Java プログラムの実行時情報を得る方法の検討と実現
— 動的バースマークへ利用できる情報の検討 —

2002MT086 樋田 洋明　　2002MT098 吉村 陵二
指導教員　真野 芳久

1　はじめに

近年、ソフトウェアの盗用が問題となっており盗用の疑いのあるプログラムの発見、立証を目的とした技術として動的バースマークが提案されている。動的バースマークとは、プログラムの実行時動作から抽出されるプログラムの特徴である。

動的バースマークについてはその取得に着目した研究はあるが、そのための実行時情報取得に関する研究は少ない。動的バースマーク取得の際には動的な情報を必要とするためその取得方法の開発は重要であると考えられる。そこで、本研究では種々のバースマークの取得を支援するためにプログラムの実行時情報を得るための手法の検討と試作を行なう。

対象とする言語については盗用の危険性が他の言語と比べて高く、Java プログラムを対象としたバースマーク取得を目的とした実行時情報取得ツールの必要性が高いと考えられるため Java とする。また、本研究では、デバッガの操作・記述を容易にするためのパッケージである Java Debug Interface (JDI)[5] を有効に活用することにより実行時情報取得の方法を検討する。

2　JDI

JDI を使用することにより、 VM (仮想マシン) の状態、クラス、配列などへの内部的なアクセスや VM の実行を明示的に制御することができる [5]。しかし、JDI を利用して実行時情報の取得を行なう際にはクラスファイルにデバッグ情報を付加する必要がある。JDI には応用サンプルプログラムである Trace が用意されている。vTrace の利用により、API、ユーザ定義のメソッド名及びメソッドを定義しているクラス名、フィールドへのデータの受け渡しを抽出できる。Trace は対象となるプログラムの VM のミラーとイベントのやり取りを行なう。ここでのイベントとは、Java プログラム内部で発生するもので処理を通知するために使われるものである。

本研究ではこの Trace を活用し研究を進める。以下、プログラム Trace を TRACE、実行時情報を取得することをトレース、トレース対象のプログラムを対象プログラムと呼ぶ。

3　関連研究

玉田ら [3] は Java を対象としたトレーサ埋め込みツール AddTracer を開発した。実行中の変数の値やメソッド呼び出し関係を出力するトレーサを直接クラスファイルに埋め込むため、ソースコードがないプログラムにも適用でき、プログラム中の任意の箇所にトレーサを埋め込める。また、松本ら [4] は Java の VM 上で動くプログラムの実行過程で現れる実行時情報を捕捉するソフトウェア DataExtractor を開発した。これは、プログラムの計算途中の中間値も捕捉可能であることが特徴である。しかしこれらの研究では、限られた情報の取得しかできなかったり、不必要な情報を大量に取得してしまう可能性があるためバースマーク取得を目的とする場合は十分ではないと考えられる。

4　実行時情報の抽出方法の提案

本研究では Java プログラムの実行時情報の抽出・取得をするための検討を行ない、その中でもバースマークとして利用可能と考えられる情報を得ることのできる方法の実現を目的とする。これを実現するために前述の JDI と TRACE を活用し以下の 2 つを実現する。

1. TRACE をそのまま利用して得られる実行時情報に対して、Perl スクリプトを用いてバースマークとして利用できる実行時情報を抽出する。
2. JDI 機能を最大限に活用することで、バースマークとして有用となる実行時情報を取得する。

1. の方法は、TRACE を利用し実行時情報を取得しこの中から必要な情報のみ抽出する。2. の方法は、JDI に用意されている機能を利用し新たな実行時情報を取得する。TRACE を利用して得られる情報に加えて、ローカル変数や配列などの新たな情報を取得することができる。問題点として、取得する情報によっては実行時間が増大してしまう場合がある。

5　TRACE を利用した実行時情報の抽出

5.1　情報の形式

5.1.1　TRACE によって得られる情報

TRACE によって得られる情報の例を**図 1** に示す。また、以下では TRACE によって得られる情報を TRACE 情報と呼ぶ。

```
1 : ====== thread_name ======
2 : methodA  --  classA
3 : | methodB  --  classB
4 : | | methodC  --  classC
5 : | | | methodD  --  classD
6 : methodE  --  classE
7 : ====== thread_name end ======
```

図 1 TRACE によって得られる情報の例

TRACE 情報には、動作したメソッドの名前・そのメソッドを含むクラスパッケージの名前・動作したス

レッドの名前が含まれ、それぞれ methodA、classA、thread_name などと表わされ、**図 1** のような形式で出力される。また、各メソッド間の呼び出し関係を表わす情報も含まれる。この呼び出し関係は、**図 1** の 2〜6 行目のように「|」を用いて表される。以下、このメソッド間の呼び出し関係を階層関係と呼ぶ。「|」がない階層を第 0 階層とし、それ以降は「|」の数により第 1 階層、第 2 階層・・・とする。

5.1.2 抽出される情報

本方法では必要な実行時情報として API 情報に着目する。API 情報とは API のクラスパッケージ名、そのクラスパッケージに属するメソッド名のことを指す。この情報に着目する理由は、ユーザ定義メソッドやユーザ定義クラスは名前の変更が容易であるのに対し、API 情報の改変はライブラリの解析や改変が必要になり非常に困難である [2] ためである。

5.2 実現方法の検討

TRACE から取得される実行時情報は膨大であり、バースマーク取得のためにこの実行時情報を効率的に利用できる方法が必要である。提案する方法の全体像を**図 2** に示す。この図の要求指定ファイルとは、抽出したい API 情報、抽出したくない API 情報をあらかじめ要求として与えておくために用意されるファイルである。実装には、容易な実現によって様々な処理を実現することが可能であるため Perl を利用する。

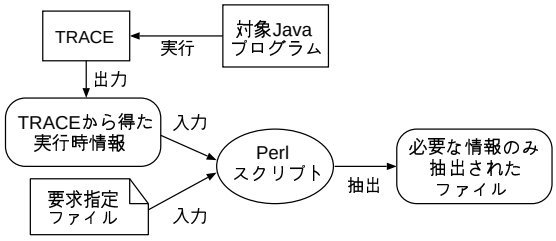

図 2 実行時情報抽出の全体像

5.3 システムの機能

実現されたシステムは入力ファイルの指定に基づき以下の方法で実行時情報の抽出を行なう。

■階層関係による抽出　前述の階層構造の中からユーザが要求した階層の情報のみまたはユーザが要求した範囲の階層の情報を全て抽出する。

■2 つの実行時情報中の一致した部分の抽出　2 つの異なる実行時情報を比較し一致する部分を全て抽出する。これを利用し実行時情報を数種比較することにより、対象プログラム特有の実行時情報が抽出可能となると考えられる。

■必要とする API 呼び出し情報の抽出　バースマーク取得に役立つ情報であると考えられるユーザが要求した API メソッドを抽出する。ユーザが要求した API メソッドとは第 0 階層で呼び出された API メソッドまたはユーザ定義メソッドが呼び出した API メソッドを指す。

6 JDI を利用した実行時情報の抽出

本節では JDI を利用した実行時情報取得を目的とし、その実現過程を以下の 2 つに大きく分ける。

- 実行時情報制御機能の利用
- 実行時情報取得方法及び取得済み実行時情報の検討

また以下では説明のため、TRACE を拡張したものを拡張 TRACE と呼ぶ。

6.1 実行時情報取得のための実行制御機能

6.1.1 ブレークポイント

ブレークポイントとはソースコード上の任意の行に付加されるマーカーである。トレース時にプログラムの各命令を順番に実行していく際、ブレークポイントが付加された行の命令を実行する直前でプログラムの実行を一時停止する。またユーザはこの時、その周辺のコンテキスト情報を見ることによりプログラムの挙動を調べることができる。そのためプログラムを一時停止することによって、ブレークポイント未設定時と比較してより詳細な実行時情報を得ることができる。その代表例にローカル変数値が挙げられる。ローカル変数値はユーザが特定の箇所においてスタックフレーム内のローカル変数値を確認する場合、プログラムを一時停止しユーザに制御権が与えられなければ監視できない。このためユーザがより多くの情報を確認したい箇所にはブレークポイントを設定し一時停止させることが必要である。JDI においてブレークポイントを利用する場合、ブレークする位置を指定する必要がある。この設定位置を任意で変えることにより、取得情報の系列を変化させることが可能である。更に JDI ではブレークポイントはそれが有効であるとき、プログラム一時停止以外の処理を付随させることができる。

6.1.2 ステップ実行

ステップ実行とは 1 個または 1 行の命令を実行した後、対象プログラムの実行を停止し、デバッガに制御を移すといった実行制御機能である。ステップ実行には停止中の行の命令にメソッド呼び出しが含まれている場合、そのメソッドを実装したコードの最初の行で再停止するステップイン、メソッド呼び出し側の次の 1 行に進むステップオーバーなどがある。JDI ではこのステップイン、ステップオーバーの機能を備えた StepEvent というイベントが提供されており、このイベントを利用することによりステップ実行を実現することができる。

6.2 実行時情報取得方法とその実行時情報

6.2.1 ローカル変数

ローカル変数はプログラムの動的特徴をあらわす情報として重要である。本節ではその取得方法及び実現結果について述べる。

VM スタックのスタックフレーム内にはローカル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 thread のオペランドスタックにアクセスするインタフェースは提供しないため JDI ではオペラ

ンドスタックへのアクセスは不可能である。しかしローカル変数領域は前述のブレークポイント及び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。ローカル変数取得の概念図を**図 3** に示す。

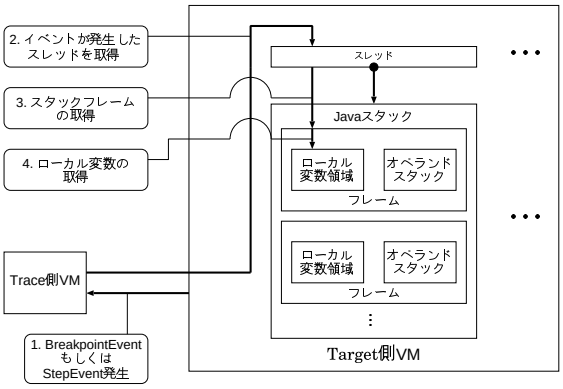

図 3 ローカル変数領域アクセスへの概念図

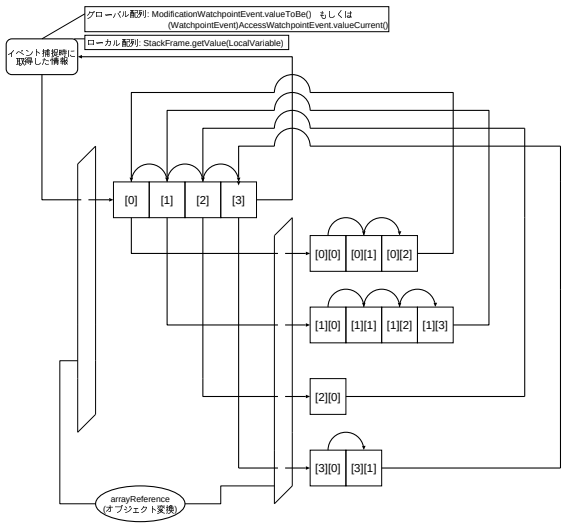

図 4 配列の各要素値取得の概念図

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. フレーム内のローカル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は拡張 TRACE によって、ターゲット VM 内における対象プログラム内でクラスの準備のイベントを捕捉する。その際に捕捉クラス内のメソッド全行に対してブレークポイントを設定することによってローカル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あらかじめ拡張 TRACE 内によって、ターゲット VM 内における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。

6.2.2 配列

TRACE では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得られるが、その配列の各要素の値まではアクセスできない。そのため本節では配列各要素値の取得及び拡張 TRACE への実装方法について述べる。

以下、取得方法別にトレース対象プログラム内のクラスにおけるすべてのメソッドの外側で生成される配列 (以下グローバル配列) 及びクラスにおける各メソッド内で生成される配列 (以下ローカル配列) の 2 つに分け、その取得方法及び実験結果について述べていく。配列の各要素値取得の概念図を**図 4** に示す。グローバル配列の場合、まず ModificationWatchpointEvent もしくは AccessWatchpointEvent の捕捉を契機にフィールドの現在値を取得する。**図 4** 中における ModificationWatchpointEvent.valueTobe() は変更フィールド値、(WatchpointEvent)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どうか検査し、配列オブジェクトでなければ変数値取得等その他の処理を行なう。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インタフェースである arrayReference を用いてオブジェクト変換を行ない、リストとして取得する。**図 4** では arrayReference を用いてフィールドを検査した後、配列情報をリストオブジェクトとして配列全要素を取得している。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、この検査を一回のみ行なうだけで配列の各要素値は取得できる。しかし配列が 2 次元以上である場合はこの arrayReference インタフェースを次元数分適用しなければ配列の各要素値は不明のままとなる。**図 4** ではまず 1 次元目の先頭要素に対してオブジェクト変換を行ない、その要素が持つ 2 次元目の配列リストが取得できる。取得要素がこれ以上配列リストを保持していなければ値の取得を行ない次の要素へ移る。リスト内要素をすべて取得した後、1 次元前のリストへ戻り、そのリストの次の要素に移り同様にオブジェクト変換を行ない要素値の取得を行なっていく。これらの処理を繰り返すことよって、取得配列の次元数及び要素数が不明である場合も最終的に各要素値まで辿り着くことができる。ローカルな配列の場合は**図 4** の ModificationWatchpointEvent.valueTobe() 、(WatchpointEvent)AccessWatchpointEvent.valueCurrent() 部分を前節のローカル変数値取得におけるスタックフレームの取得 StackFrame.getValue(LocalVariable) から arrayReference インタフェースを適用することにより同様に取得できる。

表 1 BreakpointEvent 捕捉にかかるスレッド中断指定別の実行時間

イベント捕捉回数	スレッド中断指定		
	SUSPEND_ALL	SUSPEND_EVENT_THREAD	SUSPEND_NONE
1000	2.24	2.25	1.77
10000	13.55	13.45	4.75
100000	126.34	123.87	28.73
1000000	1275.12	1219.58	274.08

実行時間の単位は sec

表 2 StepEvent 捕捉にかかるスレッド中断指定別の実行時間
ステップのサイズ: STEP_LINE(行単位)、ステップの深さ: STEP_INTO(ステップイン)

イベント捕捉回数	スレッド中断指定		
	SUSPEND_ALL	SUSPEND_EVENT_THREAD	SUSPEND_NONE
1000	2.56	2.50	2.03
10000	16.20	15.81	5.92
100000	149.28	146.61	34.99
1000000	1461.75	1456.50	338.58

実行時間の単位は sec

またオブジェクトの配列に関してはこの配列オブジェクト変換処理にクラスオブジェクト変換処理を追加することによってオブジェクトの型、すなわちオブジェクトのクラスの情報が取得できる。そのため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 実験結果とその評価

本節では 前章において利用した BreakpointEvent 及び StepEvent を捕捉した場合のトレース実行時間について考察を行なった。各イベントの捕捉回数とトレース実行時間の関係をそれぞれ**表 1**、**表 2** に示す。

また BreakpointEvent、StepEvent をそれぞれ生成する際にスレッド中断の方法には 3 つの場合があり、それぞれすべてのスレッドを中断する場合 (SUSPEND_ALL) とイベントを生成したスレッドのみを中断する場合 (SUSPEND_EVENT_THREAD)、どのスレッドも中断しない場合 (SUSPEND_NONE) となる。

尚、実行環境は OS:WindowsXP Home Edition、CPU:Intel Pentium4 3.00GHz、メモリ:1024MB RAM である。

両イベントとも SUSPEND_NONE 指定時にはスレッドの中断は行なわないが、イベント捕捉回数が増大するにつれてトレース実行時間は明らかに増大する。そのためイベント未発生時と比較すると、イベントの捕捉はトレース実行のオーバーヘッド要因となると考えられる。また SUSPEND_ALL 及び SUSPEND_EVENT_THREAD 指定時は SUSPEND_NONE 指定時と比べると実行時間は更に大きくなっている。これは SUSPEND_NONE 指定時ではオーバーヘッドがイベント捕捉時間のみであったのに対し、SUSPEND_ALL 及び SUSPEND_EVENT_THREAD 指定時にはイベント捕捉時間に加え、スレッド中断時間も加算されていることが要因となっている。また SUSPEND_ALL 及び SUSPEND_EVENT_THREAD 指定の際は、イベント捕捉回数が 10000 以降はイベント捕捉回数に比例してトレース実行時間は増大する。SUSPEND_NONE 指定時に関しても 100000 以降は同様に比例関係となる。この結果から、トレース実行時間はイベント捕捉回数及びスレッド中断の増減によって大きく影響を受けると考えられる。また両イベント間でもトレース実行時間差が生じるため、ユーザは取得目的に応じて両イベントを使い分ける必要がある。

8 おわりに

本研究では動的バースマークの取得の支援を目的として様々な実行時情報の抽出および取得を 4 章で述べた 2 つの手法に分けてその提案と実現を行なった。また 7 章では作成したシステムについての時間的性能について評価したが、イベント生成及びスレッド中断によるオーバーヘッドはイベント未生成時と比較し、かなり大きくなってしまう。そのため現実的有効性を高めるためにはシステム各処理部の役割を見直し、その機能を最大限に生かす方法を考えていく必要がある。また、更に詳細な実行時情報を取得するためには、JDI 自体の拡張を行ない、インタフェースをより充実させる必要があるとも考えられる。

参考文献

[1] 古田, 真野：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).

[2] 林, 楓, 真野：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).

[3] 玉田, 門田, 中村, 松本：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).

[4] 松本, 赤井, 中村, 大内, 竹脇, 村瀬：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.1947–1953 (2003).

[5] Sun Microsystems,Inc:Java TM Debug Interface, http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/.