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

早船 総一郎, 千葉 滋

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
www.csg.is.titech.ac.jp/~{hayafune, chiba}

概 要 標準 Java 仮想機械では静的に型の確認を行うことを前提としている。メンバーの追加は型の変更を行うため、動的に型の確認が必要となり、標準で用意されている HotSwap ではメンバーの追加を許しておらず、動的にメソッドボディを変更することしか許していない。そこで、本研究では標準 Java 仮想機械上で実行中の Java プログラムに対して動的にメンバーを追加する機構を提案する。提案手法では、まず、ロード時に各クラスへ事前に空のメンバーをいくつか追加することで、実際に動的な追加を行わずに済むように準備する。次に、ユーザーがメンバーの追加の操作を行ったときに、用意しておいたメンバーを追加するメソッドの内容に書き換え、さらにそのメンバーと呼び出し側に動的に型を確認するコードを加える。これによって動的なメンバーの追加を実現する。

1 はじめに

システムを開発するにあたりプログラムを一度完成させても、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一旦停止させ、パッチを当て、再度実行する。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、より多くの変更を行える [3, 4, 9]。本システムで行う動的にメンバー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。特に新しいメンバーをクラス定義に追加することができない。

我々は標準 Java 仮想機械上で実行中の Java プログラムに対して動的にメンバーを追加する機構を提案する。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。例えば、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 [1, 10] 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ようになるだろう。

本システムでは、以下の 2 段階の処理により動的なメンバーの追加を実現した。一つ目は、ロード時に各クラスへ事前にメンバーをいくつか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。二つ目は、ユーザーがメンバーの追加の操作を行ったときに、追加し

たいメンバーの処理を事前に用意しておいたメンバーに移し替え、さらにそのメンバーと呼び出し側のメソッドのボディ内に動的に型を確認するコードを加えることである。

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 章では標準 Java 仮想機械上で実行中の Java プログラムに対して動的にメンバーを追加する機構とその実装方法について述べる。4 章では我々が作成したベンチマークを用いてオーバー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。例えば、動いている Web アプリケーションを止めずに、新しいクラス定義をセキュリティパッチとして用意する。既存のクラスの定義をこの新しいクラス定義に更新すると、セキュリティパッチを当てることができる。また、動いている Web アプリケーションを止めずに、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。さらに、デバッグモードで実行中にプログラムを止め、一部のクラスの定義を更新できれば効率よく開発ができる。

2.1 典型的なシステム構成

このような動的にクラス定義を変更し実行するには一般的に次のような構成のシステムが使われる。例えば、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、必要に応じてユーザが対話的に新しいクラス定義を送り込む。すると、Java 仮想機械は送り込まれた定義で既存のクラス定義を置き換える。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、型検査等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置換後のプログラムを別に用意して、その上で整合性が取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既にサポートされている。Java プログラムの実行状態を監視し、バイトコードレベルでクラスを定義するクラスファイルを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 [2] 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、新しい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、また新しいメンバーを追加することはできない。これは最適化の技術を阻害するからだと思われる。例えば、インライン展開などは仮想関数テーブルが変更されないことを基に行う。クラス定義を変更すると仮想関数テーブルが更新されるため、そのような最適化は無効にし、元に戻さなければならない。

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。Dynamic Virtual Machine (DVM) [11] では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しており、それを用いることで、より広範囲な種類のクラスの変更を可能にしている。さらに Java 仮想機械を改造することでメンバー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 [4, 15] 。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。だが、実用上は一般的に利用されている標準 Java 仮想機械を用いた上でクラス定義を動的に変更する方法が望ましい。

3　動的にメンバーの追加を行う機構

我々は標準 Java 仮想機械上でメンバーの追加を動的に行う機構を提案する。我々が提案する手法では、汎用性の高いメンバーを事前に追加しておき、後から実行中にメンバーが追加されたときには、このメンバーの中身を修正して追加されたメンバーと等価なものにする。これによって、HotSwap の範囲内で動的なメンバーの追加を可能にする。しかし標準で用意されている HotSwap ではメソッドボディの内部の変更のみが許されているのでシグネチャの変更ができない。そのため、準備するメンバーの型は汎用性を高くしたものとし、それに加えて型の確認を動的におこない整合性をとるコードを必要に応じて既存のコードの中に挿入する。

本システムは java.lang.instrument パッケージが提供する機能を用いて、実行されるプログラムを監視する。この機能を用いると、ユーザーが書いた 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 メソッドで後の動的なメンバーの追加に対応するための準備を実行出来る。具体的にはクラスの再定義を行う機能 HotSwap を用い、各クラスの定義のバイトコードを本章で示すような変換をほどこしたものに置き換えることで、動的なメンバーの準備とする。本システムではバイトコード生成するために Javassist [8] を用いている。

本システムを説明するために図 1, 2 のようなプログラムを用いる。クラス Data とそのサブクラス Shape 、さらにそのサブクラス Rect, Circle があるとする。クラス Data, Circle に setX があったとする。そして、静的な型が Data であるオブジェクトの setX メソッドを呼ぶ set メソッドが他のクラス Main にあったとする。これが初期のクラス定義である。

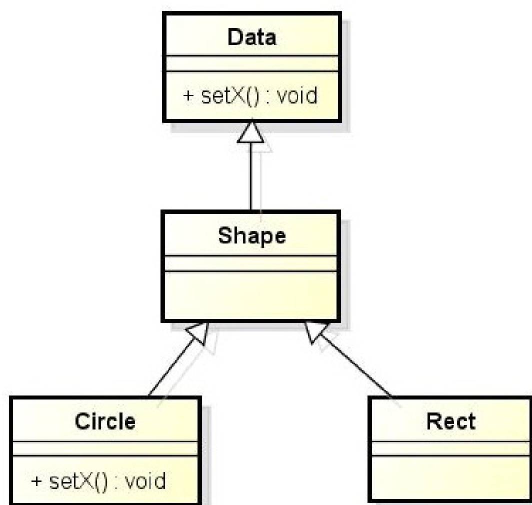

図 1. 初期のクラスの継承関係と定義されているメソッド

```
public class Main{
  public static void main(String
      [] args){
    Data data1 = new Data();
    Data data2 = new Shape();
    Data data3 = new Circle();

    // Data の setX
    set(data1);
    // Data の setX
    set(data2);
    // Circle の setX
    set(data3);
  }
  public static void set(Data d){
    d.setX(1);
  }
}
```

図 2. 図 1 の呼び出しを行う Main クラス

3.1　システム側の事前準備

標準で用意されている HotSwap ではメンバーの追加を許していないため、各クラスのロード時に本システム側でメンバーを事前に準備しておく。本システムを利用するユーザーは、事前にメンバーを準備することを意識する必要がない。準備しておくメンバーはどのようなメンバーの追加に対しても対応できるようにする必要がある。

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にも対応できるように汎用性が高いものである必要がある。そのため、型は Object とし、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 blank$ という名前で始まるフィールドは各クラスで定義されないという仮定を設けている。ロード時に準備のために追加するフィールドは以下のようなものになる。個数は将来、追加が予想される個数で、数個である。

```
Object blank$0 = null;
Object blank$1 = null;
…
```

3.1.2 メソッドの追加

準備しておくメソッドは返り値の型、引数の型、メソッド名に汎用性を持たせる必要がある。これらは標準の HotSwap で動的に変更することができないためである。まず、返り値の型、引数の型は Object と Object の配列にしている。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、返り値を Object として受け取ることができる。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 blank$ という名前で始まるメソッドが存在しないという仮定を設けている。そのため、ロード時に追加するメソッドは基本的に図 3 のようになる。

しかし、メソッドにはオーバーライドされる可能性があるため、一概に図 3 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 3 のように準備するとオーバーライドされた場合に正しい振る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 4 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。

例えば、先程の図 1 のようなクラスがあるとする。今クラス Data に getX を blank$0 を利用して追加するとする。このとき、クラス Shape, Circle, Rect でも有効である。そのような場合、各クラスのオブジェクトに対して blank$0 が呼ばれるようになるが、その場合にも Data クラスに追加したメソッドに対応する blank$0 が呼ばれるように図 4 のようにする。

```
Object blank$0(Object[] objects){
  return null;
}
Object blank$1(Object[] objects){
  return null;
}
…
```

図 3. 継承関係がない

```
Object blank$0(Object[] objects){
  return super.blank$0(objects);
}
Object blank$1(Object[] objects){
  return super.blank$1(objects);
}
…
```

図 4. 継承関係がある

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、返り値の型とコンストラクタ名が決まっている。従って、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 Blank$ 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た。返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 の配列として渡されるようにし、先程のダミーのクラスを引数として受け取る。以上よりロード時に追加する空のコンストラクタは以下のようなものになる。

```
/* classname */(Object[] objects, Blank$0 d){

}
/* classname */(Object[] objects, Blank$1 d){

}
...
```

3.2 追加の実現

ユーザーからメソッドまたはコンストラクタの追加の指示が行われたときにはロード時に準備しておいたメンバーの中身を追加を指示されたメソッドのものと置き換える。これによりメンバーを直接追加する場合と同等の振る舞いを実現する。このときシステムは準備しておいたメンバーと追加するメンバーの対応関係を記憶しておく。また追加したメソッドを呼び出している箇所は、対応するロード時に準備しておいたメソッドを呼び出すように変更する。あらかじめ準備しておいたメソッドは引数や戻り値の方が追加するメソッドと一致しないことがあるので、その差異を吸収するコードを呼び出し側に挿入する。

追加されるメンバーがフィールドの場合は何もしない。あらかじめ準備しておいたフィールドを流用して追加するフィールドに保存すべき値を保存する。ただし準備しておいたフィールドと実際に追加したいフィールドは型が異なる可能性があるので、その差異を吸収するコードはフィールドにアクセスする側のコードに挿入する。

3.2.1 メソッドボディのコピー

メソッドを追加するときは、そのメソッドの中身をあらかじめ準備しておいたメソッドの中にコピーする。この際に、引数や戻り値の型を、実際に追加したいメソッドのものに合わせるためのコードも、準備しておいたメソッドの中に一緒にコピーする。これは標準の HotSwap の制約で引数や戻り値の型を実行中に変更できないからである。

例として図 1 の Shape に void setX(int x); というメソッドを追加する場合について述べる。先ほど準備した Object blank$0(Object[] objects); メソッドを利用して追加する。単純にコピーを行った場合は blank$0 のメソッドボディが setX のメソッドボディに変わるだけである。

```
void setX(int x){
  /* setX のメソッドボディ */
}
Object blank$0(Object[] objects){
  /* 実行時に型を変換するコード */
  /* setX のメソッドボディのコピー */
}
```

3.2.2 引数のキャスト

setX のメソッドボディは第一引数が int であるいう前提のコードになっているため、実行時にパラメータとして受け取った Object の配列の各要素を型変換し、それぞれ本来のパラメータに代入しなおすというバイトコードを blank$0 の先頭に挿入する。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必要なバイトコードが増える。同様に戻り値の型も変換するが、プリミティブ型の場合は同様にボクシングを行うためのコードを挿入する。

挿入するバイトコードは図 5 のコードで生成する。allParameterCast は Object の配列として受け取った実引数列を本来の型の引数列に変換するバイトコードを生成するメソッドである。引数は空のバイトコードと、実引数の型の配列である。Object の配列から実引数を一つ取り出し、型変換して本来の仮引数に代入するバイトコードを生成するのは parameterCast の役目である。バイトコードレベルでは、仮引数は特別な局所変数である。

Object の配列は第一引数として blank$0 に渡されるので、allParameterCast はまず setX の第二引数以降があればそれを型変換して本来の仮引数に代入するコードを生成する。その後、Object の配列から setX の第一引数を取り出して本来の仮引数に代入するバイトコードを生成し、生成済みのバイトコードの末尾に追加する。第一引数に対応するバイトコードレベルの局所変数には、元々 Object の配列が格納されているので第一引数の型変換は最後に実行しなければならない。実行後は Object の配列はそこから取り出した第一引数の値で上書きされてしまう。

parameterCast はまず Object の配列から指定された要素の Object を取り出す。次に変換後の型がプリミティブ型かどうかを確認して挿入するバイトコードを切り替える。プリミティブ型の場合は一度 Object 型の値をラッパークラスの型へ型変換し、プリミティブ型の値を取り出す。プリミティブ型でない場合は Object 型の値を単に目的の型へ型変換するだけである。最後に型変換後のオブジェクトを目的の仮引数に対応するバイトコードレベルの局所変数に代入する。parameterCast はこのような処理を行うバイトコードを生成する。

```
public Bytecode allParameterCast(Bytecode bytecode, CtClass[]
    parameterTypes) {
  for (int i = 1; i < parameterTypes.length; i++) {
    bytecode = parameterCast(bytecode, i, parameterTypes[i]);
  }
  if (parameterTypes.length > 0) {
    bytecode = parameterCast(bytecode, 0, parameterTypes[0]);
  }
  return bytecode;
}

private Bytecode parameterCast(Bytecode bytecode, int i, CtClass
    parameterType) {
  bytecode.addAload(1);
  bytecode.addIconst(i);
  bytecode.add(Bytecode.AALOAD);
  if (parameterType.isPrimitive()) {
    CtPrimitiveType pt = (CtPrimitiveType) parameterType;
    String wrapperName = pt.getWrapperName();
    bytecode.addCheckcast(wrapperName);
    bytecode.addInvokevirtual(wrapperName, pt.getGetMethodName(),
        pt.getGetMethodDescriptor());
  } else {
    bytecode.addCheckcast(parameterType);
  }
  bytecode.addStore(i + 1, parameterType);
  return bytecode;
}
```

図 5. 引数のキャストを行うバイトコードを生成する Javassist を用いたコード

3.2.3 戻り値のキャスト

同様に戻り値に関しても型を変換するバイトコードを生成し、挿入する。戻り値ではプリミティブ型の時のみ変換を行い、プリミティブ型の値をラッパークラスに変換するバイトコードを生成する。ラッパークラスを作るコードをシステムが用意しているので、プリミティブ型の戻り値を引数とし、それらのメソッドを呼び出す。

例えば、long の場合は図 6 のようなコードでプリミティブ型の値をラッパークラスに変換するバイトコードを生成する。Wrapper クラスの make メソッドを実行時に呼び出すバイトコードを実行してプリミティブ型の値をラッパークラスに変換する。returnCast メソッドでは return を行っているバイトコードを見つけ出し、メソッド呼び出しを挿入するために隙間を作り、変換を行うためのメソッド呼び出しを挿入し、その結果を返すバイトコードを追加している。

```
package hayafune.blank.agent;
public class Wrapper {
  public static Object make(long l) {
    return new Long(l);
}}

public class Redefine{
public void returnCast(CtMethod addMethod, CtMethod declaredMethod)
                        throws NotFoundException, BadBytecode {
  MethodInfo methodInfo = declaredMethod.getMethodInfo();
  ConstPool cp = methodInfo.getConstPool();
  CtClass returnType = addMethod.getReturnType();
  if (returnType.isPrimitive()) {
    CodeIterator ci = methodInfo.getCodeAttribute().iterator();
    CtPrimitiveType pt = (CtPrimitiveType) returnType;
    while (ci.hasNext()) {
      int pos = ci.next();
      int c = ci.byteAt(pos);
      if (c == Opcode.LRETURN) {
        ci.writeByte(Opcode.NOP, pos);
        ci.insertGap(pos, 3);
        ci.writeByte(Opcode.INVOKESTATIC, pos);
        ci.write16bit(cp.addMethodrefInfo(cp
          .addClassInfo("hayafune.blank.agent.Wrapper"),
          "make", "(J)Ljava/lang/Object;"), pos + 1);
        ci.writeByte(Opcode.ARETURN, pos + 3);
}}}}}
```

図 6. 返り値 (long) のキャストを行うバイトコードを生成する Javassist を用いたコード

3.2.4 呼び出し側の変換

次に、新しく追加したメソッドを呼んでいる場所では、代わりに blank$0 のようなメンバーを呼び出すように変換を行う。追加するメンバーがメソッドである場合、動的ディスパッチ及びメソッド・オーバーライドを考慮して、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全て変換する。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。例えば図 1 の Shape

に setX を追加すると Main のメソッドが追加した Shape の setX メソッドを呼ぶ可能性があるので、その setX メソッドを呼び出す Main のメソッドはそれを考慮して変換する必要がある。この場合の変換方法は 3.2.5 節で述べる。

メソッド・オーバーライドを考慮しない場合のコード変換は次のようになる。まず、メンバーの呼び出し側は Javassist が提供する ExprEditor を用いて置き換える。Javassist のコンパイラでは $ から始まるいくつかの特別な意味を持つ識別子が用意されている。$_ が戻り値、$r が戻り値の型、$0 が実際の引数の 0 番目、$$ がすべての引数、$args が Object の配列に変換した引数を表している。これらの識別子を利用し、呼び出し側に実行時に型を変換するコードを追加する。フィールドのアクセスは読み出しと書き込みがあり、以下のコードを用いて変換する。

```
$_ = ($r) $0.blank$0
$0.blank$0 = $1
```

読み出しは上の、書き込みは下の文を Javassist を使ってコンパイルし、目的となる箇所に得られたバイトコードを挿入する。読み出し用に得られるバイトコードは、例の場合、対象となるオブジェクトの blank$0 フィールドを読み、得られた Object 型の値を追加されたフィールドの本来の型に変換して、元のフィールドの読み出しの結果の値とする。この値は、後続のバイトコードに渡される。書き込み用に得られるバイトコードは、例の場合、書き込む値を Object 型の値に変換し、対象となるオブジェクトの blank$0 フィールドに書き込む。

一方、メソッド呼び出しは基本的に以下のコードを用いて変換する。

```
$_ = ($r) $0.blank$0($args)
```

この文を Javassist でコンパイルして得られるバイトコードは、元々のメソッド呼び出しの引数全体を Object 型の配列に変換し、それを引数として blank$0 メソッドを呼び出す。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。変換後の値は元々呼び出されていたメソッドの戻り値であるかのように、続くバイトコードに渡される。

最後にコンストラクタの呼び出しは以下のコードを用いて変換する。/* classname */ とある箇所は、実際に作成するオブジェクトのクラス名で置き換えてから Javassist に渡してコンパイルする。コンストラクタには元の実引数列の他、Blank$0 型の null 値が引数として渡される。

```
$_ = new /* classname */($$, (Blank$0) null)
```

3.2.5　オーバーライドに対する制御

メソッドはオーバーライドのため、メソッド呼び出しの振る舞いは呼び出し対象のクラスやそのサブクラスにメソッドの定義があるかないかによって変更される。そのため、新たにメソッドを追加した場合は、呼び出し側のコードもそれを考慮した変換を行わなければならない。

例えば、図 1 のようにクラスが定義されているとする。ここで 図 7 のように setX メソッドと同じシグネチャのメソッドを Shape に追加する。この場合は元から存在しているコードからも新たに追加するメソッドの呼び出しが行われる。例えば、図 2 の Main クラスはユーザーによって変更を加えていないが、Shape クラスの親クラスである Data クラスの setX メソッドを set メソッドの中で呼んでいる。よって、ユーザーが Shape クラスに新しいメソッドを追加した場合、システムは Main クラスのメソッドもコード変換する。

図 8 に set メソッドの定義を示す。引数で Shape クラスのインスタンスを受け取る可能性がある。Shape クラスには setX メソッドが追加され、Data クラスのメソッドを上書きするが、実装上追加されるのは blank$0 のようなメソッドである。そのままでは Data クラスの setX メソッドは上書きされない。そこで setX を呼び出す側のコードを変換し、呼び出し対象オブジェクトのクラス

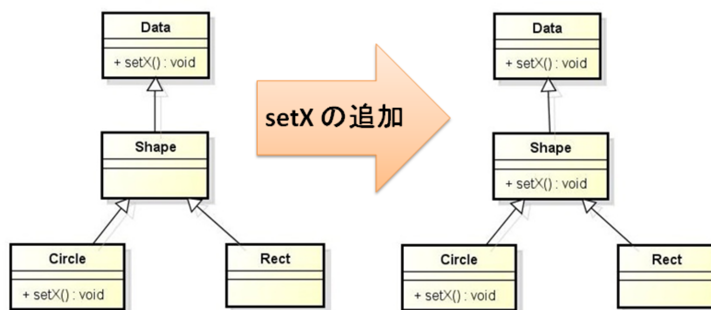

図 7. 継承関係による呼び出すメソッドの違い

を実行時に検査し、Shape オブジェクトである場合には setX ではなく blank$0 を呼ぶようにする。変換例を図 9 に示す。継承関係を考慮して、Shape クラスに追加された setX (実装上は blank$0) を継承しているクラスの場合も blank$0 が呼ばれるようなコードになっている。

```
public static void set(Data d){
  d.setX(1);
}
```

図 8. 型の確認なし

```
public static void set(Data d){
  if (d instanceof Circle){
    d.setX(1);
  } else if (d instanceof Shape){
    d.blank$0(new Object[]{1});
  } else {
    d.setX(1);
}
```

図 9. 型の確認あり

3.3 追加のタイミング

これまで説明したように、本システムは既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ーが追加を指示したタイミングによって決まるため、一貫性が取れない可能性がある。標準の HotSwap の仕様により、既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既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 オーバーヘッドの計測

本システムの手法では準備や呼び出しのオーバーヘッドが大きいと考えられる。そこで、我々はマイクロベンチマークを作成し、オーバーヘッドを計測した。動作環境は、CPU は Intel Core2 Quad 2.67GHz, メモリは 4GB, OS は windows7 である。

4.1 空のメンバーの準備

本システムでオーバーヘッドが大きいと予想されるロード時におけるメンバーの準備のオーバーヘッドを計測する。3.1 節で説明したシステムを使い、ロード時にメンバーを追加する準備だけを行い、ロード後になにもせずに終了するプログラムに実行し、クラスをロードするために必要な時間を計測した。計測は 500 回行い、ロードにかかる時間の平均の値を図 10 にまとめた。横軸は本システムを利用しない場合、追加できるフィールド、メソッド、コンストラクタの数がそれぞれ最大 0, 10, 20, 30, 40, 50 個であるように準備する場合を表す。追加可能なメンバーの数が増えれば増えるほどロードにかかる時間は大きくなっている。本システムを利用しない場合と追加可能なメンバーの数が最大 0 個の場合とではクラスに変更を与えないという意味では同じである。しかし、後者では事前にメンバーを追加する準備を全くしなくても HotSwap の機能を利用する、オーバーヘッドが存在する。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ーヘッドであり、図 10 より約 17 %かかっている。また、空のメンバーを一つ準備するのに約 0.47 秒かかっている。

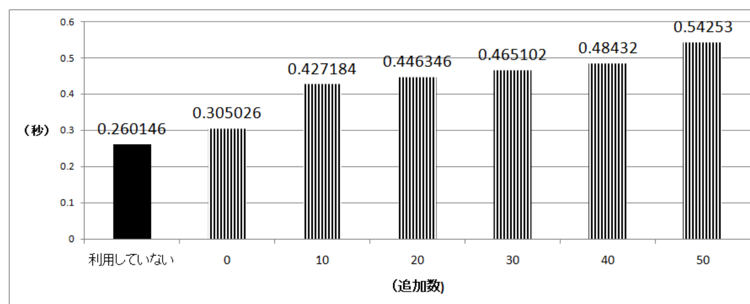

図 10. 追加可能なメンバーの最大数とロード時間

さらにメンバーを準備するオーバーヘッドを計測するため、非常に多数のクラスを準備する。各クラスはメンバーを持たず、図 11 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通り実行した場合とロード時にメンバーを追加する準備をした場合の実行時間を比較する。このとき事前に準備のために追加する空のメンバーの数は 5 個である。ウォーミングアップとしてまず 1000 個の異なるクラスをロードした。その後、m 個の異なるクラスをロードする時にかかる時間を計測した。結果は図 12 である。通常通りの場合、クラス 1 個のロード時間は平均 0.26ms である。ロード時にメンバーを追加する準備をしてロードする場合は平均 1.33ms である。4 倍から 5 倍程のオーバーヘッドがかかっている。

4.2 オーバーライドに対する制御のオーバーヘッド

振る舞いが正しくなるように制御するために本システムは実行時に呼び出し先のオブジェクトの型の確認を行う。型を確認するためのコードに伴う実行時のオーバーヘッドは大きいと予想される。この影響を確認する実験を行った。

まず実際に本システムを使ってメソッドを追加した場合に、最終的に得られるコードと同等のコードを用意した。次に同等のメソッドの追加を手でソースコードを変更して、最も効率がよくなるようにしたコードを用意し、前者と実行時間を比較した。後者のコードは 3.2.5 節で述べたオーバーライドに対する制御を行うコードを含まない。用意したコードは図 8, 9 と同様である。

実験の結果をまとめたものが図 13 である。型の確認を行わない場合では一回のメソッドの呼び出しが平均 5.262 ナノ秒であり、型の確認を行う場合では一回のメソッドの呼び出しが平均 10.04 ナノ秒であった。オーバーライドに対する制御によって約 2 倍のオーバーヘッドがかかっている。

```
class Main {
 public static void main(String[] args) {
   // warmup
   // n回ロードを行う
   new A0();
   new A1();
   …
   new An−1();

   // m回ロードを行う
   long start = System.currentTimeMillis();
   new An();
   new An+1();
   …
   new An+m−1();
   long end = System.currentTimeMillis();
   System.out.println(end − start);
 }
}
```

図 11. クラスをロードするマイクロベンチマーク

5 関連研究

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。動的言語に備わっている機構ではクラス定義を変更だけでなく、より多くの種類の変更を許す [3, 4, 9]。例えば、Ruby では open class という機能が存在している。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することができる機能である。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。Smalltalk には classbox [3] という機能が提案されている。Classbox はパッケージのように扱うことができ、クラスを import して、import した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 に存在する問題を解消している。

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、動的に振る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出来る点は、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 しか提供していない [12, 13, 14]。本システムを DAOP の実装基盤として利用すると AspectJ[10] で提案されているインタータイプ宣言に対応した DAOP が実現できると考えられる

5.3 Envelope-Based weaving

事前にメンバーを用意するという我々の提案手法の考えは、Envelope-Based weaving [6, 5] 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ーをラップするメンバーを追加する。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

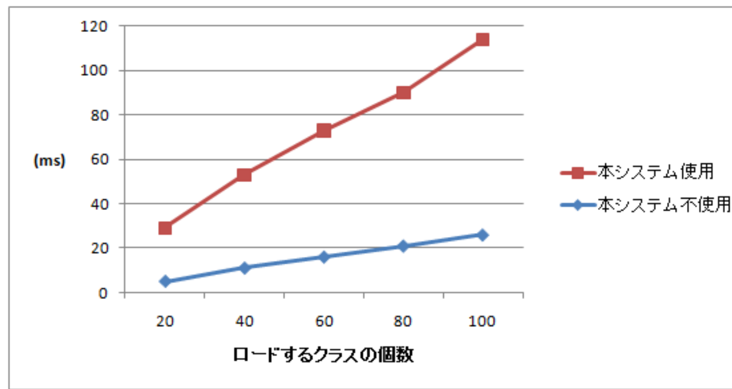

図 12. クラスのロード時間

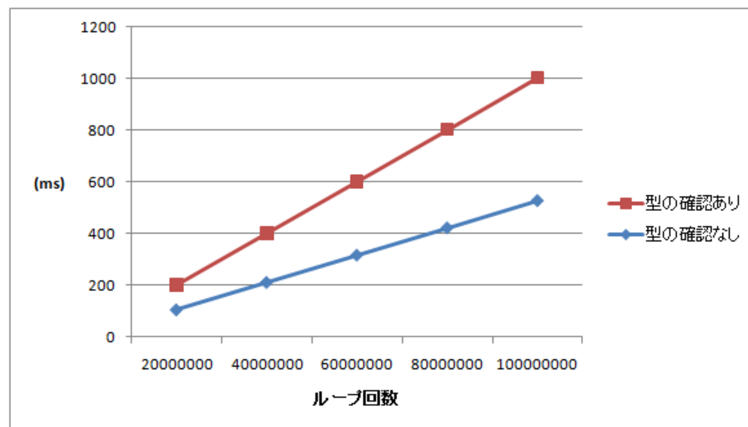

図 13. オーバーライドを制御するための実行時型検査のオーバーヘッド

ル化されたアスペクトを高速に織り込む機構を提案している。アスペクトを織り込む時にこの追加されたメンバーを利用することで、Steamloom [7] による織り込みで必要だったオフセットの計算を行う必要がなくなり、バイトコードの操作が容易になっている。動的な変更にかかる計算のオーバーヘッドとコンパイル時またはロード時にかかる準備のオーバーヘッドとの比較で、事前に準備しておく方が優れていると主張している。Steamloom は DAOP システムの一つであり、Jikes Research Virtual Machine (RVM) という Java 仮想機械を改造して実現している。我々の提案手法は同様に事前に空のメンバーを用意しておくが、それを新規メンバーの追加のために使うことが異なる。新規メンバーのために様々な型に対応できなければならない。

6　まとめ・今後の課題

6.1　まとめ

本論文において我々は、標準 Java 仮想機械上で動的にメンバーの追加を行う機構を提案した。ロード時に各クラスへ事前に空のメンバーをいくつか追加することで、実際に動的な追加を行わないように準備する。ユーザーがメンバーの追加の操作を行ったときに、用意しておいたメンバーを追加するメソッドの内容に書き換え、さらにそのメンバーと呼び出し側に動的に型を確認するコードを加える。これらによって動的なメンバーの追加を実現した。

6.2　今後の課題

本論文で提案した機構にはまだ未実装な部分が存在しているため、これを実装することが今後の課題の一つである。

また、提案した機構の有用性をより高めるためにはいくつかの拡張が必要である。本システムでは事前に用意しておくメンバーの数に制限があり、無制限に追加を行えない。無制限の追加に対応するためには、事前に追加しておいた空のメンバーで複数のメンバーの追加に対応する必要がある。これには一つのメソッド内に追加した複数のメソッドの内容を保存しておき、冒頭でどのメソッド呼び出しであるかを判定し、実行するような実装が考えられる。しかしこの方法はより多くのオーバーヘッドがかかることが予想される。

現状では、変更が反映されるタイミングに一貫性がないため動的な変更が想定するように反映されない可能性がある。標準の HotSwap も同様な問題を抱えているので、それを解決すればより有用性を高められる。

さらに有用性を高めるためにはメンバーの追加を可能にするだけでなく、メンバーの削除やクラスの階層構造の追加や削除を行えるようにしたい。また private などの修飾子に対応することも今後の課題である。なお throws 宣言は実行時に無視されるため現状でも問題ない。

参考文献

[1] The AspectJ project. `http://www.eclipse.org/aspectj/`.

[2] $Java^{(TM)}$ java.lang.instrument package. `http://download.oracle.com/javase/6/docs/technotes/guides/instrumentation/index.html`.

[3] Alexandre Bergel, Re Bergel, Stephane Ducasse, and Roel Wuyts. The classbox module system, 2003.

[4] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: controlling the scope of change in Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pp. 177–189, New York, NY, USA, 2005. ACM.

[5] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 109–124, New York, NY, USA, 2006. ACM.

[6] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based weaving for faster aspect compilers. In *NODe/GSEM*, pp. 3–18, 2005.

[7] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 83–92, New York, NY, USA, 2004. ACM.

[8] Shigeru Chiba. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[9] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pp. 130–145, New York, NY, USA, 2000. ACM.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 327–353, London, UK, 2001. Springer-Verlag.

[11] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. ECOOP 2000—Object-Oriented Programming, pp. 337–361, 2000.

[12] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pp. 233–246, New York, NY, USA, 2008. ACM.

[13] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pp. 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[14] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced runtime adaptation for java. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pp. 85–94, New York, NY, USA, 2009. ACM.

[15] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pp. 10–19, New York, NY, USA, 2010. ACM.