Vol.2015-OS-134 No.22
2015/8/5

ChefScript: ログベースの障害回復性を備えた運用ワークフロー記述言語

青山 真也[†1,a)]　廣津 登志夫[†2]

概要：近年，サービス基盤の大規模化と複雑化に伴い，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構成管理ツールの利用が広まっている．構成管理ツールの 1 つである Chef では冪等性が保証されており，一定時間が経過するとサーバがコードと同等の状態になることが保証されている．一方で，状態を監視しながらの待機などを交えながら，ある変更が反映されてから次の変更を行うといった運用業務を表す時系列記述を行うことができない．そこで本研究では，運用業務を分析・モデル化した結果から，Chef に対して運用業務を実現するために状態間の推移についての記述を拡張した ChefScript の設計及び実装を行った．また，状態間の推移処理を行う際にはログを用いることにより障害時の回復性を備えた実装を行った．

キーワード：Chef，構成管理ツール，DSL，運用業務, Mutable Infrastructure

1. 序論

現在，データセンタなどの大規模なサーバ環境では，KVM（Kernel-based Virtual Machine）や Xen といったハイパーバイザ上に展開した複数の仮想マシンを利用することで，構築コストや運用コストを削減している．また，こうした大規模なサービス基盤（IT インフラストラクチャ）は，複数のサーバとネットワーク機器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]．システム構成の記述に使われる，Chef[2]・Puppet[3], [4]・Ansible[5] といった構成管理ツールでは，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不可能であり，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，Chef に対して状態間の推移についての記述を拡張した ChefScript[6], [7] の設計及び実装を行う．

以下，2 章では Chef のアーキテクチャについて説明する．その後の 3 章では，運用業務の具体例を取り上げて考察を行い，4 章では考察結果を元に言語設計を行う．さらに 5 章でシステム設計，6 章で ChefScript の実装アーキテクチャについて，7 章で ChefScript のコード記述例と考察を述べ，最後の 8 章で本論文をまとめる．

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を Recipe(**図 1** 上) と呼び，通常は汎用的に記述した Recipe に，JSON（JavaScript Object Notation）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)．また，設定時にサーバに配置する元データとなる File/Template や，暗号化・階層化されたデータを扱う Databag と呼ばれる簡易データベースにより，設定を効率的に記述することができる．各ノードの JSON Attributes には使用する Recipe が，各 Recipe には使用する File/ Template・Databag がそれぞれ記述されており，呼び出すような形で利用される (**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』となることが保証される．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

†1 現在，法政大学大学院 情報科学研究科
Presently with Graduate School of Computer and Information Sciences, Hosei University
†2 現在，法政大学 情報科学部
Presently with Faculty of Computer and Information Sciences, Hosei University
a) 15t0001@cis.k.hosei.ac.jp

```
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

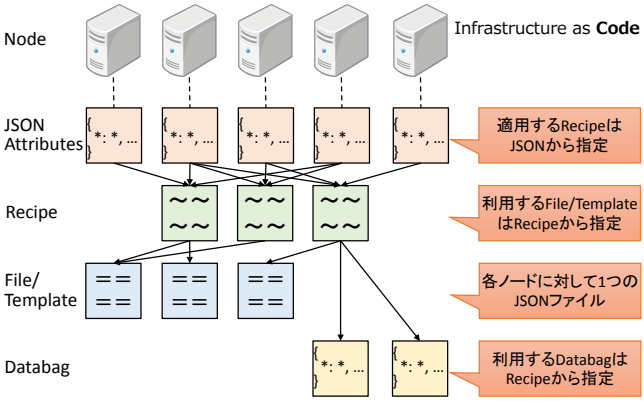

図 2 Chef を構成する主なコードの関係

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を行うこと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．

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，コードの登録を登録するなど，Chef Server を操作する Chef Workstation から構成される (**図 3**)．これらのコンポーネントが連携しあうことで，コードの管理や適用を行っている．まず，コードを Chef Server に登録する方法はコードの種別により，2 つに分けることが可能である．1 つ目は Chef Server のみにコードが存在する場合である．コードの種別としては，JSON Attributes・Databag が該当する．このパターンにおいて既に存在するコードを変更するには，Chef Server からコードを取得し，Chef Workstation で編集した後に再度アップロード処理を行う．2 つ目は Chef Workstation にコードが存在する場合である．コードの種別としては，Recipe・File/Template が該当する．このパターンでは，1 つ目のように Chef Server

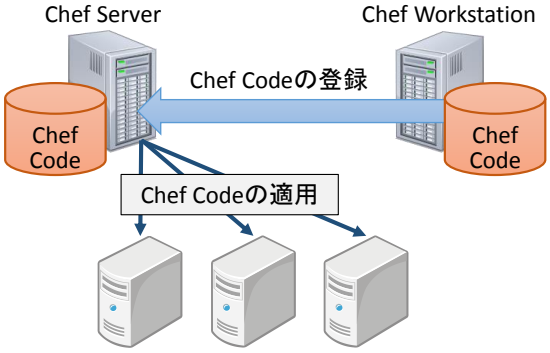

図 3 Chef のアーキテクチャ

からコードを取得することは行わない．そのため，既に存在するコードを変更するには，Chef Workstation に保存されているコードを変更し，Chef Server に複製する処理を行う．次に，ノードが Chef Server にあるコードを適応するタイミングも，Pull 型と Push 型の 2 つに分けることが可能である．Pull 型の場合には，各ノードが定期的に Pull 型で Chef Server からコード取得する．一方，Push 型の場合には Chef Workstation から Chef Server にリクエストを送信し，ノードに Push 型で通知する．

3. Chef による運用業務

状態の変化に対応しながら作業を進める実際の運用業務を Chef で記述する場合について考える．なお，ここではローリングアップデートとロードバランサ配下のサーバ切り替えの 2 つの事例を取り上げる．

3.1 1 種類の作業を繰り返す場合

memcached が導入された複数台のキャッシュサーバにおいて，ローリングアップデートを行う場合の記述について考える．アップデート作業は，JSON Attributes の属性値を変更することで行う．すなわち運用者は，ローリングアップデートを適用する全サーバに対して，下記の手順でアップデートを実行する．

(1) アップデート対象のサーバ群から 1 台を選ぶ
(2) JSON の属性値を変更しアップデート処理を行う
(3) コードの変更を適用し，バージョンを更新する
(4) 再度キャッシュが溜まるまで待機する
(5) 全サーバが完了するまで (1) - (4) を繰り返す

以上の手順を一般化すると，JSON Attributes の属性値を変更して memcached のバージョンを指定する処理は「コードを変更する処理」，コードを適用して memcached のバージョンを更新する処理は「コードを適用する処理」，アップデートを行った memcached にキャッシュが溜まったことを確認する処理は「待機する処理」と考えられる．なお，この運用業務では (2) - (4) が 1 つの繰り返しによって行われる作業となっている．

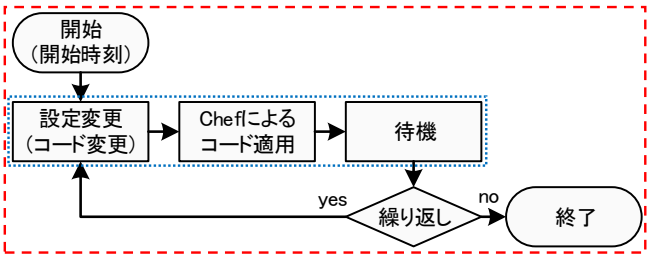

図 4　運用業務のワークフロー

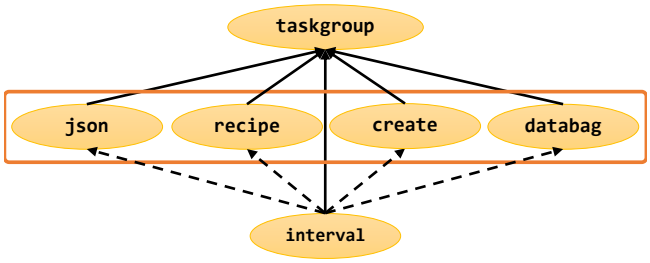

図 5　トップレベル DSL 構文の関係

3.2　2 種類の作業を繰り返す場合

次に，ロードバランサに新しいサーバを追加した後，古いサーバを取り出すことで，ロードバランサ配下のサーバ順次切り替えを行う場合の記述について考える．アップデート作業は，JSON Attributes の属性値を変更し，Recipe 経由で設定ファイル（Template）を書き換えることで行う．すなわち運用者は，既にロードバランサ配下の古いサーバ群及びロードバランサ配下に追加する新しいサーバ群に対して，下記の手順でサーバ切り替えを実行する．

(1) 切り替え対象の新サーバ群から 1 セットを選ぶ
(2) 新サーバを追加するコードに変更する
(3) コードの変更を適用し，新サーバを追加する
(4) 旧サーバを除外するコードに変更する
(5) コードの変更を適用し，旧サーバを除外する
(6) 全サーバが完了するまで (1) - (5) を繰り返す

以上の手順を一般化すると，Template に渡す属性値を変更することでロードバランサ配下のサーバが追加または除外された設定ファイルに変更する処理は「コードを変更する処理」，コードを適用してロードバランサ配下のサーバの追加および除外する処理は「コードを適用する処理」と考えられる．なお，この運用業務では (2) - (3) 及び (4) - (5) のそれぞれが 1 つの繰り返しによって行われる作業となっている．そのため，この運用業務はロードバランサへの追加・ロードバランサからの除外の 2 種類の作業を交互に繰り返し行っていくものとなっている．なお，3.2 節とは異なり，この運用業務のように「待機する処理」が存在しない場合も考えられる．

4.　言語設計

3 章で述べた実際の運用業務の流れから，一般的には「コードの変更」，「コード適用」，「待機」という手順の繰り返しを記述する必要があると考えられる (**図 4**)．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んでいる．

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 4 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

この DSL では，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 (**図 5**)．ただし，コード変更処理の後に同一の待機処理を行う場合を想定し，各 DSL 構文（json, recipe, create, databag）から待機処理を呼び出すことも可能である．Chef によるコードの適用処理については，taskgroup DSL のブロック内で apply メソッドを用いて任意のタイミングで Chef Server 経由で行う．

各トップレベルで利用する DSL 構文は，

[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，各ブロック構造内には各 DSL 構文で利用可能な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．たとえば，コード変更を行う DSL では「どのコード」を「どのように」変更するか，interval DSL では「どの周期」で「何」を確認するか，taskgroup DSL では「いつ」「どのような順序」で適応するかや「どのノード」に更新を適用するかが記述可能である．

各コード変更処理を行う DSL（json, recipe, create, databag）は，それぞれのコードに対して新規作成・変更・削除の 3 種類の操作に対応する必要がある．手動での運用を行う場合，特定のファイルの一部を書き換えたファイルを新たなファイルとして作成する場面が多く存在する．そのため，コード変更処理を行う DSL では，変更元と変更先を指定し，新規作成・変更の操作に対応する．変更元と変更先が同じ指定で，コードが存在しない場合には空のコードから新規作成を行い，コードが存在する場合にはファイル内の変更処理を行う．一方，変更元と変更先が異なる指

定では，ベースとなる変更元のコードから新規作成または変更処理を行う．なお，変更元と変更先が異なる場合の Chef Server 上の管理情報の修正処理は ChefScript 側で行われる．

5. システム設計

ChefScript では Chef 同様に DSL で記述し，ChefScript のコードに対応した Chef のコード変更処理・コード適用処理・待機処理を行うことで，『状態推移』を実現する．その際に考慮する必要がある 2 つの課題について述べる．

5.1 Pull 型と Push 型の比較

ChefScript では，コードの適用を行うタイミングを制御する必要があるため，2 章で述べた Pull 型の適用方法と Push 型の適用方法のそれぞれの場合において制御機構を検討する．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．そのため，Pull 型で実現する場合には，ChefScript による運用業務の実行中は全ノードからのリクエストを拒否し，その時点で更新処理対象のノードからのみリクエストを許可する仕組みが必要である．また，Pull 機構では，ノードにコードの更新が適用されたかの管理情報を持たない．そのため，ノード側に更新が完了した際に ChefScript に対して更新が完了したことを通知する仕組みも必要である．

一方，Push 型の場合にはコードを変更した際に，不規則に更新されることはないが，コードの変更を行った後に運用者が Push 処理を行う必要がある．また，Push 型は Pull 型と異なり，Push 処理を行うコマンドを実行後に正常に更新が完了したかの情報が返される．

Pull 型と Push 型の 2 通りの方法についての比較検討を行う．以後，ノード数を n，Pull 間隔を t_sec として表す．n 台のノードに対して，更新間隔 α_sec のローリングアップデートを考える．この時，Push 型および Pull 型のリクエスト数と実行時間はそれぞれ，

Push 型のリクエスト数: $\{ n \}$
Push 型の実行時間: $\{ \alpha \times n \}$
Pull 型のリクエスト数: $\{ n^2 \times (\alpha/t) \mid \alpha/t \geq 1 \}$
$\{ n^2 \mid \alpha/t < 1 \}$
Pull 型の実行時間: $\{ \alpha \times n + (t - \alpha) \times n \mid t - \alpha \geq 0 \}$
$\{ \alpha \times n \mid t - \alpha < 0 \}$

となる．実際に，n=10，t=40 の環境において，α = 10,20,40,80,160 の場合の Push 型及び Pull 型のリクエスト数と実行時間の推移をグラフ化した結果（**図 6**）を見ると Push 型ではリクエスト数が抑えられている反面，Pull 型では Pull 間隔よりも更新間隔が大きい場合にリクエスト数が増大している．また，更新間隔が Pull 間隔よりも短い場合，Pull 型では実行時間に遅延が発生してしまう．そこで本研究では，Push 型アーキテクチャで実装を行う．

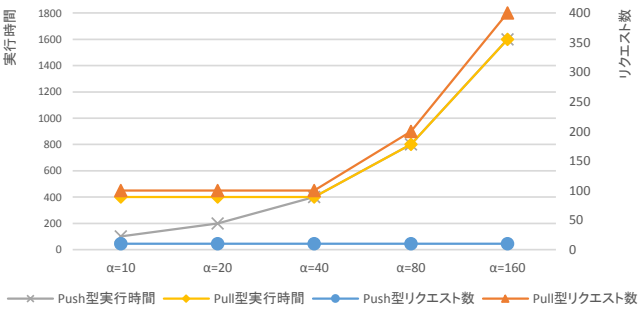

図 6 Push 型及び Pull 型のリクエスト数と実行時間

```
taskgroup “Rolling Update” do
  starts (Time.new() + 3600).to_s
  ...
end
```

```
{
  “Rolling Update”: {
    “starts”: “2011-06-21 03:12:08 +0900”,
    ...
  }
}
```

図 7 実行時に動的に変化するコード例と JSON 化したデータ

5.2 ログベースの障害回復性の保障

ChefScript では，プログラムが起動した際に Ruby 内部 DSL で記述されたコードを読み込む．その後，運用業務となる Taskgroup の実行開始時刻まで待機し，開始時刻になると個々の Task を順次実行する．既存実装 [7] では，この間に障害が発生した場合には再度 1 から処理を行う仕組みとなっているため，問題が発生するケースが 2 つ存在する．

1 つ目は ChefScript が用いる Ruby 内部 DSL を読み込んだ際に，コードの内容が動的に変わるような場合である．例えば，Taskgroup 実行時刻を現在の時刻から 1 時間後などに設定した場合，障害復旧後に DSL を再度 1 から読み込む処理フローを行ってしまうと，復旧後に即時再開されないという問題が生じる（**図 7** 上）．

2 つ目はコード変更を行う DSL を利用する際に，インクリメンタルな処理を記述している場合である．例えば，サーバの移行時には IP アドレスを一定の値だけずらす作業などが行われることがある．その場合，JSON Attributes の IP アドレスが記載された部分を抽出し，第 4 オクテットの値に+50 した値に変更するなどのコード変更 DSL を記述することが想定される．この時，半数のサーバに対する処理は完了したにも関わらず，再度 Taskgroup を 1 から実行してしまうと，前半のサーバ群の IP アドレスは+100，後半のサーバ群の IP アドレスは+50 となってしまう問題が生じる（**図 8**）．

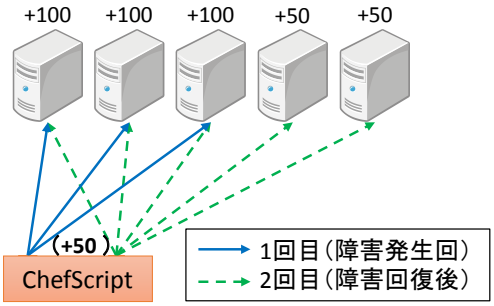

図 8 インクリメンタルな処理の問題

本論文では，これら 2 つの問題を解決するためにログベースのコミット管理を行うことにより障害回復性を保障する．まず，1 つ目の問題は DSL を読み込んだ際に Ruby のプログラムが実行されることにより決定したデータ構造を保存しておき，障害発生時には DSL を再度読み込むのではなく，保存しておいたデータ構造を読み込むことで解決を図る．図 7 上の例では、具体的に「2011-06-21 03:12:08 +0900」のような時刻が保存される（図 7 下）。この際，データ構造を保存する対象は，データ構造を完全に保存できたことが確認できる形式である必要がある．そのため，ChefScript では JSON 形式で保存を行う．JSON 形式では，先頭に {，末尾に } が必要となり，ファイルへの書き込み途中で途切れた場合には構造として解析できなくなる特性を利用する（図 7 下 太字部分）．また，デフォルトの設定では INTERRUPT SIGNAL と TERM SIGNAL による終了時には DSL を再度読み込むが，これらの終了時にも保存しておいたデータ構造から読み込むように設定することを可能とする．さらに，障害前後で DSL ファイルの数や内容が変更されたかを検知できるように，DSL を読み込んだ際に利用したファイルパスと MD5 チェックサムの値を保持しておき，DSL コード自体が変更されていないことも保障する．DSL コード自体が変更またはファイルが追加された場合，保存しておいたデータ構造を利用すると変更分が無視されることとなるため，「変更分を無視して前回のデータ構造で障害回復を行う」か「障害回復性を無視して再度読み込む」かのどちらかを選択できるようにする．

次に，2 つ目の問題は各 Task が終了したことを把握するためにコミット管理を行う．Taskgroup DSL 内に記述される個々の Task は大まかに下記の 4 つに分類される．

(1) コード変更処理（Recipe・File/Template）
(2) コード変更処理（JSON Attributes・Databag）
(3) 待機処理
(4) 適用処理

コード変更処理は，2 章で述べた通り，コードの管理方法によって 2 つに分類した．4 つの処理をシーケンス図にすると**図 9** の通りとなるため，この個々の処理を 1 つのトランザクションとして扱い，コミット管理を行うことで不整合が生じることを防ぐ．(1)(2) の場合，変更前コードを保存し，処理を行った後にコミットログを書き込む．処理の途中で障害時が発生した場合には，保存した変更前コードを用いて一度ロールバック処理を行い，再度変更処理をやり直す．(3) の場合には，待機が完了した後にコミットログを書き込む．待機処理ではインクリメンタルな動作は存在せず，冪等であると仮定し，障害時には再度を 1 からやり直す．(4) の場合には，Chef Server からの完了通知が返ってきた後にコミットログを書き込む．Chef では冪等性が保証されているおり，複数回適用処理が行われても問題がないため，障害時には再度適用処理を 1 からやり直す．

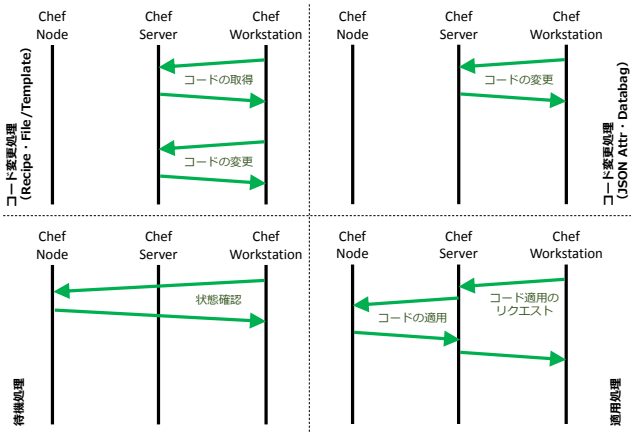

図 9 ChefScript による各処理のシーケンス図

6. 実装

図 10 にシステム構成を示す．ChefScript は, Chef Workstation 上で Ruby プログラムとして動作し，状態を監視しながら待機処理を行い，Chef Server 上のコードを変更することで，『状態推移』を実現する．最初に ChefScript のコードを読み込み，各 taskpool 及び taskgroup queue にオブジェクトとして登録する (図 10 (1))．その後，開始時刻になった taskgroup を実行（すなわち，紐づけられた複数の task を順次実行）する (図 10 (2)(3))．コード変更処理を行うオブジェクト（json，recipe，create，databag）が実行された際には，Chef Server を操作する knife コマンドを用いて対応する Chef Server のコードが変更される (図 10 (4a))．また，待機処理を行うオブジェクト（interval）が実行された際には，Chef Workstation から任意の Ruby プログラムを動作させることで，各ノードの状態などを監視して待機処理を行う (図 10 (4b))．構成管理対象のノードに対してコードを適用する場合には，Chef Server に knife コマンドを用いて Push 型でノードへの適用を依頼する (図 10 (5))．以上の処理フローで ChefScript は運用業務を実現するため，Chef Server 側の拡張は不要である．ただし，本実装では Push 型で実装を行っているため，各ノードはそれぞれが定期的に Pull することは許可しない．ゆえに，Chef Server のコードを変更及びノードへ適用する際には，全て ChefScript 経由で行う必要がある．

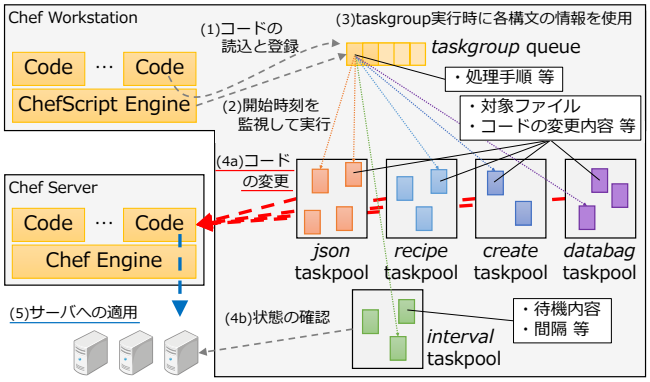

図 10 ChefScript のアーキテクチャと処理フロー

7. ChefScript のコード記述例

ChefScript 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 11** に示す．図 11 の例はキャッシュの蓄積を確認しながら memcached サーバ群の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，対象となるコードを Chef Server から取得するコマンドで標準出力に出力後，Perl・sed・awk などで変更箇所を記述し，その結果をもとに設定を行う必要があるため複雑である (図 11 下 10-16 行目)．しかし，ChefScript では体系的なコード変更の記述ができる点や，宣言的コードで記述可能である点で可読性に優れており，効率的であるといえる．さらに，bash スクリプトの場合にはコードを記述する運用者が何らかの方法で保障するプログラムを図 11 に追加で記述する必要があるのに対し，ChefScript では障害回復性は全て ChefScript コードの処理系が保障する．そのため，ChefScript では，運用者が本来記述したい最小限の情報のみを宣言的なコードで記述するだけで良い．

8. まとめ

構成管理ツールの Chef を対象に，時系列に変化するサーバの状態を記述可能な構成管理記述言語を考案し，設計と実装を行った．既存の Chef では，サーバの『状態』をコード化することは可能であったが『状態推移』のコード化が不可能，すなわち運用業務をコード化することが出来なかった．しかし，ChefScript を用いることでコードの変更内容，待機処理の内容，適用順序などに関する記述が可能となり，運用業務をコード化することが可能となった．さらに，コード変更処理・待機処理・適用処理の個々のタスクの実行を 1 つのトランザクションとしてログを用いて管理することにより，時系列に変化させている最中に強制停止した場合などの障害回復性を備えた実装を行った．

今後はハイパーバイザ・仮想マシンに関する記述性，及び関連研究 [8] を参考にネットワーク機器に関する記述性を実現し，サービス基盤全体の完全なコード化を実現する．

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:   end
9:   interval "Check cache #{nodeX}" do
10:    confirm do
11:     system "/usr/local/bin/checkcache.sh #{nodeX}"
12:    end
13:    every 10
14:  end
15: end
16: taskgroup "Rolling update" do
17:   starts "2015-01-03 04:18:30"
18:   nodes.each do |nodeX|
19:     json "JSON task #{nodeX}"
20:     interval "Check cache #{nodeX}"
21:     apply nodeX
22:   end
23: end
```

bash Script

```
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 ${nodes}
9:   do
10:  result=$(EDITOR=cat knife node edit ${nodeX} 2> /dev/null)
11:  if [ -z "${str}" ]; then
12:    result=$(EDITOR=cat knife node create ${nodeX} 2> /dev/null | sed -e '$d)
13:  fi
14:  echo ${result} > ${resultfile}
15:  perl -pi -e 's|^(     "version":) "1.1"|$1 "1.2"|' ${resultfile}
16:  knife node from file ${resultfile}
17:  knife job start chef-client ${nodeX}
18:  while true
19:  do
20:    if /usr/local/bin/checkcache.sh ${nodeX}; then
21:      break
22:    fi
23:    sleep 10
24:  done
25: done
```

図 11 ChefScript のコードと bash スクリプトの比較

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] Chef Software, Inc.: All about Chef ... - Chef Docs (online), 入手先 ⟨http://docs.getchef.com/⟩ (2014.10.30).
[3] L. Kanies.: *Puppet: Next-Generation Configuration Management*, USENIX ;login:, Vol.31, No.1 (2006).
[4] Puppet Labs.: Puppet Labs: IT Automation Software for System Administrators (online), 入手先 ⟨http://puppetlabs.com/⟩ (2014.12.07).
[5] Ansible, Inc.: Ansible is Simple IT Automation (online), 入手先 ⟨http://www.ansible.com/home⟩ (2014.12.07).
[6] Masaya Aoyama.: ChefScript - A Workflow Description Language with Chef (online), 入手先 ⟨http://masayaaoyama.github.io/ChefScript/⟩ (2015.06.21).
[7] 青山真也, 廣津登志夫.: *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, 情報処理学会 第 77 回全国大会 (2015).
[8] S. Rajagopalan, D. Williams, H. Jamjoom , A. Warfield.: *Split/Merge: System Support for Elastic Execution in Virtual Middleboxes*, 10th USENIX Symposium on Networked Systems Design and Implementation (2013).