B-032

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

An Implementation of Serializing Method for Non-procedural Object Transfer between JavaVMs

山口 大祐† 赤井 雄樹† 甲斐 宗徳†
Daisuke YAMAGUCHI Yuuki AKAI Munenori KAI

1. はじめに

過去数年にわたり、著者らは並列分散環境のための自律型アプリケーションフレームワーク AgentSphere[1][2]を開発してきた。AgentSphere はモバイルエージェントに活動の場を提供するもので、１台の PC に１つ動かすことを前提としている。AgentSphere が動いている PC が複数接続されたネットワーク上では、各 PC が対等な関係にあり、その台数は随時増加減少してよいものとしている。どのマシンの AgentSphere に行ってもモバイルエージェントを動作可能にするため、ヘテロジニアスなマシン環境に有効な Java を用いて開発している。モバイルエージェントを Java で実現するためには、Java 仮想マシン(以降 JVM と記す)を起動したままクラス情報の動的な追加・更新を行う必要がある。

JVM 同士でエージェントを移動するにあたり、Java の直列化および直列化復元を用いることができる。すなわち ObjectStream を用いることで透過的にインスタンスを扱うことができ、ローカル PC とリモート PC で継続してインスタンス操作ができる。このインスタンスとはクラスから生成されたオブジェクトであり、計算処理中のメンバ変数を保持している。しかしながら、インスタンス情報はクラス固有であり、そのクラスを構成する情報がローカル PC とリモート PC で異なる場合には、インスタンスを透過的に扱うことが出来ない。例えば、ローカル PC 上にのみクラスの情報が存在するとき、それを知らないリモート PC は動的に起動中の JVM に新しいクラスをプラグインする機構は存在しないので、そのインスタンスを取り扱えない。

一方、RMI などの多くの実装でクラス名解決する方法は、URI で指定されたサーバなどに動的に問い合わせることである。また CORBA など一般的に知られるフレームワークの中での動的なクラス名解決もサーバクライアント型で行われる。しかし、未知のクラスを含むエージェントを実行しようとするたびに、サーバへのクラス解決要求と必要なオブジェクトを集める必要があり、実行開始する PC への物理的なネットワーク通信経路によっては大きな遅延を引き起こす。さらに、タイミングによっては大量のリクエストが実行開始する PC へ多くのリモート PC から集中的に到着することが想定される。そのため、インスタンスを転送後に直列化復元させるためには、必要なクラス情報データを一緒に転送する方法を考える必要がある。

本研究ではクラスの動的更新と差分更新を可能とするクラスローダ、遠隔オブジェクトインスタンスの直列化復元を可能とするオブジェクトストリーム、未知クラスの受信を考慮した転送方式の設計および構築を行う。これらの機能を用いることで移動可能なクラスのオブジェクト転送と、転送されたオブジェクトの直列化復元およびメソッド呼び出しの実装と検証を行う。

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

2. 移動可能なクラスの転送とオブジェクト直列化

2.1 オブジェクトの直列化・直列化復元

分散オブジェクトでかつそのオブジェクトを移動可能にするためには、ローカルで認識されたクラスをリモートでも認識させる仕組みと、オブジェクト直列化・直列化復元の仕組みを一貫させる必要がある。しかし Java ではこの２つの仕組みは一貫されていない。

Java において Serializable インタフェースを備えるクラスは直列化可能である。ローカルとリモートで同じクラスを含むクラスパスを指定したときだけ、直列化して送られたクラスが直列化復元されて実行可能となる。このため、オブジェクト直列化・直列化復元の仕組みと動的クラス認識の仕組みの二つを連携させる必要がある。

インスタンスの送信元であれば、クラスからクラスローダを参照できることから送信用のデータを容易に特定出来る。しかし、送信先のリモートでオブジェクトを直列化復元するためには、予めクラス解決をする必要がある。そのため、未知クラスを受信して新たに認識するためにはクラスローダの生成・選択が必要である。

また JVM を永続的に稼働させたまま、システムが持つミドルウェアとしての機能や、アプリケーションレベルでのライブラリのみを差分更新したい。さらに、ある機能を更新しようとする場合、関連するクラスをすべて解放しないと更新できないのであれば、効率上看過し難いため、一部のクラスを差し替えるという機能が必要である。

2.2 動的更新を目的とした多階層化クラスローダ

クラスローダの一般的な設計では検索委譲先として親クラスローダへのリンクを持つとされている。つまり、ユーザクラスローダを設計する場合、祖先にシステムクラスローダを持つ必要がある。この仕組みは Java の標準 API と実行時カレントディレクトリの Java プログラムのクラスを認識しているクラスローダを必要とするからである。例えば、標準 API のクラスなどを扱うクラスをユーザクラスローダで読み込んだ場合を考える。Java では、すべてのクラスは標準 API の Object 型を継承する必要がある。このため、必ずユーザクラスローダの祖先にシステムクラスローダを持つように設計しなければならない。

しかし、最終的にシステムクラスローダへのリンクが存在すればよいことになっている。そのため、親子関係のリンク設計に関しては自由度が高く、クラスの動的追加・更新の頻度に応じてクラスローダの多階層化設計が可能である。つまり、親側で定義されたデータ構造クラスを用いることや親側で定義されたクラスを継承するといった使い方を想定した設計が可能である（図 2-1）。このクラスローダをオブジェクトの送受信を行うオブジェクトストリームに組み込むことになる。

Copyright © 2011 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

また、クラスをロードする際には、そのクラスが用いているクラスのロードを再帰的に行うことになる。この時、実行時の呼び出し順序と、直列化時の呼び出し順序、直列化復元時の呼び出し順序は少しずつ異なり、この順序が問題とならないように設計する必要がある。

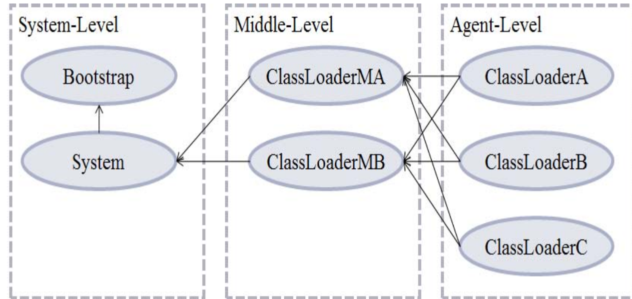

図 2-1：多階層化クラスローダ設計例

2.3 分割されたクラスローダの委譲関係

多階層化と細分化のためにクラスローダを分割するにあたり考慮すべき点がある。それはクラスローダ間の委譲関係であり、これはクラスローダの親子関係のように表現される。クラス間継承関係の親子とは異なり、「あるクラスローダがクラス解決を図る前に検索要求を一旦委譲する先」を親と呼ぶ。

クラスローダを設計するに当たり、ClassLoader クラスを継承して作成する。このとき ClassLoader の予め用意されているコンストラクタには、その委譲先としての親クラスローダを引数として渡すようになっている。つまり、クラスローダを Java の慣習に則り作成するのであれば、親クラスローダを一つ想定して作るべきである。クラスローダはこの親子関係を用いて再帰的にクラス検索とクラス解決をしており、この標準で用意されているルーチンを使って多階層化を実現する。

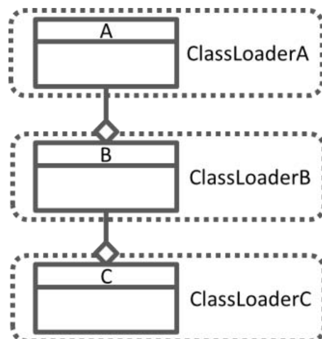

図 2-2：クラス集約例

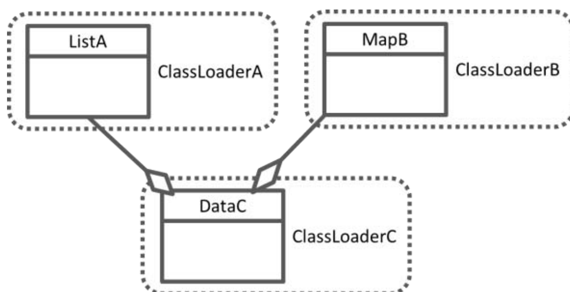

図 2-3：複数からの集約例

クラスローダの検索委譲の例を図 2-2 に挙げる。図 2-2 の例では三つのクラス関係を示す。Class A があり、Class B は Class A をメンバとして持ち、さらに Class C が Class B をメンバとして持つ場合に、Class A は Class B と同じクラスローダか、Class B より親のクラスローダに読まれている必要があり、さらに Class B は Class C と同じクラスローダか Class C より親のクラスローダに読まれている必要がある。

子クラスローダは一つの親クラスローダの参照を持ち、クラス解決のための検索時に検索を委譲する先として用いる。図 2-2 の例では Class C が出現した際に、Class C にはメンバとして Class B が含まれるため、これを解決しなければならない。その際に ClassLoaderC は、親の ClassLoaderB に検索委譲をし、Class B を解決する。同様に Class A は ClassLoaderA で解決される。この例の場合には集約に対して関連するクラスローダの委譲先が一元的であるため、問題なく可能である。また、クラス間の継承関係の場合でも単一継承しか許されないため、ClassLoader 間の委譲関係は単純で済む。

しかし、集約関係の場合には複雑なクラスローダ親子関係を必要とする場合がある。以下に例を示す。

クラス ListA とクラス MapB というクラスが存在し、それぞれが ClassLoaderA と ClassLoaderB によってロードされているとする。これらは互いに親子関係はないとする。この時クラス DataC をそれらの子にあたる ClassLoaderC で読み込むとする。クラス DataC はメンバにクラス ListA とクラス MapB を持つ。この場合、検索委譲先が一個であると、図 2-3 のように２つのクラスローダを指定できず、該当クラスを見つけることができない。

クラス DataC は意味的に複数の親を持つ必要がある。クラス間継承であれば Java は単一継承であるため、親は唯一で事足りるかもしれないが、メンバに関してはそうであるとは言えない。そのため、ClassLoaderC は ClassLoaderA と ClassLoaderB のどちらも検索できなければいけない。それを可能とするためには、クラスローダを複数内部に持つクラスローダを設計する必要がある。

図 2-4 では、クラスローダをラップするクラスローダの設計を示している。2.2 節で示した多階層化の細かな実装例である。クラスローダは検索委譲先の親を一つだけ持つことが出来るので、クラスローダをラップするクラスローダを委譲先としている。ラップするクラスローダは、ラップされるクラスローダが該当クラスを解決できるかどうかを順に調べていく。途中で解決可能なクラスローダが見つかった場合はラップされているクラスローダで解決し、見つからなかった場合は更に委譲先へと検索要求をする。

クラスローダをラップしたクラスローダを作ることは、クラス関係の構成においてあるクラスのみ中身を変更して動作の更新を行いたい場合にも有効である。なぜなら、あるクラスについてのクラスローダを直接一列につないでしまうと、クラスローダの委譲関係がコンストラクタでしか指定できない以上、効率上の問題が生じるからである。あるクラスデータを変更するためにクラスローダを付け替えるとき、それ以下につながるすべてのクラスローダについても再生成が必要になってしまうのである。ラップするクラスローダへの参照になっていれば、他のクラスローダからはその中に必要なクラスローダが含まれていればいいだ

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

けであり参照関係の変更がないため、他のクラスローダの再生成の必要がなくなる。

ラップされるクラスローダは以下の特徴を持つ。まず、ローカルで最初にクラスを取得する場合、普通にバイナリファイルから取得することができる。ここで、ロード処理は「あるクラス内で受動的に呼び出しがかかる」ため、ローカルでは問題ないが、リモートで行う場合には、ハードディスク内にクラスバイナリが存在するとは限らなくなる。そのため、コンストラクタでファイルディレクトリパス以下、または指定された圧縮ファイル内のクラスバイナリデータをあらかじめメモリに展開しておき、ロード要求時にはそのデータマップからクラス解決を行うようにした。このクラスバイナリをメモリ上で展開、保持し、それを取得できるようにする機能は、クラスローダ自身のリストアを高速に可能とする点で重要である。

ラップされたクラスローダのこの機能を用いて、クラスローダのリモートでの再生成を行う方法を次節で説明する。

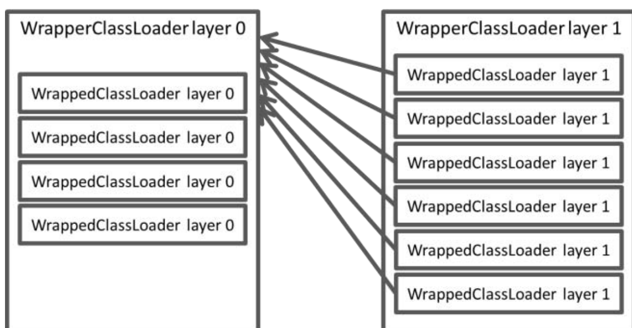

図 2-4：クラスローダをラップするクラスローダ

2.4 未知クラスの転送を考慮した転送方式

オブジェクトの直列化復元のためにはクラスのバイナリデータが必要である。しかし、クラスバイナリデータは単体であるとは限らず、独自実装した構造体のようなクラスを用いている場合がある。その場合、クラスが用いているクラスを含めて転送しなければいけない。転送すべきクラス一覧を生成する方法として、該当クラスローダ自身が解決可能なクラス一覧のバイナリデータをすべて送るという形を採用した。この方法では、ある特定のクラスにとって不要なクラスのデータさえ一緒に送ることがあり得る。それでもこの選択をした理由は次の通りである。

対案として、クラスが出現した情報をすべてロギングし、そのクラスデータを一つにまとめて送るという設計も考えられる。しかし「Java のクラス解決のタイミングは実行時に出現したタイミングである」ため、例えば、頻度の低い例外処理中でのみ使われるクラスが存在した場合、ローカルで例外が発生しない状態のまま転送されると、リモートで例外が発生した時にクラス解決が不可能になるからである。

オブジェクトの直列化に際して、ObjectStream を用いて、オブジェクトインスタンスの情報とクラスデータを一つにまとめて送ることでリモート先での直列化復元を実現する。しかし、インスタンスからクラスデータを取得し、直列化データに付加する形で転送を行うと失敗してしまう。

理由は図 2-5 の継承クラスの例に示すように、インスタンスデータ中でのクラスの出現順序（クラスローダの委譲関係）が継承関係などで必要とされる順序と異なるためである。この場合、図 2-6 に示すように直列化および直列化復元が行われ、結果としてオブジェクト直列化復元時にクラス解決の再帰呼び出しで失敗する。

この問題点を解決するために、図 2-7 に示すように、クラスバイナリデータをシリアライズし、その後ろにオブジェクトデータをシリアライズする。こうすれば ObjectStream から読み込む際に、先頭がクラスバイナリデータであるため先にクラスローダを構築させることができる。これによって、オブジェクトデータ内でクラス間がどのような依存関係を持っていても、既にクラスローダ群のツリーは完成しているため、直列化復元が可能になる。

以上のように、直列化されたオブジェクトは、それ単体で必要なクラスローダ群を正しく再生成でき、どんな継承・集約関係を持つオブジェクトでも直列化復元することができるので、ここでのオブジェクト転送の方法を非手続オブジェクト転送と呼ぶことにする。

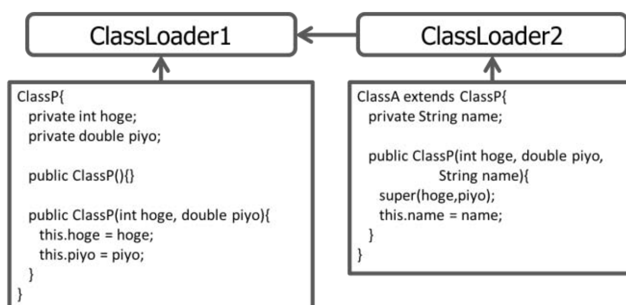

図 2-5：継承クラス例

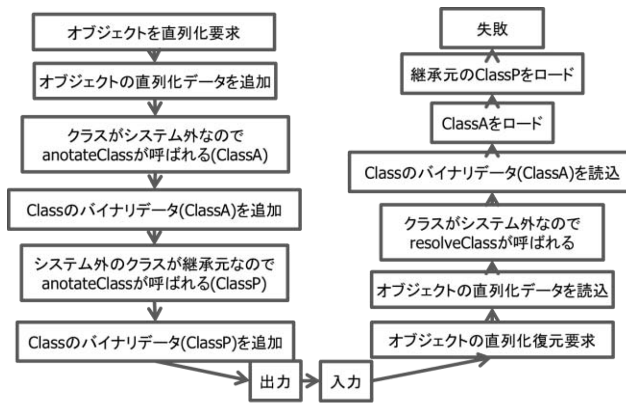

図 2-6：直列化復元に失敗する例

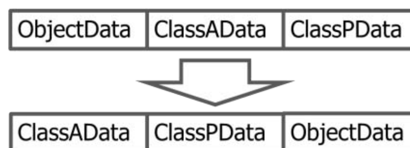

図 2-7：データ順序の変更

3. 検証

クラスデータの動的追加・更新によって JVM を停止することなく動作を変化させることが可能なことを確認したい。そのため、図 3-1 のようなウィンドウを用意した。このウィンドウの中で、Area1 は、このマシンの上で実行されたインスタンスのウィンドウが現れる領域である。

Copyright © 2011 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

Area2 と Area3 は、Area1 内のインスタンスが出力する先のテキストエリアである。ここでクラスデータの動的追加、更新を検証するため、Area1 上で動く、Swing の InnerFrame を継承した extend.TestInner という簡単なプログラムを作成した。extend.TestInner はボタンを持ち、そのボタンを押すとプログラムで指定された Area2 か Area3 へ、テキストを出力するクラスである。

まず、クラスデータの動的追加を確認する。検証プログラムの起動時点では、extend.TestInner は未知のクラスである。図 3-1 のように、JVM を停止せずに extend.TestInner のオブジェクトを生成できることでクラスローダが動的に生成されていることを確認した。

次に、extend.TestInner のクラスデータが動的更新できることを確認する。まず図 3-1 のように、extend.TestInner のボタンを押すことで、Area2 にテキストが出力される。これは extend.TestInner が最初は Area2 をテキストの出力先としていたからである。その後、ボタンを押したときに、違うテキストを Area3 へ出力するよう extend.TestInner のクラスデータをアップデートして、新たに extend.TestInner のインスタンスを生成する。図 3-2 は生成したインスタンスのボタンを押した後の状態であり、Area3 に異なったテキストが出力されている。これによりクラスデータの動的更新が行われたことがわかる。

以上により、クラスローダの動的追加・更新によるシステムの変更が可能なことが確認された。

また図 3-3 の時、古い方のウィンドウのボタンを押すとテキストが Area2 に出力されることにより、新旧のクラスデータが併存できていることがわかる。これは、Java のシステムにおいて、クラスインスタンスはそれをロードしたクラスローダへの参照を持つからである。クラスローダは過去に解決したクラスのキャッシュを持つ。つまり、クラスの動的な置き換えはクラスローダ自身の廃棄を意味するが、クラスインスタンスのクラスローダ参照を切ることは不可能である。つまり、クラスローダ自身をガベージコレクトしようとしても、そのインスタンスがあるうちは廃棄されず、同名クラスで新・旧混在する形となる。

この新旧混在状態のうち、旧クラスインスタンスを新クラスインスタンスへと更新するためには一旦直列化し、新クラスローダによってクラス解決すれば良い。

4. おわりに

本研究では、分散オブジェクトの直列化／直列化復元に関する JVM 上での現実的な実装方法を提案した。この開発は、自律型並列分散プラットフォームとなる AgentSphere のうちエージェントのマイグレーションに関わる部分の開発である。この分散処理におけるオブジェクトの非手続転送というのは AgentSphere にとって、システムの効率性や保守性を謳うために必須の部分である。ネットワーク分散処理を扱うアプリケーション以外にも、クラスローダ単体で見た場合にはオンライン取得からの動的更新自体にも実用性があると考える。さらにオブジェクト転送を可能にすることでクラス構成環境と実行途中のインスタンスオブジェクトを移動できる。そのため、並列分散処理の一つの形として汎用性と妥当性を考慮した実装ができたと考える。

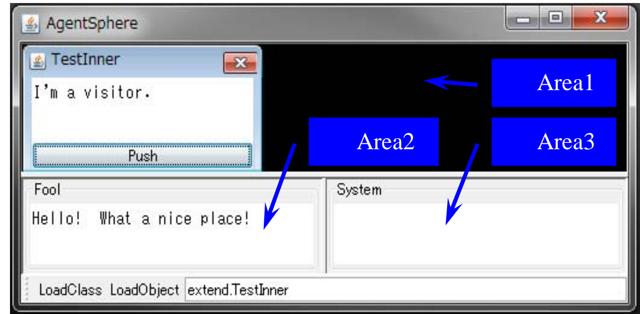

図 3-1：クラスデータ動的追加

図 3-2：クラスデータ動的更新

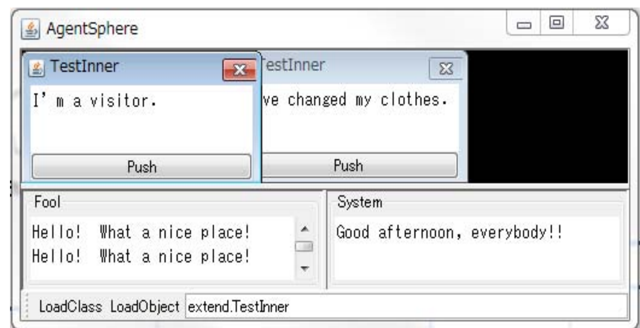

図 3-3：クラスデータの共存

謝辞

本研究は科研費（基盤研究(C)21500041）の助成を受けたものであることをここに記し、謝意を表します。

参考文献

[1]赤井雄樹・横内　貴・若尾一晃・甲斐宗徳「強マイグレーションモバイルエージェントシステム AgentSphere の開発」,FIT2009, B-011, 2009.9

[2]山口 大祐・市川 顕・白谷 浩次郎・甲斐 宗徳「強マイグレーションモバイルエージェントシステム AgentSphere におけるエージェントの活動管理」,FIT2010,B-003,2010,9

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.