情報処理学会論文誌　プログラミング　Vol. 4　No. 3　16–25 (June 2011)

部分的再ロードによるJavaプログラムの再起動の高速化

別役浩平[†1]　千葉滋[†1]

一部のクラスの定義を変えてプログラムを再起動するには，新たに作成したクラスローダを用いてアプリケーションを再ロードする必要がある．この手法は，たとえばHOT デプロイで利用されている．しかし，アプリケーションの全クラスの再ロードによる時間的コストが大きく，サーバのパフォーマンスを低下させてしまう．そこで本稿では，アプリケーションを部分的に再ロードすることで再起動を高速化する手法を提案する．再起動時にどのクラスが変更されたかが分かれば，変更対象クラスとそれに依存しているクラスのみを子ローダで再ロードすることで，その他のクラスは親ローダでロード済みのものを再利用できる．また，アプリケーションの各クラスに複数の版があり，それらを組み合わせて何度も再起動をする状況下では，各クラスの各版ごとにクラスローダを作成し，それらがロード済みの版のクラスを再利用することで再ロードのコストを軽減できる．しかし，後者の手法では多くのクラスを変更する際，削減できる再ロードのコストに対してクラスローダインスタンスの作成によるコストが大きくなりすぎる場合がある．そこで上記の 2 つの手法を per-session AOP を用いたフレームワークに実装し，実験を行った．その結果から，どちらの手法を使うべきかの指針を示した．

A Technique for Quick Restarting of Java Applications by Partial Class Reloading

Kohei Betchaku[†1] and Shigeru Chiba[†1]

To modify the definitions of some classes of an application and restart it, the application should be reloaded by a new class loader. This approach is, for example, used for HOT deployment. However, it degrades the performance of a server application if all the classes of an application are reloaded. This paper proposes a technique to speed up restarting an application by partial class reloading. When an application is restarted, since we know which classes are modified at this time, we can reuse classes that have been already loaded and are not changed at this restart. Furthermore, when each class of an application has some versions and the application is restarted repeatedly with combination of different versions of classes, we can reduce reloading costs by creating a class loader responsible for loading each version of a class. This allows us to reuse not only original classes but also various versions of each class. However, the latter approach may degrade the performance because a large number of class loader instances are created when we modify a number of classes. We applied above two approaches to our per-session AOP framework respectively and provided guidelines which approach you should use.

1. はじめに

本稿において Java プログラムの再起動とは，アプリケーションの一部のクラスを修正して，新たに作成したクラスローダで再ロードし，再実行するという一連の動作を表している．この再起動の手法は，アプリケーションサーバや DI（*Dependency Injection*）コンテナ[1)] などで利用されている．一般に，その時点でアプリケーションをロードしているクラスローダを破棄し，新たに作成したクラスローダで全クラスを再ロードし直すことで実現される．ユーザからリクエストのたびに再起動することで，対話的なアプリケーション開発を実現できるが，全クラスの再ロードによる時間的コストが非常に大きく，サーバのパフォーマンスを低下させる．

そこで本稿では，修正したアプリケーションを部分的に再ロードすることで，再起動を高速化する手法を提案する．再ロードすべきなのは変更されたクラスのみであり，全クラスを再ロードする必要はない．再起動時にどのクラスが変更されたかが分かれば，再ロードすべきクラスと再利用可能なクラスを分割できる．しかし，変更されたクラスのみを新たに作成した子ローダで再ロードし，その他のクラスは親ローダでロード済みのものを再利用するというシンプルな実装では，2.1 節で述べるクラスローダの仕様からうまく動作しない．変更したクラスに依存しているクラス群も新たに作成した子ローダで再ロードすれば，期待する動作が得られ，なおかつ親ローダでロード済みのクラスを再利用できる．また，アプリケーションの各クラスに有限個の版があり，それらの組合せを変えて再起動する状況では，各クラスの各版ごとにそれをロードするクラスローダを作成し，後の再起動時にそれらを部分的に親ローダとして活用することで，さらに多くのクラスを再利用できる．しかし，この手法では 1 度に変更するクラス数が増えれば増えるほど，それに応じて作成するクラス

†1 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology

ローダインスタンスの数も増加するため，その時間的コストにより再起動がかえって遅くなる場合がある．そこで上記の 2 つの手法を我々の per-session AOP フレームワーク[2] にそれぞれ実装し，フレームワーク上で動作する Web アプリケーションにリクエストを送信する実験を行った．その結果から，どちらの手法を使うべきかの指針を示す．

以下，2 章では，クラスローダのアーキテクチャと修正したアプリケーションを再ロードする方法について述べる．3 章では，新しく提案する 2 つの再起動の手法について述べる．4 章では，提案手法の応用例について述べる．5 章では，我々が行った実験について述べる．6 章で関連研究に触れ，7 章で本稿をまとめて，今後の課題を述べる．

2. 新規クラスローダを用いたアプリケーションの再ロード

本章ではまず，Java のクラスローダのアーキテクチャについて述べ，次に新たに作成したクラスローダを用いて，修正したアプリケーションを再ロードする方法を検討する．

2.1 クラスローダのアーキテクチャ

Java 言語ではクラスのロードはプログラムの実行中に動的に行われる[3]．クラスのロードにはクラスローダが用いられ，java.lang.Class のインスタンスを生成する．すべてのクラスローダは ClassLoader クラスのサブクラスである（**図 1**）．

クラスローダは親子関係を持ち，木構造を構成する．JVM（*Java Virtual Machine*）からロード要求があった際，loadClass メソッドを用いて次のような手順でクラスの探索を行う．

(1) findLoadedClass メソッドを呼び出して，このクラスローダでクラスがすでにロードされたかどうかを確認する．
(2) 未ロードの場合，親ローダの loadClass メソッドを呼び出す．
(3) 親ローダがロードできなかった場合，findClass メソッドを呼び出して，このクラスローダ独自の方法でクラスを探し，defineClass メソッドを呼び出してクラスを定義する．

JVM において，クラスの型は，クラスの名前とそのクラスを定義したクラスローダの対で一意に決定される．つまり，定義したクラスローダが異なれば同一名のクラスを JVM 上にロードできる．

次に，一般的な java アプリケーションがどのようにロードされるかを**図 2** を用いて示す．まず，main メソッドが定義されているクラスをアプリケーションクラスローダがロード（defineClass メソッドにより定義）する．このクラスローダを L_A とすると，JVM 上にはクラスの型 <A, L_A> が定義される．次に 6 行目の main メソッドが呼び出され，2 行目の funcA メソッドが呼ばれる．funcA メソッド内では String クラスとクラス B を参照している．クラス A 内における他のクラスの参照は，そのクラスを定義したクラスローダ（ここでは L_A）によって解決される．つまり，funcA メソッドの実行中に

L_A.loadClass(“String”);
L_A.loadClass(“B”);

が呼び出される．アプリケーションクラスローダは String クラスなどのシステムクラスをロードするシステムクラスローダを親に持つ．これを L_S とする．String クラスの参照は，先に述べたロードの手順から L_A の loadClass メソッドから呼ばれた L_S の loadClass で解決され，JVM 上には型 <String, L_S> が定義される．一方，システムクラスでないクラス B はシステムクラスローダがロードできないため，L_A 自身が実行する．つまり，L_A の findClass が呼び出され，JVM 上には型 <B, L_A> が定義される．

なお，1 度解決されたクラス内の参照に対して，再びクラスローダが呼び出されることはない．たとえば，アプリケーション内で再びクラス A の funcA メソッドが呼ばれたとしても，String クラスやクラス B の参照は，すでに各々のクラスの定義にリンクされており，

```
1  class ClassLoader {
2    public Class loadClass(String name);
3    protected Class findClass(String name);
4    protected final Class defineClass(String name, byte[] buf, int off, int
         len);
5    protected final Class findLoadedClass(String name);
6    ...
7  }
```

図 1　ClassLoader クラス
Fig. 1　ClassLoader class.

```
1  class A {
2    void funcA() {
3      String s = ”foo”;
4      new B().funcB(s);
5    }
6    public static void main(String[] args) {
7      new A().funcA();
8  }}
```

図 2　アプリケーションの実行例
Fig. 2　An example of application program.

JVM がリンク先を参照することで解決される．

2.2　変更したクラスの再ロード

2.1 節で述べたクラスローダのアーキテクチャを考慮すると，変更されたクラスのみを新たに作成した子ローダで再ロードし，その他のクラスは親ローダでロード済みのものを再利用するという再起動の手法が実現できるように思われる．しかし，このような素朴な実装ではうまく動作しない場合がある．

まず最初に**図 3** のような例を考える．クラス A，B，C のすべてが親ローダ L_P でロード済みであり，変更したクラス B を新たに作成した子ローダ L_C にロードさせるとする．2 行目の funcA メソッドが呼ばれると JVM は L_P の loadClass メソッドを呼び出し，A クラス内の参照を解決しようとする．しかし，L_P によって解決可能なクラスは，L_P の親ローダおよび自身で解決可能なクラスなので，クラス A は L_C でロードさせたクラス B を参照できず，L_P でロード済みのクラス B を参照してしまう．これではクラス B への変更をアプリケーションに反映させることができない．

新たに作成したクラスローダを用いて変更したクラスを含めた全クラスを再ロードすれば，クラスへの変更をアプリケーションに反映できるが，その時間的コストがプログラムのパフォーマンスに大きく影響する場合がある．この手法の利用例として HOT デプロイがある．HOT デプロイとはサーバを再起動することなく，コンポーネント（EAR，WAR ファイルなど）をロード（デプロイ），アンロード（アンデプロイ）する技術であり，JBoss Application Server[4]，Tomcat[5] などアプリケーションサーバや Seasar2[6] などの DI コンテナで利用されている．DI コンテナとは，DI というデザインに基づいて作られたコンポーネント群を管理するソフトウェアである．DI とはコンポーネント間の依存関係をソースコードから取り除くことで，プログラムの実行時までコンポーネントどうしが依存関係を持たないようにするデザインパターンである．依存関係は，DI コンテナが設定ファイルを読み込み，指定されたクラスのインスタンスを自動的に作成し，依存元のオブジェクトのフィールドに代入することで実行時に実現する．

```
1 class A {
2   void funcA() { new B().funcB(); }
3 }
4 class B {
5   void funcB() { new C().funcC(); }
6 }
7 class C {..}
```

図 3　呼び出し関係のある 3 クラス
Fig. 3　Three classes referring to each other.

HOT デプロイの一種にユーザからのリクエストのたびにアプリケーションの全クラスを再ロードするものがある．これは対話的な開発を行いたいときに利用されるが，リクエスト数が多いサーバにとっては，大きなオーバヘッドになりかねない．

3. 部分的再ロードによる再起動の高速化手法の提案

2.2 節で述べたように，変更されたクラスのみを子ローダでロードさせるという素朴な手法では，アプリケーションに変更を反映できない．また，変更したクラスを含めた全クラスを再ロードするという手法はパフォーマンスの面で問題がある．そこで本章では，できるだけロード済みのクラスを再利用しつつ修正したアプリケーションを再起動するための 2 つの手法について提案する．

3.1 節では，再起動時にクラスローダを 1 つしか作成しないが，比較的再ロードするクラス数が多い手法（手法 1），3.2 節では，クラスローダを多数作成するが，再ロードするクラス数が少ない手法（手法 2）について述べる．

3.1　手　法　1

できるだけ親ローダでロード済みのクラスを再利用しつつ，修正したアプリケーションを再ロードする方法を考える．2.2 節の例で問題なのは，L_P がロードしたクラス A から L_C がロードしたクラス B を参照できないことである．

この問題は，**図 4** のような loadClass メソッドを持つクラスローダ L_C にアプリケーションを再ロードさせることで解決できる[7),⋆1]．このクラスローダは変更対象クラス（クラス B）とそれに依存しているクラス（クラス A）ならばロードし，その他のクラスは親ローダ（L_P）でロード済みのものを再利用する．ここで，あるクラス X に依存しているクラスとは，クラス間の参照をたどることで X に到達可能なクラスのことである．これを用いることで，クラス A，B は L_P でロード済みのものを再利用することなく，L_C によってロードされる．このため，クラス A 内参照は L_C の loadClass メソッドにより解決され，変更された B を参照できる．一方，クラス B が参照しているクラス C は L_P がロード済みのものを再利用するため，再ロードするクラス数を削減することができる．

⋆1 手法 1 は「並列/分散/協調処理に関するサマー・ワークショップ（SWoPP 2009）」の論文の中でも紹介した．論文は http://www.csg.is.titech.ac.jp/paper/betchaku-swopp09.pdf から入手できる．

```
1 public Class loadClass(String name) {
2   if (/* name が変更対象クラス，またはそれに依存しているクラス*/) {
3     Class clazz = findLoadedClass(name);
4     if (clazz == null) return findClass(name);
5     else return clazz;
6   }
7   else return getParent().loadClass(name);
8 }
```

図 4　子ローダの loadClass メソッド
Fig. 4　The loadClass method of a child loader object.

上で述べた手法を実現するためには，ロード時に変更対象クラスに依存しているクラス群を知る必要がある．クラス間の依存関係は Javassist [8] などのバイトコードレベル API を用いれば探索可能である．

3.2 手　法　2

アプリケーションの各クラスに有限個の版があり，それらの組合せを変えて再起動する状況では，さらに再ロードするクラス数を削減できる．たとえば，アプリケーション内のクラス A，B をある版 A'，B' に変更して再起動し，次に A' のみ元の版 A に戻し，クラス C をある版 C' に変更して再起動するといった状況である．手法 1 では，初回の再起動時に新たに作成したクラスローダ L_1 が A'，B' をロードする．実際には A'，B' に依存しているクラス群もロードするが，この例では説明を簡単にするために省略する．2 回目の再起動時には，新たに作成したクラスローダ L_2 が B'，C' をロードする．2 回目の再起動時に L_1 を L_2 の親ローダとして利用すれば，L_2 で B' をロードする必要はないが，L_1 は A' をロード済みなので，期待する動作は得られない．そこで，初回の再起動時に A' をロードするクラスローダ L_A，B' をロードするクラスローダ L_B を個別に作成し，L_B を L_A の親ローダとする．2 回目の再起動時には C' をロードするクラスローダ L_C を作成し，L_B をその親ローダとして活用すれば A' の影響を受けずに B' を再利用できる．

このように，アプリケーションの各クラスの各版ごとにそれをロードするクラスローダを作成すれば，旧版のアプリケーションを部分的に再利用できる．

3.2.1 再起動時に使用するクラスローダの決定法

図 5 に決定アルゴリズムの擬似コードを示す．

Input，Output　S は再起動時に使用するクラスの版の集合（原版を除く）．P は S の要素の優先度の集合である．優先度は，そのクラスに依存しているクラス群のファイルサイズの合計など，より多くのバイトコードをロードする必要があるものが高くなるように，あらかじめ設定，計算しておく．

Input: クラスの版の集合 S とクラスの版それぞれの優先度の集合 P
Output: 再起動時に使用するクラスローダ $L_{restart}$

1: クラスローダの木 $T = (V, E)$ (初期値 $V = \{L_S\}$, $E = \phi$) について
2: $A \leftarrow \{L \in V \mid \text{available}(L) \subseteq S, \text{available}(L) \neq \phi\}$
3: **if** $A = \phi$? **then**
4: 　$L_P \leftarrow L_S$
5: **else**
6: 　$L_P \leftarrow A$ の中で $|\text{available}(L)|$ が最大となるもの
7: **end**
8: $S' \leftarrow S - \text{available}(L_P)$
9: P を使って S' の要素を優先度が高い順にソート
10: $L_{restart} \leftarrow L_P$
11: **foreach** C S' {
12: 　$L_C \leftarrow C$ をロードする新規クラスローダ
13: 　$V \leftarrow V + L_C$
14: 　$E \leftarrow E + (L_{restart}, L_C)$
15: 　$L_{restart} \leftarrow L_C$
16: }

図 5　再起動時に使用するクラスローダの決定アルゴリズム
Fig. 5　The algorithm for making a class loader used when restarting.

1–10 行目　システム内でクラスローダの木を保持しておく．V はノード（クラスローダ）の集合で，初期値はシステムクラスローダ 1 個である．E はエッジ（クラスローダの親子関係）の集合である．たとえば，(L_P, L_C) は L_P は L_C の親であることを表す．available(L) は L を親として活用することで，L の親も含めて再利用可能なクラスの版の集合を返す手続きである．3–7 行目で決定した L_P を親クラスローダとして活用することでロード済みのクラスの版を最大限再利用できる．

11–16 行目　S の要素のうち，再利用不可能なクラスの版について，それらのロードを担

```
1:  S ← S_C
2:  foreach L_P (L_C のすべての親ローダ) {
3:    if L_P がロードを担当するクラスの版 D が S 内のクラスに参照を持つ？
4:    then
5:      S ← S + S_D
6:    end
7:  }
```

図 6　クラス群の作成アルゴリズム
Fig. 6　The algorithm for selecting classes loaded by a given class loader.

当するクラスローダをそれぞれ作成し，優先度が最も高い版をロードするクラスローダを L_P の子，その次に高い版をロードするものをその子，…と親子関係を設定する．こうすることで，より多くのクラスをロード済みのクラスローダが木の根の近くに配置され，後に親として活用しやすくなる．つまり，再起動時により多くのロード済みクラスを再利用できる．

このアルゴリズムによって決定された $L_{restart}$ にアプリケーションを再ロードさせる．

3.2.2　各クラスローダがロードするクラス群

図 5 で示したクラスローダの木において，各ノード（クラスローダ）がロードするアプリケーションはそれぞれ異なる．それらを親ローダとして活用し，新たにクラスローダを作成する際には，親ローダがロードするアプリケーションのクラス間の依存関係を考慮して，ロードすべきクラス群を決定する必要がある．

あるクラスの版 C のロードを担当するクラスローダ L_C がロードするクラス群 S は**図 6** のように作成される．ここで S_C とは C と C の原版に依存しているクラスの集合である．

1 行目　原版のアプリケーションのクラス間の依存関係をもとに初期値を与える．

2–7 行目　L_C の親ローダ L_P がロードするクラスの版 D が S 内のクラスに参照を持つならば，S_D を S に追加する．

このように部分的にアプリケーションの依存関係を更新することで，小さなオーバヘッドでクラス群を決定できる．

4. 応　　用

3 章で提案した 2 つの手法の応用例として，per-session AOP フレームワークの拡張を行った．まず最初に，per-session AOP フレームワークについて述べ，次に提案手法の実装について述べる．

4.1　Per-session AOP フレームワーク

AOP（*Aspect-Oriented Programming*）とは，OOP（*Object-Oriented Programming*）では解決できないモジュール間にまたがる横断的関心事（*crosscutting concern*）をアスペクトと呼ばれる独立したモジュールに分離するプログラミング技法である．AOP 言語の処理系は，プログラムのバイトコードを書き換えることでアスペクトの内容を反映させる．この操作をウィーブ（*weave*）という．見方を変えると，アスペクトはソフトウェアの振舞いを変更するモジュールであると見なすことができる．

Per-session AOP フレームワークでは，リクエストに含まれるセッションからユーザ ID を判別し，ユーザの設定ファイルから，そのユーザが選択したアスペクトの情報を得る．そして，それらのアスペクトをウィーブすることで，ユーザごとに異なる振舞いをする Web アプリケーションを実現可能にしている．Web アプリケーションは複数のユーザによって共有されるため，あるユーザのアスペクトによる変更が，他のユーザに影響してはならない．そこで，当フレームワークでは，ユーザごとに異なるクラスローダを作成し，それぞれ異なるアスペクトをウィーブした Web アプリケーションをロード，再起動することで，これを回避している．しかし，再起動時にアプリケーションの全クラスを再ロードするため，オーバヘッドが非常に大きく，実用性に乏しいという問題があった．

この問題は 3 章で提案した手法で改善できる．図 4 でアスペクトの変更対象クラスとそれに依存しているクラス群をロードすれば手法 1 を適用できる．また，クラスの版ごとでなく，アスペクトごとにクラスローダを作成すれば，手法 2 を適用できる．次節でその方法を述べる．

4.2　実　　装

提案手法を実現するためには，変更されたクラスの名前を，再起動時に知る機構を実装する必要がある．Per-session AOP フレームワークでは，GluonJ [9] を使用している．GluonJ とは我々が開発した Java 言語用の AOP システムである．**図 7** に GluonJ のアスペクトを示す．このアスペクトは，Web アプリケーション内のクラス Webapp の doGet メソッドを 6 行目の *code* の内容で書き換える．これは Java の通常のクラスであるため，バイトコードを調べ，Customizer クラスの親クラスを調べることで変更対象クラスを知ることができる．

アプリケーションの各クラスに依存しているクラス群は，サーバの起動時にバイトコードを静的に調べ，クラス内で参照しているクラスを再帰的に調べることで決定できる．しか

```
1 @Glue
2 public class CustomizerAspect {
3   @Refine
4   static class Customizer extends Webapp {
5     public void doGet(HttpServletRequest req, HttpServletResponse res) {
6       /* code */
7     }
8   }
9 }
```

図 7　GluonJ のアスペクト
Fig. 7　An aspect in GluonJ.

し，クラス内でリフレクション API を用いてクラスの定義を取得している場合，プログラムが実行されるまでそのクラスが依存しているクラスを決定できない．そこで，クラス内で java.lang.Class クラスを参照していれば，そのクラスもその他のクラスすべてに依存しているものとした．これにより各クラスローダで再ロードするクラス数が増加してしまうが，アプリケーションの振舞いに影響はない．

手法 1 を適用した場合，リクエストの処理の流れは次のようになる．

(1)　サーバに登録されたユーザ用のアスペクトを取得する．
(2)　アスペクトを解析し，ウィーブ（変更）対象クラスの名前を取得する．
(3)　図 4 のクラスローダを作成し，(2) クラス名とそれに依存しているクラスをこのクラスローダがロードするクラス群とする．
(4)　(3) で作成したクラスローダで Web アプリケーションを再ロードし，再実行する．

なお，アスペクトのウィーブは，クラスローダに対するクラスのロード要求時に，findClass メソッド内で動的に行われる．

手法 2 を実現するためには，各クラスに依存しているクラス群に加えて，アスペクトの優先度をあらかじめ設定，計算しておく必要がある．優先度は以下のように設定する．

優先度 = (アスペクト + アスペクトのウィーブ対象クラスとそれに依存しているクラス)
　　のファイルサイズの合計 × アスペクトの利用者数

このように設定することで，ロードするバイトコード量が多いクラスローダほど，また頻繁に利用されるクラスローダほど木の上位に配置されるため，よりロードコストが大きいクラスローダを親ローダとして活用できる．

リクエストの処理は図 5 のアルゴリズムに沿って行われる．ただし，Input の集合 S はアスペクトの集合，2 行目の available(L) は L がその親も含めてウィーブ可能なアスペクトの集合を返す手続き，12 行目の L_C は C のウィーブにより変更されたクラスとそれに依存しているクラス群をロードするクラスローダとなる．

また，図 6 のアルゴリズムにおいて，S_C はアスペクト C のウィーブにより変更されたクラスとその原版に依存しているクラスの集合を表す．それに応じて 3 行目は「**if** L_P が担当するアスペクト D のウィーブにより変更されたクラスが S 内のクラスに参照を持つ?」となる．

5. 実　　験

各々の実装方法から，リクエストのたびに全クラスを再ロードする場合（手法 0 とする），手法 1 を用いる場合，手法 2 を用いる場合の順に再ロードするクラス数が少なくなるのは明らかである．しかし，手法 2 では，1 度に多くのクラスを変更する際，最悪でそのクラス数分のクラスローダを作成しなければならない．そのため，かえって再起動が遅くなる場合がある．そこで，各手法を用いた場合の再起動時間を測定し，それらの比較・分析を行った．

Per-session AOP フレームワークに 3 つの手法それぞれを実装し，そのうえで Health Watcher [10] アプリケーションを動作させた．Health Watcher とは，健康管理のための中規模 Web アプリケーションであり，コード行数はおよそ 9 KLOC である．なお，実験環境はサーバ側は CPU：Xeon 2.83 GHz，メモリ：4 GB，OS：Linux 2.6.26 で，クライアント側は CPU：Core2Duo 3.00 GHz，メモリ：4 GB，OS：Windows Server 2003 である．Web サーバは Tomcat 6.0，クライアント側で使用するサーバ負荷テスト用ソフトには Apache JMeter [11] 2.3.2 を用いた．

アプリケーションの各クラスごとに，そのクラスにフィールドを追加するだけのシンプルなアスペクトを 1 個作成し，あらかじめサーバに登録した．つまり，アプリケーションのクラス数と等しい数のアスペクトがサーバ内にあり，1 個のアスペクトは 1 個のクラスを変更する．それぞれのフレームワークの実装別に分け，さらにユーザのアスペクトの選択パターン別に分けて平均応答時間を測定した．アスペクトの選択パターン (n, m) とは，Servlet クラス用のアスペクトを n 個ランダムに選び，そのコールグラフ上のクラス用のアスペクトを m 個ランダムに選ぶというものである．Servlet クラスとは，Web アプリケーションにおいて，リクエスト処理の開始ノードとなるクラスである．たとえば，$(n, m) = (2, 5)$ の実験では，全ユーザが Servlet クラス用のアスペクトから 2 個，そのコールグラフ上のクラス用のアスペクトから 5 個選択する．そして，全ユーザ（100 ユーザ）がアプリケーション内の Servlet クラスそれぞれに対しランダムにリクエストを送り，その平均応答時間を測定

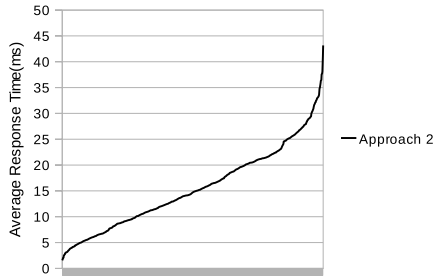

図 8　平均応答時間（手法 2）
Fig. 8　Average response time (Approach 2).

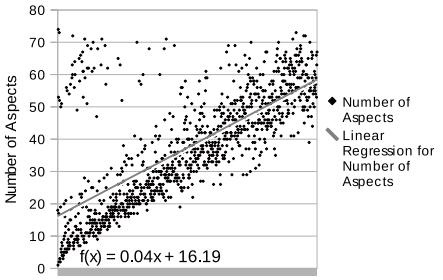

図 9　アスペクトの使用数
Fig. 9　The number of used aspects.

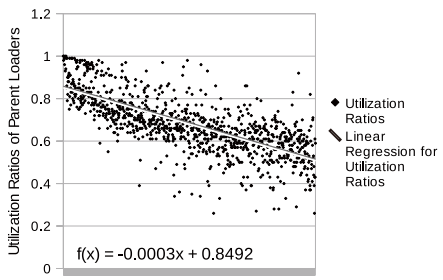

図 10　クラスローダの平均活用率
Fig. 10　Average utilization ratios of class loaders.

する．選択パターンをこのように定義した理由は，Web サイトの見栄えや機能は，サイトのユーザが最も興味を持つ部分であり，それらを実現する機構の中でメインロジックを担う Servlet クラスが最も変更されやすく，次いで，そのコールグラフ上のクラスも変更されやすいだろうと予想したからである．

5.1　実 験 結 果

図 8 に手法 2 を用いた場合の平均応答時間を示す．このグラフは，アスペクトの選択パターン別の実験結果を平均応答時間の昇順にソートしたもので，横軸がアスペクトの選択パターン，縦軸が平均応答時間である．以下，図 10〜15 はすべて，このグラフの横軸に対応するようにソートされている．

図 9 は各アスペクトの選択パターンごとのユーザのアスペクトの使用数とその線形回帰直線である．アスペクトの使用数とは選択パターン (n, m) における $n + m$ のことである．**図 10** は，手法 2 における再起動時の親ローダの平均活用率とその線形回帰直線である．クラスローダの活用率とは，図 5 の $|A|/|S|$ に相当し，再起動時に変更するクラス（再ロードが必要なクラス）のうち，どれだけロード済みのクラスを再利用できているかを表している．これらのグラフから，平均応答時間はアスペクトの使用数に比例し，親ローダの活用率に反比例すると考えられる．

図 11 と**図 12** はそれぞれ手法 0 を用いた場合と手法 1 を用いた場合の平均応答時間とその線形回帰直線である．ほとんどのアスペクトの選択パターンにおいて手法 1 を用いた再起動のほうが高速であった．

図 13 は図 11 の手法 0 の平均応答時間の線形回帰直線と図 8 の手法 2 の平均応答時間を重ねたグラフである．ほとんどのアスペクトの選択パターンにおいて手法 2 を用いた再起

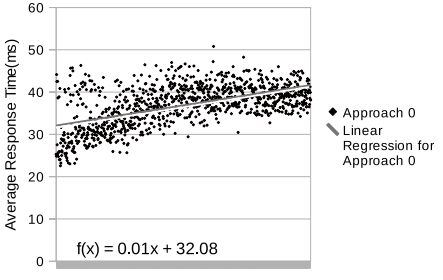

図 11　平均応答時間（手法 0）
Fig. 11　Average response time (Approach 0).

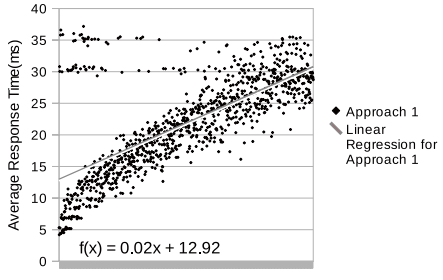

図 12　平均応答時間（手法 1）
Fig. 12　Average response time (Approach 1).

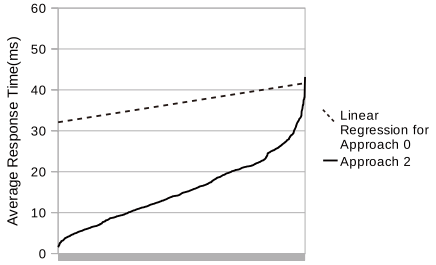

図 13　平均応答時間の比較（手法 0，手法 2）
Fig. 13　Comparison of average response time between Approach 0 and Approach 2.

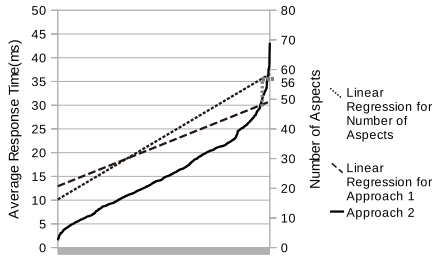

図 14　アスペクト使用数と平均応答時間の関係
Fig. 14　The number of used aspects and average response time.

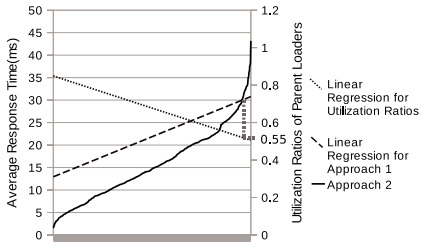

図 15　クラスローダ活用率と平均応答時間の関係
Fig. 15　Average utilization ratios of class loaders and average response time.

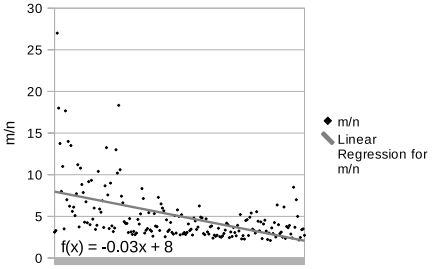

図 16　m/n の推移
Fig. 16　m/n values.

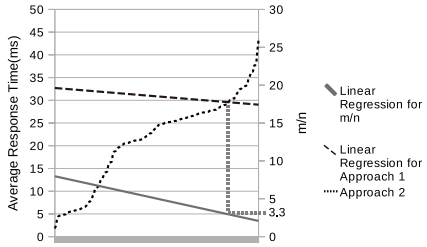

図 17　m/n と平均応答時間の関係
Fig. 17　m/n values and average response time.

動のほうが高速であった．

図 14 は手法 1，2 の平均応答時間とアスペクトの使用数との関係，**図 15** は親ローダの活用率との関係を示したグラフである．図 14 を見ると，アスペクトの使用数が 56 個（Health Watcher 内の全クラスの 76%）以上の場合は手法 1 を用いた再起動のほうが高速である．このグラフから，非常に多くのクラスを変更するときは手法 1 を使ったほうが良いと分かる．また，そのときの親ローダの活用率は 55%であった．図 9 の左上に着目すると，アスペクトの使用数が多い，つまり多数のクラスを変更しているにもかかわらず平均応答時間が短いケースが見られる．これらのすべてがアスペクトの選択パターン (n, m) の n に対して m が非常に大きい場合であった．そこで，アスペクトの使用数が 56 個以上となるアスペクトの選択パターンにおける m/n の測定を行った．その結果を**図 16** に示す．**図 17** は図 16 に手法 1 と手法 2 の平均応答時間を重ねたグラフである．このグラフを見ると，アスペクトの使用数が 56 個以上の場合でも m/n の値が 3.3 を超える場合には手法 2 のほうが高速であると分かる．また，アスペクト使用数が 56 個以上で，手法 1 よりも手法 2 のほうが速くなる選択パターンにおける親ローダの活用率の平均値は 69%であった．

各ユーザが選択した Servlet クラスに対するアスペクトが異なったとしても，そのコールグラフには多くの共通部分があるため，必然的に各ユーザが選択するコールグラフ上のクラスに対するアスペクトも類似する．このため，あるユーザが再起動の際使用したクラスローダ群を他のユーザが親ローダとして活用しやすくなったからである．

5.2　手法選択の指針

5.1 節の実験結果から，アプリケーションの大多数のクラスを繰り返し変更するときは手法 1 を，そうでないときは手法 2 を使うべきである．また，per-session AOP フレームワークと Health Watcher を用いる場合は次のようになる．

ユーザのアスペクト使用数がアプリケーションの全クラス数の 76%以上，かつ m/n が 3.3 以上ならば手法 2 を使用する．3.3 より小さいなら手法 1 を使用する．ユーザのアスペクト使用数がアプリケーションの全クラス数の 76%より小さいなら手法 2 を使用する．

手法の切替えを実行する境界値は per-session AOP フレームワークと Health Watcher の組合せに固有である．しかし，一般的な Web アプリケーションの境界値はおおむね似たような値をとると思われる．この点についての検証は今後の課題である．

6. 関 連 研 究

Liang らは論文 3) 内で，Java のクラスローダのアーキテクチャについて述べている．親子関係を用いたロードアルゴリズムや JVM 上での型の実装について記述されており，我々はこれらを利用し，修正したアプリケーションの部分的再ロードを実現した．

JVM 上にロード済みのクラスを新たな定義で置き換える技術に HotSwap がある．この技術は PROSE [12] や DJAsCo [13] などの DAOP（*Dynamic Aspect Oriented Programming*）システムで用いられている．アプリケーションのクラス間の依存関係を考慮することなく，特定のクラスのみを異なる定義で置換できるため，修正したアプリケーションの高速な再起動が実現できる．しかし，HotSwap ではメソッドボディしか変更できないという制限があるため，アプリケーションの再開発には不向きである．

その他のクラスローダの研究として，Sister Namespace [14] がある．この研究では異なるクラスローダで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ることで，あるクラスの版のインスタンスを異なるクラスの版の変数に代入可能にしている．これを修正した

アプリケーションの再起動に用いるためには，アプリケーション内のあらゆる参照をリフレクション API を用いた式に変形する必要がある．たとえば，

```
App app = new App();
```

という式は，

```
Class clazz = loader.loadClass("App");// 異なるクラスの版をロード
App app = (App) clazz.newInstance();
```

のように式変形を行う．これはアプリケーションの初回ロード時に行えばよい．しかし，super() などの特殊なコンストラクタ呼び出しは，リフレクション API を用いて変形できないため，この技術は修正したアプリケーションの再起動に用いることはできない．

7. まとめと今後の課題

部分的再ロードによる Java プログラムの再起動の高速化手法を提案した．アプリケーションの全クラスの再ロードによる再起動は，その時間的コストによりサーバのパフォーマンスを低下させるという問題があった．本稿では，変更対象のクラスとそれに依存しているクラスを子ローダで再ロードし，その他のクラスは親ローダでロード済みのものを再利用するという手法と，アプリケーションの各クラスの各版ごとにクラスローダを作成し，それを親ローダとして活用することで，旧版のアプリケーションを部分的に再利用するという 2 つの手法を提案した．また，これらの手法を per-session AOP フレームワークに実装し，その上で動作する Health Watcher アプリケーションにリクエストを送信する実験を行った．その結果から，2 つ手法のどちらを使うべきかの指針を示した．

今後の課題は，さらなる実験である．本稿ではフィールドを追加するだけのアスペクトが各クラス用に 1 つずつサーバに登録されているという条件下で実験を行っている．アプリケーションに複雑な参照を追加するアスペクトを，各クラス用に複数登録した状況下でも実験を行い，パフォーマンスにどう影響が出るかを検証する必要がある．

参 考 文 献

1) Fowler, M.: Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html
2) 戸部　敦，千葉　滋：Web アプリをユーザ毎にカスタマイズ可能にする AOP フレームワーク，楽天研究開発シンポジウム (2008).
3) Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA'98*), Vol.33, No.10 of ACM SIGPLAN Notices, pp.36–44, ACM Press (1998).
4) Fleury, M. and Reverbel, F.: The JBoss Extensible Server, *Proc. ACM/IFIP/USENIX International Middleware Conference* (*Middleware 2003*), pp.344–373, Springer-Verlag (2003).
5) The Apache Software Foundation: Apache Tomcat. http://tomcat.apache.org
6) The Seasar Project: Seasar2. http://s2container.seasar.org/2.4/ja/
7) 別役浩平，千葉　滋：ユーザ毎にカスタマイズ可能 Web アプリケーション用のフレームワークの実装，並列/分散/協調処理に関するサマー・ワークショップ（SWoPP 2009）(2009).
8) Chiba, S.: Load-time Structural Reflection in Java, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2001*), pp.313–336, Springer-Verlag (2000).
9) Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch, *The ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA 2010*), pp.539–554, ACM Press (2010).
10) Greenwood, P., Bartolomei, T.T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U. and Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2007*), pp.176–200, Springer-Verlag (2007).
11) The Apache Software Foundation: Apache JMeter. http://jakarta.apache.org/jmeter/
12) Popovici, A., Gross, T. and Alonso, G.: Dynamic weaving for aspect-oriented programming, *Proc. 1st International Conference on Aspect-Oriented Software Development* (*AOSD 2002*), pp.141–147, ACM Press (2002).
13) Daniel, L., Navarro, B., Sdholt, M., Vanderperren, W., Fraine, B.D. and Suve, D.: Explicitly distributed AOP using AWED, *Proc. 5th International Conference on Aspect-Oriented Software Development* (*AOSD 2006*), pp.51–62, ACM Press (2006).
14) Sato, Y. and Chiba, S.: Loosely-separated "Sister" Namespaces in Java, *Proc. European Conference on Object-Oriented Programming* (*ECOOP 2005*), pp.49–70, Springer-Verlag (2005).

(平成 22 年 12 月 17 日受付)
(平成 23 年 3 月 29 日採録)

別役　浩平

2009 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻在学中．システムソフトウェアの研究に従事．

千葉　　滋（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年同大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師を経て，現在，東京工業大学大学院情報理工学研究科教授．博士（理学）．システムソフトウェアの研究に従事．