Vol.2012-OS-122 No.21
2012/8/2

クラウド基盤ソフトウェアにおける Lineageを利用した障害対策手法の検討

奥畑 聡仁[1]　杉木 章義[2]　加藤 和彦[2]

概要：近年のクラウドコンピューティングの普及に伴い，データセンターの規模が拡大している．これらの多数の計算機資源を管理するデータセンターにおいて，日常的に発生している障害への対策は大きな課題である．本研究では，クラウド環境で典型的に行われるリソースプールから資源を取り出すという操作に着目し，その障害対策を Lineage と呼ばれるモデルを用いて導入する．障害が発生した資源の代わりをクラウド基盤ソフトウェアが自動的に用意し，透過的に障害に対応することを目指す．本研究では，この手法を当研究室で開発しているクラウド基盤ソフトウェアである Kumoi に導入し，いくつかのスクリプトを用いて実験を行い，障害発生時の Lineage の挙動を確認した．

1. はじめに

近年のクラウドコンピューティングの普及に伴い，計算機資源を管理するデータセンターの規模は拡大し続けており，今日では数千台から数万台もの計算機資源を管理している．[1] 多数の計算機を管理する大規模なクラウド環境で問題となるのが，その一部で常に発生している障害である．Vishwanath らの調査 [2] によれば，データセンターで管理している計算機の 8%で，1 年間に 1 度以上，ハードウェア障害が発生している．

本研究が対象とする IaaS（Infrastructure as a Service）型のクラウド環境では，管理者の指示によって，まず多数の資源を持つリソースプールから条件にあう資源を抽出し，次にそこから必要数を取り出して処理を行うという操作が典型的に行われる．このような操作の過程で障害が発生した場合，リソースプールから条件にあう別の計算機を取り出し，それまで行なっていた処理をもう一度適用することで障害に対処することができる．しかし，このような処理を個別の操作ごとに記述していたのでは，実装のコストが高い．

本研究では，Lineage と呼ばれるモデルを用いて，リソースプールから計算機資源を取り出して処理を行う場合の障害対策処理を導入しやすくする．クラウド環境上の多くの操作はこれに該当するため，様々な操作において透過的に障害に対応できることが期待される．

Lineage とは，データに対する処理の過程を記録したモデルである [3]．データをノードとして表現し，データに対する処理をエッジとして表した無閉路有向グラフを利用することで，処理を遡ったり再現したりすることができる．クラウド環境においては，リソースプールから計算機を選択して処理を行うまでの過程を，処理の内容と処理に用いられる引数の値を含めて Lineage によって記録しておく．これにより，障害が発生した際には，リソースプールから条件に適合する別の計算機資源を取り出し，記録しておいた処理を再現して適用することで，代わりとすることが可能である．

この Lineage を用いた障害対策手法を，当研究室で開発しているクラウド基盤ソフトウェアである Kumoi に導入する．Kumoi は計算機資源をオブジェクトに抽象化しており，多数の計算機資源に対する処理を，オブジェクトに対するメソッド呼び出しとリストに対するリスト操作関数によって見通しよく記述することができる．そのため，Lineage による障害対策を比較的容易に導入することができる．それぞれのリスト操作関数に Lineage を用いた障害対策を導入し，クラウド環境における様々な処理に対応できるよう実装した．

実験は，クラウド環境を管理するために想定される仮想マシンを操作するスクリプトを用いて行い，障害発生時の Lineage の挙動を確認した．

本論文は，まず 2 章で本研究が想定する環境について分析する．3 章で提案手法を説明し，4 章で議論を行う．5 章

[1] 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba
[2] 筑波大学システム情報系情報工学域
Faculty of Engineering, Information, and Systems, University of Tsukuba

で実験を行い，6 章で関連研究を紹介し，7 章で今後の課題を述べ，まとめとする．

2. クラウド環境における資源利用の分析

クラウド環境における資源利用は，その多くが以下に示すようなリソースプールから資源を取り出し，処理を適用するという一連の流れによって行われる．

(1) ユーザーが要求した条件に適合する資源をリソースプールから選択する．
(2) 選択した中から必要な数の資源を取り出す．
(3) 取り出した資源に対して処理を適用し，最終的な結果を得る．

クラウド環境で想定される操作の多くはこの流れに対応している．以下にいくつかの例を挙げる．

仮想マシンをデプロイする操作
仮想マシンを 10 台デプロイするという要求に対しては，(1) リソースプールから仮想マシンをデプロイするのに十分なリソースをもつ物理マシンを選択し，(2) 選択した物理マシンから 10 台を取り出して，(3) それぞれに仮想マシンを 1 台ずつデプロイする，という操作を行う．

仮想マシンのマイグレーションによる負荷分散を行う操作
負荷の高い物理マシンから低いマシンに仮想マシンをマイグレーションすることで負荷分散を行う場合には，(1) リソースプールから負荷の高い物理マシンを選択し，(2) 物理マシンのメモリ使用量が一定以下になるような仮想マシンの組を取り出して，3) 負荷の低い物理マシンにマイグレーションする，という操作を行う．

特定のサービスを提供する仮想マシンを起動する操作
例えば Web サーバの機能を提供する仮想マシンを 10 台起動するという要求に対しては，(1) リソースプールから Web サーバ用として配備されており，かつ負荷の低い物理マシンを選択し，(2) それらから 10 台を取り出して，(3) それぞれの物理マシン上で仮想マシンを 1 台ずつ起動する，という操作を行う．

このように，クラウド環境における操作は，多くが前述の一連の流れに当てはめることができる．そのため，このパターンに対して障害対策を導入することで，様々な操作に対応する障害対策を行うことが可能である．

3. 提案

3.1 導入対象のクラウド基盤

本研究が対象とする IaaS 型のクラウド環境では，その管理にクラウド基盤ソフトウェアと呼ばれるミドルウェアを用いる．本研究では Kumoi [4], [5] というクラウド基盤ソフトウェアを用いる．Kumoi は，物理マシンや仮想マシン，ストレージやネットワークデバイスなどの各種計算機資源を抽象化し，オブジェクトにマッピングしている．そのため，仮想マシンの起動，再起動や，物理マシンへの仮想マシンのデプロイなどといった操作を，オブジェクトに対するメソッド呼び出しで行える．また，オブジェクト指向と関数型言語を融合した Scala 言語で実装されており，クラウド環境の多数の計算機に対する操作を，計算機のリストに対するリスト操作関数によって行う．

```
 1: def addVM(pm: HotPhysicalMachine) {
 2:   val vm = pm.vmm.createVM
 3:   vm.name = pm.name+"-"+pm.vms.length
 4:   vm.add(FileDiskAuto("/path/to/"+vm.name+".img")
 5:   vm.add(NatAuto)
 6:   pm.vmm.add(vm)
 7: }
 8:
 9: pms.filter(_.cpuRatio < 0.3)
10:    .filter(_.freeMemory > 512*1024*1024)
11:    .take(10)
12:    .foreach(addVM)
```

図 1 Kumoi のスクリプト例
Fig. 1 Example script with Kumoi

Kumoi によるクラウド環境の操作の例を**図 1** に示す．これは 2 章で挙げた，10 台の仮想マシンをデプロイする例を示している．9 行目の `pms` とは Kumoi が管理する物理マシンのリストを表す．今日のクラウド環境では，このリストは数千の計算機オブジェクトを保持する．まず，この物理マシンのリストから，`filter` 関数で CPU 使用率と空きメモリ容量をもとにリソースに余裕のある物理マシンを選択する．次に，`take` 関数で先頭の 10 台を取り出し，`foreach` 関数でそれぞれに対し関数 `addVM` を適用し仮想マシンを 1 台ずつデプロイする．このように Kumoi は，クラウド環境の管理に必要とされる操作をスクリプトとして簡単に記述することができ，また，オブジェクト指向と関数型言語の特長によって処理の流れがとても分かりやすいものになっている．

3.2 アプローチ

本提案のアプローチは以下の 5 つからなる．それぞれ，以降の節で詳しく説明する．また，**図 2** に例を示す．図 2 は，図 1 の 9 行目以降の処理を図示したものである．リソースプールとみなした物理マシンのリストから条件にあうものを選択した上で，10 台を取り出しそれぞれに仮想マシンをデプロイしようとしたが，1 台に障害が発生しており失敗したところを表している．図中の番号は，以下の番号と対応している．

(1) 処理の連鎖の記録
障害発生時にそれまでに行った処理を再現するために，計算機のリストに対する処理の連鎖を記録しておく．それぞれのリスト操作関数の結果において，リストに適用した処理の内容と，一つ前の結果へのリンクを保持することで実現する．図 2 の初めの `filter` 関数であれば，引数として与えられた CPU 使用率を調

べる関数と，前のリソースプールへのリンクを保持しておく．

(2) 未使用要素の保持
障害が発生した際に代わりとして用いるために，要素を絞り込む関数において対象とならなかった要素を保持しておく．要素を絞り込む関数には，リストの先頭から指定された数の要素を返す take 関数や，リストの先頭から指定された数を除いた要素を返す drop 関数などがある．これらの関数を「**抽出関数**」と呼ぶ．このような抽出関数において，選択されなかった要素を保持しておく．これらの要素のことを「**未使用要素**」と呼ぶ．図 2 の例では，抽出関数である take 関数で未使用要素を保持しておく．ここでは先頭から 10 台を選択しているので，10 台目以降の計算機が未使用要素である．

(3) 遅延評価
抽出関数で未使用要素を保持してから障害が発生するまでの間に時間差が生じると問題が発生することがある．これを解決するために，本アプローチでは，リストに対する処理を遅延評価する．その必要性と評価のタイミングについて，3.2.3 節で説明する．

(4) 障害対応
障害が発生した際に未使用要素から別の計算機を取り出す．取り出した別の計算機に対して，記録してある処理の連鎖を再現して適用することで，代わりとなる計算機を用意できる．図 2 の例では，まず未使用要素を保持している take 関数の時点まで Lineage を遡る．次に未使用要素から別の計算機を取り出し，Lineage を戻りながら記録してある処理を順に適用する．この場合は foreach 関数のみのため，取り出した要素に対して addVM 関数を適用することで，代わりの物理マシンに仮想マシンが正常にデプロイされる．

(5) 順序の保存
リストは順序をもつため，順序を入れ替える関数では問題が発生する．処理を再現する際に順序を保存する方法について 3.2.5 節で説明する．

3.2.1 処理の連鎖の記録

処理の連鎖の記録は，各リスト操作関数の返り値で，関数に渡す引数と前の結果へのリンクを保持することで行う．各リスト操作関数の返り値として，関数に応じた「*Lineage* **リスト**」と呼ばれるオブジェクトを生成する．Lineage リストがサポートする主なリスト操作関数とその返り値を**表 1** に示す．Scala 言語の表記法に従って，「A 型の Lineage リスト」を LineageList[A] のように表記する．Lineage から始まる名前の型はすべて LineageList の一種で，それぞれ引数として与えられた関数や値を保持している．

3.2.2 未使用要素の保持

「未使用要素」とは，take 関数などの抽出関数において選択されなかった要素のことである．抽出関数には**表 2** のようなものがある．それぞれ，関数の結果の内部に未使用要素を保持しておく．障害発生時は，障害が発生した計算機の数だけ未使用要素から取り出し，代わりの計算機として用いる．この挙動によって各関数の定義が元の定義と少し異なるものになる．表 2 に示したように，基本的には元の定義に従うが，障害が発生した要素を除いた上での結果になる。accumulateWhile 関数は，本提案で独自に定義した関数である．takeWhile 関数が，リストの先頭から各要素に対して述語 p を満たす最大の長さのリストを返すのに対し，accumulateWhile 関数では，各要素と，先頭から各要素までの途中リストを述語 p に渡し，p を満たす最大の長さのリストを結果として返す．

表 1 主なリスト操作関数と返り値
Table 1 List operations and their return values

関数	返り値
`filter(p: A => Boolean)`	`LineageFiltered[A]`
`map[B](f: A => B)`	`LineageMapped[B]`
`sortBy[B](f: A => B)`	`LineageSorted[A]`
`take(n: Int)`	`LineageTaken[A]`
`reduceLeft[B >: A](f: (B, A) => B)`	`B`
`exists(p: A => Boolean)`	`Boolean`
`count(p: A => Boolean)`	`Int`
`foreach(f: A => Unit)`	`Unit`

前述のように本研究では，障害が発生した際に，抽出関数で保持してある未使用要素を取り出すアプローチをとる．一方で，障害が発生した際に，Lineage の先頭であるリソースプールからすべての処理を再現して別の要素を取り出すアプローチも考えられる．しかし，どの要素が正常に処理できたか，あるいは障害が発生したかについての管理が煩雑になる．また filter 関数などでは，障害発生時の再評価の際に条件に適合していなかった要素が適合するようになっていたり，既に通過した要素が適合しなくなっていたりする可能性があり，動作の説明が難しくなるため採用しない．

3.2.3 遅延評価

本提案では，Lineage で連結された関数を，必要になった際に一斉に評価を行う．抽出関数において未使用要素を保持したあと，障害が発生するまでに時間差があると問題が発生する可能性があるためである．問題となる場合を図 2 の例で説明する．障害発生時に用いられる未使用要素は，その前の filter 関数で条件に適合した物理マシンをもとに選ばれたものである．そのため，take 関数によって取り出された 10 台に対し，しばらくしてから foreach 関数が実行され障害が発生した場合，代わりとして未使用要素から取り出される物理マシンが，その時点においては filter 関数の条件に適合しない可能性がある．この問題を軽減するため，本研究ではリスト操作関数の処理を遅延評価する．これによって，抽出関数にて未使用要素を保持

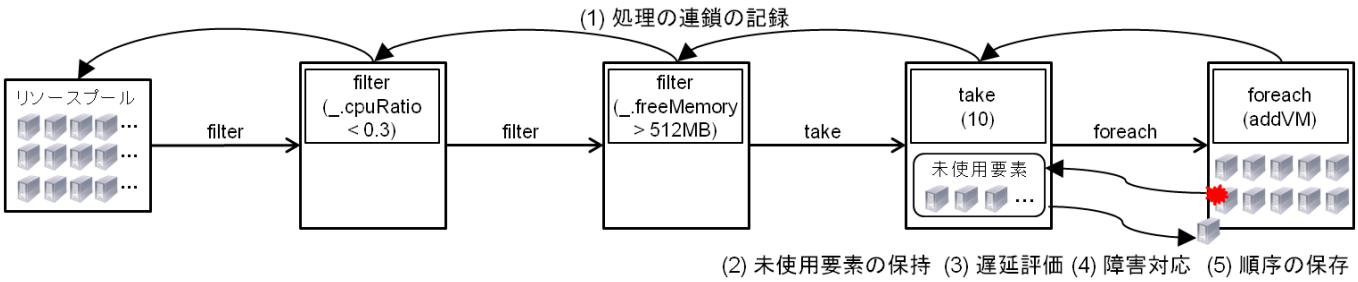

図 2 アプローチの概要
Fig. 2 Overview of the lineage approach for failure handling

表 2 未使用要素を保持する抽出関数
Table 2 List operations which hold unused elements

関数	定義	Lineage における定義
`take(n: Int)`	リストの先頭から n 個の要素を返す	障害が発生した要素を除き，できるだけリストの先頭から n 個の要素を返す
`drop(n: Int)`	リストの先頭から n 個を除いた要素を返す	障害が発生した要素を除き，できるだけリストの先頭から n 個を除いた要素を返す
`takeRight(n: Int)`	リストの末尾から n 個の要素を返す	障害が発生した要素を除き，できるだけリストの末尾から n 個の要素を返す
`dropRight(n: Int)`	リストの末尾から n 個を除いた要素を返す	障害が発生した要素を除き，できるだけリストの末尾から n 個を除いた要素を返す
`takeWhile(p: A => Boolean)`	リストの先頭から述語 p に合致する最大の長さのリストを返す	障害が発生した要素を除き，できるだけリストの先頭から述語 p に合致する最大の長さのリストを返す
`dropWhile(p: A => Boolean)`	リストの先頭から述語 p に合致する最大の長さのリストを除いたリストを返す	障害が発生した要素を除き，できるだけリストの先頭から述語 p に合致する最大の長さのリストを除いたリストを返す
`accumulateWhile (p: (List[A], A) => Boolean)`	リストの先頭から，各要素と各要素までの途中リストに対し，述語 p に合致する最大の長さのリストを返す	障害が発生した要素を除き，できるだけリストの先頭から，各要素と各要素までの途中リストに対し，述語 p に合致する最大の長さのリストを返す

するタイミングと，障害発生時に未使用要素が用いられるタイミングにできるだけ時間差が生じないようにする．

遅延させた処理の評価は，Lineage の末端で行う．以下の 2 つのタイミングを Lineage の末端であるとみなし，評価を行う．

リスト以外を返り値とするリスト操作関数が呼ばれた時
リスト以外を返り値とするリスト操作関数が呼ばれた時には，自動的に評価を行う．Lineage では一つ前のリスト操作関数の結果へのリンクを必要とする．リスト以外を返り値とする場合にはこれまで処理を行なってきたリストに対する情報が失われてしまい，これ以上処理の連鎖を記録することができなくなるためである．表 1 に挙げた中では，リストを集約する `reduceLeft` 関数や，真理値を返す `exists` 関数，整数を返す `count` 関数，`Unit` 型を返す `foreach` 関数などが該当する．

ユーザーが明示的に評価を指示した時
1 つ目のタイミングでは自動的に評価が行われるが，ユーザーは明示的に評価を行うことも指示できる．これは副作用のある操作を考慮している．仮想マシンの起動や，物理マシンへの仮想マシンのデプロイなどといった副作用のある操作は，その時点で評価を行う必要がある．しかし，これらの副作用のある操作を自動的に判別することは難しい．よって現時点ではユーザーが明示的に指示することによって副作用のある操作が行われるタイミングを判断し評価を行う．

3.2.4 障害発生時の動作

障害発生時の動作は以下の 2 つからなる．まず，未使用要素を保持している時点まで Lineage を遡る．これは各リスト操作関数が保持している一つ前の結果へのリンクを順に辿ることで実現する．次に，障害が発生した要素の数だけ未使用要素から取り出し，それらに対して Lineage を戻りながら保持してある処理を順に適用する．未使用要素が見つからなかった場合，別の計算機を用意することはできず，Lineage 導入前のように障害により例外が発生して処理が停止する．

3.2.5 順序の保存

リストは順序をもつため，`filter` 関数や `map` 関数を始

めとする全てのリスト操作関数で，順序を保存する必要がある．つまり，障害が発生して未使用要素から別の計算機を取り出してきた際，その計算機が元あったリスト内の位置に挿入する必要がある．そのために，未使用要素はそれぞれ，未使用要素として保持された時点での自身の位置を保持する．**図 3** に例を示す．この図で，角丸な四角はオブジェクト，中に書かれた数字（1〜10）はオブジェクトの ID を表す．`take` 関数によって保持された未使用要素は，自身のリスト中の位置を表す情報を保持している．障害が発生して未使用要素が取り出された際は，この情報に基づいて適切な位置に挿入する．

また，`sortBy` 関数などリストの順序を変更する関数では複雑な問題が発生しうる．図 3 では，リソースプールから取り出したオブジェクトに対して，`sortBy` や `map` を繰り返し，最終的な `map` 関数での処理の際に，6 番目の要素で障害が発生したところを表している．`sortBy` 関数においてオブジェクト ID の下に書かれた値は，それぞれの評価関数 `f`，`g` による計算結果である．この時，未使用要素から取り出した要素（9 番）を，他の要素の評価値を考慮した上で適切な場所に挿入する必要がある．

そのため，各関数において評価を行った際にその評価値を保持しておく．リストの順序を変更しない関数では，リスト内の各要素の位置を評価値として用いる．リストの順序を変更する関数には**表 3** のようなものがあり，例えば `sortBy` 関数では，評価関数による計算結果を保持しておく．障害発生時には，未使用の要素の評価値と，保持してある評価値のリストを比較し，挿入するべき位置を決定する．図 3 の 1 つ目の `sortBy` 関数においては，9 番の要素の評価関数 `f` による評価値は 0.3 であったので，保持してある評価値のリストから，2 番目の要素の後に挿入するよう位置情報を更新する．これによって，障害発生時の処理を経ても順序が正しく保存される．

4. 議論

Lineage を用いた障害対策手法では，いくつかの点において今後の拡張が可能なため，ここで議論を行う．

4.1 抽出関数の戦略

要素を絞り込む抽出関数では，クラウドの特性に応じた様々な工夫が可能である．Vishwanath らの調査 [2] によると，一度でも何らかのハードウェアの障害を起こした計算機は，一度も障害を起こしていない計算機に比べて障害が発生する確率が高くなる傾向がある．よって，クラウド環境における物理マシンの障害の起こりやすさには偏りがある．この観点から，要素を絞り込む抽出関数には以下の様な戦略も考えられる．

`takeByRandom(n: Int)`

リストの中からランダムに要素を選択する方法である．選択される計算機がばらつくため，障害の起こりやすさに偏りがあるという観点からは，障害に強い選び方であるといえる．

`takeByScore(n: Int)`

リストの中から特定の評価軸によるスコアリングの結果によって選択する方法である．例えば，マシンの使用期間が短いほど良い，新しいデータセンターに配置されているマシンほど良い，ハードディスクを沢山搭載しているマシンほど良い，などといった具合である．先述の Vishwanath らの調査では，計算機が格納されているデータセンターの場所や計算機の製造元などに依って障害の発生率に差が出ることが示されている．そのため，障害の発生率に影響を与えると思われる評価軸に基づいて選択することで，障害に強い選択が可能である．

これらの抽出関数は別の観点からも議論が必要である．通常の `take` 関数の場合，返り値は *deterministic* に決まる．つまり，同じリストに対しては何度実行しても同じ結果が得られる．`take` した要素に対する `exists` 関数，`count` 関数，`reduce` 関数などの処理は，毎回同じ要素に対して処理を行うので結果を予測しやすい．反面，処理に時間がかかる要素があると毎回同じだけの時間がかかる．一方で，上述した 2 つの方法では，前者はランダムに，後者はその都度計算した評価値に基づいて選択するため，返り値は *non-deterministic* になる．つまり，同じリストに対してであっても結果が毎回異なる．その結果に対する `reduce` 関数などの結果も毎回異なり，結果を予測しにくくなる反面，これらの関数による選び方を工夫することで，処理に時間がかかる要素を避けるなどして全体の処理時間を短くすることも可能である．

4.2 リストの分割や結合

これまでは，Lineage の連鎖が 1 つの流れの場合について説明してきた．これに対し，複数の Lineage が合流したり，反対に複数の流れに分割されることもありうる．関数の例としては，条件によってリストを二分する `partition` 関数や `splitAt` 関数，2 つのリストからそれぞれの要素のペアのリストを生成する `zip` 関数などがある．これらの関数で Lineage を形成するためには，それぞれの要素が，どちらのリストに分けられたか，あるいは，どちらのリストのものであったかを記録しておく必要がある．現在の実装では，リスト操作関数の結果である `LineageList` が保持している前の結果へのリンクを，リストの要素毎に持たせることで実現できるが，未実装であり今後の課題とする．

5. 実験

Lineage 導入による障害発生時のクラウド基盤ソフトウェアの挙動を確かめるために実験を行った．実験には，

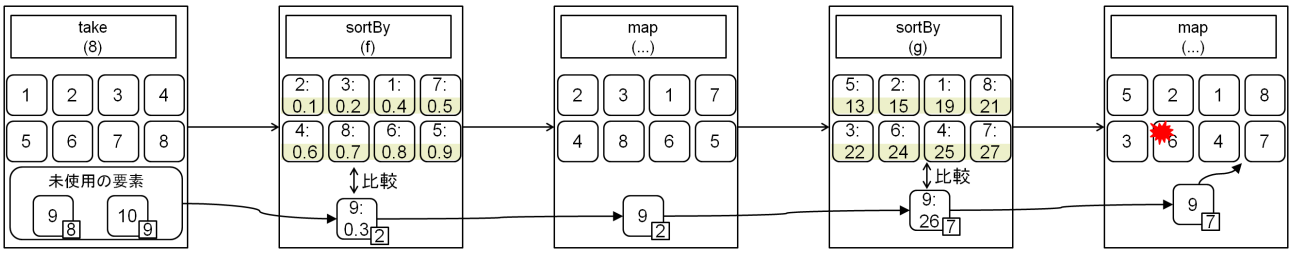

図 3 リストの順序を変更する操作の例

Fig. 3 Example of reordering list elements

表 3 要素の順序を入れ替える関数
Table 3 List operations which reorder elements

関数	保持しておく値	障害発生時の動作
`sortBy[B] (f: A => B)`	ソート後の順での，評価関数による評価値	未使用要素の評価値を計算し，保持してある評価値のリストから挿入するべき位置を決定
`reverse`	逆順での，各要素の元のリストでの位置	未使用要素の元のリストでの位置から挿入するべき位置を決定

クラウド環境を管理するために想定される仮想マシンを操作するいくつかのスクリプトを用いた．なお，以下の節はそれぞれ 2 章で挙げた例に対応している．

実験環境としては同一性能のブレードサーバ 20 台を用いた．それぞれのサーバは，デュアル Xeon 3.60GHz CPU, 2GB RAM, 32GB の SCSI 接続によるディスクを搭載し，すべて 1000BASE-T で接続されている．ソフトウェアは，CentOS 5.7, Java 1.6.0_31, Scala 2.9.1.final を使用した．

5.1 実験 1：仮想マシンのデプロイ

10 台の物理マシンに 1 台ずつ仮想マシンをデプロイする場合で実験を行う．各物理マシンに保存してある仮想マシンの起動に必要なディスクイメージのひとつ（物理マシン `ibm34` 上のもの）を意図的に破壊して実験を行った．実験に使用したスクリプトは，3.1 節の図 1 のものとほぼ同じであるが，12 行目で `foreach` 関数ではなく `map` 関数を用い，デプロイした仮想マシンを返すよう関数 `addVM` を変更した．あわせて，`map` 関数で評価を開始するための `eval` メソッドを呼び出すよう変更した．結果を**図 4** に示す．なお結果においては，リストの表示や，同じ内容の例外の表示を一部省略している．

物理マシン `ibm34` 上では，仮想マシンの起動に必要なディスクイメージが壊れており例外が発生してデプロイに失敗している．代わりを用意するために，抽出関数である `take` 関数の時点まで Lineage を遡り，未使用要素から `ibm42` を取り出し，記録してある処理を再現して適用することで，必要な数の仮想マシンがデプロイされた．

5.2 実験 2：仮想マシンのマイグレーションによる負荷分散

負荷の高い物理マシンから低いマシンに仮想マシンをマイグレーションすることで負荷分散を行う．使用したスクリプトを**図 5** に示す．このスクリプトは，9 行目から `filter` 関数で負荷の高い物理マシンを抽出し，それぞれに対して `foreach` 関数で 13 行目以降の処理を適用する．`sortBy` 関数でデプロイされている仮想マシンをメモリ使用量の多い順に並び替え，`accumulateWhile` 関数（3.2.2 節参照）で物理マシンの空きメモリ容量が一定以上になるような数を取り出し，それらを `foreach` 関数にて負荷の低い物理マシンにマイグレーションする．

実験結果を**図 6** に示す．図 6 は，物理マシン `ibm30` における 13 行目以降の処理の結果を表している．ここでは，マイグレーション先の 1 つである物理マシン `ibm34` に意図的に障害を注入して実験を行った．図 5 の 7 行目の処理で `30-3` をマイグレーションする際にマイグレーション先の物理マシン `ibm34` で障害が発生したため，抽出関数である `take` 関数の時点まで Lineage を遡り，未使用要素の中から物理マシン `ibm32` が選択され，正常に仮想マシンがマ

```
scala> deploy(10, pms)
List(ibm30, ibm31, ibm32, ibm33, ..., ibm48, ibm49)
 | filter
List(ibm30, ibm31, ibm32, ibm34, ..., ibm49)
 | filter
List(ibm30, ibm32, ibm34, ibm35, ..., ibm48)
 | take
List(ibm30, ibm32, ibm34, ibm35, ..., ibm41)
 | map
exception (map) on ibm34       // 障害による例外発生
  org.libvirt.LibvirtException: POST operation failed:
    xend_post: error ... Disk isn't accessible")
List(ibm42, ibm43, ..., ibm48) // 未使用要素
 | take(alt)
List(ibm42)
 | map(alt)

res10: List[kumoi.shell.vm.HotVM] = List(30-2, 32-3,
       35-1, 36-4, 37-2, 38-2, 39-2, 40-1, 41-2, 42-4)
```

図 4 実験結果：仮想マシンのデプロイ

Fig. 4 Result of virtual machines deployment

```
 1: def balance(pms: List[HotPhysicalMachine]) {
 2:   def migrateToAny(vm: HotVM) =
 3:     pms.lineage
 4:        .sortBy(_.cpuRatio)
 5:        .filter(_.freeMemory > vm.memory)
 6:        .take(1)
 7:        .foreach(vm.migrateTo(_))
 8:
 9:   pms.lineage.filter{ pm =>
10:        pm.cpuRatio > 0.8 ||
11:        pm.freeMemory < 256*1024*1024 }
12:      .foreach{ pm =>
13:        pm.vms.lineage
14:          .sortBy(-_.memory)
15:          .accumulateWhile{ (vms, vm) =>
16:            vms.foldLeft(0L)(_+_.memory) <
17:              pm.freeMemory + 256*1024*1024 }
18:          .foreach(migrateToAny)
19:      }
20: }
```

図 5 スクリプト：仮想マシンのマイグレーションによる物理マシンの負荷分散

Fig. 5 Script for load balancing by VM migration

```
scala> balance(pms)
-- ibm30 --------------------------------
List(30-1, 30-2, 30-3, 30-4, 30-5, 30-6)
 | sortBy
List(30-5, 30-6, 30-3, 30-2, 30-4, 30-1)
 | accumulateWhile
List(30-5, 30-6, 30-3)
 | foreach
30-5: ibm30 -> ibm36
 : (省略)
30-6: ibm30 -> ibm36
 : (省略)
30-3: ibm30 -> ibm34
List(ibm30, ibm31, ibm32, ibm33, ..., ibm49)
 | sortBy  // migrateToAny 関数 4 行目以降の処理
 : (省略)
List(ibm34, ibm32, ibm42, ...)
 | take
List(ibm34)
 | foreach
exception (foreach) on ibm34  // 障害による例外発生
  org.libvirt.LibvirtException: Unknown failure
List(ibm32, ibm42, ...)       // 未使用要素
 | take(alt)
List(ibm32)
 | foreach(alt)
30-3: ibm30 -> ibm32
```

図 6 実験結果：仮想マシンのマイグレーションによる物理マシンの負荷分散

Fig. 6 Result of load balancing

イグレーションされた．

現在の実装では，障害が発生した関数の直前の抽出関数から未使用要素を取り出す．そのため，7 行目でマイグレーションを行う仮想マシン 30-3 に障害が発生している場合でも，直前の抽出関数である take 関数から別の物理マシンを取り出して仮想マシン 30-3 のマイグレーションが行われる．これに対しては，障害が発生したオブジェクトに応じてどの抽出関数から未使用要素を取り出すかを判断する必要があり，今後の課題とする．

なお，9 行目の lineage においては抽出関数がないため，filter，foreach 関数中で障害が発生しても代わりを用意することはできず，その時点で例外を投げ処理を停止する．

```
1: pms.lineage
2:    .filter(hasWebServers)
3:    .sortBy(_.cpuRatio)
4:    .take(10)
5:    .foreach{ pm =>
6:      pm.vmm.allVMs.lineage
7:        .take(1).foreach(_.start) }
```

図 7 スクリプト：Web サーバの機能を提供する仮想マシンの起動

Fig. 7 Script for starting VMs of web servers

```
scala> (実験 3 のスクリプト)
 : (省略)
 | foreach  // 物理マシン ibm34 上での 7 行目の処理
34-1 starting...
exception (foreach) on 34-1  // 障害による例外発生
  org.libvirt.LibvirtException: Unknown failure
List(34-2, 34-3, ...)        // 未使用要素 (7 行目)
 | take (alt)
List(34-2)
 | foreach (alt)
34-2 starting...
exception (foreach) on 34-2  // さらに障害による例外発生
  org.libvirt.LibvirtException: Unknown failure
 : (省略)
exception (foreach) on ibm34             // 未使用要素を
  kumoi.impl.lng.LackOfElementsException // 使いきった
List(ibm32, ibm41, ...)      // 未使用要素 (4 行目)
 | take(alt)
List(ibm32)
 | foreach(alt)
List(32-1, 32-2, 32-3, ...)
 | take
List(32-1)
 | foreach
32-1 starting...
```

図 8 実験結果：Web サーバの機能を提供する仮想マシンの起動

Fig. 8 Result of starting VMs of web servers

5.3 実験 3：特定のサービスを提供する仮想マシンの起動

ここでは Web サーバの機能を提供する仮想マシンを 10 台起動する．使用したスクリプトを**図 7** に示す．このスクリプトでは，filter 関数で Web サーバ用として配備されている物理マシンを選択し，sortBy 関数で負荷の低い順に並び替える．take 関数で先頭から 10 台を取り出し，foreach 関数によってそれぞれの物理マシン上で仮想マシンを 1 台ずつ起動する．

実験結果を**図 8** に示す．ここでは，物理マシン ibm34 に意図的に障害を注入し，ibm34 上の全ての仮想マシンが起動に失敗するようにして実験を行った．図 7 の 7 行目で仮想マシンの起動に失敗すると，抽出関数である take 関数まで遡って代わりの計算機を用意しようとするが，代わりの計算機でも起動に失敗し，未使用要素を全て使ってしまった．すると，例外が発生し，今度は 5 行目から抽出関数である 4 行目の take 関数まで遡って別の物理マシン ibm32 を選択し，そちらで正常に仮想マシンを起動した．

6. 関連研究

6.1 Lineage

Lineage については，Bose らの論文 [3] で詳しく調査されている．Lanter [6] は，地理情報システムに Lineage を導入している．データに対する処理の過程での生成物を元データ，中間データ，結果の 3 種類に分類し，それぞれの

種類に応じたメタデータを持たせることで，任意の時点のデータに対し，どのデータにどのような処理を適用して得られた結果であるかを調査することが可能である．また，Woodruff ら [7] は，気象予測のような，膨大な観測データに対して粒度の細かい計算を行う上で，各処理の逆関数を定義し，ある結果から元の観測データを調べる手法を提案した．これによって，計算結果における外れ値から，その結果に影響を与えている観測データを特定することが可能である．

これらの研究は，処理を辿ることによって処理の内容や元のデータを調査する目的で Lineage を用いている．本研究は，障害が発生した際に代わりの要素を用意するために，障害が発生した要素に対する処理を再現する目的で Lineage を用いている点が異なる．

6.2 障害対策

Dryad [8]，Spark [9] は，クラスタ環境における大規模並列演算のためのフレームワークである．前者は処理の流れを無閉路有向グラフで表現するモデルを用い，後者はデータに対する処理の過程を Lineage を形成することで記録している．そのためどちらも，処理中のノードの障害によってデータの一部が欠落したとしても，必要な部分だけ元のデータから再計算して回復できるため，障害に対して効率的に対処することができる．

これらはいずれも，データを計算する際の計算機の障害を対象に Lienage を用いており，障害発生時には別の計算機で元のデータを再計算することで障害に対処する．本研究では，計算機がデータにあたり，データ自体が障害を起こす．そのため，障害発生時には代わりの計算機を用意することで障害に対処する．

7. まとめ

本研究では，Lineage を用いて，リソースプールから計算機を取り出して処理を行う場合の障害対策処理を導入した．リソースプールから計算機を抽出して処理を行うまでの過程を Lineage を形成して記録しておく．障害発生時には未使用要素を保持してある時点まで Lineage を遡り，未使用要素を取り出して Lineage を戻りながら記録してある処理を順に適用する．これによって障害が発生した計算機の代わりの計算機を用意することができた．

実装はクラウド基盤ソフトウェアである Kumoi に対して行った．実験では，クラウド環境を管理するために想定される仮想マシンを操作するスクリプトを用いて行い，障害発生時の挙動を確認した．

今後の課題として，まず，リストの分割や結合における Lineage の分岐や合流への対応など，実装をさらに進める．また，Kumoi が提供する並列分散処理に対応するリスト操作も考える．さらに，より実用的なクラウド環境の操作を用いて実験を行う．

謝辞 本研究の一部は科研費（2230006，22700023）の支援を受けている．

参考文献

[1] L.A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, Vol. 4, No. 1, pp. 1–108, 2009.

[2] K.V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pp. 193–204, 2010.

[3] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys (CSUR)*, Vol. 37, No. 1, pp. 1–28, 2005.

[4] A. Sugiki, K. Kato, Y. Ishii, H. Taniguchi, and N. Hirooka. Kumoi: A high-level scripting environment for collective virtual machines. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pp. 322–329, 2010.

[5] A. Sugiki and K. Kato. An extensible cloud platform inspired by operating systems. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pp. 306–311, 2011.

[6] D.P. Lanter. Design of a lineage-based meta-data base for gis. *Cartography and Geographic Information Science*, Vol. 18, No. 4, pp. 255–261, 1991.

[7] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings. 13th International Conference on Data Engineering, 1997.*, pp. 91–102. IEEE, 1997.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 3, pp. 59–72, 2007.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, 2012.