固有の属性値を与えることで記述が各ノードに適用される(図 1)．設定時にサーバに配置する元データとして File/Template を用意することができる．更に暗号化や階層化されたデータを扱うために Databag と呼ばれる簡易データベースを使うこともできる．各ノードの JSON Attributes には使用する Recipe，各 Recipe には使用する File/Template や Databag が記述されている(図 2)．

Chef は冪等性と呼ばれる性質を持ち，ノードへのコード適用が何回行われても JSON で設定した属性の示す『状態』となり安定的である．一方で，冪等性はノードを JSON で設定した属性に構築するだけであることを意味するため，状態を見ながら時系列に『状態推移』させることができない．例えば，時系列操作を必要とする典型的な運用業務であるローリングアップデートの場合，複数台の更新対象について状態の確認や待機で同期を取りながら 1 台ずつ順番に更新を適用する．しかし，既存の Chef のシステムでは，こうした時間軸によって変化する状態を記述することが出来ない．

ChefScript: A Workflow Description Language for System Configuration Management
† Faculty of Computer and Information Sciences, HOSEI University

```
package "memcached" do
  action :install
  version "#{node["memcached"]["version"]}"
end

service "memcached" do
  action [ :enable, :start ]
end

template "/etc/sysconfig/memcached" do
  source "/etc/sysconfig/memcached.erb"
end
```

```
{
  "run_list": {
    "recipe[mainbook::memcached]",
    "recipe[mainbook::nginx]"
  },
  "memcached": {
    "version": "1.4.10"
  },
  "nginx": {
    "version": "1.5.0",
  }
}
```

図 1　構成管理ツール Chef のコード例

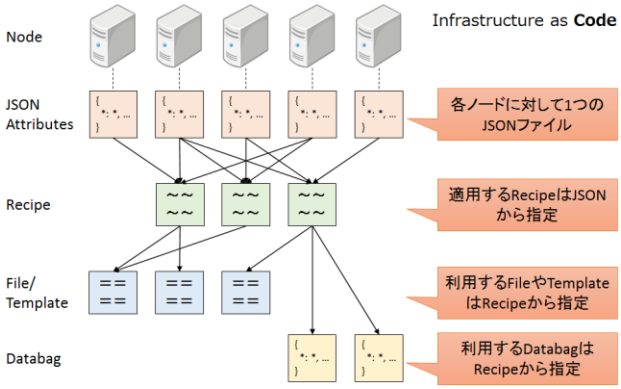

図 2　Chef を構成する主なコードの関係

3. 運用業務のモデル化と記述性の設計

運用業務の分析を行い操作要素に分解した結果，運用業務は図 3 のようなワークフローで行われていると考えられる．運用業務において，特に重要になるのは待機処理である．待機処理とは一定時間待機するだけではなく，memcached サーバ群のローリングアップデート時に「キャッシュに十分なデータが蓄積されるまで待つ」といったシステムの状態を判断する待機処理などを含む．

次に Chef のコードを操作することで『状態推移』を扱うための記述方法について考える．図 3 のワークフローを記述可能とするために Chef に類似した宣言的記述性をもつ DSL を設計する．本システムでは，主に下記の 6 種類の DSL 構文をトップレベルで使用する．

(a) **json**: JSON Attributes の作成 / 変更 / 削除
(b) **recipe**: Recipe の作成 / 変更 / 削除
(c) **create**: File/Template の作成 / 変更 / 削除
(d) **databag**: Databag の作成 / 変更 / 削除
(e) **interval**: 待機処理の内容及び確認間隔
(f) **taskgroup**: 依存関係及び開始時間

一連の運用業務のまとまりを taskgroup DSL を用いて記述し，その中にコード変更処理（json，recipe，create，databag）と待機処理（interval）を記述する形となる．各 DSL 構文は，

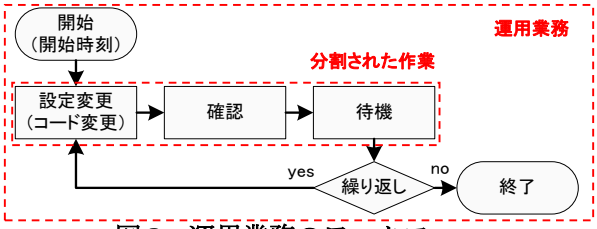

図 3　運用業務のワークフロー

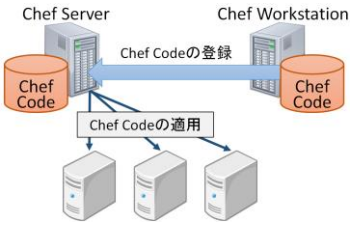

図 4　Chef のアーキテクチャ

[DSL 構文] [識別子] [ブロック構造]
という構造で定義を行う．例えば json 構文では
json “SOME_UNIQUE_NAME” do ... end
という構造となる．また，各ブロック構造には各 DSL 構文で利用する，宣言的に定義するためのメソッドを用いて詳細な処理の定義を行う．

4.　ChefScript のアーキテクチャ

ChefScript は Chef 同様に DSL で実装を行い，ChefScript コードの記述に対応した処理を Chef のコードに行うことで，『状態推移』を実現する．既存の Chef では，各構成管理対象のノードがコードを取得する Chef Server と，Chef Server に保存されているコードを操作する Chef Workstation から構成される(図 4)．また，各ノードが Chef Server にあるコードを適応するタイミングは，「各ノードが定期的に Pull 型で取得する」または，「Chef Server が Push 型で通知する」の 2 通りである．

Pull 型の場合，複数ノードが関係しているコードを変更した際に複数ノードが不規則に更新されてしまい，1 台 1 台を時系列や状態を見て更新することが出来ない．一方，Push 型の場合にはコードの変更と Push 処理をコードとして記述することが出来ない．そのため，Workstation から複数回対話的な制御コマンドを実行するか，そのコマンドで手続き型のプログラムを作成しなければならず，宣言的記述性を持たないため，Infrastructure as Code の概念に沿ったサーバの『状態推移』のコード化が出来ていない．

図 5 にシステム構成を示す．ChefScript は，Chef Workstation 上で動作し，Chef Server を適切に操作することで時系列や状態を判断した変更を行う．最初に ChefScript のコードを読み込み，各 taskpool 及び taskgroup queue に登録する(図 5 (1))．その後，開始時刻になった taskgroup を実行（すなわち複数の task を順次実行）する(図 5 (2)(3))．コードが変更(図 5 (4a))された際には，Chef Server から各 Node に対して Push 方式で変更の適用を行う(図 5 (5))．一方，待機処理(図 5 (4b))では，Workstation 上から任意のプログラムでノードの状態を確認し，状態を判断する．

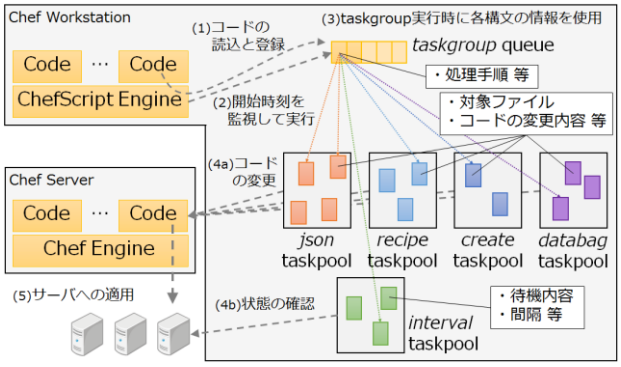

図 5　ChefScript のアーキテクチャ

ChefScript

```
1:  nodes = ["node1", "node2", "node3"]
2:  nodes.each do |nodeX|
3:      json "JSON task #{nodeX}" do
4:          node nodeX
5:          modify do
6:              content["normal"]["memcached"]["version"] = "2.2.3-1.el5"
7:          end
8:          interval "check"
9:      end
10: end
11: interval "check" do
12:     confirm do system "/usr/local/bin/checkcache.sh" end
13:     every 10
14: end
15: taskgroup "Rolling update" do
16:     starts "2015-01-03 04:18:30"
17:     nodes.each do |nodeX|
18:         json "JSON task #{nodeX}"
19:         push nodeX
20:     end
21: end
```

bash

```
1:  #!/bin/bash
2:  nodes="node1 node2 node3"
3:  time=`date`
4:  now=`date +"%Y-%m-%d %H:%M:%S"`
5:  starts="2015-01-08 13:25:15"
6:  remain=$(expr `date -d"${starts}" +%s` - `date -d"${now}" +%s`)
7:  sleep ${remain}
8:  for nodeX in ${nodes}
9:      do
10:     EDITOR=vi knife node edit ${nodeX}
11:     knife job start chef-client ${nodeX}
12:     while true
13:     do
14:         if /usr/local/bin/checkcache.sh; then
15:             break
16:         fi
17:         sleep 10
18:     done
19: done
```

図 6　ChefScript のコードと bash スクリプト

5.　ChefScript のコード記述例

ChecScript のコード例と比較のために同様の処理を bash スクリプトで記述した例を図 6 に示す．ここで示した処理はキャッシュの蓄積を確認しながらローリングアップデートを行うものである．bash スクリプトの場合，対話的なコマンドの利用(図 6 下 10 行目)や手続き的記述が必要であるが，ChefScript では宣言的コードで記述可能である．なお，bash スクリプトにおいて対話的コマンドの利用を避けた場合，標準出力に出力後，変更箇所を記述し，その結果をもとに設定を行う必要がある．その結果，コード量が増大かつ複雑になってしまい非効率化に繋がる．

参　考　文　献

[1] 宮下 剛輔，栗林 健太郎 ，松本 亮介，“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，” 電子情報通信学会技術研究報告，2014.

[2] L. Kanies, ″Puppet: Next-Generation Configuration Management,″ USENIX ;login:, Vol.31, No.1, 2006.