B-016

# マルチスレッド型エージェントに対応した AgentSphere の強マイグレーション機構の改良
# Improvement of a Strong Migration Mechanism of AgentSphere for Multi-threaded Agents

蓮見 建太†, 疋田 直也†, 甲斐 宗徳†
Kenta Hasumi, Naoya Hikida, Munenori Kai

## 1 はじめに

本研究ではモバイルエージェントの自律性を利用し，専門知識を持たない人でも並列分散処理を利用できるシステムを構築するため，自律型並列分散システム向けのプラットフォームとなる AgentSphere の開発を行っている。AgentSphere は OS やマシンに依存しないで利用できるよう，仮想マシン上でプログラムが実行される Java を実装言語として用いている。そして，エージェントの移動後に移動前の状態から実行再開するために，弱マイグレーションに加え，強マイグレーションにも対応させている。

ただし，Java ではシリアライズ機能を利用することによってヒープ領域内の情報を保存・移動する弱マイグレーションを実現することはできるが，強マイグレーションの実現のために必要となるスタック領域内の情報とプログラムカウンタを保存・移動する機能がサポートされていない。そのため，何らかの手段でこの 2 つの情報を保存する機能を実装する必要がある。

従来の AgentSphere では，ソースコード変換の手法を用いることで Java 上での強マイグレーションを実現してきた。しかし，シングルスレッドエージェントのみに対応しており，マルチスレッドを利用したエージェントに強マイグレーションをさせることができていなかった。また，ソースコード変換は AgentSphere を利用するすべてのエージェントコードに対して実行されるため，変換処理で発生するオーバヘッドを少しでも抑えることが求められる。

そこで本稿では，マルチスレッドエージェントに対応し，かつオーバヘッドの発生を軽減した新たな強マイグレーション方式について提案を行う。

## 2 AgentSphere における強マイグレーション

### 2.1 強マイグレーションの実現方法

既存の強マイグレーションモバイルエージェントシステムには，MOBA[1]などが存在する。これらのシステムでは，Java 仮想マシン（JVM）に対する変更や，ランタイムシステムの拡張やネイティブメソッドの追加を行うことによって強マイグレーションを実現している。

しかし AgentSphere では，一般的な Java コードの実行環境上で稼働することを目指しており，他の関連研究[1]のように特別な実行環境を用意する方法を考えていない。

そこで AgentSphere では特別な実行環境を用いない方法として，ソースコード変換器を用いた強マイグレーションの実現を進めてきた。

### 2.2 ソースコード変換器

ソースコード変換器では，変換対象である強マイグレーション記述されたエージェントのソースコードを，強マイグレーションの動作をする弱マイグレーションコードへと変換することによって Java での強マイグレーションを実現する。

変換内容は図 2.2.1 のようになる。図 2.2.1 左図は変換前のエージェントコードを表している。コードの記述者は，エージェントが実際に行う振る舞いに加え，移動を実行させたい任意のタイミングに強マイグレーション命令を記述しておく。

ソースコード変換器は，変換対象となるソースコード中の強マイグレーション命令の位置を基にしてソースコード変換を行う。図 2.2.1 左図のコードを変換した結果は図 2.2.1 右図のようになる。

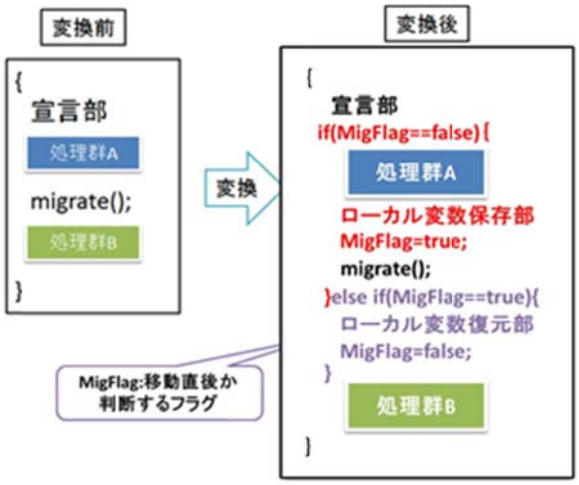


図 2.2.1 ソースコード変換前後の様子

このように強マイグレーション命令の前後を if 文で分割することで移動後のエージェントが移動前の状態から処理を再開することが可能となり，JVM に変更を加えずに強マイグレーションな振る舞いを実現する事が可能となる。

### 2.3 従来方式の問題点

ソースコード変換器を用いることにより JVM 上で強マイグレーションを実現することは可能であるが，大きなオーバヘッドの発生や，他のソースコードに依存関係がある場合のソースコードの変換，マルチスレッドで実行されるエージェントの移動への対応に関して問題が残る。

大きなオーバヘッドが発生する原因は，ソースコードの変換方法にある。ソースコードの変換操作では，変換対象となるソースコードに対して構文解析を行った後，解析結果をデータ構造として保存し，保存したデータ構造に対してソースコード変換に操作を加える。そして，変換が加えられたデータ構造からソースコードの出力を行うことによって，変換後のソースコードを取得することができる。この変換操作を実行する時間がそのままオーバヘッドとなる。

また，エージェントは複数のソースコードのファイルから構成されることが多く，エージェントを構成するファイ

† 成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University

ル中で，migrate メソッドを含むすべてのファイルに対してソースコード変換を行う必要がある。現状，変換器にかけられるソースコードのファイルが，変換処理を行う必要があるものか否かをすぐ判別することはできない。そのため，変換対象となるエージェントコードを構成するすべてのソースファイルに対して構文解析を行い，変換が必要か判断しなくてはならない。加えて，エージェントを構成するソースファイルの数は膨大な数になることが考えられ，エージェントを構成するすべてのソースファイルに対して変換操作を行うことは効率的とは言えず，より良い方法を考える必要がある。

また，AgentSphere のように JVM に変更を加えない方式では，スレッドの外部から実行状態を取得することができない。そのため，あるスレッドによって migrate メソッドのような移動命令が呼び出された時，呼び出し元以外のスレッドも強制的に移動させることが難しく，マルチスレッド記述されたエージェントを他のマシンへと移動させることが困難である。

## 3 JavaFlow を利用した強マイグレーション

### 3.1 既存の問題に対する解決方法の提案

ソースコード変換器を用いた強マイグレーションの実現が抱える課題の解決方法として，Java 言語のバイトコードを変換することによる強マイグレーションの実現を提案する。今回，バイトコード変換による強マイグレーションを実現するにあたり，JavaFlow を利用する方法を考える。

### 3.2 JavaFlow の概要

JavaFlow[2]は，オープンソースのソフトウェアプロジェクトを支援する団体である Apache ソフトウェア財団傘下のトッププロジェクトである Apache Commons で開発，提供が行われている Java コンポーネントである。

JavaFlow は，Java 言語で「実行状態の継続」を実現することを目的としており，そのために必要となるスタック領域内の情報やプログラムカウンタを含めた実行状態の保存・復元を実現するために，プログラムのバイトコード変換を行う。

JavaFlow を用いることにより，プログラムのある時点における実行状態を Continuation と呼ばれるデータ型に保存し，後で保存した状態から実行を再開することが可能となる。このクラスの実行は，クラス実行用の startWith メソッドを呼び出して行う。呼び出されたクラスは run メソッド内の処理を実行する。run メソッド内に中断命令が存在している場合，命令が呼び出された時点での実行状態を Continuation というデータとして保存し，クラス実行用のメソッドの戻り値として返す。保存された Continuation はシリアライズ可能なデータであり，別のマシンへの転送を容易に行うことができる。保存された実行状態の再開は，実行状態再開用のメソッドを呼び出すことで可能となる。また，JavaFlow 自身はすべて Java 言語を用いて実装されており，標準的な JVM 上での動作が可能となっている。

以上のように，JavaFlow を用いることによって，AgentSphere が求める「JVM に変更を加えることなく，かつ，簡単に利用できる強マイグレーション」が実現可能となる。

### 3.3 新方式の利点

ソースコード変換器を用いた方式では，強マイグレーション利用したいコードに対して構文解析や解析結果のデータに対する変換操作など，様々なコード変換処理を行わなければならない。その結果，オーバヘッドが増加するという問題が発生している。対して JavaFlow を利用した方式では，構文解析などの処理を行う必要がないので，コード変換処理を行うのにかかっていたオーバヘッドと変換方式の効率面での大幅な改善が見込める。

また今までの方式では，マルチスレッドで実行されるエージェントの移動に対応できていないという問題が存在した。この点に関しても，実行状態の中断・再開が容易であるという JavaFlow の特性を用いることで，対応が可能である。

## 4 新たな強マイグレーション方式の実装

### 4.1 新たな強マイグレーション方式の概要

JavaFlow を利用してエージェントを生成するには，強マイグレーションコードの記述とバイトコード変換を行う必要がある。

### 4.2 強マイグレーションコードの記述

前述したとおり JavaFlow を用いることで，プログラムの実行状態の中断や再開が容易に実現できる。しかし，強マイグレーションを実現させるため，中断した実行状態を移動させ，移動先で再開させる処理を記述しなくてはならず，エージェントコード記述者にとって負担となる。そこで AgentSphere では JavaFlow を利用し，処理の中断・移動・再開をすべて行う強マイグレーション命令を用意した。ソースコード変換器利用時と同様に，エージェントコード中の強マイグレーションを実行したいタイミングに命令を記述することで利用が可能となっている。

### 4.3 バイトコード変換

続いて，ユーザによって記述された強マイグレーションコードをコンパイルし，.class ファイルを生成する。そして，生成された.class ファイルに対してバイトコード変換を行う。バイトコード変換には，OS などの環境に依存しにくいビルドツールである Apache Ant を用いた。Apache Ant では，XML 文書でビルド時のルールをタスクという形で記述することが可能であり，JavaFlow におけるバイトコード変換用のタスクを用意することで行った。

### 4.4 新たな強マイグレーション方式の問題点

強マイグレーションコードの記述とバイトコード変換を行うことにより，処理の中断・再開が可能となるが，このまま単純に JavaFlow を利用しただけでは保存した実行状態を別マシン上で再開することができない。この問題は，Java が標準に搭載しているクラスローダでは，別マシンから転送されてきた未知のクラスを実行することができないということに起因する。しかし，AgentSphere では，転送されてきた未知のクラスを実行するため，階層型クラスローダを開発し，採用している。階層型クラスローダを用いることで，保存した実行状態を別のマシン上で再開させることが可能となる。

以上のように，JavaFlow を用いることで強マイグレーションの実現が可能となる。しかし，JavaFlow を用いた強マイグレーションマルチスレッドプログラムの移動に

は対応していないため，何らかの対応を行う必要がある。具体的な対応方法は 6 章で説明する。

## 5　新旧強マイグレーション方式の比較

### 5.1　新旧マイグレーション方式におけるオーバヘッドの比較

JavaFlow 利用の新方式とソースコード変換器利用の旧方式の 2 つを比較して行う。比較は，各方式を採用したエージェントが「実行までの事前処理にかかる時間」と「実行後の処理にかかる時間」を計測した結果を用いて行う。

計測は 2 台の PC 間で強マイグレーションを繰り返すことで行った。マシンの性能はそれぞれ，CPU:i5-2400CPU 3.1GHz・メモリ:8GB のものと，CPU:i5-2500CPU 3.3GHz・メモリ:4GB のものを使用した。計測対象として，四則演算など簡単な処理を行う 100 行程度・500 行程度・1000 行程度の 3 つのサイズのエージェントコードを用意した。また，各サイズのコードには，run メソッドの中に強マイグレーション命令(migrate)を図 5.1.1 に示すように適当な間隔で呼び出すように配置した。その呼出し回数は 2・4・6・8・10 回の 5 種類について実験を行った。

```
//Agentの処理を記述するメソッド
public void run(){

        long start = System.currentTimeMillis();

        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        migrate();
        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        migrate();
        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        for(int i=0;i<10000;i++)
                if(i%100==0)
                        System.out.println("i = "+i);
        migrate();
        for(int i=0;i<10000;i++)
                if(i%100==0)
```

図 5.1.1　比較用エージェントコードの一例

### 5.2　エージェント実行の事前処理にかかる時間の比較

今回比較を行う各方式では，エージェントを実行する際，事前処理を行う必要がある。そこで JavaFlow 利用方式におけるバイトコード変換とソースコード変換器利用方式におけるソースコード変換の各事前処理にかかる時間を計測し，比較を行った。図 5.2.1 は計測結果である。

図中の上の表にあるソースコード変換器利用時の結果を見ると，より行数の長いコードを変換すると変換にかかる時間が増加することがわかる。一方，図中の下の表にある JavaFlow 利用時の結果を見ると，コードの行数によって変換時間は大きく変わらず，処理に要する時間もソースコード変換器を利用した場合よりも短いことがわかる。また，両方式に共通していることとして，ソースコード中の強マイグレーション命令数の大小では，事前処理にかかる時間に大きな差が出ないことがわかる。結果を比較すると，JavaFlow を用いた方式の方が短時間で事前処理を行う際に，差が明確に表れた。

| ソースコード変換器利用時 | | コード中の強マイグレーション命令数 | | | | |
|---|---|---|---|---|---|---|
| | 行数 | 2 | 4 | 6 | 8 | 10 |
| サンプルコードの行数 | 100行 | 2640.2 | 2640.2 | 2640.2 | 2640.2 | 2640.2 |
| | 500行 | 3666.7 | 3730.2 | 3513.6 | 3366.5 | 4080.2 |
| | 1000行 | 4798.0 | 4763.4 | 4364.5 | 4933.2 | 5103.3 |

| JavaFlow利用時 | | コード中の強マイグレーション命令数 | | | | |
|---|---|---|---|---|---|---|
| | 行数 | 2 | 4 | 6 | 8 | 10 |
| サンプルコードの行数 | 100行 | 577.3 | 589.6 | 581.4 | 579.2 | 598.2 |
| | 500行 | 581.5 | 589.9 | 577.3 | 578.1 | 596.6 |
| | 1000行 | 587.7 | 583.3 | 578.2 | 598.7 | 601.4 |

（単位：ms）

図 5.2.1　各方式の事前処理にかかる時間

### 5.3　エージェント実行時の処理にかかる時間の比較

ここでは，実際に各方式で強マイグレーションを行うエージェントが処理を完了するまでの違いを調べる。図 5.3.1 は計測結果である。今回計測した結果では，両方式共に 1000 行程度のサンプルコードの実行時間が 500 行の実行時間よりも短いという結果が出た。この点については，原因が不明であるため，参考記録して扱う。

図中の上の表にあるソースコード変換器利用時の結果を見ると，エージェントコードの行数に比例して実行時間が増加することがわかる。また，同じ行数であっても，強マイグレーション命令の呼び出された回数が多いほど処理の終了までにかかる時間が多くなることがわかる。

図中の下の表にある JavaFlow 利用時を見ると，ソースコード変換器利用時と同様に，行数の増加と強マイグレーション命令の呼び出し数に比例して実行時間が増加することがわかる。

結果を比較すると，行数の短いコードを実行する際，両方式における実行時間の差は見受けられなかった。しかし，エージェントコードの行数，及び強マイグレーション命令の呼び出し回数が増加するにつれ，両方式の実行時間の差が大きくなる。そのため，行数が多いか別マシンへの移動機会が多いエージェントには JavaFlow を用いた方が短時間で処理を終了できると考えられる。

| ソースコード変換器利用時 | | コード中の強マイグレーション命令数 | | | | |
|---|---|---|---|---|---|---|
| | 行数 | 2 | 4 | 6 | 8 | 10 |
| サンプルコードの行数 | 100行 | 145.5 | 228.3 | 319.1 | 441.5 | 563.5 |
| | 500行 | 255.2 | 423.6 | 616.5 | 914.5 | 1339.2 |
| | 1000行 | 310.5 | 393.1 | 501.5 | 740.0 | 899.2 |

| JavaFlow利用時 | | コード中の強マイグレーション命令数 | | | | |
|---|---|---|---|---|---|---|
| | 行数 | 2 | 4 | 6 | 8 | 10 |
| サンプルコードの行数 | 100行 | 143.2 | 223.8 | 326.2 | 415.2 | 567.1 |
| | 500行 | 222.4 | 359.6 | 536.0 | 795.0 | 1096.4 |
| | 1000行 | 157.7 | 168.4 | 169.2 | 171.3 | 180.8 |

（単位：ms）

図 5.3.1　各方式における実行時間

### 5.4　考察

AgentSphere では，長時間実行し続けるエージェントや何度も移動を繰り返すエージェントの利用も目指している。そのため，今回の計測において，移動回数が多いエージェントコードを利用する際により有利な結果を示し，また，事前処理の時間も短い JavaFlow を利用する方が良いと考えられる。

## 6　マルチスレッド型エージェントの強マイグレーションへの対応

### 6.1　マルチスレッド型エージェントとは

マルチスレッド型エージェントとは，マルチスレッドプログラムを AgentSphere 上で実行させるためのエージェントの形式である。本形式では，マルチスレッドとして動作させたい各処理をエージェントコードとして記述し，各

エージェントを協調させることでマルチスレッドな動作を実現させる。

### 6.2 マルチスレッド型エージェントの実現方法

AgentSphere では、マルチスレッドで動くエージェントの強マイグレーションを制御するために, エージェントに「実行・確認・停止」という 3 つの状態を用意することにした。このうち,「実行」は自身が行う仕事をしている状態であり,「停止」は仕事を中断し, 移動を待機している状態である。そして,「確認」状態のエージェントは, 自分以外のエージェントの状態を確認する状態である。その際,「停止」状態のエージェントが 1 つでも存在していることを確認した場合, 自身も「停止」の状態へと遷移する。また,「停止」状態にあるエージェントが存在しない場合,「実行」の状態へと遷移し, 自身の仕事を再開する。エージェントはこれらの状態間を遷移し, その状態に応じた処理を行う。

また, マルチスレッドで行いたい各処理をエージェントとして生成し, 別々のスレッド上での初期起動を行う, メインメソッドの代わりとなるエージェント (メインのエージェント) を用意した。各エージェントの生成が終わると, メインのエージェント自身は移動を待機する状態へと移る。メインのエージェントは, マルチスレッドな動作を行うすべてのエージェントが「停止」状態になったとき, 各エージェントの実行状態を持って移動し, 移動先で中断処理の再開を行う。この手順を繰り返すことでマルチスレッド型エージェントの強マイグレーションの実現を行う。

上記で説明した各種動作を行うために必要な機能を実装し, マルチスレッド型エージェントへの対応を行った。

### 6.3 実行結果

今回, マルチスレッド型エージェントのサンプルとして, A・B・C の 3 つの処理をマルチスレッドで同時実行するエージェントを用意した。A・B・C の処理内容はそれぞれ, ループを利用して 1 から順に数を出力するものとなっている。各処理にはあるタイミングで「確認」または「停止」状態へと遷移するための命令が挿入されており, そのタイミングは各処理によって違う（例：処理 A は出力する数が 5 の倍数の時「確認」へと遷移し, 処理 B は出力する数が 3 の倍数の時「確認」. １０の倍数の時「停止」に遷移するなど)。サンプルのエージェントを実行するマシンには, 5 章で紹介したものと同じ 2 台のマシン A・マシン B を使用した。

図 6.3.1 は, マシン A 上でマルチスレッド型エージェントを起動した結果である。図中の拡大図における「MultiA」や「MultiB」といった出力は, それぞれ処理 A と処理 B の出力結果を表している。図から A・B・C の処理は同時に動いており, マルチスレッドの動作を行っていることがわかる。図中の下にある「Migrate」という記述は, 3 つの処理がすべて停止状態へと遷移したため別のマシンへと移動したことを知らせるものとなっている。

図 6.3.2 はマシン A から移動してきたエージェントがマシン B 上で処理の再開を行った結果である。図中の「Continue」という記述は, 別マシンからエージェントが移動してきて処理を再開したことを知らせるものとなっている。図中の拡大図における結果を見ると, エージェントの出力している値は 0 から始まるのではなく, 移動前の出力の続きから行われている様子が見て取れる。

また移動前と同様に, A・B・C それぞれの処理が同時に動いていることがわかる。マシン B においても各処理は出力の途中で「確認」や「停止」といった状態間を遷移し, すべての処理が停止の状態へと遷移すると図中下部のように「Migration」の記述を残して別のマシンへと移動する。

以上のように, 今回の実装によってマルチスレッド型エージェントの強マイグレーションが可能となったことがわかる。

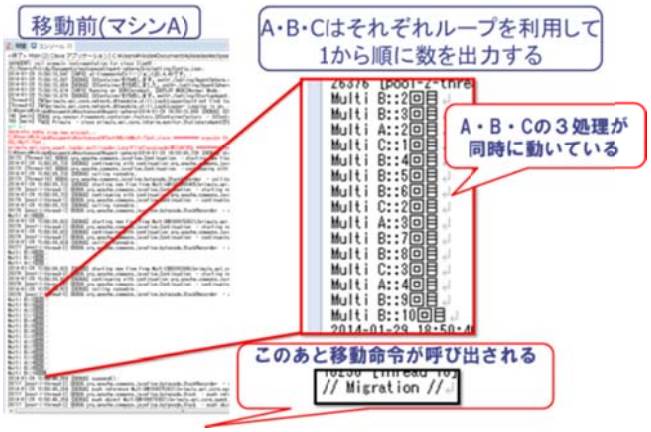


図 6.3.1 マシン A における動作の様子

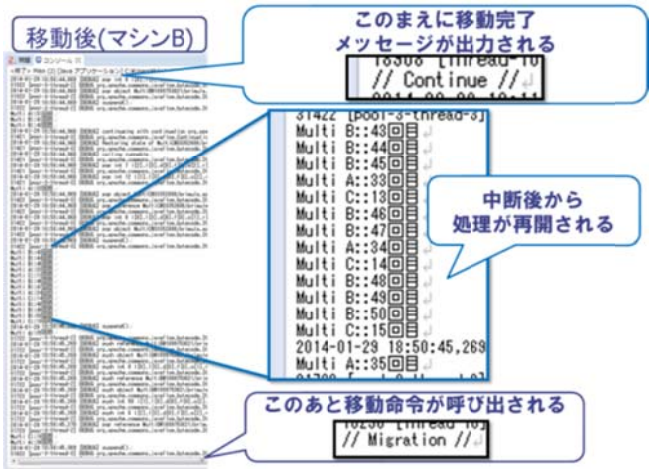


図 6.3.2 マシン B における動作の様子

## 7 終わりに

今回, JavaFlow を利用して強マイグレーションの方式を改良した。これによって, ソースコード変換方式と比較してオーバヘッドを軽減することができ, AgentSphere 上でのマルチスレッド型エージェントにおける強マイグレーションを可能にすることができた。

[参考文献]

[1] 首藤一幸：「MOBA: Mobile Agent Facilities on Java Lang. Env.」, http://www.shudo.net/moba/, Dec.(2003), Jun.2014 現在参照可.
[2] ApacheProject.JavaFlow. http://commons.apache.org/sandbox/commons-javaflow/, May.(2008), Jun.2014 現在参照可.
[3] 赤井雄樹・山口大祐・甲斐宗徳：「JavaVM 上での非手続オブジェクト転送を可能とする直列化方式の構築」 FIT2011, B-032, (2011).