Javaバイトコード変換による構造リフレクションの実現

千葉　滋[†,††]　立堀道昭[†††]

Java 標準のリフレクション API ではプログラムの内観（introspection）を行えるが，プログラムの振舞いを変更することはできない．本論文は，リフレクション API を拡張して，プログラムの振舞いを変更できるようにすることを提案する．従来の動作リフレクションに基づいた拡張と異なり，本論文で提案する Javassist は構造リフレクションに基づく．標準の Java 仮想機械（JVM）を使ったまま，実行性能の低下をともなわずにリフレクションを実行できるようにするため，Javassist ではクラスが JVM にロードされる前でのみ構造リフレクションを許す．本論文は Javassist の設計上の要点を示し，関連する研究との違いを明らかにする．

Structural Reflection by Java Bytecode Instrumentation

Shigeru Chiba[†,††] and Michiaki Tatsubori[†††]

The standard reflection API of Java provides the ability to introspect a program but not to alter program behavior. This paper presents an extension to the reflection API for addressing this limitation. Unlike other extensions enabling behavioral reflection, our extension called *Javassist* enables structural reflection in Java. For using a standard Java virtual machine (JVM) and avoiding a performance problem, Javassist allows structural reflection only before a class is loaded into the JVM. This paper presents the design principles of Javassist, which distinguish Javassist from related work.

1. はじめに

Java は標準 API（Application Programming Interface）の一部としてリフレクションの機能を提供するプログラミング言語である．しかしながら提供される機構はプログラム中で用いられるデータ構造，すなわちクラス，の定義を調べる機能（introspection）が主で，プログラムの振舞いを変更する機能は非常に限られている．

Java のリフレクション機能を強化するために，これまでいくつかのシステムが提案されてきたが，そのほとんどは**動作リフレクション**（behavioral reflection）の機能を提供している．これはメソッド呼び出しのような演算を横取りして，その動作を変更できるようにするものである．この機能を使うと，プログラマはたとえばメソッド呼び出しの動作を変更してプログラムが耐故障性を備えるようにすることができる．

† 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
†† 科学技術振興事業団さきがけ研究 21
PRESTO, Japan Science and Technology Corporation
††† 筑波大学大学院工学研究科
Doctral Program in Engineering, University of Tsukuba

本論文では，動作リフレクションではなく，**構造リフレクション**（structural reflection）の機能を提供するクラスライブラリ Javassist について述べる．構造リフレクションは，クラスや関数，レコードなどのデータ構造の定義を必要に応じて変更できるようにする機能である．本論文は，ある種の有用な言語拡張を実装するには，動作リフレクションよりも構造リフレクションの方が適していることを述べる．動作リフレクションは，そのような言語拡張を実装するには機能的に不十分であり，無理に実装しようとすると，分かりにくいプログラミング技巧で間接的に欠けている機能を補わなければならない．一方，構造リフレクションは，そのような言語拡張に必要な機能を直接提供するので実装は容易である．また動作リフレクションは構造リフレクションを使って容易に実現できる．したがって構造リフレクションさえ提供されていれば，その上に動作リフレクションを実現し，それを使って目的の言語拡張を実装することも可能である．

Javassist で構造リフレクションを実現するにあたり，我々はクラスを Java 仮想機械にロードする際にバイトコード変換を行うことで実現する方法を開発した．従来知られている実現方法では，Java 仮想機械の内部を変更する必要があったが，可搬性が重要な Java

言語ではこの方法は現実的ではない．またソースコード変換器を使って実現する方法では，ソースコードなしでは構造リフレクションが利用できない，処理速度が遅い，という問題があった．Javassist で採用したバイトコード変換による方法では，このような問題を回避できる．

以下では，まず次章で，動作リフレクションの問題点を議論し，続いて 3 章で Javassist の概要を述べる．4 章では Javassist の API の設計上の要点を示し，5 章で関連する研究との比較を行い，6 章で本論文をまとめる．

2. Java 言語のための動作リフレクション

Java 言語で動作リフレクションを実現するにあたって問題となるのは実行性能である．素朴な方法は，プログラム全体を Java 自身で書かれたインタプリタを使って実行する方法であるが，この方法は大幅な実行性能の低下をともなう．

実行性能の低下を避けるために，過去に提案されたシステムのほとんどは動作リフレクションの機能を制限している．そのようなシステムでは，メソッド呼び出しや，フィールド・アクセス，オブジェクト生成など，特定の種類の演算の動作だけが変更できる．実行系は，選択された演算の直前にフックを挿入し，その演算の実行を横取りできるようにしている．実行系はその演算を直接実行する代わりに，**メタオブジェクト**と呼ばれる，対象となるオブジェクトまたは演算と関連付けられたオブジェクトのメソッドを呼び出す．横取りされた演算の実行は，このメソッドによって実装される．プログラマは望みのメタオブジェクトを定義して，横取りされた演算の動作を自由に変えることができる．このように制限された動作リフレクションの実行時のオーバヘッドは小さい．これは，横取りされた演算のみが実効性能の低下を起こし，その他の部分はオーバヘッドなしに実行されるからである．

制限された動作リフレクションは，実行時のオーバヘッドが小さいが，一方でリフレクションの応用範囲をせばめてしまう．このような動作リフレクションは，ある種の言語拡張の実装に必要な機能を直接提供しない．そのような言語拡張の実装では，プログラム中で明示的に定義されていないクラスを処理系が自動的に定義したり，すでに定義されているクラスを変更してメソッドを追加したりする機能が必要である．動作リフレクションは，そのような機能を提供しないので，手間のかかるプログラミング上の工夫で同等の効果を間接的に達成しなければならない．

たとえば，分散オブジェクトの一般的な実装では，プロキシと呼ばれるオブジェクトのクラスを処理系が自動的に定義する必要がある．動作リフレクションの範囲内では，新しいクラスを直接定義することはできないので，動作リフレクションを使って分散オブジェクトを実装する場合，次のような工夫をする．今，定義したいプロキシのクラスを P とすると，クラス P は定義せずに，既存の別のクラス C（たとえばプロキシを通して参照されるオブジェクトのクラス）を P の代用とする． P のオブジェクトを生成するときには，代わりに C のオブジェクトを生成し，そのオブジェクトに対するメソッド呼び出し，フィールド・アクセスをすべて横取りし，そのオブジェクトがあたかも P のオブジェクトであるかのように振る舞わせれば，クラス P を新たに定義できなくても，同等の結果を得られる．

このほかにも，あるクラス D に新しいメソッド `m()` を追加するときには， D に直接メソッド `m()` を追加する代わりに， D のオブジェクトに対するメソッド呼び出しをすべて横取りし，本来未定義である `m()` が呼ばれたときには，エラーにせずに対応する処理を実行するという，直感的とはいいがたい工夫が必要となる．

本来，リフレクションの目的は，言語拡張の実装者が共通に必要とする機能の集合を提供することで，実装者の労力を軽減することである．したがって，上記のような容易には思いつけない技巧を知らなければ目的の言語拡張が実装できない動作リフレクションは，実用上望ましい機能を提供しているとはいえない．また上記のような技巧を使うと，実装された言語拡張の実行効率が悪くなるので，技巧をライブラリの形でプログラマに提供しても実用にならない．これまで，Reflective Java[18] や，Kava[17]，MetaXa[9),14] など，多くの動作リフレクションのシステムが Java 言語用に開発されてきたが，現実の応用範囲は狭いといえる．

3. ロード時の構造リフレクション

本章では，Java 言語で構造リフレクションを可能にするために，我々が開発したクラスライブラリ Javassist について述べる．構造リフレクションは，クラスやメソッドのようなデータ構造の定義をプログラム中から編集する機能を提供する．新しいクラスを定義したり，既存のクラスにメソッドを追加する機能が直接提供されるので，動作リフレクションが苦手とする応用にも用いることができる．

3.1 従来の構造リフレクションの実現法

構造リフレクションは Smalltalk[8]，ObjVlisp[4]，CLOS[11] などの言語に備えられていた．これらの言語の構造リフレクションの実装は，その言語の実行系内部に存在するデータの一部を，その実行系の上で動いているプログラムから直接変更できるような API（Application Programming Interface）を用意する，というものである．変更可能な実行系内部のデータが，クラス定義を表すものであれば，プログラムはその API を通してクラス定義を変更できることになる．たとえば，仮に C++用に構造リフレクションの機能を実装するとする．C++プログラムをコンパイルして実行すると，内部的に仮想関数表というデータが作成されるが，従来の方法は，この仮想関数表の中身を実行中にプログラムから変更できるような API を提供することで構造リフレクションを実装する，と考えられる．

しかしながら，このような従来の実装技術は Javassist に採用できない．従来の実装技術をそのまま使うと Java 仮想機械（JVM）を改造しなければならないが，Java 言語では可搬性の重要さから JVM の改造は現実的ではない．またこの技術は，JIT コンパイラが行うプログラムの静的な情報を活用した最適化を難しくする．たとえばあるクラスが `final` クラスであるか否かは，最適化において重要な情報である．もし `final` クラスが実行時に非 `final` クラスに変わる可能性があるとすると，その情報を使った最適化をあきらめるか，構造リフレクションが実行されるたびに，関係する広範なコードを再コンパイルしなおさなければならなくなる．メソッドのインライン化も同様である．インライン化されたメソッドの定義が実行時に構造リフレクションによって変更されたら，そのメソッドのすべてのコードを更新しなければならない．このためには，あるメソッドがどこでインライン化されたかをすべて記録しておかなければならず，非常に大きな記憶領域を費やすことになろう．さらに，構造リフレクションによって，クラスの親子関係が動的に変化しうるとすると，Java 言語の静的な型付けシステムは意味をなさなくなってしまう．少なくとも構造リフレクションの範囲に制限を設ける必要がある[15]．

3.2 ロード時のバイトコード変換による実現

我々は新たにロード時のバイトコード変換によって，Java 言語で構造リフレクションを実現する方法を開発した．これにより，既存の JVM をそのまま利用することが可能になり，上で述べた可搬性と最適化の問題を解決できる．Javassist は，この方式によって構造リフレクションを実現するクラスライブラリである．

Java ではプログラムのコンパイルによって得られたバイトコードはクラスごと別々に**クラスファイル**と呼ばれるファイルに納められる．Javassist は構造リフレクションによる変更を，等価なクラスファイルのバイトコード変換に翻訳することによって，構造リフレクションを実現する．変換はクラスファイルのロード時以前であれば任意の時点で行うことができるが，一般的な利用では，Javassist が提供するクラスローダを使ってロード時に変換をほどこす．変換されたクラスファイルは JVM にロードされ，その後は構造リフレクションによる変更は許されない．したがって，Javassist は様々な最適化技術が実装されている標準の JVM 上で用いることが可能である．一方，リフレクションをロード時以前に限ることで，有用性は一部損なわれる．これについては後に議論する．

4. Javassist の API

本章では Javassist が提供する API（Application Programming Interface）の概要を示す．

4.1 具現化と反映

Javassist はユーザのクラスローダとともに用いられる．Java はプログラム中で独自のクラスローダを定義し，ネットワークなど通常とは異なる供給源からクラスファイルを取得することを許している．JVM は，ユーザのクラスローダからクラスファイルの中身を `byte` 配列として受け取る．Javassist は指定された供給源からクラスファイルを読み込み，バイトコード変換をほどこした結果を `byte` 配列として返す機能を提供する．

Javassist を使用する第 1 段階は，JVM にロードするクラスファイル（に対応するクラス）を表す CtClass（compile-time class）オブジェクトを生成し，構造リフレクションを適用できるようにすることである．この段階はリフレクションの具現化（reify）処理に対応する．たとえば `stream` がクラスファイルを読み込むための入力元（InputStream）であるとすると，次のコード：

```
CtClass c = new CtClass(stream);
```

は，そのクラスファイルを表す CtClass オブジェクトを生成する．CtClass のコンストラクタには，InputStream の代わりにクラス名を String の形で渡すこともできる．その場合，Javassist はクラスパスを探してそのクラスに対応するクラスファイルを見つけ出す．

作成した CtClass オブジェクトに対して，様々なメソッドを呼び出すことで，そのクラスの定義を変更す

表 1 内観用の CtClass のメソッド
Table 1 Methods in CtClass for introspection.

メソッド		説明
String	getName()	クラス名を得る
int	getModifiers()	修飾子を得る（public など）
boolean	isInterface()	インタフェースかどうか調べる
CtClass	getSuperclass()	親クラスを得る
CtClass[]	getInterfaces()	インタフェースを得る
CtField[]	getDeclaredFields()	宣言されているフィールドを得る
CtMethod[]	getDeclaredMethods()	宣言されているメソッドを得る
CtMethod[]	getConstructors()	コンストラクタを得る

ることができる．変更された結果を反映したクラスファイルを得るには，次のようにする．

```
byte[] bcode = c.toBytecode();
```

得られたバイトコード bcode を JVM にロードすれば，CtClass オブジェクトを実際にベースレベルに反映（reflect）することができる．Class には，toBytecode() のほかに，変更を反映したクラスファイルをディスクに書き出したり，Javassist が提供するクラスローダを使って，JVM に直接ロードしたりするメソッドも用意されている．

Javassist を使うと，既存のクラスファイルを元にせず，まったく新規にクラスを作り出すこともできる．たとえば，

```
CtClass c2 = new CtNewClass();
```

は，メソッドやフィールドを持たない空のクラスを表す CtClass オブジェクトを作り，c2 に代入する．作成したオブジェクトに対して，下で説明する様々なメソッドを呼び出すことで，メソッドやフィールドを追加してゆくことができる．

4.2 内観（introspection）

クラスの内観（introspection）については，Javassist は，標準の Java リフレクション API とほぼ同等の機能を提供する．ただし newInstance() のようなメソッドはロード時には意味をなさないので，提供されない．**表 1** に，CtClass に定義されている内観のためのメソッドのうち，代表的なものを示す．

getSuperclass() や getInterfaces() が返す CtClass オブジェクトは，クラスの検索パス☆上で見つかったクラスファイルから暗黙のうちに生成される．これらのオブジェクトが表すクラスは，すでに JVM にロード済みの可能性があるため，これらのオブジェクトはつねに元のクラス定義を表し，内観は受け付けるが変更は受け付けない．変更のためには，そのクラスファイルから，別の CtClass オブジェクトを new 命令により明示的に生成しなければならない．

☆ ディレクトリ名だけでなく，URL など任意のソースを検索パスに含められるようにする仕組みが用意されている．

表 2 変更に使われるメソッド
Table 2 Methods for alteration.

CtClass のメソッド		説明
void	bePublic()	このクラスを public にする
void	notFinal()	このクラスを非 final にする
void	setName(String name)	クラス名を変更する
void	setSuperclass(CtClass)	親クラスを変更する
void	addInterfaces(CtClass[])	インタフェースを追加する
void	addConstructor(...)	コンストラクタを追加する
void	addMethod(...)	メソッドを追加する
void	addField(...)	フィールドを追加する
CtField のメソッド		**説明**
void	bePublic()	このフィールドを public にする
void	notFinal()	このフィールドを非 final にする
CtMethod のメソッド		**説明**
void	bePublic()	このメソッドをpublic にする
void	notFinal()	このメソッドを非final にする
void	instrument(...)	メソッドの中身を編集する
void	setBody(...)	メソッドの中身を置き換える
void	makeWrapped(...)	Glue code を挿入する

フィールドやメソッドの情報は，それぞれ，getDeclaredFields() により得られる CtField オブジェクト，getDeclaredMethods() により得られる CtMethod 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 と Method に相当する．現在の実装では，コンストラクタの情報は CtMethod オブジェクトにより表される．

4.3 変更（alteration）

Javassist と標準の Java リフレクション API の違いは，Javassist がクラス定義を変更するメソッドを提供していることである．しかしながら，Javassist はクラス定義の任意の変更を許すわけではない（**表 2** に代表的なものを示す）．クラス定義を変更するためのメソッドは，次に示す 3 つの設計目標を満たすように注意深く選ばれている．(1) Java のバイトコードの知識を持たなくても使えるように，プログラマにソースレベルの抽象度を提供する．(2) リフレクションによって誤ってプログラムを破壊してしまわないように補助する．(3) 構造リフレクションをできるだけ効率良く実行できるようにする．

我々は，第 1 の目標の達成基準を，Javassist の API を Java 言語（すなわちソースレベル）の語彙だけを使って記述することにおいた．Javassist API に含まれるクラスの名前やメソッド名，およびその説明には，コンスタントプールなどの Java バイトコードの語彙は使われていないので，この基準は達成できたと考える．

基準を達成するために工夫を要したのは，たとえば，メソッドを既存のクラスに追加する際のメソッドの中身の指定方法であった．生のバイトコード列を直接指定する方法では，プログラマがバイトコードの詳細を知らなければ Javassist を使えなくなる．そこで我々

は，メソッドの中身としてすでに存在する別のクラスのメソッドの中身を指定するようにした．プログラマは，あらかじめ，追加したいメソッドを別のクラスのメソッドとして Java 言語で記述，コンパイルしておき，このコンパイル済みのメソッドを，新たに追加するメソッドの中身として指定する．Javassist は指定されたメソッドのバイトコード列を，追加先のクラスに合うように修正し，追加するメソッドの中身とする．たとえば，this 変数を通じてメンバにアクセスするバイトコードは，this の型へのシンボル参照を含んでいるので，この参照は追加先のクラスに置き換えられる．これによりプログラマは，追加するメソッドの中身をソースコードで指定できるようになった．

プログラマがソースレベルの抽象度で指定したクラス定義の変更は，Javassist により自動的に等価なバイトコード変換に翻訳されてクラスファイルに反映される．この翻訳が容易であれば，プログラマにソースレベルの抽象度を提供することに，あまり実用的な意味はない．しかし，よく知られた Java のソースコードとバイトコードの構成の類似性にもかかわらず，ソースレベルの変換とバイトコードレベルの変換の間の翻訳は単純ではない．翻訳には Java バイトコードの細かい知識が必要である．

たとえば， setName() はクラスの名前を変更するが，クラス定義の整合性をとるために，同時に，そのクラスのメソッドの引数や中身などに現れる元のクラス名をすべて新しい名前に置き換える．バイトコードレベルでは，この処理は，コンスタントプール中の元クラス名を表す項目を，新しい名前で置き換えることに相当するが，場合によっては，これだけでは不足である．バイトコードレベルでは，メソッドのシグネチャを表す文字列の中にクラス名が埋め込まれている可能性があるので，クラスファイル中に含まれるすべてのシグネチャを調べ，該当するクラス名があれば変更しなければならない．バイトコードに関する細かな知識なしでは，このような変更を指定できない．

第 2 の目標は，リフレクションにより他のクラスと整合性がとれないクラスを生成してしまい，プログラムが自己破壊をおこさないよう，Javassist の API を工夫することである．この目標のため，Javassist では構造リフレクションの範囲を制限した．クラス定義を変更する Javassist の各メソッドは，バイナリ互換性[7] を保つと Java 言語の仕様が保証している変更[10] だけを行う．バイナリ互換性を保つ変更とは，残りのクラスを再コンパイルしなくても，リンク・実行が可能な変更である．たとえば Javassist では，メソッドやフィールドをクラスから除去するメソッドは提供されない．また親クラスを変更することは可能だが，任意のクラスを新しい親クラスにできるわけではない．変更後の新しい親クラスは，元の親クラスの子クラスでなければならない．つまり，元の親クラスと変更対象のクラスとの継承関係の間に，新しいクラスを挿入することしかできない．むろん，新しい親クラスは final クラスであってはならない．さらに，Javassist を使って，既存のメソッドの引数の数や型を変更することはできない．代わりに，同じ名前で異なる引数をとる新しいメソッドを追加することが推奨される．

残念ながら Java 言語のバイナリ互換性に関する仕様に厳密性はない[6]．Javassist は，リフレクションによる自己破壊に対して一定の歯止めをかけたが，より厳密な体系に基づいた自己破壊の防止法は今後の研究課題である．

4.4 新しいメンバの追加

Javassist の第 3 の目標は，構造リフレ N ションの実現に必要なバイトコード変換を高速化することである．我々は Javassist を，たとえば，モバイルエージェントシステムの中で，安全確認のコードをバイトコードに動的に挿入する処理の実装に使うことを計画している．このような応用のためには，Javassist を介してクラスをロードする際のオーバヘッドを可能な限り小さくすることが求められる．

この目標のため，Javassist はメソッドを既存のクラスに追加する際，メソッドの中身をプログラマにソースコードの形で指定させない．この機能を実装すると，ロード時にソースコードをコンパイルしなければならないからである．ソースコードをコンパイルすると実際に速度が大きく低下することは，後の 5.2 節で実験により示す．Javassist は先に述べたように，代わりに他のクラスのコンパイル済みメソッドを複製して CtMethod オブジェクトを作り，目的のクラスに追加する機能を提供する．この方式ではメソッドを事前にコンパイルしておけるので，ロード時のコンパイルが必要がなく，速度低下を避けることができる．メソッドを複製する際には，メソッド中に現れるクラス名を別のクラス名で置き換えることができる．これにより，既存の 1 つのメソッドから型が異なる複数のメソッドを動的に生成して追加することができる．

追加する CtMethod オブジェクトを操作して，メソッドの本体を glue code で包むこともできる．Glue code は，指定された型の引数列を Object 型の配列に変換して，メソッド本体に渡す働きをする．これを使うと，たとえば，次のようなメソッド：

```
Object lookup(Object[] args) {
     return hashtable.get(args[0]);
}
```

を用意しておき（ここで hashtable は， java.util.Hashtable オブジェクト），これを glue code で包むように変形して， String オブジェクトを引数にとり String オブジェクトを返す次のようなメソッド：

```
String stringLookup(String key) {
     Object[] args = { key };
     return (String)hashtable.get(args[0]);
}
```

を生成し，これを目的のクラスに追加することができる．メソッドの引数・返り値の型は， String だけでなく，任意の型を指定できる．したがって， Object 型の配列を引数にとる既存の単一のメソッドを元に，様々な型の引数をとるメソッドを動的に生成して，クラスに追加することができる．

Javassist はまた，指定された型のフィールドを表す CtField オブジェクトを生成し，クラスに追加する機能も提供する．また，追加するフィールドの初期値を指定することもできる．指定できる初期値は， int など組み込み型の定数，任意のクラスのオブジェクト，フィールドを追加したクラスのコンストラクタの何番目かの引数，などである．

4.5 メソッド本体の変更

Java バイトコードの知識なしで使えるようにするため，Javassist ではメソッド本体のバイトコードを直接変更する機能は提供されていない．代わりに，別のメソッドの中身の複製を作り，その複製で元のメソッドの中身を置き換えることができる．

また，メソッド本体の中に現れるメンバ・アクセス用の演算子を型ごとに異なる静的メソッドの呼び出しで置き換えることもできる．このアイデアは，C++ 言語の演算子の多重定義からきている．これにより new や .（メンバアクセス）のような演算子の振舞いを変更することができる．ただし演算子の置換では，型以外の文脈に応じて呼び出す静的メソッドを変えることはできない．

演算子の置換を行う CodeConverter オブジェクトのメソッドを**表 3** に示す．メンバ・アクセス用の演算子を静的メソッドの呼び出しに置換するほか，元のフィールドやメソッドとは異なるフィールドやメソッドのアクセスに変えることもできる．

4.6 Javassist の有用性

Javassist は，ロード時（またはそれ以前）にのみ構造リフレクションを可能にする．また，上で述べた

表 3 CodeConverter のメソッド
Table 3 Methods in CodeConverter .

メソッド	説明
void replaceNew()	new 式を静的メソッド呼び出しに置き換える
void replaceFieldRead()	フィールド読み込みの式を静的メソッド呼び出しに置き換える
void replaceFieldWrite()	フィールド書き込みの式を静的メソッド呼び出しに置き換える
void redirectFieldAccess()	フィールド・アクセスの式を異なるフィールドアクセスに置き換える
void redirectFieldAccess()	メソッド呼び出しを異なるメソッド呼び出しに置き換える

ような目標を達成するため，構造リフレクションによる変更の範囲を制限している．しかしながら我々は，Javassist の機能の範囲内で，Kava[17] と同等な実装方法・実行効率の動作リフレクションを，Javassist の上位層のライブラリとして実現できることを確かめた．プログラムの大きさは約 750 行ほど（コメントを含む）であった．

このとき Java バイトコードの知識は必要ない．これは Javassist が間接的に動作リフレクションの機能を包含することを意味する．Kava による動作リフレクションの実現では，リフレクションの対象となるオブジェクトのクラス定義を変更して，メタオブジェクトへのリンクを挿入すること，対象となるオブジェクトのフィールドやメソッドにアクセスする側のコードを変更して，メタオブジェクトにアクセスが通知するための「フック」を挿入すること，が必要である．文献 2) で概要を述べたとおり，これらの変更はすべて Javassist の API を使って直接可能である．

我々はまた，分散を考慮せずに書かれたプログラムを変換して，たとえば GUI（Graphical User Interface）に関連するオブジェクトだけを遠隔ホストに分散させて動かせるようにするシステム，Addistant を Javassist を使って開発した[16]．2 章で述べたように，分散オブジェクトの実現はプロキシのクラスの定義などを必要とするので，動作リフレクションを使うと，仮に実装が可能であっても，分かりにくいプログラミング技巧が必要であった．しかしながら，Javassist を使うとそれらの機能が直接提供されるので，複雑なプログラム技巧は必要なかった．さらに複数の分散オブジェクトの実現方法を Javassist の機能の範囲内で実装することが可能であったので，Addistant は複数の実現方法を組み合わせた変換を行えるようになった．

我々は，これ以外にも，クラスの定義を実行時に動的に拡張できるようにするための実行時システムも Javassist で作成できることを確かめた[19]．

5. 関連研究

5.1 Java バイトコード変換器

ロード時にバイトコードを変換するためのクラスライブラリとしては，JavaClass API[5) や JOIE[3) が知られている．このようなクラスライブラリと異なり，Javassist はソースレベルの抽象モデルを提供するため，利用者にクラスファイルの内部データ形式や Java バイトコードの知識がなくても利用できるのが特徴である．また Javassist の API は，バイナリ互換性を維持するような変換しか許さないため，JVM のバイトコード検証器に拒絶されてしまうようなクラスファイルを誤って生成しにくいように工夫されている．

リフレクションによるプログラムの自己破壊の危険性は，従来から指摘されてきたが，解決策についてはほとんど研究されていない[12)．たとえば，ソースコード変換器で構造リフレクションを実現する OpenC++[1) では，文法的に誤ったプログラムに変換してしまう可能性さえある．OpenJava[20) では，そのような誤った変換はできないが，Javassist と異なり，バイナリ互換性を失うような変換を防ぐ工夫はなされていない．

Javassist は Java で構造リフレクションを最初に可能にしたわけではない．たとえば，Kirby らは実行時に新しいクラスを動的に定義する機構を提案している[13)．彼らが，言語リフレクション（linguistic reflection）と呼ぶこの機構は，構造リフレクションの一種である．しかし，既存のクラス定義をロード時に変更することはできない．彼らの機構は，新しいクラスのソースコードを生成し，外部のコンパイラによってコンパイルし，生成されたクラスファイルをロードすることしかできない．彼らは，この機構を使うことにより，実行時の状況に最適化されたクラスを動的に生成し，全体の実行速度を改善できたと報告している．

5.2 OpenJava

OpenJava[20) は構造リフレクションを Java 言語で行うための処理系で，コンパイル時リフレクション[1) の研究に基づいて，我々が過去ノ開発したものである．Javassist が構造リフレクションによる変更をロード時のバイトコード変換で実現するのに対し，OpenJava はコンパイル時のソースコード変換で実現する．ソースコードを対象としているので，OpenJava では，構造リフレクションの枠組みの中で構文拡張を行うこともできる．たとえば，クラス宣言の構文を拡張して，既存の `public` などのほかに，独自の修飾子をクラスに付加できるようにすることができる．

一方，ソースコード変換を行うため，OpenJava は処理されるクラスそれぞれのソースファイルを必要とする．それに対し，Javassist はコンパイルによって得られたクラスファイル（バイナリファイル）しか必要としない．クラスが第三者によって提供される場合，ソースファイルがつねに手に入るとは限らないため，この点で Javassist は有利である．ソースコード変換は，また，実行時のオーバヘッドが大きい．クラスファイルではなく，ソースコードを扱うので，構文解析など，リフレクションとは直接関係のない処理に大きな時間をとられてしまう．

OpenJava と Javassist の実行時オーバヘッドを比較するため，我々は実験を行った．この実験では，様々なクラスについて，OpenJava と Javassist が，ソースプログラムまたはクラスファイルを読み込み，内部で使用するデータ構造に変換し，再びソースプログラムまたはクラスファイルに戻す時間を測定した．これはリフレクションの処理のうち，具現化と反映にかかる時間を測定したことになる．

我々は SPECjvm^{TM}98 を構成するクラスファイルのうち，ソースコードが提供されている 74 個を実験に用いた．また各ソースコードを Java コンパイラでバイトコードにコンパイルする時間も測定した．OpenJava の出力はソースコードであるので，JVM にロードする前にコンパイルしなければならない．したがって，OpenJava が構造リフレクションの処理に必要とする実際の時間は，OpenJava 自体による処理時間にコンパイル時間を加算したものである．

実験に用いた計算機は，Sun Ultra10（UltraSPARC IIi 440 MHz，Solaris 7）であり，JVM として Sun JDK 1.3.0 付属の HotSpot^{TM} Client VM を用いた．コンパイラとしては，Sun JDK 1.3.0 付属の javac コンパイラと IBM の Jikes を用いた．javac は Java 言語で記述されているが，Jikes はネイティブコードで記述されている．

図 1 に測定結果を示す．OpenJava と Javassist の測定では，クラスのロード時間を除くため，同じ処理を 2 回繰り返し，2 回目の処理時間を測定した．これを 3 回繰り返し，最少のものを選んだ．Javassist はすべて Java で記述されているにもかかわらず，処理時間が非常に短いことが分かる．34 KB のクラスファイルの処理時間も 0.7 秒であった．1 KB 程度のクラスファイルならば，0.03 秒程度である．OpenJava の処理時間は，同じような大きさのクラスでも大きく処理時間が異なることがある．これは，ソースコード中に現れるクラス名が，どのパッケージのクラスを意味するのかを決定するのにかかる時間が大きいためであ

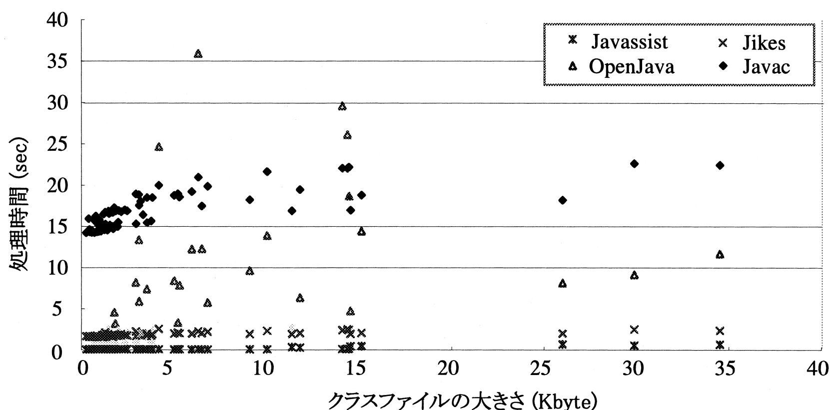

図 1 具現化と反映にかかわる処理時間
Fig. 1 Execution time of reification and reflection.

る．クラスによっては，多数のパッケージの検索を必要とし，名前解決に時間がかかる．クラスファイル中では，すべてのクラス名は，どのパッケージのクラスを意味するか明示されているので，Javassist の処理時間は比較的一定である．

6. まとめ

本論文では，Java のリフレクション API を拡張し，より強力なリフレクションの機能を実現するクラスライブラリ Javassist について述べた．他の類似システムが採用する動作リフレクションは，分散オブジェクトなどの実装に必要な新しいクラスを定義する機能などを提供していないため，そのような言語拡張の支援には適切でないことを指摘し，Javassist では代わりに構造リフレクションを採用したことを述べた．これにより，Javassist は，分かりにくいプログラミング技巧なしに新しいクラスを定義する機能などを提供することができた．一方，Javassist の機能の範囲内で，動作リフレクションを上位層のライブラリとして実現することも可能である．したがって Javassist は，動作リフレクションの機能を間接的に提供し，それに加えて新しいクラスを定義する機能などを提供するので，従来の動作リフレクションに基づいたシステムよりも強力なリフレクション機能を有するといえる．

既存の JVM を改変することなく構造リフレクションを可能にし，可搬性と実行時性能の問題を回避するために，本論文では，ロード時のバイトコード変換によって構造リフレクションを実現する方法を新たに提案した．また，従来の低レベルなバイトコード変換を支援するクラスライブラリと違い，Javassist はソースレベルの抽象モデルを利用者に提供していることを述べた．これにより，ユーザは Java バイトコードについての知識がなくても Javassist を利用することができる．さらに，Javassist はバイナリ互換性を維持する変換しか許さないため，誤った変換によってクラス定義に不整合を招きにくいようになっている．最後に，OpenJava[20] のようなソースコード変換によって構造リフレクションを実現するシステムに比べ，Javassist は変換を高速に実行することを実験で示した．

Javassist はリフレクションをロード時（またはそれ以前）に制限し，可能な変換の種類も絞っている．このため Javassist の応用範囲も狭められるが，Javassist の機能の範囲内で動作リフレクションが実現可能なため，少なくとも動作リフレクションを採用したシステムと同等の有用性は Javassist も持っているといえる．それに加え，我々は，分散オブジェクトの実装や，クラスの定義を動的に拡張する機能の実装にも，Javassist を用いることができることを実際に確かめた．

謝辞 本研究の一部は，文部科学省科学研究費の補助を受けている．

参考文献

1) Chiba, S.: A Metaobject Protocol for C++, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices, Vol.30, No.10, pp.285–299, ACM (1995).
2) Chiba, S.: Load-time structural reflection in Java, *ECOOP 2000*, LNCS 1850, pp.313–336, Springer-Verlag (2000).
3) Cohen, G.A., Chase, J.S. and Kaminsky, D.L.: Automatic Program Transformation with JOIE, *USENIX Annual Technical Conference '98* (1998).
4) Cointe, P.: Metaclasses are first class: The ObjVlisp model, *Proc. ACM Conf. on Object-*

Oriented Programming Systems, Languages, and Applications, pp.156–167 (1987).

5) Dahm, M.: Byte Code Engineering with the JavaClass API, Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin (1999).

6) Drossopoulou, S., Wragg, D. and Eisenbach, S.: What is Java Binary Compatibility?, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp.341–358 (1998).

7) Forman, I., Conner, M., Danforth, S. and Raper, L.: Release-to-Release Binary Compatibility in SOM, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp.426–438 (1995).

8) Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (1983).

9) Golm, M. and Kleinöder, J.: Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible, *Proc. Reflection '99*, LNCS 1616, pp.22–39, Springer (1999).

10) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification*, 2nd edition, Addison-Wesley (2000).

11) Kiczales, G., des Rivières, J. and Bobrow, D.G.: *The Art of the Metaobject Protocol*, The MIT Press (1991).

12) Kiczales, G.: Foil for the Workshop on Open Implementation, internet publication (`http://www.parc.xerox.com`), Xerox PARC (1994).

13) Kirby, G., Morrison, R. and Stemple, D.: Linguistic Reflection in Java, *Software — Practice and Experience*, Vol.28, No.10, pp.1045–1077 (1998).

14) Kleinöder, J. and Golm, M.: MetaJava: An Efficient Run-Time Meta Architecture for Java, *Proc. International Workshop on Object Orientation in Operating Systems* (*IWOOS'96*), IEEE (1996).

15) Malabarba, S., Pandey, R., Gragg, J., Barr, E. and Barnes, J.F.: Runtime Support for Type-Safe Dynamic Java Classes, *ECOOP 2000*, LNCS 1850, pp.337–361, Springer-Verlag (2000).

16) Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Execution of "Legacy" Java Software, *ECOOP 2001*, LNCS 2072, pp.236–255, Springer-Verlag (2001).

17) Welch, I. and Stroud, R.: From Dalang to Kava — The Evolution of a Reflective Java Extension, *Proc. Reflection '99*, LNCS 1616, pp.2–21, Springer (1999).

18) Wu, Z.: Reflective Java and A Reflective-Component-Based Transaction Architecture, *Proc. OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Fabre, J.-C. and Chiba, S. (Eds.) (1998).

19) 千葉　滋：クラス定義の発展と自己反映計算によるその対処，コンピュータソフトウェア，Vol.17, No.2, pp.35–39 (2000).

20) 立堀道昭，千葉　滋，板野肯三：クラスオブジェクトを用いたJava言語用マクロ処理系，情報処理学会論文誌，Vol.41, No.8, pp.2327–2338 (2000).

(平成13年1月29日受付)
(平成13年9月12日採録)

千葉　　滋（正会員）

1968年生．1991年東京大学理学部情報科学科卒業．1996〜97年同大学助手．1997〜2001年筑波大学電子・情報工学系講師．2001年より東京工業大学情報理工学研究科講師．博士（理学）．言語処理系およびオペレーティングシステム等システムソフトウェアの研究に従事．日本ソフトウェア科学会，ACM各会員．

立堀　道昭（学生会員）

1974年生．1997年筑波大学第三学群情報学類卒業．1997年より同大学院博士課程工学研究科．1999年同大学院工学修士号取得．言語処理系，プログラミング，システムソフトウェアに関する研究に従事．日本ソフトウェア科学会，ACM各学生会員．