Perl6のサーバーを使った実行

福田 光希[1,a)]　河野 真治[1,b)]

概要：Perl6 の実装の一つである Rakudo は, Byte code である MoarVM と, その上で動作する Perl6 の subset である nqp (Not Quite Perl) 上に構成されている,
現状の Perl6 の実行は Perl6 で記述されたコンパイラを load して JIT しながら実行することと自体に時間がかかっている．そこで, Perl6 をサーバーとして動作させ, 実行するファイルをサーバーに投げて実行する方法を実装してみた．
本論文では, サーバーで script 言語を実行する場合の利点と欠点について考察する．

キーワード：プログラミング言語, Perl6, サーバー, Raku

1. Perl6 の起動時間の改善

現在開発の進んでいる言語に Perl6 がある．スクリプト言語 Perl6 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．その為, 現在日本国内では Perl6 は実務としてあまり使われていない．

Perl6 の持つ言語機能や型システムは非常に柔軟かつ強力であるため, 実用的な処理速度に達すれば, 言語の利用件数が向上することが期待される．Perl6 は MoarVM に基づく JIT コンパイラを持っており, コンパイルされた結果はプロセッサが実行可能な機械語に相当する．

しかし現状の Perl6 は起動時間が非常に遅いことが問題である．

この問題を解決するために, 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サーバーに転送し, 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．

またサーバーでは, サーバーに投げられた Perl6 をコンパイラで実行する際に, そのスクリプトが次に実行するスクリプトに影響を与えないことを保証する必要がある．この問題を解決するために, 本研究ではサーバーのコンテナ化を行う．

研究をするにあたり得られた, サーバー上で script 言語を実行する場合の利点と欠点について述べ, 今後の展望について記載する．

2. Perl6

Perl6 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

[1] 琉球大学工学部情報工学科
[a)] k.fukuda@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

て書かれた C 実装のみだった. Perl6 は設計と実装が分離している. 言語的な特徴としては, 独自に Perl6 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. Perl6 は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 Perl6 と Perl5 は別言語としての開発方針になっている. Perl6 は現在有力な処理系である Rakudo から名前を取り Raku という別名がつけられている.

Perl6 の現在の主流な実装は Rakudo である. Rakudo は MoarVM, と NQP と呼ばれる Perl6 のサブセット, NQP と Perl6 自身で記述された Perl6 という構成である. MoarVM は NQP と Byte Code を解釈する.

NQP とは Not Quite Perl の略で Perl6 のサブセットである. その為基本的な文法などは Perl6 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 Perl6 の事を Rakudo と呼ぶ. Rakudo は MoarVM の他に JVM , Javascript を動作環境として選択可能である.

Perl6 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.

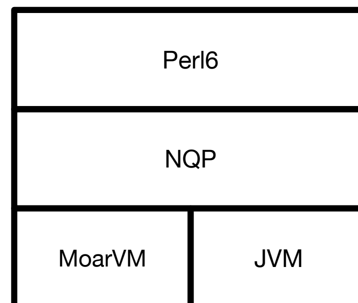

図 1: Rakudo の構成

2.1 MoarVM

MoarVM は Perl6 に特化した 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, 起動時間などが低速である問題がある. 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.

3. NQP

Rakudo における NQP は現在 MoarVM, JVM 上で動作する. NQP は Perl6 のサブセットであるため, 主な文法などは Perl6 に準拠しているが幾つか異なる点が存在する. 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. この MoarVM のバイトコードの状態を Stage0 と言う. Perl6 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. MoarVM の ModuleLoader は Stage0 にある MoarVM のバイトコードで書かれた一連のファイルが該当する.

Stage0 にあるファイルを MoarVM に与えることで, NQP のインタプリタが実行される様に

なっている．これは Stage0 の一連のファイルは, MoarVM のバイトコードなどで記述された NQP コンパイラのモジュールである為である．NQP のインタプリタはセルフビルドが完了すると, nqp というシェルスクリプトとして提供される．このシェルスクリプトは, ライブラリパスなどを設定して MoarVM の実行バイナリである moar を起動するものである．

NQP のビルドフローを図 2 に示す．Rakudo による Perl6 処理系は NQP における nqp と同様に, moar にライブラリパスなどを設定した perl6 というシェルスクリプトである．この perl6 を動かすためには self build した NQP コンパイラが必要となる．その為に Stage0 を利用して Stage1 をビルドし NQP コンパイラを作成する．Stage1 は中間的な出力であり, 生成された NQP ファイルは Stage2 と同一であるが, MoarVM のバイトコードが異なる．Perl6 では完全なセルフコンパイルを実行した NQP が要求される為, Stage1 を利用してもう一度ビルドを行い Stage2 を作成する．

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．

4. なぜ Perl6 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 Perl6 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．

5. Perl6 による Abyss の実装

提案手法で実装した Abyss サーバーは Perl6 で書かれているクライアント側から投げられた Perl6 を実行するためのサーバーである．図 3 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 Perl6 を直接立ち上げるのではなく, まず図 3 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 Perl6 スクリプトの実行を行うため, Rakudo の全体的な処理時間を短縮できると推測できる．

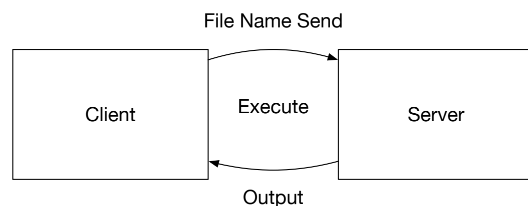

図 3: Abyss サーバーを用いたスクリプト言語実行手順

Code1 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, その後ファイルを受け取るための待機ループに入る．

Perl6 では EVAL 関数 [3] があり文字列を Perl6 のソースコード自身として評価できる

Perl6 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．

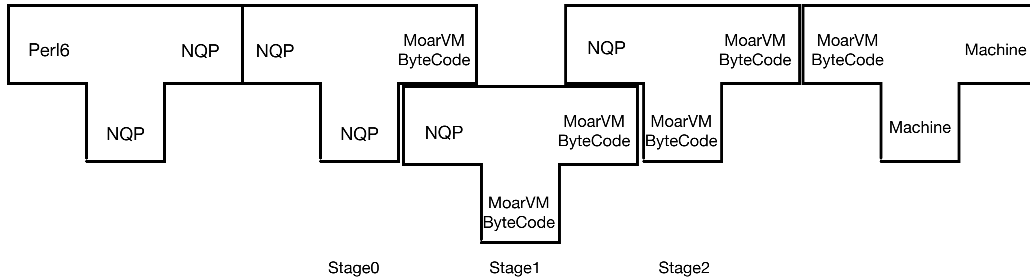

図 2: NQP のビルドフロー

Code1 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.

```
unit class Abyss::Server:ver<0.0.1>;
use MONKEY-SEE-NO-EVAL;

method readeval {
  my $listen = IO::Socket::INET.new(
    :listen,
    :localhost<localhost>,
    :localport(3333)
    );
  loop {
    my $conn = $listen.accept;
    while my $buf = $conn.read(1024) {
      EVALFILE $buf.decode;
    }
    $conn.close;
  }
}
```

Code 1: Abyss サーバーの実装の source code

```
my $conn = IO::Socket::INET.new( :host<
     localhost>,
                               :port(3333) );

$conn.print: 'FILEPASS';
```

Code 2: クライアント側の source code

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

EVAL "say␣{␣5␣+␣5␣}"; # OUTPUT: 10
```

Code 3: eval のサンプルコード

6. 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.

- 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. Perl6 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.

7. まとめ

本稿では実行する Perl6 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ことで全体的に処理時間が早くなること

を示した.

今後 Abyss サーバーでの開発をより深く行っていくにあたって以下のような改善点が見られた

- 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.
- 今回の実装では TCP ソケットを用いたが TCP ソケットを用いるとサーバーを立ち上げた際に外部からファイルを転送される可能性があるので, Unix domain socket の実装を行い, それを用いたクライアント・サーバーを作成することで安全性が高まると考えた. Perl6 には現状 Unix domain socket の実装がないので, Unix domain socket を実装し, 自分以外が実行できないようにすることが今後の課題に挙げられる.
- 今回用いた Perl6 の EVALFILE 自体にクライアント側に出力を返す実装追加することも今後の課題に挙げられる.
- 現状の Raku の EVALFILE では, 出力がサーバー側に返っているので, クライアント側から出力を見るためにクライアント側に返す必要がある.
- モジュールを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

また script 言語をサーバー上で実行する場合の欠点については以下のようなものが見られる

-

今後の開発を行っていくにあたって, 他の Python のような 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 アクセス)

[4] The Official Raku Test Suite (https://github.com/perl6/roast/)

[5] NQP - Not Quite Perl (https://github.com/perl6/nqp)

[6] ThePerlFoundation: Perl 6 Design Documents, ThePerlFoundation (online), available from (https://design.raku.org)