Ruby 処理系のコンパイル済みコードの設計

○笹田 耕一[1,a)]　松本 行弘[1,b)]

概要：
Ruby 処理系（Ruby 2.2）は、Ruby プログラムをスタックマシン型仮想機械（VM）の命令列にコンパイルし実行する。実行時にコンパイルするため、変換に時間がかかる、という問題がある。また、複数のプロセスで命令列を共有することが難しいため、マルチプロセス環境で多くのメモリを消費してしまう、という問題がある。これらの問題を解決するため、事前に Ruby プログラムをコンパイル済みコードに変換し、利用するための仕組みを検討している。本発表では、検討中のコンパイル済みコードの設計について紹介する。

Design of Compiled Code on Ruby Interpreter

Koichi Sasada[1,a)]　Yukihiro Matsumoto[1,b)]

Abstract:
Ruby interpreter (Ruby 2.2) compiles Ruby programs into instrunction sequences for a stack-based virtual machine. Because compilation is done at runtime, it can be an overhead. It is difficult to share Instruction sequences with multiple processes, there are memory consumption problem with multi Ruby processes. To problems these overcome, we are considering about features to compile Ruby program in advance and to use compiled code. In this presentation, we will show you about current design of compiled code.

1. はじめに

本発表では、Ruby スクリプトを事前にコンパイルするためのフォーマットと、その実行方式について、現在検討している内容を報告する [*1]。

本発表は検討中の内容であり、未実装の部分も多いため、結果は予備実験で得られた結果から見積もることで示す。

1.1 背景

オブジェクト指向プログラミング言語 Ruby は、その使いやすさから世界中で利用されているプログラミング言語で、ウェブアプリケーション開発の分野で、とくに多く利用されている。

Ruby 用ライブラリを流通させるパッケージングシステムとして、RubyGems[*2] というツールがあり、利用者は容易にライブラリを用いることができる。RubyGems では、ライブラリのことを Gem と呼称する。RubyGems によって、利用者は容易にライブラリを用いることができ、アプリケーションや Gem 自体が、多くの Gem を利用して開発が行なわれている。

ウェブアプリケーション開発では、Ruby on Rails (RoR) [*3] というウェブアプリケーションフレームワークを用いた開発が最もよく行なわれている。RoR を用いた開発をはじめ、Ruby アプリケーションの開発では、多くの Gem を利用して開発が行なわれる。

1 つの Gem は複数の Ruby スクリプトによって構成されているため、1 つのウェブアプリケーションを実行するために、最終的には大量の Ruby スクリプトをロードする

[1] Heroku, Inc.
[a)] ko1@heroku.com
[b)] matz@heroku.com
[*1] 本報告の最新版は、http://atdot.net/~ko1/activities に掲載するので、適宜参照されたい。
[*2] RubyGems: https://rubygems.org/
[*3] Ruby on Rails: http://rubyonrails.org/

表 1 簡単なウェブアプリケーションでの利用統計

利用している Gem の数	91
読み込んだ Ruby スクリプトの数	1,550
Ruby スクリプトの平均行数	140
Ruby スクリプトの最大行数	2,970

必要がある。メッセージを保存し表示するだけの簡単なウェブアプリケーション [*4] を試作したところ、約 100 個の Gem を用いており、アプリケーションのスクリプトとあわせ、合計で 1,550 個の Ruby スクリプトをロードしている（**表 1**。なお、行数はコメント行や空行なども含む)。

簡単なウェブアプリケーションだけでこれだけのスクリプトが必要になるため、複雑なものでは、この数倍の Ruby スクリプトをロードすることになる。

1.2 課題

このように、Ruby を用いたウェブアプリケーション開発では、多くのスクリプトを用いて実行している。そのことから、起動時間の増大と利用メモリ増が主な課題となる。

1.2.1 起動時間の増大

まず、起動時間の増大が問題となる。プログラムが複雑になり、多くのスクリプトをロードすると、プログラムのロード時間が長くなる。一度起動してしまえば、その後はプロセスを起動しないため問題は無い、という場合もあるが、例えばアプリケーションのテストなどで起動を繰り返すような場合などは、起動時間が大きく作業時間に影響する。

プログラムの読み込みに必要となる実行時間には、いくつかの要因からなる。

(1) ロードするスクリプトの特定
(2) スクリプトのディスクからの読み込み
(3) スクリプトのパース・コンパイル

このうち、ロードするスクリプトの特定には、いくつかの解決策が提案され、利用されている。Ruby では、ロードパスと呼ばれる、読み込むスクリプトを探索するディレクトリを指定する仕組みがあるが、インストール済みの Gem に比例してロードパスが長くなり、多くのディレクトリエントリを探索しなければならず、遅いという問題があった。

この問題については、Bundler[*5] という、実際に利用する Gem を事前に特定しておく仕組みによって、必要以上にロードパスを探索することを抑制し、起動時間を短縮することができるようになった。また、以前の起動時にロードしたスクリプト群のパスを記憶しておき、再起動時にロードパスを用いずに、記憶したスクリプト群を直接ロードする、といった工夫も提案されている。

スクリプトのディスクからの読み込みは、計算機が搭載するメモリ容量にもよるが、ディスクキャッシュに載せる、という、OS による古典的な解決策が期待できる。

最後の、パース・コンパイルについては、現状は bison を用いた構文解析と、Ruby 用 VM[3] のバイトコードへの独自に実装した変換器を用いて行なっている。結果をキャッシュすることはないので、Ruby インタプリタを起動するたびに、すべてのスクリプトについてパースとコンパイルを行なっている。本研究では、コンパイル済みコードを用いることで、この時間の削減を目指す。

1.2.2 利用メモリの増大

次に、メモリ使用量が増えるという問題である。簡単なウェブアプリケーションでも、130MB 程度のメモリを利用するが、その中でも 20MB 程度をスクリプトをコンパイルした結果が占める。**図 1** は、ある時点で確保しているメモリ消費量について、どの関数で割り当てられたかで分け、その上位の値と割合を示している [*6]。`iseq` と名前に入っている関数が、コンパイルして得られたプログラムの表現形式（iseq、Instruction Sequence の略）のために割り当てたメモリ消費量である。以降、この表現形式を慣例に従いバイトコードと呼ぶ。

プロセス単体で見れば、全メモリ消費 130MB 中、バイトコード関連が 20MB 消費しており、全体の 15%程度を占めており、あまり大きな割合ではない。

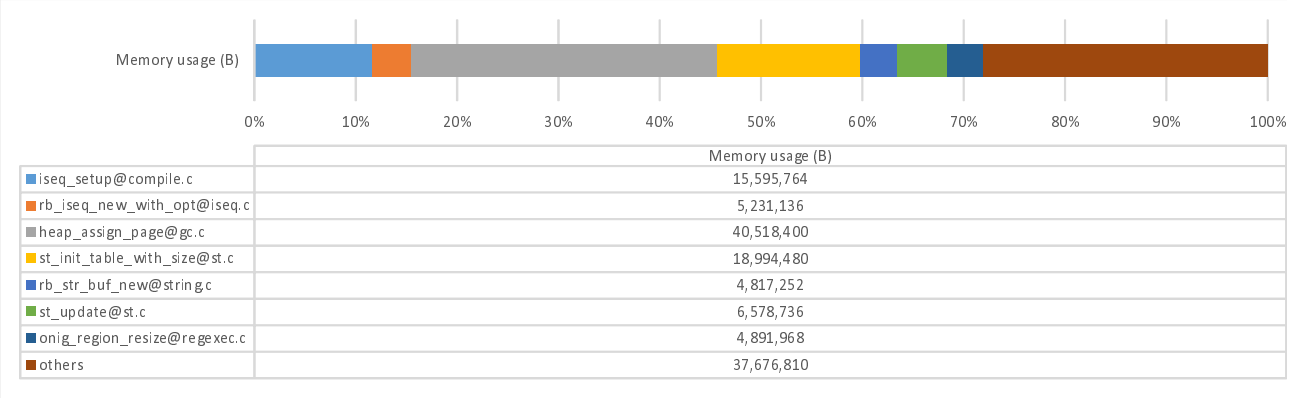
しかし、これが複数プロセス起動した場合、問題になる可能性がある。Ruby では、ウェブアプリケーションを並列処理に対応するためには、複数のプロセスを用いる必要がある。`fork` を用いて、Copy on Write により、書き込みが行なわれない限りメモリを共有することを期待することもできるが、現在のバイトコードのデータ構造は、書き込み領域と読み込み専用の領域を混在して管理しているため、共有ができないこともある。また、`fork` を用いないで複数プロセスを実行する場合は、同じスクリプトであろうとも、バイトコードを共有できない。

また、複雑なアプリケーションでは、この 2 倍、3 倍の数の Ruby スクリプトを読み込むことがある。つまりバイトコードが 40MB や 60MB のメモリを占めることになる。このような複雑なアプリケーションを複数のプロセスで起動させると、最悪の場合、バイトコードの領域のために、その領域のサイズにプロセス数をかけたメモリ消費が必要となる。例えば、8 コアのあるプロセッサを十分に利用するために、16 個プロセスを生成した場合、バイトコードのために 1 プロセスあたり $40MB$ 必要だとすると、$40MB \times 16 = 640MB$ のメモリがバイトコードのために必要と言うことになる。$60MB$ の場合、同じく $960MB$ となる。

[*4] ArigatoBook: ソースコードは https://github.com/ko1/tracer_demo_rails_app、実行は http://protected-journey-7206.herokuapp.com/ で確認できる。

[*5] Bundler: http://bundler.io/

[*6] valgrind/massif というツールを利用して調査した。



	Memory usage (B)
iseq_setup@compile.c	15,595,764
rb_iseq_new_with_opt@iseq.c	5,231,136
heap_assign_page@gc.c	40,518,400
st_init_table_with_size@st.c	18,994,480
rb_str_buf_new@string.c	4,817,252
st_update@st.c	6,578,736
onig_region_resize@regexec.c	4,891,968
others	37,676,810

図 1 簡単なウェブアプリケーションでのメモリ消費量（development モード）

ウェブアプリケーション実行プラットフォームとして、クラウド上の計算機リソースを使うことが多い。このような計算機環境では、メモリなど、計算機資源に応じて利用料金が変更するので、とくに規模の大きなウェブアプリケーションの運用者にとって、メモリ利用量の削減は重要な課題となる。

1.3 アプローチと既存研究

本研究では、起動時間とメモリ使用量の増加への対処のために、Ruby スクリプトをバイトコードへ事前コンパイルを行い、ロードはコンパイル済みコードを読み出すというアプローチをとる。

コンパイル済みコードを利用することで、いくらかの前処理は必要であるが、パース・コンパイルの手順を省くことが期待できる。また、コンパイル済みコードを、`mmap` システムコールなどで直接ファイルの内容をメモリ上にマップし、それをなるべくそのまま利用することで、複数プロセスが、そのコンパイル済みコードのメモリを共有可能とできるようにすることを目指す。

プログラムを事前に実行形式、もしくは中間形式に出力しておき、実行時に利用する、というのは多くの多くの言語処理系で利用されている。事前にコンパイルしておくことから、実行時コンパイル（JIT: Just in Time Compile）と比較して、AoT: Ahead of Time Compile と言うこともある。

Java 仮想マシンでは、アーキテクチャの異なるマシン間で移送可能なクラスファイルという形式を定義している [1]。本研究では、コンパイル済みコードも含め、ほぼ同様のアイデアを利用するが、マシン間で移送可能とはせず、ワード長やエンディアンなどは、コンパイルした計算機上のものを利用する。

mruby[*7] では、すでにバイトコードをファイルから読み込むための仕組みを備えている。mruby と Ruby ではバイトコードの形式が異なるため、そのまま利用することはできないが、発想としては同様のものである。ただし、mruby は、組込みシステムなど、メモリやメモリ管理が十分でない環境のために、コンパイル済みコードを長時間保持しないように、ロード時にコンパイル済みコードからバイトコードをコピーする。ロード終了後には、コンパイル済みコードの領域をメモリから解放することができる [*8]。

池原 [5] は、Ruby のバイトコードを圧縮する方法について提案している。本研究でも、メモリ消費量を削減するため、同等の手法を用いていることができる。ただし、池原の手法は、プロセス外に出すことを考慮していないという点が、本研究の領域とは異なる。

Ruby の高速化を目的に、Ruby のバイトコードを C 言語などを経由して機械語へ変換するというアプローチがいくつか提案・実験されている [2], [4]。本研究の目的は Ruby プログラムの処理速度の向上ではないが、これらと組み合わせることは十分可能だと思われる。

2. コンパイル済みコードの検討

コンパイル済みコードに求められる、もしくは望ましい特徴を述べる。

(1) （必須）ロードして、スクリプトを読み込んだと同様の挙動をすること
(2) スクリプトを変更した場合、コンパイル済みコードを無視、もしくは更新すること
(3) ロードにかかる時間が少ないこと
(4) 他のプロセスと共有できるように、コンパイル済みコードのデータを直接実行時に利用できること
(5) コンパイル済みコードをロードした結果、速度劣化がないこと
(6) メモリ消費量を削減するために、サイズが小さいこと

(1) は、当然のことであるが、スクリプトを読み込ん

[*7] mruby: https://github.com/mruby/mruby

[*8] mruby はプログラムにバイトコードを埋め込む機能を有しており、その場合はコンパイル済みコードを解放せずに、そのまま利用する。

だ場合と同等の挙動であることは必須である。ただし、__FILE__という、スクリプト名自体をどのようにするか、や、スクリプト自体をファイルとしてアクセスするためのDATA という機能をどのように扱うか、という点は検討を要する。本研究では、これらの点を追求しない。

(2) は、コンパイル済みコードが古くなった場合にどうするべきか、という点である。毎回、ファイルを比べ、タイムスタンプを比較する、という手段が直接的な解決策だが、そのためには、オリジナルのスクリプトを探索する方法を提供する必要がある。また、ファイルシステムによっては、タイムスタンプを取得するためにオーバヘッドが必要になる可能性がある。本研究では、これらの点も追求しない。

(3) は、ロード時間短縮という目標に直結する。保存されたコンパイル済みコードを実行可能な形に前処理を行なう必要があるが、これをできる限り短時間にすることが望ましい。ロード処理には、例えばインタプリタプロセスに非依存の情報を、プロセスに依存した形に変換する処理がそれにあたる。

(4) は、他のプロセスとメモリを共有することができるように、実行時に、なるべく情報をそのまま利用することである。この特徴が促進できれば、(3) のロード時間短縮という目的にも大きく寄与する。例えば、mmap システムコールにより、ファイルをメモリにマップするだけで済めば、前処理の時間も少なく、また実際に必要になるまでディスクアクセスを遅延することができる。

(5) は、(4) と相反する。実行時にコンパイル済みコードをなるべくそのまま利用しようとすると、その領域からはポインタでインタプリタ固有の情報にアクセスすることができないため、テーブルなどを用いた間接参照が必要になる。とくに、実行に必要となるバイトコード列が間接参照を行なうと、性能に大きな影響が出る可能性がある。つまり、速いバイトコード列を生成するためには、コピーし、インタプリタプロセス特有の情報を生成する必要があるが、前処理が長くなり、メモリ使用量が増大する、という問題が生じる。

(6) は、ディスクの消費サイズ、およびメモリ消費量の観点から、なるべくコンパイル済みコードのサイズが小さいことが望ましい。しかし、圧縮技術を利用すると、(4) のために、例えばそのまま利用する場合、(5) の実行速度と相反することが多い。ストレージ価格が安くなった現在、ディスクの消費サイズを低くする、という需要はあまりないため、本項目の優先度は低い。

このように、考えるべきパラメータがロード時間、メモリ消費量、処理速度があり、主に処理速度とロード時間、メモリ消費量が相反するというトレードオフとなる。

3. 実装技術の検討

本章では、具体的な実装方法について、現在検討している案についてまとめる。

3.1 コンパイル済みコードのデータフォーマット

コンパイル済みコードは、外部記憶に保存するため、特定のインタプリタプロセスに依存しない、独立した形としなければならない。特定のインタプリタプロセスに依存する情報としては、メソッド名などの識別子として利用するID （同じ名前について、インタプリタプロセスごとに値が変わる）や、実行時に必要となるオブジェクト（インタプリタプロセスごとに、オブジェクトを生成し、そのオブジェクトへ参照を保持しなければならない）などがある。

このような情報を適切にエンコードしコンパイル済みコードとして構成し、実行時にはロードするための仕組みが必要である。Ruby には、このような仕組みとして、すでに Marshal という機能があり、異なる Ruby プロセス間でデータを渡す時などに利用する。データフォーマットには、Marshal、もしくは、それに類するもの [*9] で生成したバイト列を格納する。

ファイルフォーマットは、このような状況を踏まえ、次のようなヘッダ部を持つ。

```
struct ibf_header {
    char magic[4]; /* YARB */
    unsigned int major_version;
    unsigned int minor_version;
    unsigned int size;

    ibf_offset_t iseq_list_offset;
    ibf_offset_t id_list_offset;
    ibf_offset_t object_list_offset;
};
```

最初の 3 要素は、バイナリフォーマットを識別するためのシグネチャである。size は、このコンパイル済みコードの長さである。

iseq_list_offset は、格納されているバイトコードの数だけエントリを持つテーブルへのオフセットであり、各エントリは、バイトコードの情報の先頭が、コンパイル済みコードのオフセットを示す。

わかりづらいので、C 言語のソースコードで示すことにする。3 番目のバイトコードの情報が欲しいときには、次のようなプログラムによって得ることができる。

```
char *code = コンパイル済みコードバイナリ列;
```

[*9] バイトコードが含む一部のオブジェクトが、Marshal では対応できないことがわかっている。また、マシンや Ruby のバージョンに非依存の形で出力するため、いくらか効率が悪い。そのため、これらの問題を解決する別の実装が必要と考えている。

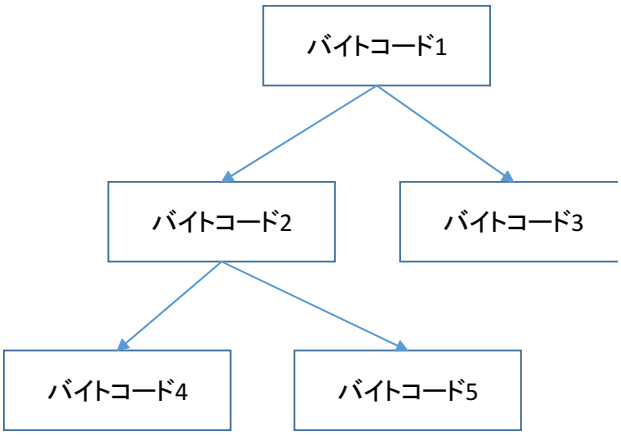


図 2 バイトコードを全てロードした場合

```
struct ibf_header header = (HEADER *)&code[0];
ibf_offset_t *offsets =
  code[hader->iseq_list_offset];
bytecode_header *bheadr =
  ode[offsets[3]]; /* 3 番目 */
```

0 番目のバイトコードがトップレベルのバイトコードであり、これを起点にバイトコードをロードしていく。

`id_list_offset` と `object_list_offset` も、同様に ID、オブジェクトを示すバイト列（`Marshal` などで生成）へのオフセットのテーブルである。

この形式への変換、およびこの形式からのバイトコードの復元は、ほぼ自明であるため、ここでは詳述しない。

3.2 バイトコードの遅延ロード

RubyVM のバイトコードは、スコープごとに別々のバイトコードに別れており、親のバイトコードを実行するために必要となるバイトコードを、子として親が参照する、という木構造となっている（**図 2**）。

Ruby のプログラムは、例外処理や、使われないメソッドといった、一度も実行されないようなバイトコードが存在し、これらを毎回ロードするのは無駄である。そこで、必要になるまでロードを遅延するようにすることが考えられる（**図 3**）。

図 3 では、バイトコード 1 がすでにロードされている状況を示している。バイトコード 1 の実行に必要なバイトコード 2、および 3 は、まだロードしていない、という状態のみで表現（具体的にはバイトコードに対応するオブジェクトを生成）しており、詳細はコンパイル済みコードの内容を確認すれば手に入るようになっている。バイトコード 2 の実行に必要となるバイトコード 4 および 5 の表現はインタプリタ内には存在しない。

実際に、バイトコード 1 の処理が進み、バイトコード 2 が実行される段階になって、その内容をロードする。このとき、バイトコード 4 および 5 を、未ロードのバイトコードオブジェクトとして生成する。

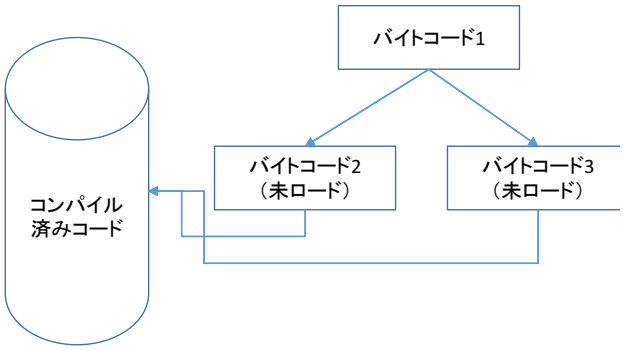


図 3 バイトコードを遅延ロードした場合

この方法では、バイトコードの実行や、情報の参照のたびに、未初期化かどうかのチェックを行なう必要があるため、実行時オーバヘッドが生じる。このオーバヘッドを避けるため、未ロードのバイトコードを実行すると、「ロードして再実行する」という処理にしておくことで、ロード済みのバイトコードを実行するたびに、毎回ロード済みかチェックする、ということは避けられる。

4. 実験と効果の見積もり

本報告の執筆時点では、まだ実装が完了していないため、実装が完了したいくつかの機能をもとに実験を行ない、コンパイル済みコードによる効果を見積もることとする。

なお、評価は Intel i5-3380M (2.90GHz) CPU で動作する Windows 7 上で動作する VirtualBox 上で動作する Ubuntu 14.04.2 LTS で行い、対象とする Ruby は 2015-11-01 trunk 52420 を用いている。

4.1 ロード時間の見積もり

現状、コンパイル済みコードを読み、内容をコピーし前処理することで、バイトコードを復元する版がほぼ完成している。この版では、コンパイル済みコードを複数プロセスで共有する、などといったことができない。また、すべての Ruby スクリプトに対応していないため、実アプリケーションでの評価は行えない。

今回は、この版を用いて、通常通りパース・コンパイルを行なう場合と、コンパイル済みコードを読み込む場合の時間の違いについて調査する。

まず、計測対象となる約 1,400 行の Ruby スクリプトを用意した。スクリプトでは、100 個のクラス定義が記述してあり、各クラスに単純な 3 つのメソッドが定義している。合計で $1(toplevel)+100(classdefinition)+100*3(methods) = 401$ 個のバイトコードが生成される。

このスクリプトを対象に、(1) ファイルから読みこんだ状態からパース・コンパイルし、バイトコードを生成する (2) コンパイル済みコードをファイルから読み込んだ状態から、ロードする、(3) 遅延ロードを利用する、という 3

表 2　ロード時間の実験結果

(1-1) パース・コンパイル	7.05 sec
(1-2) (1-1) に加え、クラス定義を実行	8.42 sec
(1-3) (1-2) - (1-1)	1.37 sec
(2-1) コンパイル済みコードのロード	2.22 sec
(2-2) (2-1) に加え、クラス定義を実行	3.41 sec
(2-3) (2-2) - (2-1)	1.19 sec
(3-1) 遅延ロード	0.00 sec
(3-2) (3-1) に加え、クラス定義を実行	2.06 sec
(3-3) (3-2) - (3-1)	2.06 sec

つについて、それぞれ 2,000 回繰り返した時間を計測した。どちらもファイルからスクリプト、もしくはコンパイル済みコードを読み込んだ状態で行なっているので、ファイルシステムの影響は排除している。

実験の結果を**表 2** に示す。(1-1) が 7.05 秒 (2-1) が 2.22 秒となり、パース・コンパイルすることに比べ、ロードするだけのほうが 3 倍程度速い、ということがわかった。また、遅延ロードを用いることにより、実際にはバイトコードのロードは $1(toplevel)+100(classdefinition)=101$ 回しか行なわれないため、読み込み・実行までの時間で比較すると、(1) と比べ、約 4 倍、(2) に比べ約 1.7 倍速いことがわかった。

実験に利用した Ruby スクリプトは、100 個のクラス定義があるという点で、実際よりも複雑であるが、大ざっぱに見積もると、(1) は 1 つのファイルのロードに 3.5ms、(2) では、1.1ms かかることがわかった。

今回は、コンパイル済みコードとデータを一切共有せず、ロード時にすべてコピーし、必要であれば改変する方式をとった。これをコンパイル済みコードとデータを共有する方式になれば、より高速にロードできるようになることが期待できる。

実行時間は、どれもまったく同じバイトコード表現を生成するため、差異はないと判断できる。ただし、コンパイル済みコードとデータを共有する方式を検討していくと、間接参照などが増える可能性があり、速度低下の要因となる可能性がある。

なお、用意した Ruby スクリプトのサイズは 19,050 バイトであり、変換したコンパイル済みコードは 237,536 バイトであった。64bit CPU を利用しており、そのワード幅をそのまま用いているため、サイズが大きくなっている。適切な圧縮方法を用いれば、容易に小さくすることができるが、ロード時間とのトレードオフとなる。

4.2　遅延ロードの見積もり

簡単なウェブアプリケーションに対して、起動してから 10 回接続した後で、生成されたバイトコードの数と、実際に実行された数を確認したところ、それぞれ 30,485 個と 4,698 個であり、およそ 15%のバイトコードしか実行されないことがわかった。つまり、残りの 85% のバイトコードは実際には利用されないということであり、遅延ロードに大きな効果が期待できることがわかった。

これは、エラーなどが起こらない、正常系のみ実行したためだからだと思われる。また、ライブラリのメソッドなどは、一般的に全てを利用しない。例えば、ファイル操作を行なう `FileUtils` というライブラリには、50 のメソッドが含まれるが、その中の 1 つを利用したい場合でも、このライブラリをロードする。この場合、その他の 49 個のメソッドは未使用ということになる。

前節で求めた 1 ファイルあたりのロード時間 1.1ms を表 1 の 1,550 個のファイルにあてはめてみると、$1.1ms \times 1,550 = 1,705ms$ であり、ロード時間が遅延ロードによってその 15%となるとすると、255ms 程度になると予想できる [*10]。

5.　おわりに

本発表では、Ruby にコンパイル済みコードの生成、およびロードする機能を設けることで、ロード時間を短縮し、メモリ削減を目指す方式について検討し、いくつかの予備評価を行なった。

実際のアプリケーションの挙動を観察すると、遅延ロードにより、85%のバイトコードの生成を抑えられる可能性が確認できた。バイトコード表現を工夫し、コンパイル済みコードとデータを共有するようにすることでロード時間を削減し、複数プロセスでのメモリ共有を促進することができるが、一般に共有によって間接参照が増えてしまい、バイトコード実行速度が遅くなるという問題を生じる可能性がある。そのため、このような工夫はしないでも十分かもしれない。今後、より多くの評価を行なって、この結果を検証していきたい。

参考文献

[1] ティム・リンドホルム，フランク・イェリン：Java 仮想マシン仕様第 2 版，ピアソン・エデュケーション (2001).
[2] 井出真広，倉光君郎：Ruby 向けトレース方式 Just-in-time コンパイラの設計と実装，情報処理学会論文誌プログラミング（PRO），Vol. 8, No. 1, pp. 1–10 (2015).
[3] 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV の実装と評価，情報処理学会論文誌 (PRO), Vol. 47, No. SIG 2(PRO28), pp. 57–73 (2006).
[4] 芝哲史，笹田耕一，　平木敬：CastOff: Ruby 用コンパイラのライブラリとしての実装，情報処理学会論文誌プログラミング（PRO），Vol. 5, No. 3, pp. 1–22 (2012).
[5] 池原潔：組込みシステムのための動的コンポーネント機構と VM の最適化.　RubyKaigi2011 http://rubykaigi.org/2011/ja/schedule/details/16S03.

[*10] ただし、前節でも述べたが、実験で利用したスクリプトは複雑な Ruby プログラムであるため、1.1ms という結果は、実際にはもっと短くなる可能性がある。