情報処理学会論文誌　プログラミング　Vol.5　No.3　40–49 (Aug. 2012)

Reserved Member方式によるメンバの実行時追加機構の提案

早船 総一郎[1]　千葉 滋[1,a)]

受付日 2011年12月16日, 採録日 2012年3月21日

概要：本論文では reserved member 方式による Java 言語のためのメンバ（メソッド他）の実行時追加機構を提案する．標準の Java 仮想機械は実行時のメンバの追加に対応しておらず，HotSwap 機能を用いても既存メソッドの本体の書き換えしかできない．メンバの型の変更などはできない．提案手法では，準備として，実行前（たとえばロード時）に予備のメンバをクラスにあらかじめ追加しておく．これを reserved member と呼ぶ．実行開始後にユーザがメンバの追加操作を行ったときには，その予備のメンバを変更して，追加されたメンバの機能を実装する．追加されたメンバを参照する式は，その予備のメンバを参照するように修正される．また追加されるメンバがメソッドであるときは，その本体が予備のメンバにコピーされる．また型の整合性をとるために，型変換のコードが適宜挿入される．提案手法はメソッドの上書きに対応しており，また少ない数の予備のメンバでも任意個のメンバを追加できる．本論文ではさらに提案方式のオーバヘッドについても述べる．

キーワード：Java，動的拡張，言語処理系

Reserved Members for Adding a Member to a Class at Runtime

Soichiro Hayafune[1]　Shigeru Chiba[1,a)]

Received: December 16, 2011, Accepted: March 21, 2012

Abstract: In this paper, we present a system using a reserved member to add a member such as a method at runtime in Java. The standard Java virtual machine does not allow adding members at runtime. It can only provide HotSwap, which allows only redefining the body of an existing method; it does not allow changing types of members. The proposed system adds a member, called a reserved member, to classes before execution, for example, at load time. When the user changes a program to add a new member to a class, all the expressions referring to the added member are modified to refer that reserved member. Then, if the added member is a method, the body of the added member is copied into the reserved member. For preserving the type consistency, the system also inserts type conversion code at appropriate places. The system supports method overriding and it allows adding any number of members by using a limited number of reserved members. The performance overheads are also presented in the paper.

Keywords: Java, dynamic evolution, language processor

1. はじめに

システムを開発するにあたりプログラムを 1 度完成させても，まだ開発は完全に終了したわけではない．プログラムが実行中でサービスを提供している状態になっていてもメンテナンスを行う．通常プログラムの更新は実行しているプログラムをいったん停止させ，パッチを当て，再度起動する．しかし，このように停止している間はサービスを提供できないため，大きな損害となる．

そこで，実行しているプログラムを停止させることなくコードを更新する技術が必要である．このような技術は昔から研究が行われ，Smalltalk，Python，Ruby のような動的な言語では実行しているプログラムに動的な変更を行える機構を備えていることが多い．動的言語に備わっている機構ではクラス定義を変更できるだけでなく，より多くの

[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, Meguro, Tokyo 152–8552, Japan
[a)] chiba@acm.org

変更を行える [1], [2], [3]．本システムで行う動的にメンバを追加するという機能はクラス定義を変更することである．一方，Java では静的な型付け言語であるため動的な変更は難しく，実行しているプログラムを変更する機能は標準 Java 仮想機械では部分的にしか提供していない．特に新しいメンバをクラス定義に追加することができない．

我々は reserved member 方式によるメンバの実行時追加機構を提案する．本システムは Java 仮想機械に手を加えていないため，既存のシステムにも導入しやすい．特に標準の HotSwap を利用している既存のシステムの実装に本システムを利用することで，新たな機能拡張を行うことができるようになる．たとえば，標準の HotSwap を利用している Dynamic Aspect-Oriented Programming（DAOP）の実装に本システムを利用すれば，AspectJ [4], [5] で提案されているインタータイプ宣言を動的に実行できるなどの応用が考えられる．

Reserved member 方式では以下の 2 段階の処理により動的なメンバの追加を実現した．1 段目は，実行前に各クラスへ事前に reserved member を追加しておくことで，実際には動的な追加を行わないように準備することである．2 段目は，ユーザがメンバの追加の操作を行ったときに，追加したいメンバの処理を reserved member で実現し，メンバの呼び出しを書き換えることである．

以下，2 章では動的なクラス定義の変更について述べる．3 章では reserved member 方式について述べる．4 章では我々が作成したベンチマークや SPECjvm2008 [6] を用いてオーバヘッドの計測を行う．5 章では関連研究について取り上げ，6 章で本論文をまとめる．

2. 動的なクラス定義の変更

Java プログラムのクラス定義を動的に変更する機能は様々な応用が考えられ，必要性が高いとされている．たとえば，デバッグモードで実行中のプログラムを一時停止させ，クラス定義を更新して，プログラムを再開することができるようになる．このような変更を行うことで効率良くプログラムの開発をすることができる．また，動いている Web アプリケーションを止めずに，セキュリティパッチを当てることができる．新しいクラス定義をセキュリティパッチとして用意し，既存のクラスの定義をこの新しいクラス定義に更新すればよい．さらに，動いている Web アプリケーションを止めずに，オンサイトで性能ボトルネックを調査することにも使える．プロファイルデータを収集するコードを組み込んだクラス定義に実行中に更新することでそのような調査が可能である．

2.1 典型的なユーザの操作

このように動的にクラス定義を変更し実行する場合，一般的に次のような手順がとられる．まず，実行中の Java 仮想機械に対し，プロセス間通信でコネクションを張り，クラス定義を変更したいときにユーザが対話的に新しいクラス定義のプログラムを送り込む．Java 仮想機械は送り込まれたクラス定義を利用し，その変更を既存のクラス定義に反映する．送り込むクラス定義はバイトコードにコンパイル済みで，型検査などはオフラインで実行するものとする．オフラインの準備では変更されないクラスも用意して，それら全体を合わせたうえで整合性がとれているか検査を実行する．

2.2 既存の手法

このような動的なクラス定義の変更の必要性はよく知られているため，標準 Java 仮想機械でも一部の機能はすでにサポートされている．これを用いれば Java プログラムの実行状態を監視し，クラスを定義するクラスファイルをバイトコードレベルで新しいものに動的に交換する Java プログラムを書くことができる．この機能は java.lang.instrument パッケージ [7] で提供されており，HotSwap と呼ばれる．このとき，新しいクラス定義のメソッドボディの内容を異なるものに変更することができる．しかし，変更後のクラスは型に関して既存の他のクラスと整合性がとれていなければならず，新しいメンバを追加することはできない．これは仮想関数テーブルを変更できないためであり，それに基づいた機能もあるためと思われる．

HotSwap 機能は非常に制限が大きいので，Java 仮想機械を改造し機能を拡張する研究が行われてきた．動的な変更のための特別なクラスとクラスローダを定義することで，より広範囲な種類のクラスの変更を可能にしているシステムが存在している [8]．さらに Java 仮想機械を改造することでメンバの追加や削除だけでなく，クラスの継承関係の変更も行えるシステムも存在している [2], [9]．このように Java 仮想機械を改造することで動的にクラス定義を置換可能にする方法はよく知られている．しかし，実装が改造される Java 仮想機械のバージョンに依存しており，Java 仮想機械のバージョンアップに対応できないという問題がある．そのため，標準 Java 仮想機械を用いたうえでクラス定義を動的に変更する方法が望ましい．

3. Reserved Member 方式

我々は reserved member 方式による Java 言語のためのメンバの実行時追加機構を提案する．我々が提案する reserved member 方式では，reserved member を事前に追加しておき，後から実行中にメンバが追加されたときには，reserved member を修正して追加されたメンバを実現する．これによって，HotSwap の範囲内で動的なメンバの追加を可能にする．標準で用意されている HotSwap ではメソッドボディの変更のみが許され，シグネチャの変更ができない．そのため，reserved member の型はどのようなメンバ

```
1 Object reserved$(Object[] objects, String key){
2   return null;
3 }
```

図 1　継承関係がない
Fig. 1　Reserved method for a class dose not have an inheritance relationship.

```
1 Object reserved$(Object[] objects, String key){
2   return super.reserved$(objects, key);
3 }
```

図 2　継承関係がある
Fig. 2　Reserved method for a class has an inheritance relationship.

でも追加できるようにし，整合性をとるため動的に型変換を行うコードを加える．さらに複数のメンバ追加に対応するため，ディスパッチを行うコードを加えて実現する．

本システムは java.lang.instrument パッケージを通して HotSwap 機能を用い，実行されるプログラムを監視する．この機能を用いると，あらかじめ用意しておいた premain メソッドが main メソッドが実行される前に呼び出される．premain メソッドは，各クラスのロード時に本システムの方式に従ってクラス定義が変換されるように Java 仮想機械を HotSwap 機能を使って設定する．クラス定義を変換するコードは Javassist [10] を用いて我々が実装した．このコードは，各クラスの定義を次に述べる reserved member を元の定義に加えたものに置き換える．この置き換えは実行時ではなくロード時に 1 度だけ行うため，標準の機能で実現できる．

ユーザが実行中に動的にメンバの追加を指示すると，外部プロセスが HotSwap 機能経由で Java 仮想機械に割込みをかける．Reserved member の中身を変更して，追加を指示されたメンバを実現したクラス定義を作り，元のクラス定義を HotSwap 機能を使って置き換える．これにより，メンバの実行時の追加を実現する．

3.1　Reserved Member の追加

Reserved member はどのようなメンバが任意個，追加されても対応できるようにする必要がある．標準で用意されている HotSwap ではメンバの追加を許していないため，各クラスの実行前に reserved member を事前に追加しておく．本システムの場合，reserved member の追加は本システムが行うため，本システムを利用するユーザは，事前にメンバを追加しておくことを意識する必要がない．

3.1.1　フィールドの追加

準備しておくフィールドはどのような追加，任意個の追加どちらにも対応できるように用意する必要がある．そのため，String を key とし Object を value とする HashMap を reserved field とする．この String の key を用いてディスパッチを行う．このフィールド名は他のフィールドと重複しないように準備する．今回は reserved$ という名前で始まるフィールドは各クラスに定義されていないなど仮定を設けている．そのため，reserved field の宣言は以下のようなものになる．

```
HashMap reserved$ = new HashMap();
```

3.1.2　メソッドの追加

準備しておくメソッドもどのような追加，任意個の追加どちらにも対応できるように用意する必要がある．どのような追加にも対応するために，戻り値の型は Object とし，引数の型は Object の配列と String にしている．これによって後にどのようなメソッドの追加があっても Object の配列に変換することで引数に渡すことができ，戻り値を Object として受け取ることができる．引数の String はディスパッチ機能のための準備であり，追加されるメソッドを実現する際に利用する．さらにメソッド名は各クラスで定義されているものと重複しないように定めなければならない．今回は reserved$ という名前で始まるメソッドが存在しないなど仮定を設けている．そのため，reserved method の宣言は基本的に**図 1** のようになる．

しかし，メソッドにはオーバライドされる可能性があるため，一概に図 1 のようにメソッドを準備してはならない．もし，すべてのメソッドを図 1 のように準備するとオーバライドされた場合に正しい振舞いを実現できない．そのため，親クラスが String のような組み込みのクラスでなく，クラス定義の修正が可能なクラスである場合には**図 2** のようなメソッドをロード時に追加する．追加するメソッドは親クラスの同名のメソッドを呼ぶだけのメソッドである．より詳細なオーバライドに関する考慮は後で議論する．

3.1.3　コンストラクタの追加

コンストラクタはメソッドと異なり，戻り値の型とコンストラクタ名が決まっている．したがって，名前を変えることで重複を避けることができない．そこで，重複を回避するために既存のものと重複しないダミーのクラスを作成し，ダミーのクラスを引数として持つことで他のコンストラクタと区別する．今回は Reserved$ という名前で始まるクラスが存在しないという仮定を設けている．戻り値の型はそのクラス自身であるので，変えることはできないが

```
1  public class Position {
2    int x,y;
3    public Position(int x, int y){
4      this.x = x;
5      this.y = y;
6    }
7    public static void main(String[] args){
8      Position p = new Position(19, 87);
9      double d = p.distance(5, 21);
10   }
11   public double distance(int i, int j){
12     double d2 = Math.pow(x - i, 2) + Math.pow(y - j, 2);
13     double d = Math.sqrt(d2);
14     return d;
15   }
16 }
```

図 3　Position クラスに distance メソッドを追加
Fig. 3　Adding the distance method to the Position class.

コンストラクタを追加するにあたり問題はない．引数はメソッドと同様に Object の配列と String，そして先ほどのダミーのクラス Reserved$ を引数として受け取る．引数の String はディスパッチ機能のための準備であり，追加されるメソッドを実現する際に利用する．以上より reserved constructor の宣言は以下のようなものになる．

```
<classname>(Object[] objects, String key,
            Reserved$ r){}
```

3.2　Reserved method の変更による追加されるメソッドの実現

ユーザが追加の指示を行ったときには，実行前に追加しておいた reserved method を変更することによって，追加されるメソッドの機能を実現する．2 章の典型的なユーザの操作のように新しいクラス定義が本システムに渡される．新しいクラス定義と古いクラス定義を比較し，追加されたメソッドを見つけ出し，その追加されたメソッドの機能を reserved method に実現する．Reserved method に対して，メソッドボディのコピー，引数と戻り値の型変換，追加されたメソッドの切替えを挿入することで追加されたメソッドの機能は実現できる．

たとえば，**図 3** のように Position クラスに distance メソッドが追加されたときを考える．ユーザはこのソースコードをコンパイルして，新しいクラス定義であるクラスファイルを準備し，本システムに与える．そして，本システムは reserved method が追加されている Position クラスに対して**図 4** のような変更を加えて distance メソッドの追加を実現する．

3.2.1　Reserved Method の呼び出し

新しく追加したメソッドを呼び出している場所では，代わりに reserved method を呼び出すように変換を行う．この際に動的ディスパッチおよびメソッド・オーバライドを考慮して，追加するメソッドを呼び出す可能性があるメソッドの呼び出し式をすべて変換する．追加するメソッドを呼び出すのはメソッドの追加時に一緒に新規に加えられたクラスだけとは限らず，元から存在しているクラスが呼び出す場合もあるため，すべてのコードを確認し，必要に応じて変換を行って標準の HotSwap で既存のコードと置き換える．

Reserved method の呼び出しでは，引数を Object 型の配列に変換し，加えて，呼び出したい本来のメソッドのシグネチャを文字列として引数とする．Object 型の戻り値は，元々呼び出されていたメソッドの戻り値の型に変換される．追加されたメソッドの切替えで利用するキーを引数に持たせて呼び出しが行われる．

図 4 では 10, 11 行目が reserved method の呼び出しである．int 型の引数 2 つを Object 型の配列に変換し，distance メソッドのシグネチャの String を引数に持って，reserved method を呼び出している．Reserved method の戻り値は Object 型であるので，double 型に型変換を行う．

3.2.2　追加されたメソッドの切替え

ユーザによって行われるメソッド追加は複数であることもあり，事前に準備した 1 つの reserved method が複数のメソッドを実現する．Reserved method の呼び出しの際に引数として渡された String を使って切替えを行う．追加されたメソッドは，reserved method にメソッドボディをコピーし，引数，戻り値の型変換を追加することで実現される．そのため，実現されたメソッドを囲うような if 文を作ることで切替えを行う．戻り値の型変換によって実現されたメソッドは必ず return 文を持っているので，else がなくても複数の処理が行われることはない．追加されたメソッドの機能を切り替えるために追加されたメソッドのシグネチャを文字列にして，比較を行う．その文字列は reserved member の呼び出しの際に実引数の 2 番目として渡され，追加されたメソッドの切替えのみに利用され，選択されたメソッドのボディ内では利用されない．図 4 では 14 行目のような if 文が作成される．第 2 引数と事前に用意してい

```
1  public class Position {
2    int x, y;
3    public Position(int x, int y){
4      this.x = x;
5      this.y = y;
6    }
7    public static void main(String[] args){
8      Position p = new Position(19, 87);
9      /* reserved method call */
10     double d = (double) p.reserved$(new Object{5, 21},
11       "distance(I,I)D");
12   }
13   public Object reserved$(Object[] v1, String v2){
14     if(v2.equals("distance(I,I)D")){ // dispatch
15       /* parameterCast */
16       v2 = (int) v1[1];
17       v1 = (int) v1[0];
18       /* distanceの処理のコピー */
19       double v3 = Math.pow(x - v1, 2) + Math.pow(y - v2, 2);
20       double v4 = Math.sqrt(v3);
21       /* returnCast */
22       return Wrapper.make(v4);
23     }
24     return null;
25   }
26 }
27
28 public class Wrapper {
29   public static Object make(double d) {
30     return new Double(d);
31   }
32 }
```

図 4　Reserved method を利用して distance メソッドを実現
Fig. 4　The distance method is achieved by the use of the reserved method.

た文字列を比較する．この文字列は distance メソッドで，引数が int，int 型で，戻り値が double 型であることを表している．

3.2.3　引数の型変換

引数の型に関して整合性をとるため，reseved method の引数を適切な変数に準備する必要がある．単純に追加されるメソッドのボディをコピーしただけでは，整合性がとれないアクセスを行ってしまう．プリミティブ型に変換する場合には，アンボクシングのためのバイトコードが増える．ソースコードからバイトコードにコンパイルする場合にはオートボクシングが行われるが，バイトコードレベルでの変換なので，システムがアンボクシングを行うバイトコードを生成する必要がある．

Java では処理を行う際にメソッドの引数と局所変数は同じ変数として扱われ，前から順に番号付けされている．変数の前から順に引数が利用して，残りを局所変数が利用することを考慮し，Object の配列で渡された値を展開する．この展開の際に Object 型から適切な型に変換を行う．

単純に追加されるメソッドのボディをコピーした場合，いっさいの変更が加えられてないため，Object の配列や String 型の変数，何も用意されていない変数に整合性のとれないアクセスをしてしまう．そのため，先頭の引数が利用する変数に Object の配列で渡された値を変数として展開できればよい．この際に reserved method の本来の引数である Object の配列と String を考慮しなければならないが，String 型の変数はディスパッチのみで利用し，実現されたメソッドが選択された後は利用されない．また，Object の配列も同様に渡された値を展開した後では，Object の配列を利用されない．そのため，コピーしたメソッドボディではそのものを利用することがないため上書きしても問題はない．よって，先頭の引数が利用する変数に Object の配列で渡された値を変数として展開できればよい．この展開を行う際に Object から本来の型に変換を行う．また展開を行う際に Object の配列から値を取り出すため，Object の配列を上書きしてしまうとそれ以上値を取り出すことができない．そのため，第 2 引数以降を展開してから，最後に Object の配列を本来の第 1 引数に上書きする．

また，利用していない局所変数に展開する方法もあるが，追加されるメソッドのボディに変更を加えて引数へのアクセスをその局所変数に置き換える必要があり，利用する局所変数が増えるため，上記の方法を選択して最適化を行った．

例として，distance メソッドが reserved method を利用して実現する場合，引数の型変換は図 4 の 16，17 行目のようになる．バイトコードレベルでは，引数と局所変数を同様に扱うので，説明のため先頭から番号付けして v1, v2, … とする．型変換ではバイトコードレベルで行われるので，ソースコードレベルでは正しく記述できない．そのた

め，図 4 では類似したコードで表す．distance メソッドのボディでは int 型の v1，v2 を利用する．

3.2.4　メソッドボディのコピー

メソッドを追加するときは，そのメソッドボディをあらかじめ準備しておいた reserved method にコピーする．このようなコピーを行う際に問題となるのは this や super の扱いである．this に関しては同じクラスに定義されたメソッドにコピーが行われるため，変更を加える必要がない．super に関してはオーバライドを考慮する必要がある．後で議論するがオーバライドを考慮する際に変更を加えなくてよいように対応している．したがって，直接メソッドボディをコピーするだけでよい．例として，distance メソッドが reserved method を利用して実現する場合，コピーされた処理は図 4 の 19，20 行目のようになる．

3.2.5　戻り値の型変換

戻り値に関しても型を変換する必要があり，プリミティブ型のときのみボクシングを行う．バイトコードレベルでは適切なボクシングを行うのが難しかったため，システム側で用意しておいたメソッドを呼び出す．そのメソッドを呼び出すときにはプリミティブ型の戻り値を引数として呼び出しを行い，void もプリミティブ型なので，必ず return 文が存在することになる．

例として，distance メソッドが reserved method を利用して実現する場合，戻り値の型変換は図 4 の 21 行目のようになる．distance メソッドは戻り値が double なので，プリミティブ型の値をボクシングして，Object 型として戻す Wrapper クラスの make メソッドを実行時に呼び出す．

3.3　Reserved Constructor の変更による追加コンストラクタの実現

コンストラクタは特別なメソッドとして扱えるため，reserved constructor は reserved method とほぼ同様に利用することができる．コンストラクタとメソッドの違いとして戻り値の型が自身のクラスであるということが決まっているため，戻り値の型変換は行わなくてよい．呼び出しではメソッドと同様に Object の配列と String を持つが，加えて重複を避けるためのクラスを引数として持つ必要がある．その際にインスタンスを作成する必要はなく，null を Reserved$ クラスに変換すればよい．

3.4　Reserved field を利用することによる追加フィールドの実現

フィールドはボディが存在しないため，引数の型変換とボディのコピーと戻り値の型変換が必要なく，アクセス側の変換のみで実現する．元々のフィールド名をキーに get メソッドを呼び出すことでハッシュ表から値を取り出し，put メソッドを呼び出すことでハッシュ表に値を代入する．

3.5　詳細の考慮

3.5.1　オーバライドの考慮

メソッドはオーバライドがあるため，追加する reserved method は親クラスの reserved method を呼び出すようにしていた．ユーザによってメソッドが追加された場合，対象のクラスの reserved method に追加されるメソッドが実現される．この対応で，継承関係のあるクラスですでに定義されてないメソッドの追加は実現される．

追加するメソッドが継承関係のあるクラスですでに定義されている場合は，すでに定義されているメソッドと reserved method でオーバライドの関係が生じる．異なるシグネチャ間でオーバライドの関係を実現するのは難しいため，既存のメソッドを reserved method で処理を実現し，reserved method でオーバライドの関係を実現する．

組み込みのクラスでは reserved member を追加することができないため，上記のように reserved method に処理を実現することができない．しかし，そのようなクラスは限定されているため，用意すべきメソッドは有限で収まる．そのため，あらかじめ対象メソッドを reserved member の追加と同じタイミングで追加を行うことで対応する．

3.5.2　修飾子など

HotSwap では修飾子も変更できないため，事前に準備しておく reserved member は修飾子に関しても考慮されて用意される．修飾子の一部はアクセス修飾子と呼ばれ，そのメンバの振舞いに影響を与える．そのため，複数種類の reserved member を準備することで対応するが，それにあわせて名前を変更して追加する必要がある．

コンストラクタはアクセス修飾子によって記述を制限されるが，すべてのアクセスを許可されたコンストラクタに置き換えても，振舞いに影響はない．フィールドは static であるかどうかによって行われる処理が異なる．そのため，2 種類の reserved field を用意しなければならない．他のアクセス修飾子に関しては振舞いに影響しない．メソッドは static であるかどうか，private であるかどうかに影響を受け，その振舞いを変える．protected や記述を省略したメソッドは，記述に制限があるが public なメソッドと同じ振舞いであり，public なメソッドでその機能を実現する．そのため，4 種類の reserved method を用意する．アクセス修飾子以外では synchronized は reserved method を利用してバイトコード変換を行うことで実現できる．

修飾子と似たようなものに throws 節がある．しかしこれは実行時に無視されるため何もしなくても問題ない．

3.5.3　制限

Java では 1 つのメソッドで利用できる容量として 64 KB という制限がある．そのため，ディスパッチ機能で 1 つの reserved member が複数のメンバを実現することができるが，追加される量によってはその限界を超えてしまう．対応策としては，reserved member の数を増やすことで容量

を増やすことが可能である．しかし，この対応策では実行前にその判断をしなければならず，予想を上回る追加が行われれば，同様な問題が発生する．

これまで説明したように，本システムはすでに実行中のクラスを新しいクラスで置き換える．この変更が反映されるタイミングはユーザが追加を指示したタイミングによって決まるため，一貫性がとれない可能性がある．標準のHotSwapの仕様により，すでに呼び出しが行われ，アクティブなスタックフレームが存在する場合，メソッドを再定義しても，元のメソッドのバイトコードが引き続き実行される．再定義されたメソッドは新たに呼び出しが行われたときに初めて実行される．

さらに，本システムで追加を行う場合には呼び出し側の変換が必要となるため，HotSwapの仕様では再定義されたメソッドが呼び出されるタイミングであっても呼び出されないことがある．呼び出し側のメソッドがすでにアクティブなスタックフレームを持っている場合にはメソッドの再定義は反映されない．

4. 実験

Reserved member方式では準備や呼び出し，メモリ使用量のオーバヘッドが大きいと考えられる．そこで，我々は自作のマイクロベンチマークやSPECjvm2008を用いて，オーバヘッドを計測した．動作環境は，CPUはIntel Core2Duo 1.83 GHz，メモリは1.5 GB，OSはWindowsXPである．

4.1 Reserved Memberの追加

Reserved member方式でオーバヘッドが大きいと予想されるのは，実行前にreserved memberを追加する処理である．まずこのオーバヘッドを計測する．今回はロード時に全クラスにreserved memberを追加するシステムを用いて計測を行った．Reserved memberを追加するオーバヘッドを計測するため，非常に多数のクラスを準備する．各クラスは**図5**のようなプログラムを用いてロードする．このプログラムを通常どおり実行した場合とreserved memberを追加せずHotSwapの機能のみ有効な場合，reserved memberを5個追加して，新しいメンバを追加できる場合の実行時間を比較する．ウォーミングアップとしてまず1,000個の異なるクラスをロードした．その後，1,000個の異なるクラスをロードするときにかかる時間を計測した．したがって図5のnとmは1,000である．

結果は通常どおりの場合，クラス1個のロード時間は平均0.410 msであり，標準偏差は0.015 msである．HotSwap機能のみの場合はクラス1個のロード時間は平均0.561 msであり，標準偏差は0.007 msである．Reserved memberを追加した場合はクラス1個のロード時間は平均1.642 msであり，標準偏差は0.036 msである．HotSwap機能の利用にともなうオーバヘッドは，java.lang.instrumentパッケージが提供する機能を用いて，実行されているプログラムを監視するオーバヘッドであり，約37%かかっている．またreserved memberの追加を行った場合は約4倍ほどのオーバヘッドがかかっている．

4.2 Reserved Methodの呼び出し

Reserved member方式では振舞いが正しくなるように必要な処理を加えているため，そのオーバヘッドを計測する．一部のメソッドをreserved methodに置き換えることでオーバヘッドを計測するが，そのためにまず対象のメソッドがどれくらいの回数呼び出されるかを確認した．SPECjvm2008のベンチマークの1つXMLを選び，メソッド呼び出しを確認したところ，transform.ExtOutputStream.isDosNewLineとtransform.ExtOutputStream.isUnixNewLineがそれぞれ全体の約33%ずつの回数呼ばれていた．

次に通常どおり実行した場合と約66%のメソッドをreserved methodで実現した場合を比較する．通常どお

```
1  class Main {
2   public static void main(String[] args) {
3     // warmup
4     // n回ロードを行う
5     new A0();
6     new A1();
7     …
8     new An−1();
9
10    // m回ロードを行う
11    long start = System.currentTimeMillis();
12    new An();
13    new An+1();
14    …
15    new An+m−1();
16    long end = System.currentTimeMillis();
17    System.out.println(end − start);
18  }
19 }
```

図5　クラスをロードするマイクロベンチマーク
Fig. 5　The micro benchmark loads classes.

り実行した場合は平均 20.62 ops/m であり，標準偏差 0.423 ops/m である．Reserved method で実現した場合は平均 20.80 ops/m であり，標準偏差 0.461 ops/m である．約 0.8%のオーバヘッドとなり，非常に小さいことが分かる．

4.3　メモリ使用量

Reserved member を追加することでクラスファイルが大きくなり，それによってメモリの使用量が増加すると考え，通常のクラスファイルと reserved member を追加したクラスファイルの大きさを比較した．SPECjvm2008 の spec.benchmarks.check，spec.benchmarks.sunflow パッケージを除いて変換を行った．その結果，945 KB と 1,038 KB となり，約 10%の増加となった．

4.4　差し替え能力の評価

本システムの機能である実行中にコードを差し替える機能の能力をオープンソフトウェアのバージョンアップを行うことで評価した．本システムはユーザによる変更が行われた際にクラス定義を差し替えることで動的な振舞いを変更するが，実験ではオープンソフトウェアである JHotDraw のバージョンアップをユーザによる変更ととらえ，実験を行った．

本システムは JHotDraw のバージョン 7.5 を実行して，その実行中に JHotDraw のバージョン 7.5.1 のクラスにすべて差し替えることに成功した．このバージョンアップにおける差分は 11 個の java ファイルが変更されている．それにともないドキュメントの html ファイルやビルドを行う xml ファイルなどが変更されているが，動的に振舞いを変更するのに影響を与えない．先ほどの 11 個の java ファイルに対して行われた変更は，コメントなどバイトコードに影響を与えないものが 29 カ所，標準の HotSwap 機能で変更できるものが 11 カ所，reserved member 方式を用いたことで変更が可能となったものが 1 カ所となった．Reserved member 方式を用いたことですべての変更が反映できた．

同様に JHotDraw のバージョン 7.4 からバージョン 7.4.1 に差し替えることに成功した．java ファイルはメソッドボディの変更のみであり，標準の HotSwap 機能で解決できる変更であった．

同様に JHotDraw のバージョン 7.4.1 からバージョン 7.5 に差し替えを行おうとしたが，いくつか問題があり，すべてのクラスの差し替えは行えなかった．大きなバージョンアップでは様々な変更が含まれるため，reserved member 方式では解決できない場合が存在している．その多くが継承関係に対して変更が行われている場合であり，それにともない多くのメンバが変更されている．また，java ファイル以外にも動的な振舞いに影響があるライブラリなどのファイルにも変更があり，変更を反映することができなかった．

5.　関連研究

本研究の初期の成果は日本ソフトウェア科学会プログラミング論研究会のワークショップにて口頭発表（論文は公表されていない）された [11]．本論文の reserved member 方式はこれを拡張し，任意個のメンバ追加にも対応したものである．

5.1　他のプログラミング言語

実行しているプログラムを動的に変更するという考えは古くから存在している．Smalltalk，Python，Ruby のような動的な言語では実行しているプログラムを動的に変更を行える機構を備えている．そのような動的言語に備わっている機構ではクラス定義の変更だけでなく，より多くの種類の変更を許す [1], [2], [3]．たとえば，Ruby では open class という機能がある．Open class は既存のクラスに対して，自由にメソッドを新たに定義したり上書きしたりすることができる．Open class があると，自由にメソッドを追加できるが，同じ名前のメソッドを追加すると，正しく動かなくなるなどの問題も存在する．このため Smalltalk には classbox [1] という機能が提案されている．Classbox はパッケージのようなもので，classbox を import して，その中のクラスを影響範囲を限定して拡張することができる．これにより open class に存在する問題を回避している．

5.2　Dynamic Aspect-Oriented Programming（DAOP）

DAOP では後から横断的関心事を追加することで，動的に振舞いを変更することができる．実行時に機能を追加したり削除したりすることができる点は，我々の提案と似ている．ほとんどの DAOP システムは実装に HotSwap を用いているが，HotSwap ではメソッドボディの変更しかできないので，その範囲で実現可能な DAOP が提供されている [12], [13], [14]．本システムを DAOP の実装基盤として利用すると AspectJ [5] で提案されているインタータイプ宣言に対応した DAOP が実現できるなどの応用が考えられる．

5.3　Envelope-Based Weaving

事前にメンバを用意するという我々の提案手法の考えは，Envelope-Based weaving [15], [16] に似ている．Envelope-Based weaving ではコンパイル時またはロード時に各クラスのメンバをラップするメンバを追加しておく．この技術によりアスペクト指向プログラミング（AOP）によってモジュール化されたアスペクトを高速に織り込むことができる．我々の提案手法は同様に事前に resserved member を用意しておくが，それを新規メンバの追加のために使う点がこの研究と異なる．新規メンバの追加のためには様々

な型に対応できなければならないため，Envelope-Based weaving の方式をそのままあてはめただけでは実現できない．

5.4 Java7

Java7 [17] では動的言語サポートのために動的に型付けされるメソッドの呼び出しを可能にしている．そのため新たなバイトコード命令の invokedynamic が追加され，それにともなった新しい構成要素の method handle というリンケージの仕組みが追加されている．invokedynamic は動的言語のためにターゲットの型を指定せずにメソッド呼び出しを表現できるようにしている．これによって動的言語のコンパイラは容易に Java バイトコードを生成することができる．このままではメソッド呼び出しを解決することができないが，method handle を用いることで呼び出すメソッドとの関連を解決している．

Java7 のサポート機能を使うことで，呼び出しを行うバイトコードを生成するための変換を簡素化することができる．しかし，それまで存在しなかったメンバを動的に追加することはできない．このため，4 章で示したような実行中にメンバの追加を行うような変更はできない．

6. まとめ・今後の課題

本論文において我々は，reserved member 方式によるメンバの実行時追加機構を提案した．実行前に reserved member を追加することで，実際に動的な追加を行わないように準備する．ユーザがメンバの追加の操作を行ったときに，reserved member を利用して追加するメンバを実現し，呼び出し側を変換する．これらによって動的なメンバの追加を実現した．

今後の課題としては，システムの拡張を行い実用性を高めることである．他のシステムの実装基盤として利用する場合，追加するメソッドの大きさの合計が追加可能である 64 KB を超える場合がある．そのため，他のアプリケーションを reserved member で実装するようにすべて置き換えた場合，何個の reserved member を用意すれば対応できるかなど実用性を考慮したい．

参考文献

[1] Bergel, A., Bergel, R., Ducasse, S. and Wuyts, R.: *The Classbox Module System* (2003).

[2] Bergel, A., Ducasse, S. and Nierstrasz, O.: Classbox/J: controlling the scope of change in Java, *Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'05*, New York, NY, USA, pp.177–189, ACM (online), DOI: http://doi.acm.org/10.1145/1094811.1094826 (2005).

[3] Clifton, C., Leavens, G.T., Chambers, C. and Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java, *Proc. 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'00*, New York, NY, USA, pp.130–145, ACM (online), DOI: http://doi.acm.org/10.1145/353171.353181 (2000).

[4] The AspectJ project team: The AspectJ project, available from ⟨http://www.eclipse.org/aspectj/⟩.

[5] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ, *ECOOP'01: Proc. 15th European Conference on Object-Oriented Programming*, London, UK, pp.327–353, Springer-Verlag (2001).

[6] Standard PerformanceEvaluation Corporation: SPECjvm2008 (Java Virtual Machine Benchmark), available from ⟨http://www.spec.org/jvm2008/index.html⟩.

[7] The Java project team: *Java*(TM) java.lang.instrument Package, available from ⟨http://download.oracle.com/javase/6/docs/technotes/guides/instrumentation/index.html⟩.

[8] Malabarba, S., Pandey, R., Gragg, J., Barr, E. and FritzBarnes, J.: Runtime Support for Type-Safe Dynamic Java Classes, *ECOOP 2000 — Object-Oriented Programming*, pp.337–361 (2000).

[9] Würthinger, T., Wimmer, C. and Stadler, L.: Dynamic code evolution for Java, *Proc. 8th International Conference on the Principles and Practice of Programming in Java, PPPJ'10*, New York, NY, USA, pp.10–19, ACM (online), DOI: http://doi.acm.org/10.1145/1852761.1852764 (2010).

[10] Chiba, S.: Load-Time Structural Reflection in Java, *Proc. 14th European Conference on Object-Oriented Programming, ECOOP'00*, London, UK, pp.313–336, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=646157.679856⟩ (2000).

[11] 早船総一郎，千葉　滋：標準 Java 仮想機械上で動的にメンバーの追加を行う機構の提案，プログラミングおよびプログラミング言語ワークショップ論文集，PPL2011, pp.131–144 (2011).

[12] Nicoara, A., Alonso, G. and Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs, *Eurosys'08: Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, New York, NY, USA, pp.233–246, ACM (online), DOI: http://doi.acm.org/10.1145/1352592.1352617 (2008).

[13] Sato, Y., Chiba, S. and Tatsubori, M.: A selective, just-in-time aspect weaver, *GPCE'03: Proc. 2nd international conference on Generative programming and component engineering*, New York, NY, USA, pp.189–208, Springer-Verlag New York, Inc. (2003).

[14] Villazón, A., Binder, W., Ansaloni, D. and Moret, P.: Advanced runtime adaptation for Java, *GPCE'09: Proc. 8th international conference on Generative programming and component engineering*, New York, NY, USA, pp.85–94, ACM (online), DOI: http://doi.acm.org/10.1145/1621607.1621621 (2009).

[15] Bockisch, C., Haupt, M., Mezini, M. and Mitschke, R.: Envelope-Based Weaving for Faster Aspect Compilers, *NODe/GSEM*, pp.3–18 (2005).

[16] Bockisch, C., Arnold, M., Dinkelaker, T. and Mezini, M.: Adapting virtual machine techniques for seamless aspect support, *OOPSLA'06: Proc. 21st annual ACM SIGPLAN conference on Object-oriented pro-*

gramming systems, languages, and applications, New York, NY, USA, pp.109–124, ACM (online), DOI: http://doi.acm.org/10.1145/1167473.1167483 (2006).
[17] Oracle Corporation and/or its affiliates: New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine, available from ⟨http://java.sun.com/developer/technicalArticles/DynTypeLang/⟩.

早船 総一郎　（学生会員）

2010 年東京工業大学理学部情報科学科卒業．2012 年同大学大学院情報理工学研究科数理・計算科学専攻修了．プログラミング言語，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，東京工業大学大学院情報理工学研究科教授を経て，2012 年より東京大学大学院情報理工学系研究科教授．博士（理学）．プログラミング言語，システムソフトウェアの研究に従事．