2005年度　卒業論文

Java仮想マシンとオブジェクトキャッシュの構造と動作に関する理解

指導教員　　清水尚彦　教授

東海大学電子情報学部コミュニケーション工学科

1ADT2413　　横山　圭

目 次

1　はじめに

Java 仮想マシンの構造と動作についての理解と、オブジェクトキャッシュの動作について理解することを目的とする。

2　Java 仮想マシンの構造

Java 仮想マシンは、Sun Microsystems 社のプログラミング言語 Java で作ったソフトを動かすための土台となるプログラムである。ここでは、Java 仮想マシンの構造について説明する。

2.1　Java 仮想マシンとは

Java 仮想マシンは、名前から想像される通り「Java を実行するための仮想的な計算機」のことである。Java 仮想マシンは Java 言語をサポートするように設計されており、その命令セットであるバイトコードやクラスファイルフォーマットは Java 言語を前提に設計されている。

Java 仮想マシンとは、Java バイトコードをそのプラットフォームのネイティブコードに変換して実行するソフトウェアである。Java 言語で開発されたソフトウェアは、配布時にはプラットフォームから独立した独自の形式（Java バイトコード）になっており、そのままでは実行することができない。このため、そのプラットフォーム固有の形式（ネイティブコード）に変換するソフトウェアを用意して、変換しながら実行する。この変換と実行を行なうのが Java 仮想マシンである。実行前にまとめて変換することで実行時のオーバーヘッドをなくし、実行速度を向上させたものを JIT コンパイラという。また、Java 仮想マシンはスタックマシンであり、スタックを中心に処理を進めている。通常、Java 仮想マシンには複数のスレッドがあり、各スレッドにはプログラムカウンタとスタックが一つずつある。各スタックは複数のフレームよりできており、各フレームにはオペランドスタックとローカル変数があります。フレームはメソッドが呼び出されるごとに新たに生成され、メソッドが終了するとともに破棄される。メソッドからは最後に生成されたフレーム（カレントフレーム）のみにアクセス可能である。なお、各オペランドスタックとローカル変数の最大値は、各メソッドごとに決まっており，コンパイル時に静的に決定されクラスファイルに記録される。

2.1.1　Java バイトコード

Java のコンパイラ（人間がプログラミング言語で記述したソフトウェアの設計図（ソースコード）を、コンピュータが実行できる形式（ネイティブコード）に変換するソフトウェア) が生成する実行用コードのこと。中間コードの一種で、特定の環境に依存しないという特徴を持つ。Java バイトコードは Java 仮想マシン内のインタプリタ（記述したソースコードを、ネイティブコードに変換しながら、そのプログラムを実行するソフトウェア）によってネイティブコードに変換されてから実行される。

2.1.2　プラットフォーム

アプリケーションソフト（文書の作成、数値計算など、ある特定の目的のために設計されたソフトウェア）を動作させる際の基盤となる OS（多くのアプリケーションソフトから共通して利用される基本的な機能を提供し、コンピュータシステム全体を管理するソフトウェア）の種類や環境、設定などのこと。

Windows や UNIX、Mac OS は、それぞれ異なるプラットフォームである。

アプリケーションソフトにせよ OS にせよ、対応しているプラットフォームはあらかじめ決まっており、それ以外のプラットフォームでは動作しない。

2.1.3　ネイティブコード

コンピュータに理解できる言語 (マシン語) で記述されたプログラム。オブジェクトコードとも呼ばれる。数値の羅列として表現されるため、そのままの形で人間が理解するのは困難。通常は、人間がプログラミング言語を使って作成したソースコードを、コンパイラなどの変換ソフトウェアを使ってネイティブコードに変換する。

2.1.4 JIT コンパイラ

Java プログラムを実行する際に、プラットフォームから独立した形式のプログラム (Java バイトコード) を、実行前にまとめて一気にそのプラットフォームで実行可能なプログラム (ネイティブコード) に変換し、実行する機構のこと。少しずつ変換しながら実行する従来のインタプリタ型の方式より実行速度は速いが、変換に時間を要するので実行を始めるまでにかかる時間は従来より長くなる。

2.1.5 スタックマシン（スタック）

最後に入力したデータが先に出力されるという特徴をもつ、データ構造のマシン。

本を机の上に積み上げるような構造になっており、データを入れるときは新しいデータが一番上に追加され、データを出すときは一番上にある新しいデータが優先して出てくる。

このように、「最後に入った物が最初に出てくる」というデータの入出力方式は「Last In, First Out」あるいは「First In, Last Out」、略して「LIFO」、「FILO」と呼ばれる。アセンブリ言語のプログラミングではもっとも頻繁に利用されるデータ構造の一つで、多くの CPU はスタックにデータを出し入れするための専用の命令を用意しており、簡単に利用することができる。サブルーチン（プログラム中に何度も同じ処理が出て来る場合、その同じ部分を別プログラム（サブルーチン）として用意し元のプログラムからはこの部分を何度も利用する（呼び出す）形にするとプログラムの記述量が減る。さらに、何度も同じものを書かないのでプログラムミスの危険性も減少する）や関数を呼び出す際に、処理中のデータや戻りアドレスなどを一時的に退避する場合に使うことが多い。

なお、スタックとは逆に、「最初に入った物が最初に出てくる」というデータの入出力方式は「First In First Out」を略して「FIFO」と呼ばれる。先に入力したデータが先に出力されるデータ構造は、キュー (待ち行列) と呼ばれている。

2.1.6 スレッド

マルチスレッド（1 つのアプリケーションソフトがスレッドと呼ばれる処理単位を複数生成し、並行して複数の処理を行なうこと。CPU の処理時間を非常に短い単位に分割し、複数のスレッドに順番に割り当てることによって、複数の処理を同時に行っているようにみせている）に対応した OS 上での、ソフトウェアの実行単位。1 つのプログラムは最低 1 つのスレッドを持つ。

同じプログラムに属するスレッドはメモリなどのリソースを共有する。複数のスレッドは CPU を交互に占有することによって見かけ上同時実行が可能。

高度な処理を行なうアプリケーションソフトなどでは、スレッドを複数走らせることにより、同時に複数の処理を実行できる。時間のかかる演算処理中にユーザからの入力を受け付けるといった工夫が可能になる。

2.1.7 プログラムカウンタ

命令をメモリから次々と読みだして演算装置に送るためには次に読むメモリ番地を覚えておくことが必要である。このために命令を読み出す度に次の命令の番地にカウントアップされるカウンタが演算装置に組み込まれている。これをプログラムカウンタという。

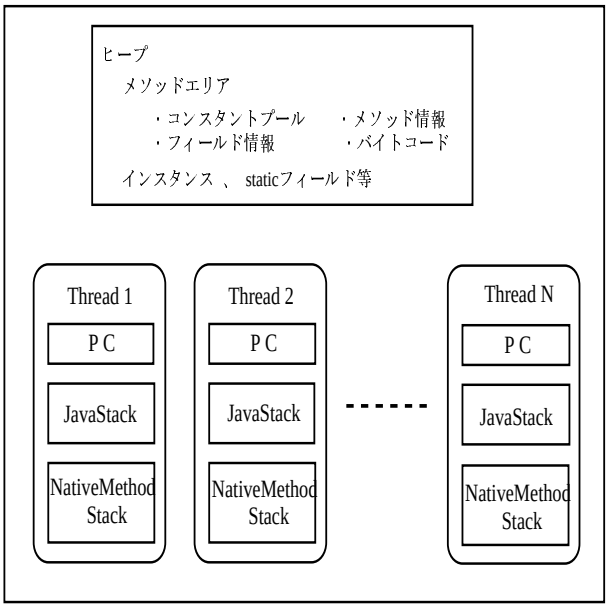
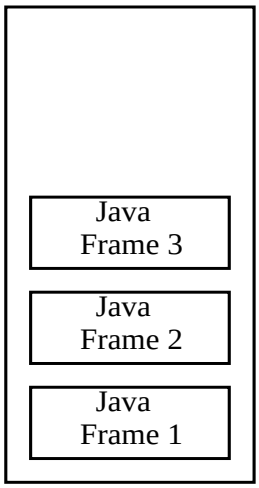
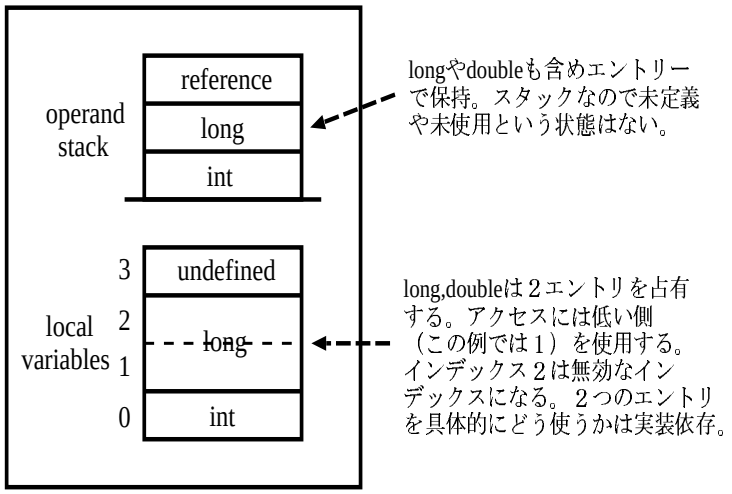


図 1: Java 仮想マシン全体



JavaStackは複数のフレームで構成される。トップのフレームがカレントフレームで、通常はカレントフレームにのみアクセス可能。

図 2: Java スタック



オペランドスタックとローカル変数以外については実装依存。実際には呼び出し元に関する情報を保持することが多い。

図 3: フレーム

上記の図 1、図 2、図 3 は Java 仮想マシンの構造を図示したものである。

2.2 メモリの構造

Java 仮想マシンのメモリの構造は、Java スタック、フレーム、メソッドエリア、ヒープ、ローカル変数、オペランドスタックなどから成り立っている。

2.2.1 Java スタック

Java 仮想マシンの各スレッドは、スレッドと同時に生成される個別の Java スタックを保持している。Java スタックとは、Java 仮想マシンのフレームをストアするもので、C のような従来からある言語のスタックに相当する。すなわち、ローカル変数と結果の一部を保持し、メソッドの呼び出しやリターンを行う役割を担っている。スタックは、フレームのプッシュとポップを除けば決して直接操作されることがないため、実際にはヒープとして実装され、Java フレームがそのヒープ領域に割り当てられる。Java スタック用のメモリは連続している必要は無い。

2.2.2 フレーム

Java 仮想マシンのフレームとは、ダイナミック・リンクを行ったり、メソッドの値を返したり、例外の処理を行ったり、データと結果の一部をストア（オペランドスタックの先頭にある値をポップし、ローカル変数にプッシュする命令）するために使用するものである。

Java のメソッド呼出しごとに、新たなフレームが生成される。正常終了しても、異常終了しても、フレームはメソッドの終了時に破棄される。フレームは、それを生成するスレッドの Java スタックから割り当てられる。各フレームは、ローカル変数とオペランドスタックを個別に保持している。ローカル変数エリアとオペランドスタックのサイズはコンパイル時に明らかであり、フレームのデータ構造体が Java 仮想マシンの実装にのみ依存しているため、こういった構造体に対して同時にメモリスペースを割り当てることができる。

2.2.3 メソッドエリア

Java 仮想マシンは、すべてのスレッド間で共有されるメソッドエリアを保持している。メソッドエリアとは、従来の言語におけるコンパイル済のコードの格納

領域に似たものである。メソッドエリアはクラス構造体ごとに、例えば、コンスタントプール、フィールドとメッソドのエリア、メソッドと構築子のコード、そしてクラス型やインスタンスの初期化やインターフェース型の初期化で使用する特別なメソッドを格納している。

2.2.4 ヒープ

Java 仮想マシンは、すべてのスレッド間で共有されるヒープを保持している。ヒープとは、すべてのクラスインスタンスおよび配列の割り当てを行う実行時のデータエリアのことである。

Java のヒープは、仮想マシンの開始時に生成される。オブジェクトのヒープストレージは、自動の記憶域管理システム（典型的にはガーベッジコレクタ）によって再利用される。すなわち、オブジェクトは明示的に再割り当てされるわけではない。Java 仮想マシンは特定の形式の自動記憶域管理システムを想定しているわけではないため、実装者はそのシステム要求に沿った記憶域管理技法を選択することができる。Java のヒープは、固定サイズとすることもできるし、また、処理で必要になった際に拡張したり、大きなヒープが不必要になった際に縮小したりすることもできる。Java のヒープ用メモリは、連続している必要はない。

2.2.5 メソッド・エリアとヒープの関係

Java 仮想マシンはクラスファイルを読み込むと、クラスをメモリに展開する。このメモリ中のクラスの置き場所がメソッド・エリアである。そのクラスのインスタンスを生成すると、そのインスタンスはヒープに置かれる。ただし、実装によっては、ヒープの中にメソッド・エリアを割り当てる場合もある。

2.2.6 Java スタックとフレームの関係

メソッド・エリアとヒープの他に、Java スタックというメモリ領域がスレッド毎に割り当てられている。Java スタックはコールスタックと呼ばれることもある。それは、メソッドがコールされるたびにフレームが積まれるスタックだからである。

スレッドと Java スタックの関係は常に 1 対 1 なので、スレッドが 1 つ生成されると、そのスレッドの Java スタックが 1 つ生成される。また、Java スタックの一番上に積まれたフレームのメソッドのことを、カレント・メソッドといい、カレント・メソッドの所属するクラスのことを、カレント・クラスという。スレッドが実際に実行しているのは、常にカレント・メソッドだけである。

2.2.7 ローカル変数

Java 仮想マシンのローカル変数は、Java 仮想マシンメソッドの実行中にだけ存在する値の格納場所である。Java 言語のコンパイラが Java 言語のメソッドを変換して Java 仮想マシンのメソッドを生成するとき、Java 言語のローカル変数とメソッド引数を Java 仮想マシンのローカル変数に割り当てる。このとき、Java 言語のローカル変数は、自由な名前が付けられるが、Java 仮想マシンではローカル変数は名前ではなく、番号 (0 から始まる整数) で指定される。このローカル変数の配列の大きさは、メソッド毎に異なり、メソッドのコンパイル時にコンパイラによってそのサイズが計算される。

2.2.8 オペランド・スタック

Java スタックがスタックであるように、オペランド・スタックもスタックなので、何かを積み上げておく場所である。この場合、オペランドを積み上げておく場所になる。Java スタックとオペランド・スタックはまったく別のものなので、混同しないようにしなければならない。Java スタックはフレームを積み上げていく場所であり、その各フレームの中にローカル変数配列とオペランド・スタックが 1 つずつ入っている。

2.3 クラスファイルの構造

メモリの構造と密接に関係しているのが、クラスファイルの構造である。通常、クラスファイルは Java 言語で書かれたソースをコンパイルすることで生成されるが、他の言語からコンパイルすることもできたり、バイナリエディタで直接書くことも可能である。JavaClass のようなツールで変更を加えても構わない。また、必ずしもファイルである必要もない。Java 仮想マシンはこのクラスファイルを読み込んで実行する。クラス

ファイルの詳しい仕組みは「3. クラスファイル」で説明する。

2.3.1 クラスをロードする方法

Java 仮想マシンにクラスファイルをロードする (読み込む) 方法は大きく分けて 2 つあり、1 つは、仮想マシンに元から組み込まれているクラスのローディング機構を使ってクラスをロードする方法である。これをシステムクラスローダと呼ぶことにする。もう 1 つは、ユーザ定義のクラスローダを使う方法である。ユーザ定義のクラスローダは、java.lang.ClassLoader というクラスのサブクラスのインスタンスとして実装する。これをユーザクラスローダと呼ぶことにする。

システムクラスローダは 1 つの Java 仮想マシン中に 1 つしか存在しないが、ユーザクラスローダは 1 つの Java 仮想マシンに複数存在することができる。システムクラスローダは、java.lang.ClassLoader オブジェクトを使わずにクラスをロードする。

UNIX や Windows の JDK の Java 仮想マシン実装では、システムクラスローダは CLASSPATH で指定された場所からクラスを読み込む。CLASSPATH 以外の場所からクラスを読み込むには、ユーザクラスローダを使わなければならない。

Web ブラウザの場合はネットワークからクラス (アプレット) をロードするので、Web ブラウザを実装するにはユーザクラスローダを使わなければならない。普通、ユーザクラスローダはネットワークやローカルディスクからクラスファイルを取ってきて読み込むが、別に外部ファイルやデータベースからクラスファイルを読み込まなくてもかまわない。要はクラスファイルを表すバイト列さえあればクラスが作れるので、実行時にクラスファイルを表すバイト列を配列として作成し、そのバイト配列をクラスファイルとしてメモリにロードすることもできる。

2.4 データ型

Java 仮想マシンは、Java 言語と同様、プリミティブ型と参照型という 2 種類の型の操作を行う。これら 2 つの型に対応する値としてプリミティブ値と参照値が存在しており、これら 2 つの値は変数にストアしたり、引数として渡したり、メソッドからリターンしたり、操作したりすることができる。

Java 仮想マシンは、大半の型検査が Java 仮想マシン自身によってではなくコンパイル時に行われるものと期待している。特に、データのタグ付けを行う必要はないが、タグ付けを行わない場合には型を決定するための検査を可能にしておく必要がある。これを代替するものとして、Java 仮想マシンの命令セットは特定の型の値を操作する命令によってそのオペランド型を識別する。例えば、iadd,ladd,fadd,dadd はみな 2 つの数値を加算する Java 仮想マシンの命令であるが、そのオペランドの型はそれぞれ int,long,float,double でなければならない。

また、Java 仮想マシンはオブジェクトを明示的にサポートしている。オブジェクトへの参照は、Java 仮想マシンの参照型を保持しているとみなされ、参照型の値はオブジェクトへのポインタとして考えることができる。また、1 つのオブジェクトに対して、2 つ以上の参照が存在する場合もある。Java 仮想マシンは、オブジェクトへ働きかける場合でも決して直接にはアドレッシングを行わず、必ず参照型の値を通して操作、引き渡し、判定を行う。

2.5 プリミティブ型とその値

Java 仮想マシンがサポートしているプリミティブデータ型は数値型と returnAddress 型であり、数値型は整数型および浮動小数点型で構成されている。整数型は、

- byte
  値は 8 ビットの符号付き 2 の補数整数。
- short
  値は 16 ビットの符号付き 2 の補数整数。
- int
  値は 32 ビットの符号付き 2 の補数整数。
- long
  値は 64 ビットの符号付き 2 の補数整数。
- char
  値は Unicode バージョン 1.1.5 の文字を
  表現する 16 ビットの符号なし整数。

そして浮動小数点数型は、

- float
  値は 32 ビットの IEEE 754 浮動小数点数。

- double
  値は 64 ビットの IEEE 754 浮動小数点数 。

returnAddress 型の値は Java 仮想マシンの命令のオペコードへのポインタである。また、returnAddress 型だけは Java 言語の型として存在していない。

2.5.1 整数型とその値

Java 仮想マシンの整数型の値は、Java 言語の整数型の値と同じである。

- byte
  -128 以上 127 以下

- short
  -32768 以上 32767 以下

- int
  -2147483648 以上 2147483647 以下

- long
  -9223372036854775808 以上
  9223372036854775807 以下

- char
  ’＼u0000’ から’＼uffff’ まで。char は符号なしであるため、’＼uffff’ は式中で使用する際には-1 ではなく 65535 となる。

2.5.2 浮動小数点型とその値

Java 仮想マシンの浮動小数点数型の値は Java 言語の浮動小数点数型の値と同じである。浮動小数点数型である float と double は、IEEE Standard for Binary Floating-Point Arithmetic,ANSI/IEEE Std.754-1985(IEEE New York) が規定する、単精度 32 ビットと倍精度 64 ビットのフォーマットによる IEEE 754 の値を表現する。

- float 型で正の有限の最大浮動小数点数リテラルは 3.40282347e+38F である。また、正のゼロではない最小浮動小数点数リテラルは 1.40239846e-45F である。

- double 型で正の有限の最大浮動小数点数リテラルは 1.79769313486231570e+308 である。また、正のゼロではない最小浮動小数点数リテラルは 4.94065645841246544e-324 である。

2.5.3 returnAddress 型とその値

returnAddress 型を使用する Java 仮想マシンの命令は jsr、ret、jsr-w という命令である。そして returnAddress 型の値は Java 仮想マシンの命令のオペコードへのポインタとなっている。数値のプリミティブ型とは異なり、returnAddress 型は Java のいかなるデータ型にも対応していない。

2.5.4 boolean 型は存在しない

Java は boolean 型を定義しているが、Java 仮想マシンには boolean 値の演算専用の命令は存在していない。その代わりに Java 仮想マシンは、boolean 値の演算を行う Java の式を、int のデータ型を使用した boolean 変数を表すものとしてコンパイルする。

Java 仮想マシンは boolean 型の配列生成をサポートしているが、boolean 配列の原始要素に対するアクセスや更新はサポートしていない。しかし、byte の配列命令を使用することにより、任意の boolean 型に対するアクセスや更新を行うことができる。

2.6 参照型とその値

参照型にはクラス型、インタフェース型、配列型の 3 種類があり、動的に生成したクラス・インスタンス、配列、インタフェースを実装するクラスのインスタンスや配列を値として参照する。参照値は特別な参照である null 参照、すなわちオブジェクトを指さない参照である場合があり、これを null として表現することができる。null 参照は初期状態では実行時の型を持たないが、任意の型にキャストすることができる。

2.7 ワード

Java 仮想マシンはそのデータ型のサイズを規定しない代わりに、プラットフォームの仕様に依存したサイズとなるワードについての抽象的な概念を定義してい

る。1 ワードとは、byte、char、short、int、float、参照型、returnAddress 型の値、もしくはネイティブ・ポインタを保持するために必要な大きさである。そして 2 ワードとは、より大きな型である long と double の値を保持するために必要な大きさである。Java の実行時のデータ・エリアはすべて、これらの抽象的なワードとして定義されている。

2.8 オブジェクトの表現

Java 仮想マシンは、オブジェクトに対して特別な内部構造体を必要としていない。Sun Microsystems 社の現行の Java 仮想マシンの実装では、クラス・インスタンスへの参照は、それ自身が 1 組のポインタを形成しているハンドルへのポインタとなっている。1 つは、オブジェクトのメソッドと、オブジェクトの型を表す Class オブジェクトへのポインタを保持しているテーブルへのポインタであり、もう 1 つは、オブジェクト・データに対して Java のヒープから割り当てられるメモリへのポインタである。

Java 仮想マシンの他の実装では、メソッドテーブルのディスパッチではなくインラインキャッシングのようなテクニックを使用し、ハンドルを使用しないこともある。

2.9 クラスローダ

プログラム実行中に、プログラムモジュールの一部を動的にロードするようなプログラムは柔軟であるといえる。さらに欲をいえば、どのプログラムモジュールをロードするかを実行時に動的に決定できれば、さらに柔軟で拡張性の高いプログラムになる。Java ではこの、ユーザプログラムによるモジュールの動的ローディング機構がとてもよく整備されており、ユーザプログラムによるクラスローダ、つまりユーザクラスローダがこの機能を強力にサポートしている。

Java 仮想マシンを組み込んだ Web ブラウザはこのユーザクラスローダの応用例の 1 つである。アプレットとはプログラムモジュールそのものであり、どのアプレットをロードするかは、ユーザがどの Web ページを開くかという実行時の情報によって決定される。

2.9.1 概要

Java 仮想マシンには 2.3.1 で説明したように、システムクラスローダとユーザクラスローダの 2 種類がある。システムクラスローダは 1 つの Java 仮想マシンプロセスに対して 1 つだが、ユーザクラスローダは 1 つの Java 仮想マシンに対して複数個存在できる。また、ユーザクラスローダはユーザプログラムを定義できる。具体的には、ユーザプログラムで java.lang.ClassLoader クラスのインスタンスを 1 つ生成するたびに、1 つのユーザクラスローダができる。

2.9.2 クラスローダとクラスの名前空間

クラスローダの数だけクラスの名前空間が存在する。つまり、1 つの名前空間には同名のクラスが複数存在することはできないが、名前空間が異なれば同名のクラスが同じ Java 仮想マシンプロセスのメソッドエリアに複数存在するようなことも許される。Java 対応の Web ブラウザはこれを利用して、アプレットクラスを矛盾なくロードしている。いろいろな実装が考えられるが、例えば HTTP サーバ毎にユーザクラスローダを生成することで、サーバ毎に別々の名前空間を用意し、クラス名の衝突を回避できる。

2.9.3 ユーザクラスローダの応用例

ユーザクラスローダを使えば、プログラム自身の一部であるプログラムモジュールをヒープで合成し、合成したものをそのままオンメモリで実行してしまうようなことができる。これは、いろいろな応用が考えられる。例えば、実行時に GUI やコンソールからユーザ入力を読み込んで、それに基づいて実行動作を決定するようなプログラムの場合、プログラム中に一種のインタプリタのようなロジックが現れることがある。そしてそのインタプリタ部分がループの底であるような場合 (ある種の統計処理や科学計算プログラムなど) そのインタプリタ処理部分がボトルネックになってしまうようなことがある。この場合、どの計算方法を行うかいちいち判定しながら実行されることになり、実行効率があまりよくならないので、ユーザ入力を一括してマシン語コードに変換し、その後そのマシン語を実行するような処理が必要になる。これは、Java 仮想マ

シン実装そのものが抱える問題と本質的に同じ問題であり、Java 仮想マシンの場合この問題は JIT という方法で解決することができる。

Java 仮想マシンの JIT は簡単に実装できるようなものではないが、Java のユーザクラスローダをうまく使えば、簡単にユーザ入力をマシン語に変換することができる。その方法は、次のような流れになる。

1. ユーザ入力から Java バイトコードを生成する。
2. 生成した Java バイトコードを Java 仮想マシンで実行する。
3. Java 仮想マシンは JIT を使って Java バイトコードからマシン語を生成する。
4. Java 仮想マシンは生成されたマシン語を実行する。

これらは、いっさいの中間ファイルを使わずにオンメモリで実行できる。ユーザ入力の量にもよるが、それほど複雑なことを行わければ、ユーザ入力データが最終的にマシン語変換されて実行されるまでの時間はほんの一瞬にすることができる。

2.10 ベリファイア

単なるテキストファイルの拡張子を Class に変えたものを Java 仮想マシンで実行しようとしても、実行できない。それは、Java 仮想マシンがそのクラスファイルの中身を調べ、それが Java のクラスファイルでないと判断するためである。このようにクラスファイルの中身が単なるテキストファイルであるというような極端な例でなくても、たった 1 バイトだけクラスファイルのフォーマットに従わないバイトが混じっているだけで、そのクラスは正しい Java のクラスとは見なされない。このクラスファイルのフォーマットの検証を行う処理過程をベリフィケーションといい、この検証処理を行うプログラムやモジュールをベリファイアという。

2.10.1 ベリファイアが行う処理の概要

Java 仮想マシン仕様が定めているクラスファイルの制約事項とベリファイアが行うべき処理を総計すると、かなりの数になるので、ここでは、ベリファイアが行う作業のうち、代表的なものをいくつかピックアップしてみる。

1. フォーマット検査
   フォーマットのベリファイアでは、単純な文法エラーのみを検出する。例えば、次のようなクラスファイルはベリファイアをパスできない。
   - 先頭の 4 バイトが「CA FE BA BE」でない場合。
   - メソッドの要素数が 4 と宣言されているのに、7 つのメソッドがある場合。
   - メソッドの Code 属性の code 配列の長さは 0 であってはいけないのに、0 になっている場合。
   - メソッドディスクリプタが文法エラーを起こしている場合。
2. クラスファイル内の項目間の一貫性
   文法的にはエラーがなくても、クラスファイル内の各項目が矛盾している場合、ベリファイアをパスすることはできない。
   - 例えば、コンスタントプール配列の長さが 37 なのに、45 番目の CONSTANT __ Utf8 エントリを参照する CONSTANT __ Class が存在する場合。
   - goto インストラクションの飛び先のアドレスが code 配列の先頭より前か、最後尾よりも後になっている。つまり、存在しないアドレスへジャンプしようとしている場合。
3. 他のクラスファイルとの一貫性
   クラスファイルの中の項目の関係に矛盾がなくても、他のクラスの構造と矛盾していると、ベリファイアはパスできない。
   - CONSTANT __ Class エントリで参照されているクラスが存在しない場合。
   - CONSTANT __ Fieldref エントリで参照している別クラスのフィールドが存在しない場合。
4. その他のチェック
   文法的なエラーがなく、すべての項目の関係が一

貫していたとしても、そのコードが安全だということにはならない。ベリファイアをパスするアセンブラプログラムを書くにあたって、特に注意しなければならないのは次にあげるような点である。

- インストラクションのオペランドと、スタックに積まれている値の型にずれがないようにする。
- メソッド引数の型とスタックの値にずれがないようにする。
- オペランド・スタックとローカル変数のサイズの上限を越えないようにする。(メソッドのバイトコードには、そのメソッドで使うオペランドスタックとローカル変数の上限値の情報が含まれているため)
- スタックに値を積むコードとそれを取り出して使うコードとの距離は極力短くする。

2.11 Java 仮想マシンの命令セット

Java 仮想マシンの命令は、実行するオペレーションを定義した 1 バイトのオペコードの後に、引数を提供するゼロ個以上のオペランドと、オペレーションが使用するデータを続けた構成となっている。オペランドを持たず、1 つのオペコードのみで構成される命令も数多く存在している。また、命令のフォーマットダイアグラム中の各セル (枠) は単一の 8 ビットを表現しており、ニーモニックは命令の名前を示している。オペコードは、その数値表現を 10 進数と 16 進数の双方で記載している。Java 仮想マシンの class ファイルのコード中に実際に存在するのは数値表現のみである。

実行時に算出され、オペランド・スタック上に置かれる「オペランド」とは別に、コンパイル時に生成され、Java 仮想マシンの命令内に組み込まれている「オペランド」があるという点に注意しなければならない。こういったオペランドはいくつかの異なった場所で指定されるが、すべて同じもの、すなわち実行している Java 仮想マシンの命令が操作するべき値を表している。例えば、大半のオペランドをオペランド・スタックから暗黙のうちに取得することによって、コンパイルされたコード中に追加のオペランド・バイトやレジスタ・ナンバーとしてのオペランドを明示的に表現するよりも、Java 仮想マシンのコードをコンパクトに保つことができる。

2.11.1 型と Java 仮想マシン

Java 仮想マシンの命令セット中の大半の命令は、実行するオペレーションについての型情報をコード化している。例えば、iload 命令はローカル変数の内容をオペランド・スタックにロードするが、それは int でなければならない。fload 命令は float 値で同じことを行う。この 2 つの命令は同一の実装となる可能性があっても、それぞれが個別のオペコードを保持している。また、型付けされた命令の大部分は、その命令の型がオペコード・ニーモニック中に明示的に文字表現されている。int のオペレーションは i、long は l、short は s、byte は b、char は c、float は f、double は d、参照型は a となっている。

Java 仮想マシンのオペコードが 1 バイトのサイドであることを考えた場合、型をオペコード中にコード化することによって、その命令セットの設計が制約を受けることになる。もしも型付けされた命令それぞれが Java 仮想マシン実行時のデータ型のすべてをサポートするとなれば、命令の数は 1 バイトで表現できるよりも多くなってしまうので、Java 仮想マシンの命令セットはその代わりに、ある種のオペレーションに対して、低レベルな型サポートしか提供しない。いいかえれば、命令セットはあえて直交していない。分離命令を使用することにより、サポートされていないデータ型とサポートされているデータ型との間で必要に応じて交換を行うことができる。

2.11.2 ロード命令とストア命令

ロード命令とストア命令は、Java 仮想マシンのローカル変数とオペランド・スタックの間で値を移動させるものである。

- ローカル変数をオペランド・スタックにロードする。
  iload、lload、fload、dload、aload など
- 値をオペランド・スタックからローカル変数にストアする。
  istore、lstore、fstore、dstore、astore など

- 定数をオペランド・スタックにロードする。
  bipush、sipush、ldc など
- wide 添え字を使用することによって、より多くのローカル変数にアクセスしたり、より大きな即値オペランドにアクセスすることができる。

オブジェクトのフィールドと配列の原始要素にアクセスする命令は、オペランド・スタックへデータを移動したり、オペランド・スタックからデータを移動したりすることもできる。

2.11.3 計算命令

計算命令とは通常、オペランド・スタックにある 2 つの値の関数結果を計算し、その結果をプッシュしてオペランド・スタックに返すものである。計算命令には大きく分けて、整数値の演算を行うものと、浮動小数点数値の演算を行うものの 2 種類がある。これら 2 種類の計算命令はそれぞれ、Java 仮想マシンの整数値に特化している。byte 型、short 型、char 型の整数計算に対する直接的な命令は存在していない。それは、これらの演算は int 型を操作する命令によって処理されるからである。また、整数と浮動小数点数の命令は、オーバーフロー、アンダーフロー、ゼロによる除算でその振る舞いが異なっている。計算命令は以下のようなものである。

- 加算
  iadd、ladd、fadd、dadd
- 減算
  isub、lsub、fsub、dsub
- 乗算
  imul、lmul、fmul、dmul
- 除算
  idiv、ldiv、fdiv、ddiv
- 剰余演算
  irem、lrem、frem、drem
- 符号反転
  ineg、lneg、fneg、dneg
- シフト演算
  ishl、ishr、iushr、lshl、lshr、lushr
- 論理和
  ior、lor
- 論理積
  iand、land
- 排他的論理和
  ixor、lxor
- ローカル変数のインクリメント
  iinc

2.11.4 型変換命令

型変換命令を使用することによって、Java 仮想マシンの数値型間の変換を行うことができる。また、こういった変換を使用することによって、ユーザ・コードで明示的な変換を実装したり、Java 仮想マシンの命令セットにおける直交性の欠如を低減することができる。

Java 仮想マシンは以下のようなワイドニング数値変換、すなわち Java のプリミティブ型へのワイドニング変換のサブセットを直接的にサポートしている。

- int から long、float、double のいずれかへの変換。
- long から float または double への変換。
- float から double への変換。

ワイドニング数値変換命令には i2l、i2f、i2d、l2f、l2d、f2d があり、これらのオペコードのニーモニックは、型付けされた命令のネーミング・ルールとして、2 で 「to」を表す語呂合わせを行った単純なものである。例えば、i2d 命令は int 型を double 変換する。ワイドニング数値変換は数値の全体的な大きさについて情報を損失することはない。実際に、int 型から long 型へ、そして float 型から double 型へというワイドニング変換は、いかなる情報も一切喪失せず、数値は正確に保存される。しかし、int や long の値を float へ、あるいは long な値を double へという変換には精度の欠落が発生する。つまり値の下位ビットを喪失するおそれがある。結果の浮動小数点数値は、IEEE 754 の近似値への丸めモードを使用して整数値を正しく丸めたものである。

また、Java 仮想マシンは、次のようなナローイング数値変換、すなわち Java のプリミティブ型へのナローイング変換のサブセットを直接的にサポートしている。

- int から byte、short、char のいずれかへの変換。
- long から int への変換。
- float から int または long への変換。
- double から int、long、float のいずれかへの変換。

ナローイング数値変換命令には i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l、d2f がある。ナローイング数値変換は、異なった符号や異なった順序、あるいはその双方となることがある。したがって、精度の欠落が発生するおそれがある。　double から float へのナローイング数値変換は、IEEE 754 に従って行う。結果は IEEE 754 の近似値への丸めモードを使用して正しく丸められる。このとき、値が float として表現するには小さすぎる場合は float 型の正か負のゼロに変換され、値が float として表現するには大きすぎる場合は float 型の正か負の無限大に変換される。

2.11.5　オブジェクト生成と操作

クラス・インスタンスと配列はともにオブジェクトであるが、Java 仮想マシンは個別の命令セットを使用してクラス・インスタンスや配列を生成、操作する。

- new
  新たなクラス・インスタンスを生成する。
- newarray、anewarray、multianewarray
  新たな配列を生成する。
- getfield、putfield、getstatic、putstatic
  クラスのフィールド (クラス変数として知られている static フィールド) やクラス・インスタンスのフィールド (インスタンス変数として知られている非 static フィールド) にアクセスする。
- baload、caload、saload、iaload、laload、faload、daload、aaload
  配列構成要素をオペランド･スタックにロードする。
- bastore、castore、sastore、iastore、lastore、fastore、dastore、aastore
  配列構成要素としてオペランド・スタックから値をストアする。
- arraylength
  配列の長さを得る。
- instanceof、checkcast
  クラス・インスタンスや配列の属性を検査する。

2.11.6　オペランド・スタック管理命令

オペランド・スタックを直接操作するためにいくつかの命令が提供されている。
pop、pop2、dup、dup2、dup__x1、dup2__x1、dup__x2、dup2__x2、swap

2.11.7　制御の移行命令

Java 仮想マシンは制御の移行命令によって、条件の有無にかかわらず、その命令の次にくるものとは異なる命令から実行を継続することができる。制御の命令は次の通りである。

- 条件分岐
  ifeq、iflt、ifle、ifgt、ifge、ifnull、ifnonnull、if__icmpeq、if__icmpne、if__icmplt、if__icmpgt、if__icmple、if__icmpge、if__acmpeq、if__acmpne、(lcmp、fcmpl、fcmpg、dcmpl、dcmpg)
- 複合条件分岐
  tableswitch、lookupswitch
- 非条件分岐
  goto、goto__w、jsr、jsr__w、ret

Java 仮想マシンは int、long、float、double、参照型のデータを比較して条件分岐する個別の命令セットを保持している。byte、char、short 型のデータの比較は int の比較命令を使用して行う。このように、int の比較はより重要であるため、Java 仮想マシンでは int 型に対する条件分岐命令の数の方が、他の型に対するものよりも多くなっているのである。また、Java 仮想マシンは、null 参照に対して判定を行う条件分岐命令を個別に保持しているため、null に対しては具体的な値を規定をする必要はない。

2.11.8　メソッド呼び出しとリターン命令

メソッドを呼び出す命令には以下の 4 種類がある。

- オブジェクトの (仮想の) 型によってディスパッチを行い、オブジェクトのインスタンスメソッドを呼び出す
  invokevirtual(これは Java における通常のメソッド・ディスパッチである)
- 特定の実行時オブジェクトが実装しているメソッドの中から適切なメソッドを検索し、インターフェースが実装しているメソッドを呼び出す
  invokeinterface
- 特殊な取り扱いを要求するインスタンス・メソッド、インスタンス初期化メソッド¡init¿、private メソッド、スーパークラス・メソッドのいずれかを呼び出す
  invokespecial
- 名前付けされたクラスによってクラス (static) メソッドを呼び出す
  invokestatic

戻り型で識別されるメソッド・リターン命令には、ireturn(byte、char、short、int 型の値を返すために使用する)、lreturn、freturn、dreturn、areturn がある。さらに、void 宣言しているメソッドからリターンするための命令として return 命令がある。

2.12 例外

Java プログラムが Java 言語のセマンティックス (意味のこと) 上の制約に違反した場合、Java 仮想マシンはこのエラーを例外としてプログラムに通知する。

2.12.1 例外処理

ここでは、例外処理が Java 仮想マシンレベルでどのように行われているのかを解説する。例外が発生すると、Java 仮想マシンはそのメソッドに定義されている例外ハンドラを探す。例外ハンドラとは、Java 言語における try-catch 文である。もし、カレントメソッドで例外ハンドラが見つからないときは、Java スタックからフレームを 1 つ pop して、次のフレームのメソッドで定義されている例外ハンドラを探す。これを、例外ハンドラが見つかるまでこれを繰り返す。すべてのフレームを pop しても例外ハンドラが見つかなければ、最終的にはシステムの例外ハンドラが呼び出される。通常システムハンドラは例外情報をコンソール出力するだけである。

2.12.2 例外を発生させるインストラクション

例外を発生させるには、athrow インストラクションを使う。これは、Java 言語の throw 文にあたるものである。

2.12.3 例外の宣言

Java 言語メソッドの throws 節で宣言する例外は、Java 仮想マシンメソッドの Code 属性の Exceptions 属性にコンパイルされる。実は、Java 言語では、throws 節での宣言は必須だが、Java 仮想マシンレベルでは、throws 節での宣言はしてもしなくても動作に支障はない。それは、throws 節は、Java 言語である特定のコーディングスタイルを推奨するために導入された言語規則であり、Java 仮想マシンレベルでのセマンティクスは持たないからである。なので、バイトコードレベルやアセンブラレベルでこのあたりをいじってしまうと、逆コンパイラで生成した Java 言語プログラムを再コンパイルしても、そのプログラムはコンパイルエラーになってしまう。

3　クラスファイル

Java 仮想マシンのクラスファイルと、Java 言語のクラスは 1 対 1 で対応する。この 1 対 1 の対応関係は、Java 仮想マシンがクラスファイルをメモリにロードした後も変わらない。1 つのクラスファイルは、1 つのクラスとしてメモリに展開され、メモリに展開されたクラスは、java.lang.Class クラスのインスタンスとしてアクセスされる。この、Java 言語のオブジェクトと Java 仮想マシンのオブジェクトの 1 対 1 の対応関係は、メソッドにもあてはまる。クラスファイルのバイトコードはいくつかの区画に分かれており、そのうちメソッドが定義されている区画の構造を見ると、Java 仮想マシンのメソッドが並んでいることがわかる。これを見ると、Java 言語のメソッドと、Java 仮想マシンのメソッドは 1 対 1 で対応していることがわかる。また、クラスファイルの主な特徴には次のようなものがあげられる。

- 各クラスファイルには一つのクラス、又はインターフェースの 情報が保持される。
- 基本的に 8bit のバイトストリーム。
- マルチバイトデータ項目は全てビッグエンディアン。
- クラスファイルのデータ構造は C ライクな疑似構造体で定義される。
- 疑似構造体の記述に使う、u1、u2、u4 は、それぞれ符合なしの 1 バイト、2 バイト、4 バイト数を表す。

そして、全てのクラスファイルは、必ずただ一つのクラスファイル構造体より構成されている。

3.1　クラスファイルの最上位構造

3.1.1　magic

クラスファイルの最初の 4 バイトは magic といい、その値は 16 進で「CA FE BA BE」と決まっている。すべての Java 仮想マシンクラスファイルは、この 4 バイトから始まる。先頭が「CA FE BA BE」でないファイルは Java のクラスファイルではなく、ベリファイアをパスすることはできない。

3.1.2　minor __ version と major __ version

クラスファイルを生成したコンパイラのバージョン番号である。メジャーバージョンの違いは大きな違いであり、マイナーバージョンの違いは小さなちがいである。たいていの Java 仮想マシンは 1 つのメジャーバージョンと、ある一定の範囲内のマイナーバージョンをサポートしている。

3.1.3　constant __ pool __ count と constant __ pool

constant __ pool は配列である。この配列には、constant __ pool __ count-1 個のコンスタントプールエントリ (以下、CP エントリ) が含まれている。

1 つのコンスタントプールエントリの構造は、cp __ info 構造体で表される。cp __ info 構造体のバイト長は一定ではない。cp __ info 構造体の詳細については、で説明する。

コンスタントプールエントリは、番号 (index) で参照される。コンスタントプールエントリの index は 1 から始まり、constant __ pool __ count-1 で終わる。index が 0 のコンスタントプールエントリは、クラスファイル中にはない。index が 0 のコンスタントプールエントリは、Java 仮想マシンランタイムが内部的に使用するために予約 (reserve) してある。

3.1.4　access __ flags

アクセスフラグはクラスだけではなく、フィールドやメソッドにもあり、これらのアクセスフラグはすべて 2 バイトで表される。access __ flags と複数形になっていることからもわかるように、この 2 バイト中に複数のアクセスフラグが詰め込まれている。1 つのビットが 1 つのフラグを意味する。

ACC __ PUBLIC、ACC __ FINAL、ACC __ ABSTRACT については、Java 言語の public、final、abstract と同じである。例えば、クラスを宣言するときに public と宣言してあれば、ACC __ PUBLIC ビットが立つ。これは、final、abstract についても同様のことがいえる。

3.1.5 this __ class と super __ class

this __ class と super __ class のどちらも、コンスタントプールのエントリ番号がしまわれている。このクラスと、このクラスのスーパークラスへのクラス参照のエントリ番号である。したがって、これらのエントリはどちらも CONSTANT __ Class である。java.lang.Object の場合は、スーパークラスを持たないので、super __ class の値は 0 であり、インターフェースの場合は、この値は常に java.lang.Object への参照を格納する CONSTANT __ Class エントリのエントリ番号である。

3.2 attribute __ info

attribute __ info とは属性の基本フォーマットであり、全ての属性はこのフォーマットに従わなければならない。逆に言えばこのフォーマットに従っており、かつ定義済み属性で予約されていない有効な名前を持つ属性ならば、新たに定義することはユーザーの自由である。

3.3 クラスのみで使われる属性

3.3.1 SourceFile __ attribute(SourceFile 属性)

SourceFile __ attribute とはソースファイルに関する情報を格納する属性であり、この属性はクラスのみに使われる属性である。また、すべての属性に共通するデータ構造は attribute __ info で定義してあり、それを継承する形で SourceFile __ attribute を定義している。

3.3.2 InnerClasses __ attribute(InnerClasses 属性)

InnerClasses __ attribute とは内部クラスに関する情報を格納しており、この属性はクラスのみに使われる属性である。これも SourceFile __ attribute と同じで、すべての属性に共通するデータ構造は attribute __ info で定義してあり、それを継承する形で InnerClasses __ attribute を定義している。

3.4 cp __ info(コンスタントプールエントリ)

コンスタントプールとは定数やクラス名などを保持するためのものである。コンスタントプールは可変長の要素を持つ、0〜constant __ pool までの一種の配列であるが、クラスファイルには 0 番目の要素は含まれていない。これは、0 番のエントリは Java 仮想マシン内部用に予約されており、クラスファイルで使用することはできないからである。同時に 0 番は無効なインデックスであり、これを使用するクラスファイルはベリファイアをパスすることはできない。 また、コンスタントプールは配列として表現されてはいるが、要素が可変長であるため、添字から各エントリへ単純にアクセスすることはできない。

3.4.1 CONSTANT __ Utf8 __ info

CONSTANT __ Utf8 __ info は、それ自体は意味を持たない単なる文字の並びを Utf8 形式で保持している。Utf8 形式は 2 バイトの文字コードを 1〜3 バイトで表現しているので、メモリ消費は最小では 1/2、最大では 1.5 倍になる。なお、CONSTANT __ Utf8 __ info の length は Utf8 形式でのバイト数を表すため、必ずしも文字数とは一致しない。

コンスタントプール内で実際に文字の並びを持っているのは CONSTANT __ Utf8 __ info だけである。それ以外の構造体で文字の並びを必要とするものは、CONSTANT __ Utf8 __ info へのインデックスのみを保持する。

CONSTANT __ Utf8 __ info はそれ自体では『意味』を保持していない。同じ CONSTANT __ Utf8 __ info でも、それを指す側によって String インスタンスの内容になったり、クラス名になったり、或はメソッドディスクリプタやフィールドディスクリプタになったりする。ただし、例えばクラス名として解析された CONSTANT __ Utf8 __ info がクラス名として有効な文字の並びを保持していない場合は、そのクラスファイルはベリファイアをパスしない。その他の場合についても同様のことがいえる。

また、すべてのコンスタントプールエントリはタグバイトから始まる。CONSTANT __ Utf8 __ info の tag の値は常に 1 である。

- CONSTANT __ Integer __ info とは、int 型リテラルを格納するコンスタントプールエントリである。tag の値は常に 3 であり、bytes には int リテラルの値が直接入る。32bit 固定小数点数の数値を保持している。

- CONSTANT __ Float __ info とは、float 型リテラルを格納するコンスタントプールエントリである。tag の値は常に 4 であり、bytes には float リテラルの値が直接入る。32bit 浮動小数点数の数値を保持している。

- CONSTANT __ Long __ info とは long 型リテラルを格納するコンスタントプールエントリである。tag の値は常に 5 であり、64bit 固定小数点数の数値を保持している。

- CONSTANT __ Double __ info とは double 型リテラルを格納するコンスタントプールエントリである。tag の値は常に 6 であり、64bit 浮動小数点数を保持している。

- CONSTANT __ Class __ info とはクラスまたはインターフェースへの参照を格納するコンスタントプールエントリである。クラスそのものの定義が入っているわけではなく、あくまで、そのクラスへの参照を表現したものである。tag の値は常に 7 である。

- CONSTANT __ Firldref __ info とはフィールドへの参照を格納するコンスタントプールエントリである。フィールドそのものの定義が入っているわけではなく、あくまで、そのフィールドへの参照を表現したものである。tag の値は常に 9 である。

- CONSTANT __ Methodref __ info とはクラス (インターフェースではない) で宣言されたメソッドへの参照を格納するコンスタントプールエントリである。これに対してインターフェースで宣言されたメソッドへの参照は、CONSTANT __ InterfaceMethodref __ info に格納される。tag の値は常に 10(十進数) である。

- CONSTANT __ InterfaceMethodref __ info とはインターフェース (クラスではない) で宣言されたメソッドへの参照を格納するコンスタントプールエントリである。これに対してクラスで宣言されたメソッドへの参照は、CONSTANT __ Methodref __ info に格納される。tag の値は常に 11(十進数) である。

- CONSTANT __ NameAndType __ info とはフィールドやメソッドの参照の名前と型をまとめたものを格納するコンスタントプールエントリである。tag の値は常に 12(十進数) である。

CONSTANT __ NameAndType __ info 以外は、それぞれクラス又はインターフェース、フィールド、メソッド、インターフェースメソッドへの参照を表している。CONSTANT __ NameAndType __ info はフィールド参照、メソッド参照、インターフェースメソッド参照から間接的に使われるだけで、それ自体が直接使われることはない。

これらの参照は全てシンボル参照、すなわちクラスやメソッドの名前などを表す文字の並びとして表現されていますが、実際に文字の並びそのものを保持しているわけではない。実際に文字の並びを保持するのは CONSTANT __ Utf8 __ info のみであり、こちら側はそれへのインデックスのみを保持している。

String インスタンス

- CONSTANT __ String __ info は java.lang.String クラスのインスタンスを表す。参照の時と同様にこれ自体は文字の並びを持たず、実際に持っているのは Constant __ Utf8 __ info である。なお、CONSTANT __ String __ info より String インスタンスを取出す時は、内部的に String.intern() メソッドを呼び出して正規化する必要がある。

3.5 field __ info(フィールド)

field __ info とはフィールド定義を格納する。すべてのフィールドは可変長の field __ info 構造体で記述される。field __ info 構造体の項目は、access __ flags(フィールドへのアクセス許可とフィールドの属性を記述するために使用する修飾子のマスクのことである) と、name __ index(name __ index の値は、constant __ pool テーブルへの有効な添え字であり、Java の修飾子として、ストアする Java の有効なフィールド名を表現していな

ければならない) と、discriptor __ index(Java の有効なフィールドディスクリプタを表現していなければならない。フィールドディスクリプタとはフィールドの型を表現した文字列のこと) と、attributes __ count(属性配列の要素数) と、attributes(属性配列) である。

3.6 フィールドでのみ使われる属性

- ConstantValue __ attribute(ConstantValue 属性)
  ConstantValue 属性とは、field __ info 構造体の attribute 属性で使用する固定長属性のことである。ConstantValue 属性は、(明示的に、または暗黙のうちに)static である必要である定数フィールド値を表現している。つまり、ConstantValue 属性には static フィールドの初期値が入り、この属性はフィールドのみに使われる。

3.7 method __ info

method __ info とはメソッド定義を格納している。メソッドでのみ使われる属性には Exceptions __ attribute(Exceptions 属性) と、Code __ attribute(Code 属性) がある。

3.7.1 Exceptions __ attribute(Exceptions 属性)

Exceptions 属性とは、method __ info 構造体の attributes テーブル中で使用する可変長の属性のことである。Exceptions 属性は、メソッドがスローすることができるチェック例外を示している。各 method __ info 構造体中には Exceptions 属性を 1 つだけ記述することができる。

また、メソッドは次の規準のうちの 1 つを満たしている場合にのみ、例外をスローする必要がある。

- 例外が RuntimeException のインスタンスであるか、またはそのサブクラスの 1 つである場合。

- 例外が Error のインスタンスであるか、またはそのサブクラスの 1 つである場合。

3.7.2 Code __ attribute(Code 属性)

Code 属性とは、method __ info 構造体の attributes テーブル中で使用する可変長属性のことである。Code 属性は、単一の Java メソッドやインスタンス初期化メソッド、クラスまたはインターフェースの初期化メソッドに対する Java 仮想マシンの命令と補助的な情報を保持している。また、Java 仮想マシンのすべての実装は、Code 属性を認識しなければならない。Code 属性は各 method __ info 構造体中に 1 つだけ存在する必要がある。

Code __ attribute 構造体の項目は以下の通りである。

- attribute __ name __ index
  attribute __ name __ index の値は、constant __ pool テーブルへの有効な添え字でなければならない。その添え字が示す constant __ pool エントリは、文字列 "Code "を表現した CONSTANT __ Utf8 構造体でなければならない。

- attribute __ length
  項目 attribute __ length の値は、最初の 6 バイトを除いた属性の長さを示している。

- max __ stack
  max __ stack の値は、メソッドが使用するオペランド・スタックの上限値。この値以上の高さにオペランドが積まれることはないということを表す。Java 言語コンパイラが自動計算して生成する。

- max __ locals
  max __ locals の値は、使用するローカル変数配列の要素数を表す。例えばこの値が 26 の時、ローカル変数の 0〜25 を使うことができる。この値も Java 言語コンパイラが自動計算して生成する。

- code __ length
  code 配列 (インストラクションコード列) のバイト数である。

- code
  code 配列 (インストラクションコード列) である。インストラクションコードとその直渡しオペランドが並んだもの。

- exception __ table __ length
  項目 exception __ table __ length の値は、excep-

tion __ table テーブル中のエントリ数を保持している。

- exception __ table
  exception __ table 配列中の各エントリは、code 配列中の例外ハンドラを表現している。

- attributes __ count
  attributes __ count の値は、Code 属性の属性数を示している。

- attributes
  attributes のそれぞれの値は、可変長属性の構造体でなければならない。Code 属性は、オプションとして任意の数に関連した属性を保持することができる。

- start __ pc と end __ pc はどちらも code 配列 (インストラクションコード列) の添え字 (index) であり、pc はプログラムカウンタ (program counter) の略である。ハンドラは start __ pc から end __ pc-1 の間のインストラクションコードが実行しているときに発生した例外を処理する。

- handler __ pc
  実際に例外が発生してハンドラが起動されたときに実行されるコード列の開始地点のプログラムカウンタである。

- catch __ type
  catch __ type には、CONSTANT __ Class エントリの index が入る。この index で示されたクラスの例外が発生したときだけハンドラが起動される。catch __ type の値が 0 のときは、例外のクラスが何であるか関わらず、ハンドラが起動される。

3.8 メソッドの Code 属性で使われる属性

LineNumberTable 属性と LocalVariableTable 属性は、Code 属性の属性である。これを見てわかるように、属性自体が属性を持つことができる。

3.8.1 LineNumberTable __ attribute (LineNumberTable 属性)

LineNumberTable 属性とは、Code 属性の attributes テーブル中で使用するオプションの可変長属性のことである。デバッガ (バグを探し出すために使われるソフトウェアのこと) はこの属性を利用して、Java 仮想マシンの Code 配列のどの部分が、元の Java ソース・ファイル中の指定した行番号に対応しているかを特定することができる。また、LineNumberTable 属性は、任意の順番で Code 属性の attributes テーブル中に記述することができ、さらに、複数の LineNumberTable 属性が Java ソース・ファイルの指定した行を表現するよう記述することもできる。つまり、LineNumberTable 属性は、ソース・ファイルと一対一の関係である必要はない。

3.8.2 LocalVariableTable __ attribute (LocalVariableTable 属性)

LocalVariableTable 属性とは、Code 属性の可変長のオプション属性のことである。デバッガは、この属性を利用して、メソッドの実行中に指定されたローカル変数の値を特定することができる。LocalVariableTable 属性が Code 属性の Code 属性の attribytes テーブル中に存在している場合、任意の順序で記述することができる。また、LocalVariableTable は、Code 属性中のローカル変数ごとに 1 つだけ保持することができる。

3.9 フィールドとメソッドの両方で使われる属性

- Synthetic __ attribute(Synthetic 属性)
  コンパイラなどによって暗黙的に自動生成されたフィールドやメソッドには Synthetic 属性がつく。

4 Java 仮想マシンの動作

4.1 Java プログラムの実行

Java プログラムは Java 仮想マシンによって実行処理される。Java 仮想マシンは、実行時に様々な記憶域処理を行なう抽象的なコンピューターとして定義され、Java 仮想マシン命令が実行可能である。また、一般的に既存のコンピューターシステム上では、Java 仮想マシンはエミュレータとして実現され、実装形態は問われるものではない。一般的なコンピューター上では、Java プログラム実行の際にはエミュレータを起動し、実行する Java プログラムは、その引数として渡される。ここでは例としてクラス foo を実行すると仮定し、典型的な Java 仮想マシンの動作を簡単に説明する。

Java 仮想マシンを実行すると、まずガーベッジコレクタの起動、ヒープ領域の確保等、自らの初期化動作を行なう。初期化の終了後、foo の実行に移るところで、クラス foo がロードされていないことを検出し、クラスがロードされていない場合は、指定されたクラスパスより該当するクラスをロードする。

ロードされたクラス foo は、実行時クラスに展開されリンクされる。ここでまず、Java 仮想マシンはクラス foo の Java クラスファイルとしての妥当性を検証し、ここで問題が検出された場合は、エラーとなり、Java 仮想マシンは終了してしまう。問題が検出されなかった場合は、foo の任意のデータを Java 仮想マシン内部で使用する各種テーブルに登録する。ここで、実装によっては静的にリンクされるシンボル参照の解決を行なうが必須ではない。ここで静的なシンボル参照を解決しない場合は、実行時に必要に応じて解決されていく。

リンクされたクラスは、初期化コードに応じて初期化され、この際にクラス foo にスーパークラスが存在する場合、スーパークラスが初期化されている必要がある。そして、クラスが初期化された後、Java 仮想マシンは main メソッドの実行を開始し、実際にユーザーが動作させたいプログラムが開始されることになる。以後、Java 仮想マシンは必要に応じてクラスのロードを行ないながら foo を実行していく。

クラスファイルには、実行可能なメソッドの Java 仮想マシン（バイトコード）命令配列が含まれている。Java 仮想マシン命令は、組み込まれた命令実行エンジンによって解釈、実行される。実行エンジンは実装により形は様々であり、インタプリター型、コンパイラ型等の種類がある。Java 仮想マシンはスタックマシンであるため、多くの命令は単純に処理を行なうことが可能である。しかしながら、一部の命令は次に示すような複雑な処理を必要とする。

- オブジェクト参照
- メモリ確保
- 例外処理

これらの処理を効率的に行なうことが Java のパフォーマンスを向上させるポイントである。

Java 仮想マシンは命令実行とともにガーベッジコレクタを動作させており、Java 仮想マシンに割り当てられたヒープ領域（メソッド領域を含む場合もある）を監視し、容量が足りなくなった場合にガーベッジコレクションを実行する。ガーベッジコレクタは、プログラム中で必要が無くなったデータを削除し、必要なデータも再割り当てすることでヒープの空き容量確保する。多くの場合、ガーベッジコレクションの実行が Java プログラムの実行の最大のオーバーヘッドになる。

4.2 オブジェクト操作命令

前述の通り、Java 仮想マシンは実行時に命令実行以外に様々な処理を行ない、また、命令実行自体も複雑な処理を行なう。オブジェクト参照の解決に関する処理もその一つである。Java 仮想マシン命令中には、実行時にコンスタントプール内に文字列として表されるオブジェクト参照を、実際のメモリアドレスに変換してオブジェクトにアクセスする必要がある命令が含まれる。ここでは、これらの命令をオブジェクト操作命令と呼び、以下に示す。

- anewarray
- checkcast
- getfield
- getstatic
- putfield

- putstatic
- instanceof
- invokeinterface
- invokespecial
- invokestatic
- invokevirtial
- ldc
- ldcw
- ldc2w
- multianewarray
- new

まず、オブジェクト操作命令に必要となる参照の解決処理について説明する。

4.2.1 命令動作

オブジェクト操作命令の命令動作を説明する。ここでは、例として getstatic 命令の命令動作について説明する。命令列から getfield がフェッチされると、Java 仮想マシンは直渡しオペランドとして 2 バイトのコンスタントプールインデックスを得る。getfield 命令の場合、この直渡しオペランドはコンスタントプール内の CONSTANT __ Fieldref エントリーのエントリー番号を指す。このフィールド参照エントリーには CONSTANT __ Class エントリーと CONSTANT __ NameAndType エントリーのエントリー番号がそれぞれ一つずつ格納されている。CONSTANT __ Class エントリーにはフィールドの所属するクラスへのクラス参照で、Jasmin コードの< field-spec >で指定したクラス名が格納されていて、CONSTANT __ NameAndType エントリーには< field-spec >で指定したフィールド名と< descriptor >で指定したディスクリプタの組が格納されている。この CONSTANT __ Class と CONSTANT __ NameAndType がコンスタントプール内の CONSTANT __ Utf8 エントリーを指し、ここから目的のフィールドを指す Utf8 文字列を取得する。この文字列を使用して、まずクラステーブルよりクラスの検索を行い、該当するクラスのフィールドテーブルより目的とするフィールドの情報を得る。このような文字列による参照をシンボリック参照という。getfield 命令の場合、フィールド情報よりフィールドエントリーのオブジェクト先頭からのオフセットアドレスを取得し、スタック上のオブジェクトの先頭アドレスを示すオブジェクト参照に加算を行い、目的とするフィールドのアドレスを得ている。ここで問題となるのが、フィールドのアドレスを得るまでの処理の過程である。ここではクラス検索およびフィールド検索で使用される検索キーは Utf8 文字列であり、複数のテーブルの中で文字列比較を行いながら、目的となるフィールドを検索していかなくてはならない。このような文字列検索処理は、他のオブジェクト操作命令においても実行しなければならず、これら命令の実行処理が複雑化する主な原因となっている。このように、コンスタントプール中の Utf8 文字列より実際のオブジェクトのアドレスを得ることを、オブジェクト参照の解決という。ここで、オブジェクト操作命令が実行されるたびに参照の解決をしているとすれば、プログラム実行上の大きなオーバーヘッドになってしまうので、それを避ける為に Java 仮想マシンには一度解決されたオブジェクトキャッシュを保存する機能が必要となる。

4.3 配列アクセス命令

Java 仮想マシンにおいて配列に対するアクセスは単純にデータをアクセスするだけではなく、実行時の例外を検出しなければならない。配列にアクセスするための命令を以下に示す。

- iaload
- laload
- faload
- daload
- aaload
- baload
- caload
- saload

- iastore
- lastore
- fastore
- dastore
- aastore
- bastore
- castore
- sastore
- arraylength

配列アクセス命令時に発生する例外と命令動作について説明する。

4.3.1 配列アクセス時に発生する例外

Java ではデータの安全性を確保するために配列データへのアクセス時に、アクセス対象のデータに対するインデックス値や格納するデータのデータ型を監視している。もし、データの格納される配列の長さを超えるインデックス値だった場合、メモリ内データの不正な取得や書き換えを行なうことになってしまい、これはプログラム実行上での安全性を損なう危険があるためである。そのような場合、Java 仮想マシンは例外を発生させる必要がある。また、参照型として定義されている配列に、参照型とは違う型でデータを書き込むことは、ポインタの不正な書き換えになってしまうので、ソースのデータ型と異なるかを比較し、データ型が異なっていた場合は例外を発生させる。

4.3.2 命令動作

ここでは、例として配列アクセス命令の中の配列ロード命令の動作を説明する。

スタック上には配列参照型と配列の要素へのインデックス値が積まれている。配列参照型は配列オブジェクトのアドレスを示し、配列オブジェクトには配列長、配列のデータ型、配列の要素、あるいは配列の要素へのポインタ等のデータが含まれている。これらのデータを取得し、例外のチェックを行ない目的のデータにアクセスを行なう。このとき、複数のアドレス計算とメモリアクセスが必要となってしまう。つまり、配列アクセスに伴う例外のチェックはプログラム実行時のオーバーヘッドになってしまう。これらの処理を最適化することにより、Java 仮想マシンの性能向上が見込める。

5　オブジェクトキャッシュ

オブジェクトキャッシュとは、オブジェクト参照の解決の有無と命令動作に必要なデータを保持することで、以降の命令動作を高速に行なうことが可能である。

5.1　理論

前章で述べてきたように、解決されたオブジェクト操作命令、コンスタントプールエントリーを保持しておくことが出来れば、命令実行の際のオーバーヘッドをなくすことができ、命令実行を最適化することが可能になるが、問題はこれらを保存していく領域をどう確保していくかである。オブジェクトキャッシュは、オブジェクト参照の解決の有無、直接参照、間接参照、命令実行に必要であるデータをキャッシュする領域である。まず、オブジェクトキャッシュがオブジェクト参照の解決を保存する領域であることを説明する。

「4.2. オブジェクト操作命令」で説明したオブジェクト操作命令は、命令の直渡しオペランドとしてコンスタントプールのインデックス値を持っている。このインデックス値は、各々の命令に対して有効なコンスタントプールエントリーを指し示し、ここから目的のオブジェクトが検索されていく。また、命令実行に必要となる情報は、目的とするオブジェクトをメンバーとして持つオブジェクトの構造体に含まれていると考えられる。つまり、オブジェクト操作命令の実行に必要となる情報は、直渡しオペランドであるインデックス値から決定されるコンスタントプールエントリーから得られる。また、一度解決されたコンスタントプールエントリーは、同じエントリーを参照している他の命令に対しても有効であるといえる。そこで、個々のコンスタントプールエントリーに関連付けたデータ領域を用意することで、オブジェクト操作命令実行時に、この領域から命令実行に必要なデータを取得してくることが可能である。これが、オブジェクトキャッシュの考え方である。

前述したように、オブジェクトキャッシュの任意のエントリーはコンスタントプールの任意のエントリーに関連付けられている必要がある。Java 仮想マシン内では、コンスタントプールはクラスに対して一意のものと認識され、各エントリーはインデックス値によって決定できる。つまり、任意のオブジェクトキャッシュエントリーをクラスに固有の数値とコンスタントプールエントリーへのインデックス値に関連付けることで、オブジェクトキャッシュ内のデータ検索を一意に行なえる。

オブジェクトキャッシュへのデータの登録は、最初にコンスタントプールエントリーが解決された際に行なう。キャッシュに登録するデータは、解決されたコンスタントプールエントリーの種類、Java 仮想マシン内で用いられるデータ構造体に依存する。登録されたオブジェクトキャッシュエントリーは、ガーベッジコレクトによってデータの再配置が行なわれるまで有効である。ガーベッジコレクトが実行されると、オブジェクトキャッシュが保持するデータと実際のオブジェクトに関する情報の間で一貫性が保てなくなる場合があるので、ガーベッジコレクトの実行後は、オブジェクトキャッシュ内の全てのエントリーを無効化する操作が必要になる。

オブジェクトキャッシュに配列に関するデータを格納する場合は、配列オブジェクトを指し示す参照型に関連付けられなければならない。クラスに一意なデータとしては、クラスに固有なアドレス値を用いることが典型的な例である。また、参照型は通常はメモリアドレスであることから、上記のように関連付けるキーを選ぶことで、参照の解決データと配列データが衝突することはない。データを登録するタイミングは、配列が生成される際、配列アクセス命令の実行時にキャッシュミスであった場合、配列オブジェクトの構成が変更された場合等が考えられる。

5.2　オブジェクトキャッシュを使用する場合の実行処理の方法

ここでは、オブジェクト操作命令と配列アクセス命令の、オブジェクトキャッシュを使用する場合の実行処理の方法に関して説明する。

バイトコード配列から命令がフェッチされ、それがオブジェクト操作命令、または配列アクセス命令であった場合、Java 仮想マシンはオブジェクトキャッシュ内のデータ検索を行なう。検索用のキーはクラス固有のアドレス＋コンスタントプールへのインデックス値もしくは配列オブジェクトの参照型を用いる。目的データがオブジェクトキャッシュ内に存在し、さらに、そ

のデータが有効である場合にキャッシュヒットとなり、オブジェクトキャッシュよりデータが転送される。目的データがオブジェクトキャッシュ内に存在しない、あるいは目的データが無効である場合は、シンボリック参照を用いた解決、または通常の配列アクセスが行なわれ、命令実行完了後にオブジェクトキャッシュへの登録が行なわれる。以降の命令実行動作は、参照するコンスタントプールエントリーの種別によって異なってくる。

5.2.1 静的なオブジェクトの参照

- getstatic 命令
- putstatic 命令
- invokestatic 命令
- invokespecial 命令（一部）

命令が静的オブジェクトの参照をする場合、図 5 のように動作する。これには、上記の 4 命令が含まれる。オブジェクトが静的なデータである場合、その配置は Java 仮想マシン中で一意に決定できるので、コンスタントプールエントリーが静的なデータを指し示している場合、オブジェクトキャッシュには目的データへのポインタが格納されていれば良い。

コンスタントプールエントリーがフィールド参照の場合は、目的フィールドへのポインタと、データ型がオブジェクトキャッシュより転送されてくる。データ型よりデータ長を判別し、ポインタを使ってアクセスを行なえばよい。メソッド呼び出しの場合は、メソッド構造体のアドレス、引数の数等が転送されてくる。メソッド呼出し命令は、命令完了に複数の段階を要する命令であり、その動作も Java 仮想マシンの実装、内部データ構造体に依存する。

5.2.2 動的なオブジェクトの参照

- getfield 命令
- putfield 命令
- invokevirtual 命令
- invokeinterface 命令
- invokespecial 命令（一部）

命令が動的オブジェクトの参照をする場合、図 6 のように動作する。これには、上記の 5 命令が含まれる。動的オブジェクトを参照している場合、参照するデータはどのインスタンスオブジェクトに属しているかによって位置が異なってくるので、一意に決定するには、スタック上の objectref と関連付けなければならない。そこで、インスタンスオブジェクト構造体を、同一クラスから生成されたインスタンス同士は同一の構造を持つように構成することにより、スタック上の objectref が異なる場合でも、インスタンスオブジェクト内のバイトオフセット値を保持していればデータにアクセスが可能になる。つまり、オブジェクトキャッシュには目的データのインスタンスオブジェクト内のバイトオフセット値が格納されていれば良い。

5.2.3 定数の参照

- ldc 命令
- ldc __ w 命令
- ldc2 __ w 命令

命令が定数の参照をする場合、図 7 のように動作する。これには、上記の 3 命令が含まれる。これらの命令は、スタックに定数をプッシュする。図 7 では、定数、あるいは定数への参照がオブジェクトキャッシュより転送されてくる。文字列定数の場合は定数への参照が使われ、その場合はオブジェクトキャッシュエントリーのサイズに依存する。

5.2.4 クラスの参照

- anewarray 命令
- multianewarray 命令
- checkcast 命令
- instanceof 命令
- new 命令

命令がクラスの参照をする場合、図 8 のように動作する。これには、上記の 5 命令が含まれる。この中で、

anewarray 命令、multianewarray 命令、new 命令は、ヒープ領域確保を行なう命令であり、複雑な処理を必要とする。checkcast 命令、instanceof 命令は、スタック上の objectref の型をチェックする。オブジェクトキャッシュからは、参照すべきデータが複数の場合は、クラス構造体のアドレスと参照するデータへのオフセット値が、単一の場合は、そのアドレスが転送される。

5.2.5 配列アクセス

命令が配列アクセスを行なう場合、図 9 のように動作する。これには全ての配列アクセス命令が含まれる。オブジェクトキャッシュには配列長とデータ型が保持されている。配列要素へのアクセスを行なう場合、まず、配列長とインデックスの比較と型の比較をし、例外のチェックを行なう。例外が発生しない場合、インデックス値を型のバイト長に合わせて左シフトした値と、配列データの先頭アドレスによってアドレス計算を行なうことで、データの存在する先頭アドレスが得られる。

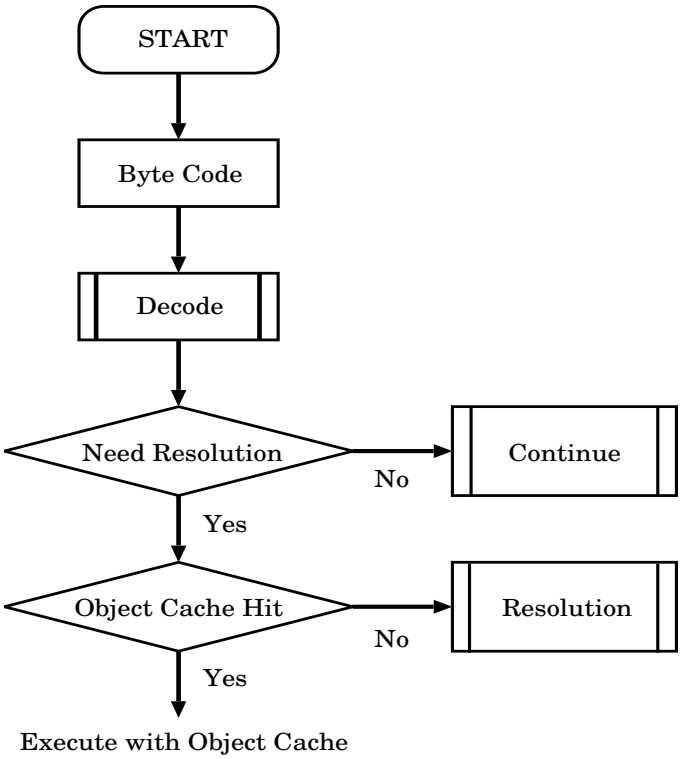
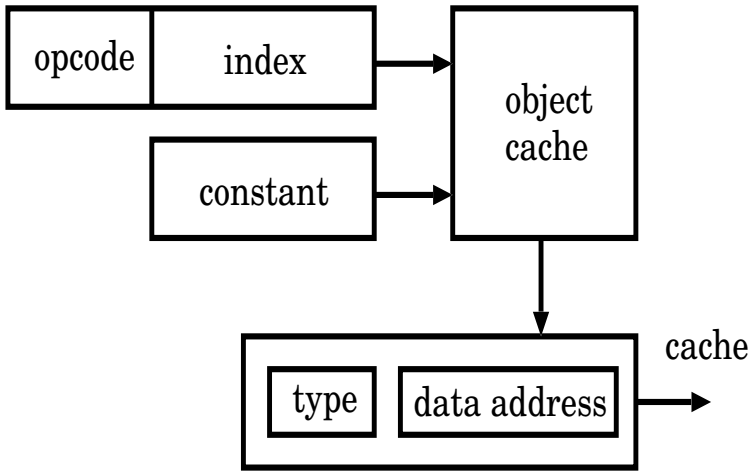
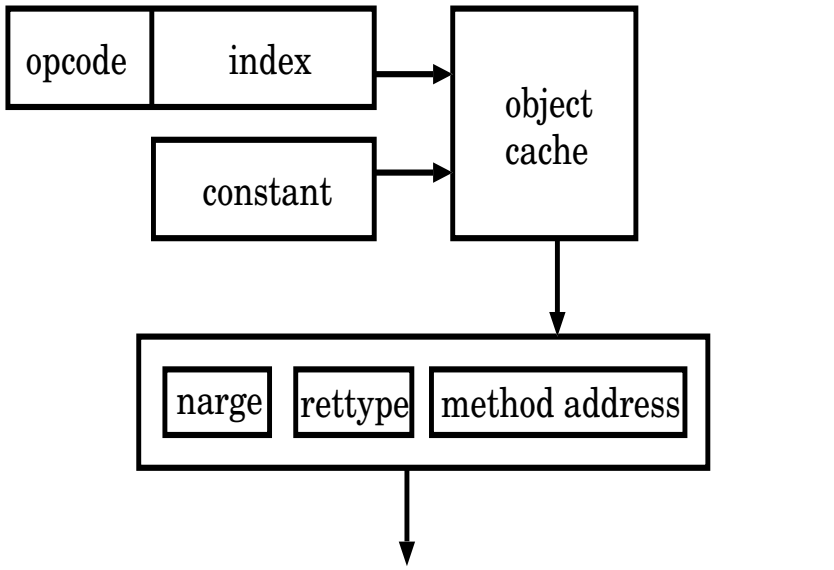


図 4: オブジェクトキャッシュを使用する場合の基本的な流れ



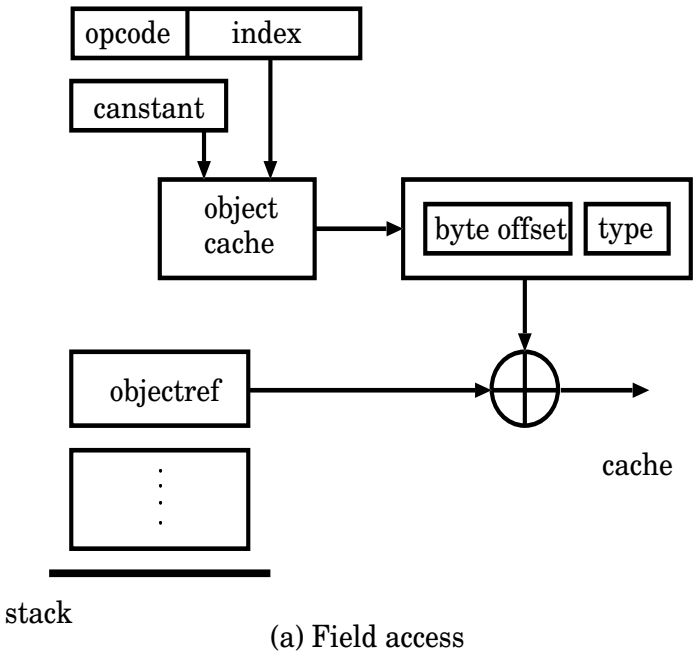
(a) Field access



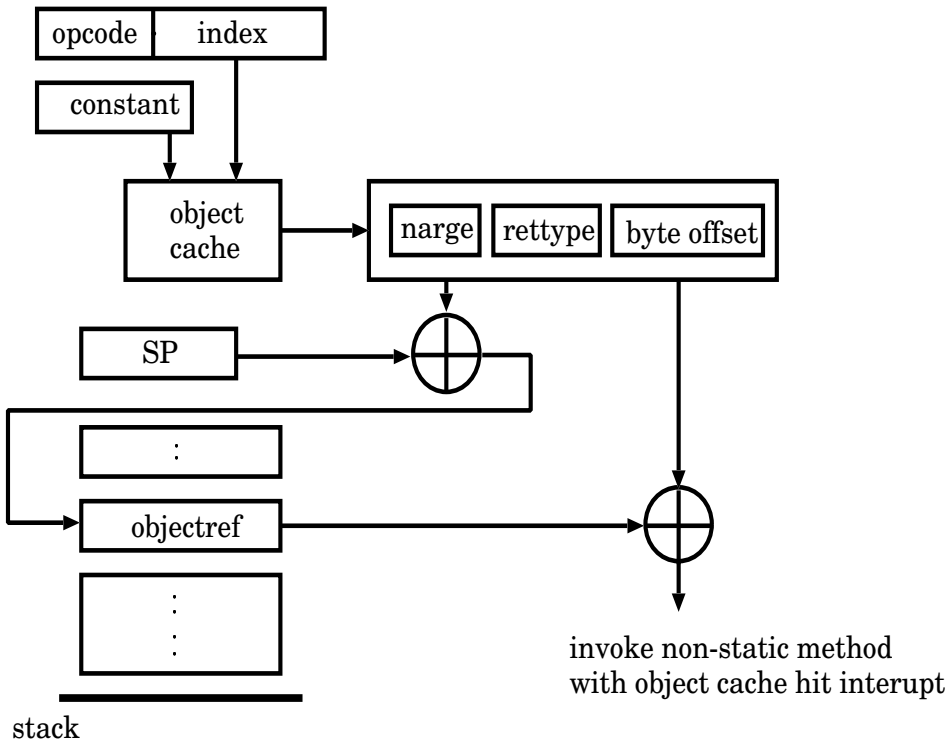
invoke static method with object cache hit interupt

(b) Invoke method

図 5: 静的オブジェクトの参照をする場合



(a) Field access



(b) Invoke method

図 6: 動的オブジェクトの参照をする場合

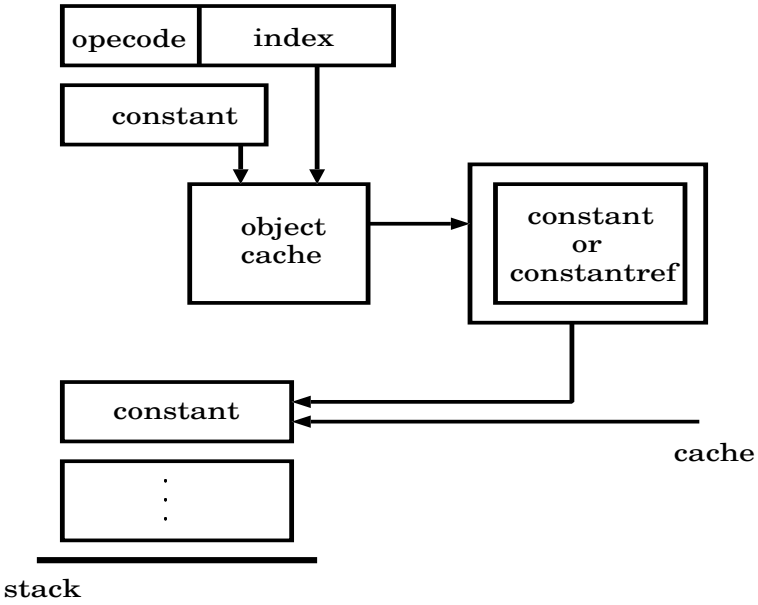


図 7: 定数の参照をする場合

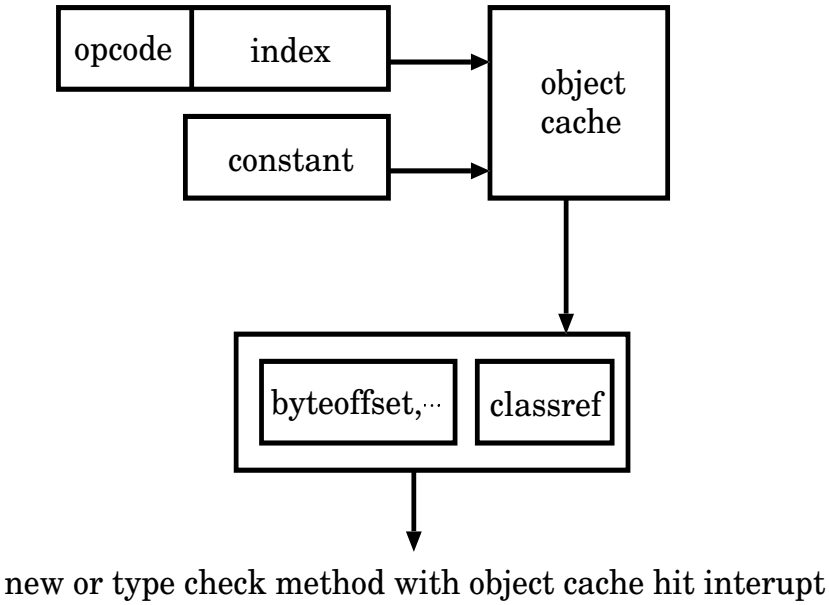


図 8: クラスの参照をする場合

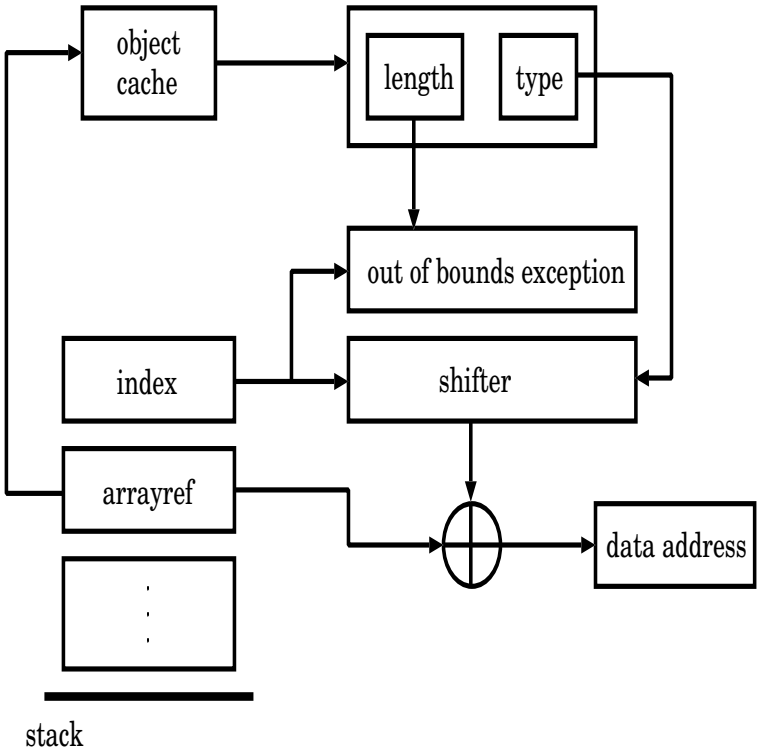


図 9: 配列アクセスを行なう場合

5.3 オブジェクトキャッシュの利点

5.3.1 クラスファイルの静的データ化

今までの参照の解決に対する既存技術である__quick擬似命令やコンスタントプールタグを使用する場合、参照の解決を保存するために、バイトコード配列、あるいはコンスタントプールを書き換える必要があるので、クラスファイルを ROM 化する場合に柔軟なメモリの割り当てが難しくなってしまう。これに対して、オブジェクトキャッシュを使用する場合は、クラスファイルに含まれるデータの書き換えをする必要がなく、メモリの割り当ても柔軟に行なえる。

5.3.2 ガーベッジコレクタ時の初期化が容易

オブジェクトキャッシュを使用する場合、参照の解決に関するデータは全て専用の領域に格納されているので、従来のような上書きされたデータと、そうでないデータの切り分けを行なう必要がない。参照の解決に関するデータを初期化するには、オブジェクトキャッシュ内の全エントリーを無効化すればよい。

5.3.3 データの大きさを制限されない

オブジェクトキャッシュを用いる場合、参照の解決に関する情報を書き込むスペースを制限されることが無く、必要であれば大きなデータを保存しておくこと可能である。

5.3.4 動的オブジェクトの直接参照が可能

「5.3.3」で説明したとおり、データの大きさを制限されないので、解決したオブジェクトメンバーをキャッシュ内に持つインスタンスオブジェクトを示す参照型を保存しておけば、実行時にスタック上の参照型と比較することで、どのインスタンスのデータを判別することが可能である。また、探索キーに参照型を使用することで、動的なオブジェクトの位置を一意に決定することが可能となるので、オブジェクトキャッシュからの直接参照が可能になる。

5.3.5 ハードウエア指向

従来のデータの上書きを必要とする手法は、複数回のデータアクセスが必要となるため、命令の処理動作が複雑になってしまうので、ハードウエアでの実現を考えた場合に性能向上を見込むことは難しいが、オブジェクトキャッシュは従来のキャッシュと同様の機構で実現する事ができ、オブジェクトキャッシュの使用に伴う命令処理の実行もハードウエア的に容易に行なうことが可能である。

6　まとめ

Java 仮想マシンの構造と動作と、オブジェクトキャッシュの構造を中心に述べてきた。

Java 仮想マシンは、Java 仮想マシン命令を実行し、その中で、オブジェクトアクセス、配列アクセス、オブジェクトの参照、メモリの確保、例外の処理のような複雑な処理を行ないながらデータを取得してくる。オブジェクトキャッシュは、一度、Java 仮想マシンが取得したオブジェクト参照の解決の有無、直接参照、間接参照、命令実行に必要であるデータを保存する領域であり、オブジェクト参照のような複雑な処理を最適化するための非常に有効的な手法であることがわかった。

7　謝辞

卒業研究をするにあたって、清水先生をはじめ、研究室の先輩、同期の方々には大変お世話になりました。深くお礼を申し上げます。ありがとうございました。

8　参考文献

1. Meyer J. and Downing T. 著, 鷲見 豊 訳:”Java バーチャルマシン” オライリージャパン (1997)
2. Lindholm T. and Yellin F. 著, 野崎 裕子 訳:”The Java 仮想マシン仕様” アジソン・ウェスレイ・パブリッシャーズ・ジャパン (1997)
3. 近:”Java 仮想マシンにおけるオブジェクトキャッシュ機構の検討と評価および Java プロセッサに関する研究” 2002 年度 東海大学大学院 修士論文
4. http://www.netgene.co.jp/java/documents.html
5. http://www.netgene.co.jp/java/index.html
6. http://e-words.jp/