Ravaで見る Java仮想マシンのしくみ

Rava Rubyで作ったJava VM

東京農工大学大学院工学研究科
ささだ こういち Kohichi SASADA
sasada@namikilab.tuat.ac.jp

はじめに

本稿ではJava仮想マシン（以下JVM）のしくみについて紹介します．ただ紹介するだけではつまらないので，プログラミング言語Ruby [1] で実装したJVMであるRava [2] の解説を交えながら進めていきます．本稿は，Javaプログラムが実際にどうやってJVM上で動いているのか疑問に思っている方を対象にしているため，ある程度Javaプログラムを書くことができるということを前提としています注1．

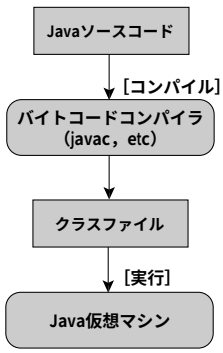
なお，本稿はJVMの詳細な仕様や実装までは述べません．そのため，自分でJVMを作ってみようという方は，必ず『Java仮想マシン仕様』[3] を参照してください（英語では全文オンラインで読めます [4]）．JVMを作ろうとまでは思わない方も，JVMの仕様を勉強することでJavaについての理解が深まると思うのでおすすめです．本稿も基本的にこの仕様に基づいて説明します．また，村山敏清さんが以前本誌で連載していたJVMの解説記事 [5] を含めた，組み込みJavaの情報をWeb上で公開 [6] されていますので，そちらもぜひ参考にしてください．

★ Javaプログラムはどう動く？

ご存知のとおり，Javaプログラムを実行するためには，だいたい図1のような実行手順を踏みます．

Javaで記述されたソースプログラムは，Javaバイトコードコンパイラ（javacなど）でコンパイルすることにより，クラスファイルへと変換されます．そして，クラスファイルのとおりにJVMは動作します．JVMはクラスファイルのみを相手にするため，Java以外の言語で記述されたプログラムでも，それを適切にバイトコードへ変換するコンパイラさえあれば，JVM上で実行することが可能です．

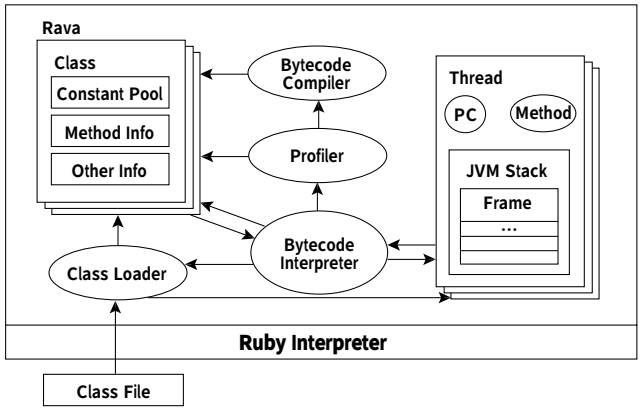
●図1 Javaプログラムが動くまで



☞ 参考文献
[1] オブジェクト指向言語Ruby：http://www.ruby-lang.org/ja/
[2] Rava / JavaVM on Ruby JPサポート用：http://www.namikilab.tuat.ac.jp/~sasada/prog/rava_jp.html
[3]『Java仮想マシン仕様第2版』／ティム・リンドホルム，フランク・イェリン著／村上雅章訳／ピアソン・エデュケーション2001
[4] The Java Virtual Machine Specification：http://java.sun.com/docs/books/vmspec/
[5]「Java仮想マシン入門」／村山敏清著／『JAVA PRESS』Vol.14～23連載／技術評論社
[6] 組込みJava情報源：http://www.netgene.co.jp/java/index.html

注1 偉そうなこと書いてますが，きっと私より皆さんのほうがJava得意だと思います．

●図2　Rava システムの全体像



そもそもなぜJVMが必要なのか，というのは使い古された説明で恐縮ですが，仮想マシンを実際の計算機システム上に被せることでハードウェアアーキテクチャやオペレーティングシステムなどから独立をはかる，つまり，どこでも一緒のプログラムを動かしたいという目的によります．また，JVM自体にセキュリティ機能をもたせることで実行の堅牢性を保証することができます．計算機システムにおいて，仮想マシンのような抽象層を1枚被せるとシステムは10倍遅くなる，というような話もあるそうですが，そのようなデメリットを差し引いても互換性や堅牢性が欲しかったわけです．もちろんJVMの実装技術が発達すれば，プラットフォーム依存のプログラムの実行速度とのギャップは縮まる（もしくは逆転する？）はずです．

★ Rava って何？

Ravaとは，すべてRubyで記述した，Ruby100%，まじりっ気なしのJava仮想マシンです[注2]．

従来JVMはCやC++で実装されてきました．筆者の知る限りでは，ほかにはJavaでの実装 [7] しかありません．その最大の理由は実行速度です．前述したとおり，JVMという抽象層があるためシステムの処理速度が遅くなってしまうので，できるだけ実行速度を速くするようなプログラミング言語が選ばれるわけです．Rubyはお世辞にも処理速度に優れた実行環境があるとは言えません．

では，なぜRubyでRavaを作ったのか？ その動機は，誰もやってなかったからです．作るのが楽しかったからです．結果としては，処理速度としてはやはり遅いものになりましたが，JVMの理解は深まったと思いますし，この記事を書かせてもらうこともできました．

Ravaのシステムの全体像としては，図2のようになります．

★ Rava を実行してみる

何はともあれ，Ravaを実際に動かしてみましょう．Ravaのインストール，セッティングについては本稿サポート用ページ [2] をご覧ください．

セットアップが終了したら，UNIX系OSならばシェル，Windowsならばコマンドプロンプト上で`ruby rava.rb Counter`と入力してください．次のような出力が得られます．

```
$ ruby rava.rb Counter
init Counter
init Counter
init Counter2
1
extended add
extended add
5
```

☞ 参考文献
[7] Jalapeno project：http://www.research.ibm.com/jalapeno/

注2　まだまだ仕様を満たしてない部分もあるのですが．

サンプルに利用したクラスCounterおよびCounter2を記述したソースをリスト1に示します．サンプルは見てのとおりとても簡単なもので，クラス名にもやる気が感じられません．しかし，Ravaがこのサンプルくらいは実行可能であることを示しました．これくらいできなくてJVMを名乗るな，という気もします．

なお，プログラム中のrj.out()メソッドは，Rava用のコンソール出力メソッドだと思ってください．ちなみに，Java 2 SDK（以下JDK）付属のJava環境でjava Counterのように実行しても，同様の出力を得ることができます．

●リスト1　Counter.java

```
// Base Counter Class
public class Counter{
    private int counter;
    public static void main(String[] args){
        Counter cnt  = new Counter();    // オブジェクト生成
        Counter cnt2 = new Counter2();

        rj.out(cnt.add(1));       // メソッド起動
        cnt2.add(2);              // 派生クラスメソッド起動
        cnt2.add(3);
        rj.out(cnt2.get());       // 出力
    }
    Counter(){                    // コンストラクタ
        counter = 0;
        rj.out("init Counter");
    }
    public int add(int a){
        for(int i=0;i<a;i++){     // ローカル変数アクセス・計算
            counter++;            // フィールドアクセス・計算
        }
        return counter;
    }
    public int get(){
    return counter;
    }
}
// Extended Counter Class
public class Counter2 extends Counter{
    Counter2(){                   // 派生クラスコンストラクタ
        rj.out("init Counter2");
    }
    public int add(int a){
        rj.out("extended add");
        return super.add(a);      // 基底クラスメソッド呼び出し
    }
}
```

クラスファイル

JVMがどのように動くのかを記述してあるのがクラスファイルでした．ここでは，クラスファイルにはどんな情報が詰まっているかを概観します．

★クラスファイルの中身を覗く

とりあえずクラスファイルをエディタで開いてみると，よくわからないデータの並びが表示されます．これはクラスファイルがバイナリデータであるためです．そこで，JDKに付属しているクラスファイルのディスアセンブラであるjavapコマンドを利用します．このコマンドはクラスファイルを人間が読める形にしてくれます．

では，Counterクラスのクラスファイルを覗いてみましょう．javap -c Counterと実行してみます（図3）．

javapコマンドによって，❶Counterクラスに存在するメソッドやフィールド，❷各メソッドの動作，などがわかります．

☞ **参考文献**

[8]『オブジェクト指向スクリプト言語 Ruby』／まつもとゆきひろ，石塚圭樹著／アスキー 1999

Column ★プログラミング言語 Ruby

プログラミング言語Rubyは，まつもとゆきひろさんによってデザインされたオブジェクト指向なスクリプト言語です．

Rubyは本格的なオブジェクト指向言語であり，たとえばクラスやその派生などの概念は本誌読者の方には馴染み深いものであると思います．それに加え，Mix-inや特異メソッドなどをサポートしています．そして，Rubyはすべてがオブジェクトです．

インタプリタであるため，コンパイルなどの作業を行わずにRubyスクリプトを実行することが可能です．また，テキスト処理の能力にも優れており，Perlと同じくらい強力です．さらにシンプルな文法と，例外処理やイテレータなどの機構によって，よりわかりやすいプログラミングを可能にしています．その上，ガーベッジコレクタやスレッドなどを備えながら，移植性が高くさまざまなプラットフォームに移植されています．

Rubyの文法を学習するためには俗にいうRuby本[8]（オススメ）どがあります．また，公式ページ[1]から日本語の有用なドキュメントが辿れます．

本稿執筆時点（2003年6月）でのRubyのバージョンは，安定版が1.6.8であり，次期安定版である1.8.0のリリースを準備中という状況です．

❷はメソッド先頭からのオフセットとバイトコード，およびそのバイトコードに関する付加情報の並びとなっています．バイトコードとは仮想マシンであるJVMの命令のことです．たとえば，addメソッドが最初に実行する命令はiconst_0で，その次にistore_2を実行する，という意味になります．

★クラスファイルの構造

では，クラスファイルの構造はどのようになっているのでしょうか．JVM仕様[3]によると，Cライクな文法で表現した場合，リスト2のようになります．

リスト中のu1，u2，u4はそれぞれビッグエンディアンの符号無しの1，2，4バイト整数を表します．細かいところ，面白くないところは省略しますので，詳細はJVM仕様を確認してください．ちなみに，インタフェースもクラスファイルによって表現されます．インタフェースの場合，access_flagsにその旨が記録されます．

まず，magicとはこのファイルがクラスファイルであることを示す数値0xCAFEBABEという値です．おしゃれですね．

コンパイル時に確定する定数や文字列などは，コンスタントプール（constatn_pool）に詰め込まれます．クラスファイルではいろいろなところからこの情報を利用します．たとえば，そのクラスと基底クラスの情報を格納するthis_class，super_classは，実際はコンスタントプールのインデックスとして表現されます．

methodsにはメソッドの数だけエントリがあり，それぞれのエントリが1つのメソッドの情報を格納しています．メソッドの情報にはそのメソッドがどのようなバイトコード列を持つか，例外が発生した場合どうするか，などの情報が格納されています．

fieldsやinterfacesも見ればわかりますね．どんなフィールドがあるか，どんなインタフェースが定義されているか，という情報です．

★クラスファイルのロード

やっとRavaの話です．Ravaはもちろん実行時にクラスファイルをロードします．

JVM仕様としては，クラスファイルが正しいものであるかどうか，検査（Verify）しなければならず，その項目も決められています．これは悪意のあるものがクラスファイルを改竄してJVMが不正使用されるようなことを防ぐための機構です．

しかし，Ravaではそのようなチェックを行っておりません．扱うクラスファイルはすべて信頼できるものとして実行します．これはネットワーク越しでクラスファイルをロードするようなことがないという前提でRavaを開発していたからです[注3]．

●図3　javapの実行結果

```
$ javap -c Counter
Compiled from RavaTest.java

class Counter extends java.lang.Object {
    public static void main(java.lang.String[]);
    Counter();
    public int add(int);
    public int get();
}                                                    ❶
        :
    (中略)
        :
Method int add(int)
   0 iconst_0
   1 istore_2
   2 goto 18
   5 aload_0
   6 dup
   7 getfield #9 <Field int counter>
  10 iconst_1
  11 iadd
  12 putfield #9 <Field int counter>                 ❷
  15 iinc 2 1
  18 iload_2
  19 iload_1
  20 if_icmplt 5
  23 aload_0
  24 getfield #9 <Field int counter>
  27 ireturn
```

●リスト2　クラスファイル構造体（C言語ライクな表現）

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

注3：一番の理由は面倒くさかったから，ということなんですが．

そのため，クラスロード時の一大イベントである検証のプロセスはすっ飛ばして，クラスをロードします．

クラスファイルはバイナリファイルということは先ほど述べました．Rubyなどのスクリプト言語はバイナリファイルを扱うのが苦手である，というイメージがあるかもしれませんが，それがそうでもありません．Rubyの場合，`String::unpack`や`Array::pack`メソッドを利用してバイナリデータを簡単に扱えます．

また，クラスファイルをロードしたときに基底クラスがロードされていなかった場合，基底クラスもロードします．これによって，クラスの派生関係をRava内部で正しく把握することができます．

解釈実行

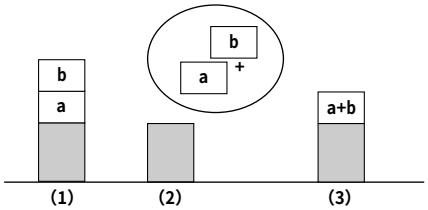
クラスファイルを読み込んだら，それを解釈して実行します．基本的には例外を無視すれば，次のようなループを実行します[注4]．

```
do{
    PCが指す場所の命令を取得;
    取得した命令の操作を実行;
}while(まだすることが残されているか?);
```

PCは実行すべき命令位置を示すもの（Program Counter）です．JVMの主な仕事はこれだけです．JVMって簡単でしょ？[注5]

本節では以降，JVMの実行に不可欠な要素を概観していきます．

●図4　スタックマシンで足し算（a+b）を行う様子



(1) 値a，bをスタックに積む
(2) 値a，bをスタックから取り出し，a+bを計算する
(3) 値a+bをスタックに積む

★スタックマシン

JVMはスタックマシンとして実現されています．スタックマシンとは，演算する対象や演算した結果をスタックに保存するアーキテクチャです．たとえば足し算をする場合，スタックから2つの値を取り出し，それらを足し合わせた結果をスタックに積みます（図4）．

JVMではこのような計算をオペランドスタックというスタックを用いて実行します．

RavaではオペランドスタックをRubyの配列で実現しました．Rubyの配列にはスタック操作をするための十分な機能がサポートされているためです[注6]．

★型とデータ表現

Javaの型はおおまかに言って，プリミティブ型とリファレンス型に分かれます．要するに数値などのデータか，オブジェクトか，ということです．

JVM上では型に応じた命令セットを持ちます．ただし，`byte`，`short`，`char`型については，だいたい`int`型と同じ命令によって計算されます．

RavaではプリミティブはそのままRubyの数値型で表します．リファレンス型，たとえばJavaのオブジェクトは，フィールド領域とそのクラスへのリファレンスを持つRubyオブジェクトとして表現しました．フィールド領域はフィールド名をキーとするハッシュによって表現しました．

★JVMの命令セット

JVMでは，200もの命令が規定されています．これらを詳細に述べていくと，誌面がすぐに尽きてしまうので，どのような命令セットであるか，JVM仕様にある命令セットのカテゴリを紹介します．

- ロード命令とストア命令
- 算術命令

注4：JVM仕様の記述と若干変更してあります．
注5：これ以外の主でない部分が面倒なんですけどね．
注6：たとえばJavaの配列は固定長ですが，Rubyの配列は可変長です．要するにRubyの配列が`java.util.ArrayList`のようなものということです（もっと高機能ですが）．効率は犠牲になりますが，プログラミングは楽になります．

- 型変換命令
- オブジェクト生成と操作
- オペランドスタック管理命令
- 制御の移行命令
- メソッド起動とリターン命令
- 例外のスロー
- finallyの実装
- 同期化

各命令は1バイトのオペコードに続いて，引数，あるいは命令が用いるデータとなるオペランドを0個以上続けたものから構成されます．中には可変長のオペランドを持つ命令もあり，なかなか凶悪です．

余談ですが，仮想じゃない機械の命令セットは，命令のビットパターンで，この命令はどういう処理をする，ということがわかるそうですね．命令デコードが簡単になるからだと思います．が，JVMの命令セットはただ機能ごとに番号を振っていっただけなのでそのようになっていません．以前JVMをチップ上に載せようとしていた人が嘆いていました．

★メソッド起動

メソッドを起動すると，各スレッドが1つずつ持つJVMスタック上に“フレーム”を作ります．フレームを利用してJVMはメソッドを実行します．

フレームにはそのメソッドで利用する“ローカル変数領域”とメソッド実行用の“オペランドスタック”，そしてそのメソッドのクラスに対する実行時コンスタントプールへの参照が含まれます．

メソッドが終了（リターン命令による終了や例外発生による途中終了）したら，JVMスタックからそのメソッドが構築したフレームが取り除かれ，呼び出し元メソッドの実行が再開されます．値を返す場合，呼び出し元メソッドのオペランドスタックへ値を積みます．

Ravaでは具体的に図5のようなフレームを作ることにしました．まずローカル変数領域があり，その上に呼び出し元メソッドのフレームを復帰するためのフレーム情報を積み，その上をオペランドスタックとして利用するようにしました．JVMスタックが1つの配列で表現されており，フレーム，そしてオペランドスタックをその配列の一部として実現しました．

メソッドが終了すると，フレーム情報を利用して呼び出し元メソッドのフレームを復元します．

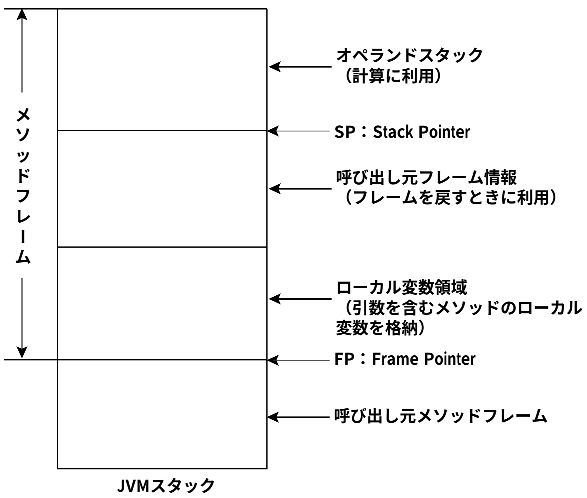
★ネイティブメソッド

JVM命令セットには，入出力（環境依存の操作）やスレッドの制御（VMの操作）などを行うための命令はありません．これらを実現するためにJava仕様，およびJVMにはネイティブメソッドがあります．

ネイティブメソッドとはJava以外の言語によって記述されたメソッドを実行するためのもので，たとえばC言語で記述したメソッドによって入出力を実現したり，JVMのスレッド生成を行います．Javaプログラム中，メソッド名の前に`native`として記述すれば，コンパイラはそれをネイティブメソッドとして認識し，クラスファイルにもそのように記述されます．ネイティブメソッドをどのようにJVMとバインドするか，などはOSのローダなどとの兼ね合いから実装依存となっています．また，ネイティブメソッドをC言語などでどのように記述すればよいかを定めたAPIであるJNI（Java Native Interface）[9] という仕様もあります．

JVM仕様では，どのようにネイティブメソッドを記

●図5　Ravaのメソッドフレーム



☞ 参考文献
[9] Java Native Interface：http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jni/

述し実行させるかということを規定していないため，RavaではネイティブメソッドをRubyで記述できるようにしました．つまり，Javaプログラム中でネイティブメソッドが起動されたとき，対応するRubyメソッドを起動するのです．

このおかげで従来はC言語などで記述しなければいけなかったネイティブメソッドがRubyで簡単に記述できるようになりました．しかし逆に，ネイティブメソッドに関する過去の資産が使えません．そのため，たとえばウィンドウシステムを利用するようなJava APIはRavaではサポートしてません．

ちなみにリスト1の`rj.out`メソッドはネイティブメソッドを利用する出力メソッドです．

★例外処理

Javaには例外処理機構があります．`try/catch`で挟むあれです．`try`節で発生した例外は，その例外が`catch`節でフックされていれば，その`catch`節が実行されます．また，`finally`節がある場合，例外が起こったか起こらなかったかに関わらず`finally`節が実行されます．`catch`されない例外は，呼び出し元メソッドへ例外を伝播します．

さて，JVMではこれを“例外テーブル”（表1）というものを利用して実現します．例外テーブルは各メソッドそれぞれが保持しています．例外テーブルによって，“どこで例外が起こったか”，“どの例外が発生したか”という情報をキーとして，“どのような処理を行う”という情報を引き出し，例外処理機構を実現します．

●表1　例外テーブルの例

start_pc	end_pc	catch_type	handler_pc
10	20	A	100
10	20	B	200
…	…	…	…

※オフセット10～20を実行中に例外Aが発生したら処理を100に，Bなら200に移す

☞ **参考文献**

[10]『Rubyソースコード完全解説』／青木峰郎著／まつもとゆきひろ監修／インプレス 2002

Ravaではこの動作をRubyの例外を利用して実現しました．つまり，解釈実行のループ中でJavaの例外が発生した場合，Rubyの例外を発生させてしまいます．ループから出た時点ですべてのRubyの例外を補足し，現在実行中のJavaメソッドの例外テーブルを検索します．もし該当エントリがあればその情報を元にJVMの実行を再開します．なければメソッドフレームをさかのぼって検索を続けます．

★GCやスレッド

ガーベッジコレクション（GC）は不要なメモリをシステムが勝手に回収し，再利用可能にする機能です．スレッドは実行する命令流を仮想化し，ソフトウェアで複数制御できるようにしたものです．

なぜこの2つを並べて書いているかというと，Ravaではこれらの実装をRubyのGCおよびスレッド機能でそのまま実現しているためです．たとえば，Javaのオブジェクトを`new`すると，RavaはRubyのオブジェクトを生成します．そのJavaオブジェクトが必要なくなったとき，RubyのGC機能がいいようにしてくれます．楽です．

スレッドは，Javaのスレッドが生成されたとき，Rubyのスレッドを生成して，その上でバイトコードの解釈実行を行います．楽です．

ただし，これらの機能はまだ厳密にJVM仕様を満足しておらず，たとえばJavaのファイナライズやスレッドの同期などはRavaでは未着手です[注7]．特にスレッドの同期に対応していないのはスレッドプログラミングにとっては致命的なので，Ravaではスレッドはなんちゃってサポートです．

では，RubyでGCやスレッドをどのように実現しているか，そもそもRubyインタープリタはどのように動いているのか，という詳細は文献[10]を参照してください．

もっと速く！

前節までで説明した範囲を実装することで，Ravaをとりあえず動かすことができました．しかし，実行

注7：ごめんなさい．面倒くさかったんです．

速度が涙が出るほど遅いものになってしまいました．

これは，Ravaがスタックマシンの挙動を非常に素直に，悪く言えば馬鹿正直に実行しているためです．なんとかならないものでしょうか．

世の中ではさまざまなJVMの高速化手法が考案されています．そこで，それらを少し紹介してみます．

JIT（Just-In-Time）コンパイラによる高速化は，実行時にバイトコードをコンパイルする手法です．一般的に，バイトコードを実行環境の機械語列へ変換することでより高速な実行を可能にします．ただし，メモリ使用量が非常に増えます．ほとんどのJVMはこの機構により高速化を行っているようですが，組み込みJVMなどでは利用可能リソースの関係からこれの適用が難しい場合があるようです．

AOT（Ahead-On-Time）コンパイラは，実行時にコンパイルを行うJITコンパイラと違い，実行前にさっさとコンパイルをやっておこうというものです．実行前にコンパイルを終わらせてしまうため，コンパイル時間を縮小できるという利点があります．

Column
★静的な言語，動的な言語

Rubyでは，evalメソッドがあったり，クラスを実行時にいじったり，メソッドを上書きできたりと，動的で柔軟な言語です．Javaはそのような点で静的な言語と言えると思います．

動的な言語では，プログラミングの幅が広がります．たとえば，本稿で述べた高速化の試みでは，動的にメソッドをクラスへ追加するという実装を行いました．

しかし，すべてが実行時まで決まらないため，Rubyは非常に最適化がしにくい言語になっています．たとえば，1+1というRubyプログラムを，パースした時点で2とすることが出来ません．これは，足し算を行うときに呼ばれるFixnum::+というメソッドの挙動が実行時に変更される可能性があるためです．

Javaは，Rubyに比べるとコンパイル時にほとんどのことが確定するため，最適化に適したつくりと言えます．プログラミングのしやすさを取るか，実行効率をとるか，そのバランスの違いがRubyとJavaとの違いの1つなのかな，と思います．

SunのHotSpotVMなどでは，実行時プロファイリングによって何度も実行する部分（HotSpot）のみをコンパイルしようというアプローチをとっています．たとえば初期化のために一度しか呼ばれないメソッドをコンパイルするよりは，何度も実行されるメソッドをコンパイルしたほうが処理速度は向上します．

★Ravaの場合

どうやら，バイトコードをなにかに変換すると実行速度を速くすることができそうです．そこで，実行時にバイトコードをそれと同等の働きをするRubyスクリプトに変換し，それを実行するような機構（JITコンパイラ）を試作してみました．どのバイトコードをコンパイルするかはメソッド起動回数を集計するような簡単なプロファイラを用意し，起動回数によってコンパイルするメソッドを決定するようにしました．

Rubyではevalメソッドによって実行時にRubyスクリプトを評価（Evaluate）することができるため，そのような処理がやりやすかったのです．

この試作では，実行速度がインタプリタのままの性能よりも25倍も向上することができました[注8]（詳細はサポートページ[1]をご覧ください）．

おわりに

いかがでしたでしょうか．JVMはこんなことをやっているのか，と雰囲気だけでも掴んでいただければ幸いです．ついでに，Rubyに興味を持った，使ってみたい，と思ってもらえるとこちらの思うツボです．

言語処理系をいじるのは楽しいです．ちゃんと動いてくれると本当に嬉しいです．処理系上で動作するプログラミング言語への理解も深まります．Rubyだといろいろ簡単に作れますし．まだまだやめられそうにありません．

最後に，本稿執筆にあたり，お世話になった方々，相談にのっていただいた方々，そして最後まで読んくれたあなたに，感謝いたします．JP

注8　でも，JDK 1.3.1の環境にくらべると20倍ほど遅いのですが．要するに元がとんでもなく遅かったのです．