Rakuのサーバーを用いた実行

大蔵 海斗[1,a)]　河野 真治[1,b)]

概要：現在開発の進んでいる言語に Raku がある. Raku はコンパイラが Raku 自身で書かれてるため, 起動時に毎回コンパイラのロードとコンパイル, JIT コンパイルを繰り返すことになる. そのため, 起動時間及び処理速度が Perl5 や Python, Ruby など比較し非常に低速である. そこで, この問題を解決するために, 既にコンパイラをロードしてあるサーバーを同一ホスト内に用意し, このサーバー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している. Raku に対しては, 当研究室にて Abyss サーバーを開発している. 本稿では, Abyss サーバーの改善を図り, 研究をするにあたり得られた改善点について述べ, 今後の展望について記載する.

キーワード：プログラミング言語, Raku, Rakudo, roast, 高速化

1. はじめに

最近のスクリプト言語ではコンパイラが自身で書かれているケースが多い. 例えば, PyPy や Go 言語, Haskell などである.

現在開発の進んでいる言語に Raku がある. Raku はコンパイラが Raku 自身で書かれてるため, 起動時に毎回コンパイラのロードとコンパイル, JIT コンパイルを繰り返すことになる. そのため, 起動時間及び処理速度が Perl5 や Python, Ruby など比較し非常に低速である.

そこで, この問題を解決するために, 既にコンパイラをロードしてあるサーバーを同一ホスト内に用意し, このサーバー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している. Raku に対しては, 当研究室にて Abyss サーバーを開発している.

[1] 琉球大学大学院理工学研究科工学専攻知能情報プログラム
[a)] k218576@ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

本稿では, Abyss サーバーの改善を図り, 研究をするにあたり得られた改善点について述べ, 今後の展望について記載する.

2. 基礎概念

2.1 Raku の概要

Raku は元は Perl5 の後継言語の Perl6 として開発されており, 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. オブジェクト指向以外にも, Raku 自身で言語拡張ができる Grammar や漸進的型付けなどの特徴がみられる.

しかし, Raku の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. 従って現在では名称が Perl6 から Raku へと変更されている. Raku という名称は, 現在有力な処理系である Rakudo が由来となっている.

2.2 roast

roast とは the repository of all spec tests の略

であり, Raku の仕様を決めるテストスイートである.

Raku は仕様と実装が明確に区分されており, roast をどれだけパスするかでその実装が Raku の仕様に準拠しているかを判断する. Raku の現在の主流な実装は Rakudo であるが, roast には Rakudo がパスしないテストも多く含まれている.

2.3 fudge

Raku の処理系は Rakudo の他にも pugs があり, 今後もそれら以外の処理系が登場することが予想される. それぞれの処理系がパスする roast のテストは異なっており, 同じ処理系でもバージョンが変わればパスするテストは異なる.

roast には fudge という機能があり, 指定した処理系に合わせた新たなテストを生成する. 新しく生成されたテストでは, 現時点ではパスしないことが明らかなテストのスキップや例外処理ができる.

2.4 Rakudo の構成

Rakudo とは Raku の現在の主流な実装である. Rakudo の構成は, Rakudo のために構築された VM である MoarVM, Raku のサブセット言語である NQP, そして Raku である. Rakudo は MoarVM の他に JVM や JavaScript を動作環境として選択可能である.

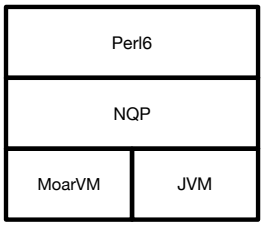

図 1: Rakudo の構成

2.5 MoarVM

MoarVM は Rakudo と NQP のために構築された NQP とバイトコードを解釈する VM であり, 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 独自のバイトコードがあり, NQP からこれを出力する機能などが存在している.

2.6 NQP

NQP とは Not Quite Perl の略であり, Raku のサブセット言語である. この NQP で記述された Raku のことを Rakudo と呼ぶ. 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, ビルドの最初には既に書かれた MoarVM のバイトコードを必要とする.

2.7 Rakudo が遅い原因

通常, Ruby のようなスクリプト言語ではまず VM が起動し, その後スクリプトをバイトコードに変換して実行という手順を踏む. 対して Raku は, コンパイラの Rakudo 自体が Raku と NQP で書かれているため, MoarVM を起動し, Rakudo と NQP のバイトコードを読み取り, Rakudo を起動し, その後スクリプトをバイトコードに変換して実行という手順を踏む.

そのため, 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. また, Raku 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 メソッド呼び出しのバイトコードである.

3. Abyss の実装

本研究で使うファイルのディレクトリ構造を Code [1] に示す.

```
Abyss-Server
    - lib
        - Abyss
            - EvalAsMod.pm6
            - Server.pm6
    - other
        - client.p6
        - startup.p6
```

```
        - compare
            - abyss.sh
            - local.sh
        - namespace
            - eval.p6
            - evalasmod.p6
            - intadd.p6
            - stradd.p6
    - roast
        - S01-perl-5-integration
        - S02-lexical-conventions
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        - S32-trig
        - fudge
        - packages
            - Test-Helpers/
        - test_list
```

Code 1: tree

3.1 サーバーの構成

Abyss サーバーはクライアントから投げられた Raku スクリプトを実行するためのサーバーである.

まず, Abyss サーバーを起動し, ユーザーはクライアントのスクリプトを通して, Abyss サーバーに実行したいファイルパスをソケット通信で送信する. Abyss サーバーは受信したファイルパスを元にファイルを実行し, 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, Abyss サーバー上で事前に起動した Rakudo を再利用し, Rakudo の起動時間を省略できると考えられる.

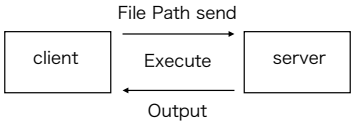

図 2: Abyss サーバーを用いた Raku の実行

3.2 Server の実装

Abyss サーバーは起動するとソケットを生成し, クライアントからの接続を待つ. 接続すると Abyss サーバー側の標準出力と標準エラー出力にソケット記述子を割り当て, 出力結果をソケットに書き込めるようにする. そして, クライアントから受け取ったファイルパスを元にスクリプトをモジュールとして実行し, 出力結果をソケットに書き込む.

```
use v6;
unit class Abyss::Server:ver<0.0.1>:
    auth<cpan:ANATOFUZ>;
use Abyss::EvalAsMod;
use NativeCall;

sub dup(int32 $old) returns int32 is
    native { ... }
sub dup2(int32 $new, int32 $old)
    returns int32 is native { ... }

method readeval {
    my $listen = IO::Socket::INET.new(:
        listen, :localhost<localhost>,
        :localport(3333));

    say DateTime.now;
    my $stdout = dup(1);
    my $stderr = dup(2);

    loop {
        my $conn = $listen.accept;
        my $path = $conn.recv();
        say "evaling "~$path;
        dup2($conn.native-descriptor(),
             1);
        dup2($conn.native-descriptor(),
             2);

        try EVALASMOD $path;
        say $! if $!;

        dup2($stdout, 1);
        dup2($stderr, 2);
        $conn.close;
    }

    $listen.close;
}
```

Code 2: Server.pm6

3.3 client の実装

client は既に起動してある Abyss サーバーのソケットに接続し, 実行したいファイルパスを送信する. その後, サーバーからソケットに書き込まれたファイルの実行結果を読み取り, 出力する.

```
sub send-paths (Str $path) {
    my $conn = IO::Socket::INET.new(:
        host<localhost>,:port(3333));

    $conn.print: $path;
    while my $buf = $conn.recv(:bin) {
        print $buf.decode;
    }

    $conn.close;
}

sub MAIN(**@ARGS) {
    send-paths $_.IO.resolve.absolute
        for @ARGS;
}
```

Code 3: client.pm6

3.4 EVALASMOD

Raku では EVAL 関数と EVALFILE 関数がある. EVAL 関数は文字列をソースコードとして評価できる. EVALFILE 関数は ファイルの中身を文字列として開き, EVAL 関数にソースコードとして評価させる. EVAL 関数と EVALFILE 関数はインジェクション攻撃をされる恐れがあるため, 通常は使用できないようになっているが MONKEY-SEE-NO-EVAL というプラグマを実行することで使うことができるようになる.

従来の Abyss サーバーではこの EVALFILE 関数を用いてファイルの実行を行なっていたが, 複数のファイルを実行させると同名の関数などが衝突し, 適切に実行できなかった. そこで, 実行する各ファイルをモジュールとして実行することで名前空間を分離する手法を取り, EVALASMOD を実装した.

```
unit module EvalAsMod;
use MONKEY-SEE-NO-EVAL;

sub EVALASMOD($filename, :$lang = 'Raku
    ', :$check) is export {
    my $code = 'my module Mod {'~"\n"~
        slurp($filename)~'}';
    EVAL $code, :$lang, :$check, :
        context(CALLER::), :$filename;
}
```

Code 4: EvalAsMod.pm6

3.5 NativeCall

Raku では NativeCall という標準ライブラリを用いて, C 言語のライブラリを扱うことができる.

Code [2] では, C 言語の標準ライブラリである dup と dup2 を呼び出しており, 標準出力と標準エラー出力にソケット記述子を割り当てるのに使っている.

4. 従来の Abyss サーバーとの変更点

本研究で改善した Abyss サーバーは, 従来の Abyss サーバーと比較して以下のような変更点がある.

4.1 従来の Abyss サーバー

(1) Abyss サーバーとクライアントとのソケット通信が 1 度しか行われず, クライアント側にファイルの実行結果の全てを出力できない.
(2) クライアント側に表示される出力結果に不要な改行が混じることが多々ある.
(3) 標準エラー出力ができない.
(4) クライアント側のソースコードに実行したいファイルの絶対パスを埋め込む必要があり, 実行したいファイルを変更するたびにクライアント側のソースコードを書き換える必要がある.
(5) 1 度のクライアントの実行につき, Abyss サーバーへ 1 つのファイルパスしか送信できない.
(6) Abyss サーバーがエラーを含むファイルを実行した場合, Abyss サーバーが強制終了してしまう.

4.2 改善後の Abyss サーバー

(1) クライアント側がファイルの実行結果を全て受信するまでソケット通信を行い, 全ての実行結果を出力することができる.
(2) クライアント側に表示される出力結果の不要な改行を取り除いた.
(3) 標準出力の他に標準エラー出力が出力される.
(4) 実行したいファイルの絶対パスもしくは相対パスをコマンドライン引数から指定できる.
(5) Abyss サーバーへ一度に任意の数のファイルパスを送信できる.
(6) Abyss サーバーがエラーを含むファイルを実行しても Abyss サーバーは強制終了せず, 引き続き他のファイルを実行できる.

4.3 名前空間の分離

従来の Abyss サーバーでは複数のファイルを実行させると同名の関数などが衝突し, 適切に実行できなかった.

そこで, EVALASMOD を実装し, この問題を解決した. EVALASMOD によって実行されるファイルはレキシカルスコープのモジュールとして実行されるため, ファイル内で宣言した変数や関数はファイルの実行終了後に破棄される. そのため, 現在の Abyss サーバーでは, 同名の関数を持つ複数のファイルの実行や, 同じファイルを複数回実行することが可能である.

4.4 名前空間の分離の検証

実際に EVALASMOD が名前空間の分離に成功しているかの検証を行う.

実行する題材として行うのは, 文字列を連結して出力するだけのプログラム stradd.p6 (Code [5]) と単純な計算を行い結果を出力するだけのプログラム intadd.p6 (Code [6]) である. どちらのプログラムも add という同名の関数を定義している.

```
our sub add ($a, $b) {
    say $a ~ $b;
}

add 'hello', 'world'; # helloworld
```

Code 5: stradd.p6

```
our sub add ($a, $b) {
    say $a + $b;
}

add 7, 11; # 18
```

Code 6: intadd.p6

4.4.1 EVALFILE による実行

Code [7] は EVALFILE を用いて stradd.p6 と intsdd.p6 を実行するサンプルプログラムである. これを実行すると add 関数の再定義によりエラーが出力される.

従来の Abyss サーバーはこの EVALFILE を用いてファイルを実行していたため, 同名の関数などが衝突していた.

```
use MONKEY-SEE-NO-EVAL;

EVALFILE $?FILE.IO.parent.add('stradd.
    p6');
EVALFILE $?FILE.IO.parent.add('intadd.
    p6');
```

Code 7: EVALFILE のサンプルプログラム

4.4.2 EVALASMOD による実行

Code [8] は EVALASMOD を用いて stradd.p6 と intsdd.p6 を実行するサンプルプログラムである. これを実行すると エラーを起こさずに実行結果が出力されることが確認できる.

この EVALASMOD を Abyss サーバーに実装することで, 関数名の衝突を避けることに成功した.

```
use lib $?FILE.IO.parent(3).add: 'lib';
use Abyss::EvalAsMod;

EVALASMOD $?FILE.IO.parent.add('stradd.
    p6');
EVALASMOD $?FILE.IO.parent.add('intadd.
    p6');
```

Code 8: EVALASMOD のサンプルプログラム

5. 動作比較検証

通常の手法と Abyss サーバーを用いた手法 (以下, 提案手法) で実行速度の比較を行う. Raku は開発中の言語であるため, 主なアプリケーションはコンパイラと roast しかない. そのため, 本稿の動作比較検証では roast を用いる.

5.1 test_list の作成

現時点では Abyss サーバーにてコンパイルができずに動作しない roast のテストファイルがいくつか存在する. 通常の手法との適切な動作比較検証を行うために, 通常の手法と提案手法の両方において正しく動作するテストファイルのファイルパスのみを並べた test_list というファイルを作成した.

本稿の動作比較検証では, test_list に記載されているファイルパス群を実行し, その動作結果を比較, 考察する.

5.2 通常の手法

通常の手法は Code [9] に示すシェルスクリプトを用いることにする.

test_list に記載されているファイルパス群を cat コマンドで取得し, for 文を用いてファイルを 1 つずつ Raku で実行する. Test-Helpers は roast のテストを実行するのに必要なモジュールである.

1 つのファイルを実行する度に Raku の起動と Test-Helpers の読み込みを行うため, 比較的実行に時間がかかると思われる.

```
#!/bin/zsh

filepath=‘pwd‘/$0
root=${filepath%/*/*/*}
lib=$root’/roast/packages/Test-Helpers
    /’

time (
for f (‘cat $root/roast/test_list‘)
do
    raku -I $lib $f
done
)
```

Code 9: 通常の手法での実行

5.3 提案手法 1

まず最初に, startup.p6 (Code [10]) を実行し, Abyss サーバーを起動する. その際に Test-Helpers を Abyss サーバーにロードする.

```
use v6;
use lib $?FILE.IO.parent(2).add: ’lib’;
use lib $?FILE.IO.parent(2).add(’roast/
    packages/Test-Helpers’);
use Abyss::Server;

my $abyss = Abyss::Server.new;
$abyss.readeval();
```

Code 10: 提案手法での実行

次に, 別のターミナルを起動し, Code [11] に示すシェルスクリプトを用いることにする. 通常の手法と同じように, cat コマンドと for 文を用いてファイルを 1 つずつ実行する. ファイルパスはクライアントを通して Abyss サーバーに送信され, 実行結果がクライアント側に出力される.

通常手法と同じように, 1 つのファイルを実行する度に Raku の起動を行うが, 予め Abyss サーバーに Test-Helpers の読み込みを済ませているため, 通常手法よりも短い時間で実行できると思われる.

```
#!/bin/zsh

filepath=‘pwd‘/$0
root=${filepath%/*/*/*}

time (
for f (‘cat $root/roast/test_list‘)
do
    raku $root/other/client.p6 $f
done
)
```

Code 11: 提案手法での実行

5.4 提案手法 2

提案手法 1 と同じように, 最初に, startup.p6 [10] を実行し, Abyss サーバーを起動する. その際に Test-Helpers を Abyss サーバーにロードする.

次に, 別のターミナルを起動し, Code [12] に示すシェルスクリプトを用いることにする. test_list に記載されているファイルパス群を cat コマンドで取得し, パイプ処理と xargs コマンドを用いることで client.p6 の引数にする. 引数にされたファイルパス群はクライアントを通して Abyss サーバーに送信され, 実行結果がクライアント側に出力される.

Raku の起動は一度で済み, 予め Abyss サーバーに Test-Helpers の読み込みを済ませているため, 通常の手法や提案手法 1 よりも実行時間を短縮できると思われる.

```
#!/bin/zsh

filepath=‘pwd‘/$0
root=${filepath%/*/*/*}

time (
    cat $root’/roast/test_list’ | xargs
         raku $root’/other/client.p6’
)
```

Code 12: 提案手法での実行

5.5 実験結果

表 1: test_list の速度比較

手法	実行時間
通常の手法	13.74 s
提案手法 1	8.739 s
提案手法 2	4.452 s

提案手法はどちらも通常の手法と比較して実行時間が短縮した. 予めモジュールを Abyss サーバーに読み込ませたことと, Rakudo を起動した回数の違いによって実行時間に差が生まれたと考えられる.

6. Abyss サーバーの欠点とその改善策

6.1 Abyss サーバーの欠点

(1) ファイルパスを送信する client を Raku で実装しているので, Abyss サーバーに ファイルパスを送信するたびに Raku を起動することになってしまう. そのため, 単体のファイルの実行の場合はソケットによる通信を挟む分処理速度が通常より落ちてしまう.

(2) Abyss サーバーで実行するファイル内に exit が含まれていると, 実行ファイルだけでなく Abyss サーバー自身も終了してしまう.

(3) Raku では, モジュール内で一部のプラグマが使用できない. Abyss サーバーはファイルをモジュールとして実行するので, 実行するファイル内にそのプラグマが含まれていると, コンパイルエラーを起こしてしまう.

(4) 例えばファイル操作などといった相対パスを扱う内容のファイルを Abyss サーバーで実行すると, 相対パスが Abyss サーバー側に依存してしまうため, 期待した通りの動作をしない.

(5) 標準入力を用いたファイルの実行に対応していない.

(6) 通信に INET ドメインソケットを利用しているため, EVALASMOD を悪用したインジェクション攻撃が予想される.

6.2 改善策

(1) client はソケットにファイルパスを送信し, 出力結果を受け取るだけなので他の言語でも書くことができる. c 言語などのコンパイラ言語で client を実装することで速度の向上が見込める.

(2) Abyss サーバー側で exit に対してモンキーパッチを施すことで, exit による Abyss サーバーの終了の回避が可能だと考える.

(3) それぞれのプラグマに対して適切な例外処理を施し, モジュールの外部でプラグマの使用をすることでコンパイルエラーを回避することが可能だと考える.

(4) クライアント側のカレントディレクトリのパスを Abyss サーバーに送信し, Abyss サーバー側で何らかの処理をすることで相対パスをクライアント側に依存させることが可能だと考える.
(5) 今回の実装で dup を用いて標準出力や標準エラー出力のファイル記述子を操作したように, 標準入力のファイル記述子を操作することで標準入力への対応が可能だと考えられる.
(6) 通信に UNIX ドメインソケットを利用することでインジェクション攻撃を防ぐ.

7. まとめ

本研究では従来の Abyss サーバーが抱える様々な問題点を修正した. 中でも EVALASMOD の実装によって実行ファイルごとの名前空間の分離や, 複数のファイルの実行という成果が得られ, それに伴って roast のテストファイル群の実行に成功した.

roast の実行を通した検証で得られたように, Abyss サーバーを用いることによる Rakudo の起動時間の省略とモジュールのロードの省略は有効であると考えられる.

EVALASMOD の実装や roast の実行を通して新たな課題がいくつか見えてきた. 今後は Abyss サーバーが一部のプラグマを実行できない問題の解決や UNIX ドメインソケットを適用する開発を行っていく.

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] 福田光希, 河野真治. Raku のサーバーを使った実行. 2020.
[4] Raku Documentation (https://docs.raku.org) (2021/11/09 access)
[5] rakudo (https://github.com/rakudo/rakudo) (2021/11/08 access)
[6] roast (https://github.com/Raku/roast) (2021/11/08 access)
[7] NQP (https://github.com/perl6/nqp) (2021/11/08 access)
[8] MoarVM (https://github.com/MoarVM/MoarVM) (2021/11/08 access)
[9] Raku 入門 (https://raku.guide/ja/) (2020/2/11 access)