オブジェクト指向スクリプト言語Ruby次期バージョンの開発

The development of the next version of Ruby

松本 行弘
Yukihiro MATSUMOTO

(株) ネットワーク応用通信研究所
〒690-0826 松江市学園南 2-12-5 HOYO パークサイドビル 2 階
E-mail: matz@netlab.co.jp

ABSTRACT. Ruby the object-oriented scripting language, which is designed and implemented by the author, shares attributes with well known scripting languages such as Perl, Python. This project developed the base technologies, such as multilingualization, perfomance improvement by the virtual machine and generational garbage collection, for the next version of Ruby.

1 背景

オブジェクト指向スクリプト言語 Ruby[1] は、筆者によって 1993 年より開発されており、1995 年の一般公開以来、国内外で広く使われている。 Ruby はすべてのデータがオブジェクトであること、そして充実したクラスライブラリが特長となっている。

Ruby と同様の分野を対象とするスクリプト言語には、他にも Perl[2] や Python[3] があるが、それらと比較すると Ruby は Perl よりもプログラムが読みやすく、Python よりもプログラムが簡潔になる傾向がある。また、Ruby は日本製であるため、Perl や Python よりも日本語表記に用いられる EUC や SJIS などの多バイト文字列をよく取り扱える。処理系の性能の面から言えば、実行速度は多くの場合 Perl よりは数割程度劣るが、Python よりは高速である傾向がある。

2 目的

このように現状においても他言語に対して一定の競争力をもつ Ruby ではあるが、今後の改善の余地は残されている。

その第一は性能である。インタプリタ型言語である Ruby はもともとそれほど性能は高くなく、従来はそれを求められてもいなかったが、スクリプト言語の適用範囲が広まるにつれ、処理系の実行性能に対する要求は高まりつつある。

次に、多バイト文字列の扱いに対しての要求もある。現状の Ruby は EUC、SJIS、UTF-8 という主に日本で扱われるコードセットを扱えるに過ぎない。真の意味の国際化という観点から考えるとこれでは不足である。

しかし、過去の多言語化システムの経験から考えると、Unicode 対応で問題が解決するとは思えない。少なくとも近い将来では解決しない。既存の多バイト文字列データを活用し、柔軟な枠組みが必要とされている。

本開発では、これらの問題を解決を目指し、スクリプト言語のベストソリューションとして世界的に認知されることを目標とする。

3 全体構成

本開発は以下の部分からなる。

(1) インタプリタ実装の改善

(2) GC 実装の改善

(3) 多言語対応

このうち、「インタプリタ実装の改善」および「GC 実装の改善」は主に Ruby のインタプリタ性能の向上を目的とし、「多言語対応」は国際化のベースとなる複数言語のためのエンコーディングを扱う枠組みを提供することを目的とする。

3.1 インタプリタ実装の改善

現在の Ruby インタプリタは単純な構文木インタプリタを改造して作り上げたものである。インタプリタに与えられたプログラムは、まず構文解釈部でノードがリンクして構成される構文木に変換される。文法エラーなどはこの構文木への変換の時点で検出される。

インタプリタのコアはこの構文木をたどりながら解釈実行を行う。現在の実装では、構文木の枝の解釈は、解釈関数を再帰的に呼び出すことで実現されている。

この実装は素朴であるが、以下の欠点がある。

- 処理系がつぎはぎで見通しが悪く、保守性に欠ける
- 構文木のトラバース処理などが負荷となり性能が低い
- 再帰と alloca を多用しているためスタック消費量が大きい
- C 言語での再帰のため、末尾再帰などが実現しにくい
- 構文木 (ノード) の個数が多いためメモリ消費量が大きい

そこで、この構文木インタプリタを一種の仮想マシンに置き換えることを考える。このことにより、以下のような改善が期待できる。

- 再設計による見通しの良さ

  現在のインタプリタは Ruby の文法と仕様の拡大につれ、つぎはぎ的に拡張されてきたが、文法や仕様がほぼ安定した現在、再設計することにより無駄をなくした見通しの良い実装が期待できる。

- スタック消費量の低減

  仮想マシンの実装には再帰を使わないことと、仮想マシン自身のスタックはヒープにとることになると思われることから、スタック消費量は画期的に低減することが期待できる。

- 命令フェッチコストの低減

  プロファイルの結果から現在の実装では、次の命令をフェッチするためポインタをたぐって構文木をトラバースするコストが無視できないことが観測されている。仮想マシンを導入することで、命令のフェッチのコストの低減が期待できる。

- 実行時のメモリ消費量の低減

  ヒープ上に割り当てた構造体のリンクである構文木よりも、仮想マシンコードの法がメモリ消費量が小さいことが期待できる。

- さらなる最適化の可能性

  変形の難しい構文木よりも仮想マシンコードの方が最適化の余地が大きいことが期待できる。

このような予想されるメリットが大きいので、インタプリタのコアとして仮想マシンの導入を考える。ただし、現在の Ruby の仕様は相当複雑であり、今回の開発の期間ではとうてい全体を完成させることはできない。そこで、今回はコアの仮想マシンのごく一部を実装し、これらの期待が本当かどうかを検証する。

3.2 GC 実装の改善

現在の Ruby インタプリタはコンサバティブなマークアンドスイープ方式のガーベージコレクタ (garbage collector, GC) を採用している。この GC は、他スクリプト言語で多く採用されているリファレンスカウント方式と比較すると以下の利点がある。

- マークアンドスイープ方式では、C 言語によるメソッドの実装でオブジェクト参照数を維持する必要がなく、リファレンスカウント方式でしばしば発生する参照数の更新忘れに由来するメモリリークが発生しない。

- リファレンスカウント方式では、オブジェクトが直接、間接に自分自身を参照することによって発生する参照のサイクル構造を回収できない。マークアンドスイープ方式ではサイクル構造に対しても問題が発生しない。

一方、マークアンドスイープ方式にはデメリットもある。

- マークアンドスイープ方式では、処理を一時中断し、オブジェクト群をスキャンすることで、すでにどこからも参照されていないオブジェクトを発見する必要がある。このスキャンによる処理時間が現時点で使われている (生存している) オブジェクト数と最後のスキャン処理以降に生成されたオブジェクト数の合計に比例するため、大量にオブジェクトが生存している局面では、実行効率に問題がある。

スクリプト言語の通常の利用では GC が問題になるケースは希だが、上記のデメリットの存在により、マークアンドスイープ方式を採用している Ruby のような言語では、リファレンスカウント方式を採用している Perl や Python のような言語と比較して、大量のオブジェクトを生成するタイプのプログラムの実行性能が低い傾向があることが知られている。

本開発では、この点を改善するために世代別方式の GC (generational garbage collector) を実装する。

オブジェクトの振舞いを観測すると、生成されたオブジェクトのほとんどは比較的寿命が短く、割合の少ない長寿命のオブジェクトは、ほとんどプログラム全体にわたって生存するという性質が一般的である。世代別方式とは、この性質を利用して、生成されて間もない「若い」オブジェクトを重点的にスキャンする方式である。通常の GC におけるスキャンでは若い世代に属するオブジェクトのみをスキャンし、ごく低い頻度で古い世代も含めた全体のスキャンを行う。

若い世代のみをスキャンする場合には、古い世代に属するオブジェクトのスキャンは行わないため、そこから参照されている若い世代のオブジェクトが、参照されていないとみなされないように、そのようなオブジェクトは「リメンバードセット (remembered set)」と呼ばれるテーブルに登録しておく必要がある。

世代別方式によれば、大量のオブジェクトが生存している局面でも、そのほとんどは長寿命オブジェクトとして分類されていることが期待されるため、スキャンを行うオブジェクト数が抑えられ、結果として GC の速度が向上することが期待される。

このように世代別方式の GC では、現状の Ruby の GC の弱点の解消が期待できる。

3.3 多言語対応

現在の Ruby ではユーザが明示的に指定することにより、ASCII、EUC、SJIS、UTF-8 の各エンコーディングを処理することができる。しかし、現段階では一般的な日本語処理に十分であるというレベルであり、国際化プログラミング機能としては不足していると考えられる。

Perl や Python では、多言語化 (multilingualization, M17N) の問題に対して、文字集合の和を目指す Unicode[4] を利用して対応しようとしている。しかし、Unicode は決して万能ではない。

まず、既存文字エンコーディングとの変換における時間・空間的コストの問題がある。既に既存エンコーディングによる大量のマルチバイトデータを持つ我々日本人などにとって、全てのマルチバイトデータを処理する前に Unicode に変換するコストは膨大である。

また、Unicode の文字集合は漢字を含む日中韓の文字集合において「同じ文字」とみなされる文字の共通化 (Han

Unification) が行われているため、既存文字集合の文字の順序は保存されず、その結果、それらの文字集合のエンコーディングと Unicode の間には計算による変換が行えない。よってエンコーディングの変換には、巨大なテーブルに依存する必要があるため、プログラムサイズが肥大化する点も無視できない。

また、日本では多バイト文字データの扱いの歴史、特に複数エンコーディングの並立の歴史が長く、Unicode と各エンコーディングの間の変換規則が一意に定まらないケース (yen sign problem など) も問題となる。

最後に Unicode の文字集合よりも大きな数万から数十万文字を含む TRON コード [5]、GT コード [6] や文字鏡 [7] のような大文字集合の存在がある。スクリプト言語によってこれらの文字集合を扱うニーズがある以上、Unicode であれば十分という結論は安易である。

本開発では、他言語にみられるような単なる Unicode 対応に過ぎない多言語対応を超えて、積極的に複数エンコーディングをそのまま処理できる機能を実現する。具体的にはマルチバイト文字列のエンコーディングをユーザが定義できることを許し、複数のエンコーディングを変換なしで同時に取り扱うことを許す。

過去の Ruby 開発において、EUC, SJIS, UTF-8 の 3 エンコーディングをそのまま取り扱った経験から、マルチバイト文字列の操作は実はいくつかの基本関数をそれぞれのエンコーディングごとに定義することによって、比較的効率良く実現できることが分かっている。そこでそれを自由にユーザ定義できる API を開放することを考える。

このことによって、ユーザは拡張ライブラリを定義することによって自分が使いたいエンコーディングをサポートする機能を Ruby に追加できる。

4 評価

4.1 インタプリタ実装の改善

今回は開発期間の関係上、開発は仮想マシンの設計と実行系の一部の実装にとどまった。今回実装したのは、メソッド定義、ローカル変数参照 (一部)、メソッド呼び出し部のみである。今回のプロトタイプは、構文解析を行いバイトコードを生成する部分を持たないため、試験としてに用いるための、再帰的関数呼出しのベンチマークにしばしば用いられる tak (竹内) 関数が、バイトコードにハンドコンパイルされて配列データとしてプログラム中に埋めこまれている。

tak 関数の実行時間は以下の通りである。

従来の Ruby	5.122sec
プロトタイプ	0.922sec

プロトタイプはいくつかの重要な処理 (たとえば構文解析) を行っていないため、この数値は参考値に過ぎないが、今後より高速なインタプリタの実現の可能性を示唆している。

4.2 GC 実装の改善

今回の世代別 GC の実装により、occur.rb の実行時間は以下のようになった。

従来の Ruby	4.517sec
世代別 GC	3.879sec

これにより、世代別 GC が実行速度向上に有効であることが明らかになった。また、GC のログからは GC の回数は３５％増加したものの、スキャンを行ったのベオブジェクト数は実に 1/30 に抑えられていることが明らかになった。

今回計測した occur.rb は、大量のオブジェクトを生成し、ゴミがほとんど発生しないという、マークアンドスイープ方式にとって最悪、世代別 GC にとって理想的な状況ではあるが、その他のプログラムにおいても５％から５０％程度の GC 処理時間の減少が観測されている。心配されていたメモリ消費量の増加についても、最悪のケースでも１０％強程度の増加でとどまっていることが明らかになった。

4.3 多言語対応

今回の開発によって、たとえば、フレームワークを利用して EUC-JP 対応を追加するために必要なコード量は C プログラムでわずか 109 行であった。UTF-8 の場合にはもう少し多くて 132 行であった。

このエンコーディングサポートのフレームワークによって、現時点ではサポートしていない Big5(台湾) や EUC-KR(韓国)、KOI-8 (ロシア) のようなエンコーディングも、同様にわずかの記述で実現できると考えられる。

また、Ruby の文字列クラスライブラリに対して、これらのエンコーディング情報を用いるような拡張を行った。これによりユーザ定義エンコーディングに応じた「文字」の定義で文字単位の操作が可能になった。

5 今後の対応

5.1 インタプリタ実装の改善

今回は開発期間の関係上、インタプリタ実装は満足の行くレベルの実装は叶わなかった。Ruby インタプリタのコア部分は本質的に複雑で、今回実装したプロトタイプは実用的なレベルではない。しかし、今回実証されたメソッド呼び出しが 5 倍から 7 倍高速化される可能性に対する期待は大きい。特に Ruby の場合、整数の加算のような単純な処理も含めてほとんどの処理がメソッド呼び出しによって行われているのでメソッド呼び出しの高速化はより影響が大きい。

今後はこのプロトタイプをベースにより高速な実用レベルのインタプリタを完成させる予定である。

5.2 GC 実装の改善

世代別 GC による GC 実装の改善は性能的に十分に満足のいく成果をあげた。今後は残る些細な技術的問題を解消して、Ruby のリリースバージョンに反映し、広く公開して活用してもらいたい。

5.3 多言語対応

今回の開発でエンコーディングをユーザが定義できるようなフレームワークは完成した。しかし、今回開発したフレームワークでは対応できるユーザ定義エンコーディングには以下の制限が存在する。

- 対応するエンコーディングがステートレスであること
- 対応する文字集合が ASCII のスーパーセットであること
- 対応するエンコーディングが ASCII のスーパーセットであること

これらの制限が緩和できるかどうかについて、さらなる検討が必要だと思われる。

現時点の感触では、ファイルエンコーディングとしては、最初の制限以外は緩和することが可能であると思われる。最初の制限についてはなんらかのコード変換を行うことで対応するものと考えられる。

次に、このエンコーディングフレームワークを用いた文字列クラスライブラリであるが、今回の開発により、とりあえず「文字」を扱うメソッドでエンコーディングに応じた文字単位の一連の処理が可能になった。しかし、現在の仕様では、プログラム中で頻繁に明示的なエンコーディングの指定を必要とする。これはプログラムの簡潔性と明瞭さを疎外する可能性があるので、今後はユーザの立場からより使いやすい仕様を検討する必要がある。

最後に、文字列クラスライブラリの使いやすさの問題とも関連するが、複数のエンコーディングを扱うということは、実際には複数の文化を扱うことでもある。単に複数のエンコーディング扱えるという多言語化のレベルを越えて、各言語の背景となる文化や慣習まで扱う国際化に対してスクリプト言語がどのように支援できるのかという点は、非常に奥の深い問題を内包している。今後はこの問題にも挑戦したい。

6 参加企業および機関

- (株) ゼータビッツ
- (株) ネットワーク応用通信研究所

7 参考文献

[1] http://www.ruby-lang.org/

[2] http://www.perl.org/

[3] http://www.python.org/

[4] http://www.unicode.org/

[5] http://www.chokanji.com/ck3/

[6] http://www.l.u-tokyo.ac.jp/GT/

[7] http://www.mojikyo.org/