ソフトウェアの動的進化を可能にする Negligent Class Loader

Negligent Class Loaders for Software Evolution

佐藤 芳樹† 千葉 滋†

Yoshiki SATO Shigeru CHIBA

† 東京工業大学大学院情報理工学研究科

Graduage School of Information Science and Engineering, Tokyo Institute of Technology

{yoshiki,chiba}@csg.is.titech.ac.jp

Java におけるクラスの動的進化を実現するために、コンポーネント間のバージョンバリアをあらかじめ指定した兄弟クラスローダの間でだけ緩められるクラスローダ (NCL : Negligent Class Loader) を提案する。バージョンバリアとは、あるバージョンのクラスのインスタンスを別バージョンのクラスの変数へ代入させない仕組みである。Java では同じ名前でも別のクラスローダでロードされたクラスは異なるクラスとして認識され、本稿では、そのようなクラスを異なるバージョンと表現する。バージョンバリアが緩められることで、古いアプリケーションは進化のために新クラスローダによりロードされた別バージョンクラスのインスタンスを受け取れる。また、バージョンの異なるクラスのインスタンスがどちらのバージョンのクラスとしても安全にアクセスされるように、NCL は一方のバージョンがもう一方のバージョンに互換するようにクラスをロードする。

1 はじめに

近代的なソフトウェアにとって、その構造や振る舞いを動的に進化させられる機能は魅力的である。商用オンラインサービスでは、アプリケーションの非稼働時間が巨大な損害を生じさせるため、オンザフライの保守・運用が望ましい。また、基幹サービスを補助する巨大ミドルウェアを用いた開発では、ミドルウェアのリスタートが開発効率を著しく落としてしまう。特に大規模システムに用いられる C++ や Java のような高速化のために静的に強く型付けされた言語には、もう一歩踏み込んだ動的進化の実現が必要だと考えられる。本稿では、開発者によって実行中のプログラムがその動作を止めずに変更されることを動的進化と呼ぶ。

しかしながら、近年、Java プログラムにおける動的進化が、クラスローダの厳しすぎる制約によって強く妨げられていると考えられるようになった。Java クラスローダは、一旦ロードしたクラスのアンロードやリロードを許さない。Java ではクラスローダ毎に名前空間が作られるので、別クラスローダを用いれば同名クラスを別バージョンで同一アプリケーションにロードすることもできる。しかしながら、それを利用した動的進化は、バージョンバリアと呼ばれる制約によって強く制限されている。

本稿では、二つのクラスローダで別々にロードされたコンポーネント間のバージョンバリアを緩められるクラスローダ (*NCL : Negligent Class Loader*) を提案する。新旧コンポーネント間のバージョンバリアを部分的に緩めることで、古いアプリケーションは新しいバージョンのクラスのインスタンスを受け取り動的に進化できる。バージョンバリアとは、あるバージョンのクラスのインスタンスを別のバージョンのクラスの変数へ代入させない仕組みである。Java では、別々のクラスローダでロードされたクラスはプログラム上で異なる型として見なさる。本稿では、別々のクラスローダによってロードされた同名クラスを、そのクラスの異なるバージョンと表現する。同一クラスの異なるバージョンのクラスのインスタンスを代入できる変数の型は、バージョンバリアによりそのクラスの同一バージョンのスーパークラスかインタフェースでなければならない。しかし、バージョンバリアを含むこのような Java 言語の型システムは、ブラウザ上の同名アプレットや tomcat 上の同名サーブレットでそれぞれが相互に干渉しないことを保証する重要な役割を担っている。

以下では、まず次章で、本研究の動機例を示し、続いて 3 章で安全でオーバーヘッドの小さいバージョンバリアの緩め方を目指した NCL の設計と実装を述べ、4 章で本稿をまとめる。

2 動的進化に不十分な Java クラスローダ

本章では、現在の Java クラスローダの不便さを理解するために、バージョンバリアの弊害を二つの実例とともに示す。

2.1 動的アスペクトウィービング

実行時にアスペクトと呼ばれるコードモジュールとアプリケーションを合成 (weave) する Dynamic AOP(Aspect-Oriented Programming) システムが注目されている。Dynamic AOP システムではアスペクトを動的に切り替えることでソフトウェアを適応的に進化させられる。

しかし、Java では標準の ClassLoader クラスのサブクラスを定義してその挙動を変更したクラスローダを使ったとしても、Dynamic AOP システムを実装するのは複雑で難しい。そのようなカスタムクラスローダを利用するとクラスロード時の挙動をカスタマイズできるため、クラスロード時に動的にアスペクトを合成できる。ところが、バージョンバリアはカスタムクラスローダによってロードされたクラスのインスタンスをアプリケーションプログラム中で直接扱うことを許さない (図 2.1)。例えば、下のプログラムでは、4 行目の左辺と右辺の Product 型は同名でも、一方がシステムクラスローダ、他方がカスタムクラスローダで定義されるので型が異なり ClassCastException が引き起こされる。

```
1: loader = new CustomClassLoader(…);
2: byte[] modifiedclass
     = weaver.compose("Product","Log");
3: Class c = loader.load(modifiedClass);
4: Product p = (Product) c.newInstance();
```

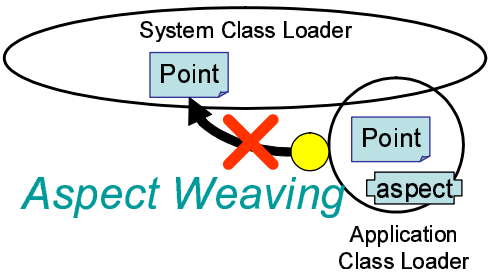

図 1: アスペクトを合成し、拡張クラスローダでロードしたクラスのインスタンスを扱えない

2.2 コンポーネントのホットデプロイ

多くの Web アプリケーションサーバでサポートされているホットデプロイ機能は、アプリケーションの開発効率を劇的に向上させている。ホットデプロイとは、アプリケーションサーバをリスタートすることなくコンポーネント (EJB-JAR、EAR、WAR など) を進化 (デプロイ、アンデプロイ) させられる機能である。それぞれのコンポーネントは単独でホットデプロイされるために別々のクラスローダでロードされる。Java のクラスローダはクラスのリロードを許さないため、コンポーネントをクラスローダごと破棄し、新しいクラスローダでロードし直すことでホットデプロイが実現される。

しかし、クラスローダを利用したホットデプロイは、デプロイ後の新コンポーネントが旧コンポーネントで使われていたインスタンスを扱えないという問題がある (図 2.2)。例えば、キャッシュやクッキー、セッションのような継続して利用されるべきデータの型は新しいコンポーネントでは別の型として認識されるため、バージョンバリアによって使用が制限される。新旧コンポーネント間で共通して使うクラスを親クラスローダに委譲して共有する方法もある。だが、それによって細粒度なホットデプロイができなくなってしまう。

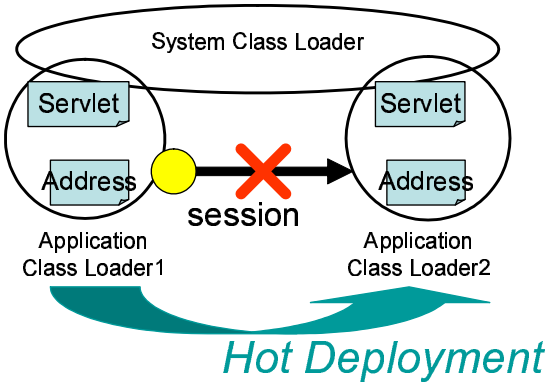

図 2: 旧コンポーネントの内部データはホットデプロイ後に破棄される

細粒度のホットデプロイを許しつつコンポーネント間でデータを受け渡すのは難しい問題であり、妥当な解決方法は示されていない。初期の J2EE では、その問題を避けるためにコンポーネント間の通信すべてが RMI だったため非常に遅かった。また、コンポーネントの更新頻度に応じて最適なパッケージン

グを推奨するアプリケーションサーバもあるがとても複雑である。また、JBoss の UCL(Unified Class Loader) ではすべてのコンポーネント間で同じクラスを共有できるが、名前空間が破壊され複数のバージョンのクラスが共存できない。

3 Negligent Class Loader (NCL)

我々は、ソフトウェアを動的に進化させるために Negligent Class Loader (NCL) を提案する。NCL とその兄弟クラスローダの組を明示的に指定することによって、それらがロードするクラスの間でバージョンバリアが緩められ、一方のバージョンのクラスのインスタンスは他方のバージョンのクラスの変数へ代入できるようになる。兄弟クラスローダとは、クラスを共有する委譲関係での親クラスローダが等しいクラスローダを指し、NCL との間でバージョンバリアが緩められる兄弟クラスローダを便宜上 FCL (Friend Class Loader) と呼ぶことにする。例えば、NCL によってロードされたクラス Product のインスタンスは、FCL の Product 型の変数にのみ代入できる。また、二つの Product クラスは型階層が異なっていたり、アクセス制限を弱めたりしない場合にのみバージョンバリアが緩められる。

また、NCL を利用した動的進化では、緩められたバージョンバリアを越えて別バージョンのクラスの変数へ代入されるインスタンスのクラスのバージョンは変わらない。つまり、新旧コンポーネントで生成されたインスタンスは、他方へ渡されても生成時のバージョンのクラスで定義されたメソッドが呼び出される。したがって、インスタンスのアップデートに伴うオーバーヘッド [3] は生じない。

しかし実は、安易にバージョンバリアを緩めると、セキュリティ上の重大な問題が発生してしまう。例えば、別バージョンの型のインスタンスによる型詐称 (type spoofing) により、プログラムに存在しないメソッドを呼び出させて JVM(Java Virtual Machine) をクラッシュさせられる。実際、Sun JDK1.1 にはクラスローダを利用してバージョンを詐称できる致命的なバグ [4] が存在したため、ローダ制約 [2] というバージョンバリアを強める機構が導入された。バージョンバリアを安全に緩めるための簡単な手法として、すべてのインスタンスアクセスでクラススキーマ (フィールドとメソッド) をチェックする方法が考えられる。このようなスキーマチェックは、CLOS や Smalltalk などの動的型付け言語の素朴な実装にも用いられる。そのように実装された処理系では、もし存在しないメンバがアクセスされたとしても正しく例外を投げてくれるが、ランタイムチェックによる性能劣化が無視できない。

3.1 スキーマ互換とバージョンチェック

過度のスキーマチェック無しでも安全にバージョンバリアが緩められるように、NCL は二つのバージョンのクラススキーマを互換させる (スキーマ互換)。スキーマ互換されたクラスでは、メソッド呼び出し (`invokevirtual`) やフィールドアクセス (`getfield`)、代入 (`astore`) などの直前で逐一スキーマチェックを行う必要が無い。したがって、スキーマチェックの代わりにバージョンチェックを行うだけで良いことになる。このバージョンチェックは、インスタンスの受け渡しをスキーマ互換されたクラスの間、すなわちバージョンバリアが緩められたクラスの間に制限するのに行われる最小限の検査である。

スキーマ互換 スキーマ互換のために、NCL はクラスロード時にクラスのあらゆる情報を保持する TIB (Type Information Block) と呼ばれるクラスの JVM 内表現を旧バージョンのものに互換させる。具体的には、TIB には仮想メソッドテーブルの情報などが保持されているが、NCL は一方のインスタンスが他方の変数へ代入された後でも安全にアクセスされるように新旧 TIB の仮想メソッドテーブルを書き換える。

スキーマ互換により、仮想メソッドテーブルが持つ関数ポインタのセットは、同じインデックスでシグニチャ(名前、引数の型と順序、戻り値の型) の等しいメソッドが引けるようになる。例えば、NCL が新バージョンのクラス N を旧バージョンのクラス F に互換させる場合、NCL は N と F の TIB がメソッドシグニチャの等しいエントリを同じインデックスに持つように TIB を生成する (図 3)。

バージョンチェック バージョンチェックは明示的なキャスト命令と同じ場所でだけ行う。具体的には、`checkcast` バイトコード命令で、キャスト元とキャスト先のクラスの間でバージョンバリアが緩められているかどうかをチェックするだけで良い。なぜなら、緩められるバージョンバリアが NCL と FCL との間に制限されるので、それら二つの兄弟クラスローダ間には bridge-safety[4] という性質が満たされる

```
1:  if  F はロード済み？  &&
2:   F の定義ローダは FCL？ then
3:     foreach m (F のすべてのメンバ) {
4:        if N はメンバ m を持つ？ then
5:           N の TIB に m を挿入
6:        else
7:           N の TIB に SHF を挿入
8:        end
9:     }
10:    N にのみあるメンバを TIB に挿入
11:    if F は N のサブセット？ then
12:       F 用に新しく TIB' を生成
13:       foreach m (N のすべてのメンバ) {
14:          if F はメンバ m を持つ？ then
15:             TIB' に m を挿入
16:          else
17:             TIB' に SHF を挿入
18:          end
19:       }
20:       F の TIB を TIB' と交換
21:    end
22: end
```

図 3: 疑似コードによる NCL の TIB 構築アルゴリズム

3-10 行目 等しいメソッドシグニチャを持つメソッドが N と F で同じインデックスになるように N の TIB を生成する。N に存在しないメソッドが F にある場合、そのエントリへ secure handling function (SHF) を挿入する。SHF は、Runtime-Exception を投げるメソッドであり、不正なメンバアクセスを正常に終了させる。

11-21 行目 F に存在しないメソッドが N にある場合、N のエントリ数と同じ大きさの TIB' を F 用に生成する。そして F に存在しないメソッドのエントリへ SHF を挿入する。

20 行目 F のクラスオブジェクトからの TIB ポインタを TIB' へ差し換える。

からである。bridge-safety が満たされると、一方にロードされたクラスのインスタンスが他方にロードされたクラスに渡されるときに、必ず親クラスローダにロードされたスーパークラスやインタフェースにアップキャストされる。したがって、兄弟ローダ間の bridge を横切る時に必ず明示的なキャスト、すなわち `checkcast` 命令でバージョン互換をチェックすれば良いのである。(図 4)。

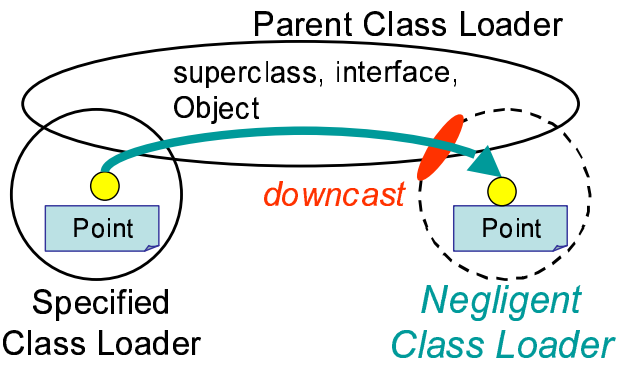

図 4: NCL と FCL の間でのインスタンス渡しは必ずダウンキャストを伴う。

4 関連研究

ソフトウェア進化を目指した技術は数多く提案されている。Liang[2] らは、複数バージョンのクラスのインスタンスをインタフェース型で扱うことを推奨している。しかしながら、アップデートされる可能性のあるクラスをすべてインタフェースでプログラミングすることは煩雑であり、さらにクラススキーマの変更も許されない。Hjalmtysson[1] らは、C++ のテンプレート機能を利用した Dynamic Class という

技術をライブラリとして提供した。しかし、その実現はラッパークラス (プロキシ) で実現されているためオーバーヘッドが大きい。Malabarba[3] らは JVM を拡張してクラスのリロード機能を実装した。しかしながら、インライン展開やバイトコードの quick 命令などの多くの実行時最適化を犠牲にしている。また、関連するインスタンスをアップデートするために、クラスのリロード後にライブオブジェクトを走査しなければならない。また、J2SE 5.0 SDK でも `java.lang.instrument` パッケージにホットスワップと呼ばれるクラスのリロード機能が追加される。しかし、このホットスワップ機能はクラススキーマの変更をサポートしていない。

5 現状と課題

現在、我々は Negligent Class Loader を IBM の Jikes Research Virtual Machine 上に実装している。存在しないフィールドや配列要素へのアクセスを安全に取り扱うために、スキーマ互換アルゴリズムを改良していく必要がある。たとえば、フィールドアクセス前にスキーマチェックを行ったり、アクセサを使うように変換してからメソッドと同様のスキーマ互換を適応したりといった方法が考えられるが、性能や自由度においてトレードオフがあるだろう。

参考文献

[1] Gísli Hjálmtyśson and Robert Gray. Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In *In Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, 1998. USENIX.

[2] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA'98, Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, number 10 in SIGPLAN Notices, vol.33, pages 36–44, Vancouver, British Columbia, Canada, oct 1998. ACM.

[3] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *ECOOP 2000 - Object-Oriented Programming, 14th European Conference*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer, jun 2000.

[4] Vijay Saraswat. *Java is not type-safe*, 1997.