Javaプログラムの種々の実行時情報の効果的取得方法

楓 基靖 [*1]　林 晃一郎 [*1]　樋田 洋明 [*2]　吉村 陵二 [*2]　真野 芳久 [*2]

広く利用されるようになっている Java プログラムは解読され盗用される危険性が高い。盗用の発見または盗用の事実の立証を目的とする動的バースマークはプログラムの実行時情報から形成されるが、有効な動的バースマークを構成するためには種々の実行時情報を効果的に取得できる必要がある。本稿では最初に、デバッグ情報を含むクラスファイルに対して局所変数の実行時情報の取得方法を述べ、次にデバッグ情報を含まないクラスファイルに対してもデバッグ情報を含むクラスファイルとほぼ同様の実行時情報を得る方法について述べる。JDI と BCEL 機能を有効に活用することによって、局所配列の値の変遷などの実行時情報を得る方法についても述べる。ここで述べた方法を使うことによって、デバッグ情報を含まない流通ソフトウェアに対しても多くの実行時情報を得ることができる。

1　はじめに

近年インターネットが急速に普及し、それに伴いプラットフォームに依存しない Java 言語がその特性から広く利用されるようになっている。Java プログラムの実行形式であるクラスファイルには一部のソースコード情報が含まれており、また高性能な逆コンパイラなどの解析ツールが作成されているため、復元・解読され改変・盗用などの対象となる危険性が高い。

盗用の発見または盗用の事実の立証を目的とする技術として、電子透かし、フィンガープリント、バースマークなどがある。いずれも実行動作を前提としない静的技術と実行動作を前提とする動的技術に分類できる。ここで述べる実行時情報取得方法は、プログラムの実行時情報そのものの特徴を表現したものでありその一致をもって盗用を発見する技術である動的バースマークへの応用を主な目的としている。

動的バースマークの構成と評価に関しては、[1] や [2] などの研究がある。また我々も、Java プログラムのための動的バースマークを系統的に構成する方法とその実現例の研究開発 [3][4] を進めている。動的バースマークはプログラムの実行時情報から形成されるが、有効な動的バースマークを構成するためには種々の実行時情報を効果的に取得できる必要がある。

プログラム実行時の動作情報を得るためのツールとして、種々のデバッガ、トレーサなどが開発されてきた。これらはプログラム開発を主な目的とし、プラットフォームに依存するものが多い。また、有用な情報を得るためにはソースコードを必要としており、実行形式ファイルの状態での盗用検査を主な目的とする動的バースマークのための道具としては適さない。

Java プログラムのための実行時情報取得ツールとして、DataExtractor[5]、AddTracer[6] などがある。Java プログラムでは局所変数や各メソッド内部での実行時の挙動を詳細に観測することが一般に困難である。[5]、[6] はこの問題に対し後述の方法で解決しようとしている。本稿では、Java プログラムを対象とし、Java の標準的な開発環境である J2SDK を前提にして、別のアプローチによる実行時情報の効果的な取得方法について述べる。

Java の実行時情報の取得には、JPDA (Java Platform Debugger Architecture)[7] が提供する Java 用デバッガ開発パッケージ JDI (Java Debug Interface) を利用できる。しかし、対象とするクラスファイル内に含まれる情報を利用する JPDA の制約上、実行時情報を効果的に取得することができない場合もある。

他方、ソースコードがない状態でも BCEL (Byte Code Engineering Library)[8] を使って Java クラスファイルを直接操作し変更することは可能である。

本稿では JDI と BCEL を有効に活用することで、Java 言語に対する動的バースマーク技術への適用を目的とする実行時情報の効果的な取得方法について述べる。

2　Java クラスファイル

2.1　Java クラスファイルの構成

Java 言語で書かれたプログラムはクラスファイルにコンパイルされ、Java プログラムのための仮想機械 JavaVM

[*1] 南山大学 数理情報研究科
[*2] 南山大学 数理情報学部 情報通信学科

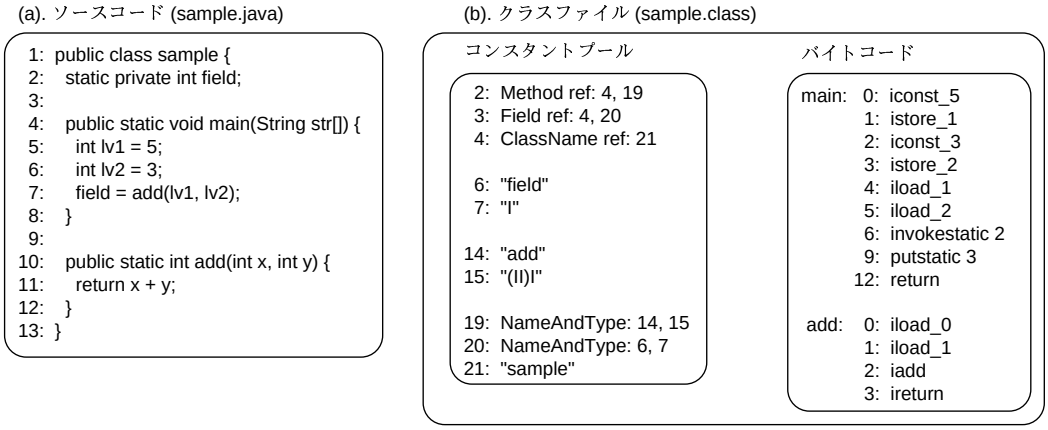

図 1　クラスファイルの例 (一部抜粋)

によって実行される。クラスファイルは、ソースコード内の宣言名や型などを保持しているコンスタントプール、プログラム動作のための命令列であるバイトコード部分などから構成される。

コンスタントプールには、そのプログラム中で使われているメソッド名などの文字列、変数の型、コンスタントプール要素の位置などが保持される。コンスタントプールは配列で表現され、保持される値はバイトコード部分やコンスタントプール自身から参照される。参照される場合は配列の添字値が使われる。

バイトコードは JavaVM のための命令語であり、約 200 種類用意されている。これらはメソッド毎に生成されるオペランド・スタックに対する操作などを行ない、1 つの命令はオペコードと 0 個以上のオペランドを持つ。

コンスタントプールとバイトコードの例を**図 1** に示す。(a) は元となったソースコードである。(b) のコンスタントプール中の左側の数値は配列表現における添字値を表わす。(b) のバイトコードの 2 列目の数値はメソッド先頭からの offset を表わす。offset 6 と 9 にあるオペランドの値はそれぞれコンスタントプールへの参照であり、これらを辿ることでメソッド名やフィールド名を得ることができる。

2.2　デバッグ情報

Java プログラムのデバッグを支援することを主な目的として、次の 3 種類のデバッグ情報がクラスファイル内に付加されることがある。なお、メソッド内で宣言される名前は局所的、そうでない名前は大域的と呼ぶことにする。

- ソースファイル情報 (SourceFile)
  クラス毎に存在し、そのクラスの元となる Java ソースファイルの情報を保持する
- 局所変数表 (LocalVariableTable)
  メソッド毎に存在し、各局所変数は Index, Name, Type, From, To で構成される[*3]。Index は変数が格納される局所変数領域内の配列への添字、Name と Type はソースファイル内での名前と型、From-To はその変数の値が保持されているとして扱われる範囲 (バイトコードの offset 値で表現) を表わす。
- 行番号表 (LineNumberTable)
  メソッド毎に存在し、ソースコードの行番号 (SourceIndex) とバイトコードの offset (CodeIndex) との対応情報 (行情報と呼ぶ) を保持する

これらのデバッグ情報のクラスファイルへの付加は、コンパイル時の “-g” オプションによって制御できる[*4]。デバッグ情報は実行する上では必要でないため、最終の配布プログラム版ではメモリの最適化などの目的で削られている可能性が高い。

図 2 に、デバッグ情報を付加するように図 1 のソースコードをコンパイルした時の局所変数表と行番号表のデバッグ情報を示す。

3　BCEL と JDI の特性と利用

3.1　BCEL

BCEL を利用することで、クラスファイルを容易に解析、変更することができる。クラスファイルを直接操作することで、メソッドや特定の処理の追加、クラスファイル

[*3] Java 仮想マシン仕様 [9] では、To に相当する情報は位置ではなく先頭からの長さである。ここでは理解性を高めるため終了位置で表現している。

[*4] ソースファイル情報と行番号表についてはデフォルトで付加されるが、付加しないこともできる。局所変数表については明示的にオプション指定することで付加される。

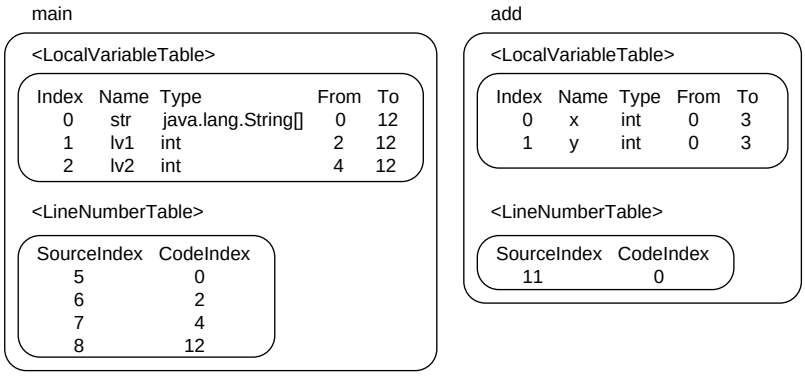

図 2　局所変数表と行番号表のデバッグ情報の例

のゼロからの生成なども可能である。逆に、クラスファイル中で不要な部分を削除する場合にも使うことができ、メモリ最適化のためのデバッグ情報の削除がその例となる。

BCEL を利用して実行時情報を得るツールの例に AddTracer[6] がある。これは、特定の動作命令の直前あるいは直後にその動作に関連する情報の出力命令をクラスファイルに付加することで、実行時情報を出力させる。例えば、アクセスされた局所変数の情報を得たい場合には、ロード、ストアなどのバイトコード命令の直後にアクセスされた局所変数の情報を出力する命令を付加する。本来の実行には関係ない命令を挿入することになるので、プログラムの仕様に影響を与えないように、またベリファイアのチェックに合格するように、スタック操作などのコードの追加をする必要がある。

3.2　JDI

JPDA を構成する要素の 1 つである JDI は Pure Java で記述されている Java インタフェースで、デバッガの実装を主な目的として提供されている。JDI では、デバッグ対象の JavaVM と、その実行をモニタするデバッグ側の JavaVM とを接続することでデバッグに必要な情報を得る。

JDI を利用することで、デバッグ対象 JavaVM の以下の動作情報 (イベント) を取得することができ、イベント発生時にはスレッドを停止させることができる。

- ブレイクポイント (BreakpointEvent)：ソースコード行単位で予め設定された実行位置の直前で発生
- ステップ実行 (StepEvent)：ソースコード行単位かバイトコード単位で発生
- クラスの準備、破棄：クラスを使う前段階のクラスロード実行とクラスアンロード実行時に発生
- 例外：例外通知の発生
- メソッドの開始、終了：メソッドの開始時と終了時に発生 (ユーザ定義メソッドと標準 API メソッドの区別はない)
- ウォッチポイント (フィールドアクセス)：フィールド値の参照・更新時に発生
- スレッドの開始、終了：スレッドの開始時と終了時に発生

ただし、ブレイクポイント、ステップ実行のイベントのためには行番号表の情報が必要である。

JDI を利用する場合、デバッグ側では必要なイベントを要求する。JDI ではデバッグ対象プログラムの実行中にこれらのイベントが発生した場合、そのイベントをイベントキューに保持する。要求されたイベントがキューから取り出され、デバッグ側に渡される。従って、これらのイベントに直接関連する情報については JDI をそのまま利用することで容易に取得することができる。

これらの情報に加え、プログラムが動作する上で必要不可欠であり個々のプログラムの特徴をより効果的に取得できる情報と考えられる局所変数、配列、クラスインスタンスに関する実行時情報の取得も望まれる。これらの情報は動的バースマークのための基本情報として有用と考えられるにも関わらず、局所変数に関する情報については、クラスファイル中に局所変数表のデバッグ情報が付加されていない場合は取得することができない。

JDI を利用して実行時情報を得るツールの例に DataExtractor[5] がある。これは耐タンパー性の検証のために実行時の全情報を出力することを目的として開発された。JPDA の制約から局所変数表のデバッグ情報のないクラスファイルでは局所変数にアクセスできないため、DataExtractor では JavaVM でステップ実行される度にオペランドスタックとヒープエリアを参照することで局所

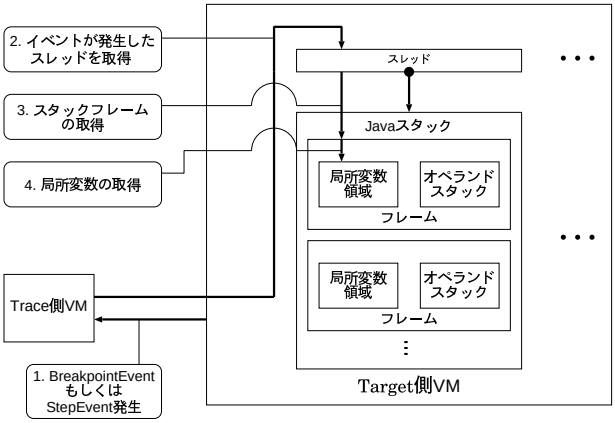

図 3　局所変数領域アクセス法の概要

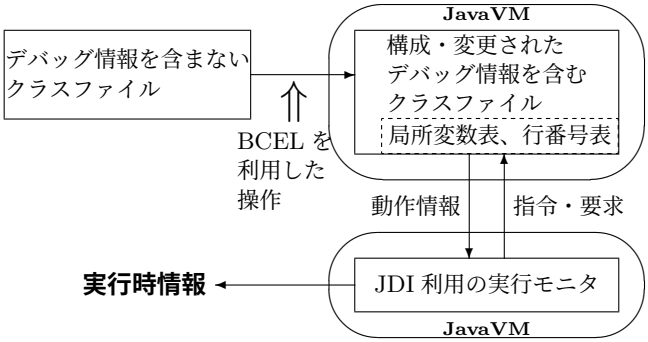

図 4　デバッグ情報の構成・変更による実行時情報取得手法の概念図

変数に関する情報の取得を実現している。しかし、JPDA を拡張して実現している制約上 J2SDK1.4 以降の環境での動作をサポートしていない問題点がある。

標準に付属されている Java デバッガ JDB においても、局所変数にアクセスする場合には局所変数表のデバッグ情報が必要である。また、対話型デバッガであるのでプログラム実行開始から終了までの一連の実行時情報をすべて取得するのには適していない。

このように、DataExtractor、JDB、及び JDI によって実現されるデバッガは、クラスファイルに局所変数表がない場合には局所変数へのアクセスができないという JPDA の制約からの影響を受けている。既に述べたように、動的バースマークの検証において対象となるクラスファイルの多くは局所変数表を持たないことに注意しなければならない。

4　デバッグ情報が存在する時の局所変数に関する実行時情報取得

本節では、局所変数表および行番号表のデバッグ情報が存在することを前提として、局所変数の実行時情報の取得方法について述べる。

JavaVM のスタックフレーム内には局所変数及び引数に加えてオペランドスタックが置かれているが、JDI は Java スレッドのオペランドスタックにアクセスするインタフェースを提供していない。しかし、局所変数領域はブレイクポイントまたはステップ実行を利用することによってアクセス可能である。局所変数領域へのアクセス方法の概要を**図 3** に示す。

取得までの流れは

1. BreakpointEvent もしくは StepEvent の捕捉
2. 捕捉したイベントを生成したスレッドの取得
3. スレッド内スタックフレームの取得
4. フレーム内の局所変数領域へアクセス

となる。

BreakpointEvent によるプログラム一時停止を行なう場合、対象プログラム内で発生するクラス準備イベントを捕捉する。その際にロードされたクラス内のメソッド全行に対してある種のブレイクポイントを設定することによって局所変数値の代入系列がトレース可能となる。

StepEvent によるプログラム一時停止を行なう場合は、予めターゲット JavaVM 内で起動中のすべてのスレッドを取得する。その中から main メソッドを実行しているスレッドに StepEvent を生成することによって main メソッド開始時から終了までステップ実行が可能となる。

5　デバッグ情報の構成・変更による実行時情報取得

本節では、局所変数表と行番号表のデバッグ情報を構成・変更することによって、JDI の持つ機能を単純に用いることで局所変数の実行時情報を取得できること、デフォルトとは異なるタイミングでの実行時情報を取得できることを述べる。**図 4** にその手法の概念図を示す。しかし、ここで構成されるデバッグ情報は、元のソースコード情報を反映していないものであり、得られる実行時情報の有効性について検討する必要がある。以下では、デバッグ情報の構成・変更手法の詳細、得られる実行時情報の有効性の検討について述べる。

5.1　局所変数表の構成

局所変数表の存在しないクラスファイルを BCEL で操作し、局所変数表を構成する方法を検討した。

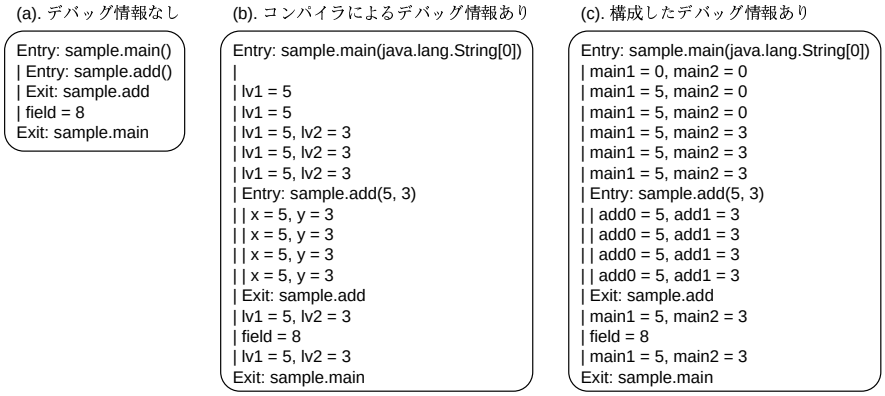

図 6　局所変数の実行時情報の取得結果

main

Index	Name	Type	From	To
0	str	java.lang.String[]	0	12
1	main1	int	0	12
2	main2	int	0	12

add

Index	Name	Type	From	To
0	add1	int	0	3
1	add2	int	0	3

図 5　構成した局所変数表

Index 情報については、クラスファイル内に記述されている局所変数の個数を入手し、0〜局所変数の個数-1 を用いる。

Name 情報については、元プログラムにある名前への復元、あるいは意味を表わす名前の付加は不可能である。ここでは、有効範囲を表わすことも兼ねて、メソッド名を先頭に付ける系統的な名前を与える。

Type 情報については、その局所変数を使うバイトコードから類推できる型を与える。iload, istore, iconst などは int 型の変数であることを類推させる。例えば、“iload 7” 命令が存在する場合、Index 7 の局所変数の Type は int 型であると判断できる。

必ずしも元プログラムと同じ正確な型を類推できるわけではないが、プリミティブ型の場合は Java のワイドニング変換の性質により情報欠損なく値を取り出すことができる。参照型の場合は Object 型と設定することで、少なくとも参照値を取り出すことはできる。

なお、Name と Type の情報として実際に局所変数表に存在するのはコンスタントプールへの参照である。Name の文字列と Type の型をコンスタントプールに追加し、その添字値を局所変数表に入れる。

From-To 情報については、本来は値が格納されてからメソッドの終了までであるが、特に実行への影響はないためメソッドの全範囲の情報を与える。

Java バイトコードはロード時にベリファイアのチェックを受け設定値にはいくつかの制約はあるが、プログラムの実行と直接関係ない情報であるため、From>To のような明らかな矛盾である場合を除き厳しいチェックはしないようである。

この構成によって、局所変数表を持たないクラスファイルから局所変数に関する情報も含めて実行時情報を取得することが可能となった。図 1 (a) のプログラムに対して構成した局所変数表の例を**図 5** に示す。また、局所変数表の違いによる実行時情報の相違結果を**図 6** に示す (Str に関する出力は紙面の都合上省いている)。図 6 の (a) はデバッグ情報を含まない場合、(b) はコンパイラによって生成されるデバッグ情報 (図 2) を用いた場合、(c) は前述の方法で構成したデバッグ情報を用いた場合である。

図 6 は各バイトコードの実行毎にトレースしたもので、| はメソッド呼び出しの深さを表わしている。図 6 の (b) と (c) の実行時情報を比較すると、以下の相違がある。

- 名前の相違：付加された名前は機械的に生成されており、意味を表現していないが、特定のメソッドに限定する実行時情報の抽出などでは機械的な処理が容易になる。
- 出現開始時期の相違：実際には宣言されて以降が有効範囲だが、付加された有効範囲はメソッド全体としている。未定義状態時の仮の値も実行時情報として出力されるが、重要な相違とは言えない。なお、何らかの解析の結果、特定の局所変数についての実

表 1　行番号表における行情報の例

	行情報	ソースコード	バイトコード
main	(5, 0)	5 lv1 = 5	0 iconst_5
			1 istore_1
	(6, 2)	6 lv2 = 3	2 iconst_3
			3 istore_2
	(7, 4)	7 field =	4 iload_1
		add(lv1, lv2)	5 iload_2
			6 invokestatic 2
			9 putstatic 3
	(8, 12)	8	12 return
add	(11, 0)	11 return x+y;	0 iload_0
			1 iload_1
			2 iadd
			3 ireturn

行時情報の範囲を空にするあるいは限定できたのであれば、有効範囲を操作することで実行時情報出力を制御できる。

5.2　行番号表の構成・変更

コンパイル時に構成される行番号表に含まれる各行情報は、ソースコード行の行番号とそれに対応する最初のバイトコードの offset との組である。図 1 のプログラム例に対する行番号表は図 2 に示されている通りであるが、これはソースコード行とバイトコード offset の対応を示す**表 1** (ソースコード、バイトコード各行先頭の数値がそれぞれ行番号、offset を表わす) から理解することができる。

JDI で局所変数を取得するには、実行中のプログラムをそれぞれ取得したい場所で一時停止させる必要がある。プログラムを一時停止させることのできる位置は大きく分けて次の 3 つである。

- JDI の定めるイベント発生時
- ソースコード行単位
- バイトコード単位

ソースコード行単位で一時停止させる場合は、行番号表に含まれる行情報のバイトコード位置で実行停止が起こる。行番号表はこの主のブレイクポイントの設定目的で使われる。行番号表がない場合には JDI はブレイクポイント情報を捕捉する事ができず、ソースコード行単位でのステップ実行も不可能となる。表 1 の行情報の状況でソースコード行単位での一時停止の場合、main メソッドでは offset が 0, 2, 4, 12、add メソッドでは offset が 0 のバイトコードで停止する。この場合の実行時情報を**図 7** に示す。

図 6 の実行時情報はバイトコード単位で一時停止させて得たものであったが、図 6 と図 7 は同じ実行過程を異なるタイミングで見ているものである。行番号表を変更することで、注目するタイミングに応じた実行時情報の取得ができると言える。

```
Entry: sample.main(java.lang.String[0])
| main1 = 5, main2 = 0
| main1 = 5, main2 = 3
| Entry: sample.add(5, 3)
| | add1 = 5, add2 = 3
| Exit: sample.add
| field = 8
| main1 = 5, main2 = 3
Exit: sample.main
```

図 7　ソースコード行単位での停止による取得結果

```
Entry: sample.main(java.lang.String[0])
| main1 = 5, main2 = 0
| main1 = 5, main2 = 3
| Entry: sample.add(5, 3)
| Exit: sample.add
| field = 8
Exit: sample.main
```

図 8　行番号表変更後の取得結果

動的バースマークへの応用を考えた場合、局所変数情報を得たい場所として例えば値の変更時がある。これは、行番号表に含まれる行情報をバイトコードの store 命令の位置を持つものにすることで可能となる。表 1 の例を使うと、行番号表を $\{(0,1),(1,3)\}$ として、ソースコード行単位で停止させる方式で実行させればよい。そのようにして得られる実行時情報は**図 8** となり、局所変数が変更された時のみ取得されている。このように行情報を変更することで、変数が参照されるタイミングなどの他の要求に対しても対応可能である。

行番号表に対するベリファイアのチェックは厳しくないが、バイトコードの offset 値は有効な値であることが要求される。

6　JDI の応用としての配列の実行時情報取得

対象プログラム内に配列情報が存在する場合、配列であるという情報は容易に得られるが、その配列の各要素の値までアクセスするには多少の工夫が必要である。配列の各要素値取得方法の概要を**図 9** に示す。

大域配列の場合、まずフィールドの現在値を取得する。図 9 における ModificationWatchpointEvent.valueToBe( ) は変更されたフィールド変数値、AccessWatchpointEvent.valueCurrent( ) はアクセスされたフィールド値の取得を示す。そのフィールドが配列オブジェクトであるか否かを検査し、配列オブジェクトであれば配列オブジェクト要素へのアクセスを提供する

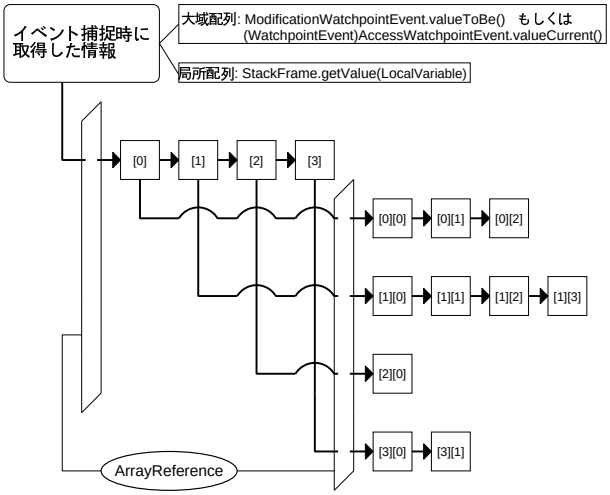

図 9　配列の各要素値取得方法の概要

インタフェースである ArrayReference を利用する。ArrayReference を介することにより配列全要素をリストとして取得することができる。図 9 ではフィールドを検査した後、配列への参照情報から配列全要素をリストオブジェクトとして取得している。要素取得後はリストを辿ることにより、全要素値が取得可能となる。

1 次元の配列の場合、この検査を 1 回のみ行なうだけで配列の各要素値は取得できる。2 次元以上の配列の場合はこのインタフェースを次元数分適用することになる。すなわち、取得要素が配列リストを保持していなければ要素値の取得を行ない、保持していれば配列リストに変換するメソッドを介して次の次元の配列情報を取得する。局所配列の場合は、図 9 に示すように、大域配列においてフィールド値を取得する部分を局所変数取得の処理に変更することで同様に取得できる。

またオブジェクトの配列に関しては、上記の配列オブジェクト取得後にオブジェクトのクラスを調べるインタフェースを介してそのオブジェクトのクラス情報を取得できる。そのクラスから更にそのクラスに属するメソッド、フィールド情報が取得可能となる。

7　実行時間の増加についての調査と検討

局所変数の実行時情報の取得では、指定したバイトコード行に対してブレイクポイントを設定し BreakpointEvent または StepEvent を介して実行時情報を得ているため、各イベントで発生するオーバーヘッドが懸念される。特に、JDI によって局所変数にアクセスする場合には、実行中のスレッドを一時中断する必要があるため実行時間の増加が問題となることも起こりうる。

```
public class sample {
  public static void main(String str[]) {
    int size = Integer.parseInt(str[0]);
    for (int i = 0; i < size; i++) {
      method_1(i);
    }
  }
  static int method_1(int i) {
    return i + 1;
  }
}
```

図 10　オーバーヘッド調査のための実験プログラム

この種のオーバーヘッドを見積もるために、中断のオーバーヘッドおよび局所変数へのアクセスという基本動作に要する時間を計測することとした。**図 10** のプログラムを用意し、メソッド開始イベント MethodEntryEvent を要求して実験を行なった。

メソッド呼出しの回数 (a) を変化させ、スレッドの中断には SUSPEND_ALL [*5]を使い、それぞれについて中断なし (b、イベント捕捉時にスレッドを中断させない)、中断あり (c、イベント捕捉時にスレッドを中断させるがその他の処理は行なわない)、中断 + 参照 (d、スレッドを中断しスタックフレームにアクセスして局所変数値を取り出す) の各条件で得た実験結果を**表 2** に示す (5 回の実行の平均時間)。中断増加は $(c-b)/a$、参照増加は $(d-c)/a$ を計算したものであり、それぞれ 1 回のイベント発生によってスレッド中断のためのオーバーヘッド、局所変数参照のための所要時間の平均である。なお、他のイベントでも実験したが、これらのオーバーヘッドにはイベントの種類による大きな差はなかった。

表 2 から、1 回のスレッドの中断によるオーバーヘッドは約 0.23 msec、局所変数参照のための時間は 1 回あたり約 0.39 msec であることが分かる。局所変数にアクセスする度にこれらの和だけの時間のオーバーヘッドが存在していることになる。挙動がある程度予測される膨大な回数の繰返しの中では、中断・参照を抑制することも考えなければならない程度の数値と思われる。

8　おわりに

Java プログラムのための動的バースマークへの応用を目的として種々の実行時情報を効果的に取得する手法をいくつか検討し実現した。ここで得られる実行時情報を動的バースマークに応用する方法については別に検討すること

[*5] 動作しているすべてのスレッドの中断の要求。

表 2　中断・局所変数参照によって生じるオーバーヘッド (msec)

呼出し数 (a)	中断なし (b)	中断あり (c)	中断・参照 (d)	中断増加	参照増加
10,000	1,825	4,350	8,416	0.253	0.407
100,000	14,999	37,593	77,150	0.226	0.396
1,000,000	147,390	376,750	762,774	0.229	0.386

(Windows XP, Pentium 4 2.4 GHz, 1.0 GB メモリ)

になるが、耐タンパー性の検証などの他の応用も検討する価値はあろう。

得られる実行時情報は一般に膨大であり、時間的・空間的負担は大きい。動的バースマークなどへの実用的応用を考えた場合、予め要求される情報の種類を与えることで選択的に出力させられることが望ましい。これについても今後の課題として挙げられる。

参考文献

[1] Ginger Myles and Christian Collberg：“Detecting Software Theft via Whole Program Path Birthmarks”, Information Security Conference 2004 (ISC2004) (Sep.2004). also in LNCS 3225 pp.404-415 (2004).

[2] 岡本圭司、玉田春昭、中村匡秀、門田暁人、松本健一：“ソフトウェア実行時の API 呼び出し履歴に基づく動的バースマークの実験的評価”、第 46 回プログラミング・シンポジウム報告集、pp.41-50 (2005).

[3] 古田壮宏、真野芳久：“実行系列の抽象表現を利用した動的バースマーク”、電子情報通信学会論文誌 D-I、Vol.J88-D-I No.10, pp.1595-1598 (2005.10).

[4] 林晃一郎、楓基靖、真野芳久：“特徴抽出と抽象化による動的バースマークの構成とその検証”、情報処理学会研究報告、2005-CSEC-31, pp.31-36 (2005.12).

[5] 松本勉, 赤井健一郎、中村豪一、大内功、竹脇和也、村瀬一郎：“Java 対応ランタイムデータ捕捉ソフトウェア”、情報処理学会論文誌、Vol.44 No.8, pp.1947-1954 (2003.8).

[6] 玉田春昭, 門田暁人, 中村匡秀, 松本健一：“Java プログラムの動的解析のためのトレーサ埋め込みツール”, 第 46 回プログラミング・シンポジウム報告集、pp.51-62, (2005.1).

[7] Sun Microsystems, Inc.：“Java™ Platform Debugger Architecture”、http://java.sun.com/j2se/1.3/ja/docs/ja/guide/jpda/.

[8] Apache Software Foundation：“The Apache Jakarta Project, BCEL”、http://jakarta.jp/bcel/.

[9] Tim Lindholm, Frank Yellin (村上雅章訳)：Java™ 仮想マシン仕様 第 2 版、ピアソン・エデュケーション (2001).