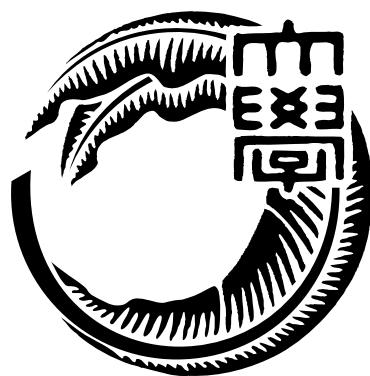


令和元年度 卒業論文

Rakuのサーバーを使った実行



琉球大学工学部情報工学科

165727F 福田光希

指導教員 河野真治

目次

第 1 章	スクリプト言語の高速実行	1
1.1	Perl6 の起動時間の改善	1
1.2	論文の構成	1
第 2 章	Raku の概要	2
2.1	Raku	2
2.2	Rakudo	3
2.3	Rakudo の語源	3
2.4	MoarVM	3
2.5	NQP	4
2.6	なぜ Raku は遅いのか	4
2.7	Raku と他言語との起動時間の比較	5
2.7.1	実行環境	5
2.7.2	起動時間の比較	5
第 3 章	Raku による Abyss の実装	6
3.1	サーバーの構成	6
3.2	Abyss Server 側の実装	7
3.3	Abyss Client 側の実装	7
3.4	Raku の Unix domain socket 実装	8
3.5	Raku の EVAL	8
3.6	NativeCall	9
3.7	出力を Socket に書き込む際に dup() している理由	9
3.8	通常実行との速度比較	9
第 4 章	Abyss Server の性能評価	11
4.1	比較	11
4.2	Abyss Server の利点	12
4.3	Abyss Server の欠点	12
第 5 章	まとめ	13

目 次

2.1 Rakudo の構成	3
3.1 Abyss サーバーを用いたスクリプト言語実行手順	6

表 目 次

2.1	起動時間の比較	5
3.1	helloworld の速度比較	9
3.2	フィボナッチ数列の例題の速度比較	10

ソースコード目次

3.1	Abyss Server 側の source code	7
3.2	Abyss Client 側の source code	7
3.3	eval のサンプルコード	8

第1章 スクリプト言語の高速実行

1.1 Perl6 の起動時間の改善

現在多くのスクリプト言語はインタプリタ型言語であり, 実行時にインタプリタの立ち上げ, モジュールを読み込み, スクリプトの解釈, スクリプトの実行 といったような処理を担っている. これらの処理の中には OS 上で事前に行うことで, より起動時間, 及び処理時間の短縮が予想される.

現在開発の進んでいる言語に スクリプト言語 Raku がある. Raku は任意の VM が選択できるようになっており, 主に利用されている VM に C で書かれた MoarVM が存在する. MoarVM は JIT コンパイルなどをサポートしているが, 全体的な起動時間及び処理速度が Perl5 や Python , Ruby などの他のスクリプト言語と比較し非常に低速である.

本研究では, スクリプト言語 Raku の起動時間及び, 処理速度の改善を図り, 研究をするにあたり得られた, OS 上でスクリプト言語を実行する方法の改善点について述べ, 今後の展望について記載する.

また, その手法として同一ホスト内で終了せずに実行を続けるサーバープロセスを立ち上げ, このサーバープロセス上で立ち上げておいたコンパイラに実行するファイル名を転送し, サーバー上でコンパイルを行う手法を提案する. 著者らは, この提案手法に沿って『Abyss サーバー』を実装している.

1.2 論文の構成

本論文は全 5 章で構成される. 2 章では Raku の概要について紹介する. 3 章では 提案手法で述べた「Abyss Server」の具体的な実装について解説する. 4 章では Abyss Server の性能評価について解説する. 5 章はまとめとなっている.

第2章 Raku の概要

2.1 Raku

Raku は 2002 年に LarryWall が Perl を置き換える言語として設計を開始した。Perl5 の言語的な問題点であるオブジェクト指向機能の強力なサポートなどを取り入れた言語として設計された。Perl5 は設計と実装が同一であり、Larry らによって書かれた C 実装のみだった。言語的な特徴としては、独自に Raku の文法の拡張が可能な Grammar, Perl5 と比較した場合のオブジェクト指向言語としての進化も見られる。また Perl6 は漸進的型付け言語である。従来の Perl の様に変数に代入する対象の型や、文脈に応じて型を変更する動的型言語としての側面を持ちつつ、独自に定義した型を始めとする様々な型に、静的に変数の型を設定する事が可能である。Raku は元々は Perl6 という名称であったが、言語仕様及び処理実装が Perl5 と大幅に異なっており、言語的な互換性が存在しない。従って現在では Raku と Perl5 は別言語としての開発方針になっている。Raku は現在有力な処理系である Rakudo から名前を取り Raku という名前がつけられている。

Raku の起動は、MoarVM を起動、NQP をロード、Rakudo をロードもしくはコンパイルし、その後 JIT しながら実行する。

2.2 Rakudo

Rakudo とは Raku の現在の主流な実装である。(Raku は言語名, Rakudo はコンパイラ)

Raku は仕様と実装が明確に区分されており, Rakudo という実装, roast という Raku の仕様 (テストスイートがある).

Rakudo は MoarVM, NQP と呼ばれる Raku のサブセット, NQP と Raku 自身で記述された Raku という構成である.

Rakudo は MoarVM の他に JVM や Java Script を動作環境として選択可能である.

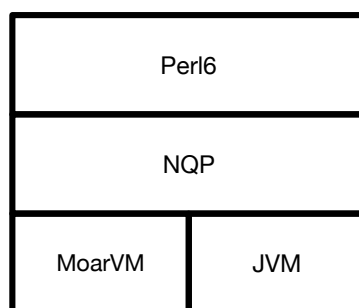


図 2.1: Rakudo の構成

2.3 Rakudo の語源

ちなみに Rakudo の語源は, 楽土と駱駝道で
楽土の方は日本語で「楽園」という意味で
駱駝道の方は Perl のマスコットキャラクターがラクダだったからである

2.4 MoarVM

MoarVM は Rakudo, NQP のために構築された VM である. C 言語で実装されている. JIT コンパイルなどが現在導入されているが, 起動時間が低速であるなどの問題がある. MoarVM 独自の ByteCode があり, NQP からこれを出力する機能などが存在している.

MoarVM は NQP と Byte Code を解釈する.

2.5 NQP

NQP とは Not Quite Perl の略で Raku のサブセットであるその為基本的な文法などは Raku に準拠しているが、変数を束縛で宣言するなどの違いが見られる。

この NQP で記述された Raku の事を Rakudo と呼ぶ。

Rakudo における NQP は現在 MoarVM, JVM 上で動作する。NQP は最終的には NQP 自身でブートストラップする言語であるが、ビルドの最初にはすでに書かれた MoarVM のバイトコードを必要とする。Raku の一部は NQP を拡張したもので書かれている為、Rakudo を動作させる為には MoarVM などの VM, VM に対応させる様にビルドした NQP がそれぞれ必要となる。現在の NQP では MoarVM, JVM に対応する Stage0 はそれぞれ MoarVM のバイトコード, jar ファイルが用意されている。

Perl6 のテストスイートである Roast やドキュメントなどによって設計が定まっている Perl6 とは異なり NQP 自身の設計は今後も変更になる可能性が開発者から公表されている。現在の公表されている NQP のオペコードは NQP のリポジトリに記述されているものである。

2.6 なぜ Raku は遅いのか

通常 Ruby のようなスクリプト言語ではまず YARV などのプロセス VM が起動し、その後スクリプトを Byte code に変換して実行という手順を踏む。Rakudo はインタプリタの起動時間及び、全体的な処理時間が他のスクリプト言語と比較して非常に低速である。これは Rakudo 自体が Raku と NQP で書かれているため、MoarVM を起動し、Rakudo と NQP の Byte code を読み取り、Rakudo を起動し、その後スクリプトを読み取り、スクリプトの Byte code 変換というような手順で進むためである。また Perl6 は実行時の情報が必要であり、メソッドを実行する際に invoke が走ることも遅い原因である。invoke は MoarVM の method 呼び出しの Byte code である。

2.7 Raku と他言語との起動時間の比較

Raku と他言語の起動時間の比較を行った。題材として perl5, ruby, raku, python で helloworld を出力するプログラムを用いて行った実行結果である。

2.7.1 実行環境

- macOS Mojave version 10.14.5
- メモリ 8GB
- プロセッサ 2.7GHz Intel Core i5

2.7.2 起動時間の比較

Language	version	Time
Raku	2019.03.1	249 ms
Perl5	v5.18.4	4 ms
Python	2.7.10	13 ms
Ruby	2.3.7p456	83 ms

表 2.1: 起動時間の比較

Raku は最も早い perl5 の約 62.25 倍, python の約 19 倍, ruby の約 3 倍起動速度が遅いことがわかる。

第3章 Raku による Abyss の実装

提案手法で述べた、同一ホスト内で終了せずに実行を続けるサーバープロセスを立ち上げ、このサーバープロセス上で立ち上げておいたコンパイラに実行するファイル名を転送し、サーバー上でコンパイルを行う手法に沿い『Abyss Server』を実装した。

3.1 サーバーの構成

Abyss サーバーはクライアント側から投げられた Raku スクリプト を実行するためのサーバーである。図 3.1 は、Abyss サーバーを用いたスクリプト言語実行手順である。Abyss サーバーはユーザーが Raku を直接立ち上げるのではなく、まず図 3.1 右側の Abyss サーバーを起動し、ユーザーは Abyss サーバーにファイルパスをソケット通信で送り、Abyss サーバーがファイルを開き実行し、その実行結果をユーザーに返す。

この手法を用いることで、サーバー上で事前に起動した Rakudo を再利用し、投げられた Raku スクリプトの実行を行うため、Rakudo の起動時間を短縮できると推測できる。

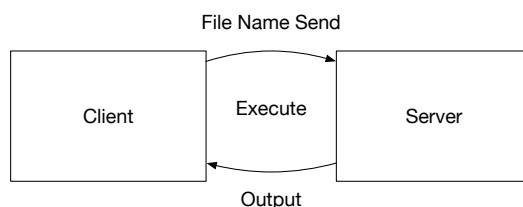


図 3.1: Abyss サーバーを用いたスクリプト言語実行手順

3.2 Abyss Server 側の実装

[3.1] は Abyss サーバーのソースコードである。Abyss サーバーは起動すると、まず自身にファイルパスを転送するためのソケットを生成し、その後ファイルパスを受け取り実行して、出力結果をソケットに書き込む

Listing 3.1: Abyss Server 側の source code

```
1 use v6.c;
2 unit class Abyss::Server:ver<0.0.1>:auth<cpan:ANATOFUZ>;
3 use MONKEY-SEE-NO-EVAL;
4 use IO::Socket::Unix;
5 use NativeCall;
6
7 sub close(int32) returns int32 is native { ... }
8 sub dup(int32 $old) returns int32 is native { ... }
9 sub dup2(int32 $new, int32 $old) returns int32 is native { ... }
10
11 method readeval
12 {
13     my $listen = IO::Socket::Unix.new( :listen,
14                                       :localhost<localhost>,
15                                       :localport(3333) );
16     my $backup = dup(1);
17     say DateTime.now;
18
19     loop
20     {
21         my $conn = $listen.accept;
22         my $sock_msg;
23         my $buf = $conn.recv();
24         $sock_msg = $buf;
25         close(1);
26         dup2($conn.native-descriptor(), 1);
27         EVALFILE $sock_msg;
28         dup2($backup, 1);
29         close($backup);
30         $conn.close;
31     }
32
33     $listen.close;
34 }
```

3.3 Abyss Client 側の実装

ユーザーは Abyss Server を起動後、Client 側からファイルパスをサーバーに送信し、[3.2] のように Socket に書き込まれた実行結果を読み取る。Client 側は Raku で実装されているが、Client 側は Socket を生成し、filepath を送信するだけなので他の言語でも書くことが可能だ。

Listing 3.2: Abyss Client 側の source code

```
1 use IO::Socket::Unix;
2
3 my $conn = IO::Socket::INET.new( :host<localhost>,
4                                  :port(3333) );
5
```

```

6 $conn.print: 'Absolute_file_path';
7
8 my $sock_msg;
9
10 while my $buf = $conn.recv(:bin)
11 {
12     $sock_msg = $buf.decode;
13     last;
14 }
15
16 say $sock_msg;

```

3.4 Raku の Unix domain socket 実装

今回、通信する Socket には Unix domain socket を用いた。

TCP socket や Async socket を用いた場合、他者からスクリプトを送りつけられる可能性がある。そのため、今回は Unix domain socket を用いて実装を行なった。Raku には現在 Unix domain socket の実装がないため、Unix domain socket の実装を行なった。

IO::Socket が role として定義されている

Raku での role は他の言語の interface に相当するものである

現状 Raku には IO::Socket::INET と IO::Socket::Async の実装がある

前述したように、INET と Async はセキュリティの問題で使えない

IO::Socket を実装した IO::Socket::Unix を実装した

IO::Socket::Unix の中では nqp の機能を使う必要がある

3.5 Raku の EVAL

Raku では EVAL 関数 [3.3] があり文字列を Raku のソースコード自身として評価できる

Raku では、EVAL は通常は使用できないようになっており、MONKEY-SEE-NO-EVAL という pragma を実行することで使うことができるようになる。EVALFILE はファイルパスを受け取るとファイル開き、バイト文字列に変換し読み込む、その後読み込んだバイト文字列にデコードし、ファイルパスの文字列を読み込み、ファイルの中身を EVAL と同様に解釈する。

[3.1] の 2 行目にある MONKEY - SEE - NO - EVAL は Perl6 上で EVALFILE を使用可能にする pragma である。

Listing 3.3: eval のサンプルコード

```

1 use MONKEY-SEE-NO-EVAL;
2
3 EVAL "say_{5+5}"; # OUTPUT: 10

```

3.6 NativeCall

Raku では Native Call という標準ライブラリを用いて、C のライブラリを扱うことが可能である。

指定がない場合は C の標準ライブラリが呼ばれる。

[3.1] で利用した C のライブラリは以下のものである

- `dup(int fd) - dup()` は引数で指定したファイルディスクリプタを複製して、未使用のファイルディスクリプタから最小番号を割り当てる。また `dup` はシステムコールの一種である。
- `dup2(int newfd, int oldfd) - dup2()` は `oldfd` を `newfd` に関連づけます。
- `close(int fd) - close()` は指定された file descriptor を閉じます。

3.7 出力を Socket に書き込む際に dup() している理由

Raku では標準出力を Socket に書き込む API が Raku 側からは提供されていない。

そのため、通常 Code 3.1 のように EVALFILE を実行した際、出力はそのまま Server 側の標準出力に返ってしまう。

この問題を解決するために、Raku の NativeCall を用いて C のライブラリを使用した。

Code 3.1 の 26 行目で `dup2()` を用いて 標準出力に Socket の file descriptor を割り当て、その後 EVALFILE を実行することで出力を Client 側に返すことに成功した。

3.8 通常実行との速度比較

今回は、提案手法での実行速度と通常実行での実行速度、この二つの速度の比較を行う題材として行うのは helloworld を出力するだけのプログラムとフィボナッチ数列の例題である。

実験結果

Language	Time
通常実行	0.2695 sec
提案手法	0.0238 sec

表 3.1: helloworld の速度比較

提案手法は通常実行に比べて約 10 倍早い実行結果になった

実験結果

Language	Time
通常実行	0.2128 sec
提案手法	0.0415 sec

表 3.2: フィボナッチ数列の例題の速度比較

先ほどと同様, 提案手法は通常実行に比べて早い結果となり, 約 5 倍早い実行結果になった
内部で処理が走っている時間だけさが縮まったと考えられる.

第4章 Abyss Server の性能評価

4.1 比較

- Microsoft CLR

.NET Framework には、共通言語ランタイム (Common Language Runtime) と呼ばれるランタイム環境がある。 .NET 対応のソフトウェアは、様々なプログラミング言語で書かれたソースコードから、いったん共通中間言語 (Common Intermediate Language) による形式に変換されて利用者のもとに配布される。 CIL 形式のプログラムを解釈し、コンピュータが直に実行可能な機械語によるプログラムに変換して実行するソフトウェアが CLR である。 現状の Abyss サーバーはプロセスとして立ち上げているが、CLR は OS に直接組み込む必要があるが、Abyss サーバーはプロセス上で実行しているため OS に手を加えず実装が容易である。

- FastCGI

FastCGI とは CGI を改良したものである。 通常の CGI は、ユーザーから要求があるたびにプロセスを生成しプログラムを実行し、プロセスを破棄する。 FastCGI では一度プログラムが起動し、ロードされるとメモリ上にしばらく残り、次回実行される際にそのプログラムを再利用する。

FastCGI は Abyss Server とよく似ているが、FastCGI と Abyss Server の違いは、FastCGI は HTTP リクエストが起動のトリガーになっており、Abyss Server はユーザーが実行したいプログラムを送信することが起動のトリガーになっていることである。

- PyPy

PyPy は Python の実装の一つであり、Cpython のサブセットである RPython で記述された処理系である。

PyPy は JIT コンパイルを採用しており、実行時にコードを機械語にコンパイルして効率的に実行させることができる。 PyPy は Cpython より実行速度が速いが起動速度は Cpython と比較して約3倍遅い。 Raku と同様、PyPy は Cpython と比較して起動時間が遅いため今回提案した手法を応用できると予測できる。

4.2 Abyss Server の利点

Abyss Server の利点には以下のようなものが挙げられる。

- Abyss Server を用いて実行することで、サーバー上で事前に起動した Rakudo を再利用し、投げられた Raku スクリプトの実行を行うため、Rakudo の起動時間を短縮できる。
- 一度投げられたスクリプトのバイトコード、もしくは計算結果をキャッシュで保存しておき、再度実行する際に、そのキャッシュを用いてコンパイル時間を省くような仕組みを入れやすいと考えられる。
- 他の起動時間遅いスクリプト言語や、モジュールの読み込みが遅い言語などにも、応用しやすいと考えられる。
- 普通のスクリプト言語だと実行するたびに fork して実行シインタプリタの立ち上げという処理になるが、プロセス毎回起動しなくて済む

4.3 Abyss Server の欠点

Abyss Server の欠点は以下のようなものが挙げられる。

- 現在 Abyss Server には 一度スクリプトを実行した後にサーバー内の環境をリセットする機能が存在しないため、スクリプトがサーバー内の環境に影響を及ぼした場合、通常実行と違う挙動をする危険性がある
- 同時に二つ以上のタスクを与えられると実行順のスケジューリングができない
- 異常に長いタスクが投げられた場合、次のタスクが前のタスクが終わるまで実行ができない
- 起動時のオプションが選択出来ない
- Client 側を Raku で書いてしまうとファイルパスを送信する際に Raku を起動することになるので、起動時間が通常実行よりかかってしまう

第5章 まとめ

本稿では実行する Raku スクリプトのファイル名をサーバーに転送し、コンパイラサーバーでコンパイルを行い実行する手法の提案を行なった。また、提案手法に沿って「Abyss Server」を実装した。

Raku の速度改善において、同一ホスト内でサーバープロセスを生成し、サーバープロセス内であらかじめコンパイラを立ち上げて置き、実行するファイル名を転送し、サーバープロセス上でコンパイルを行う手法は有効であると考えられる

OS 上でスクリプト言語を実行する際の最適な方法として、提案手法のようにコンパイラを再利用する方法は有効である。

また OS 上でスクリプト言語を実行する際に、OS 側で用意されてあるべき API としては以下のようなものが挙げられる

- 提案手法のように一度立ち上げられたインタプリタを立ち上げたままにする機能
- 複数回投げられたスクリプトの実行結果もしくは basic block を保存できる機能
- モジュールを Runtime でロードする機能
- 実行するスクリプトの周りにある Json ファイルをあらかじめ Parse しておく

また今後の Abyss サーバーの開発において期待される改善点には以下のようなものがある

- コンパイラの起動が遅い言語だけでなく、モジュールの読み込みが遅い言語などを、あらかじめサーバーを側でモジュールを読み込んでおき、それを利用してプログラムを実行する手法も応用できるように改良を行う。
- スクリプトを送る際にモジュールも送信する機能の追加
- プログラムの実行終了したらモジュールを削除する機能の追加
- 数回スクリプトが投げられたら、サーバーを落とすなどのリソース管理の実装

今後の開発を行っていくにあたって、他の script 言語にも応用できるように開発を行っていく。

参考文献

- [1] Andrew Shitov. Perl6 Deep Dive
- [2] 清水隆博, 河野真治. CbC を用いた Perl6 処理系. プログラミングシンポジウム論文, 2019.
- [3] Perl6 Documentation
(<https://docs.perl6.org>) (2019/10/22 access)
- [4] rakudo
(<https://github.com/rakudo/rakudo>) (2020/2/14 access)
- [5] The Official Raku Test Suite
(<https://github.com/perl6/roast/>)
- [6] NQP - Not Quite Perl
(<https://github.com/perl6/nqp>)
- [7] MoarVM
(<https://github.com/MoarVM/MoarVM>) (2020/2/14 access)
- [8] ThePerlFoundation: Perl 6 Design Documents, ThePerlFoundation (online), available from (<https://design.raku.org>)
- [9] C Documentation
(<https://devdocs.io/c/>) (2020/2/14 access)
- [10] The Hot New Language Named Rakudo
http://blogs.perl.org/users/zoffix_znet/2017/07/the-hot-new-language-named-rakudo.html

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜りました河野真治准教授に深く感謝致します。本研究の遂行及び本論文の作成にあたり、数々の貴重な御助言と細かな御配慮を戴いた、東恩納琢偉さん、外間政尊さん、桃原優さん、清水隆博さん、並びに並列信頼研究室の皆様に深く感謝致します。

また、毎週のゼミでお世話になっているハリーズの皆様に感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2020年2月

福田光希

付録

Raku(Perl6) の build 方法

以下に MacOS 上での Raku の build 方法を記述する

GitHub から rakudo をダウンロードしてくる

```
git clone https://github.com/rakudo/rakudo.git
```

ダウンロードしてきた directory に cd して以下のコマンドを実行する

```
perl Configure.pl --gen-moar --gen-nqp --backends=moar --moar-option='--compiler=clang'
```