平成13年度学士論文

Josh：バイトコードレベルでの Java用 Aspect Weaver

東京工業大学 理学部 情報科学科
学籍番号 98-1927-9
中川 清志

指導教官
千葉 滋 講師

平成14年2月7日

概要

現在オブジェクト指向言語はプログラミング言語の主流となりつつある．しかしある種の処理は複数のオブジェクトに同様に分散することがあり，これにオブジェクト指向は対応できていない．アスペクト指向はこの問題に対処した考え方であり，そのような処理をモジュールとして扱える．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オブジェクトに分散する処理を記述したアスペクトを合成 (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している．しかし標準の AspectJ コンパイラ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．この欠点に対処するために本稿は Josh を提案する．Josh はバイトコードレベルでアスペクトを weave する AspectJ コンパイラである．また Josh はクラスのロード時に weave することにより，必要のない weave の手間を省くことができる．そして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示す．最後に Josh を使って実験をし，ロード時のオーバヘッドが小さいことを示す．

謝辞

本研究を進めるにあたり，研究の方向付けや論文の組立て方についての助言をして頂いた指導教官の千葉先生に感謝致します.

筑波大学の立堀道昭氏には，論文の組立て方やプログラムの設計について，さらに参考文献に至るまでの助言を頂き感謝致します.

また東京大学の光来健一氏には論文のスタイルファイルを作って頂きました．筑波大学の横田大輔氏には深夜まで発表練習に付き合って頂きました．並びに同室で共に研究をしてきた皆様にも励ましていただきました．重ねてお礼を申し上げます.

目 次

第 1 章　はじめに **6**

第 2 章　AspectJ とそのコンパイラによる実装の問題点 **8**
- 2.1 アスペクト指向言語 . . . 8
- 2.2 AspectJ 言語のアスペクト記述 . . . 10
- 2.3 AspectJ コンパイラによる実装の問題点 . . . 20

第 3 章　Josh **24**
- 3.1 バイトコードレベルでのロード時 weave . . . 24
- 3.2 部分コンパイラ . . . 26
- 3.3 Josh weaver . . . 29
  - 3.3.1 Introduction . . . 34
  - 3.3.2 Advice . . . 36
  - 3.3.3 ロード時の weave . . . 41
- 3.4 weave にかかる時間の測定 . . . 43

第 4 章　関連研究 **46**
- 4.1 MultiJava . . . 46
- 4.2 MixJuice . . . 46
- 4.3 Binary Component Adaptation . . . 46

第 5 章　まとめ **48**

付 録 A　weave 後のソースファイル **51**

付 録 B　実験に用いたプログラム **54**
- B.1 Java ソースファイル . . . 54
- B.2 アスペクト . . . 55

図 目 次

2.1 AspectJ コンパイラの weave の流れ 21

3.1 Josh の weave の流れ 25
3.2 アスペクトの分類 31
3.3 ロード時の weave にかかる処理時間 44

表 目 次

2.1 JoinPoint クラスに定義されているメソッド例 20

3.1 Class クラスの内観用メソッド 30
3.2 CtClass クラスの内観用メソッド 30
3.3 Introduction 実現のための CtClass のメソッド 34
3.4 CodeConverter クラスのメソッド 37

第1章　はじめに

オブジェクト指向はデータに観点をおいてモジュールに分割する考え方である．オブジェクト指向プログラミングは個々のモジュラリティが優れており，そのため現在の主流となりつつある．しかしながらログ処理やメモリ管理などのシステマティックな要素に関する処理は，オブジェクト指向では十分に扱いきれない部分がある．このような処理は複数のオブジェクトに分散してしまうことが多く，その結果プログラムの保守性，再利用性が下がってしまう．アスペクト指向はこれに対処するための考え方であり，複数のオブジェクトに分散する処理をアスペクトとしてモジュール化して扱える．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合成 (weave) する．そして合成された 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などがある．

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．直接バイトコードを操作することにより 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．現在 Josh は実装過程にあり処理できるのは AspectJ 言語の一部である．

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，クラス構造の内観をするだけでなく変更を可能にする．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.

第2章　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．

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は優れている．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複数のオブジェクトに同様に分散して存在する．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割されてしまう．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分散することが多く，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．

例えばログ処理などがあげられる．以下のようなクラスがあったとする．

```
class Car {
    void start() {
        Log.print("start");
        ..
    }
}

class Bike {
    void start() {
        Log.print("start");
        ..
    }
}

class Log {
    static void print(String msg) {
        System.out.println(msg);
    }
}
```

Car クラスと Bike クラスは，メソッド実行の前にログ出力をするクラスである．この例ではそもそもの Car クラスの意味合いとは関係なくログ出力の処理が Car クラスに入り込んでしまっている．例えばここで，メソッド実行の後にログ出力をするという変更があったとする．その場合は Car クラスと Bike クラスを開いていちいち修正しなければならない．もちろん Log クラス自体に何らかの変更をした場合にも，Car, Bike クラスの修正は必要である．この例だと 2 クラスしかないがクラス数が増えてくると修正し忘れが起きたり，新クラスの追加の際にこの処理を書き忘れたりすることもありうる．本来ならばログ出力に関係する処理を変更した場合は，Log クラスの修正だけで済むようにしたい．しかしオブジェクト指向ではこのような問題に対応しきれていない．

アスペクト指向はこれらの問題を解決するための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分散する処理を，オブジェクトとは別の側面から考慮し，それをアスペクトとしてモジュー

ル化する．そして全体に分散してしまう処理を集約して扱える．アスペクトはシステマティックな要素を扱うことが多く，その例は

- メモリ管理
- 同期制御
- 例外処理
- ログ処理

などがある．

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．この処理によりアスペクトとして一括に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組み込むことができる．つまりプログラムコード全体に分散してしまう同一の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もつかを定義するが，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え r 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.getX() {
            return x;
        }
(例 3)  public Point.new(String s) {
            super(s);
            initialize();
        }
(例 4)  declare parents : Point extends SuperPoint;
(例 5)  declare parents :
            Point implements java.io.Serializable;
(例 6)  static final double Point.PI = 3.14;
(例 7)  public String (Point || Line || Square).getName() {
            return name;
        }
```

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 getX() というメソッドを追加する，(例 3) は Point クラスに public で String 型の引数を持つコンストラクタを追加する，という意味である．(例 2) においてメソッドの実行コード内にある x というフィールドは，Point クラス内のフィールドを指している．このためもし Point クラスに x というフィールドがないならばエラーが起きる．同様に (例 3) において initialize() というメソッドは，Point クラス内のメソッドを呼んでいるのでもし存在しないならばエラーが起

きる．そして (例 4) は Point クラスが SuperPoint クラスを継承している，(例 5) は Point クラスが java.io.Serializable を継承している，というようにクラスの階層構造を変更するという意味のアスペクトである．また’,’ で区切ることにより複数のインタフェースの追加が可能である．その場合は追加する全てのインタフェースに関して，そのインタフェース内で宣言されているメソッドを実装していないならばエラーが起きる．スーパクラスの継承変更においても’,’ で区切られた複数のクラスを継承することを許す．しかしこれらのクラスが全てどれかのクラスの親クラス，または子クラスでないならばエラーが起きる．(例 6) もフィールドの追加であるが，このような構文にすることによりフィールドの初期化をできる．さらに’||’ 演算子を使うことにより (例 7) のように複数のクラスに同時に同じメソッドやフィールドを追加することができる．このときも全てのクラスに name というフィールドがないならばエラーが起きる．

Pointcut

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概念が **pointcut** である．pointcut の例は，

(例 1)　`call(int Point.getX())`
(例 2)　`set(int Point.x)`
(例 3)　`handler(java.io.IOException)`

などがある．call や set は pointcut 指定子であり，特定の join point を指している．これらを組み合わせることにより pointcut を定義する．(例 1) 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 2) 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．(例 3) は java.io.IOException 例外が投げられる瞬間を捉える．この他の代表的な pointcut 指定子の例を以下に示す．

- 特定メソッドが実行されたとき

execution(void Point.setX(int))

- 特定フィールドから読み込んだとき
  get(int Point.x)
- 特定フィールドに書き込んだとき
  set(int Point.x)
- 特定オブジェクトを生成したとき
  initialization(Point.new())
- 特定クラスの静的初期化子を実行したとき
  staticinitialization(Point)
- 特定クラス内のコード実行に関わる全ての join point
  within(Point)
- 特定メソッド，コンストラクタ内のコード実行に関わる全ての join point
  withincodes(int Point.getX())
- 現在実行中のオブジェクトに関わる全ての join point
  this(Point)
- ターゲットのオブジェクトに関わる全ての join point
  target(Point)
- 特定クラスを引数とする全ての join point
  args(Point)

以上の pointcut 指定子で表される複数の join point を操作するために，or(||),and(&&),not(!) の 3 種類の演算子がある．また

⋆ ワイルドカードが使える

1. execution(∗ ∗(..))
2. call(∗ set(..))
3. call(*.new(int,int))

(1) は任意の返り値であり，任意の名前であり，引数も任意のメソッドの実行を，(2) は任意の返り値であり，引数も任意である set という名前のメソッド呼び出しを捉える．(1),(2) においてメソッドの引数が (..) となっているものは任意の引数をあらわしている．(3) は int,int という引数を持つ全てのクラスのコンストラクタ呼び出しをそれぞれ捉える．

⋆ 名前の一部にもワイルドカードが使える

1. call(void Point.make*(..))

(1) は Point クラス内で void 型の返り値を持ち，引数が任意であり，”make” で始まる全てのメソッド呼び出しを捉える．

⋆ 指定子を組み合わせて使える

1. taget(Point) && call(int ∗())
2. call(∗ ∗(..)) && (within(Point) || within(Line))
3. within(∗) && execution(∗.new(int))
4. this(∗) && !this(Point) && call(int ∗(..))

(1) は Point クラスをターゲットとし，返り値が int 型であり，引数を持たない全てのメソッド呼び出しを捉える．(2) は Point クラス内または Line クラス内からの全てのメソッド呼び出しを捉える．(3) は全てのクラス内の，一つの int 型引数をもつ全てのコンストラクタ実行を捉える．(4) は Point クラスのオブジェクト以外の全てのオブジェクトが，int 型の返り値を持つ全てのメソッドを呼ぶ瞬間を捉える．

⋆ 修飾子やその否定に基づいて，メソッドやコンストラクタを選べる

1. call(public ∗ ∗(..))
2. execution(!static ∗ ∗(..))
3. execution(public !static ∗ ∗(..))

(1) は全ての public メソッド呼び出しを，(2) は全ての static でないメソッド呼び出しを，(3) は public でありかつ static でない全てのメソッド呼び出しをそれぞれ捉える．

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() : call(void Point.setX(int));
```

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．

pointcut はプログラムの実行の瞬間を捉えるが，この瞬間の文脈から特定の値を取り出すこともでき，pointcut のパラメータとして使うことができる．このときに取り出されたパラメータは後述する advice に使われる．以下の pointcut があるとする．

```
pointcut setter(Point p) : target(p) &&
    (call(void setX(int)) ||
     call(void setY(int)));
```

これはターゲットが Point オブジェクトである，setX(int) もしくは setY(int) メソッド呼び出しを捉える．つまり':' の右側の事象が起きたときに，p という名前に束縛された Point オブジェクトを取り出すことができる．

Advice

Advice とは，ある join point 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．join point 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，

before または *after* : *pointcut* の指定 { 新実行コード }
返り値の型 *around* : *pointcut* の指定 { 新実行コード }

という並びで形成されている．pointcut の指定には前述した *pointcut* により定義済のものでも，一回の advice のために定義する自律型のものでもよい．ここでごく簡単な advice の例を示す．

（例 1）
```
before() : call(void Point.setX(int)) {
        System.out.println("before Point.setX()");
    }
```

```
(例 2)  before() : mysetX() {
            System.out.println("before Point.setX()");
        }
(例 3)  after() : call(Point.new(..)) || call(Line.new(..)) {
            System.out.println("after new");
        }
```

(例 1) は Point クラス内の void 型の返り値である setX() というメソッド呼び出しを捉え，その join point の動作の前に新実行コードを実行する．これは自律型 pointcut だが，(例 2) は定義済の mysetX() という pointcut を使い advice を表している．そして (例 3) は Point クラスまたは Line クラスの全てのコンストラクタ呼び出しを捉え，その join point の動作の後に新実行コードを実行する．

advice には before, after, around の 3 種類があると述べたが after にはさらに **retuning** と **throwing** の派生がある．returning は join point に付随する動作が正常に終了したときにのみ実行され，反対に throwing は Error もしくは Exception が投げられたときにのみ実行される．after はどちらの場合にも実行される．例えば以下のアスペクトがあったとする．

```
after() : call(void Point.function()) {
    System.out.println("after");
}
after() returning : call(void Point.function()) {
    System.out.println("returning");
}
after() throwing :  call(void Point.function()) {
    System.out.println("throwing");
}
```

そして

```
Point p = new Point();
p.function();
```

という Java コードがあったとする．このとき 2 つのコードを weave したならば，Java コードは以下のような意味合いを持つコードに変換される．

```
Point p = new Point();
```

```
try {
    p.function();
    System.out.println("returning");
} catch (Throwable e) {
    System.out.println("throwing");
    throw e;
} finally {
    System.out.println("after");
}
```

こうすることにより正常終了時のみ，異常終了時のみ，若しくはその両方の時といったように区別して実行することができる．

また advice では pointcut のパラメータを使用できる．

```
pointcut setter(Point p) : target(p) &&
                    (call(void setX(int)) ||
                     call(void setY(int)));
before(Point p) : setter(p) {
    System.out.println("set value : " + p);
}
```

1 つめの文で setter という pointcut を定義している．この setter(Point p) は Point オブジェクトをターゲットとし，setX(int) メソッドまたは setY(int) を呼ぶ瞬間を捉える．そして 2 つ目の文で before() アドバイスを定義している．この before(Point p) は setter(p) という事象が起きたときに，p という名前に束縛された Point オブジェクトを使うことを意味する．そして p がメソッドの引数で与えられたのと同じように扱うことができる．この例では p を println() メソッドの引数として使っている．もう一例示す．

```
pointcut setXY(Point p, int x, int y) :
    target(p) && args(x, y) && call(void Point.setXY(int,int));

before(Point p, int x, int y) : setXY(p, x, y) {
    p.printXY();
    System.out.println(p + " moved to (" + x + ", " + y + ") ");
}
```

このように before(Point p, int x, int y) というように複数のパラメータをとることもでき，p のメソッドを呼ぶこともできる．もちろん Point クラスが printX() メソッドを実装していないならばエラーが起きる．

もう1種類の advice として around がある．around を使うと，join point に付随する動作を実行する代わりに違う動作を実行することができる．例えば

```
void around() : call(void Point.function()) {
    System.out.println("function is redirected");
}
```

というアスペクトを weave したとする．これは Point クラスの function() メソッドの呼び出しを捉えている．このとき Point クラスの function() というメソッドが呼ばれると，function() の中身は実行されずに，アスペクト内の新実行コードが実行される．利用方法としては例えばソースファイルを編集せずにアスペクトを編集するだけで，メソッドの中身を編集したのと同じ結果が得られる．また around には *proceed* という特別メソッドが用意されている．これを使えばもともとの join point に結び付いていた動作を実行できる．例えば

```
pointcut setXY(Point p, int x, int y) :
    target(p) && args(x, y) && call(void Point.setXY(int, int));

void around(Point p, int x, int y) : setXY(p, x, y) {
    if (x <= XMAX && y <= YMAX)
        proceed(p, x, y);
    else
        System.out.println("over the max value!");
}
```

というアスペクトを weave したとする．これは Point オブジェクトをターゲットとして，setXY(int,int) というメソッド呼び出しを捉えている．そして around により本来実行されるはずのメソッドを実行せずに，新コードを実行している．しかし特別メソッド *proceed* を使い，本来のメソッドを実行することもできる．

アスペクトのリフレクション機能

AspectJ 言語では特別な参照用オブジェクトとして，**thisJoinPoint** がある．これは現在の join point の自己反映的な情報を含み，advice のために使われる．丁度 Java の *this* が static でないメソッドやコンストラクタで使われるのと同じように，thisJoinPoint は advice の中でだけ使われる．使用するには

```
before():call(void Point.setX(int)){
    System.out.println("thisJoinPoint is : " + thisJoinPoint);
}
```

のようにする．このアスペクトを weave した Java コードを実行したならば,

```
thisJoinPoint is : call(void Point.setX(int))
```

と表示される．advice の中では thisJoinPoint は org.aspectj.lang.JoinPoint 型のオブジェクトであり，join point に関する静的，及び動的な情報を持つ．もし静的な情報のみ (例えば join point のシグネチャのみ) を扱いたいならば，より軽量な **thisJoinPointStaticPart** オブジェクトがある．これも advice の構文の中ならば，thisJoinPoint と同じようにコード中から参照できる．thisJoinPointStaticPart は動的な情報を持たないが，パフォーマンスの面で thisJoinPoint よりも優れている．JoinPoint 型に定義されているメソッドを一部書く.

ここまでで AspectJ 言語のアスペクトの説明をしてきたがそれらは全てコード断片であった．一つのまとまったモジュールとして扱うには **aspect** と宣言する．aspect は class のようなものでありフィールドやメソッドも持てる.

```
aspect PointAspect {
    public final double Point.PI = 3.14;
    before() : call(int Point.getX()) {
        System.out.println("before getX()");
    }
}
```

メソッド名	説明
java.lang.String getKind()	この join point の種類を表す文字列を返す.
org.aspectj.lang.Signature getSignature	この join point のシグネチャを返す.
org.aspectj.lang.JoinPoint.StaticPart getStaticPart()	thisJoinPointStaticPart を返す.
java.lang.Object getTarget()	ターゲットオブジェクトを返す.
java.lang.Object getThis()	現在実行中のオブジェクトを返す.

表 2.1: JoinPoint クラスに定義されているメソッド例

これにより完全なアスペクトのファイルになり weave 可能になる．

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．また全ソースコードを読込んだ後に解析し，weave 後のソースコードを再びローカルディスクに書き込むので時間がかかる．

そのうえ AspectJ コンパイラは，weave の後に 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 2.1)．例を挙げて AspectJ 処理系の経過を追ってみよう．以下のような HelloWorld というクラスと，Trace というアスペクトがあったとする．

```
class HelloWorld {
    public static void main(String[] args) {
        new HelloWorld().printMessage();
    }

    void printMessage() {
        System.out.println("Hello world!");
```

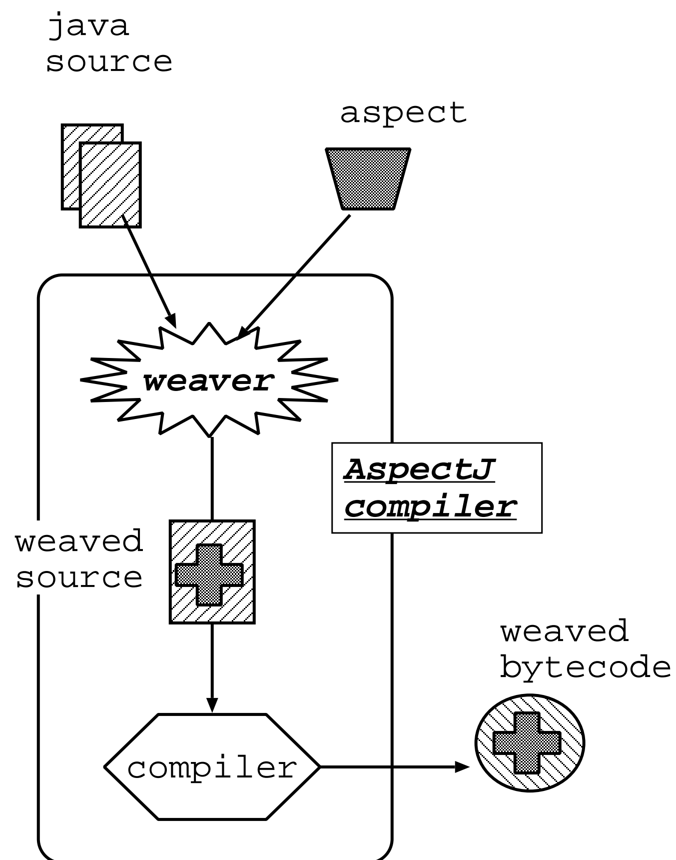

図 2.1: AspectJ コンパイラの weave の流れ

```
    }
}

-------------------------------------------

aspect Trace {

    pointcut printout() : call(void printMessage());

    before() : printout() {
        System.out.println("*** Entering printMessage ***");
    }

    after() : printout() {
        System.out.println("*** Exiting printMessage ***");
    }
}
```

このアスペクトの pointcut は void 型の printMessage() というメソッド呼び出しを捉える．そしてその前後にメッセージを表示するという advice を持つ．アスペクトを weave した HelloWorld を実行すると

```
%java HelloWorld
*** Entering printMessage ***
Hello world!
*** Exiting printMessage ***
```

という結果になる．しかしこのとき AspectJ コンパイラは作業ディレクトリを作成し，その中に weave 済のソースファイルを保管している (付録 A に添付)．このソースファイルをコンパイルすることにより weave をしたクラスファイルを出力している．つまり AspectJ コンパイラでは weave はコンパイルの前処理として実行されているのがわかる．

しかしこのように weave 後にコンパイルをする手法は効率が悪い．例えばアスペクトを weave せずに再実行したいとする．このときも図 2.1 からわからるように全てのソースファイルを再コンパイルしなければならない．同様にアスペクトの変更の度にも weave をし，全体を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．例えば実行時引数などで使用するクラスを使い分ける場合などには無駄がでてしまう．具体例を示すためにある通信プログラムを挙げる．このプログラムは実行時引数により，異なる暗号方式を用いるというものである．

```
public class TestEncryption {
    public static void main(String[] args) {
        Encryption encryption = null;
        if (args[0].equals("RSA"))
            encryption = new RSAEncryption();
        else if (args[0].equals("DES"))
            encryption = new DESEncryption();
        else
            :
            :
}
```

仮に実行時引数を”RSA” としたならば DESEncryption クラスを使うことはないとする．その場合は使われない DESEncryption クラスに weave をしている必要はない．しかしこのプログラムを weave した時点では，どのクラスを使うかは限定できない．そのため全てのクラスへ weave をしておかなければならないという無駄がでてしまう．

第3章　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．図 3.1 は Josh Weaver の流れを表している．この図からわかる通り weave はコンパイルの後に行われる．従ってアスペクトを weave しないでの実行も即座に行える．またアスペクトを変更したとしてもコンパイル済のクラスファイルには全く影響がない．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ので Josh の開発効率は AspectJ コンパイラよりも高い．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 できる．これにより 2.3 章で示したような必要のないファイルへの weave という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により，ロード時の weave を可能にしている．

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．

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，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．この章ではそれぞれの機能の説明と，Josh weaver の性能評価をする．

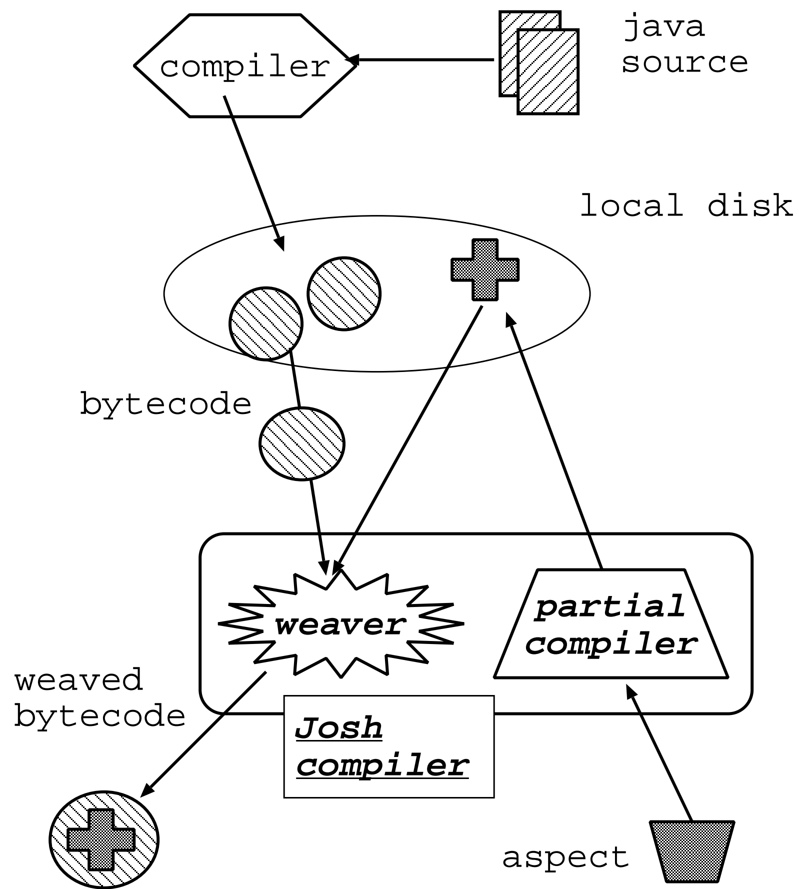

図 3.1: Josh の weave の流れ

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，フィールド名やメソッドのシグネチャは取得できる．メソッドのシグネチャを獲得するには以下のようにする．引数としては CtClass オブジェクトを渡しているが，Class クラスのオブジェクトでも可能である．

```
public static String outputMethods(CtClass cc) {
    String result = "";
    CtMethod[] ctmethods = cc.getDeclaredMethods();
    try {
        for (int i=0; i < ctmethods.length; i++) {
            result += "public"
                + ctmethods[i].getReturnType().getName() + ""
                + ctmethods[i].getName()
                + readMethodParameters(ctmethods[i])
                + readMethodBody(ctmethods[i]) + "\n";
        }
    } catch (NotFoundException e) {}
```

```
    return result;
}
```

まず引数として渡されたクラスの全メソッドを獲得する．そして各々のメソッドについて，返り値，名前，引数の並び，中身を文字列として書き出せばよい．引数の並びと中身に関してはさらに下請メソッドが行う．引数の並びを書き出す下請メソッドは以下のようになっている．仮引数名は本質的に関係がないので"_arg"という接頭辞と数字に変えてある．

```
private static String readMethodParameters(CtMethod ct) {
    CtClass[] ctargs = null;
    try {
        ctargs = ct.getParameterTypes();
    } catch (NotFoundException e) {}
    if (ctargs.length == 0)
        return "()";
    else {
        String result = "(";
        int i=0;
        for (i=0; i < ctargs.length-1; i++) {
            result ;= ctargs[i].getName() + " _arg" + i + ", ";
        }
        result += ctargs[i].getName() + " _arg" + i + " )";
        return result;
    }
}
```

メソッドの中身も本質的なところではないので，そのメソッドの中身に応じて適宜変えればよい．例えば return null; や return 0; にする．フィールド名に関しても同様な処理により用意に獲得できる．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 _arg0) {
        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．リフレクションとは，計算システムが自分自身の構成や計算過程に関する情報を取得して計算することである．自分自身の構造や計算過程をモデル化した表現を持ち，その表現を参照できることを内省という．その表現を変更すること，つまり自己改変もリフレクションという．Java 標準のリフレクション API では内省は行えるが，自己改変は行えない．例えば java.lang.Class クラスは内省用のメソッドは提供しているが，自己改変用のものは提供していない．そのためいくつかのリフレクション機能を強化するシステムが提案されてきた．Javassist はその一つであり構造リフレクションの機能を提供する．構造リフレクションとは内部のデータ構造を，必要に応じて変更できる機能である．つまり Javassist を使えば標準 Java リフレクションだけではできない，自己改変を行える．

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．それには

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

メソッド名	説明
Constructor[] getConstructors()	コンストラクタを得る
Field[] getFields()	フィールドを得る
Method[] getMethods()	メソッドを得る
int getModifiers()	修飾子を得る
String getName()	クラス名を得る
Class getSuperclass()	親クラスを得る
boolean isInterface()	インタフェースかどうかを調べる
boolean isPrimitive()	プリミティブ型かどうかを調べる

表 3.1: Class クラスの内観用メソッド

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．ClassPool は必要に応じてクラスファイルを読込み CtClass オブジェクトを生成する．そしてそのオブジェクトをファイルなどに出力するまで，それを保持しておく．また Javassist ではフィールド，メソッド，コンストラクタの情報は，CtField，CtMethod，CtConstructor オブジェクトにより表される．これらのクラスはそれぞれ Java リフレクション API の Field, Method, Constructor に相当する．Class クラスが提供するのと同じように，CtClass が提供するメソッドにより CtField などは得られる．

メソッド名	説明
CtConstructor[] getConstructors()	コンストラクタを得る
CtField[] getFields()	フィールドを得る
CtMethod[] getMethods()	メソッドを得る
int getModifiers()	修飾子を得る
String getName()	クラス名を得る
CtClass getSuperclass()	親クラスを得る
boolean isInterface()	インタフェースかどうかを調べる
boolean isPrimitive()	プリミティブ型かどうかを調べる

表 3.2: CtClass クラスの内観用メソッド

さらに Javassist では新しい CtField などを作るためのコンストラクタ

も用意されている．

以後では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．それに先立ち構文解析の説明をしておく．

構文解析

AspectJ 言語ではアスペクトを表すモジュールは aspect として宣言される．aspect は class のようなものでありフィールドやメソッドを持てる．そして aspect 特有の introduction と advice を持つ．しかし Josh の aspect は introduction と advice のみを持つということに制限している．従って aspect ファイル内の要素は図 3.2 のように分類される．構文解析器は，各

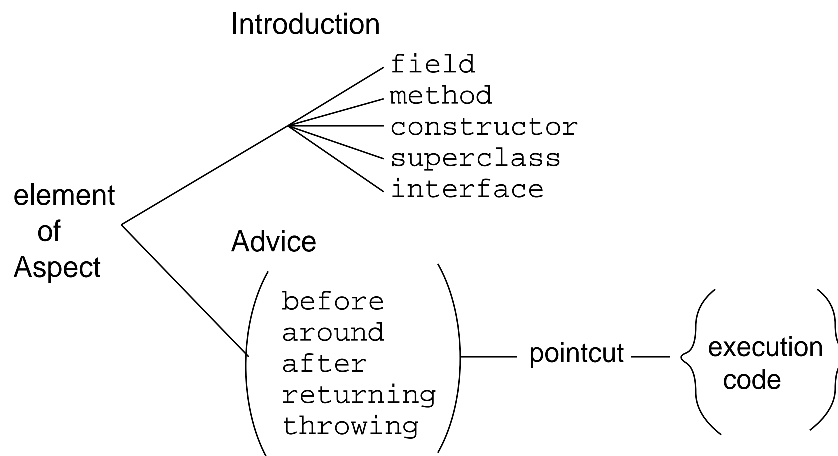

図 3.2: アスペクトの分類

種類の要素にアスペクトを分類し，後の利用のために必要な情報を保管しておく．introduction とは以下のようなコードであった．

```
private int Point.x;
public int Point.getX() {
    return x;
}
declare parents : Point extends SuperPoint;
```

上から順にフィールド，メソッド，スーパクラスをそれぞれ追加する．フィールドイントロダクションについては，修飾子，型，追加先のクラス

名，フィールド名，初期化の有無についての情報が必要である．メソッドイントロダクションについては，修飾子，返り値，追加先のクラス名，メソッド名，引数の型と並び，実行コードの情報が必要である．スーパクラスイントロダクションについては，追加先のクラス名，追加 (変更) されるスーパクラス名の情報が必要である．それらを以下のクラスのオブジェクトとして保持し，必要に応じて取り出せる状態にしておく．コンストラクタ，インタフェースの introduction についても同様なクラスが用意してある．

```
public abstract class Introduction {
        :
    public abstract void introduce(String classname);
}

public class FieldIntroduction extends Introduction {
    int modifier;
    String type;
    String addclass;
    String fieldname;
    boolea initneed;
    FieldInitializer init;
         :
         :
}

public class MethodIntroduction extends Introduction {
    int modifier;
    String returntype;
    String addclass;
    String methodname;
    String paramstext;
    String code;
        :
        :
}
```

```
public class SuperclassIntroduction extends Introduction {
    String addclass;
    String superclass;
        :
        :
}
```

MethodIntroduction の paramstext とは，引数の型と仮引数名の並びをアスペクト内のコードのまま保管したものである．各種の introduction は Introduction クラスのサブクラスであり，Introduction オブジェクトの配列として保管しておく．また advice とは以下のようなコードであった．

```
before() : call(int Point.getX()) {
    System.out.println("before call getX");
}
after() returning : get(d Point.setX(int)) {
    System.out.println("returning call setX");
```

advice において必要な情報は，advice の種類，pointcut の種類，実行コードである．この例だとメソッド呼び出しという join point であるので，メソッドを特定するためにシグネチャを保管する．そしてこのシグネチャを保持する join point を表すオブジェクトを生成する．最後にこの join point と実行コードを持つオブジェクトを生成すれば，全ての情報を保管することができる．

```
public class MethodSignature extends Signature {
    String declaring;
    String rtype;
    String mname;
    String[] params;
        :
        :
}

public class CallPoint extends JoinPoint {
    MethodSignature signagure;
        :
        :
```

```
}

public class BeforeAdvice extends Advice {
    JoinPoint joinpoint;
    String code;
        :
        :
}
```

introduction, advice ともに実際に weave 処理を行うタイミングについては後に述べる．

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．ここでは Javassist を用いて同様の処理が可能であることを示す．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 (図 3.3).

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 setSuperclass(..)	親クラスを変更する

表 3.3: Introduction 実現のための CtClass のメソッド

以下の例を具体的に実装してみる．

```
(1)  private int Point.x;
(2)  private final Point.PI = 3.14;
(3)  public double Point.getX() {
         return x;
     }
(4)  declare parents : Point extends SuperPoint;
```

```
(5)  declare parents :
         Point implements java.io.Serializable, java.lang.Runnable;
```

まず先ほど説明した方法により，Point クラスを表す CtClass オブジェクト，ctpoint を手にいれる．そしてそのオブジェクトに対して表 3.3 にあるメソッドを使えばよい．しかし実際にフィールド追加するためには，そのフィールドを表す CtField オブジェクトを作らなければならない．メソッド，コンストラクタの追加の場合も同様に CtMethod, CtConstructor オブジェクトを作らなければならない．具体的に (1) の introduction をするには以下のコードを実行する．新しく CtField オブジェクトを生成し追加している．

```
CtField ctfield1 =
     new CtField(CtClass.intType, "x", ctpoint);
ctpoint.addField(ctfield1);
```

(2) の場合は同時に初期化しているので javassist.FieldInitializer クラスを使う．このクラスは CtField オブジェクトを CtClass オブジェクトに追加するときに初期化をしてくれる．

```
CtField ctfield2 =
    new CtField(CtClass.doubleType, "PI", ctpoint);
FieldInitalizer init =
    FieldInitializer.constant(3.14);
ctpoint.addField(ctfield2, init);
```

(3) はメソッドの追加である．CtMethod オブジェクトもコンストラクタで作ることが可能だが，メソッドの中身としては別に存在するクラスのメソッドの中身を指定しなければならない．しかし (3) の場合は新しい実行コードが記述されており，これはどこのクラスにも存在しない．そこで Josh 部分コンパイラを用いて必要な部分をバイトコード化する．部分コンパイラが”DummyJosh”という名前でコンパイルしたならば，そのクラスには目的とするメソッドが存在しているはずである．こうして作成した目的とするメソッドを追加するには,

```
CtClass ctdummy = pool.get("DummyJosh");
CtMethod ctmethod = ctdummy.getDeclaredMethod("getX");
ctpoint.addMethod(ctmethod);
```

とすればよい．pool は ClassPool クラスのオブジェクトである．コンストラクタの追加も同様に，部分コンパイラにより手にいれた CtConstructor オブジェクト，ctconstructor を

```
ctpoint.addMethod(ctconstructor);
```

とすればよい．(4) はスーパクラスの変更である．AspectJ 言語では同時に複数のスーパクラスの introduction を許している．しかし追加する全てのクラスが，他のクラスの親クラスまたは子クラスでなければならない．つまり結果としてはグループの中で一番の子クラス (グループ内の全てのクラスを親クラスとして持つクラス) を，直接の親クラスに指定するのと同じことである．ユーザがアスペクトを記述する際に複数のクラスの階層構造がわからない場面というのは考えにくい．従って現在の Josh では指定できるスーパクラスは一つに制限している．複数のクラスの階層構造は Java リフレクション API を使えば調べられるので，複数の親クラスの指定も実装は可能である．(4) の実行には以下のようにする．

```
CtClass super = pool.get("SuperPoint");
ctpoint.setSuperclass(super);
```

(5) はインタフェースの追加である．インタフェースは複数追加可能であるのでそれぞれについて処理する．

```
String[] interfaces =
    {"java.io.Serializable", "java.lang.Runnable"}
for (int i = 0; i < interfaces.length; i++) {
    CtClass ctinter = pool.get(interfaces[i]);
    ctpoint.addInterface(ctinter);
}
```

しかし例えば Runnable インタフェースが実装しているメソッド run() を Point クラスが実装していないならば，javassist.CannotCompileException() が投げられる．以上の操作の組み合わせにより introduction を実現できる．

3.3.2　Advice

Advice とはある join point 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．この節では advice と等価なバイトコード

変換処理をする，CodeConverter オブジェクトの作成方法を示す．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 (図 3.3.2)．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 insertAfterMethod(..)	メソッド呼び出しの後に別メソッドを呼ぶ
void insertBeforeMethod(..)	メソッド呼び出しの前に別メソッドを呼ぶ
void redirectFieldAccess(..)	フィールド参照を別フィールド参照に代える
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 replaceFieldWrite(..)	フィールド書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 3.4: CodeConverter クラスのメソッド

この節では以下の例を実装していく．

```
(1)  before() : call(int Point.getX()) {
         System.out.println("before call getX");
     }
(2)  after() returning : call(void Point.setX(int)) {
         System.out.println("returning call setX");
     }
```

(1) は call(int Point.getX()) というメソッド呼び出しの join point の前に新コードを実行するものである．そして新コード実行というのはメソッド実行であっても構わないはずである．つまりこの advice はあるメソッドを呼ぶ前に別メソッドを呼ぶ，という意味合いに置き換えることができる．そのためには 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 call getX”);} 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．まず origin メソッドを特定する．origin メソッドは (1) の場合は int 型の返り値を持ち，Point クラス内に定義されており，引数を持たない getX というシグネチャである．そして既に構文解析をして MethodSignature オブジェクトに格納されている．MethodSignature クラスは，自分のシグネチャと別のメソッドを比較するためのメソッド，getCorrespondMethod() を持っている．そのメソッドの引数は CtClass オブジェクトであり，そのオブジェクトが表しているクラスの全てのメソッドと比較する．getCorrespondMethod() の中身では様々な比較をしている．以下にその中身を示す．

```
public class MethodSignature extends Signature {
    String declaring;
    String rtype;
    String mname;
    String[] params;
    ClassPool pool;
        :
        :

    /* cc クラス内で, この signature と,
     * マッチするメソッドを探す.  */
    public CtMethod getCorrespondMethod(CtClass cc) {
        CtClass cc = null;
        try {
            cc = pool.get(declaring);
        } catch (NotFoundException nfe) {
            return null;   //一致するメソッドは存在しない.
        }
        CtMethod[] ctmethods = cc.getDeclaredMethods();
        for (int i=0; i < ctmethods.length; i++) {
            if (compareMethod(ctmethods[i]))
                return ctmethods[i];
```

```
        }
        return null;
    }

    /* この signature とメソッドが等しいかを比べる. */
    private boolean compareMethod(CtMethod ct) {
        if ( declaring.equals(ct.getDeclaringClass().getName()) &&
             compareRtype(ct) &&
             compareMethodname(ct) &&
             compareParams(ct) )
            return true;
        else
            return false;
    }

    /* 返り値を比べる */
    private boolean compareRtype(CtMethod ct) {
        if (rtype.equals("*"))
            return true;
        else
           try {
               if (rtype.equals(ct.getReturnType().getName()))
           return true;
               else
                   return false;
            } catch (NotFoundException nfe) {
                throw new RuntimeException();
            }
    }

    /* メソッド名を比べる */
    private boolean compareMethodname(CtMethod ct) {
        if (mname.equals("*"))
            return true;
        else {
```

```
            if (mname.equals(ct.getName()))
                return true;
    else
        return false;
        }
    }

    /* 引数を比べる */
    private boolean compareParams(CtMethod ct) {
        CtClass[] ctparams = null;
        try {
            ctparams = ct.getParameterTypes();
        } catch (NotFoundException nfe) {}
        if (ctparams.length != params.length)
            return false;
        else {
            for (int i=0; i < ctparams.length; i++)
                if (!ctparams[i].getName().equals(params[i]))
                    return false;
            return true;
        }
    }
```

このメソッドを使い join point となっているメソッド，つまり origin メソッドを特定できる．origin メソッドは 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．次に before メソッドの方を手にいれなければならないが，before メソッド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，(1) の advice 実現のためのバイトコード変換をする CodeConverter を得られる．以下のコードにより目的とするバイトコード変換を保持する CodeConverter オブジェクトが手にはいる．

```
CodeConverter converter =  new CodeConverter();
```

```
converter.insertBeforeMethod(origin, before);
```

(2) については implementAfter(CtMethod origin, CtMethod after) を使えばよい．このメソッドは origin メソッドが正常に実行された後に after メソッドを実行するようにバイトコードを変換してくれる．つまり after returning と同じ役割を果たす．(2) の場合も (1) と同じように origin, after メソッドを手にいれて，

```
CodeConverter converter =  new CodeConverter();
converter.insertAfterMethod(origin, after);
```

とする．これらにより advice と等価な，バイトコード変換処理をするためのオブジェクトが手にはいる．他の advice，例えば after や after throwing を実現するには，redirectMethodCall(CtMethod origin, CtMethod subst) を使えばよい．これは origin メソッドの代わりに subst メソッドを呼ぶようにバイトコードを変換する．そして呼び代え先のメソッド，subst 内でエラー処理を含めた新コードをを実行すればよい．また，その他の join point に関しても get, set は replaceFieldRead(), replaceFieldWrite() を使えばよい．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メソッド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．そしてまた現在 Josh は実装中であり，これらの after や after throwing，get や set の処理は対応できていない．

3.3.3　ロード時の weave

前節では advice と等価なバイトコード変換を行う CodeConverter を得られた．この節では実際にロードされたクラスに変換情報を反映，つまり weave する方法を示す．

ロードされる各クラスとと weave するために，Josh では javassist.Loader でクラスをロードしている．Loader は java.lang.ClassLoader のサブクラスであるが，通常のクラスローダと違い，ClassPool からバイトコードを手にいれる．ClassPool は外部 (ローカルディスクや JVM) にクラスを出力するときは，必ず write() メソッドを呼ぶ．また ClassPool には javassist.Translator が登録されており，write() が行われたことをそれに通知するようにできている．そのため Loader がクラスを JVM にロードする際

にも必ず ClassPool.write() が呼ばれ，登録されている Translator にも通知される．Translator は以下のようなインタフェースである．

```
public interface Translator {
    public void start(ClassPool pool)
        throws NotFoundException, CannotCompileException;
    public void onWrite(ClassPool pool, String classname)
        throws NotFoundException, CannotCompileException;
}
```

start() メソッドはこの Translator が ClassPool に登録されたときに呼ばれる．onWrite() メソッドは ClassPool オブジェクトの write() メソッドからの通知を受取ったときに呼ばれる．onWrite() の第 2 引数は ClassPool オブジェクトが出力するクラスの名前である．つまりロードされる各クラス名を引数とする onWrite() メソッドが必ず呼ばれるので，このメソッド内に weave 処理を記述すればよい．

バイトコード変換情報を持つ CodeConverter オブジェクトを，実際に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をその 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．これらを行うための onWrite() メソッドを実装した JoshTranslator クラスを以下に示す．解析済の advice と introduction の全てをこのクラスが保持している．

```
public class JoshTranslator implements Translator {
    CodeConverter[] converters;
    Introduction[] introductions;
        :
        :
    public void onWrite(ClassPool pool, String classname)
        throws NotFoundException, CannotCompileException {

        CtClass ctclass = pool.get(classname);
        for (int i=0; i < introductions.length; i++) {
            introductions[i].introduce(classname);
        }
        for (int i=0; i < converters.length; i++) {
```

```
            ctclass.instrument(converters[i]);
        }
        :
        :
    }
}
```

そして ClassPool への，Translator 登録と Loader 関連づけの方法を示す．run() メソッドにより Loader を用いたアプリケーション実行を開始する．

```
ClassPool pool = ClassPool.getDefault();
JoshTranslator translator = new JoshTranslator();
ClassPool joshpool = new ClassPool(pool,translator);
JoshLoader loader = new JoshLoader(joshpool);
    :
    :
loader.run(...);
```

以上で具体的なロード時 weave の方法を示せた．しかしながら上記の通り Josh では独自クラスローダを使用している．このため java パッケージや javax パッケージなどのシステムクラスに対しては，ロード時に weave することができない．

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 種類のアスペクトを weave し，その時間を計測したものである．実験に用いたクラスファイルのソースと 3 種類のアスペクトを付録 B に添付する．図 3.3 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．オーバヘッドの目安として，通常通りのクラスのロードにかかる時間も併記する．Introduction のプロットは，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

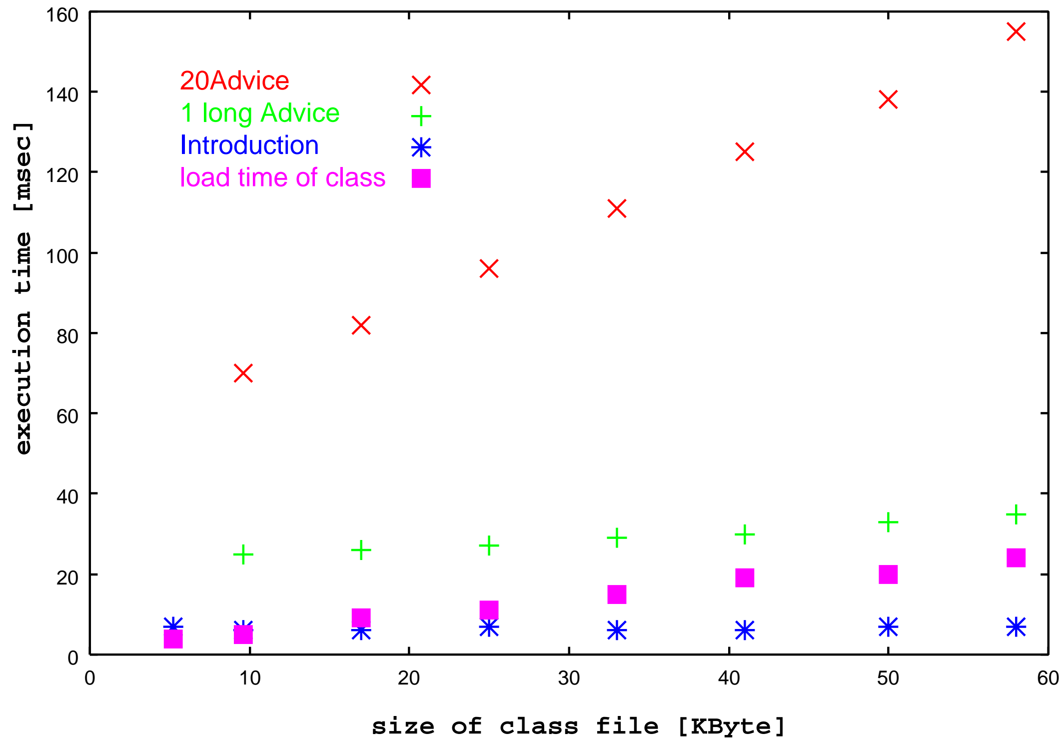

図 3.3: ロード時の weave にかかる処理時間

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 のプロットは，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，それで 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 のプロットは 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．全体としては，オーバヘッドが爆発的に増えるようなことがないことも確認できる．

これらとは別に現在はアスペクトの解析をロード時に行っているので，それによるオーバヘッドもある．その中でも特に 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，著しく大きい．advice，若し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 つあたり約 1300〜1500msec かかる．これをロード時に実行するには時間がかかり過ぎるので，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．

第4章　関連研究

4.1　MultiJava

MultiJava[6] 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．

4.2　MixJuice

MixJuice[5] 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，観点の分離の支援もできている．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．

4.3　Binary Component Adaptation

Binary Component Adaptation(BCA)[7] 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．

第5章　まとめ

本稿では AspectJ 言語をバイトコードレベルで weave するシステム 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，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．そしてロード時に weave することにより開発効率を改善した．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，ロード前のアスペクトの静的な解析である．

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] 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.

[6] 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.

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

[8] 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*

[9] 渡部卓雄，リフレクション，コンピュータソフトウェア，vol.11 No.3 May 1994 pp.165-174

[10] AspectJ – Aspect-Oriented Programming(AOP) for Java, http://aspectj.org/servlets/AJSite

[11] Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist

[12] 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

付 録A　weave後のソースファイル

以下に 2 つのソースファイルを示す．2.3 章において HelloWorld.java という Java ソースと Trace.java というアスペクトを weave した結果のソースファイルである．

```
/*   Generated by AspectJ version 1.0beta1 */
package helloworld;
class HelloWorld {
  public static void main(String[] args) {
    HelloWorld.printMessage$method_call(new HelloWorld());
  }

  void printMessage() {
    try {
      Trace.aspect$.before0$ajc();
      this.printMessage$ajcPostCall();
    } finally {
      Trace.aspect$.after0$ajc();
    }
  }

  public HelloWorld() {
    super();
  }
  void printMessage$ajcPostCall() {
    System.out.println("Hello world!");
  }
```

```
  private static void printMessage$method_call(HelloWorld arg$callThis) {
    try {
      Trace.aspect$.before0$ajc();
      arg$callThis.printMessage$ajcPostCall();
    } finally {
      Trace.aspect$.after0$ajc();
    }
  }

}

------------------------------------------------------------


/*   Generated by AspectJ version 1.0beta1 */
package helloworld;
class Trace {
  public final void before0$ajc() {
    System.out.println("*** Entering printMessage ***");
  }

  public final void after0$ajc() {
    System.out.println("*** Exiting printMessage ***");
  }

  public Trace() {
    super();
  }
  public static Trace aspect$;
  public static Trace aspectOf() {
    return Trace.aspect$;
  }

  public static boolean hasAspect() {
    return Trace.aspect$ != null;
  }
```

```
  static {
    Trace.aspect$ = new Trace();
  }

}
```

付 録B　実験に用いたプログラム

B.1　Java ソースファイル

実験に用いたクラスファイルのソースを載せる．この例は Class30.java であり newHello() メソッドを 400 個持っている．このメソッドの数を変えることにより複数の大きさのクラスファイルを作成した.

```
public class Class30 {
    public static void main(String[] args) {
        Class30 cc = new Class30();
        cc.exe();
    }

    public void exe() {
        newHello0();
        newHello1();
        newHello2();
            :
            :
            :
        newHello98();
        newHello99();
    }

    /************************************************************/

    public void  newHello0() { System.out.println("hello0 "); }
    public void  newHello1() { System.out.println("hello1 "); }
```

```
    public void  newHello2() { System.out.println("hello2 "); }
                :
                :
                :
    public void  newHello398() { System.out.println("hello398 "); }
    public void  newHello399() { System.out.println("hello399 "); }
}
```

B.2　アスペクト

3.4 章の実験に用いたアスペクトを 3 種類載せる．各ファイルは単調な繰り返しが続くので抜粋してある．また現在の仕様により予約語 *aspect* とそれに続く名前などはファイルに記述されていない．それぞれの例は Class30.class というクラスと weave するためのアスペクトである．他のクラスと weave するときは適宜名前を変えて実験を行った．

- Introduction

これは 1000 個のフィールドの introduction をするアスペクトである．

```
public int Class30.newField0;
public int Class30.newField1;
public int Class30.newField2;
        :
        :
        :
public int Class30.newField998;
public int Class30.newField999;
```

- 20 Advices

これは 20 種類の advice をするアスペクトである．

```
before():call(void josh.sample.lisp.Class30.newHello0())
    { System.out.println("before0"); }
before():call(void josh.sample.lisp.Class30.newHello1())
    { System.out.println("before1"); }
before():call(void josh.sample.lisp.Class30.newHello2())
    { System.out.println("before2"); }
```

```
          :
          :
          :
before():call(void josh.sample.lisp.Class30.newHello18())
    { System.out.println("before18"); }
before():call(void josh.sample.lisp.Class30.newHello19())
    { System.out.println("before19"); }
```

- 1 long Advice

これは 1 つの advice だが，その advice の中身が非常に長いというアスペクトである．

```
before():call(void josh.sample.lisp.Class30.newHello1()) {
    System.out.println("before0");
    System.out.println("before1");
    System.out.println("before2");
        :
        :
        :
    System.out.println("before998");
    System.out.println("before999");
}
```