Josh：バイトコードレベルでの Java 用 Aspect Weaver

中川 清志* 立堀 道昭† 千葉 滋‡

要 旨

アスペクト指向とは，ログ出力やメモリ管理などプログラム全体に散らばってしまう処理を，アスペクトという新しいモジュール概念によりモジュール化する技法である．アスペクトはオブジェクトとは独立に記述されており，それらは weaver と呼ばれる言語処理系により 1 つのプログラムに合成される．AspectJ は標準的な汎用アスペクト指向言語の一つであり，Java を拡張した言語である．AspectJ は Java ソースファイルと，複数のファイルに散らばる処理を記述したアスペクトを weave することにより，両方の機能を持つソースファイルを作り出す．しかし AspectJ はソースコードレベルで weave をおこなっており，これによる欠点は少なくない．本稿で提案する新コンパイラ Josh はこの欠点に対処したものであり，バイトコードレベルで weave を実行する．また構造リフレクションを提供する Javassist により実装したことで，weave の各操作がリフレクションの機能の組合わせで実現できることを示す．

1 はじめに

オブジェクト指向ではデータに観点をおいて全体をオブジェクトに分割し，モジュール化していく．しかし同様の処理が複数のクラスに散らばってしまい，プログラムの保守性，再利用性が下がってしまうことがある．これに対処するためにこのような散らばりがちな処理をモジュールとして扱えるようにしたのがアスペクト指向の考え方である．アスペクトはオブジェクトとは独立に記述されており，それらは weaver と呼ばれる言語処理系により 1 つのファイルに合成される．AspectJ[3, 4] は標準的な汎用アスペクト指向言語の 1 つであり，Java にアスペクト指向を取り入れて拡張したものである．AspectJ コンパイラは独自アスペクトと Java ソースコードをソースコードレベルで weave し，その後に合成済の Java ソースコードをコンパイルしている．この欠点として，ソースコードが必要であるうえに，アスペクト変更による再コンパイルの必要があることがあげられる．

本稿は AspectJ 言語で記述されたアスペクトをバイトコードレベルで weave するコンパイラ Josh を提案する．Josh ではロード時にアスペクトと Java クラスの weave を行う．これによりロード時にオンデマンドで weave が行われることになり，AspectJ コンパイラによる weave で問題となっていた，アスペクト変更による Java ソースの再コンパイルの必要がなくなる．また Josh は，通常のクラスローダの代わりに独自のクラスローダを使うことにより，ロード時の weave を可能にするが，直接バイトコードを操作して weave するので，ロード時のオーバヘッドはあまり大きくならないことが予想できる．

Josh ではバイトコード操作を Javassist[1] を用いて行っている．Javassist は Java に構造リフレクションを提供するライブラリであり，クラス構造の内観だけでなく変更も可能にする．本稿では Javassist を使い実装することにより，weave の各操作はリフレクションの機能の組合わせで実現できることを合わせて示す．

以下ではまず次章で，AspectJ の説明とその欠点の指摘をし，3 章で Josh の設計と実験による評価を述べる．4 章では関連する研究と比較をし，5 章で本稿をまとめる．

2 AspectJ とそのコンパイラによる実装の問題点

この章では，アスペクト指向言語について説明し，AspectJ のコンパイラによる実装（AspectJ 1.0）の欠点を指摘する．

*東京工業大学理学部情報科学科
Tokyo Institute of Technology, Faculty of Science
†筑波大学大学院工学研究科
Doctral Program in Engineering, University of Tsukuba
‡東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

2.1 アスペクト指向言語

現在プログラミング言語には関数型，手続き型，オブジェクト指向など様々な種類が存在している．これらの違いは，それぞれが採っているモジュールの違いとも考えられる．モジュールとはある意味を持ったコードのまとまりであり，それはまた人間にとって理解しやすい大きさに分割されたプログラムの単位である．そして我々は分割されたモジュールを組立てることにより，全体のプログラムを構成していく．つまりプログラミング言語においてモジュールというのは重要な概念であるといえる．そして上手にモジュール化されたプログラムは保守性，理解性，変更容易性などに優れている．

オブジェクト指向はデータに観点をおいて個々のモジュールに分割する考え方であり，そのモジュールをオブジェクトと呼ぶ．個々のオブジェクトは通常はなんらかの機能を持つように分割される．そしてオブジェクト指向プログラミングはそれぞれのオブジェクトに求められる要件を，機能モジュールとしてカプセル化する目的に優れている．そのためオブジェクト指向プログラミング言語は現在広く普及している．しかしながら，一般にある種の処理は，どうしても複数のオブジェクトに散らばってしまう．つまり本来オブジェクトに持たせた機能とは関係のない処理までもが個々のオブジェクト内に分散されてしまう．その結果そういった処理の変更の度にたくさんのソースファイルを修正する必要がでてしまう．特にシステマティックな要素を扱う処理は複数のオブジェクトに散らばることが多く，オブジェクトだけでは十分きれいにモジュール化できない．

アスペクト指向はこの問題に対応したプログラミング技法である．アスペクト指向言語では複数のオブジェクトに散らばる処理をアスペクトとしてモジュール化する．これにより全体に散らばってしまう処理を集約して扱う．

このようにして別個の意味合いでモジュール化されているオブジェクトとアスペクトを，両方の機能を持つモジュールに合成することを **weave** するという．別の言い方をするならば，アスペクトによるオブジェクトの機能の変換と言えよう．この合成処理によりアスペクトとしてモジュール化されていたものを，複数のオブジェクトに同時に埋め込むことができる．つまりプログラムコード全体に散らばってしまう処理を一括管理することができ，保守性や再利用性を高めることができる．

AspectJ は標準的な汎用アスペクト指向言語の 1 つであり，Java にアスペクト指向を取り入れて拡張したものである．AspectJ 1.0 で提供される専用コンパイラは，Java のソースコードを AspectJ 言語で記述されたアスペクトのソースコードに基づいて変換し，通常の Java のソースコードを生成する．生成された Java のプログラムは通常の Java コンパイラでバイトコードに変換されるため，標準の Java 実行環境で動作させることができる．

2.2 AspectJ 言語のアスペクト記述

ここでは AspectJ 言語について説明する．本稿では，AspectJ 言語と AspectJ コンパイラという 2 種類の言い方で前者を後者から区別する．前者は AspectJ におけるアスペクトの記述方法を表す．これは，アスペクト構文と記述されたアスペクトがどのようにな意味を持つかを定義するが，それが具体的にどのように元のソースコードに組み込まれるかは定義されない．アスペクトをどう組み込むかは実装依存である．これに対して後者は，Java ソースコードにアスペクトの組み込みをおこなう処理系である．AspectJ 言語は後者からは独立しており，後者はその 1 つの実装を与えるに過ぎない．以下では AspectJ 言語が与えているアスペクトの 2 つの要素の説明をする．

Introduction

AspectJ 言語において，既存のクラスに新たな要素を追加することを **introduction** という．追加できる要素には，フィールド，メソッド，コンストラクタがある．また既存クラスのスーパクラスの変更とインタフェースの追加ができ，これも introduction に含まれる．

以下に具体例を示す．

```
(例 1)  private int Point.newfield;
(例 2)  public void Point.setX(int newx)
            { this.x = newx; }
```

上記のような AspectJ 言語アスペクトの一部があったとする．(例 1) は Point クラスに private で int 型の newfield というフィールドを追加する，(例 2) は Point クラスに public で void 型の setX(int newx)

というメソッドを追加する，という意味のアスペクトである．

Advice

プログラムの実行途中で「何かが起こったとき」を **join point** という．例えばそれはメソッド呼び出し，メソッド実行，インスタンス生成，コンストラクタ実行，フィールド参照，例外ハンドラ実行などがある．プログラムはこの「何かが起こったとき」に「何かが実行される」という連鎖により動いている．つまりある join point には，ある動作が付随しているといえる．これらの join point の集合体を捉えるプログラミング上の要素が **pointcut** である．pointcut の例は

```
(例 3)  call(int Point.getX())
(例 4)  set(int Point.x)
```

などがある．(例 3) は Point クラス内の，int 型の返り値を持ち引数を持たない getX メソッドの呼び出しを捉える．(例 4) は Point クラス内の，int 型の x というフィールドへの値の書き込みを捉える．予約語である *pointcut* を使えば，pointcut に名前をつけておくことができる．それには以下のようにする．

```
(例 5)  pointcut mysetX() :
            set(int Point.x);
```

こうすることにより mysetX という pointcut が定義され，同じシグネチャの pointcut の再利用ができる．

Advice とは，ある pointcut に対し新しい命令を実行することである．Advice は新しい命令を実行するタイミングにより 3 種類に分類される．pointcut に付随する動作の前に実行する **before**，後に実行する **after**，若しくは付随する動作の代わりに実行する **around**，である．Advice は，

> *advice* の種類： *pointcut* の指定 { 新実行コード }

という並びで形成されている．ここでごく簡単な advice の例を示す．

```
(例 6)  before() : call(int Point.getX())
          { System.out.println
                ("after getX"); }
(例 7)  after() : mysetX()
          {System.out.println("set x");}
```

(例 6) は Point クラス内の int 型の返り値である getX() というメソッド呼び出しを捉え，その pointcut の動作の前に新実行コードを実行する．(例 7) は mysetX という pointcut を捉え，その pointcut の動作の後に新実行コードを実行する．mysetX という pointcut は前もって定義しておいたものである．

2.3 AspectJ コンパイラによる実装の問題点

AspectJ コンパイラはアスペクトと Java ソースの weave をソースコードレベルで行っている．そのためソースコードが必要であるという制約があり，ソースコード無しのサードパーティ製クラスに適用できない．

そのうえ AspectJ コンパイラは，weave のあとに weave 済のソースコードのコンパイルをするという順番をとっている (図 1)．しかしこのように weave 後にコンパイルをする手法では，アスペクトの変更の度に全体を再コンパイルしなければならない．アスペクトの有効化・無効化にも手間がかかる．同様にどれか一つでもファイルを変更した場合にも再コンパイルしなければならない．これにより開発効率が落ちてしまう．また weave 時には最終的なアプリケーションではどのクラスを使うかが完全にはわからないので，結果的に使われないクラスにも weave をしなければならないという無駄がでてしまう．

3 Josh

AspectJ コンパイラによる実装では 2 章でふれたような様々な問題があった．そこで我々はその問題に対処した新コンパイラ Josh を提案・開発した．

3.1 バイトコードレベルでのロード時 weave

Josh ではロード時にアスペクトと Java クラスの weave を行う．これによりアスペクトの変更による Java ファイルの再コンパイルの手間を省くことができる．また AspectJ コンパイラとは違いロード時に weave することにより，オンデマンドで weave するので無駄を省くことができる．Josh は通常のクラスローダの代わりに独自のクラスローダを使うこと

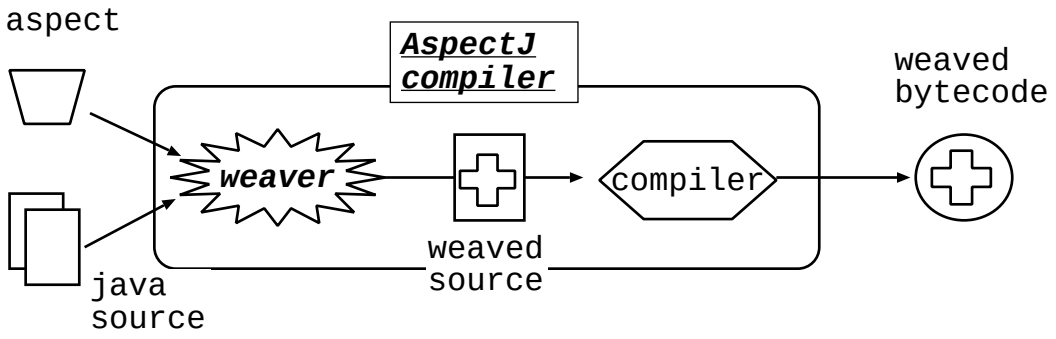

図 1: AspectJ コンパイラの流れ

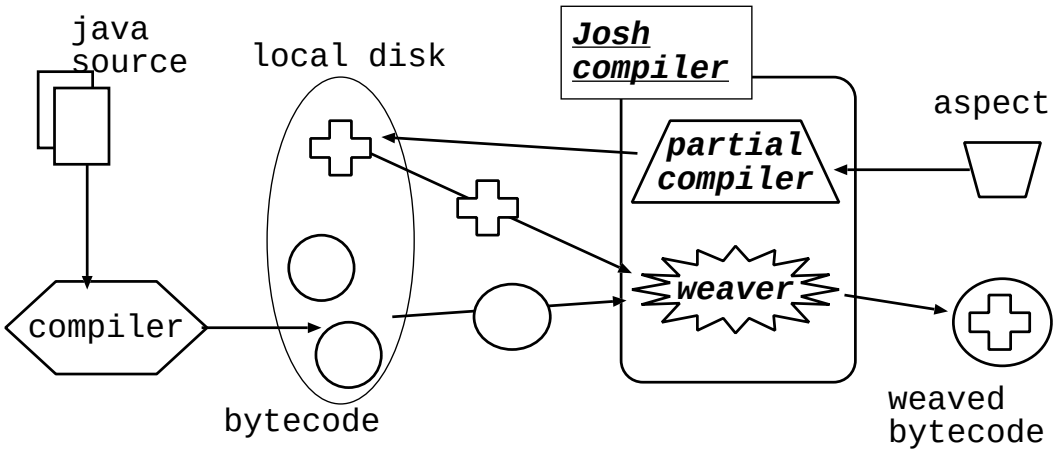

図 2: Josh の流れ

により，ロード時の weave を可能にした．

またバイトコードを操作して weave をするのでソースコードが必要ない．これによりサードパーティから提供されたソース無しのクラスにも適用できる．さらに jar アーカイブ化されたクラスやネットワークからダウンロードしてきたクラスへの weave も可能である．

この Josh の利点によりソースコードとアスペクトを別の人間が記述するという場合もありうる．アスペクトを意識しないで記述されたプログラムに対しアスペクトを記述することは可能である．例えば Addistant[10] は，分散環境を意識しないで記述されているプログラムを分散化することができる処理系である．分散化のためのアスペクトを既存プログラムとは分離して記述し，weave をして分散プログラムを生成する．この際に元のプログラムのソースコードは不要である．他の例としてはセキュリティ機構の追加も考えられる．セキュリティをあまり考慮せずにつくられたプログラムに，安全に実行するための機構をアスペクトで追加できれば有用であろう．

これとは反対にバイトコードレベルで weave することによる限界もある．例えばメソッドのインライン展開には対応できない．アスペクトにはメソッド呼び出しを捉えるものがあったが，インライン展開によりメソッドを呼んでいるという情報自体を失ってしまうからである．またセキュリティ・セーフティの問題がある．ソースコードレベルでの weave では，コンパイル時にあらかじめエラーを検知することができる．しかしバイトコードレベルでは実行時エラーとなってしまう．同じシグネチャのメソッドの追加などはこの種の問題を引き起こし，Josh はまだ対応できていない．まとめるとソースコードのないプログラムのアスペクトを書けることは有意義だが，基本的にはソースコードのあるプログラムのアスペクトを書くのが望ましい．

Josh は部分コンパイラと weaver で構成されている．部分コンパイラは既存外部コンパイラを使ってアスペクト内の Java ソースコード断片をバイトコードの断片に変換する．weaver は Javassist を使って，このバイトコード断片を適切な join point に挿入する．この 2 つの機能の組合わせによりバイトコードレベルでの weave を可能にした．

3.2 部分コンパイラ

Josh のアスペクトはソースコードレベルなので，直接バイトコードとして扱うことができない．つまり Javassist を使って weave するには，アスペクト内の実行コードをバイトコードに変換する必要がある．この節では，どのようにしてこの問題に対処していくかを述べる．

アスペクト内で実際にバイトコード化する必要があるのは，

1．メソッド及びコンストラクタの introduction

```
(例 A)  public int
            Point.newPlus(int a, int b)
                { return a + b; }
```

2．全ての advice

```
(例 B)  before():call(int Point.getX())
            {System.out.println("before");}
```

である．これらのソースコードの断片を，バイトコードに変換しなければならない．

そこで Josh では外部の標準的な Java コンパイラによりバイトコード化する手法をとった．その手法はダミークラスを用意して，必要なソースコードを書き込んでコンパイルするというものである．コード断片だけではコンパイルできないので，それをダミークラスのメソッドとし，さらに通常コンパイルに必要なコードを加える必要がある．他選択肢として専用の外部部分コンパイラを使う手段もあるが，Josh が他のツールに依存してしまい Josh のポータビリティが損なわれてしまうので今回の手段を取った．

具体的に，(例 A) の introduction の必要部分をバイトコード化する方法を以下で示していく．バイトコード化したいメソッド部分をダミークラスに書き込むと，そのときのダミークラスの中身は以下のようになる．

```
public class DummyJosh {
    public int newPlus(int a, int b){
        return a + b;
    }
}
```

これをコンパイルすれば，(例 A) を表すメソッドのバイトコードを手にいれることができる．

しかしこのままだと，メソッド内からの参照を解決できない場合がある．例えば，

```
(例 C)  public void Point.newPrintX()
            { System.out.println
                  ("x = " + x); }
```

という introduciton であったとする．これを上記と同様にダミークラスに書き込むと，

```
public class DummyJosh {
    public void newPrintX() {
        System.out.println("x = " + x);
    }
}
```

のようになる．しかしこのダミークラス内には x というフィールドは存在しないのでコンパイルエラーになる．

この問題の解決方法は，メソッドを追加する先のクラス (この場合は Point クラス) のフィールド及びメソッドを，ダミークラスに全て書き込むことである．Point クラスはバイトコードでしか存在しない場合もあるが，リフレクションの機能を使えば，メソッドのシグネチャは簡単に取得できる．中身は空でいいので返り値に応じて適宜変えればよい．こうした対処をするとダミークラスは以下のようになる．

```
public class DummyJosh {
    public void newPrintX() {
       System.out.println("x = " + x);
    }
    public int x;
    public int y;
    public int getX() {
        return 0;
    }
    public void setX(int newx) {
        return;
    }
}
```

ただし Point クラスは，フィールド x,y を持ち，インスタンスメソッド getX(),setX(int newx) を持つとする．こうした処理をしてできたダミークラスをコンパイルすることにより，目的とするコード断片をバイトコード化できる．このダミークラスに対して Javassist が提供するリフレクションの機能を使えば，目的とするメソッドのバイトコードが手にはいる．Advice においても同様の処理をして実行コードをバイトコード化すればよい．

3.3 Josh weaver

Josh weaver は Javassist を使って実装されている．Javassist は構造リフレクションを提供するクラスライブラリである．Javassist を使ってアスペクトの weave をすることにより，weave の各操作はリフレクションの機能の組合わせで表現できることを示す．ここで 2 章の例を再掲載しておく．

(例 1) `private int Point.newfield;`

(例 6)
```
before() : call(int Point.getX())
        {System.out.println("before");}
```

このアスペクトを weave する方法を示す．

3.3.1 Introduction

Introduction とは既存のクラスに新しい要素を加えることであり，その概要は 2 章で説明した．Josh はこれと同じ処理を以下のようして実現している．Javassist を使用する第 1 段階は，クラスファイルを表す CtClass (compile-time Class) オブジェクトを生成することである．従って (例 1) の処理をするには，まず

```
ClassPool pool = ClassPool.getDefault();
CtClass ctpoint = pool.get("Point");
```

とする．ClassPool とは CtClass オブジェクトの入れ物であり，CtClass の生成の管理をする．CtClass が新しい要素を加えるためのメソッドを提供しているので，それにより introduction を実現する．
実際にフィールドを追加するためには，そのフィールドを表す javassist.CtField オブジェクトをつくらなければならない．CtField 及び CtMethod は，Java リフレクション API の Field と Method に相当する．具体的に (例 1) の introduction をするには以下のコードを実行する．

メソッド名	説明
void addConstructor(..)	コンストラクタを追加する
void addField(..)	フィールドを追加する
void addInterface(..)	インタフェースを追加する
void addMethod(..)	メソッドを追加する
void setSueprclass(..)	親クラスを変更する

表 1: Introduction 実現のための CtClass のメソッド

```
CtField ctfield =
     new CtField(CtClass.intType,
                 "newfield", ctpoint);
ctpoint.addField(ctfield);
```

ただし ctpoint は Point クラスを表す CtClass オブジェクトである．

3.3.2 Advice

Advice とはある pointcut に対し，新しい命令を実行することであることは既に 2 章で説明した．以下では Javassist を使えば advice と同様の処理がリフレクションの枠内で可能であることを示す．javassist.CodeConverter クラスは，指定された join point のバイトコードを変換する機能を提供している．これにより様々な join point に対し新しい命令を実行するようにでき，advice を実現できる．

メソッド名	説明
void insertBeforeMethod(..)	メソッド呼び出しの前に別メソッドを呼ぶ
void redirectMetodCall(..)	メソッド呼び出しを別メソッドに呼び代える
void replaceFieldRead(..)	フィールド読込みを別メソッドに呼び代える
void replaceNew(..)	インスタンス生成を別メソッドに呼び代える

表 2: CodeConverter クラスのメソッド

以下で (例 6) の実現方法を具体的に示す．ここでは insertBeforeMethod(CtMethod origin, CtMethod before) メソッドを使えばよい．これは origin メソッドの呼び出し前に，before メソッドを呼ぶようにバイトコードを変換する．このときに before メソッドの中身が {System.out.println(”before”);} というコードになっていれば求める実行結果になる．ただし両メソッドともに CtMethod オブジェクトで参照できなければならない．origin メソッドの方は Point クラス内にあるので，Javassist が提供するリフレクションの機能により手にはいる．しかし before メソッドの方は中身がソースコードで存在するだけで，どこのクラスにも属していない．この問題を解決するために，まず before メソッドを Josh 部分コンパイラによりバイトコード化する．次に before メソッドを Point クラスに追加する．そして再び Javassist が提供するリフレクションの機能を使うと，before メソッドを表す CtMethod オブジェクトが手にはいる．これらにより，(例 6) の advice 実現のためのバイトコード変換をする CodeConverter オブジェクトを得られる．実行コードは以下のようになる．ただし ctpoint は Point クラスを表す CtClass オブジェクトである．

```
CodeConverter converter =
    new CodeConverter();
converter.insertBeforeMethod
    (origin, before);
ctpoint.instrument(converter);
```

CodeConverter の持つ変換情報をクラスに反映するメソッドが instrument() である．Josh ではアスペクト内の全ての advice の情報を CodeConverter に記憶しておく．そしてロードされる各クラスとその CodeConverter で insrument することにより，ロード時のバイトコード変換，つまり weave を行っている．以上により (例 6) の advice を Javassist により weave できる．その他の advice についても同様に，部分コンパイラと合わせて使うことにより実現可能である．しかしながら現在の Javassist ではサポートしていない join point もある．例えば，例外ハンドラ実行を捉える機能を提供していないので，その join point に対しては advice を実装することはできない．

3.4 weave にかかる時間の測定

Josh によるロード時オーバヘッドを計測するために，我々は実験を行った．実験に用いた計算機は，Sun Blade 1000(Ultra SPARC III 750MHz × 2 CPU，Solaris 8) であり，JVM は Sun JDK1.3.1 付属の HotSpot^{TM} Client VM を用いた．コンパイラは Sun JDK 1.3.1 付属の javac コンパイラを用いた．

実験は複数の大きさのクラスファイルに 3 種類の weave を行い，その時間を計測したものである．図 3 に測定結果を示す．計測時間に部分コンパイルの時間は含まれない．図の Introduction は，1000 個のフィールドを introduction するというものである．既存クラスに新たなフィールドを追加するというバイトコード変換が主な処理であり，その時間は極めて短い．またクラスサイズにはほぼ影響を受けないことがわかる．

20Advices は，20 種類の advice をするというものである．advice を意味する CodeConverter を作成し，それを instrument してバイトコード変換をするのが主な処理である．この処理はクラスサイズに影響を受けていることが図よりわかる．その理由として，CodeConverter 作成時に advice 内の pointcut 特定にかかる時間の増加が考えられる．pointcut とは「何かが起こったとき」を表していることは既に述べたが，その瞬間を全てピックアップするには，クラスファイルを全て解析しなければならない．クラスサイズが大きければこの処理に時間がかかるのは当然であろう．これに比べピックアップ済の pointcut に対してバイトコード変換をするのは，クラスサイズの影響をあまり受けない．

1 long Advice は 1000 行ほどの 1 つの advice をするというものである．これはあまりクラスサイズの影響を受けていない．この場合は advice は 1 つだけなので pointcut 特定の手間が小さいからことからも，上記の推論が裏付けられる．

最後にオーバヘッドの目安として load time of class がある．これはクラスファイルをロードするのに通常かかる時間である．20Advices はこれを大きく越えているが，20 種類の advice をするという大がかりなアスペクトでも爆発的な時間にはなっていないことがわかる．

これらとは別に Josh 部分コンパイラによるバイトコード生成にかかる時間は大きい．advice，若し

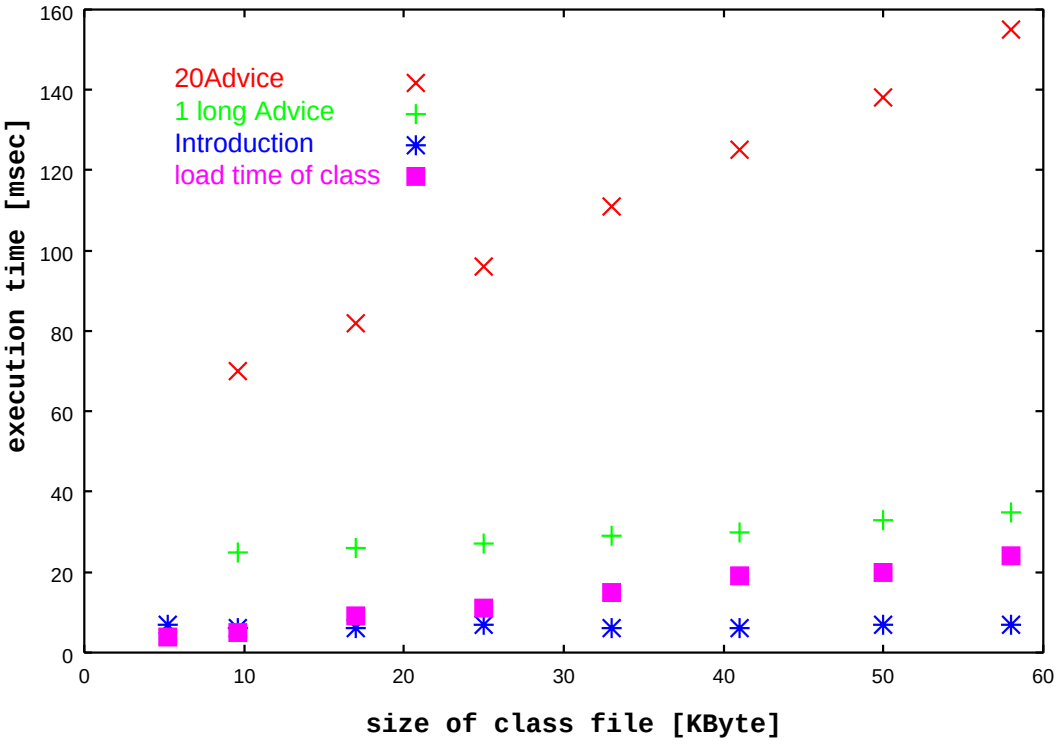

図 3: ロード時の weave にかかる処理時間

くはメソッド及びコンストラクタの introduction を部分コンパイラによりバイトコード化するならば，1 つあたりおよそ 1500msec 前後かかる．同程度のアスペクトを AspectJ コンパイラにより weave する場合は，weave 済バイトコードを出力するまでにおよそ 2500msec 前後かかる．ソースコード全体をコンパイルする時間の方が大きいことがわかる．Josh は現在ロード時にこのバイトコード生成をおこなっているが，将来的には Josh はロード時の前に静的にアスペクトを解析することを目指す．

4 関連研究

4.1 MixJuice

MixJuice[7] はクラスではなく差分をモジュールとして扱うことを Java に取り入れて拡張した言語である．差分モジュールとしては，クラス，フィールド，メソッドの定義に対する追加や修正を記述する．このモジュールを組み合わせることによりアプリケーションを構築していく．つまりメソッドやフィールド追加を記述したモジュールを組み合わせるので，introduction は実現できる．また同モジュールを複数のモジュールに同時に加えることもできるので，散らばってしまうコードの集約も可能である．しかし MixJuice では join point を捉える手段がなく，advice のように特定のフィールド参照やインスタンス生成を呼び変えたりすることができない．

4.2 MultiJava

MultiJava[8] は Java に open classes と symmetric multiple dispatch という機能を追加して拡張した言語である．open classes とは既存クラスを修正したりサブクラスを作ったりすることなく新しいメソッドを追加できる機能であり，これは introduction と同じことである．しかし MuliJava では追加できる要素がメソッドだけに限られてしまっている．

4.3 Binary Component Adaptation

Binary Component Adaptation(BCA)[5] は，Java を拡張してロード時のクラス定義変更を可能にした言語である．BCA では introduction と同じ

く，メソッド，フィールドの追加やスーパクラス，インタフェースの変更ができる．さらにメソッドやフィールドの名前の変更もできる．これらの機能を組み合わせればメソッド呼び代えや，特定フィールド参照を別フィールド参照にすることを実現できる．しかしながらフィールド参照を別メソッド呼び代えにしたり，例外ハンドラを扱うことなどはできない．BCA では delta-file と呼ぶ Java に近い文法を記述したファイルに，変更内容を記す．そして，独自の Java Virtual Machine(JVM) を使うことによりロード時に delta-file の内容を反映させている．このため BCA を使うためには専用の JVM もインストールする必要があり可搬性が低い．

5 まとめ

本稿では AspectJ 言語をバイトコードレベルで weave するコンパイラ Josh について述べた．Josh ではバイトコードレベルで weave することにより，必要のないコンパイルの手間を省略可能にした．またソースコードの無いサードパーティ製クラスにも weave を可能にした．また Javassist を使って実装をし，リフレクションの基本操作の組合わせで weave を実現できることを示した．最後にロード時 weave によるオーバヘッドを測定し，結果を示した．weave をするクラスサイズに影響を受けるが，十分に実用範囲の時間内で処理できることを確かめた．

現在 Josh は実装途中であり完成度は高くはない．さしあたっての課題は，使用できる advice の種類の追加と，ロード時前のアスペクトの静的な解析をする機能を追加することである．

参考文献

[1] Shigeru Chiba, Load-time Structural Reflection in Java, In *ECOOP 2000 – Object-Oriented Programming*, LNCS 1850, pp.313–336, 2000.

[2] 千葉滋，立堀道昭，Java バイトコード変換による構造リフレクションの実現，情報処理学会論文誌，42 巻 11 号，pp.2752-2760，2001 年 11 月

[3] Gregor Kiczals, John Lamping, Anurag Mendhekar, Chris Maeda,Cristin Lopes, Jean-Marc Loingtier, and John Irwin, Aspect-Oriented Programming, In *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pp.220–242, 1997.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G.Griswold, An Overview of AspectJ, In *ECOOP 2001 – Object-Oriented Programming*

[5] Ralph Keller and Urs Hölzle, Binary Component Adaptation, In *ECOOP 1998 – Object-Oriented Programming*, LNCS 1445, pp.307-329

[6] Günter Kniesel and Pascal Costanza, JMangler – A Framework for Load-Time Transformation of Java Class Files, In *IEEE Workshop on Source Code Analysis and Manipulation(SCAM), November 2001*

[7] 一杉裕志, 田中哲, 差分ベースモジュール，クラス独立なモジュール機構, 産業技術総合研究所テクニカルリポート, AIST01-J00002-1, 2001.

[8] Curtis Clifton, Gary T.Leavens, Craig Chambers and Todd Millistein, MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, In *Proceedings of OOPSLA 2000*, SIGPLAN Notices, Vol.35, No.10, pp.130–145, 2000.

[9] 鵜林尚靖，玉井哲雄，アスペクト指向プログラミングへのモデル検査手法の適用，オブジェクト指向最前線 2001 情報処理学会 oo2001 シンポジウム pp.41-48

[10] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba and Kozo Itano, A Bytecode Translator for Distributed Execution of "Legacy" Java Software, In *Proceedings of ECOOP 2001*, LNCS 2072, ECOOP 2001 – Object Oriented Programming, pp.236–255, Springer–verlag, 2001