

はじめに

本書はLinux専門誌の「日経Linux」で、2014年4月号から2016年12月号まで連載した「作りながら学ぶプログラミング言語」という記事をまとめた上で加筆・編集したものです。

「プログラミング言語を作る」というテーマの書籍はたくさんあります。私の家の本棚にも何冊も並んでいます。

これらの「プログラミング言語を作る」系書籍のほとんどはプログラミング言語の実装について取り扱っています。例えばyaccやlexというツールをどう使って構文解析器や字句解析器を作るかとか、インタープリタをどのように実装するかというようなことを、サンプルとして比較的単純な言語の実装を通じて解説しています。

プログラミング言語の実装は、実態を知らない人からは非常に高度で困難なものと思われがちですが、実際に段階を踏んで実装するとそこまで難しいものではありません。実際、言語処理系を実装するというのは大学のコンピュータサイエンスの授業では珍しくない課題です。若い人を対象としたプログラミングコンテスト（例えば私が長らく審査員を務めているU22プロコン）などで自作言語を応募してくる高校生もいます。

先入観を除いて取り組めば、言語の設計と実装は、プログラマの知的チャレンジとしては、むしろ楽しく面白い部類だといってもよいでしょう。そういう意味でこれらの「言語を作る」系書籍の意義はそれなりあると考えています。

とはいえ、これらの書籍に不満がないわけではありません。

これらの書籍はあくまでもプログラミング言語の実装について解説するものでしかありません。サンプルとして使われている言語もほとんどは既存の言語の（シンプルすぎる）サブセットになっています。「プログラミング言語を作る」活動のうち、最も知的チャレンジとして面白い「言語をデザインする」部分が全くといっていいほど取り扱われていないのです。

まあ、仕方がないことだとは分かるのです。限られたページ数を有効活用するためにテーマを絞

るのは必然でしょうし、シンプルな言語仕様に収めなければ、とても解説しきれないでしょう。さらに言えば、実際に既存のもののサブセットでないようなプログラミング言語をデザインした経験を持つ人はそれほどたくさんはいません。ましてや、そのデザインした言語が世界中で広く使われた経験のある人など皆無と言っても言い過ぎではないでしょう。

どのように言語をデザインすべきか経験から語れる人など、そうはいないのです。まずは言語処理系を実装する技術の解説が優先で、デザインについては後回しになるのは必然といってもよいでしょう。

数少ない例外としては、C++ 設計者であるBjarne Stroustrup氏による「The Design and Evolution of C++」(邦題「C++の設計と進化」、ソフトバンクパブリッシング)があります。この本を読めばC++がなぜこのようになっているのか、何を目指していたのかが分かる貴重な書籍です。しかし、これをもって自分が言語をデザインするときの方法が学べるかというかなり疑問です。

そのような本が世の中に存在しないのであれば、自分で書くしかありません。幸いにも私は世界的にも数少ない「世界中で広く使われているプログラミング言語」の設計経験があります。さらに趣味をこじらせて、世界中のさまざまな言語のデザインに関する知識があります。さらにさらに、日経Linuxの連載をこなし、複数の書籍の著書としての経験もあります。プログラミング言語のデザインについて語る書籍を執筆するのに、これ以上ふさわしい人材は日本中どこを探してもほかにいないでしょう(自画自賛)。

そのような思いから本連載は、日経Linux 2014年4月号からスタートしました(表1)。

言語を作る動機から、言語デザインにおける葛藤など、ほかの書籍では扱わない領域まで踏み込んだ記述ができたのではないかと自負しています。

ただし、毎月毎月執筆し、書き直しできない雑誌連載を書籍化する場合の宿命として、時間経過に伴う内容の齟齬や不備が付きまといまいます。本書も例外ではありません。

連載時のテーマは表1に示した通りですが、本連載では結局取り扱わなかった組込向けRuby「mruby」についての記述(2014年4月号と6月号の一部)と、本書のスコップから若干外れる型についての解説(2014年9月号から11月号まで)は、本書には掲載しませんでした。さらに、本文の掲載順序を変更し、Streamの現状に合わせて一部手を入れてあります。

しかし、文章の構成などは連載時から大きく変更していません。そこで、解説の一貫性維持や

理解を促進するため、各節の終わりに「タイムマシンコラム」を付けることにしました。「タイムマシンコラム」とは、原稿執筆時に未来のことが分かっていたら、きっとこんな風に書きたろうという補足のニュアンスです。

タイムマシンコラムの存在は、著者としての自分が備える能力の限界を示すようで複雑な気持ちなのですが、未来は誰にも分からないのだと割り切ることになりました。

それでは、言語デザインの世界に皆さんをご案内しましょう。

2016年12月
まつもとゆきひろ

日経Linux 掲載号	タイトル	本書	日経Linux 掲載号	タイトル	本書
2014年4月号	第1回 自作言語入門	1-1	2015年8月号	第17回 ソケットプログラミング	4-1
2014年5月号	第2回 言語処理系の仕組み	1-2	2015年9月号	第18回 CSV	5-3
2014年6月号	第3回 バーチャルマシン	1-3	2015年10月号	第19回 基本データ構造	4-2
2014年7月号	第4回 言語デザイン入門（前編）	1-4	2015年11月号	第20回 さまざまなオブジェクト指向	3-1
2014年8月号	第5回 言語デザイン入門（後編）	1-5	2015年12月号	第21回 Streamのオブジェクト指向	3-2
2014年9月号	第6回 言語の型デザイン（その1）	—	2016年1月号	第22回 オブジェクト表現とNaN Boxing	4-3
2014年10月号	第7回 言語の型デザイン（その2）	—	2016年2月号	第23回 ガーベージコレクション	4-4
2014年11月号	第8回 言語の型デザイン（その3）	—	2016年3月号	第24回 バイブラインプログラミング	5-1
2014年12月号	第9回 抽象的コンカレントプログラミング	2-1	2016年4月号	第25回 バイブライン構成要素	5-2
2015年1月号	第10回 21世紀のコンカレント言語	2-2	2016年5月号	第26回 ロックフリーアルゴリズム	4-5
2015年2月号	第11回 Stream言語の自作	2-3	2016年6月号	第27回 時間表現	5-4
2015年3月号	第12回 Stream言語のコア	2-4	2016年7月号	第28回 統計基礎の基礎	5-5
2015年4月号	第13回 マルチスレッドとオブジェクト	2-5	2016年8月号	第29回 Stream文法再訪	3-3
2015年5月号	第14回 キャッシュとシンボル	2-6	2016年9月号	第30回 パターンマッチ	3-4
2015年6月号	第15回 AST（抽象構文木）	2-7	2016年10月号	第31回 乱数	5-6
2015年7月号	第16回 ローカル変数と例外処理	2-8	2016年11月号	第32回 ストリームグラフ	5-7
			2016年12月号	最終回 あとがきと落ち穂拾い	—

表1 本連載のテーマ一覧

まつもとゆきひろ 言語のしくみ

目次

はじめに.....	3
-----------	---

第1章 さあ、どんな言語を作ろう 9

1-1 自ら言語を作る価値	10
1-2 言語処理系の仕組み	20
1-3 バーチャルマシン	30
1-4 言語デザイン入門(前編)	42
1-5 言語デザイン入門(後編)	53

第2章 新言語「Stream」の設計と実装 65

2-1 抽象的コンカレントプログラミング	66
2-2 新言語「Stream」とは	78
2-3 文法チェッカーをまず作る	89
2-4 イベントループ	100
2-5 マルチスレッドとオブジェクト	114
2-6 キャッシュとシンボル	126
2-7 AST(抽象構文木)に変換	136
2-8 ローカル変数と例外処理	149

第3章 オブジェクト指向機能を実装する 161

3-1 さまざまなオブジェクト指向	162
3-2 Streamのオブジェクト指向	172

3-3	Stream文法再訪	183
3-4	パターンマッチ	196

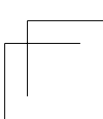
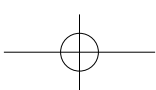
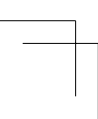
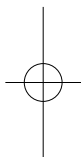
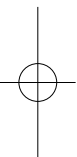
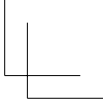
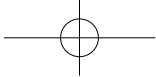
第4章 Streamオブジェクトを実装する 209

4-1	ソケットプログラミング	210
4-2	基本データ構造	222
4-3	オブジェクト表現とNaN Boxing	233
4-4	ガーベージコレクション	244
4-5	ロックフリーアルゴリズム	254

第5章 ストリームプログラミングを強化する 267

5-1	パイプラインプログラミング	268
5-2	パイプライン構成要素	280
5-3	CSV処理機能	291
5-4	時間表現	302
5-5	統計基礎の基礎	314
5-6	乱数	326
5-7	ストリームグラフ	338

おわりに	351
------------	-----



第 1 章

さあ、どんな言語を作ろう

1

- 1

自ら言語を作る価値

プログラミング言語を実際に作りながら、言語の設計と実装の方法を学んでいきましょう。オープンソースの普及により、自分で言語を作る敷居はぐっと下がりました。言語を作れば、技術者としての自らの価値を高め、より大きな“楽しみ”を得られるようになります。

私はプログラミング言語「Ruby」の作者として世に知られていますが、その正体は無類のプログラミング言語マニアです。Rubyとは、私が趣味で調べてきたプログラミング言語に関する研究の集大成であり、むしろRubyの方が趣味の副産物といってもよいでしょう。副産物でこれだけ広まったのだから大したものですが、これはひとえに私の実力のたまもの、というよりは運が大きかったと思います。Rubyが誕生してから20年と少し、これまでにあった、さまざまな出来事や出会いがなければRubyはとてもここまで来られなかったと思います。

言語を作るという世界

ところで、みなさんはプログラミング言語を作ったことがありますか？ プログラミングをしたことのある人にとってプログラミング言語は非常に身近な存在ではあります。しかし、ほとんどの方にとってプログラミング言語は「すでにそこにある」ものであり、自分で作ろうとは思わないかもしれません。その感覚は自然です。

しかし、人間の話す言語（自然言語）と異なり、世にあるすべてのプログラミング言語は、どこかの誰かが設計し、実装したものです。しかも自然に発生したのではなく、明確な意図と目的を持って「デザイン」されたものなのです。ですから、過去に言語を作ろうとした人（言語デザイナー）がいなければ、私たちは今でもアセンブラでプログラミングしていたかもしれません。

プログラミングの歴史のごく初期から、言語はプログラミングに寄り添ってきました。プログラミングの歴史は言語の歴史と言っても過言ではありません。

本書の目標は、このプログラミング言語を自分で作ることです。「今更言語を作ってどんな意味があるの?」と思った方には、後でじっくりお答えするので少し待ってください。まずは自作言語の歴史をひも解いてみましょう。

■ 自作言語の歴史

初期の言語は企業の研究所 (FORTRAN、PL/I)、大学 (LISP)、規格委員会 (ALGOL、COBOL) など、「仕事」で「真面目」に言語に関わる人たちによって開発されました。いわば「プロの犯行」ですね。しかし、この流れはコンピュータが個人の手が届くようになった1970年代以降変化していきます。個人レベルで自分のコンピュータを手に入れたホビーイストたちは、趣味でプログラミングを行い、さらには趣味の一環としてプログラミング言語まで作り始めたのです。

その代表的なものがBASICです。BASICそのものは米Dartmouth大学で教育用プログラミング言語として誕生しました。その言語仕様がシンプルで、最低限の実装がごく小規模で済んだため、1970年代のホビーイストたちは好んでこの言語を使うようになりました。

ホビーイストはさらに、BASIC自体の開発も始めました。当時のパーソナルコンピュータ^{*1}のせいぜい数Kバイトしかないような貧弱なメモリーでも実行できるような小規模なものです。これらの小規模BASICはプログラムサイズが1Kバイト以下くらいで、4Kバイト程度のメモリーでも動作しました。現代の言語処理系から考えると驚異的です。

マイコン雑誌の時代

個人レベルで開発されたBASICをはじめとする小規模言語処理系 (タイニー言語) は、そのうちさまざまな形で配布されることになります。当時のソフトウェアは、コンピュータ雑誌にダンプリストが掲載されたり、プログラムデータを音声変換してソノシート付録に収録されたりすることで配布されました。ソノシートと言っても今の人は分からないでしょうね。ソノシートとはプラスチックの薄っぺらいレコードのことです。レコードもほとんど死語ですが、当時ホビーイストに一般的な外部記憶装置だったカセットレコーダーの代わりに、レコードプレイヤーをコンピュータにつないで読み込ませたのだそうです。

1970年代から80年代にかけてはコンピュータ雑誌 (当時はマイコン雑誌と呼ばれた) の全盛期で、

- RAM (廣済堂出版)
- マイコンピュータ (電波新聞社)
- I/O (工学社)
- ASCII (アスキー)

の4誌がしのぎを削っていました。この4誌の中で今でも残っているのはI/Oだけで、それもあの

^{*1} マイクロコンピュータやマイコンピュータの略として、しばしばマイコンと呼ばれました。

頃とはだいぶ雰囲気も変わってしまいました。当時を知るものとしては寂しい限りです。

この後「マイコンピュータ」から「マイコンベーシックマガジン（通称ベーマガ）」が誕生したりといろいろと歴史があるのですが、年寄りの昔話になりそうなので、この辺でやめておきます。30代から40代のプログラマに話を聞くと、その多くが喜々としてこの時代のことを語り出すのではないのでしょうか。

さて、当時のマイコン雑誌は先ほど述べたようにBASICを収録したソノシートを付録につけていたりしていたのですが、それ以外にもいくつも小規模言語を紹介していました。例えば、GAMEやTL/1などがあります。これらは時代を反映したとても面白い言語なので、別コラムで紹介します。p.16～19のコラム「マイコン雑誌で紹介されていたタイニー言語」をぜひ読んでみてください。

現代の自作言語

なぜ1970年代後半から1980年代前半にかけて自作言語が台頭したのでしょうか。最大の理由は、やはり開発環境の入手が困難だったからだと思います。

1970年代後半にマイコンとして一般的だったのは、TK-80（図1）のような基盤むき出しのワンボードマイコンでした。多くは半完成品で、自分でハンダ付けする必要がありました。開発環境など付いてくるはずもなく、ソフトウェアは機械語を自分で入力して動かす必要がありました。

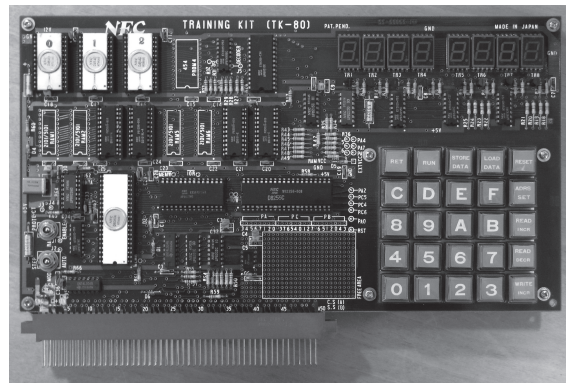


図1 TK-80

1970年代末期にはPC-8001やMZ-80のような完成品の「コンピュータ」が登場します。しかし開発環境はせいぜいBASICで、自分で自由に開発言語を選ぶのは困難でした。商用の言語処理系は販売されていましたが、Cコンパイラの定価が19万8000円するなど、庶民の手に簡単には入りませんでした。となると、自作言語を開発しようという意欲も出るというものです。

しかし現代では、そのように開発環境の入手で困ることはもうありません。各種のプログラミング言語や開発環境がオープンソースソフトウェアとして公開されています。オープンソースでなくても無償のものがインターネットを通じて簡単に入手できます。

となると、今更自作言語を作ろうというのは無意味なことなのでしょうか？この質問への答えが「はい」だと本書は第1章のはじめにしておしまいになってしまいます。

あくまでも個人的な意見ですが（そして本書のためには当然の答えでもあります）、答えは「いいえ」です。たとえ現代であっても新しい言語を個人レベルで設計することには意味があります。それもかなり重要な意味が。

それに、現在広く使われている言語の多くも、開発環境が容易に入手できるようになってから、

そのような個人レベルで設計・開発されています。個人レベルの言語開発が本当に無意味であるならば、Rubyも、Perlも、Pythonも、Clojureも誕生しなかったのです。

あ、それでもJavaやJavaScript、Erlang、Haskellは誕生していたと思います。これらは業務や研究の一環として誕生していますから。

■ なぜ言語を作るのか

では、現代において個人が自作言語を設計・開発することのモチベーションはいったいなんだろうか。

私自身を振り返ったり、ほかの言語作者からの情報を基に考えたりすると、その背景には以下のようなものがあると考えられます。

- プログラミング能力の向上
- デザイン能力の向上
- 自己ブランド化
- 自由の獲得

まずプログラミング言語の実装は、コンピュータサイエンスの総合芸術といえるでしょう。言語処理系の基礎である字句解析や構文解析は、ネットワーク通信のデータプロトコル実装などにも応用できます。

言語機能を実現するライブラリとそれに含まれるデータ構造の実装は、コンピュータサイエンスそのものです。特にプログラミング言語は応用範囲が広く、どのような局面で利用されるか事前に予測することが困難なので、より難易度が高く、それだけ面白いともいえます。

また、プログラミング言語はコンピュータと人間をつなぐインタフェースでもあります。そのようなインタフェースをデザインすることは、人間がどのように考え、暗黙のうちに何を期待しているかについての深い考察が求められます。そのような考察を重ねることは、言語以外のAPIのデザインや、ユーザーインタフェース(UI)、ひいてはユーザーエクスペリエンス(UX)の設計に役立つでしょう。

自己ブランドを高められる

人によっては意外と思うかもしれませんが、IT業界においてプログラミング言語そのものに興味を持つ人は少なくありません。プログラミングと言語は切っても切れない関係にあるので当然なのかもしれませんが。その一端は、言語に主眼をおいた勉強会やカンファレンスなどがたくさんの出席者を集めていることから分かります。そんな状況ですから、ネットで新しい言語を見かけたらとりあえず試す、という人はたくさんいます。Rubyも1995年にインターネットで公開した直後に、わ

ずか2週間ほどでメーリングリスト参加者が200人以上も集まり、驚いたものです。

しかし言語を試す人はそれなりにいても、プログラミング言語、それも雑誌の付録のような「おもちゃ」のレベルを超えて、なんとか実用になるような言語を設計・実装する人はほとんどいません。実用的なプログラミング言語を作ったよというだけで尊敬してもらえること請け合いです。

このオープンソース時代において技術者がサバイバルするためには、技術者コミュニティにおける存在感が非常に重要です。オープンソースソフトウェアを公開することだけでもかなりサバイバル効果が高いのですが、プログラミング言語の「特別感」は、そのブランド効果をさらに高められそうです。

何より楽しい

そして、何よりもプログラミング言語の設計と実装は楽しいのです。本当です。コンピュータサイエンスのさまざまなところに関連するチャレンジングなプロジェクトであることもそうです。さらにプログラミング言語をデザインすることは、それを使うプログラマの思考を支援したりデザインしたりすることにもなるので、そういう点でも面白いです。

しかも普通なら、プログラミング言語はどこから「与えられて」、不可侵なものというイメージですよね。自分で作った言語なら、そんなことはありません。自分が好きのようにデザインし、気に入らなければ、あるいはより良いアイデアを思いつけば、自由に変化させることができるのです。これはある意味、究極の自由だといえるでしょう。

プログラミングとはある意味、自由の追求なのだと思います。自らプログラミングすれば、ほかの人の作ったソフトウェアを単に使っているだけでは享受できない自由を手に入れます。少なくとも私にとってはこれがプログラミングをする重要な動機の一つです。私には、プログラミング言語をデザインすることは、さらに1段高い自由を手に入れる手段であり、また、楽しみと喜びの源泉なのです。

なぜ言語を作る人が少ないのか

とはいうものの、プログラミング言語を作るのは誰でもがすることではありません。前述の通り、プログラミング言語に関心を持つ人はそれなりにいますが、作ろうとする人はほとんどいません。まあ、「関心を持つ人はそれなりにいます」といっても、全人口比からいえば誤差の範囲程度の少なさです。その中からさらに「自分の言語を作ろう」というほどのモチベーションを持つ人は、ほとんどいないというもの当然かもしれません。

私自身はプログラミングに関心を持った数年後にはすでにプログラミング言語に魅了されていました。ところが「すべての人がプログラミング言語に関心を持つわけではない」という事実が気が付いたのは、大学に入学してコンピュータサイエンスを専攻してからでした。幸か不幸かすごく田舎で育ったもので、周りに比較できるプログラミング好きがいなかったためです。

「あれ、もしかして私は変わってる?」と自覚した時の衝撃は相当でした。だって、当時のマイコ

ン雑誌ではTL/1などの言語記事も結構載っていました。プログラミングに関心を持つ人は（自分同様に）かなりの確率でプログラミング言語にはまるものだと思っていたんですが、事実はそうではなかったのですね。

まあ、そもそもプログラミング言語に関心がない人たちは仕方がないとして、関心がある層でも自分でデザインするところまでなかなかいきません。

それはなぜかということについて私は長らく考えていました。先輩言語デザイナーとして、プログラミング言語関連のイベントに参加したときなどには「あなたもどうですか」と勧誘してみたりもしました。しかし、なかなか良い反応が得られません。もちろん、何か新しいことを始めるには相当のエネルギーが必要です。それにしただって、反応が悪過ぎます。

難しく考える必要はない

いろいろ聞いてみたところ、実際に手を動かさない最大の理由が分かってきました。新しい言語を始めることに興味はあっても、どうやらその先になにか心理的な壁があるようなのです。つまり、「言語というものはすでに存在しているものでそもそも自分で設計・開発するようなものではない」という意識です。珍しくそんな壁を感じない少数派の人がいても、今度は「言語を実装するのは難しそう」と思っているようなのです。面白そうだからやってみたいけど、どうやって実装したらよいのか分からない、というわけです。

考えてみれば、プログラミング言語の実装に関する書籍は、意外とたくさん出版されていますが、ほとんどは大学の教科書レベルでかなり難解です。また、「文法クラス」とか「フォロー集合」など難しい用語も頻出します。

しかし考えてみれば、私たちの目的は、自分のために、自分の楽しみのために、自分の言語をデザインすることです。プログラミング言語をゼロからの実装するために必要な知識を、すべて正しく身に付けることではないはずです。むしろ、正しい知識を身に付けるまでプログラミング言語の実装に全く取り掛かれないならば大きな問題です。自分の胸の内にある情熱の火が小さくなってしまいます。

偉大なことを成し遂げるのに1番必要なものは情熱ですから、これではいけません。必要な知識などは、後からおいおい必要に応じて身に付ければよいのです。

本書では言語実装のための難しい部分には踏み込まないで、簡単な言語処理系を作るのに必要な最低限の知識とツールの使い方をカバーします。理論的背景よりは言語をどのように設計するかという点に注力したいと考えています。

マイコン雑誌で紹介されていたタイニー言語

GAME

GAME (General Algorithmic Micro Expressions) はBASIC から発生したタイニー言語です。最大の特徴は、予約語がすべて記号である点と、すべての文が代入文である点でした。

例えば「?」に代入すると数値出力になり、逆に「?」を変数に代入すると数値入力になります。文字列の入出力には「\$」を用います。さらに「#」に行番号を入力するとgoto、「!」への行番号の代入はgosub (サブルーチンの呼び出し) になります。

あとは"ABC"のように文字列を置くと文字列出力になります。それから「/」は改行出力です。

なかなか興味深い言語ですが、そのサンプルを図Aに示します。BASICのような、そうでないような感覚をお楽しみください。

GAMEは非常にシンプルな言語なので、8080アセンブラで記述されたインタープリタは、1Kバイト未満でした。さらに中島聡氏 (なんと当時高校生) によって開発されたGAME自身によって記述されたGAMEコンパイラも存在し、それはわずか200行ほどでした。驚くべきはGAMEの記述力か、それとも中島さんの技術力でしょうか。

TL/1

同じ頃「ASCII」で発表されていたタイニー言語にTL/1 (Tiny Language/1) というものがありました。名前は米IBM社で開発されたプログラミング言語PL/Iのもじりなんでしょうね。BASICに影響を受けつつ記号を駆使したGAMEとは異なり、TL/1はPascalっぽい文法を持ち、より「普通の言語」の印象がありました。またTL/1の処理系はコンパイラで、インタープリタ主体のGAMEよりも高速であるとの評判でした。実際には前述のようにGAMEにもコンパイラは存在したのですが。

言語としてのTL/1の特徴は、Pascalっぽい構文と変数の型が1バイト整数しかない点です。これでどうやってプログラムを書くのかと思うかもしれません。しかし当時の主流は8ビットCPUですから、これでも自然といえないこともないです。もっともいくら8ビットCPU上だとはいえ、ほかの言語はGAMEも含めて16ビット整数を提供していました。

では、1バイトで表現しきれない255を超える数値はどうやって表現するかというと、1バイトずつ分割して複数の変数を組み合わせて表現します。例えば16ビット整数を格納するには変数を二つ使うことになるわけです。演算時には計算がオーバーフローしたときのキャリーフラグを見ながら計算します (図B (a))。当時の8ビットCPUでは、ほとんどの処理は16ビット整数で十分でしたから (アドレスも16ビットあれば全アドレス空間をアク

セスできました)、タイニー言語としてこれは十分な仕様でしょう。その気になれば明示的にキャリアフラグを見ることで、複数の変数を使って24ビット演算でも32ビット演算でも可能です。

ポインタも文字列も扱える

あと、1バイトだけでは表現できないのはポインタです。こちらはmem配列を使ってアクセスしました。つまり、

```
mem(hi, lo)
```

でhi, loで表現される16ビットで指定するアドレスの内容を参照して、

```
mem(hi, lo) = v
```

で、そのアドレスの値をvに書き換えます。当時のパーソナルコンピュータ(マイコン)は、せいぜい32Kバイト程度しかメモリーがありませんでしたから、16ビットアドレスでアクセスできれば十分だったのです。

あとは文字列です。もちろん文字列をバイトの並びと考えると、1バイトずつ操作することも可能ですが、それでは処理が大変です。そこでTL/1では、データの出力に用いるWRITE文を工夫しました。

例えばHello WorldプログラムをTL/1で記述すると図B(b)のようになります。TL/1では変数は1バイト整数しかないはずなのに文字列が登場しています。実は、WRITEは特別に文字列が扱えるように文法へ組み込まれています。

WRITEの外では文字列を扱えないので、一般的な文字列処理はできません。操作できるのはあくまでも1バイト整数のみです。現代の感覚からは奇妙に思えますが、タイニー言語というわけでもないPascalやFORTRANでも入出力は特別扱いされていましたから、当時はそれが普通だったのかもしれません。

```
100----- Comment -----
110|   行番号の直後がスペースでない行はコメントとなる
120-----
130
200 / " FOR は 変数名=初期値, 終値 ... @(変数名 + ステップ) " /
210  A=1,10
```

```

220    ?(6)=A
230    @=(A+1)
240
300 / " IF 文の例 " /
310    B=1,2
320    ;=B=1 " B=1 " /
330    ;=B=2 " B=2 " /
340    @=(B+1)
350
400 / " 数値入力と演算 " /
410    "A = ?" A=?
410    "B = ?" B=?
420    "A + B = " ?=A " + " ?=B " = " ?=A+B /
430    "A * B = " ?=A " * " ?=B " = " ?=A*B /
440
500 / " 配列と文字出力 " /
505-----配列のアドレスを$1000とする
510    D=$1000
520    C=0,69
525-----2バイト配列として書きこみ
530    D(C)=(C+$20)*256+C+$20
540    @=(C+1)
560    C=0,139
570-----1バイト配列として読み出して文字出力
580    $=D:C)
590    @=(C+1)
600
700 / " GOTO と GOSUB " /
710    I=1
720    I=I+1
730    !=1000
731*   ?(8)=I*I
740    ;=I=10 #=760
750    #=720
760
900 / " プログラムの終了 " /
910    #=-1
920
1000 / " サブルーチン " /
1010   ?(8)=I*I
1020 ]

```

図A GAMEのサンプル


```
% (a)
% コメントは行頭の「%」。当時は日本語は使えない
BEGIN
  A := 255
  B := A + 2    % overflow
  C := 0 ADC 0  % add with carry
END

% (b)
BEGIN
  WRITE(0: "hello, world", CRLF)
END
```

図B TL/1 サンプルプログラム

タイムマシンコラム

当初は「mruby」を改造するつもりだった

2014年4月号から始まった連載第1回に相当する部分です。言語を設計することについて熱く語っていますね。

この連載第1回の時点では、どのような言語を作るのか思い付いていませんでした。この時点の構想では私が作っている言語処理系（の一つ）であるmrubyを改造するつもりでいました。その結果、連載時には、mrubyのソースコードの入手法や、ソースツリーの構成概要などについても解説していました。しかし、実際にはmrubyのソースコードは全く用いなかったため、本書では割愛しました。

とはいえ、mrubyが比較的シンプルで言語実装の教材として有効であることは変わりありません。もし、mrubyのソースコードを読んで勉強したいな、と思われた方は「<http://www.mruby.org/>」をスタート地点としていろいろ調べてみるとよいと思います。また、ソースコードは「<https://github.com/mruby/mruby>」から入手できます。

その後、疑問・要望が出たり、バグを見つけたりしたら、GitHubのイシュートラッカーを通じて報告してください。ただし、mrubyの開発は国際化が進んでいるので、イシューの記述も英語が推奨されています。英語でははばかられるという人がもしいらっしゃれば、（できればシンプルな英語でチャレンジしてほしいのですが）私のツイッター（@yukihiko_matz）を通じてレポートしてくださいと日本語で対応可能だと思います。

1-2 言語処理系の仕組み

今回は自作言語のデザインを始める準備として、プログラミング言語とその処理系の関係、そして仕組みについて概観します。まずは電卓プログラムを作りますが、実用的な言語処理系の例としてmrubyの実装についても紹介しましょう。

プログラミング言語を作ろうと言っても、それがどんな作業なのか具体的にイメージできる人はさほど多くないのではないかと思います。ほとんどの人は既存の言語を学ぶだけで、言語をデザインするなど考えたこともないでしょうから。

言語と言語処理系

プログラミング言語は多層的な構造を持ちます。まず大きく分類すると、プログラミング言語とは、コミュニケーションのルールである「言語」と、その言語を処理してコンピュータで実行させる「言語処理系」に分けられます。多くの人は、プログラミング言語という単語を使うとき、言語と言語処理系がごっちゃになっていると思います。

そして言語は、「文法」と「語彙」から構成されます。文法とは、どのような記述がその言語でのプログラムの表現になるのかを定めたルールです。もう一方の語彙は、その言語で記述されたプログラムから呼び出せる機能の集合です。この語彙は、後からライブラリのような形で増やしていきます。言語をデザインするという文脈で、語彙の話をするときには、その言語が初めから備えている組み込み機能のことをいいます。

お気づきでしょうが、この文法や語彙を定めるのに、ソフトウェアは不要です。「ボクの考えた最強の言語」をデザインするのにコンピュータは必要ないのです。事実、私もプログラミングの技術をほとんど持っていなかった田舎の高校生時代、いつかプログラミング言語を作りたいと、妄想プログラミング言語で記述したプログラムをノートに書きつけていました。以前、実家に帰ったときに探したのですが、そのときのノートはもうどこかに行ってしまっていました。たぶん捨てちゃったんでしょね。もったいない。もうどんな言語だったのかも覚えていません。PascalとLispに強く影響されていたような気がするのですが…。

一方、言語処理系の方は、その文法や語彙を実際にコンピュータ上で実行できるようにするためのソフトウェアです。プログラミング言語が単なる妄想を越えて、実際の「言語」になるためには、

やはり処理系が必要でしょう。実行できないプログラミング言語はコンピュータを動かせない以上、厳密にはプログラミング言語とは呼べませんから。

言語処理系の仕組み

「言語処理系を作るぞ」と言ったところで、言語とその処理系がいったいどんな仕組みになっているのか知らなければ、作ることもありません。とはいえ、ここでは、既存の言語処理系を利用して楽するのが基本テーマです。技術的詳細にはこだわらず、まずは概要を押さえることにしましょう。

言語処理系というのは、コンピュータサイエンスの塊のようなものですから、非常にエキサイティングなソフトウェアではあります。コンピュータサイエンス専攻の大学生であれば、言語処理系の作り方は多少なりとも学んだことがあるはずです。コンピュータサイエンスの基礎（の一つ）と言っても過言ではありませんね。「言語処理系を作る」書籍や教科書が世の中にたくさんあるのも無理はありません。

とはいっても、多くの「プログラミング言語の作り方」を扱った書籍では、処理系をいかに作るかという点に重点を置き過ぎています。言語のデザインがどうあるべきか、どのように考えて言語をデザインするかという点に触れているものは、ほとんど（または全く）ありません。まあ、そういう本での「プログラミング言語の作り方」の意味は「プログラミング言語処理系の作り方」なのでしょうから、そういう点では自然なのかもしれませんが。

教科書の目的は、いつか言語を作ろうと思ったときに、その言語をどう作るのか、その手段を教えることが目的です。いつか言語を作る日が本当に来るのかどうかはスコープの外になっています。しかし本書はどちらかというと、前者、つまり「言語デザイン」の方に焦点を当てようと思っています。とはいえ、かつての私のようにノートの上に「理想の言語」を夢想するだけというのでは現実味がありません。導入として、言語処理系に関する最低限の知識については解説します。

まずは言語処理系がどのように構成されているかについて、です。

言語処理系の構成

言語処理系は、大きく分けると文法を解釈する「コンパイラ」、語彙に相当する「ライブラリ」、そしてソフトウェアを実際に動作させるのに必要な「ランタイム（システム）」に分けられます。この処理系の三つの構成要素は、言語や処理系の性質によって比重が変化します（図1）。

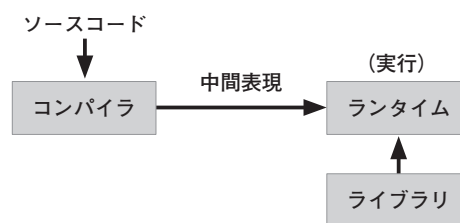


図1 言語処理系の構成要素

古いタイプの言語、例えば、TinyBASICのようなシンフルなシステムでは文法も小さいし、コンパイラはほとんど仕事をしないでランタイムに任せてしまいます。このような処理系を「インタプリタ」と呼びます(図2)。

しかし現在では、このような純粋なインタプリタ型はむしろ少数派です。近代的な言語では多くの場合、「プログラムを最初に内部表現にコンパイルしてから、内部表現をランタイムで実行する」タイプの処理系になっています。もちろんRubyもその一つです。このような「コンパイラ+ランタイム」を組み合わせた形も、外から見るとソースコードを変換しないで実行しているように見えることから「インタプリタ型」と呼ぶことがあります。

あるいは、Cのような機械に近いレベルで効率を追求する言語では、ランタイムは極限まで薄く、文法を解釈するコンパイラの部分だけが突出することになります。このような言語処理系を「コンパイラ型」と呼びます(図3)。言語処理系の構成要素と、言語処理系自身の分類が同じ名前であって紛らわしいですね。Cなどでは変換結果のプログラム(実行形式)は直接実行できるソフトウェアですから、実行を担当する「ランタイム」は不要です。メモリー管理など一部のランタイムはライブラリや、OSのシステムコールが担当します。

Rubyのような「外から見るとインタプリタ型だが中ではコンパイラが動作している」タイプの言語処理系があれば、逆にJavaのように「外から見るとコンパイラ型だが中ではインタプリタ(仮想マシン)が動作している」というものもあります。Javaでは「プログラムを仮想的なコンピュータの機械語(JVMバイトコード)に変換し、実行時には仮想的なコンピュータ(JVM)が実行を行う」というハイブリッド型になっています(図4)。

さらにJavaでは、実行効率を高めるため、ランタイムの中でバイトコードを実際のコンピュータのマシン語に変換してしまう「Just In Time Compiler」などという仕組みまであり、どんどん複雑になっていきます。

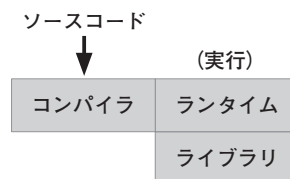


図2 BASIC言語処理系
コンパイラとランタイムが一体化した「インタプリタ型」の例。多くの場合、ライブラリも分離されていない。

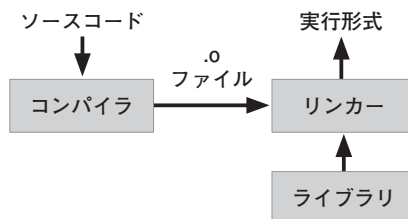


図3 C言語処理系
実行形式を出力する「コンパイラ型」の例。実行形式は直接実行されるためランタイム相当はほとんどなく、ライブラリが一部ランタイムに相当する(メモリー管理など)。

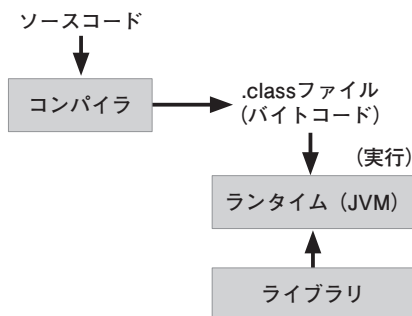


図4 Java言語処理系
仮想マシンコンパイラ型の例。コンパイラは仮想マシンのマシン語(バイトコード)を出力。実行はランタイム(Java仮想マシン-JVM)が担当する。

コンパイラの構成

それでは言語処理系を構成する各要素の中身について、さらに見ていきましょう。まずはコンパイラです。

コンパイラはプログラミング言語のソースコードを実行できる形式に変換するのが仕事です。「Compile」というのは「編集する」という意味です。

多くのコンパイラは、この変換処理を複数の段階に分けて行います。ソースコードに近い順から「字句解析」「構文解析」「コード生成」「最適化」です。ただし、この分類はあくまでも目安で、すべてのコンパイラがこれらの段階のすべてを実行しているわけではありません。

(1) 字句解析

字句解析とは簡単に言うと「文字の列であるソースコードを、意味のあるトークンの列に変換する」工程です。単なる文字列であるソースコードを、もう少し意味のある塊(トークン)にまとめることで、後の段階の処理を簡単にします。例えば、Rubyのプログラム

```
puts "Hello\n"
```

を字句解析にかけると

```
識別子(puts) 文字列("Hello\n")
```

と変換されます。単語の並びの持つ「意味」などは後ろの構文解析の工程で解釈されます。トークンは、構文解析工程で扱うための「単語」のようなものです。

字句解析の処理は基本的に次のようになります。構文解析器が関数を呼び出して次のトークンを要求すると、その関数内部でソースコードから1文字ずつ取り出します。そしてトークン一つ分まとまった時点で、次のトークンを返すという形です。

字句解析関数の実装のために「lex」というツールがあり、トークンを記述するルールから自動的に字句解析関数を生成してくれます。例えば、数字と簡単な四則演算のための字句解析関数を作るlex記述は図5のようになります。

数値のところのルールを見れば分かるように、トークンを構成するパターンの記述に正規表現を使えます。この例では、演算子、数値、空白しかトークンがありませんが、もちろんこの延長線上で、さまざまなトークンを増やしていくことができます。このlex記述（仮にcalc.lというファイルに保存されていると仮定します）をlexにかけるとlex.yy.cというCファイルを生成します。これをコンパイルすると字句解析をするyylex()という関数が使えるようになります。

このようにlexを使えば、字句解析を簡単に実現できますが、実はmrubyはlexを利用していません。これはRubyでは構文解析で決まる状態によって、同じ字面でも違うトークンを発生させることがあるからです。実際には、lexでも状態付き字句解析関数を記述するのは可能なんです。でも自力で書くのはそんなに難しいことではないし、「自分で書いてみたかった」というのも理由に含まれます。Rubyを作ったあの頃、私は若かった。

(2) 構文解析

で、字句解析フェーズが用意してくれたトークンが文法に適合しているかどうかをチェックし、文法に即した処理をするのが構文解析です。

構文解析をする手法はいくつかあるのですが、最も有名で、かつ簡単なのが構文解析関数生成ツール、別名「コンパイラコンパイラ」を使う方法です。コンパイラコンパイラの代表的なものにyacc(yet another compiler compiler)があります。mrubyもyacc、より正確にはそのGNUバージョンであるbisonを利用しています。コンパイラコンパイラにはyaccのほかにもANTLRとかbnfcとかがありますが、ここでは説明しません。

yaccでは、コンパイラが解釈する文法はyacc記述という記法で記述します。例えば、電卓の入力の構文は図6のようになります。

最初の「%%」までの部分は定義部分で、トークンの種別や型を定義します。また「%[」と「%}」に囲まれた部分は、生成されるCプログラムにそのまま埋め込まれるので、ヘッダーファイルのインクルードなどをします。

```
%%
"+"          return ADD;
"-"          return SUB;
"*"          return MUL;
"/"          return DIV;
"\n"         return NL;

([1-9][0-9]*)|0|([0-9]+\.[0-9]*) {
    double temp;
    sscanf(yytext, "%lf", &temp);
    yylval.double_value = temp;
    return NUM;
};

[ \t] ;

. {
    fprintf(stderr, "lexical error.\n");
    exit(1);
}
%%
```

図5 電卓のためのlex記述「calc.l」

「%%」と「%%」で囲まれた部分が電卓の文法定義です。これはBNF (Backus-Naur Form) と呼ばれる文法定義の記法がベースになっています。後ろの「%%」以降もCプログラムにそのまま埋め込まれるので、アクション部から呼び出す関数の定義などをここに置きます。

“電卓”の文法を見てみる

では電卓の文法を見てみましょう。最初のサンプルですから、非常にシンプルな文法にしています。通常の電卓同様、演算子の優先順位も何もありません。

最初のルールから見てみます。BNFはルールの記述になっており、デフォルトでは最初のルールが先頭になります。最初のルールは以下のようになります。

```
program : statement
        | program statement
        ;
```

これは正しい電卓文法はprogramであり、「programとはstatementまたはprogramの後ろにstatementが続くものである」と読みます。「:」が定義、「|」が「または」そして単語の並びはそれぞれの定義が連続するという意味になります。このルールでは単語programが右辺にも登場しており、再帰していることとなりますが、それは構いません。yaccでは繰り返しはこのように再帰を利用して記述します。

次のルールを見てみましょう。

```
statement : expr NL
        ;
```

```
%{
#include <stdio.h>

static void
yyerror(const char *s)
{
    fputs(s, stderr);
    fputs("\n", stderr);
}

static int
yywrap(void)
{
    return 1;
}

}%

%union {
    double double_value;
}

%type <double_value> expr
%token <double_value> NUM
%token ADD SUB MUL DIV NL

%%

program      : statement
              | program statement
              ;

statement    : expr NL
              ;

expr         : NUM
              | expr ADD NUM
              | expr SUB NUM
              | expr MUL NUM
              | expr DIV NUM
              ;

%%

#include "lex.yy.c"

int
main()
{
    yyparse();
}
```

図6 電卓構文解析「calc.y」

これは「statementとはexprの後ろにNLが来たものである」という意味です。NLの意味はここでは定義されていませんが、字句解析が改行に出会ったときに渡すトークンになります。

さて、次です。今度はexprの定義です。

```
expr : NUM
| expr ADD NUM
| expr SUB NUM
| expr MUL NUM
| expr DIV NUM
;
```

これは、exprが「NUM（数値を表すトークン）、またはexprに続いて演算子そして数値が続いたもの」という意味です。ここでも再帰による繰り返しが使われています。ですから、「1」は数値ですからexprです。「1+1」はexprである1と演算子「+」そして数値の並びですからexprです。同様の理屈で「1+2+3」などもexprです。

だいたいBNFの仕組みがイメージできましたか？

ここまで記述してきた電卓プログラムを実行してみます（図7）。図5のプログラムをlexにかけ、図6のプログラムをyaccにかけた上で、生成されたy.tab.cというCソースファイルをコンパイルすると、電卓の文法チェックが完成します。計算する部分を全く実装していないので、単なる文法チェックしか行いません。入力が文法的に正しければ何もせず、文法エラーがあれば「syntax error」と表示して実行が終了します。

```
% lex calc.l  ← 字句解析生成
% yacc calc.y ← 構文解析生成
% cc y.tab.c  ← コンパイル
% a.out       ← 実行
1 + 1         ← 正しい式を入力
2             ← これも正しい式
1 +           ← 文法エラーを試す
syntax error  ← エラー表示して終了
%
```


 このマークで改行

図7 電卓プログラムのコンパイル・実行

電卓プログラムを実装する

計算もできない電卓には意味がありませんので、実際に計算させてみましょう。yaccではルールが成立したときに実行する「アクション」を記述できます。図6のyacc記述に実際のアクションを追加して計算や表示をすれば電卓が完成します。具体的には、図6のプログラムのstatementとexprのルールの部分を図8のものと置き換えます。

この電卓サンプルではアクション部で直接計算や表示をしています。いわば「純粋インタープリタ」ですね。が、実際のコンパイラではこのように直接処理を実行することはめったにありません。このままではループによる繰り返しやユーザー関数定義などに対応できませんから。例えばmrubyでは、文法構造を表現する木構造を作って、次のコード生成処理に渡しています。木構造を作る

一つの例として、mrubyのif文は、図9のようなS式（実体は構造体のリンク）に変換されます。

(3) コード生成

コード生成処理では、構文解析処理で生成した木構造をたどりながら仮想機械の機械語を生成します。

JVMの台頭以降、この「仮想機械の機械語」のことを「バイトコード」と呼ぶことが多い気がします。確かにJVMの機械語はバイト単位なのでバイトコードで間違いありません（その元になったSmalltalkもバイト単位でバイトコードです）。しかし、mrubyの機械語は32ビット単位なので本当は「ワードコード」と呼ぶべきなのかもしれません。バイトコードは不正確ですし、ワードコードは一般的に使われていない用語ですから、mruby内部ではiseq（instruction sequence: 命令列）と呼んでいます。またiseqにシンボル情報などを付加したプログラム情報（コード生成の最終的な結果）のことをirep（internal representation: 内部表現）と呼んでいます。

mrubyのコード生成処理は、さほど難しい解析はしません。このためやろうと思えば、構文解析処理のアクション部で直接コードを生成することも不可能ではありません。しかし、いくつかの理由からフェーズを分割して、中間表現としてS式類似の木構造を採用しました。

```
statement : expr NL
          {
            fprintf(stdout, "%g\n", $1);
          }
          ;

expr      : NUM
          | expr ADD NUM
            {
              $$ = $1 + $3;
            }
          | expr SUB NUM
            {
              $$ = $1 - $3;
            }
          | expr MUL NUM
            {
              $$ = $1 * $3;
            }
          | expr DIV NUM
            {
              $$ = $1 / $3;
            }
          ;
```

図8 電卓プログラムのアクション

```
# このRubyプログラムが
if cond
  puts "true"
else
  puts "false"
end

# こういうS式に変換される
(if (lvar cond)
  (fcall "puts" "true")
  (fcall "puts" "false"))
```

図9 mruby構文木構造

mrubyではコード生成を分離

最初の理由はメンテナンス性です。確かに構文解析のアクション部でコードを生成することは可能ですし、それによってプログラムサイズは（若干ですが）縮小されるでしょう。しかし構文解析とコード生成が一体化することで、プログラムはより複雑になり、何か問題があったときに発見するのが困難になると思います。

アクション部はルールのパターンにマッチした順に呼び出されるので、手続き的な動作に比べて実行順序の予測が難しく、デバッグが困難になるケースがあります。メンテナンス性のためにも、アクション部で実行することは構文木の生成だけというシンプルな構成にとどめておくのが賢明だと判断しました。

そうすると組み込みも視野に入れるmrubyとしては、メモリー消費量の増加が気になるのですが、幸いなことに（構文解析やコード生成を含む）コンパイラ部は、実行時に分離することが可能です。Rubyプログラムをあらかじめirepに変換しておくことで、実行時にはコンパイラを不要にする構成が可能になるのです。メモリー容量の厳しい環境でも、そのような対処によってメモリーの節約が可能なので、そこまでメモリー消費量について厳しく考える必要はないと思いました。

さて構文解析結果の木構造をコード生成処理にかけると図10のようなirepを作ります。元々irepはバイナリー（構造体）ですが、それでは意味が分からないので、人間に分かる形式に変換しています。

irepアドレス	使用レジスタ数	ローカル変数の数	定数の数	シンボル数	参照するirep数
irep 0x86ec8b8	nregs=4	nlocals=2	pools=2	syms=1	reps=0
000 OP_LOADF	R1			← R1にfalseを代入	
001 OP_JMPNOT	R1	006		← R1が偽なら006へジャンプ	
002 OP_LOADSELF	R2			← R2にselfを代入	
003 OP_STRING	R3	"true"		← R3に"true"を代入	
004 OP_SEND	R2	:puts	1	← R2のputsメソッドを呼び出し	
005 OP_JMP		009		← 009へジャンプ	
006 OP_LOADSELF	R2			← R2にselfを代入	
007 OP_STRING	R3	"false"		← R3に"false"を代入	
008 OP_SEND	R2	:puts	1	← R2のputsメソッドを呼び出し	
009 OP_STOP				← 実行終了	

図10 コード生成結果（irep）

(4) 最適化

コンパイラの実装によっては、コード生成の前後に「最適化」処理を加えることがあります。mrubyの場合には、Rubyという言語の性質上、最適化を加えにくいので、ごくわずかな最適化だ

けがコード生成処理の最中に加えてあります。

これは「ピープホール（のぞき穴）最適化」と呼ばれるもので、命令生成する時点で直前の命令を参照するだけで可能な最適化をします。mrubyコンパイラが実施している最適化の一部を表1に示します。

種別	元命令	最適化後
無意味な代入の削除	R1=R1	削除
交換命令の削除	R2=R1; R1=R2	R2=R1
代入の削減	R2=R1; R3=R2	R3=R1
代入の削減	R2=1; R3=R2	R3=1
return命令の重複削除	return; return	return

表1 mrubyの最適化

コンパイラの処理の後

mrubyの場合、コンパイラの処理が終わった後に行うことは2種類あります。一つはコンパイル結果をそのまま実行することです。実行にはmruby用の実装された仮想CPUを用います。また仮想CPUの実行には、オブジェクト管理などのランタイムやライブラリも用います。

もう一つは、コンパイラの処理結果を外部ファイルに書き出すことです。これにより、コンパイル結果を直接リンクしたプログラムを作ることができ、コンパイラを取り外した状態でRubyプログラムを実行できます。メモリー制限の厳しい組込システムなどでは有効な方法です。

まとめ

今回は言語処理系の構成について学びました。書いている方としては、言語デザインの話がなくて不満だったのですが、解説のためには仕方ありません。

タイムマシンコラム

言語処理系の解説の難しさ

2014年5月号掲載分です。言語処理系関連の本では付きものの、yaccの使い方などを解説する回です。準備の都合もあって、電卓プログラムという陳腐なサンプルになっているのはあまりほめられませんね。

今回の内容で「ちょっとマシン」な点は、電卓のようなおもちゃプログラムだけでなく、mrubyという実用的言語処理系の構成についても解説している点です。電卓プログラムの解説でコード生成や最適化について触れるのはムリですからね。

ただ、そうは言っても今回の解説は「そういうものがある」と触れるレベルでしかないので、内心不満は残ります。ここは悩ましいところで、mrubyの実装についてあまり詳細まで解説すると必要以上に難しくなり過ぎます。かといって説明しないと不満が残るし、難しいものですね。

ここで説明したyacc記述は、後でStreemの実装解説で何度も登場するので、そのときにこの解説が効いてくるはずです。

1-3 バーチャルマシン

今回はプログラミング言語処理系の心臓部、バーチャルマシンの実装について解説します。バーチャルマシンを実装するための四つのテクニックを紹介した後、mrubyのバーチャルマシンが持つ実際の命令を見てみます。

1-2でも解説した通り、ソースコードをコンパイルした結果を実行するのが「ランタイム」です。ランタイムの実装方法はいくつかありますが、今回のテーマである「バーチャルマシン」（仮想マシン、VM）もその一つです。

ソフトで実現したCPUで動かす

バーチャルマシンという単語はいくつかの異なった使われ方をしますが、今回解説するのは「ソフトウェアで実現された（実際のハードウェアを伴わない）コンピュータ」としてのバーチャルマシンです。

これは仮想化ソフトやクラウドなどの文脈で登場するものとは異なります。仮想化ソフトなどでは「実際に存在するハードウェアをある種のソフトウェアでラップすることで仮想化し、複数実行やハードウェア間の移動を実現する技術」をバーチャルマシンと読んでいます。Wikipediaではそれを「システムバーチャルマシン」、今回解説するものは「プロセスバーチャルマシン」として分類しています。

```
int
vm(node* node) {
    while(node) {
        switch (node->type) {
            case NODE_ASSIGN:
                /* 代入の処理 */
                ...
                break;
            case NODE_CALL:
                /* メソッド呼出の処理 */
                ...
                break;
            ...
        }
        /* 次のノードへの遷移 */
        node = node->next; /* ← ここが遅い */
    }
}
```

図1 構文木インタープリタ（概略）

Rubyの実装では、バージョン1.8までは（プロセス）バーチャルマシンを持たず、コンパイラが生成した構文木（ポインタでリンクされた構造体によって表現されるRubyプログラムの文法に対応した木構造）をたどりながら実行していました（図1）。この方法は非常にシンプルなのはよいのですが、一つの命令を実行するたびにポインタを参照するコストが無視できません。Ruby1.8以前に「Rubyが遅い」といわれていた原因の一つがこれです。

昔のRubyが遅かった理由

このシンプルな構成がなぜそんなに遅いのかという理由は説明が必要だと思います。ハードディスクへのアクセスがメモリーのアクセスに比べて圧倒的に遅いのは多くの人がご存知でしょう。しかしメモリーアクセスの速度についてはどうでしょうか。普通にプログラムを書く上で、メモリーの速度を気にすることはめったにないと思います。

しかし実際には、CPUとメモリーの間はずいぶん「遠い」のです。指定したアドレスにあるデータをメモリーバス経由で取り込んでくる時間は、CPUの実行速度に比較すると圧倒的に低速です。このメモリーアクセスの間、CPUはデータが届くのを待つしかなく、この待ち時間は実行速度に影響を与えます。

このような待ち時間を減らすために、CPUは「メモリーキャッシュ」（memory cache）、あるいは省略して単に「キャッシュ」という仕組みを備えています。キャッシュはCPUの回路の中に組み込まれている少量の高速メモリーです。事前にメインメモリーからキャッシュにデータを読み込んでおき、メモリーの読み書きを高速メモリーに対して行うことでメモリーアクセスの待ち時間を減らし、処理を高速化します。

CPU内部に組み込まなければならない性格上、キャッシュの容量には厳しい制限があり、読み込んでおけるデータはわずかです*1。キャッシュが有効に働くためには、これからアクセスするメモリー領域が既にキャッシュに読み込まれていることが必要です。しかし未来を予測して、これからプログラムが読み書きする領域をキャッシュに取り込んでおくことはなかなか困難です。一般的に、これが可能なのは、メモリーアクセスの局所性が成立するときです。つまりプログラムが一度にアクセスするメモリー領域が十分に小さく、かつ近接していて、一度キャッシュに取り込んできた領域を何度も読み書きする場合ですね。

VMでキャッシュを生かす

残念ながら、図1のような構文木インタープリタはキャッシュアクセスという観点からは最悪です。構文木を構成するノード一つひとつは別々の構造体ですから、それぞれのアドレスが近いとは限りませんし、当然連続もしていません。このため事前にキャッシュに読み込んでくるのは困難です。

*1 現代のCPUはキャッシュを多段に重ねて容量を増やそうとしています。それでもメインメモリーと比較して圧倒的に容量が小さいことと、先読みの困難さが解決されるわけではありません。

ここで、構文木を命令列に変換して、連続したメモリー領域に格納すれば、メモリーアクセスの局所性は高まり、キャッシュのおかげで性能がずいぶん向上するでしょう。

このような方法で性能向上を実現したのが、Rubyバージョン1.9で導入された「YARV」と呼ぶバーチャルマシンです。YARVは「Yet Another Ruby VM」（もう一つ別のRubyバーチャルマシン）の略です。これは開発当時、既にRuby実行を目的としたバーチャルマシンが複数開発されていたことにちなみます。当初YARVは実験的プロジェクトだったのですが、数ある実装のうち実際にRuby言語のフルセットを実行できるまでの完成度に到達したのはYARVだけでした。結果としてYARVが本家Rubyの実行系を置き換えることになりました。

バーチャルマシンの利点と欠点

バーチャルマシンを採用した言語で最も有名なものは、なんといってもJavaでしょう。しかしバーチャルマシンというテクニックは別にJavaで発明されたわけではなく、1960年代の後半には既に登場していたと考えられます。例えば、1970年代初頭に登場したSmalltalkは初期からバイトコードを採用していることで（一部で）有名です。さらに古くは、後にPascalを設計したNiklaus Wirth氏がAlgol68の拡張として設計したEularという言語でバーチャルマシンを実装していたそうです。このEularのバーチャルマシンが、SmalltalkのバーチャルマシンのヒントになったとSmalltalkの父、Alan Kay氏は述べています。

Pascalといえば、UCSD Pascalを思い出します。カリフォルニア大学サンディエゴ校で開発されたUCSD PascalはPascalプログラムをP-codeというバイトコードに変更してから実行するものでした。P-codeを経由することで様々なOSやCPUのコンピュータへの移植が容易になり、UCSD Pascalは移植性の高いコンパイラとして広く使われました。

このことからわかるように、バーチャルマシンのメリットはなんといっても移植性でしょう。それぞれのCPUに合わせてマシン語を生成する「コード生成」処理は、コンパイラの中でも最も複雑な部分です。それを次々と登場するさまざまなCPUに合わせて開発し直すのは、言語処理系開発者にとってかなりの負担でした。

現在ではx86系やARMなどが支配的でCPUのバリエーションは昔と比べてずいぶん減りましたが、1960年代、70年代は次々と新しいアーキテクチャーが登場していました。同じ会社の同じシリーズのコンピュータでも、機種が違えばCPUが全く異なることは珍しくありませんでした。バーチャルマシンはこのような負担の軽減に役立ちました。

さらにバーチャルマシンはターゲットとなる言語に合わせて設計できます。このため用意する命令セットがその言語の実現に必要なものだけに限定できます。汎用CPUと比較すると仕様をコンパクトにすることが可能ですし、その結果、開発も簡単になります。

しかし、メリットばかりではありません。ハードウェアで直接実行するのに比較して、仮想的なCPUをエミュレートして実行するバーチャルマシンは性能上の問題を抱えることになります。バーチャルマシンを採用した処理系は、最低でも数倍、場合によっては数百倍のパフォーマンスペナルティー

が発生します。ただし、「JIT コンパイル」などのテクニックを用いて、このペナルティーを軽減することができます。

バーチャルマシンの実装テクニック

ハードウェアで実装される実際のCPUと、ソフトウェアで実装されるバーチャルマシンでは、性能特性が異なります。ここではバーチャルマシンの性能特性に関連する実装テクニックについて解説します。以下が代表的なものです。

- (1) RISC vs CISC
- (2) スタック vs レジスタ
- (3) 命令フォーマット
- (4) ダイレクトスレッディング

RISCとはReduced Instruction Set Computerの略で、命令の種類を減らし、回路を単純化することでCPUの性能向上を目指すアーキテクチャーです。1980年代に流行したアーキテクチャーで、代表的なCPUとしてはMIPSやSPARCなどがあります。モバイルデバイスに広く使われているARMプロセッサもどちらかというところRISC系ですね。

CISCはRISCと対比されて導入された用語で、Complex Instruction Set Computerの略です。簡単に言うと「RISCでないCPU」のことです。一つひとつの命令が行う処理がかなり大きくて命令の種類も多く、実装も複雑になっています。

しかし、RISC vs CISCの対立があったのは20世紀までのことで、現在、実際のハードウェアCPUでは、RISCとCISCの対立にはあまり意味がなくなっています。なぜならば純粋なRISCのCPUは人気なくなり、あまり見かけなくなりました。それでもSPARCはまだ残っていて、国産スパコンの京などに使われています。

RISC系の中で成長株なのはARMですが、これもどんどん命令が追加されてCISC的に成長してきました。一方で、CISCの代表格のように思われていたインテルx86アーキテクチャーは、表面的には過去との互換性を維持した複雑な命令セット^{*2}を提供しながら、内部ではその命令をRISC的な内部命令(μ op)に変換することで、実行の高速性を実現しています。

バーチャルマシンではCISCが有利

しかし、バーチャルマシンではRISC対CISCに別の意味があります。ソフトウェアで実現されるバーチャルマシンでは、命令を取り出してくる処理(フェッチといいます)のコストが無視できません。

^{*2} 先日、x86のmove命令はあまりに複雑なのでそれだけでTuring完全性を実現できるという発表がありました。つまり、原理的にはmove命令だけで任意のアルゴリズムが記述できるということです。

つまり、同じ処理をするために必要な命令数は少なければ少ない方がよいということです。つまり優れたバーチャルマシンの命令セットは、一つひとつの命令の粒度が高いCISCなアーキテクチャーが優れているということになります。

バーチャルマシンの命令はできるだけ抽象度が高いものにして、プログラムサイズが小さくなるように設計する方が有利です。ある種のバーチャルマシンでは、頻繁に連続して呼ばれる複数の命令を一つにまとめた複合命令を用意して、さらにコンパクト化を目指すものがあります。このようなテクニックを「命令融合」とか「スーパーオペレーター」と呼びます。

(2) スタック vs レジスタ

バーチャルマシンのアーキテクチャーの2大流儀がスタックマシンとレジスタマシンです。スタックマシンは、データの操作を原則的にスタック経由で行います(図2)。一方のレジスタマシンでは、命令の中にレジスタ番号が含まれていて、原則的にレジスタに対して操作をします(図3)。

スタックマシンとレジスタマシンを比べると、スタックマシンの方がシンプルでプログラムサイズが小さくなる傾向があります。しかし、すべての命令がスタックを経由してデータをやり取りする関係で、命令間の順序依存が大きく、命令の並び替えを伴うような最適化は難しくなります。

一方、レジスタマシンはレジスタ情報を命令の中に含むためプログラムのサイズは大きくなる傾向があります。ただし後に述べるようにプログラムサイズと命令フェッチのコストは必ずしも相関しないので注意する必要があります。また、レジスタを明示的に指定するため順序依存性が低く、最適化の余地が広いといわれます。ただし、小規模言語で高度な最適化をする例はあまりないので、この点はさほど重要ではないかもしれません。

バーチャルマシンのアーキテクチャーとして、スタックマシンとレジスタマシンのどちらが優れているかという点についてはいまだに結論が出て

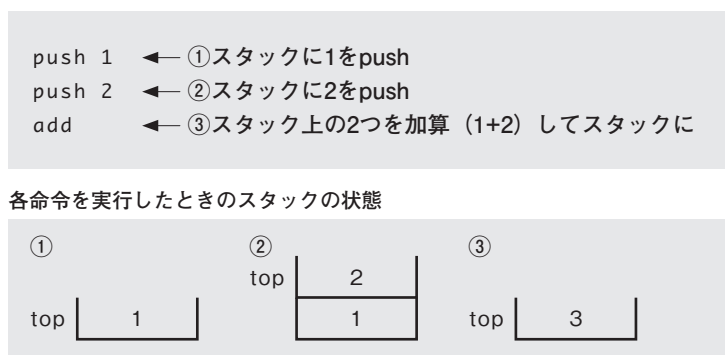


図2 スタックマシンの命令とその仕組み

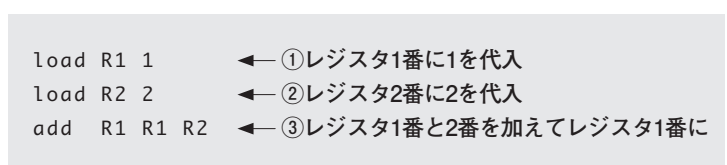


図3 レジスタマシン命令

言語	VM名	アーキテクチャー
Java	JVM	スタックマシン
Java	Dalvik (Android)	レジスタマシン
Ruby	YARV (Ruby1.9以降)	スタックマシン
Ruby	mruby	レジスタマシン
Lua	lua	レジスタマシン
Python	CPython	スタックマシン

表1 各種バーチャルマシンのアーキテクチャー

いません。結果として、どちらのアーキテクチャーを採用したバーチャルマシンもたくさんあります。各種言語向けのバーチャルマシンがどちらを採用しているかを表1に示します。同じ言語でも実装によってスタックマシンを採用したり、レジスタマシンを採用したりしているのが面白いですね。

(3) 命令フォーマット

Smalltalk 以来、バーチャルマシンが解釈するマシン語(命令列)のことをしばしば「バイトコード」と呼びます。これは Smalltalk の命令がバイト単位であったことに由来します。また、この単語はバイト単位という性質を受け継いだ Java によってすっかり広まりました。

しかし、すべてのバーチャルマシンがバイト単位の命令セットを持っているかというそうとは限りません。例えば、YARV も mruby も命令セットは32ビット整数で表現します。32ビット整数は多くの CPU にとって最も扱いやすいサイズの整数として「ワード」と呼ぶことが多いので、これらの命令列は正式には「バイトコード」ではなく、「ワードコード」と呼ぶべきなのかもしれません。しかしワードコードは呼びにくいし、その意味もピンとこないので全然普及しません。我々もそろそろ諦めて、つい「バイトコード」と呼んでしまうことがあります。

“ワードコード”の損得

バイトコードとワードコードにもそれぞれメリットとデメリットがあります。1命令当たり必ず32ビットを消費してしまうワードコードと比較すると、バイトコードの方がプログラムサイズがコンパクトになります。一方、バイトコードは単位となる1バイトが8ビットで256状態しか表現できないため、オペランド(命令の引数)は命令に続くバイトに格納するしかありません。このため命令列からデータを取り出してくるフェッチ回数が増えてしまいます。既に述べたように、ソフトウェアで実現されるバーチャルマシンでは、この命令フェッチのコストが大きいので、性能的には若干ワードコードの方が有利です。

さらにワードコードは「アラインメント」の点でも有利です。ある種の CPU では、アドレスが特定の数の倍数になっていないアドレスへのダイレクトアクセスはエラーになります。その場合はアラインメント(アドレスが特定の数の倍数に揃っている状態)されているアドレスからデータを取り出し、ずれたぶんを切り出すようなことをする必要があります。エラーにならなくても倍数のアドレスとそうでないアドレスとで(内部的に上記の切り出しを行うなどの理由で)アクセス速度がかなり異なる CPU は珍しくありません。

アドレスを2の倍数にする方がよいものを16ビットアラインメント、4の倍数にするのがよいものを32ビットアラインメントと呼びます。ワードコードではすべての命令がアラインメントに合っていることが確実ですが、バイトコードではそうはいきません。CPU の種類やアドレスの状態によってはバイトコードの方が1命令フェッチあたりのコストが高いことはあるでしょう。

まとめるとバイトコードの方が命令列の長さが短くなる傾向があり、消費メモリー量的には有利だが、性能面では、命令フェッチの回数的にも、1フェッチ当たりの所要時間的にもワードコードの方が有利といえるでしょう。

mrubyの命令をしてみる

では、バーチャルマシンの命令の実例を見てみましょう。今回、サンプルとして使うmrubyの命令は図4のようになっています。

mrubyの命令は末尾7ビットによって命令の種類が決まります。命令種別を指定するのが7ビットということは、最大128種類の命令を実現できるということですね。実際には予備の五つを含めて81種類の命令が用意されています。

命令が32ビット幅で命令種別が7ビットということは、残りの25ビットがオペランドのために使えるわけです。mrubyの命令はこのオペランド部の使い方(分け方)によって四つのタイプがあります。

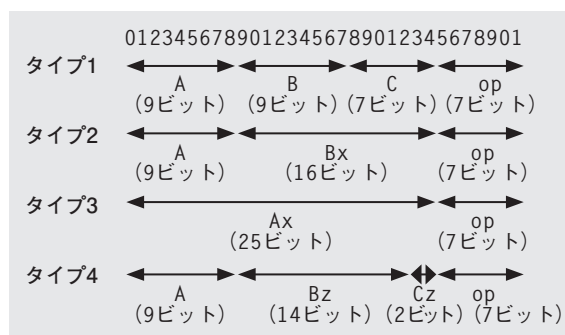


図4 mrubyの命令構造

タイプ1: オペランド3個

命令タイプ1は、A、B、Cの三つのオペランドを持ちます。このときAは9ビット、Bも9ビット、Cは7ビット幅になります。つまりオペランドAとBの最大値は511で、オペランドCの最大値は127です。オペランドAとBはレジスタを指定するのに用いることが多いです。例えば、レジスタ間の移動命令「OP_MOVE」はこのタイプで

```
OP_MOVE A B
```

という命令になります。これはオペランドBで指定されたレジスタの内容をオペランドAで指定されたレジスタにコピーします。OP_MOVE命令ではオペランドCは用いません。

オペランドCを使う命令としては、例えばメソッドを呼び出すOP_SENDがあります。

```
OP_SEND A B C
```

これはオペランドAで指定されたレジスタ（レジスタAと呼びましょう）に格納されているオブジェクトの、オペランドBで指定されたシンボル*（正確にはシンボルテーブルのB番目のシンボル）で指定されたメソッドを呼び出します。A+1番目のレジスタからオペランドCで指定された個数分がメソッド引数になります。メソッド呼び出しの戻り値はレジスタAに格納されます。

先ほど説明したOP_MOVEのようにタイプ1のいくつかの命令は使わないオペランドがあります。

【シンボル】 言語処理系が内部でメソッド名などの識別に使っている値で、任意の文字列に対して異なった値が割り当てられる。

その分ビットパターンがムダになっているわけですが、アクセスの簡便さと効率のために多少のムダを許容しています。

タイプ2:オペランド2個

命令タイプ2は、オペランドB、Cの代わりに一つの大きな（16ビット）オペランドがあります。これには符号なし（Bx）オペランドと符号ありオペランド（sBx）の2種類があり、命令ごとに使い分けています。Bxを使う例としては、OP_GETIVなどがあり、これは

```
OP_GETIV A Bx
```

という形式でシンボルテーブルのBx番目のシンボルで指定されるselfのインスタンス変数の値をレジスタAに格納します。

sBxを指定する命令としてはジャンプ命令があり、例えば

```
OP_JMP sBx
```

は次に実行される命令のアドレスを現在のアドレスからsBx離れた位置に移動させます。sBxは符号付きですから、前方にも後方にもジャンプできます。OP_JMP命令はオペランドAを使いません。一方、条件付きジャンプの例は

```
OP_JMPIF A sBx
```

で、こちらはレジスタAが真であればsBxだけジャンプします。

タイプ3:オペランド1個

命令タイプ3はオペランド部を全部一つにして25ビットのオペランド（Ax）として取り扱います。タイプ3の命令は、OP_ENTERの一つだけです。

```
OP_ENTER Ax
```

OP_ENTERはAxに指定されたビットパターンを見て、メソッドの引数チェックをします。OP_ENTERは25ビットのうち23ビットを5/5/1/5/5/1/1に分割して引数指定と解釈します。各ビットの意味を表2に示します。

ビット	内容
5	必須引数の数
5	オプション引数の数
1	rest引数(*引数)の有無
5	末尾の必須引数の数
5	キーワード引数の数(未対応)
1	キーワードrest引数(**引数)の有無(未対応)
1	ブロック引数の有無

表2 OP_ENTERの引数指定
Axオペランド25ビットを先頭から分割する。

タイプ4:タイプ1の変形

最後の命令タイプ4は、BCオペランド部(16ビット)を14ビットのBzオペランドと2ビットのCzオペランドに分割したものです。命令タイプ4の命令はOP_LAMBDAだけです。

mrubyでは命令からオペランドを取り出すマクロを用意しており、これらを駆使することで、命令(のワード)からオペランドが得られます。これらのマクロでは命令型チェックなどはしませんので、正しいマクロの適用については開発者が責任を持つ必要があります。オペランドを取り出すマクロを表3に示します。

マクロ名	意味
GET_OPCODE(i)	命令種別を取り出す
GETARG_A(i)	Aオペランド(9ビット)
GETARG_B(i)	Bオペランド(9ビット)
GETARG_C(i)	Cオペランド(7ビット)
GETARG_Bx(i)	Bxオペランド(16ビット)
GETARG_sBx(i)	sBxオペランド(符号付き16ビット)
GETARG_Ax(i)	Axオペランド(25ビット)
GETARG_b(i)	Bzオペランド(14ビット)
GETARG_c(i)	Czオペランド(2ビット)

表3 mruby命令のオペランドを取り出すマクロ

インタープリタループ

さて、このような構成でソースプログラムをバーチャルマシンの命令列に変換できれば、バーチャルマシン自身の基本的な実装は意外と簡単です。

バーチャルマシンの中心部分、すなわちインタープリタループは、疑似コードで表現すると図5のようになります。

あまりにシンプルなので驚きましたか?命令が増えても、図5のswitch文のcaseが増えるだけです。

ただし基本的構造はシンプルでも、実用的な言語を実装するためには、いろいろと考える必要があります。ここでは省略した実行時スタックをどのように実装するかとか、メソッド呼び出しの仕組みや例外処理の仕組みをどうするか、などです。原理と実践の間にはやはり大きな溝があるのです。

```
typedef uint32_t code;

int
vm_loop(code *pc)
{
    code i;

    for (;;) {
        switch (GET_OPCODE((i = *pc))) {
            case OP_MOVE:
                stack[GETARG_A(i)] = stack[GETARG_B(i)];
                break
            case OP_SEND:
                ...
                break;
                ...
        }
    }
}
```

図5 バーチャルマシンの基本的構造 (switch文を利用)

(4) ダイレクトスレッディング

多くの場合、実用的なバーチャルマシンは速度優先になりますので、インタープリタループも効率化したいところです。バーチャルマシンのインタープリタループの効率化テクニックとして有名なものは、GCC (GNU Compiler Collection) の拡張機能を用いたダイレクトスレッディングと呼ぶものがあります。

GCCではラベルのアドレスの取得と、そのアドレスへのジャンプができます。ラベルのアドレスは「&&ラベル名」で取得できます。ラベルへのジャンプは、「goto *アドレス」です。この機能を使うと、switch文の代わりにジャンプを使ってバーチャルマシンを構築できます。

ダイレクトスレッディングを用いたインタープリタループの実装の概要を図6に示します。

```
typedef uint32_t code;

#define NEXT i=**++pc; goto *optable[GET_OPCODE(i)]
#define JUMP i=*pc; goto *optable[GET_OPCODE(i)]

int
vm_loop(code *pc)
{
    code i;
    /* 命令番号の順序で並べたラベルアドレス */
    static void *optable[] = {
        &&L_OP_MOVE, &&L_OP_SEND, ...
    };

    JUMP;

L_OP_MOVE:
    stack[GETARG_A(i)] = stack[GETARG_B(i)];
    NEXT;
L_OP_SEND:
    ...
    NEXT;
    ...
}
```

図6 ダイレクトスレッディングを用いた場合

実際にはmrubyを含めて、ダイレクトスレッディングを採用しているほとんどのバーチャルマシンの実装では、コンパイルオプションでswitch文かダイレクトスレッディングのいずれかを選べるようになっています。ラベルアドレスの取得はあくまでもGCCの拡張機能なので、いつも使えると限らないからです。切り替えマクロを用いたループは図7のようになります。

```
typedef uint32_t code;

/* GCC拡張のあるコンパイラのみ対応 */
#if defined __GNUC__ || defined __clang__ || defined __INTEL_COMPILER
#define DIRECT_THREADED
#endif

#ifdef DIRECT_THREADED

#define INIT_DISPATCH JUMP;
#define CASE(op) L_ ## op:
#define NEXT i=*++pc; goto *optable[GET_OPCODE(i)]
#define JUMP i=*pc; goto *optable[GET_OPCODE(i)]
#define END_DISPATCH

#else

#define INIT_DISPATCH for (;;) { i = *pc; switch (GET_OPCODE(i)) {
#define CASE(op) case op:
#define NEXT pc++; break
#define JUMP break
#define END_DISPATCH }}

#endif

int
vm_loop(code *pc)
{
    code i;
#ifdef DIRECT_THREADED
    static void *optable[] = {
        &&L_OP_MOVE, &&L_OP_SEND, ...
    };
#endif

    INIT_DISPATCH {
        CASE(OP_MOVE) {
            stack[GETARG_A(i)] = stack[GETARG_B(i)];
        }
    }
```

```
    NEXT;  
    CASE(OP_SEND) {  
        ...  
    }  
    NEXT;  
    ...  
}  
END_DISPATCH;  
}
```

図7 切り替えマクロを用いた場合

このテクニックを用いると、GCC 拡張のないコンパイラでも、switch 文でそれなりの速度になります。GCC 拡張のあるコンパイラなら、ダイレクトスレッディングテクニックを使って、より高速なバーチャルマシンを実現できます。

まとめ

今回はランタイムの心臓部、バーチャルマシンの実装について解説しました。これで言語処理系の基本部分の解説はひと通り片付けました。来月からはしばらく言語デザインを中心にした解説に移行しようと思います。

タイムマシンコラム

StreemにもVMを採用したいが…

2014年6月号掲載分です。前回のyaccの解説に続き、今度はVMの実装を解説しています。VMの例題がmrubyですが、これはあまりシンプルな例だとVM実装のイメージがつかめないだろうということがあります。そして何よりも最大の理由は、この後に実装するつもりの言語のVMとしてmrubyのものを改造して利用しようと考えていたからです。

実際には、Streemの実装は構文木を直接渡り歩くシンプルなインタープリタを採用したので、ここでの解説はStreemには役に立ちませんでした。しかし、まあ、VM実装の解説としての価値はあると思うので、書籍でも割愛せずに残しました。作者の心づもりとしては、これからStreemのシンプルなインタープリタを、ここで解説したようなVMを利用するものに置き換えたいのですが、最大の難関はどのようにして、その実装のための時間を捻出するかです。時間管理が最大の障壁になるのは、今回に限らずいつものことです。

1-4 言語デザイン入門（前編）

言語の実装については一通り学んできましたので、ここからは言語のデザインについて考えてみましょう。今回はケーススタディーとしてRubyの初期のデザインについて振り返ります。Rubyは、スクリプティングを支援するオブジェクト指向言語として開発を始めました。

あなたが、新しいプログラミング言語を作りたいと仮定しましょう。それも単に自分のおもちゃとしてではなく、あわよくば世界中で使われるような「人気の言語」を目指すのだとしましょう。だとすれば、どのように言語を作ったらよいのでしょうか。

人気の言語の作り方

新しいプログラミング言語の人気を決めるのは、性能や機能よりもむしろ言語仕様です。しかし、どのように言語をデザインしたらよいのかを教えてくれるような書籍もWebページもほとんどありません。

改めて考えてみれば、ちゃんとしたプログラミング言語をデザインしたことのある人はほとんどいません。プログラミング言語に関する教科書はありますが、それらで扱っているのは「プログラミング言語の作り方（実装法）」です。そこで扱っている言語も、あくまでもサンプルですから、既存の言語、あるいは既存の言語のサブセットにすぎません。言語デザインについては教科書のスコープ外ですし、そもそも、その教科書を書いている作者でさえ「人気の言語」をデザインした経験があるわけではないでしょう。

それはそうでしょう。世界中で広く使われるレベルのプログラミング言語は、さほど数はありません。歴史上の「人気の言語」をすべて数え上げても数百もないのではないのでしょうか。まあ、「人気」の定義によるでしょうが。ということは、それらの言語の設計者も世界中で数百人程度しかいないということになります。かつ、そのうち何人かはすでに亡くなっています。

となれば、そのような数少ない言語設計者の一人として、言語デザインの秘訣について紹介しておくのは、私の使命ではないかと考えるようになりました。本書の真の目的はそこにあります。

湧き上がる疑問

言語設計者を志すものが、新しい言語を作ろうとするときに、いつも頭をよぎる疑問に以下のようなものがあります。

- 本当に新しい言語が必要か
- その言語は何をするのか
- 想定されるユーザーは誰か
- どんな機能を採用するか

しかし、これらの疑問の多くは悩むだけ無駄です。そんなことに悩んでいても、良いものができる助けには全くなりません。

考えてみましょう。例えば「本当に新しい言語が必要か」どうかですが、「チューリング完全」な言語であれば、あらゆるアルゴリズムが記述可能です。既存のプログラミング言語がチューリング完全であることはすでに証明されています。だからソフトウェアを開発する（＝アルゴリズムを記述する）という観点からは、未来に登場するものも含めて、新しい言語は全く不要です。

しかし言語を作るということは、そういうものではないでしょう。過去50年以上にもわたって言語が作られ続けてきたのは、別に今までの言語ではアルゴリズムの記述が不可能だったからではありません。新しい言語の方が「書きやすい」あるいは「気分がいい」からです。そもそもこのような疑問を持つということは、あなたの心の奥底に「言語を作りたい」という気持ちがあるからでしょう。そういう気持ちがあるのならば「必要があるかどうか」などと悩む必要はゼロでしょう。

自分が使いたければそれで十分

「その言語は何をするのか」と「想定されるユーザーは誰か」の疑問については、少しだけ補足する必要があります。

私は長年の「プログラミング言語おたく」として、数多くのプログラミング言語について学んできました。また、Rubyが広く知られるようになってからは、何人もの言語設計者と交流してきました。例えば、C++の設計者であるBjarne Stroustrup氏、PerlのLarry Wall氏、PythonのGuido van Rossum氏、PHPのRasmus Lerdorf氏などなど。その経験から知る範囲でいえることがあります。「設計者自身が自分で使うつもりで設計した言語以外で人気になったものはほとんどない」ということです。

自分で使うつもりがなければ、細部まで気を使って設計することもできないでしょうし、人気が出るまで育て上げるモチベーションも維持できないでしょう。言語に人気が出るまで、時として10年以上かかることは珍しくありません。細部への気遣いとモチベーションの維持、どちらも人気の言語のためには不可欠の要素です。つまり人気の言語の「想定されるユーザー」は、まずは設計者

自身であり、またそれと似たような特質を持つユーザーでしょう。そして、「何をするか」は設計者自身が何をしたいかに依存することでしょう。

想定するユーザーと対象とするジャンルが決まれば、最後の疑問である「どのような機能を採用するか」はあまり悩む必要はありません。ただし、そこにも秘訣やコツはあるのですが、それについては後で説明します。

■ R u b y 開 発 の き っ か け

とはいえ、私が勝手に奇麗にまとめても説得力はないでしょうから、ケーススタディーとして、Rubyの例を取り上げます。Rubyであれば、この20年間私が深く関わってきたので、いろいろ語ることができますし。ここでは、特に言語設計の方向性を定めることになった開発初期に集中して振り返ってみることにしましょう。

まずは、Rubyの開発が始まった背景から見てみます。

Rubyの開発を始めたのは、1993年です。私自身がプログラミング言語に関心を持ったのは1980年代初頭、まだ鳥取県に住んでいた高校時代です。その頃からPascal、Lisp、Smalltalkをはじめとしたプログラミング言語に深く関心を持っていました。

まだコンピュータを個人で所有しておらず、ろくに自分でプログラムを書けない頃からなぜだかプログラミング言語の方に関心があったのは不思議なことです。どんなプログラムを書くかよりも、プログラムを書くための手段であるはずの言語に魅了されていたのです。

そんなおかしな「プログラミング言語おたく」な高校生でしたが、なにぶん住んでいたのが地方なので、資料・文献がなくて苦勞しました。まだインターネットというものはありませんでしたし。学校の図書館にもコンピュータ関連書などほとんどない時代でしたから、本当に困りました。

プログラミング言語に関する情報を得るには、コンピュータ雑誌で言語について取り上げられるのを探るか、近所の本屋の店頭で大学の教科書のような書籍が並んでいたのを立ち読みするか（高くて購入できませんでした）くらいしかありませんでした。今でもお世話になった本屋さんには感謝しています。

その後、大学入学後には、図書館に各種書籍や雑誌・論文などが一通りそろっていて、ここは天国ではないかと思いました。そのようにして身に付けたプログラミング言語関連の知識が、後の言語設計に大いに役立ちました。本を読まない小説家や、過去の棋譜を知らないプロ棋士がほとんどいないように、新しい言語を設計するにも既存のプログラミング言語に関する幅広い知識が重要なのです。

1993年にたっぷり時間ができた

で、1993年です。その頃、すでに大学も卒業し、職業プログラマーとして就職していた私は、業務命令に従ってソフトウェアを開発していました。その少し前まで、私が開発していたのは、社内で利用するためのシステムでした。UNIXワークステーション上にデスクトップや添付付きのメールシステムなどを実現していました。今ではWindowsやMacで当たり前になっていますが、当時のUNIXワークステーションにはそのようなシステムはありませんでした。似たようなものがあっても日本語が扱えなかったりで、独力で開発するしかなかったのです。

しかし、1992年頃にバブル経済が破綻すると、会社全体の景気が悪くなりました。コストがかかるだけの社内システムの新規開発はストップになってしまい、すでに開発が終わった部分だけは運用を続けるという経営判断がなされました。

開発チームは解散し、わずかな人員だけがメンテナンス要員として残されました。幸か不幸か、私はメンテナンス要員に割り当てられ、居残ることになりました。といっても、開発は停止していますから、やることなどあまりなく、たまに「コンピュータの動作がおかしいんだけど」と電話がかかってくるのに対して、「では、再起動してください」と応える程度の日々でした。完全に窓際ですね。

書籍の企画がきっかけ

しかし、悪いことばかりでもありません。景気の悪さのため、残業は抑制され、さらにボーナスがなくなったので、バブル期に比較すると収入はずいぶん減りましたが、(当時、新婚だった私には経済的に厳しかったものの)クビにはなりません。このため仕事を探す必要はなく、目の前にはコンピュータがあり、仕事が少なく重要度も低いので管理もされません。時間と気持ちに余裕があり、これは何かやってやろうかという気分になりました。しばらくは、いくつか小さなユーティリティプログラムを開発していましたが、ふとしたきっかけから長年の夢であるプログラミング言語を作ってみようと思ったのです。

その「ふとしたきっかけ」とは、こうです。当時、私と同じ部署にいた先輩が、書籍を出版する企画を立てました。そして、執筆開始という段になって私に相談が来ました。いわく「言語を作りながら学ぶオブジェクト指向というような企画を立てただけで、言語の部分を手伝ってくれないか」。

言語おたくの私としては非常に心惹かれる企画だったので、引き受けることにしました。結局、その企画は編集会議を通らず、すぐにボツになってしまいます。しかし、だからといってせっかくやる気になったのを止める気にはありませんでした。自分の言語を作ることこそ、私の長年の夢だったからです。夢ではあったものの、言語が完成する姿をイメージできず、なかなかモチベーションが上がらないでいました。それが、ようやく火が付いたので、ここで止めるのはあまりにもったいない。

この「やる気」こそが、ここから始まるRuby 20年の歴史の始まりだったのです。この時には、世界中で広く使われるプログラミング言語に成長するなどは夢にも思っていなかったのですが。

■ R u b y の 初 期 デ ザ イ ン

さて、言語を作るとなると頭をよぎる疑問についてはすでに述べました。「本当に新しい言語が必要か」といった四つの疑問です。

20年後の今であれば、そんなことを気にすることはないと断言できるのですが、当時の私はまだ若かった。ここでしばらく考えました。しかし、少し考えた後、自分の言語は自分のために作ろうと決心しました。今思えば、この時のこの判断がすべてを決めたような気がします。

当時の私はCプログラマで、主に使う言語といえばCかシェルでした。仕事で中規模以上のシステムを開発するときにはC、日常的に利用する比較的小規模なプログラムを開発するときにはシェルを使っていました。当時（実は今も）、Cにはあまり不満を感じていませんでしたし、Cが対象にするようなシステム言語を新規に作ることはあまり魅力を感じませんでした。その頃すでにC++が存在していたのもありますし、大学の卒業研究でCベースのオブジェクト指向言語の一つ設計していたのも（あまり満足できる完成度ではありませんでした）、関係していたのかもしれませんが。

シェルに不満があった

むしろ、不満があったのはシェルでした。当時、私はbashを使っていましたが、コマンドラインを並べて簡単な制御構造を追加するくらいならば、これくらいのシンプルな言語で十分でした。しかし、プログラムがだんだん成長して複雑になってくると何をやっているのか分からなくなりがちなのも、ちゃんとしたデータ構造を持たない点も不満でした。要するに、シェルというのは、ちょっとした制御を付け加えられるコマンドライン入力でしかなく、あくまでも「簡易言語」なのが問題でした。

当時、シェルに近い領域（スクリプティングと呼ばれます）で、もうちょっと普通の言語に近い言語としてはPerlがありました。しかし、私の視点からは、この言語もやはり「簡易言語」の雰囲気がありました。データ構造がスカラー（文字列および数値）と配列、ハッシュしかないのも不満です。これではある程度以上複雑なデータ構造を直接的には表現できません。

当時はまだPerl 4の時代で、Perl 5のオブジェクト指向機能は噂のレベルでしか存在していませんでした。しかし、伝え聞くPerl 5のオブジェクト指向もさほど満足できるものではなさそうでした。Perlよりもリッチでデータ構造をきちんと表現できる言語がよいと思いました。また、高校時代からのオブジェクト指向プログラミングのファンとしては、ただ単に構造体などを扱えるだけでなく、ちゃんとしたオブジェクト指向言語にしたいとも考えました。

Pythonは“普通”過ぎた

一方、Pythonという別の言語も存在しているということでした。まだこの言語に関する情報があまりなかった頃なので、苦労していろいろ調べました。その結果、オブジェクト指向機能が後付けなのが不満なのと、もう一つ、今度はあまりにも普通の言語過ぎるのが気に入りません。どれだけ

わがままなのか、と自分でも思うのですが、「理想の言語」の話をさせると言語おたくは止まりません。

「普通の言語すぎる」とはどういうことか、と思われる方もいらっしゃるかもしれません。Python は言語レベルでの正規表現サポートもないし、文字列処理機能も貧弱で、言語仕様レベルでスクリプティングを支援しているようには思えませんでした（あくまで20年前のPythonの話ですよ）。

Python の特徴の一つである、インデントでブロックを表現する試みは面白いとは思いましたが、これには逆に欠点もあります。例えば、テンプレートからプログラムを自動生成しようとしたときに、インデントを正しく維持しないとプログラムが正常に動作しない点とか、あるいはインデントによるブロック構造のために、言語仕様上、式と文が明確に区別されている点とかです。

これだったら普通にLispの方言を使ったときと比べて、若干文法が読みやすい以外に違いはないなと感じました。今思えば、コミュニティとか資産とかをまるっきり無視しているわけですが、当時はまだそれらの重要性には気付いていなかったのです。

スクリプトにオブジェクト指向を

しかし、ほかの言語を見ることによって、自分が作りたい言語の姿がはっきり見えてきました。それはシェルに近い領域で、Perlよりは普通の言語に近く、データ構造を自由に定義でき、かつオブジェクト指向機能を備えている言語です。当然、Pythonよりもシームレスにオブジェクト指向プログラミングを行うことができ、文字列処理をはじめとするスクリプティングに特徴的な機能を支援する機能やライブラリを備えていなければなりません。

この頃、次第にスクリプティングの重要性が高まっており、PerlやPythonの出番も増加しつつありました。しかし、同時にこのようなスクリプティング分野でのオブジェクト指向プログラミングの必要性はまだ十分に認識されていませんでした。

当時のオブジェクト指向プログラミングといえば、Smalltalkか（大学などではLispも）、あるいはC++を指しました。一般には大学における研究のプログラミングか、大規模で複雑なシステム開発にだけ用いられるテクニックと見なされていました。スクリプティングのような小規模かつ比較的シンプルなプログラミングには不必要と認識されていたのです。その状況がようやく変化の兆しを見せ始めていました。

Perlはこれからようやくオブジェクト指向機能の導入を計画中、Pythonはすでにオブジェクト指向言語ではあったものの、機能は後付けですべてのデータがオブジェクトというわけではなく（当時）、やろうと思えばオブジェクト指向プログラミングもできる手続き型言語という立ち位置でした。ここで、スクリプティングを主眼にした手続き型プログラミングもできるオブジェクト指向言語というのが登場したら、きっと使いやすいに違いありません。少なくとも私自身は喜んで使うでしょう。

なんだかやる気が増してきました。私はプログラマの三大美德^{*1}の一つである傲慢をたっぷり

^{*1} Perlの作者Larry Wall氏によれば、プログラマの三大美德とは、無精・短気・傲慢だそうです。普通は美德と呼ばれなさそうな特質ですね。

保持した人物なので、作るからにはPerlやPythonに負けないものを作ると決心しました。またそれが可能であると信じ込んでいました。根拠のない自信というのは恐ろしいものです。しかし、往々にしてこのような無根拠な自信こそがモチベーションの源泉になるのです。

■ R u b y の 開 発 を 開 始

そこで開発開始です。最初に決めたのは名前でした。“名前重要”。Perlが真珠から名付けられたので、それにならって宝石から名付けようと考えて、いくつかの候補の中からRubyを選びました。宝石の名前はDiamondとかEmeraldとか長いものが多くて、なかなか良いのが見つからなかったのですが、最終候補に残ったのがCoral(サンゴ)とRubyでした。結局、Rubyの方が短く、美しいということで、こちらを採用することにしました。このときは深く考えなかったのですが、プログラミング言語の名前は実に頻繁に呼ばれるものですから、呼びやすくて印象的なことが大変重要です。

もし皆さんが、自分の言語を開発することになったら、ぜひ張り切って良い名前を考えてください。言語の特徴を端的に表現する名前があれば、それに越したことはありません。しかしRubyのように言語の性質とは全く関係のない名前でも問題ないでしょう。最近の問題は、ありふれた名前では「グーグナビリティー」(ネット検索での見つけやすさ)が低いということでしょう。Rubyを開発し始めた1993年にはなかった問題です。

ブロック構造の表現方法で思案

名前の次に決めたのは、ブロック構造の表現にendを用いることです。CやC++、Javaはブロック構造で複数の文をまとめるのにブレース({})を用います。このやり方には一つ問題があります。単文を複文にするとき、ブレースを忘れがちなことです(図1)。ブレースの代わりにbeginとendを使うPascalでも結局は単文と複文の区別があるので同じ問題があります。

私はこの単文と複文の問題が好きではなかったので、自分の言語ではそのような問題がそもそも発生しないようにしようと考えました。そのための方法は三つあ

```
// 複文のときにはブレースで囲む
if (cond) {
    statement1();
    statement2();
}

// 単文のときにはブレースで囲まなくてよい
if (cond)
    statement1();

// 単文を複文にするときブレースを忘れる
if (cond)
    statement1();
    statement2(); // 文法エラーにならない
```

図1 単文と複文の問題

りました。

- (1) 単文でのブレースの省略を許さない Perl 方式
- (2) ブロック構造をインデントで表現する Python 方式
- (3) 単文と複文の区別なく、ブロックを持つ構文は end で終わる Eiffel 方式 (図2)

「オートインデント」が課題だった

ところで、私は長年エディタとして Emacs を使ってきていて、このエディタの言語モードに慣れ親しんでいます。そして、この言語モードが提供するオートインデントをこの上なく気に入っています。適当にプログラムを入力すると、エディタが自動的にインデントしてくれるのを見ると、ああ、エディタと協調してプログラムを開発しているなという気分になります。

(2) の Python 方式では、インデントそのものがブロック構造を表現するのでオートインデントの余地がありません（まあ、Python の場合、行末のコロンでインデントが深くなる方は可能ですが）。また、ブロックでインデントを表現する Python では文と式の区別が明確になってしまい、式と文の区別のない Lisp の強い影響を受けていた私はあまり気に入りませんでした。そこで、Python 式のブロックによるインデントは採用しないことにしました。

私が学生時代に強い影響を受けた言語は Eiffel です。学生時代に『Object-oriented Software Construction (邦題:オブジェクト指向入門)』という本を読んで、すっかりカブれた私は、卒業研究としてセマンティクス的に Eiffel っぽい（だけど文法は C っぽい）言語をデザインしました。その試みは正直成功したとは言い難かったのですが、今度はセマンティクスではなく、文法を Eiffel からいただいてくるのはどうだろうかと考えたのです。

Emacs で言語モードを自作

そこで再び懸念になるのがオートインデントです。当時の Emacs の言語モードで、オートインデントしてくれるのは C のようなブロックを記号で表現するものが主流でした。Pascal などの予約語を使ってブロック構造を表現する言語のモードはキー操作でインデントを深くしたり、浅くしたりするものばかりでした。これでは、オートインデントの「気持ち良さ」を失ってしまいます。

■ 単文の場合

```
if cond
  statemen1();
end
```

■ 複文の場合（単文との区別がない）

```
if cond
  statemen1();
  statemen2();
end
```

■ 複数のブロックがあると櫛のようになる

```
if cond
  statemen1();
elsif cond2
  statemen2()
else
  statemen3()
end
```

図2 Eiffel方式（^{クシ}櫛型ブロック構造）

そこで数日間 Emacs Lisp と格闘しました。正規表現などを使って簡易的に Ruby の文法解析を行って、end のある文法でもオートインデントができる Ruby モードのプロトタイプを作成しました。これにより、end のある Eiffel っぽい文法の言語でもオートインデントが可能であることが証明されたのです。これにより、安心して Ruby の文法を end を使うものにできました。逆に言えば、このときオートインデントする Ruby 言語モードの開発に成功していなければ、今の Ruby の文法はなかったということです。

この end を使うブロック構造というデザイン上の選択には、予想もしなかったメリットがありました。Ruby の大部分は C で実装したので、必然的に私は C と Ruby を使い分けることになりました。しかし C と Ruby は見掛けが全く違うために、現在どちらの言語で作業しているかが一目で分かります。それで脳のモード切り替えのコストが低くなりました。もちろん、そのコスト差はわずかなものでしょうが、プログラミング時のノリを妨げないために非常に有益でした。また後に Perl、Python、Ruby がスクリプト言語のライバルと見なされるようになったとき、それぞれの言語が全く違うブロック構造を持っていた (Perl はブレース、Python はインデント、Ruby は end) ことも、生き残りに有効だったのではないかとにらんでいます。

else if か elsif か elif か

余談ですが、上記のような複文対策をすると、その副作用として C のような else if が書けなくなります。C の else if は、else の後ろにブレースなしの if が単文で登場していると解釈されるからです (図3)。Ruby のような文法で、同じことをすると図4のようになります。

```
// (a) else ifを使った以下の文は、
if (cond) {
    ...
}
else if (cond2) {
    ...
}

// ブレースを省略しないとうなる
if (cond) {
    ...
}
else {
    if (cond2) {
        ...
    }
}
```

図3 Cのelse if

```
# つまり、Rubyでelsifがないと
# いつも以下のように書く必要がある
if cond
    ...
else
    if cond2
        ...
    end
end

# やっぱりelsifがあった方がいいね
if cond
    ...
elsif cond2
    ...
end
```

図4 Rubyのelse if

図4を見る限り、やはりelsifがあった方がよさそうです。ちなみにPerlとRubyはelsifで、シェルとPython（とCプリプロセッサ）はelifです。この違いは興味深いですね。

PythonはシェルやCプリプロセッサからelifを継承し、シェルなどは古いAlgol系から引き継いだと聞いた気がします（あとシェルのfiやesacのような始まりの予約語を逆につづったもので終わるのも同じ起源だと聞きました）。

Perlがなぜelsifにしたのかは残念ながら知らないのですが、Rubyの場合は、以下の理由があります。

- else ifを「発音そのまま」で「最短」にしたものがelsif（elseifでは長いし、elifでは発音が変わってしまう）。
- Rubyが基本文法上最も参考にした言語であるEiffelもelsifを採用していた。

予約語一つにも歴史と理由があるものですね。

さらに余談ですが、Perlの文法はCとほぼ同じですが、プレースが省略できないため、図3の厘屈によりelse ifは許されません。しかし、文法に明示的にelseとifの組み合わせを組み込めば、else ifを許すことは可能です。もうずいぶん昔の話になりますが、ある日、ふと思い立ってPerlのソースコードをいじってみました。yacc記述をちょっと変更するわずか数分の作業でelse ifを許すPerlを作ることができました。今でもなぜPerlコミュニティの人たちがそうしないのか不思議ではありません。

実装開始

基本的な方針が決まり、文法の方向性も決まったら実装に入ります。幸い、以前にお遊びで作ったおもちゃ言語のソースが残っていたので、それをベースに開発することになりました。

開発を開始したのが1993年の2月、それから構文解析器とランタイムの基礎部分が一通り出来上がって、最初のRubyプログラム（Hello Worldでした）が動作したのが半年後の8月でした。

正直言うと、Rubyの開発の中でこの時が一番辛い時期でした。というのもプログラマという人種は、自分が書いたコードが動作する様子を見て、プログラミングの喜びを感じるものです。この時期はRubyとして動作するために必要なものが何もないので、書いても書いても動作する状態にならないのです。この時は、モチベーションの維持が大変でした。

構文解析器を書いても、できるのはせいぜい文法チェックだけです。いざプログラムを動作させようと思っても、"Hello World"は文字列オブジェクトなので、文字列クラスが必要です。文字列クラスのためにはObjectを頂点としたオブジェクトシステムが必要です。出力させるにはIOを管理するオブジェクトが必要で、というように必要なものが芽づる式に増えていきます。プログラマ三大美德の一つ、短気を十分に備えた私がこの半年を耐え忍ぶことができたのは、奇跡のようなものです。

人気は細部に宿る

しかし、Hello Worldが出力できたからといって、それだけでは言語は全く使いものになりません。ここまででは教科書のサンプルレベルです。人気の言語を目指すには、むしろここからが本番です。

どのように言語を特徴付けるのか、人気を集めるにはどうするのか、Rubyの初期のデザインでどのようなことを考えたのか、語ることはまだまだあります。

しかし年寄りの昔話が過ぎたようで、今回は紙面が尽きてしまいました。1-5では続いて、Rubyデザインのケーススタディーの後半をお送りしようと思います。お楽しみに。

タイムマシンコラム

広く使われた言語の歴史を知ろう

2014年7月号掲載分です。とうとう言語設計の話が始まりました。今回は、Rubyという言語を作り始める前の背景や歴史的経緯を語っています。「なぜ作ろうと思ったのか」とか「どんなところに苦労したか」とか「なぜこんな文法を採用したのか」という疑問に答える内容になっています。

年寄りの昔話ではありますが、実際に広く使われた言語の背景や、さまざまな設計の背後にある理由などを語れる人はあまりいませんから、この回と次の回は、本書のハイライトの一つではないかと思います。

とはいえ、昔話だけでは単なる役に立たない知識です。後進のためにも、ぜひこれらの話から教訓を引き出してもらいたいものです。とりあえず、今回の範囲から学べる教訓は、

- デザインとは決定である
- 構文のような基本的なことでも、さまざまな考慮点がある
- よく考えないと間違ったデザインをしてしまう
- よく考えても間違ってしまうことはある

ということでしょうか。

1-5 言語デザイン入門（後編）

1-4では「Rubyの誕生」について語りましたが、今回はその続きで、Ruby言語デザインで考えたことについて語ります。変数名の付け方、継承の考え方、エラー処理、イテレーターなどをどう決定したかを紹介し、そこから設計のコツを導き出します。

1-4までで基本的な文法構造は決まり、言語としての第一歩を踏んだRubyですが、これだけではどこにでもある凡庸な言語です。まだブロックを「do ~ end」で囲う、「else if」に「elsif」を使うくらいしか決めていません。ここから細部にまで文法を詰めていく必要があります。

設計ポリシー

この時点で私がRubyをどのような言語にしたかったのかというと、「オブジェクト指向言語にしたい」というような機能的な面以外では、およそ次のようなイメージでした。

- 簡易言語からの脱却
- 書きやすく、読みやすい
- 簡潔

「簡易言語からの脱却」とは、言語仕様に手抜きがない、ということです。当時、特にスクリプト言語の分野では、仕事を果たすことが最優先で、言語仕様としては手を抜いた（と見える）言語が横行していました。例えば、特に必要もないのに実装が楽だからという理由で変数名に記号が付いていたり、ユーザー定義関数と組み込み関数で呼び出し方が異なっていたり、というようなことなどです。

「書きやすく、読みやすい」というのは比較的抽象的ですが、プログラムというのは一度書いて終わりではありません。デバッグなどの過程で何度も読み、何度も書き直すものです。同じことをするのにプログラムの規模が小さい方が何をしているか理解しやすいので、簡潔に記述できることが望ましいですが、簡潔すぎるのも困ります。

世の中には異常に簡潔だが、後で見返すと何をやっているか分からないというような言語も存在します。そのような言語はしばしば「Write Once Language」と呼ばれます。書いたら書きっ放し

ということですね。後からプログラムを読解するよりも、もう一度最初から書いた方が早いような言語です。適切な「書きやすさ、読みやすさ」は結局バランスによって達成されるということでしょう。デザインというものはいつもそうですが。

また、私だけではないと思うのですが、プログラムを書いていて、やりたいことの本質と関係のないことを記述するように強制されるとき、ほんの少しですがムカつきます。開発中はそのソフトウェアが何を行うべきかという本質に集中したいのです。分かりやすさを犠牲にしない範囲で、できるだけ本質以外の部分を切り落とした簡潔な記述が望ましいと考えました。

変数名

Rubyの開発初期に参考にした言語の一つがPerlです。Perlでは変数名の先頭に記号が付きます。その意味は表1のようになっています。

興味深いのは配列アクセスなどです。配列(@foo)の0番目の要素の取り出しなのに、記号は\$です。ハッシュも同様です。これはつまり、先頭の記号はその変数(または式)の型を示しているのです。Perlは変数名によってその型を明示する静的型の言語だったのです(びっくり)。

もっとも後にリファレンスというものが導入されて、配列やハッシュを含めてあらゆるものがスカラーとして表現可能になります。このため、この静的型の原則はあまり重要ではなくなってしまうのですが。

しかし変数名を見たときに一番知りたいのは、その型ではなくてスコープです。いくつかの言語(例えばC++)のコーディングルールでは、グローバル変数やメンバー変数には特定のプリフィックスを付けるというものがあります。一方、例えば以前米Microsoft社で多用されていたハンガリアン記法のような型情報を変数名に埋め込むコーディングルールは、最近ではすっかり見かけなくなりました。型情報は明示する必要がないという証明でしょう。

そこで、Rubyでは変数名にスコープを示す記号を付けることにしました(表2)。例えば\$はグローバル変数、@はインスタンス変数です。しかし、最も多用されるローカル変数や定数(クラス名など)にまで記号を付けてしまうとPerlの二の舞いです。

変数名	意味
\$foo	スカラー (文字列 or 数値)
@foo	配列 (スカラーの配列)
%foo	ハッシュ (連想配列)
\$foo[0]	配列要素アクセス
\$foo{n}	ハッシュ要素アクセス

表1 Perlの変数名ルール

種別	記号	例
グローバル変数	\$	\$foo
インスタンス変数	@	@foo
ローカル変数	アルファベット小文字	foo
定数	アルファベット大文字	Foo

表2 Rubyの変数名ルール

ローカル変数などは簡潔に

考えた末に、ローカル変数はアルファベット小文字を、定数はアルファベット大文字を先頭に置くというルールにしました。これならば醜い記号を目にすることは少なくて済みます。また、グローバル変数を多用するとプログラム全体にあまり美しくない\$記号が散見されるため、自然により良いスタイルを推奨することにもつながりそうです。

この変数名にスコープ情報を埋め込むことのメリットは、その変数の役割に関する情報がコンパクトな形ですぐそこにあるため、いちいち宣言を探す必要がない点です。変数宣言というのは、コンパイラにその変数のスコープや型などの情報を教えるためにあるわけですが、それらは処理の本質とは無関係です。できれば、そのようなものは書きたくない、ましてや、プログラムの読解のために宣言を探したくないという理由で、このようになっていきます。同様の理由で、Rubyには変数宣言そのものがありません。変数は最初に代入した時点で誕生するので、それが宣言の代わりになります。

念のため補足しておきますが、私は宣言、特に型宣言の持つメリットを否定するものではありません。実際に実行しなくてもコンパイル時に型の不整合として多くのエラーを検出できる静的型は素晴らしいと思います。しかし、それと同時に処理の本質に集中したい、型宣言とかを書きたくないという思いも強く、現時点では動的型に傾倒している事情があります。

スクリプトにオブジェクト指向を

Rubyを設計する際にもう一つ最初から考えていたことは、この言語を「ちゃんとした」オブジェクト指向言語にしたいということでした。

当時のオブジェクト指向言語といえば、SmalltalkかC++で、大学の研究などではLisp系のオブジェクト指向言語（Flavorsとか）も使われているといった状態でした。世の中にはEiffelも存在していて海外の金融系などでは使われているという話でしたが、実際の処理系は商用のものしかなく日本での入手は困難でした。

そういう事情もあったので、オブジェクト指向プログラミングはまだまだ身近ではありませんでした。日常的なプログラミング、特にスクリプティングのテキスト処理のような規模も小さく、複雑さもさほど高くないようなジャンルでは必要だと思われていなかったようです。

その結果、当時のスクリプト言語では、オブジェクト指向機能を最初から備えていたものではありませんでした。たとえなんらかのオブジェクト指向プログラミング支援機能を持っていたとしても、後付けで一体感に欠けるものばかりでした。

しかし、高校生の頃からSmalltalkのわずかな資料を読んで、オブジェクト指向プログラミングについて「これは理想のプログラミングだ」と感じていた私です。スクリプティングの分野においてもオブジェクト指向はきっと有効であると信じていました。自分で言語を設計するのであれば、最初からオブジェクト指向に基づいてデザインしたいと感じるのはごく当然のことでした。

単一継承対多重継承

そこで悩んだのが継承機能の設計についてです。ご存知かもしれませんが、オブジェクト指向プログラミングを支援する言語機能のうち、継承には単一継承（単純継承と呼ぶこともあります）と多重継承があります。既存のクラスの機能を引き継いで、それに新しい機能を付け加えて新しいクラスを作る継承機能において、基になる既存のクラス（スーパークラスと呼びます）を一つに限定するものを単一継承、複数許すものを多重継承と呼びます。

単一継承は多重継承のサブセットですから、多重継承があれば単一継承は実現できます。多重継承はLisp系オブジェクト指向言語で発達し、C++にも（後から）取り入れられました。1993年当時はどうだったかな*¹。

一方、多重継承には単一継承にはない課題があります。単一継承の場合、クラス間の継承関係は単なる列になり、クラス階層全体は木構造になります（図1）。一方、多重継承で複数のスーパークラスの存在を許すと、クラス関係はネットワーク状になり、DAG(Directed

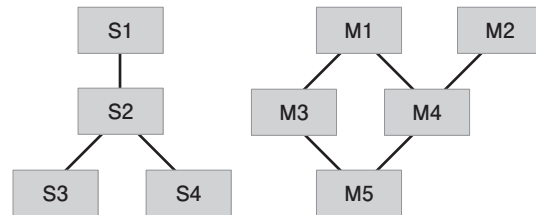
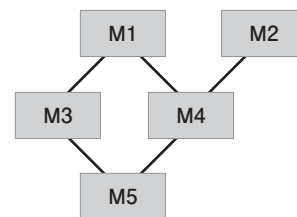


図1 単一継承（左）と多重継承（右）

Acyclic Graph、有向非再帰グラフ）と呼ぶ構造になります。多重継承では、継承したスーパークラスがさらに複数のスーパークラスを持っている場合もあります。かなり気を付けないとクラスの関係はすぐに複雑になってしまいます。

単純継承では、クラス関係が単純な列ですから、継承の優先順位について心配する必要はありません。メソッドを探すときも単純に下（サブクラス）から上（スーパークラス）へと順に探索すれば済むことです。

ところが多重継承でクラス関係がDAG構造となる場合には、探索順序は一意には定まりません（図2）。深さ優先探索もあれば、幅優先探索もあります。多くの多重継承を持つ言語では、そのいずれでもないC3探索という方法を選択するようです（CLOS、Pythonなど）。



深さ優先：M5→M3→M1→M4→M1→M2
幅優先：M5→M3→M4→M1→M2
C3：M5→M4→M3→M2→M1

図2 DAGの探索順位

しかし、いずれの探索方法を選択したとしても、直感的とは言い難いケースが存在します。そもそもこれほど複雑な継承関係は人間の理解を難しくします。

では、よりシンプルな単一継承にすれば問題はないのでしょうか。先程述べたように単一継承ではクラスの関係はシンプルで非常に理解しやすくなります。しかし単一継承にも問題がないわけではないのです。

単純継承の問題

単一継承の問題とは、継承のラインを越えてメソッドなどのクラス属性を共有する方法がないことです。共通のスーパークラスが存在しない場合、属性を共有できず、コピーするしかありません。DRY原則*によれば、コードのコピーは悪で、あまり良くない性質です。

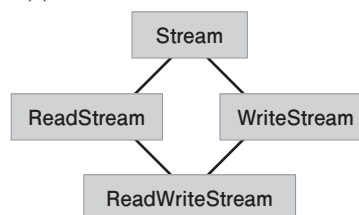
*1 『C++の設計と進化』（ソフトバンクパブリッシング刊）によれば、C++に多重継承が導入されたのは1989年のVersion2.0だそうです。1993年当時は登場したばかりで、存在していたがあまり使われていない程度の状態だったように思います。

【DRY原則】「Don't Repeat Yourself」の略で、ソフトウェア開発において繰り返しや重複を避けるという設計原則のこと。

実例を見てみましょう。Smalltalkには入出力を司るStreamクラスがあり、これには読み込みをするReadStreamと、書き込みをするWriteStreamという二つのサブクラスがあります。そして、読み書き両方ができるサブクラスReadWriteStreamというクラスも存在しています。

多重継承のできる言語であれば、ReadWriteStreamをReadStreamとWriteStreamの両方から継承することでしょう(図3(a))。多重継承にとって理想的といえるケースです。しかし、Smalltalkには多重継承がないので、ReadWriteStreamをWriteStreamのサブクラスとし、ReadStreamのコードをコピーしています(図3(b))。仮にReadStreamに仕様変更があった場合、コピーされたReadWriteStreamも同じように変更しなければなりません。さもないと、面倒なバグの原因になります。それに何よりかっこ悪い。

(a) 多重継承によるReadWriteStream



(b) SmalltalkのReadWriteStream

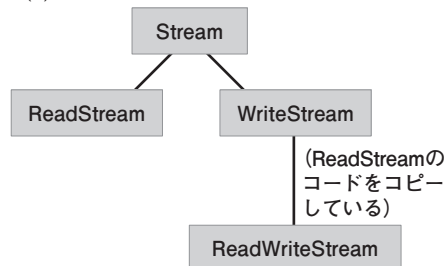


図3 ReadWriteStream

Mix-in

この問題を解決するヒントがMix-inです。Mix-inはLisp系オブジェクト指向言語Flavorsで誕生したテクニックです。Flavorsは多重継承を持つオブジェクト指向言語ですが、既に述べたような多重継承の問題を軽減するために、二つ目以降のスーパークラスに以下のような制約を課しています。

- インスタンスを持たない
- (通常の) スーパークラスを持たない。ほかのMix-inはOK

このルールに従うことにより、多重継承のネットワーク構造は、1番目のスーパークラスによる木構造に、2番目以降のスーパークラスによる枝が生えたような構造になります。これによって図3と同じ構成を実現すると図4のようになります。ストレートに多重継承を用いたものと比較するとだいぶ構成は異なりますが、単一継承のシンプルさを維持したまま、コードのコピーを排除しているのが分かります。

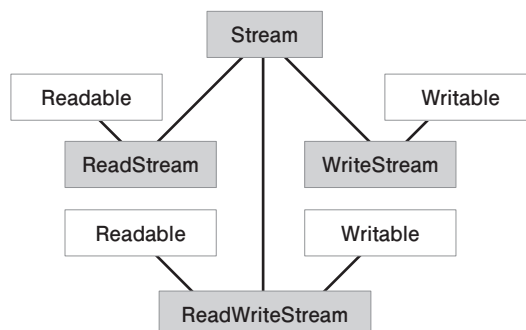


図4 Mix-in

Rubyのモジュール

このようにMix-inというのは優れたアイデアですが、あくまでも多重継承の使い方のテクニックであって、強制力はありません。そこで、言語仕様としてMix-inの使用を強制してはどうかと考えました。つまり、2種類のクラスを用意して、一つは主たる継承をする通常のクラス、もう一つはMix-inとしてだけ使える特別なクラスとします。そしてその特別なクラスは、Mix-inのルールに従って、インスタンスを作ることと、通常のクラスから継承することを禁止します。

このような発想で誕生したのがRubyのモジュールです。module文で定義されるものは、まさに前述のMix-inの性質を満たします(図4のReadableとWritableがそれに当たります)。これによって多重継承の欠点を回避しつつ、ある程度複雑さを避けられるようになりました。

このような機構はほかの言語(例えば旧Sun Microsystems社で研究されていたSelf)でもtraitとかmixinとかいう名前で提供されています。これらはRubyとほぼ同時期に、それぞれ独自に発想されたようです。

エラー処理

ソフトウェアを開発していて、一番面倒なのはエラー処理です。開こうとしたファイルが存在しなかった、ネットワーク接続が切れた、メモリーが足りなくなった、などなど、ソフトウェアが期待通り動作しない「例外的状況」はいくらでもあります。Cのような言語では、例外的状況が発生するような関数呼び出しの後では、その関数実行が正常に終了したかチェックする必要があります(図5)。

図5を見ると、わずか1行で表現できる「ファイルをオープンしたい」という意図に対して、エラー処理が煩雑になることが見てとれます。これは簡潔に意図通りに表現するというRubyの設計ポリシーに反します。Rubyではこの点についてなんとかせねばならないと感じました。

```
FILE *f = fopen(path, "r");
if (f == NULL) { // オープンが正常に終了しなかった
    switch (errno) { // エラーの詳細情報は変数errnoに格納される
        case ENOENT: // ファイルが存在しない
            ...
            break;
        case EACCES: // アクセス権エラー
            ...
            break;
        ...
    }
}
```

図5 Cのエラー処理

最初に検討したのはIconという言語のエラー処理機構でした。米テキサス大学で開発されたIconという言語は、すべての関数呼び出しは成功（値を返す）か失敗かのいずれかの結果を持ちます。失敗した関数呼び出しは、呼び出した関数の実行も失敗させます。この点はC++やJavaなどにも見られる例外処理機構に似ています。Iconに特徴的なのは、この言語ではこの失敗が真偽値としても扱われるという点です。つまり、

```
line := read()
```

は、なんらかの事情で失敗したときにはこの呼び出した関数を失敗させ、実行は中断されます。そして

```
if line := read() then  
  write(line)
```

とするとread()が成功したときにはwrite()で書き出しをして、そうでないときは何もしない、という挙動になります。さらに

```
while write(read())
```

とすると、read()が成功する間は繰り返しその戻り値をwrite()する、read()またはwrite()のいずれかが失敗すれば繰り返しをやめるという意味になります。

この仕組みは特別な構文を導入しなくても例外処理を自然に記述できるという点で魅力的でした。しかし逆に例外処理が目立たなくなってしまうのではないかという点、あまり「普通」の言語から離れてしまうと親しみにくくなるのではないかという点、さらに実行効率についての不安から不採用としました。これを採用していれば、Rubyは今とずいぶん違った印象の言語になっていたかもしれませんね。

言語デザインにおいて、人気の言語をデザインしようと思うとき、ほかの言語と大きく異なる設計を採用したくなったら、その点が言語の目玉になるかどうかを考える必要があります。そこにこだわりがあって譲る気がないなら全く問題はありません。しかし、さしてこだわりもないのに深く考えずに「変わった文法」を採用すると後々までそれが禍根になることがあります。

例外の予約語にはこだわった

さて、Icon流の例外処理は諦めたものの、Rubyになんらかの例外処理は欲しいと思っていました。そこで、C++などに見られる（Javaはまだ世に出ていませんでした）「普通」の例外処理機構を採用することにしました。ただ、予約語には少しこだわりがありました。

C++の例外処理機構では、try ~ catch構文を用いますが、私はこの予約語があまり好きではありませんでした。「try」というのは「成功するかどうか試してみよう」というニュアンスが感じられますが、あらゆるメソッド呼び出しが例外を発生する可能性がある以上、「試してみる」というのは適切ではないと感じたからです。また「catch」もあまり例外処理をイメージさせません。

そこで、構文ブロックのときにも参考にさせてもらったEiffelからrescueという単語を借りてきました。危機的状況に陥ったときに救出するというイメージは例外処理にぴったりだと思いました。

それから例外が発生してもしなくても確実に後処理を行うための構文（いくつかの言語ではfinallyという名前前で指定します）の予約語もEiffelからensureというものを借りてきました。もっともEiffelのensureは例外処理のためではなく、DBC（Design by Contract）で用いられるメソッド実行後に成立しているべき事後条件表明のために用いられていた予約語なのですが。

ブロック

最後に取り上げるのは、Rubyの最大の特徴としてしばしば取り上げられるブロックです。しかし、これは元々それほど重要視してはいなかったもののなのです。言語設計者としては意外な展開です。

MITで開発されたCLUという言語があります。CLUを一言で説明すると、オブジェクト指向言語の前段階とでもいうべき抽象データ型言語とでも呼ぶのでしょうか。

CLUにはいくつかの目立った特徴があるのですが、その一つがイテレーターです。イテレーターは「ループの抽象化を行う関数」です。

イテレーターは例えば以下のように呼び出します。

```
for i:int in times(100) do
...
end
```

これはtimesという名前のイテレーター関数を呼び出しています。イテレーター関数は、for文の中でだけ呼び出せる特殊な関数です。このtimes関数の定義をCLUで記述すると図6のようになります。

イテレーター関数の中でyieldが呼ばれると、yieldに渡された値が、for文で指定した変数に代入され、doからのブロックが実行されます。

同じようなことをCで実装しようとすれば「for文を使う」か「関数ポインタを使う」のいずれかになるでしょう。しかしfor文を使えば、ループ変数や内部構造へのアクセスなどが隠蔽できません。関数ポインタを使えば、隠蔽は可能ですが（Cにはクロージャーがないので）、変数などの受け渡

```
times=iter(last:int) yields(int)
n:int := 0
while n < last
  yield(n)
  n := n + 1
end
end times
```

図6 CLUによるtimes関数の実装

しが面倒です。

CLUのイテレーターにはそのような問題が存在しないので、ループの抽象化として大変望ましい性質を持っています。

構文で試行錯誤

ですから、これをRubyにも導入しようと考えました。しかし、しばらく考えるうちに、これをそのまま導入するのは良くないのではないかと思うようになりました。CLUのイテレーターは確かにうまくループを抽象化してくれます。しかし、この仕組みはループを越えても利用可能なのではないだろうか、CLUの構文ではループ以外の目的に利用しようとするものの障害になるのではないかと思ったのです。

元々の発想として、SmalltalkやLispには関数(Smalltalkではブロック)を引数として渡してループなどの処理をする関数やメソッドがたくさんあります。例えば、Smalltalkでは

```
[1,2,3] collect: [:a| a * 2].
```

で配列の各要素を2倍した新しい配列が得られます。仮にこのような処理をCLUの構文で記述すると

```
for a in [1,2,3].collect() do
  a * 2
end
```

とでもなるのですが、あまり直感的ではありません（これはあくまでも擬似コードで、実際のCLUではこのようなコードは書けません）。

そこでもうちょっと良い記法はないものかと考えました。この頃、ちょうど長女が生まれたばかりだったので、夜遅くなかなか寝付かない赤子をあやしながら構文について考えたものです。

最初に考えたのは

```
do [1,2,3].collect using a
  a * 2
end
```

という構文です。明らかにCLUの影響が見えますね。usingというのは、Actorという名前のPC向け言語から拝借してきました。このActorは並列実行のアクター理論とは全く関係がありません。しかし、これでもSmalltalkのブロックやLispの無名関数のような読みやすさは実現できていません。

試行錯誤のすえ、実際に実装したのは以下のような構文でした。

```
[1,2,3].collect {a| a * 2}
```

だいぶ Smalltalk 寄りですね。変数名を示す「|」が1本しかないところも Smalltalk の影響を受けています。その後、変数がないとき省略できるように「|」を2本にし、また、ほかの end で終わる構文に混じったとき違和感がないように do ~ end でもブロックを表現するように進化しました。

使い方の範囲が広がった

このように CLU を参考にしつつ、元々ループの抽象化のために誕生したブロックですが、いざ導入してみるとさまざまな使い方が登場してきました。例えば、

- ループの抽象化（もちろん）
- 条件の指定（select など）
- コールバックコードの指定（GUI など）
- スレッドや fork の実行部
- スコープの指定（DSL など）

などおよそあらゆる領域に利用されるようになりました。驚きです。

もっともブロックというのは、Lisp でも、Smalltalk でも、そのほかの関数型言語でも広く使われている高階関数（関数を引数として取る関数）の特別な記法にすぎませんから、それらの使い方ができるのはむしろ当然ではあります。しかしループの抽象化に特化したため、いくつかの制限があったにも関わらず、その制限が全く妨げにならなかったのは予想外でした。

制限というのは、

- 直接関数オブジェクトを作る構文がない（バージョン 1.9 で lambda 構文ができるまで）
- ブロック付き呼び出しが文法に組み込まれているので一つのメソッドに一つしかブロックを渡せない

というものです。実際にはこれらの制限があるために、ほとんどのケースでは読みやすく、設計しやすかったりしたようです。何が幸いになるか分からないものです。

ある調査によると、関数型言語の一つである OCaml の標準ライブラリに大量に登場する高階関数のうち、98% は一つしか関数引数を取らないのだそうです。Ruby のブロックが一つしか取れないことがさほど問題にならなかったのは、そういう理由もあったのかもしれない。

言語設計のコツ

さて、ここまで見てくると、言語設計のコツみたいところが少し分かってきたような気がします。

第一に、既存の言語にどのような問題があるのか、それを解決する方法にはどのようなアイデアがあるのかということを十分に調査することです。些細な「困ったこと」を解決することの積み重ねが「良い設計」へとつながります。

第二に、独自性の追求はここぞというところに限定することです。Iconの例外処理のところでも触れましたが、独自のアイデアは入門者のつまずきになる危険性があります。ここぞというところ以外では冒険せずに保守的な態度を守ることにも人気につながります。しかし、保守的すぎると技術的に面白くないので、ユーザーを引き付けられない危険性もあるので難しいところです。

第三に、客観的な視点です。設計はみんなで相談して決めると良いものができることはよく知られています。みんなが妥協して意見をまとめることにより、尖ったところがなくなり、尖った設計の良さが失われてしまうからです。設計について相談するにしても、あくまでも意見を聞くにとどめ、最終的な責任は一人が取る形でなければ、良い設計はできません。

私がブロックの設計のときにやったように、赤ん坊やティンバーを相手に相談するのも一つの手です。「何を馬鹿な」と思われるかもしれませんが、相手が返事してくれなくても自分の考えを説明し、相談してみることでアイデアが練られたり、より良いものが出てきたりすることは珍しくありません。

まとめ

さて1-4と1-5を通して、Rubyの設計初期に考えてきたことを紹介してきたわけですが、いかがでしたでしょうか。

些細な言語仕様一つとっても設計者がいろいろなことを調べたり、考えたりして決めていることを少しは感じていただけたでしょうか。

言語に限らず設計というものはすべてトレードオフです。完璧な設計というのは存在しなくて、ただある条件の下では「マシな」妥協点が存在するだけなのです。ただ、その妥協点をより広い範囲で役に立つ、できるだけ最適に近づけることが設計者の腕の見せどころだと思います。

しかし、言語というのは息の長いものです。なんとなく新参者の印象があるRubyでさえ、開発開始以来20年以上が経ちます。その間にコンピュータの性能は向上し、環境は変化してきました。そして、新しいトレードオフも誕生してきました。例えばRuby誕生の頃にはマルチコアコンピュータは全く一般化していませんでしたから、スレッドがマルチコアを有効活用することは求められていませんでした。しかし、現代では個人向けPCでもデュアルコア、クアッドコアが当たり前になってきました。

そのような環境の変化に応じて、言語設計であったり、実装だったりを日々見直し続けているのです。

言語設計のコツは「デザイン」一般に通じる

2014年8月号掲載分です。前回に引き続き言語デザインの裏側の話をしています。今回はRubyにおける「変数の命名ルール」や「オブジェクト指向機能の設計」「例外処理」などの設計の背景について語っています。

普通は言語というのは、「こういう風になっているものなのだ」という学び方をして、「なぜそうなっているか」という点については解説されないものです。今回はRubyの開発者として、普段あまり語らない（私は比較的語っている方ではありますが）、理由の部分に踏み込んで語ることができたのは、原稿を書いているのもとても楽しかったのを覚えています。

ところで、この回の原稿を書いた時点ではあまり明確に意識していなかったのですが、英語ではデザインも設計も両方とも「design」です。しかし、日本語ではデザインという言葉は、なんとなく設計とは違って聞こえますね。

デザインの方は「意匠を決める」感じで、設計の方は「仕組みを考える」感じでしょうか。私の仕事、というか一番得意なことはプログラミング言語がどのような機能を持ち、どのような文法を持つかを決めることです。もっぱらプログラミング言語の「見栄え」に注目していますから、最近は「言語デザイン」という言葉を使うようにしています。「言語デザイナー」って肩書はカッコいいと思いませんか。

今回の後半にある「言語設計のコツ」の部分は、結構大切なことを言っているんじゃないかと思います（自画自賛）。この原則は言語デザインのみならず、ソフトウェア設計全般にもいえることではないかなと思います。プログラミング以外の分野の経験は乏しいのですが、多分、ひいてはプログラミングの領域を超えて、設計とか仕様決定とか意思決定全般に適用できる原則なのではないかと思います。

第2章

新言語「Stream」の 設計と実装

2-1

抽象的 コンカレントプログラミング

マルチコア時代が到来し、通常のプログラミングにおいても並列プログラミングが必要とされるようになってきました。今回からは新時代の並列プログラミングについて考察します。それを支援する新言語を設計しましょう。でも、その前に並列プログラミングを支援する言語が必要とされる背景について考えてみましょう。

いまや普通の電気屋さんで売っているような「普通のPC」でもマルチコアCPUを搭載しています。それどころかスマートフォンでさえ、クアッドコア（4コア）あるいはオクタコア（8コア）のCPUを搭載した機種が珍しくなくなっています。このようにマルチコア化が進んだのには理由があります。

ムーアの法則のおかげで過去50年間、CPUをはじめとする半導体の集積度は指数関数的に向上してきました。CPUの性能もそれに伴って向上しています。しかし、しばらく前からこの状況にも終焉しゅうえんが見えてきました。CPU（コア）の性能向上が頭打ちになってきたからです。読者の皆さんも、ここしばらくCPUのクロックが2GHz周辺で推移して、以前のように劇的に向上しなくなっているのにお気づきだと思います。

マルチコア化の流れが続く

というのも近年、LSIの集積度が向上し過ぎて、配線の幅がわずか原子数個分になってしまい、電子が絶縁体の壁を乗り越えてしまう「トンネル現象」など、量子力学的な振る舞いが問題になるようになってきました。また、回路の微細化は熱密度の向上も招いています。最近のCPUコアの熱密度はホットプレートを超えています。そのうち、適切な冷却をしなければ電源を入れた途端に回路が溶けてしまうでしょう。特に複雑な処理を行うCPUコアは微細化の点でも熱密度の点でも限界が近く、もはや単一コアでの大きな性能向上は望めないのが現実です。

しかし、すべての回路の熱密度や複雑さが同じように高いわけではありません。メモリーやバスを構成する回路はCPUコアと比較するとややシンプルになっていて、熱密度などの問題は起きにくくなっています。そこで複数のコアを一つのチップに配置することで、平均的な熱密度を下げようというのが、マルチコアの背後にある動機（の一つ）です。

このハードウェア進化の傾向に近い将来覆る可能性はほとんどありません。個別のCPUコアの劇的な性能向上は望めず、新しいコンピュータの性能を最大限に活用しようと思えば、否が応でもマルチコア対応が必要になります。

並列と並行プログラミング

マルチコアを活用するには、複数の作業を同時に行う必要があります。このようなプログラミングに対して「並列」と「並行」という用語があります。

並列とは「パラレル (Parallel)」の訳で、複数の処理を同時に行うことをいいます。並行とは「コンカレント (Concurrent)」の訳で、少なくとも見かけ上は複数の処理を同時に行うことです。分かりますか？

つまり仮にCPUが一つしかなかったとしても、複数の処理を細切れに分割して、入れ替わりに実行し、見かけ上同時実行しているように見えたら、それは並行プログラミングになります。しかし実際にはCPUが一つしかなければ1度には一つの処理しか実行できないので、並列プログラミングにはなりません。マルチコア環境ではCPUが複数あるので、適切に処理を複数コアに分担させることができれば、並列プログラミングになります。

紛らわしいのですが、並列にはパラレルの「レ」が含まれており、並行にはコンカレントの「コ」が含まれているという覚え方が(私にとっては)効果的です。とはいえ、あまりに間違えやすいので、以後はパラレルとコンカレントと表記します。

ほとんどのソフトウェア開発者にとって重要なのはコンカレントプログラミングです。つまりコンカレントなソフトウェアを開発しておけば、OSや実行環境がCPUの数に応じて、一つしかなければ見かけ上の同時実行に、複数あれば真の同時実行に切り替えてくれることが多いからです。OSや実行環境の開発者だけがパラレルとコンカレントの違いに注意すればよいわけです。

それでは、これまでコンカレントプログラミングを支援するために考えられてきた仕組みを概観しましょう。

コンカレントプログラミングを支援する機構のうち、代表的なものはプロセスとスレッドです。ほとんどの現代的なOSではこの二つを提供しています。

■ プロセス

コンカレントプログラミングの支援機構のうち、最も原始的な仕組みがプロセスです。歴史的には「タスク」という仕組みの方がより古いようですが、機能的な違いはそれほど大きくありませんし、Linuxプログラミングでは登場しませんから割愛します。

プロセスは「実行中のプログラム」のことを表すOSの仕組みです。現代的なOS^{*1}は、複数のプログラムを同時に動作させられます。

*1 昔のOS、例えばMS-DOSは同時に一つのプログラムしか実行できませんでした。当時は複数のプログラムを同時実行できるOSを「マルチタスクOS」と呼んでいました。最近はマルチタスクが当たり前になり、すっかりそんな用語も使われなくなりましたが。

UNIXのforkは“複製”する

新しいプロセスを作るには、UNIXではforkというシステムコールを使います。forkシステムコールは、実行中のプログラム（これもまたプロセスです）の複製を作ります。

forkによって複製が行われると、複製元のプログラム（のプロセス、または親プロセス）ではforkが新しいプロセスのID（整数）を返します。一方、新しく複製されたプロセスではforkはゼロを返します。子プロセスは親プロセスの複製ですから、ここまでの実行結果（のほとんど）、例えば変数の値であるとか、メモリー割り当てなどは親プロセスのコピーです。しかしforkの戻り値だけが違うので、ここからの実行は分岐することになります。

実際にはほとんどの場合、子プロセスでは（必要に応じて少々の下準備をした後）、別のプログラムを起動することになります。自分のプロセスでプログラムを起動するためにはexec系システムコールを用います。exec系システムコールでは、同じプロセスで実行するプログラムがすり替わることになります。

Cのシステムコールプログラミングはやや煩雑になりますので、ここではほぼ同じことができるRubyプログラムを示します（図1）。

```
pid = fork() # Rubyの場合、子プロセスではnilが返る
if pid
  # 子プロセスの終了状態をチェックする
  Process.waitpid(pid)
else
  # echoを起動する
  exec "echo", "hello world"
end
```

図1 Rubyによるforkとexec

この複製をするforkとプロセスが実行するプログラムをすり替えるexecの組み合わせはUNIX系OSに独特なもので、ほかの多くのOS（Windowsとか）では、直接プログラムを起動するspawnというような名前のシステムコールが提供されることが多いようです。

プロセスは生成コストが大きい

このようにして起動された複数のプロセスは、OSが適当に実行時間を割り当て、少なくとも見かけ上は同時に実行されます。コンピュータが複数のCPUを持っていれば（そしてOSがマルチCPUをサポートしていれば）、OSは複数のCPUにプロセスの実行を割り当て、パラレル実行が可能になります。

プロセスの特徴は、それぞれのメモリー空間が独立していることです。forkで生成されたプロセスは元のプロセスの複製なので、子プロセスがメモリー状態を変更しても親プロセスには影響を与えません。ですから、少々子プロセスで乱暴なことをしても、親プロセスの方は安全になります。

プロセスの欠点はコストです。メモリー空間全体を複製することから、forkによるプロセス発生のコストはかなり高めです。最近のOSでは、親子でメモリーを共有し、書き換えが必要ときに初めてコピーするCoW(Copy on Write)などの仕組みを取り入れています。しかし、それでもコピーのコストは無視できません。

例えば、以前は広く使われていた動的Webページのための仕組みであるCGI*があまり使われなくなった最大の原因がこのプロセス生成のコストです。

プロセス間通信も困難

プロセスのもう一つの欠点は、プロセス間通信の困難さです。メモリー空間が分離されていることは、安全性という点からはメリットではあるのですが、複数プロセス間で情報を共有することは難しくなります。

UNIX系OSにおいてプロセス間で情報交換する手段は限られています。親子プロセスでパイプを共有することによるバイトストリームを用いた通信、ソケットを用いた通信、ファイルを経由した情報共有、そして共有メモリーです。またセマフォやシグナルによって、排他制御などのタイミングを調整することができます。

いずれの情報共有手段でも伝えられるのは、結局バイト列です。数値や配列、マップなど文字列以外のデータを送るためには、文字列に変換して送り、受け取った文字列を解釈してデータに戻す必要があります。通信データが増えると、この変換コストも馬鹿になりません。

■ スレッド

一つのプログラムの中でメモリー空間を共有しつつ、複数の制御の流れを実現するための機構がスレッドです。スレッドはプロセスと比較すると、メモリー空間の複製を伴わないので、比較的生成コストが低いという利点があります。

私がプログラミングを始めた頃、スレッドはどこでも使える機能ではありませんでした。しかし現在では広く使われるほとんどのOSでスレッド機能を利用できます。スレッドはPOSIXでも標準化されていますし、Windowsでも(APIは異なりますが)利用できます。

.....
【CGI】Common Gateway Interface。Webページのリクエストに対してプロセスを起動し、そのプロセスの出力をブラウザに転送することで、動的なWebページを提供する仕組み。

通信コストが低い

スレッドの特徴はなんといってもメモリー空間を共有しているので、スレッド間の通信コストが低い点です。文字列や整数だろうが構造体であろうが、どんなに複雑なデータ構造であってもコストなしでアクセスできます。

しかし、メモリー空間の共有はうれしいことばかりではありません。複数の処理が同時に動くということは、データがいつの間にか変化している可能性があります。気が付かないうちにデータが破壊されていることなどもあり得ます。

データの整合性が維持できないような問題も発生し得ます。例えば図2のプログラムは、aから1000を引いて1000を足しているわけですから、結果は元通りの5000になりそうですが、場合によってはそれ以外の値になることがあります。つまり「a = a + 1000」の部分でaの値を取り出してから新しい値を代入するまでの微妙なタイミングで、別のスレッドがaの値を書き換えると、期待した結果とは異なる値が得られます（図3）。

```
a = 5000
th = Thread.fork{
  a = a + 1000
}
a = a - 1000
th.join
puts "a=", a    # 結果は？
```

図2 問題のあるスレッドプログラム

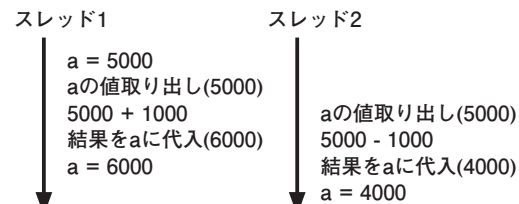


図3 スレッドによる整合性の破壊

排他制御が煩雑になる

この問題を回避するためには、整合性を維持しなければならない領域の同時実行を避ける「排他的制御」を行わなければなりません（図4）。複数のスレッドがアクセスする可能性のあるデータ（この例の場合は変数a）を変更する前にロックをかけて、このデータにアクセスするスレッドが一つしかないことを保証します。

既に述べたように、スレッドは複数の制御の流れが同時に動作しますので、同じプログラムを複数回実行しても、タイミングによって問題が発生したりしなかったりします。このような再現性の乏しいバグを俗に「ハイズンバグ^{*2}」といいます。私自身も経験がありますが、この手のバグは大変発見しづらく、遭遇すると泣けてきます。

```
m = Mutex.new
a = 5000
th = Thread.fork{
  m.lock
  a = a + 1000
  m.unlock
}
m.lock
a = a - 1000
m.unlock
th.join
puts "a=", a    # 結果は
5000
```

図4 排他的制御による整合性の維持

*2 ハイゼンバグの名前は、量子力学における「不確定性原理」を提唱した物理学者ハイゼンベルグから付けられました。

スレッドの欠点、または難しい点はこればかりではありません。メモリー空間の複製を伴わないのでスレッドはプロセスよりも生成コストが低いと言いましたが、それでもスレッド生成のためにはシステムコールの呼び出しが必要です。

システムコールの呼び出しには、特権命令が実行できるカーネル空間への切り替えが必要です。CPU命令数にすると膨大な数が実行されます。スレッド生成の頻度が高いと、このコストは無視できません。さらに1スレッドを生成するたびにスタック領域として数Mバイトのメモリーを割り当てます。1000個もスレッドを生成するとそれだけで1Gバイトもメモリーを消費することになります。

■ 理想のコンカレントプログラミング

OSが提供するコンカレントプログラミングの仕組みであるプロセスもスレッドもコスト的には理想ではありませんでした。さらに言えば、これらはあまりにも直接的で排他制御など気にしなければならないことも多すぎます。

そこで、もうちょっと高度なコンカレントプログラミングの仕組みがいろいろと模索されてきました。これまでに考案されてきたものの一部を紹介してみましょう。

アクター

アクターは1973年頃、米MIT（マサチューセッツ工科大学）のCarl Hewitt氏が考案した計算モデルです。アクターモデルでは、すべてのオブジェクトはアクターという独自の制御の流れを持った存在です。

アクターは非同期のメッセージをやり取りできます。非同期ということは、メッセージを送ったら送りっ放しで結果を待つことはない、ということです。メッセージに対する応答は、アクターから送り返されるメッセージとして受け取ることになります。

元々オブジェクト指向はシミュレーションのためのオブジェクトを操作するために誕生したもので、アクターモデルはさらにそれを押し進めたものだと考えられます。

しかし実際にすべてのデータをアクターで表現するプログラミング言語もいくつか設計されましたが、広く使われるには至りませんでした。アクターベースの言語といえば、例えば東京大学で開発されたABCL/1とかですが、実用的には使われなかったようです。

Erlangの“プロセス”

直接アクターモデルを実現した言語はあまり広まりませんでしたが、その影響を受けた言語はそれなりにあります。その代表的なものはErlangでしょう。

1986年に誕生したErlangは信頼性の高い分散コンカレントプログラミングを支援するための言語です。スウェーデンEricsson社の電話交換機などのソフトウェアを開発する目的で開発されたの

だそうです。

Erlangは関数型言語でオブジェクトを持ちませんが、その代わりに「プロセス」を持ちます。Erlangの設計者Joe Armstrong氏によれば「スレッドと違いメモリーを共有しないのでプロセスと呼ぶ」のだそうですが、正直な印象を述べると、OSのプロセスと大変紛らわしいのでやめてほしかったところでは。

ErlangのプロセスはOSのプロセスと比較すると大変軽量で、1プロセス当たり数百バイトしか消費しません。さらに生成はユーザーレベルで行われ、システムコールの呼び出しも伴いませんから、時間的にもコストが安いです。

このプロセスがアクターモデルにおけるアクターに相当します。Erlang処理系はコンピュータの持つCPUコア数に応じてスレッドを生成し、各Erlangプロセスの実行はそれらのスレッドに割り当てます。ですから、マルチコア環境では複数CPUコアを最大限に活用できるようになっています。

データ共有がシンプル

Erlangのプロセスにはメッセージを送れます。メッセージ送信は一方方向で非同期です。つまり送ったら送りっ放しで、結果を待つことはありません。結果が欲しい場合には、メッセージの中に送信元のプロセスIDを含めておき、結果をメッセージとして送り返してもらうことになります(図5)。

Erlangは関数型プログラミング言語で、ほとんどのデータ構造が書き換え不可(immutable)です。また、Erlangのプロセス間ではメッセージを介する以外のデータ共有手段はほとんどありません(実は、別の情報共有手段として組み込みのデータベースがあります)。その結果、スレッドのところで紹介したような問題はErlangではあまり発生しません。

このようなメッセージ送信のプログラムでしばしば問題となり得るのは、プログラムのバグでメッセージが送られず、全体の実行が停止してしまうことです。しかし高信頼性をうたうErlangでは、メッセージ受信がタイムアウトしたときの処理を比較的簡単に記述できたり、プロセスの異常終了を検知して再起動させたりするなどのエラー対応処理の記述にも優れています。

```
-module(pingpong).
-export([start/0, ping/2, pong/0]).

% Nがゼロのときだけこっちが呼ばれる
% Pongにfinishedメッセージを送る
ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

% Nがゼロのときだけこっちが呼ばれる
% 「Pong_PID ! {ping, self()}}」はメッセージ送信
```

```
% receiveでメッセージ受信
% pongメッセージを受け取ってループ
% Erlangには明示的ループがないので再帰する
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

% Pongの実装
% finishedで終了
% pingでpongを送り返してループ
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

% ここから始まる
% spawnで二つのプロセスを作りpingpongさせる
start() ->
    Pong_PID = spawn(pingpong, pong, []),
    spawn(pingpong, ping, [3, Pong_PID]).
```

図5 Erlangのメッセージ送信

Goのgoroutine

Erlangと似たようなメッセージ通信ベースのコンカレントプログラミングを提供しているのが米Google社が開発したGo言語です。Goの場合にはプロセスの代わりにgoroutine（ゴルーチン）と呼びます。名前が紛らわしくないのは大変良いことです。goroutineはErlangプロセスと同様に、メモリー消費量的にも、実行時間的にも生成時のコストが小さく、一つのプロセスの中で大量に生成できます。また実行がCPU数に応じた複数スレッドに割り振られ、マルチコアが活用できる点も同様です。

Goではgo文で新しいgoroutineを作ります。また、Erlangではプロセス自身がメッセージの送

信先になりましたが、Goではメッセージ送信の対象はchan（チャンネル）というオブジェクトです。ですから、Erlangにおける

- spawnしてプロセスを作る
- プロセスにメッセージを送る
- receiveでメッセージを受け取る

という処理は

- あらかじめchanを作る
- chanを渡してgoroutineを作る
- chanにメッセージを送る
- selectでchanからメッセージを受け取る

という処理になります。Erlangと違って明示的にメッセージ通信のチャンネルを渡さないといけない点と、逆にgoroutineのIDを取ることができない点が特徴的です。Erlangのpingpongをgoに直したものを図6に示します。私がGoにあまり慣れていないので、図6のプログラムはGoにおける慣習に反しているかもしれません。

正直なところ簡潔さではErlangの方が上ですね。Goでは、チャンネルを明示的に生成したり、渡したりしないといけない点は少々面倒です。しかしGoには静的な型があるので、メッセージとしてどのようなデータを渡すのか明示する必要があることや、複数のチャンネルを使い分けるようなより柔軟な設計を目指したのではないかと推測します。

```
package main
import "fmt"

func ping(n int, ping, pong chan int) {
    for {
        pong <- n;
        if n == 0 {
            fmt.Println("ping finished");
            return;
        }
        <- ping
        n = n - 1;
    }
}
```

```
func pong(ping, pong, quit chan int) {
    for {
        n := <- pong
        if n == 0 {
            fmt.Println("pong finished");
            quit <- 0;
            return;
        }
        fmt.Println("pong sending ", n);
        ping <- n;
    }
}

func main() {
    pingc := make(chan int);
    pongc := make(chan int);
    quetc := make(chan int);
    go pong(pingc, pongc, quetc);
    go ping(3, pingc, pongc);
    <- quetc;
}
```

図6 Goのpingpongプログラム

ClojureのSTM

スレッドの解説のところで、共有している情報があると適切な排他制御をしないとデータが壊れる（一貫性を失う）ことを紹介しました。

これと類似の問題は複数のクライアントから更新を含むアクセスを受け付けるデータベースでも発生します。データベースではACID原則というものが存在しています。ACIDとは、Atomicity（原子性）、Consistency（一貫性）、Isolation（独立性）、Durability（持続性）の頭文字を取ったものです。

Atomicityとは、データベースに対する操作が、完了するか、何もしないかのいずれかで、中途半端な状態は許さないという性質です。もうこれ以上分割できないという意味で「原子」という言葉が使われます。

Consistencyとは、データベースの状態はいつも与えられた条件を満たすことを保証する性質です。与えられた条件を満たさないような処理は取り消されます。例えば預金口座の管理プログラムにおいて、残高は常に正であるべきという条件が与えられていたとすると、預金残高以上に引き出す操作は条件を満たさないので実行できません。

Isolationとは、Atomicityが維持されている一連の処理の中間状態をのぞき込むことができないという性質です。

Durabilityとは、Atomicityが維持されている一連の処理が完了した時点で、その結果は保存され、結果が失われないという性質です。

データベースの概念を導入

データベースでは、このAtomicityの単位をトランザクションといいます。トランザクション中ではデータを参照・更新できますが、その中間状態は外に見えず、更新結果はトランザクションが成功したときに初めて外部から参照できるようになります。なんらかの理由でトランザクション中の処理が失敗した場合には、それまでに行ってきた更新はキャンセルされます。

このデータベースで採用されているトランザクション処理を通常のプログラミングに導入したものがClojureで採用されているSTM (Software Transactional Memory) です*³。

ClojureでSTMを使ったプログラムの例を図7に示します。トランザクションはdosyncで囲まれた範囲内で、共有情報の生成はref、参照はderef、更新はref-setをそれぞれ用います。Clojureのデータ構造は基本的に書き換え不可なので、情報の更新をするためには原則的にこのトランザクションを用いることになります。

```
(define a (ref 5000))
(define th #(Thread. (fn []
  (dosync
    (ref-set a (+ (deref a) 1000))))))
(dosync
  (ref-set a (- (deref a) 1000)))
(.join th)
(dosync
  (println (str "a=" (deref a)))))
```

図7 ClojureのSTM

まとめ

今回はコンカレントプログラミングの必要性和、それぞれの言語がその支援のために導入した機構について解説しました。2-2ではこれを踏まえて、コンカレントプログラミング言語の理想について考察します。

*3 ただし、STMの結果はあくまでも一時記憶であるメモリーに反映されるものなので、十分なDurabilityはありません。そこはデータベースとは異なるところです。

マルチコア時代に求められる言語

2014年12月号掲載分です。いよいよ、コンカレントプログラミングを支援する言語のデザインが始まります。もっとも今回はコンカレントプログラミングの背景の説明だけです。

コンカレントプログラミングを支援する言語をデザインしたいという気持ちはずいぶん昔から温めてきました。元々は、Rubyにおいてもコンカレントプログラミングを支援しようとかかなり早い時期からスレッドを導入したりしています。しかし、Rubyを作り始めた1990年代のコンピュータはシングルコアが当たり前で、パラレル実行の環境について考慮する必要はありませんでした。そういう背景からRubyのスレッドはマルチコア対応を考慮しておらず、マルチコアのコンピュータが常識化した近年では、Rubyのスレッド実装に対する不満がしばしば聞かれるようになりました。

それだけでなく、スレッドを活用しようと思うと、本編で述べたような難しさが気になります。近年のマルチコア環境を最大限に活用した上で、より簡単に間違いなくコンカレントプログラミングが可能な言語へのニーズが高まっているのではないかと感じました。本編で紹介したErlangやGoもそのようなニーズへの対応を考慮しています。しかし、もうちょっと抽象度が高い別の形のコンカレントプログラミングというものがあり得るのではないかと考えたのが、ここから先で開発するStreamの発想の源になっています。

2-2

新言語「Stream」とは

2-1ではコンカレントプログラミングの基礎について解説しました。今回はマルチコアが当然になった21世紀にふさわしいコンカレント言語のあり方について考察します。そして、それに見合う形で設計した新言語「Stream」をお披露目しましょう。

マルチコア環境が一般的になった現在、シェルスクリプトの価値が（一部で）見直されています。シェルスクリプトの基本的計算モデルは複数のプロセスをパイプラインでつなぐものです。その各プロセスの実行はマルチコア対応のOS環境下では複数コアに分散されるため、自動的にマルチコアを活用した形になります。これは計算モデルを適切に選択すれば自然な形でコンカレント実行ができる良い例になっています（図1）。

実際に業務システムの根幹部分をシェルスクリプトを用いて処理するような事例も登場していると聞いています。情報の取捨選択や加工などをシェルスクリプトで行うわけですが、従来の手法に比べて変更のコストが低いなど柔軟性の高さが評価されているそうです。

今のシェルスクリプトでは厳しい

しかし、シェルスクリプトが理想的かというところとも言い切れないところがあります。

まず、以前にも説明したようにOSのプロセス生成のコストはかなり大きいため、小さい粒度でプロセスを大量に生成するシェルスクリプトは性能的に不利になります。

コストについて言えば、プロセス間をつなぐパイプラインは結局はバイト列しか送れないため、構造のあるデータを受け渡すためには、送信側でバイト列に変換し、受信側でデータに戻す必要があります。例えば、コンマで区切ったCSV（Comma Separated Values）や、JavaScript表現を用いたJSON（JavaScript Object Notation）が用いられることが多いでしょう。データをこれらの形式のバイト列に変換したり、バイト列を解釈してデータに戻すコストはばかにできません。

マルチコアは、大量のデータを処理したい、高い実行性能が欲しいという理由で活用したくなる

●パイプライン

`command1 | command2`

●パイプラインを2 CPUで実行

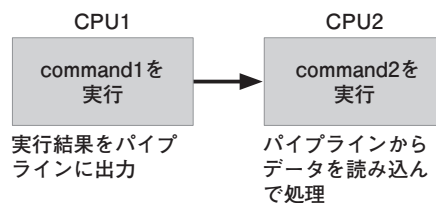


図1 コンカレントシェルスクリプト

ことが多いものです。このため、データ変換やプロセス生成のコストは無視できないことが予想されます。この点はシェルスクリプトの欠点といってもよいでしょう。

さらに言うと、パイプラインを構成するプロセスになるコマンドは、バラバラに開発されたものです。それぞれのコマンドのオプションの指定方法などの使い方に一貫性がないことが多く、使いこなす難易度が上がります。

21世紀のシェルスクリプト

となると、シェルスクリプトの良さと、いわゆる汎用プログラミング言語の良さを組み合わせることができれば、最強の言語ができそうな予感があります。

そのために最強言語が備えるべき必要な条件を考えてみましょう。

第1の条件は軽量なコンカレント実行でしょう。OSレベルのプロセスもスレッドも生成コストが高いので、できるだけ作りたくありません。実装としては、一つのOSプロセスの中に、コンピュータのコア数(+ α)のスレッドをあらかじめ生成しておき、それらが交代で実行を引き受けるようなものが現実的だと思われます。このような実装はコンカレント言語として定評のあるErlangやGoでも採用されています(図2)。Erlangの「プロセス」、Goの「goroutine」に相当するものをここでは「タスク」と呼んでいます。

第2の条件はコンカレント実行における競合条件の排除です。具体的には「状態」の排除になります。つまり、変数や属性の値が変化すると状態が発生するので、それによるタイミングに依存する問題が起こる危険性があります。そこで、すべてのデータをイミュータブル(変更不能)として、タイミングバグの発生を回避します。

第3の条件は計算モデルです。2-1で解説したスレッドのようなモデルでは、表現の幅は広いですが、自由度が高過ぎて簡単に把握できないようなプログラムになってしまいます。そこでシェルの実行モデルを参考に、抽象度が高いコンカレント計算モデルを導入します。これは抽象度が高い代わりに表現の自由度は低いので、記述を工夫する必要があります。しかしその分、最終的なプログラムはデバッグしやすいものになるはずです。

■ 新 言 語 S t r e e m

では、早速このような条件を満たす言語を設計してみましょう。ストリームを計算モデルとする言

タスクは待ち行列に並ぶ



図2 コンカレント機構アーキテクチャ
順番が来たタスクはOSスレッドが並列実行。I/O待ちなどのタイミングで入れ替わる。

語なので、「Streem」という名前を付けることにします。ストリームのスペルはStreamですが、これでは「グーグラビリティ」が低すぎます。ちょっとだけでもじったStreemという名前はタイプミス以外ではほとんど使われていないようですし、幸い、streem.orgドメインも入手可能でした。

個人的な信条ではありますが、言語デザインは、まず文法からです。といっても、シェルがベースですから大した文法はありません。基本的な文法は以下のようなものです。

```
式1 | 式2 | ...
```

これを使うと標準入力から読み込んで、標準出力に書き出すcatと同じ働きをするプログラムは以下ようになります。

```
stdin | stdout
```

なんと簡単。stdinやstdoutは定数で標準入出力を表現するオブジェクトになります。Streemのプログラムの中では、stdinは標準入力を読み込んできた行（文字列）を次々と渡してくる「流れ」のように見えます。このようなデータの流れを表現するオブジェクトを「ストリーム」と呼びます。stdoutは、逆に文字列を受け付けて外の世界（標準出力）に流すストリームですね。ファイル名を指定して読み込む場合には

```
fread(path)
```

とし、名前を指定して書き込むなら

```
fwrite(path)
```

とします。これらはそれぞれ読み込み用、書き込み用のストリームを返します。

式

式には、定数、変数参照、関数呼び出し、配列式、マップ式、関数式、演算子式、if式があります。それぞれの文法を表1に示します。普通の言語の経験があれば、さほど難しくはないでしょう。

名前	文法	例
文字列定数	"文字列"	"foobar"
数値定数	数値表現	123
変数参照	識別子	FooBar
関数呼出	識別子(引数...)	square(2)
メソッド呼出	式.識別子(引数...)	ary.push(2)
配列式	[式,...]	[1,2,3]
マップ式	[式:式,...]	[1,2,3]
関数式	{変数...->文...}	{x->x+1}
演算子式	式 演算子 式	1 + 1
if式	if (式) {文..} else {文...}	if (true) {0} else {2}

表1 Streamの式

代入

Streamの代入には2種類あります。一つは通常の言語と同じような等号を用いた代入です。

```
ONE = 1
```

もう一つは「=>」記号を使った逆方向の代入です。つまり、上記の等号を使った代入を「=>」記号を使って表現すると

```
1 => ONE
```

になります。後者の代入はパイプラインの実行結果を変数に格納するときに実行の流れに沿って表現できるので便利です。

いずれの形式の代入も、状態変化を避けるために、以下のルールに従う必要があります。

- ルール1: 同じ変数に複数回代入することはできない。一つの変数への代入はスコープの中で1度だけ。
- ルール2: 対話環境トップレベルの変数についてだけは、再代入が可能。ただし、これは同じ名前の別の変数に代入したものと見なされる。

複文

Streamでは文を複数並べられます。文と文の間はセミコロン「;」または改行で区切ります。複数の文は記述した順に実行されると思って構いません。依存関係がない場合、実際の実行では並列に実行されるかもしれません。

Streamプログラム例

それではStreamのプログラム例をいくつか見てみましょう。

先ほどはcatを実装してみました、もうちょっと違った例も見てください。例題としてしばしば用いられるFizzBuzzゲームをStreamで記述してみます(図3)。これは参加者が1から順に数字を言っていきますが、3で割り切れるときは「Fizz」、5で割り切れるときは「Buzz」、両方で割り切れるときは「FizzBuzz」と答えるというゲームです。

seqという関数は1から指定された数までの数列を作ります。パイプラインに関数が指定されると列の各要素に対して適用され、その結果が次のパイプラインに渡されます。stdoutは受け

```
seq(100) | map{x->
  if (x%15==0)
    "FizzBuzz"
  else if (x%3==0)
    "Fizz"
  else if (x%5==0)
    "Buzz"
  else
    x
} | stdout
```

図3 StreamによるFizzBuzz

取った値（の列）を出力します。

こうして見ると Stream によるパイプライン記述は、なかなかストレートに表現できているような気がしてきませんか？

```
1対1、1対n、n対m
```

図3のような、値の列を加工して書き出すようなプログラムは Stream で簡単に記述できることが分かりました。しかしプログラムはこのような 1 対 1 の関係のものばかりではありません。例えば grep（単語検索）のように「条件に合うものを探す」ようなタイプもあれば、wc（単語カウント）のように集計をするものもあります。

Stream にはそのような場合に用いる予約語がいくつかあります。

emit は 1 度の実行で複数の値を返すときに使います。複数の値が渡されたら複数の値を返すことになります。つまり、

```
emit 1, 2
```

は

```
emit 1; emit 2
```

と同じ意味です。それから配列の前に「*」を付けることで、配列要素を 1 度に返せます。つまり、

```
a = [1,2,3]; emit *a
```

は

```
emit 1; emit 2; emit 3
```

と同じ意味になります。emit の使用例を図 4 に示します。このプログラムでは、1 から 100 までの数をそれぞれ 2 回ずつ表示します。

```
# 値を2回ずつ繰り返す
seq(100) | map{x->emit x, x} | stdout
```

図4 emitの利用例

return は関数実行を終了して戻り値を返します。return は複数の値を返すことができ、その場合には複数の値を emit したことになります。ここまで説明してきませんでしたが、関数本体の式が一つしかない場合、return がなくてもその式の値が戻り値になります。

emitやreturnを使えば、渡されたより多くの値を生成できます。逆に渡されたものよりも少ない値を生成するためにはskipを使います。skipを使うとその関数の実行は終了し、値は生成されません。skipを使った例を図5に示します。図5のプログラムでは1から100までの数から偶数を選び出しています。

```
# 奇数を選ぶ
seq(100) | map{x -> if (x % 2 == 0) {skip}; x} | stdout
```

図5 skipの利用例

イミュータブル

既に述べたようにStreamではすべてのデータ構造は、競合を避けるためイミュータブルです。配列やマップ(RubyのHash相当)もイミュータブルになります。要素の追加などは、既存のデータを改変するのではなく、元のデータはそのまま要素を追加した新しいデータを作ることになります(図6)。

```
a = [1,2,3,4] # aは4要素の配列
b = a.push(5) # bはaの末尾に5を追加した配列
              # aは変化せず
```

図6 イミュータブルデータの更新

一般的なオブジェクト指向プログラミングでは、属性(インスタンス変数など)が変化することで処理が進んでいきますが、Streamではそのようなことが許可されていないので注意が必要です。この点は関数型プログラミング言語に似ているといえるでしょう。

単語カウント

では、Streamの別のプログラム例として、MapReduceの例題として頻繁に用いられる単語カウントをStreamで記述してみましょう(図7)。

まず、図7のプログラムに初めて登場した文法を解説します。flatmap関数など呼び出しのところで、Ruby

のブロックのようなものが付加されています。Streamでは構文糖として引数リストの後ろに関数が付加されている場合、それを最後の引数として追加します。つまり、

```
flatmap{s->s.split(" ")}
```

```
stdin | flatmap{s->
  s.split(" ")
} | map{x->[x, 1]} | reduce_by_key{
  k,x,y -> x+y
} | stdout
```

図7 Streamによる単語カウントの例

という式は

```
flatmap({s->s.split(" ")})
```

という文の別形式です。これは Ruby のブロックと同様の見栄えを通常の間数呼び出しの仕組みのまま追加するための工夫です。

図7のプログラムの働きを見てみると、stdin から1行ずつ受け取り、splitで単語ごとに分割したものを、flatmapで展開します。それからmapで[単語, 1]という配列に変換し、reduce_by_keyで各単語からその登場回数へのマップを作ります。reduce_by_keyは[キー, 値]の2要素の配列のストリームを受け取り、キーごとにグループ化したストリームを返します。すでに登場したキーが再びストリームに現れたときには、引数として渡した関数は(キー, 古い値, 新しい値)の3引数で呼び出されます。そして関数の戻り値がキーに対応する新しい値になります。この例では[単語, 1]というストリームにreduce_by_keyを適用することで、最終的に[単語, 登場数]というストリームを得ています。

最後はそのマップをstdinにパイプラインでつなぐと、キーと値の組み合わせが出力されるので、単語とその登場回数が表示されることになります。今回は何もしていませんが、必要であれば出力前に単語の表示順をソートするパイプラインを追加することも可能でしょう。

ソケットプログラミング

UNIXでもストリームベースで設計されているソケットは、当然Streemで取り扱うことができます。図8のプログラムはソケットを用いたネットワークサービスで最も簡単なEchoサーバー(受け付けた入力をそのまま返す)です。

いやあ、簡単ですね。Streemでは、ストリームモデルにうまくマッチするプログラムはものすごく簡単に記述できます。

一応、プログラムを解説しておきましょう。tcp_server関数は指定したポート番号のサーバーソケットをオープンし、クライアントからの接続を待ちます。Streemではサーバーソケットは、クライアントソケットのストリームになります。

クライアントソケットはクライアントからの入出力のストリームになりますから、

```
s | s
```

とすることで、「入力してきたものをそのまま送り返す」という振る舞いになります。入力に対してなんらかの加工を行う場合には、このパイプライン間に加工をするストリームを挟み込むことになります。

```
# ポート番号8007のサービスをオープン
tcp_server(8007) | {s->
  s | s
}
```

図8 Echoサーバー

配管業務

これまで見てきたようにパイプラインの構成は

式1 | 式2... | 式n

という構成になり、式1が値を生成するストリーム（ジェネレーターと呼びましょう）であり、式2以降が値を変換・加工するストリーム（フィルター）で、末尾に来る式nが出力先（コンシューマー）になります。

ジェネレーターには、stdinのような外からの入力をストリームとして取り込むものもあれば、seq()のように計算によって値を作り出すものもあります。ジェネレーターの位置に関数式が与えられ、その関数を呼び出してreturnやemitから与えられた値を生成するジェネレーターになります。

フィルターは、多くの場合は関数で、与えられた値を引数として呼び出し、emitやreturnで与えられた値を後ろのストリームに引き渡します。

最後のコンシューマーは、受け取るばかりで値をemitしないストリームです。

Streamの基本的なプログラムはこのようなストリームをつないだパイプラインを用意して、ジェネレーターからの値を流すことで値を加工することになります。いわば配管業務といってもよいでしょう。この計算モデルはなんでもできるわけではありませんが、抽象度が高く、分かりやすくコンカレントプログラミングができるという利点があります。ある種の割り切りが分かりやすさを生む例でしょう。

しかし、あらゆるプログラムでデータの流れが一つで済むわけではありません。そのようなプログラムを一切諦めてしまうというのも割り切りが過ぎるでしょう。もうちょっと複雑な配管も必要になります。具体的には、複数のストリームを一つにまとめる（マージ）と、一つのストリームを複数に分割して同報する（ブロードキャスト）の二つが明らかに足りなさそうです。

さらに言うとストリームとストリームの間をつなぐときに、どのくらいのバッファを用意するか指定できた方がよいでしょう。

パイプラインのマージ

これまでの例のようにデータの流れが1本しかない場合には非常にシンプルで良いと思います。しかし、それですべてが解決というわけにはいきません。

時には複数のパイプラインを一つにまとめたり、あるいはパイプラインを分離したりする必要があります。パイプラインのマージには「&」記号を使います。

パイプライン1 & パイプライン2

すると、パイプライン1からの値とパイプライン2からの値を配列にまとめた新しいパイプラインができます。新たなパイプラインはマージしたいいずれかのパイプラインが終了したときに全体が終了します。例えば最初のcatの例を、行番号を追加するcat -n相当にするためには、図9のようになります。

「&」演算子は「|」演算子よりも優先順位が高いため、

```
a & b | c
```

は

```
(a & b) | c
```

と解釈されます。seq()は引数を省略した場合、1から無限に繰り返します。stdinは標準入力から1行ずつパイプラインに書き出すので、そのマージ結果は

```
[行番号, 行]
```

という配列になります。その配列をstdout（標準出力）に書き出せば、行番号付きのcatの出来上がりです。実用のためには行番号の桁そろえなどのフォーマットが必要ですが、それはstdoutの前にフォーマットするパイプラインを置けばよいことです*1。

チャンネルバッファリング

末尾がコンシューマーではないパイプラインは「チャンネル」というオブジェクトを返します。ですから、

```
seq() & stdin -> sequence
```

としたとき、このsequenceはseq()からの数列とstdinからの入力をマージしたストリームを表現するチャンネルになるわけです。パイプラインとはストリーム処理を行う「タスク」がチャンネルによって、つながれたものと考えられます。

各ストリームのデータ処理速度には当然違いが出てきます。前段のデータ生成速度が速過ぎれ

*1 ただしStreamはまだ設計上でフォーマットに関する仕様は決まっていません。

ばチャンネルに値がたまっていった、メモリーを大量に消費してしまいます。逆にチャンネルに値をためることを全くしなければ、前段の待ち時間ばかり長くなり非効率です。

そこで、チャンネルは適当な数（多過ぎず少な過ぎないのが望ましい）だけバッファリングすることになります。しかし真に適切なバッファサイズはプログラムによって決まるので、完全な推測は不可能です。場合によっては性能のため、明示的にバッファサイズを指定してやる必要があります。そのような時に役に立つのがchan()関数です。

chan()関数は明示的にチャンネルオブジェクトを生成します。パイプライン演算子「|」は右辺がチャンネルオブジェクトだった場合、そのチャンネルをそのまま出力先にします。また、chan()関数に引数として整数を渡すとそれはバッファサイズになります。つまり、図9のプログラムのバッファサイズを明示的に3に指定すると図10のようになります。

```
seq() & stdin | stdout
```

図9 cat -nの実装例

```
seq() & stdin | chan(3) | stdout
```

図10 バッファサイズ指定cat -n

バッファサイズをゼロにすると、一つ生成してはそれが消費するまで待つという、パイプラインが前後で交互に実行される形になります。シングルコア環境では便利なこともあるかもしれません。

ブロードキャスト

例えばチャットサービスを実装するときに1人から送られてきたメッセージを参加者全員に分配する必要があります。チャンネルはそのような目的にも用いることができます。chan()関数によって生成したチャンネルを複数のストリームに接続すると入力として与えられた値を接続しているすべてのストリームにブロードキャストします。

```
broadcast = chan()
# ポート番号8008のサービスをオープン
tcp_server(8008) | {s->
  broadcast | s    # みんなからのメッセージを返信
  s | broadcast    # メッセージをみんなに送信
}
```

図11 Chatサーバー

図8のEchoサーバーを、参加者全員にメッセージを配布するChatサーバーに書き換えたものを図11に示します。

勘の良い人はお気づきかもしれませんが、ブロードキャストチャンネルには状態があります。つまり、broadcastにつなげられたストリームは送信先としてbroadcast自身に記録されます。また、出力先ストリームが閉じてしまったり、あるいは明示的にdisconnectメソッドで取り除かれたときにはそのストリームは送信先ではなくなります。イミュータブルが基本のStreamですが、分かりやすいプログラミングのためには純粋さを犠牲にする必要もあるということです。もちろん、broadcastの状態変化は内部的に排他制御されているので、パラレル実行で問題が発生することはありません。

まとめ

パイプラインを計算モデルの中心に採用した自作言語 Stream を設計してみました。ストリーム処理にうまくはまるプログラムであれば驚くほど簡単に書けます。

実際には Stream 言語は設計が始まったばかりで実用レベルに到達するためにはまだまだ考えなければならないことがたくさんあります。例えば、例外処理はどうするのかとか、ユーザー定義のストリームをどうやって用意するのかとか、「オブジェクト」はどのように定義されるのかとか。ソフトウェアの規模が大きくなるにつれて言語が考慮しなければならないことも増大します。

「この言語では規模の大きなプログラムは書かないから」というのは言語設計者がしばしば用いる「言い訳」です。しかし、その言語が使いものにならなかった場合を除けば、その言い訳が役に立った試しがありません。

2-3 では引き続き Stream の設計を深めるとともに、その実装についても考えることにしましょう。

タイムマシンコラム

言語名も文法も構想時は違っていた

2015 年 1 月号掲載分です。とうとう本書の要である Stream 言語の開発が始まりました。雑誌掲載当初は Stream という言語名でしたが、ちょっと混乱を呼びそうだったので、書籍化に当たって編集してあります。

文法そのほかにも当初のものとはちょっとだけ変化しているので、そこも訂正してあります。具体的には、右代入の記号が「 \rightarrow 」から「 $=>$ 」に変化したところと、関数式の引数の部分が「 $\{x,y|x+y\}$ 」が「 $\{x,y\rightarrow x+y\}$ 」に変わっています。また、標準入出力を示す `stdin`, `stdout` の名前が大文字から小文字に変わっています。掲載時の状態のまま読んでもらうのも一興かなと思ってだいたひ葛藤したのですが、使えない文法のサンプルを見せても、歴史的資料以外には役に立たないので、修正することにしました。

この原稿を書いていた時点では当然全く言語処理系はできていなかったもので、ここで書かれているものはすべてこの時点では実装の裏付けのない構想に過ぎませんでした。ただし、矛盾していたり、実装できないような文法になることを避けるために、最低限の yacc 記述だけは用意していました。これをバックアップ代わりに GitHub にアップロードしただけで、海外ニュースの Hacker News などで話題になったのは良い思い出です。

あと、最後に一つだけ注意点ですが、今回説明した構想の中には、書籍出版時でもまだ実装されていないものがあります。例えば、chat サーバーに使われている `chan()` などです。当然、今後実装する予定はあるのですが、いつまでに実装できるのか約束することはできません。

2-3

文法チェッカーをまず作る

では、いよいよ2-2で設計したストリームベース言語の実装に取りかかりましょう。ストリームモデルの妥当性を検証した後、まずは文法の妥当性を確認するため「文法チェッカー」を開発することにします。いよいよ実装が始まり、わくわくしますね。

さて、2-2でストリームをベースにした計算モデルによる並列プログラミングの記述について解説しました。通常のコンカレント処理の記述であれば、取りあえず十分な記述力を備えていそうです。まずはその辺りの妥当性について、もう少し検討してみましょう。

タスク構成パターン

メッセージをベースにしたコンカレント処理にはタスク（またはスレッドやプロセス）の構成について、ある種のパターンがあります。これらのパターンを用いると、タスク構成がこんがらかることがありません。典型的なパターンは以下のものです。

- 供給者・消費者パターン
- ラウンドロビンパターン
- 放送パターン
- 集約パターン
- 要求・応答パターン

それでは、それぞれのパターンがどのようなものか、そしてストリームモデルではどのように表現されるのかを見てみましょう。

供給者・消費者パターン

供給者 (Producer) がデータを生成し、消費者 (Consumer) に渡すメッセージによるコンカレント処理の基本パターンです (図1)。典型例はシェルのパイプラインですが、それ以外にもあらゆるところで登場します。



図1 供給者・消費者パターン

ストリームモデルでは、供給者と消費者の間にストリームをつなぐ形で記述します。Stream 言語では供給者をP、消費者をCとした場合、

```
P | C
```

で表現します。このパターンの自然な拡張として、Cが、受け取ったデータを加工する供給者となり、次の消費者にデータを送るケースがあります。この場合には中間のCはデータを加工するのでStreamの用語的には「フィルター (Filter)」と呼びます。パイプラインとしては、

```
P | F | C
```

のようになります。もちろん、フィルターは複数つなげることができます。

ラウンドロビンパターン

供給者・消費者パターンの変形として、マルチコアを最大限活用するために、複数の消費者を用意して処理を分担させるパターンがあります。このように順番に回すことを「ラウンドロビン (Round Robin)」と呼びます (図2)。

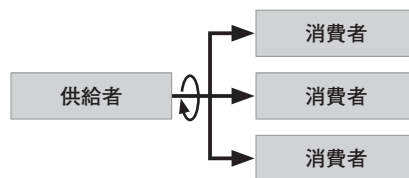


図2 ラウンドロビンパターン

ラウンドロビンの目的は負荷分散ですので、通常は同じ処理をするタスクを複数用意しておき、データ処理を順番に割り当てます。このとき何も考えず単に順番に割り当てると、前のデータ処理がなかなか終わらず、罪もないデータ処理が待たされることになるので注意が必要です。「順番に」といっても、すべてのタスクに順番に割り当ててではなく、空いているタスクに順に割り当てなどの工夫が必要です。

ラウンドロビンパターンの例としては、(マルチスレッド) Web サーバーがあります。クライアントから受け取ったリクエストをワーカーに分配するやり方は、まさにこのラウンドロビンパターンです。

Stream の場合、ラウンドロビンへの対応は言語処理系に組み込むことにします。つまり、通常の生産者・消費者パターンとして記述すれば、タスクを勝手に作り出して処理してくれるということです。具体的には

```
P | C
```

というストリームを用意すると、CPU コア数に応じた数の消費者タスクを生成して処理を進めます。コア数が増加すると、追加で何も記述しなくても性能が向上するというアイデアです。

放送パターン

同じメッセージを複数のタスクに分配する場合を放送 (broadcast) パターンと呼びます (図3)。

放送パターンの例はチャットです。1人の参加者のつぶやきが、チャットルームにいる全員に分配されます。

Streamでは一つのストリームに複数のストリームをつなぐことで放送パターンを記述できます。Pを供給者、C1..Cnを消費者とした場合、全員にメッセージを渡すためには以下のように記述します。

```
P | C1  
P | C2  
:  
P | Cn
```

あるいはすべての消費者が配列 ary に格納されているのであれば、以下ようになります。

```
ary.map{c -> P | c}
```

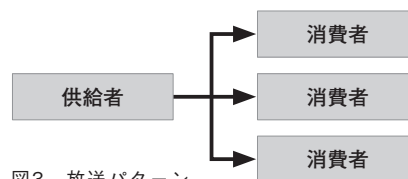


図3 放送パターン

集約パターン

一つの供給者が複数の消費者にメッセージを届ける放送パターンの逆として、複数の供給者が一つの消費者にメッセージを渡す集約パターンというものがあります (図4)。集約パターンの典型例にはログの収集があります。各所で発生したログを1カ所にまとめて保存するようなケースです。

Streamにおける集約パターンの実現は、放送パターンのちょうど逆です。P1..Pnを供給者、Cを消費者とした場合、以下のような記述になります。

```
P1 | C  
P2 | C  
:  
Pn | C
```

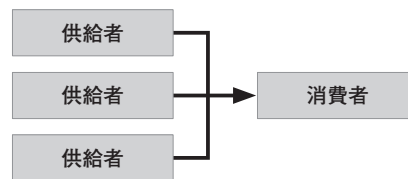


図4 集約パターン

あるいはすべての供給者が配列 ary に格納されているのであれば、以下ようになります。


```
ary.map{|p| p | C}
```

要求・応答パターン

最後は要求 (request) ・応答 (response) パターンです。これは要するにメッセージを送ったタスクから、処理済みのデータをまたメッセージとして返してもらうパターンです (図5)。

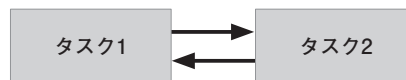


図5 要求・応答パターン

しかし正直なところ、コンカレントプログラミングにおいて要求・応答パターンの多用は推奨されません。もしメッセージを送ってその答えを待つのであれば、同一タスクの中で処理した方がメッセージ送信のコストがない分効率的です。また待つことを避けて、要求の送信と応答の受信を非同期に行うことは、プログラムの構成をいたずらに複雑にし、理解しにくいものにしてしまいます。

そのためStreamは、要求・応答パターンに対応していません。ここは割り切って、同期的な通常の関数呼び出しで対応することをお勧めします。

■ 自作言語、最初の一步

さて、Streamの背景となるストリームモデルには十分な妥当性がありそうです。それではいよいよ実装を開始しましょう。なんだかワクワクしますね。

皆さんが自作言語を開発するとして、まず何から始めますか？

もちろん、どこから実装を始めなければならないという決まりはありません。私が20年以上前にRubyの開発を始めたときには、文法の正しさをチェックしてくれる文法チェッカーから始めた記憶があります。一方、2010年にmrubyの実装を始めたときはVM (バーチャルマシン) からでした。

試行錯誤する場所から開発する

これにはそれなりの理由があります。Rubyの開発を始めた当初、どのような文法の言語にするのか、まだはっきりと決まっていませんでした。言語オタクの私にとって、言語デザインにおいて最重要視したのは文法でした。このため文法を固めるために試行錯誤することは優先事項だったのです。実際に初期のRubyは現在とは文法的にかなり異なっていて、さまざまな試行錯誤を通じて、現在の文法に「成長」しました。これが本家CRubyの開発を構文解析部からスタートした最大の理由です。

一方mrubyの方は、既存のRubyと文法的には互換であることが最初から決まっていて、文法については試行錯誤の余地がありませんでした。むしろ省メモリでありながら、ある程度の実行

効率の良いバーチャルマシンの実装が主眼でした。そこでバーチャルマシンの構成や命令セットをいろいろと試すことから始めました。当初はバイトコード(バーチャルマシンの命令列)を手書きして、そのままバーチャルマシンに与えていました。これがある程度動くようになってから、CRuby からコピーしてきた文法定義をベースに構文解析部やコード生成部を開発しました。

要するに、最も試行錯誤が必要な部分を最初に開発するのが成功のコツのようです。試行錯誤のためには、早い時期からなんらかの動作をする必要があります。CRuby の場合には、まず Ruby プログラムを与えると文法の正しさをチェックする文法チェッカーを作って試行錯誤をしました。mruby では手書きしたサンプルプログラムのバイトコード(階乗を計算するプログラムでした)を実行するバーチャルマシンをまず作りました。

Stream も文法チェッカーから

それでは、Stream をどうしましょうか。

Stream の実装で試行錯誤が必要そうなのは何か所かあります。Stream は新しい文法を持った新言語ですから、文法の良しあしはやはり気になることです。また、Stream の真髄はマルチコアに対応できる抽象度の高いコンカレントプログラミングですから、それを担うタスクスケジューラの部分も難易度が高く試行錯誤が必要そうです。

いろいろと考えたのですが、今回も CRuby のときと同様に文法チェッカーの実装から始めることにしましょう。言語オタクとしてはやはり文法の良しあしが気になります。

とはいうものの、CRuby の頃とは違って yacc の使い方にも慣れてきたので、昔よりはスムーズに開発できると思います。

まずはプロジェクトを開始します。今風のプロジェクトにありがちのやり方として、ソース管理は GitHub にお願いしましょう。まずは GitHub にログインして、空のリポジトリを作りました。今回は「matz/stream」というリポジトリを使います。

その新たに作成したリポジトリを手元にコピーします(図6)。まずは、このプロジェクトがどんなものかを示す README ファイルを用意しましょう。GitHub では Markdown フォーマットの利用が推奨されていますから、拡張子 .md を使った README.md というファイルを作成します(図7)。

ここから開発がスタートします。

```
$ git clone git@github.com:matz/stream.git
```

図6 GitHubからのリポジトリ取得

このマークで改行

```
# Stream - stream based concurrent scripting language
```

```
Stream is a concurrent scripting language based on programming model  
similar to shell, with influence from Ruby, Erlang and other  
functional programming languages.
```

```
In Stroom, simple `cat` program is like this:
```

```
    stdin | stdout
```

```
# Note
```

```
Stroom is still under design stage. It's not working yet.  Stay tuned.
```

```
# License
```

```
Stroom is distributed under MIT license.
```

```
Copyright (c) 2015 Yukihiro Matsumoto
```

図7 README.md

ソフトウェア構成

現時点での（単なる文法チェッカーである）Stroom処理系の構成は図8のようになっています。



図8 Stroom処理系のソフトウェア構成

まず字句解析器が読み込んだプログラムを「トークン」と呼ばれる記号の列に変換します。トークンとは「予約語」とか「数値」「文字列」「演算子」など「複数の文字列からなるが意味的にはつながっている固まり」に名前を付けたものです。

構文解析器は、このトークンの列を解釈し、文法に沿った記述がされていることを確認し（間違っていればエラー）、正しければ文法の意味に合わせたアクションを取ります。現時点では単なる文法チェッカーですが、将来的にはもちろん実行のために必要なプログラム構造を構築することになります。構文木やバイトコードを生成することになるでしょうね。そして、その情報を実行部（バーチャルマシンなど）に送って実際の実行をすることになるでしょう。

それでは今回開発する字句解析器と構文解析器の開発について見てみましょう。

字句解析器の開発

字句解析器の作り方にはいくつかありますが、今回は yacc のお供として字句解析器を自動生成してくれるツールである lex を用いることにしましょう（正確には GNU 拡張版である Flex）。Ruby では手書きの字句解析器を開発しましたが、いろいろ調べてみると lex でも結構なところまで達成できます。使えるのであればツールを使ったほうが信頼性が増します。将来、機能や性能で問題が起きるようであれば、その場合には Ruby のように手書きの字句解析器で置き換えるかもしれませんが、今から心配するのはムダというものです。

lexは字句の定義ファイルを用意すると、字句解析をしてくれるCの関数を自動生成してくれます。具体的にはyylex()という名前呼び出すと「次のトークン」を返してくれる関数を作ってくれます。

後述のyaccが生成する構文解析器は、トークンの取り出しのためにこのyylex()という関数を呼びますから、大変都合が良いです。字句解析部を手書きするためには、このyylex()という名前の関数を記述することになります。

文法定義の記述

まずは文法チェッカーを実装しましょう。構文解析部の実装方法にはいくつかあり、Streamくらいのサイズの文法であれば手書きの「再帰下降構文解析法」でも十分に記述できるでしょう。しかし手書きにこだわる必要もないので、yaccを使うことにします。まずは、parse.yというファイルを用意します。

これは前述のyaccというツールで処理するためのソースコードです。yacc記述の詳細を解説すると、それだけで1冊の本が書けてしまうので、ここでは詳細には立ち入りませんが、概要を理解するために必要な最低限の知識だけは説明しておきましょう。1-2でも解説しましたが、その復習も兼ねます。

lex記述の文法

lexの入力である定義ファイルは普通拡張子に「.l」を用います。lex定義の例を図9に示します。

定義ファイルは大きく分けると以下の三つ部分に分割され、それぞれの部分は「%%」で区切ります。

- 宣言部分
- 字句定義部分
- C部分

宣言部分ではlexで用いるオプションなどを宣言します。ここに「%{」から始まり「%}」で終わるコードを記述すると、生成されるCコードに直接埋め込まれます。ヘッダーのインクルードや変数、関数プロトタイプなどの宣言を記述することができます。

```
/* 宣言部分は空 */
%%
"+"          return ADD;
"-"          return SUB;
"*"          return MUL;
"/"          return DIV;
"\n"         return NL;

((([1-9][0-9]*)|0)(\.[0-9]*)? {
    double temp;
    sscanf(yytext, "%lf", &temp);
    yylval.double_value = temp;
    return NUM;
});

[ \t] ;

. {
    fprintf(stderr, "lexical error.\n");
    exit(1);
}
%%
/* C部分も空 */
```

図9 lex定義ファイル例

2番目の字句定義部分はトークンを表現する正規表現とそれに対応するアクションを記述します。

3番目のC部分では、字句解析器で用いるCの関数定義などを行います。この部分も生成されるCコードにそのまま埋め込まれます。文法定義部分のアクションで使われる関数などは、ここで定義することが多いです。

図9のファイル（仮にlex.lと名付けます）をlexにかけると「lex.yy.c」というファイルが生成されます。実際にはオリジナルのlexではなく、GNUの拡張されたlexであるflexを用います。実行手順を図10に示します。

```
$ flex calc.l
$ head lex.yy.c ← 最初の10行だけ表示
```

```
#line 3 "lex.yy.c"

#define YY_INT_ALIGNED short int

/* A lexical scanner generated by flex */

#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 5
```

図10 flex実行手順

yacc記述の文法

字句解析が終われば、今度は構文解析です。構文解析器の生成に用いるツールであるyaccの名前はYet Another Compilerの略です。

yaccは1970年代に開発されたコンパイラの構文解析部を生成するツールですが、当時このようなツールは「コンパイラコンパイラ」と呼ばれて数多く開発されていたのだそうです。その中でも後発だったyaccは、既存のものの「付け足し」というようなニュアンスで「yet another(もう一つ別の)」という名前が付けられました。ところが結局この時代に作られたコンパイラコンパイラで、現代にも生き残っているのはyaccだけというのは皮肉なものです。

「yet anotherを付けると生き残る」というのは、一種のジンクスのようなものです。Rubyのバーチャルマシンも「YARV (Yet Another Ruby VM)」と名付けられたものが生き残っていますね。

yaccの入力である定義ファイルは普通、拡張子に「.y」を用います。lexと同様に定義ファイルは大きく分けると以下の三つの部分に分割され、それぞれの部分は「%%」で区切ります。

- 宣言部分
- 文法定義部分
- C部分

宣言部分ではyaccで用いられるトークンの宣言、演算子の優先順位、ルールのデータ型などを宣言します。また、Cルーチンが用いる宣言も含めることができます。lexと同様に、宣言部分では

「%{」から始まり、「%}」で終わる部分までは生成されたCコードに直接埋め込まれます。ヘッダーのインクルードや変数、関数プロトタイプなどの宣言を記述できます。

先に3番目のC部分を説明すると、ここには構文解析器で用いるCの関数定義などを行います。この部分も生成されたCコードにそのまま埋め込まれます。文法定義部分のアクションで使われる関数などは、ここで定義することが多いです。

「BNF」で文法を定義

2番目の文法定義部分は、構文解析器が理解する文法をBNFに近いルールで記述できます。BNFとは「Backus Naur Form」の略で、BackusさんとNaurさんがAlgolという言語の文法を定義するときに発明した記法です(図11)。

図11の例は、以下のような意味になります。「数式(expr)とは

- 値
- 数式 + 値
- 数式 - 値
- 数式 * 値
- 数式 / 値

のいずれかであり、値(val)とは

- 数値
- '(' 数式 ')'

のいずれかである」です。このルールには記述されていませんが、実際は個別のルールに対応するアクションを記述できます。アクションは「{ }」で囲んだCのコードで、構文木を作ったり、コードを生成したりなど、ルールに適合した場合に実行する処理を記述します。

図11で定義されたルールに従って、

```
expr      : NUM
           | expr '+' NUM
           | expr '-' NUM
           | expr '*' NUM
           | expr '/' NUM
           ;

val        : NUM
           | '(' expr ')'
```

図11 BNF例

●正しい文法

1 + 2 + (3 * 4)

```
1 -> 数値 -> 値 -> 数式
2 -> 数値 -> 値
1 + 2 -> 数式 + 値 -> 数式
3 -> 数値 -> 値 -> 数式
4 -> 数値 -> 値
(3 * 4) -> (数式 * 値) -> (数式) -> 値
1 + 2 + (3 * 4) -> 数式 + 値 -> 数式
```

●正しくない文法

1 - * 2

```
1 -> 数値 -> 値 -> 数式
* -> 該当するルールなし (エラー！)
```

図12 BNFの解釈

```
1 + 2 + (3 * 4)
```

というプログラムを解釈すると図 12 のようになります。

yacc を実行すると `y.tab.c` というファイルを作ります。ただし、オリジナルの yacc は複数のスレッドセーフな構文解析器を作ることができないなどいくつかの制限がありますから、拡張版である GNU bison を利用します。

```
$ bison parse.y
```

と起動するだけです。実は、Ubuntu など多くの Linux ディストリビューションでは yacc を起動しても、bison が実行されます。

Stream の文法

で、とうとう Stream 言語の文法定義ですが、定義の全容は本書の誌面には収まりそうにありません。残念。

そこで、実際のコードは GitHub で見てもらうことにしましょう。Stream のソースコードは github.com/matz/stream で参照できるようにしておきます。また、今回の解説に対応するところには

```
201502
```

というタグを打っておきますので、参照してください。

ソースコード全体を説明することはできませんが、Stream の設計の基本的な方針については解説しておきましょう。

Stream は、mruby のソースコードを参考にして開発します。その結果、随所に mruby の影響が見られるでしょう。

一方、文法が複雑化している Ruby とは異なり、Stream の文法は比較的小規模に抑えています。例えば、文字列に式を埋め込むようなことはできませんし、「ヒアドキュメント」のような機能也没有。

少なくとも現段階では、Stream の設計に当たって文法に凝るよりも、シンプルに保ったまま、ストリームモデルの可能性を探る方が重要だと考えました。将来、Stream が実用レベルに進歩することがあれば、その過程において文法が強化されることはあるかもしれません。

それでは引き続き、次回に向けて Stream 処理系の開発に励むとしましょう。

まとめ

今回は、コンカレントプログラミングにおけるストリームモデルの妥当性を示すため、各種タスク構成パターンがストリームモデルでどのように表現されるかについて見てみました。

また自作言語の第一歩として、文法チェッカーの開発に取りかかりました。yaccやlexを使うことで比較的簡単に自作言語を開発することができます。

2-4では言語としての実行系について考えてみることにしましょう。また、文法チェッカーでいろいろ遊ぶうちに最初に設計したStreamの文法に不満も出てきたので、その辺りについても検討します。

タイムマシンコラム

くつがえるオープンソースの常識

2015年2月号掲載分です。文法チェッカーだけとはいえ、とうとう言語処理系の開発が始まりました。プログラマ魂が燃えますね。

ほかのところで書きましたが、ここで開発した文法チェッカーのソースコード(200行ほどの`parse.y`)をGitHubにアップロードしたら、世界中から注目を受けることになりました。スターも1000以上集まりました。まあ、その後のんびり開発していたら、ほとんどの人は飽きてしまったようですが。

私自身がフリーソフトウェアの開発に関わってからもう25年以上たちますが、Streamの立ち上がりのときに起きたことは予想をはるかに超えた事態でした。だいたい、オープンソースソフトウェアの立ち上げというものは、公開された時点のソースコードがちゃんと動作するものでなければ、人の注目を受けることができません。また逆に複雑すぎて第三者が手が出せないものはコミュニティが形成されづらいという微妙なバランスに成り立っています。

ところが、Streamと来た日には、ちゃんと動作するどころか、どんな言語かも想像しづらいような文法チェッカーだけなのにこんなに注目を集めてしまったのです。もちろん、Ruby開発者としての私の知名度のせいなのでしょうが、知名度があれば常識をひっくり返すことができるというのは驚きの知見でした。

GitHubのタグについても、少し説明しておきましょう。雑誌掲載時にその月に開発した時点でタグが打ってあります。例えば、今回は雑誌掲載が2015年2月号ですから、201502というタグが打ってあります。タグの方は書籍の状態に合わせて書き直したりしていませんので、当時のままのソースコードになっています。タグを参照しながら過去を探索する方がもしいらっしゃるのであれば、本書での解説とソースコードの状態が若干食い違っている部分もある点には注意してください。

2-4 イベントループ

引き続きStreem言語の実装を続けます。Streemは海外のニュースサイトでも紹介されるなど、正直、作者の私にも予想外の展開になってきました。ここでは、Streemの文法を少し改良した後、Streem実装のコアであるストリーム実行のプロトタイプ開発に取りかかります。

雑誌というのは、原稿を執筆してから、編集・校正・印刷などの工程を経てから書店に並びます。日経Linuxが発売されるのは、毎月8日頃ですが、執筆はそれよりはるか前になります。皆さんが日経Linuxである回を読む頃には、筆者は1カ月先の原稿を執筆していることになります。

2-3の原稿を執筆している途中で、Streemの文法チェッカーを作りました。これに簡単なREADMEを付けて「GitHub」にアップロードしたところ、目ざとい人が見つけて、Streemの話題がSNSなどで拡散されました。雑誌が発売される1カ月前の2014年末のことです。

ちゃんと動く実装でもないのに、プルリクエストなどを送ってくれる人もいて、感動しました。

まさか文法チェックしかできないプロトタイプが、Hacker Newsなどで話題になるとは思いもしませんでした。元々1月8日発売の2015年2月号の解説用コードなのでCopyright表記を2015にしていたら、目ざとくそれを指摘されたりなど、本当に予想外の展開でした。どうやら「Rubyを作ったあのMatzが新しい言語を作った」というのは、私の想像以上の話題性があったようです。

長い間オープンソース活動に関わってきて、オープンソースが盛り上がるためには、動く実装とコミュニティが必要だと考えていました。しかし、このStreemの経験によって新しい知見が得られたように思います。たとえプロトタイプでも、ちゃんと動く実装がなくても、話題性があって、話を始めることができれば、それだけでオープンソースは動き始めるものなのですね。

■ コア実装を手掛ける

さて、文法についてはだいぶ固まってきたので、今度は別の方面からアプローチしましょう。構文解析器に続くコード生成と仮想マシンの実装も気になりますが、とりあえずStreem実装のコアであるストリーム実行のプロトタイプに手を付けることにしましょう。

まず以下のようなStreemプログラムを考えます。

```
stdin | {x->x.toupper()} | stdout
```

このプログラムの実行は、以下の三つのタスクから成るパイプラインを構築します。

- 標準入力から1行読み込む
- 読み込んだ行に関数を適用
- その結果を標準出力に書き込む

最後に構築されたパイプラインの処理を実行することになります。

まずは、このパイプラインの構築と実行をするCプログラムをプロトタイプとして開発します。

プロトタイプmain

プロトタイプのCプログラムのmain関数を図1に示します。

まずパイプラインを構成する要素を表現する構造体がstrm_streamになります。main関数の先頭で今回のプロトタイプのパイプラインを構成する三つの要素をまず初期化しています。

そして、strm_connect()関数で、これら三つのstrm_streamをつないでいます。Streamプログラムを実行すると、内部ではこのようなパイプライン構築処理が行われることになります。

そして最後にstrm_loop()関数でパイプライン処理を実行します。処理が完了するとstrm_loop()の実行が終わり、プログラムが終了します。

```
int
main(int argc, char **argv)
{
    strm_stream *strm_stdin = strm_readio(0 /* stdin*/);
    strm_stream *strm_map = strm_funcmap(str_toupper);
    strm_stream *strm_stdout = strm_writeio(1 /* stdout */);

    /* stdin | {x->x.toupper()} | stdout */
    strm_connect(strm_stdin, strm_map);
    strm_connect(strm_map, strm_stdout);
    strm_loop();

    return 0;
}
```

図1 プロトタイプmain関数

パイプラインの構造

パイプラインを表現する構造体「strm_stream」の定義を図2に示します。

```
struct strm_stream {
    strm_task_mode mode; /* 生成/フィルター/消費のいずれか */
    unsigned int flags; /* フラグ */
    strm_func start_func; /* 開始関数 */
    strm_func close_func; /* 後処理関数 */
    void *data; /* ストリーム固有データ */
    strm_stream *dst; /* 出力先ストリーム */
    strm_stream *nextd; /* 出力先リンク */
};
```

図2 strm_stream

strm_streamの構造体はdstポインタでリンクされてパイプラインを構成します(図3)。一つのストリームが複数のストリームにつなげられた場合、それらのストリームがdstのリンク先のnextdフィールドから参照されます(図4)。このように複数に分岐したり、あるいは複数のストリームから一つのストリームが参照されたりすることで、ストリームのネットワークが構成されます(ほとんどの場合は直列でしょうが)。このネットワークのことをパイプラインと呼びます。

Streamのプログラムの本質は、このストリームの定義と、パイプラインの構築になります。それ以降は構築されたパイプラインにイベントが流れていく形で処理が進みます。

イベントループは独自に実装

実際のイベント処理を行うのがstrm_loop()関数です。この関数が実施すべきことは以下の通りです。

- I/Oからの入力などに応じてイベントを発生させる。

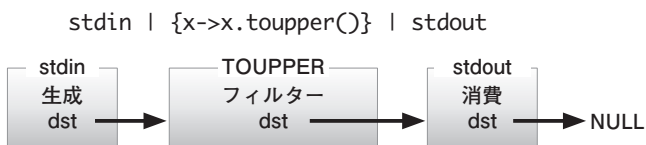


図3 パイプライン (catの例)

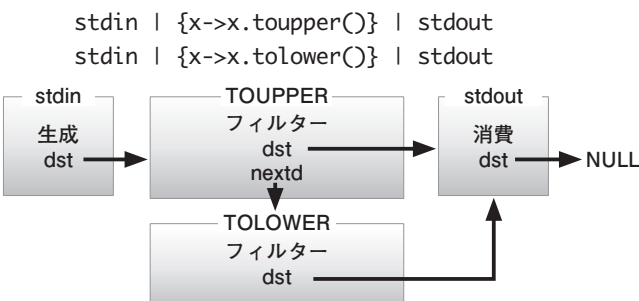


図4 複数ストリームへの結合

- 発生したイベントに対応する処理を行う。入力であればデータを読み込み行分割するなど。
- イベント処理の結果をパイプラインの次のストリームに送り、そちらの処理を行う。
- 以後、繰り返し。

このようなイベント処理を行うライブラリは世の中に既にあって、著名なものだけでも以下があります。

- libevent
- libev
- libuv

libevent はイベント処理ライブラリの老舗です。I/O などに伴うイベント処理を、コールバックを使って実現する一連のライブラリの元祖ではないでしょうか。

libev は libevent のいくつかの点を改善したライブラリです。API は異なりますが、一つのファイルディスクリプターに対する watcher の数の制限撤廃や速度改善が主要な変更点のようです。

libuv は node.js のために libev をベースに開発されたイベント処理ライブラリです。最大の違いは UNIX 以外の OS（つまり Windows）で動作する点です。また、スレッド関連の API が充実しています。

最初は Stream もこれらを使って実装しようと考えたのですが、マルチスレッドのところでつまづきました。Stream の実装ではマルチスレッドを活用したいのですが、これらのイベント処理ライブラリはマルチスレッドに対応していないのです。より正確にはスレッドを越えてイベントの受け渡しできません。また Stream では、seq() のような I/O ではないイベントも多用されるだろう点にもつまづきました。そこで、少なくとも当面は自前でイベントループを実装することにしました。

しかし、これらのつまづきは単純に私の無知からくるものである可能性は否定できません。なにしろ私自身、イベントドリブンプログラミングは本当に久しぶりで、マルチスレッドプログラミングについての経験は全くないと言っても言い過ぎではありません。その割には初期の Ruby のスレッドシステムは自分で実装していたりするのですが。

そういえば大学卒業後、新卒で入社した会社で X Window ツールキットを独自開発したときには、イベントドリブンプログラミングだったなあ。

そういうわけですから、イベント処理ライブラリに関する調査は今後も継続しようと思います。なんらかの方法で要求が満たせるのであれば、独自の実装はしない方がよいに決まっています。

I/O イベントの検出

イベントドリブンプログラミングの入出力において重要なのは「ブロック」の回避です。この場合のブロックとは、まだデータが届いていないタイミングで読み込みをすると、データが届くまでシステ

ムコールが停止することです。そこで実際の読み込みの前にファイルディスクリプターにデータが届いている（読み込んでもブロックしない）ことを知る必要があります。

そのための手段はいくつかあって、その最も古典的なものはselectシステムコールです（図5）。しかしselectには欠点があって、待ち受けられるファイルディスクリプターの数に上限があることと、n個のファイルディスクリプターのチェックにnに比例する時間がかかってしまうことです。これらの制限は近年重要視されている大量のアクセスをさばく場合に不向きです。そこで、もう少しマシなシステムコールが考案されました。

そのような「マシなシステムコール」としては、Linuxではepollシステムコールが、BSD系OSではkqueueシステムコールがあります。残念なのは、これらのシステムコールはまだ規格化されておらず、OSごとに切り替える必要があることです。実際に先に挙げたイベント処理ライブラリlibevent、libev、libuvでは内部的にこれらのシステムコールを使い分けています。

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

/* selectシステムコール
   nfds - 待ち受けする最大のファイルディスクリプター
   readfds - 読み込み待ちファイルディスクリプター集合
   writefds - 書き込み待ちファイルディスクリプター集合
   exceptfds - 例外待ちファイルディスクリプター集合
   timeout - 最大待ち時間（またはNULLでI/Oが来るまで待つ） */
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

/* fd_setの初期化 */
void FD_ZERO(fd_set *set);
/* fd_setにfdをセット */
void FD_SET(int fd, fd_set *set);
/* fd_setにfdがセットされているかチェック */
int FD_ISSET(int fd, fd_set *set);
/* fd_setの単一fdをクリア */
void FD_CLR(int fd, fd_set *set);
```

図5 selectシステムコール

epollシステムコール

今回のプロトタイプではepollシステムコールを使うことにしました。epollはLinuxでしか使えませんが、本稿は日経Linuxの連載向けですから、その点について（少なくとも執筆時点の今は）

心配する必要はないでしょう。ただしStream言語の最終実装では、epoll以外のI/Oチェックが必要になるでしょう。

epollシステムコール(実際にはepoll_create、epoll_ctl、epoll_waitの三つのシステムコール)は図6のように使います。

selectシステムコールと違って、epollでは届いたイベントの情報が構造体に渡されるので、待ち受けしているI/Oの数について考慮する必要がありません。また、epoll_ctlとepoll_waitはスレッドが異なっても動作することが保証されているので、一つのスレッドがepoll_waitでイベントループを構成し、別のスレッドから待ち受けするI/Oを登録するという構成が可能です。

```
#include <sys/epoll.h>

/* epollのためのファイルディスクリプターを作る */
int epoll_fd = epoll_create(10);

struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.ptr = data; /* このデータがepoll_waitに渡される */
/* epollに待ち受けするファイルディスクリプターfdを登録 */
/* EPOLLINは読み込み待ち */
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, fd, &ev);

struct epoll_event events[10];
for (;;) {
    /* nfds - データが届いたfdの数 */
    int nfds = epoll_wait(epoll_fd, events, 10, -1);
    for (int i=0; i<nfds; i++) {
        /* epoll_ctlに渡したdataが得られる */
        data = events[i].data.ptr;
        ...
    }
}
```

図6 epollシステムコール

イベントキュー

Streamプロトタイプのシステム構成を図7に示します。

まずは初期化の段階でストリーム(strm_stream)が生成されます。その後、ストリームが結合されてパイプラインが構成されます。I/O待ちのストリームは、ここでepollを用いてI/O待ちに登録されます。

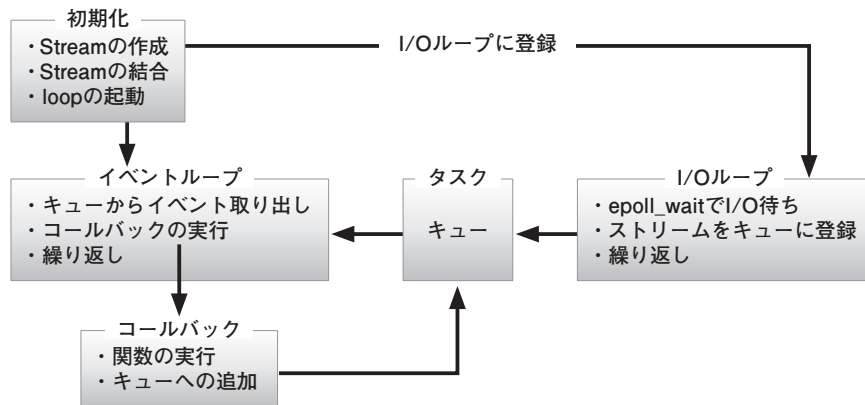


図7 Streamシステム構成

その後、I/O ループが独立したスレッドで起動されます。このスレッドでは `epoll_wait` を用いて I/O を待ち受け、データが届いたストリームをタスクキューに登録します。タスクキューに登録される情報は、実行対象のストリーム、実行内容の関数ポインタ、前のストリームから引き渡されるデータ (`void*`) の三つです。パイプラインの先頭のストリームには、引き渡されるデータがありませんので、`NULL` が渡されます。

その後、メインスレッドではイベント処理のループが走ります。メインのイベントループでは、タスクキューから一つずつ情報を取り出し、関数を実行します。

関数の中では、入力の読み込みや文字列化とか、渡されたデータの加工とか、出力とかの実際の処理が行われます。関数の中で、次のストリームにデータを渡したいときには、`strm_emit()` 関数を実行します。

`strm_emit()` 関数は三つの引数を取ります。1 番目は現在のタスクに対応する `strm_stream` 構造体（データを渡す先のストリームでない点に注意。パイプラインの構成が変化しても関数を変更する必要がないように）、2 番目は受け渡すデータ (`void*` の形)、3 番目は受け渡し後、この関数の続きを実行するコールバック関数です。

`strm_emit()` が呼び出されても、次の処理が直ちに実行されるわけではありません。渡されたデータはまずタスクキューに登録されます。関数の実行が終わるとメインループの中で次のタスクが取り出されます。

処理関数のパターン

このようなタスクキューの存在により、Stream の処理関数はあまりループを使わないで実現されます。処理関数の実装パターンは大きく分けると以下の 3 種類でしょうか。

1. コールバックによる暗黙的なループ

入力処理のような要素を生成するタイプのストリームの実装に用いられるパターンです。処理を

複数のC関数に分割し、`strm_emit()` するごとにコールバックに「続き」の関数を指定します。コールバックに同じ関数を指定することで暗黙的にループを構成します。プロトタイプのソースコードでは、`io.c` の `read_cb` と `readline_cb` がこのパターンを使っています。

2. 渡された要素をそのまま処理

渡された要素一つひとつを処理するだけで、自分自身は繰り返しをしないパターンです。`main.c` の `map_recv()` や `io.c` の `write_cb()` がこのパターンに属します。このパターンでは `strm_emit()` のコールバックに `NULL` を指定します。

3. ループ内でのemit

もちろん理屈ではループの中で `strm_emit()` することも可能です。このやり方は分かりやすいのですが、タスクがキューにたまるばかりで、関数が終了してメインループに戻るまで処理が全く進まないのをお勧めしません。

プロトタイプ のソースコード

ここで解説したプロトタイプ のソースコードは、<http://github.com/matz/streem> リポジトリの `lib` ディレクトリーに置いておきます。そのディレクトリーで `make` コマンドを実行すると `a.out` という実行ファイルができます。やっていることは、標準入力を大文字化して出力するだけです。恐らくは出版のタイムラグの関係で、最新版とは異なっているでしょうが、今回の記事時点のソースコードに `201503` タグを打っておきます。

今後の展開

まだプロトタイプを作って実験中ではありますが、次第に今後の展開が見えてきました。

まずやりたいことは、`epoll` を採用したことでLinux限定になってしまったことの解消です。`libuv` を採用するように大幅に書き換える方法や、`epoll` がない環境では、`kqueue` や `select` などを使う方法が考えられます^{*1}。

もう一つやりたいことは、マルチスレッド化です。今回の実装範囲ではI/O待ち用のスレッドが一つ、イベントキューを処理するスレッドが一つという構成にしました。マルチコアを最大限活用するためには、CPUコア数に応じたスレッドで処理を分担したいものです。

実は、今回の原稿執筆準備の途中で、複数スレッドが一つのイベントキューからイベントを取り出す実験をしたのですが、処理時間の関係で実行が前後する事態が発生してしまいました。このままでは、例えばファイルを出力すると行の長さによって順番が変わってしまいます。いくらなんでもそれはマズイので今回はイベント処理を単一スレッドで行うことにしました。恐らくはスレッドを割り

*1 原稿執筆後に `select` を使うブルリクエストが来て、WindowsやMacでも動作するようになりました。

当てる単位を個々のイベントではなく、パイプラインごとにする事でこの問題は解決できそうですが、今回は時間切れでした。

まとめ

あくまでも原稿のネタでありサンプルとして誕生した Stroom 言語ですが、予想外の注目を集めることになりました。しかし、単なるプロトタイプで終わっては意味がないので、今後も(連載とともに)実用的なレベルにまで成長させていこうと思います。雑誌連載でリアルタイムにプログラミング言語の開発過程を解説するというのは非常に珍しい試みだと思います。今後にも御期待ください。

タイムマシンコラム

コンカレントプログラミングは“難しい”

2015年3月号掲載分です。今回は Stroom 言語処理系のコアになるイベントループの部分を開発しています。

Stroom の場合、字句解析・構文解析などの言語処理を行う部分と同じくらいイベントループの実装が重要になります。イベントループの実装はなかなか難しいので、本当は既存のライブラリを使いたかったのですが、Stroom で要求されるようなマルチスレッドに対応するようなライブラリが存在しなかったので、仕方なく自作しています。今回開発したのはシングルスレッド版です。

次の 2-5 で、これをマルチスレッド化することになりますが、問題山積みです。正直なところ、本書のコンカレントプログラミングに関する部分は試行錯誤と失敗の記録にしかありません。言い訳にしかありませんが、私のような、うかつものにはコンカレントプログラミングは難解です。だからこそ、抽象度が高く、コンカレントプログラミングの難しさを隠蔽してくれる Stroom のような言語が必要なのです。しかし、誰も代わりに作ってくれないので、自分で作るしかありません。

今回はイベントループ以外にも文法を変更しています。しかし、書籍版では理解しやすいのため、段階的な文法変更の記述は本文から省いてあります。一方で、なぜ文法を変更したのかという理由の部分は言語デザインの資料として有効だと思うので、最後にコラムとして切り出しておきます。今回の変更前の文法は

●関数表現は $\{x\} \dots$ のように引数を $\{ \}$ で囲む

というものでした。これを $\{x \rightarrow \dots\}$ という形式に変更しています。また、原稿掲載時には「if 文の条件式をカッコで囲まない」という文法でしたが、これはこの先、文法の単純化のために、結局条件式をカッコで囲む文法に変更されます。

文法を改善した理由

さて文法チェッカーも完成して、Stream 言語のサンプルプログラムをいくつか書いてみたのですが、ちょっと気に入らない点が出てきました。

Stream では、Ruby から継承してブロック（無名関数）の引数を記述するのに「|」を使います。こんな感じです。

```
{ |x| x * 2 }
```

しかし Stream では、「|」記号はストリームの結合に頻繁に使われる文字です。2-3 の例でも以下のようなものがありました。

```
ary.map{|c| P | c}
```

これは混乱のもとですし、はっきり言うともう美しくありません。そこで無名関数を表現する文法をちょっと変更することにしました。新しい文法では、こうなります。

```
{ x -> x * 2 }
```

引数がないとき、あるいは省略するときには「->」ごと省略できます。

```
{ print "hello\n" }
```

これらの文法は、Groovy や Swift を参考にしました。ほんのちょっとモダンな感じがしますね。

これで、もう少し文法が美しくなります。文法が美しいかどうかは（少なくとも私にとっては）プログラミングのモチベーションに関係するので、これは良い変更です。

文法衝突が発生

しかし、この変更により、文法の衝突が発見されました。Stream では if 文などの文をまとめるのにも「{ }」を用いますが、これと無名関数が衝突したのです。正確には、関数呼び出しの後ろにブロックを付けると最後の引数として無名関数が渡されるという文法上の仕組みがあるのですが、これが衝突の原因でした。

「文法の衝突」と言っても、なんのことか分からない人もいらっしゃるかもしれません。

字句解析器によってトークンに分割されたプログラムを読み込みながら、文法ルールに当てはめていくときに、どのルールに当てはめていいのか決められなくなることを「衝突」と呼びます。

今回の場合は、

```
if foo(x) {  
    print("hello\n")  
}
```

というプログラムで、「foo(x)」まで読み込んだ次に「{」という文字が来たら、これがif文の本体を表すためか、あるいはfooという関数呼び出しに付加された無名関数なのかを判別できない事態が発生しました。

解決方法は三つ

この衝突の解決方法はいくつか考えられます。一つは、if文の条件式の部分をCのように必ずカッコでくるようにすることです。

```
if (foo(x)) {  
    print "hello\n"  
}
```

これで、関数呼び出しとif文本体は明確に区別できます。しかしモダンな言語では、if文の条件式にカッコがないのが主流になっていますし (SwiftとかGoとか)、なによりタイプ量が増えるのであまり望ましくない気がします。

別のアイデアとしては、無名関数の始まりが単なる「{」で、if文本体と区別できないことが原因なのだから、別の記号を導入して、無名関数を明示することが考えられます。例えば、無名関数をRubyのlambdaのように、「->」で始めるのはどうでしょう？ そうなれば無名関数は

```
-> x { x * 2 }
```

のようになり、関数呼び出しは

```
map-> x { x * 2 }
```

となります。無名関数の方はこれでもよいのですが、関数呼び出しの形式は、あまり制御構造っぽくありません。Rubyのブロックの美しさを失っているように思います。実は、Rubyに「->」による無名関数を導入した時点で、このようなブロック文法の導入を検討したことがあるのですが、あまり気に入らなかったので不採用とした過去があります。

結局、第3のアイデアで対応することになりました。それは、無名関数と関数呼び出しの文法は変えずに、ifの条件式の中では関数呼び出しに無名関数を付けることを許さないということです。具体的には図Aようになります。

文法ルールのコピーが発生するのは実装的に最善ではないような気もしますが、気に入った文法のために払うべきコストだと割り切ることにします。

しかし、実は2-3で解説した文法でもこの衝突は起きていたはず。気が付かなかったのはなぜでしょう。それは、私がうっかり関数にブロックを付けるルールを記述し忘れていたからでした。あらら。

```
/* 文法衝突の回避部分（抜粋） */

/* 通常の式 */
expr      : expr op_plus expr
          | expr op_minus expr
          | expr op_mult expr
          | expr op_div expr
          | expr op_mod expr
          | expr op_bar expr
          | expr op_amper expr
          | expr op_gt expr
          | expr op_ge expr
          | expr op_lt expr
          | expr op_le expr
          | expr op_eq expr
          | expr op_neq expr
          | op_plus expr           %prec '!'
          | op_minus expr          %prec '!'
          | '!' expr
          | '~' expr
          | expr op_and expr
          | expr op_or expr
          | primary
          ;
```

```

/* ifなどの条件式（ほぼexprのコピー） */
condition : condition op_plus condition
          | condition op_minus condition
          | condition op_mult condition
          | condition op_div condition
          | condition op_mod condition
          | condition op_bar condition
          | condition op_amper condition
          | condition op_gt condition
          | condition op_ge condition
          | condition op_lt condition
          | condition op_le condition
          | condition op_eq condition
          | condition op_neq condition
          | op_plus condition           %prec '!'
          | op_minus condition          %prec '!'
          | '!' condition
          | '~' condition
          | condition op_and condition
          | condition op_or condition
          | cond
          ;

/* 基本式 (primary) の共通部 */
primary0 : lit_number
         | lit_string
         | identifier
         | '(' expr ')'
         | '[' args ']'
         | '[' ']'
         | '[' map_args ']'
         | '[' ':' '}'
         | keyword_if condition '{' compstmt '}'

opt_else
    | keyword_nil
    | keyword_true
    | keyword_false
    ;

/* ブロックのない関数呼び出し */
cond : primary0
     | identifier '(' opt_args ')'
     | cond '.' identifier '(' opt_args ')'

```



```
        | cond '.' identifier
        ;

/* ブロックのある関数呼び出し */
primary  : primary0
        | block
        | identifier block
        | identifier '(' opt_args ')' opt_block
        | primary '.' identifier '(' opt_args ')'
        ' opt_block
        | primary '.' identifier opt_block
        ;
```

図A 衝突回避文法

2-5

マルチスレッドとオブジェクト

引き続きStreamのコア部分を実装していきます。今回はスレッドによるマルチコア活用にチャレンジします。タスクをできるだけ並列に処理できるようにするため、タスクを処理するワーカースレッドをCPUのコア数分だけ動かします。

2-4の最後に「マルチスレッド化したら、実行が前後する事態になった」と書きました。

マルチスレッド化

何が起きたのか、もうちょっと考えてみましょう。まず、おさらいをするとlibディレクトリーにあるプロトタイプ(a.out)は、標準入力から読み込みを行い、その文字列をすべて大文字に変換し、それを標準出力に書き込みます。

```
% echo foo | a.out
FOO
```

さて、2-4で紹介したシングルスレッド版（正確には入出力待ちをするI/Oスレッドと実際の処理をするワーカースレッドが一つずつ）のプログラム構成を図1に示します。

I/Oスレッドが入力できるデータの存在を検出したり、ワーカースレッドが次のタスクヘデータをemitしたりすると、その処理内容がタスクキューに送られます。ワーカースレッドはタスクキューから一つずつタスクを取り出し、処理を続けます。

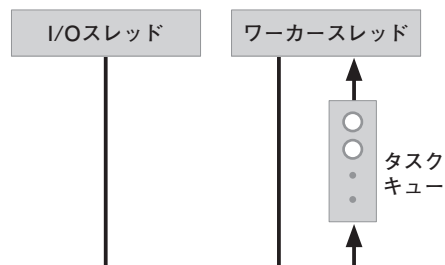


図1 シングルスレッド版

マルチコア化の試み

さて、これをマルチスレッド化することを考えます。マルチコアを最大限に活用したいので、CPUのコア数だけワーカースレッドを起動し、手が空いたワーカースレッドがタスクキューからタスクを取

り出し、処理するという構成を考えます(図2)。これまでと同様に、タスク実行中にemitが発生すると、パイプラインの続きにデータを渡すため、キューにタスクを積みます。またタスクの続きがある場合には、自分自身(の続き)もキューに追加します。

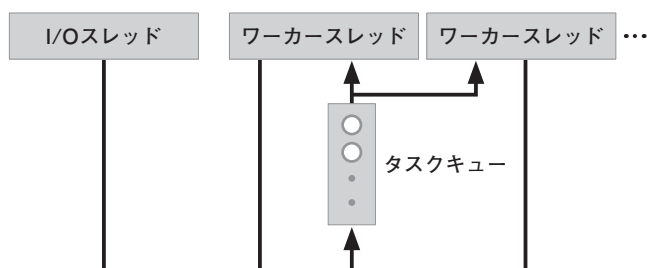


図2 マルチスレッド化 (第1版)

この構成のプログラムを起動する

と、短い入力や、標準入力にキーボードから入力しているときにはうまくいきます。しかし、ある程度長いファイルを入力にすると、おかしいことが起きます。変換結果の出力において行の順番が前後してしまうのです。これはまずい。

しばらく考えて、やっと分かりました。入力されたデータをStreamの文字列に変換したり、文字列中の文字を大文字に変換したりする処理は行の長さに応じた時間がかかります。一方、ワーカーズレッドは早いもの勝ちでキューからタスクを取り出し、処理を進めてしまいます。すると、ある程度長い文字列の処理に時間がかかっているうちに、後ろの短い行の処理が追い越してしまうのです。マルチスレッド環境で順番やタイミングが保証されないことを忘れるのは、私のようなシングルスレッド処理のプログラムばかり開発してきたプログラマがよく陥る失敗です。マルチコアは活用したいが順番が保証されないのは困ります。

マルチコア化の再挑戦

そこで順番が保証されるような構成を考えます。順番を保証するためには、いくつかの方法があり得るでしょうが、最も手っ取り早そうなのは、どのタスクがどのワーカーズレッドで実行されるのかを固定する方法です。一つのタスクを同じワーカーズレッドで実行すれば、必ず投入した順番に処理されます。

そのために、第1版と比較して以下のような変更をしました。

- タスクキューをワーカーズレッドごとに持つ。
- 初回実行時にタスクが実行するワーカーズレッドを決めてしまう。

タスクが実行されるスレッドが決まっているので、順番が前後することはなくなり、これでちゃんと動くようになりました。

さらにマルチコアを活用することを期待して、以下のようなルールを定めました。

最初のルールは、ファイル入力のような生産者タスクを最初に起動するときには待ちが最も少ないキューにタスクを追加します。これはワーカーズレッドの負荷分散を狙っています。次に、ワーカーズレッドnで実行されているタスクがemitをするとき、emitするタスクが実行されるワーカーズレッド

The diagram illustrates a multi-threaded system architecture. It features three threads: an 'I/Oスレッド' (I/O Thread), a 'ワーカースレッド' (Worker Thread), and another 'ワーカースレッド' followed by an ellipsis. The I/O thread is connected to a common bus. The worker threads are connected to a 'タスクキュー' (Task Queue) which feeds into them. Arrows indicate the flow of tasks from the queue to the worker threads.

この第2版のプログラム構成図を
図3に示します。

優先度付きキュー

パイプラインでのタスクは、生産者→フィルター→…→消費者という形で、生産者がパイプラインを開始し、そこから渡されたデータを後段のフィルターが次々と加工し、最後に消費者が使い切る形で処理されます。ここで、パイプラインの段数が長くなると、パイプラインの後ろになるほど処理が進まなくなる危険性があります。

このケースで処理は以下のように進むと考えられます。

- (1) ワーカー 1 でタスク 1 (生産者) が emit、キュー 2 にタスク 2 追加
- (2) ワーカー 2 でタスク 2 (フィルタ) が emit、キュー 1 にタスク 3 追加
- (3) ワーカー 1 でタスク 3 (消費者) が実行

116

そこで、これを解消する方法として、今回は優先度付きキューを導入しました。つまり、パイプラインの先頭である生産者タスクよりも、データを加工するフィルタータスクや消費者タスクを優先的に処理するようにします。これでキューがむやみに長くなることを避けようというアイデアです。

このため、キューの実装にちょっと手を入れました。

キューの実装

では、キューの実装を見てみましょう。Streamのキューを表現する構造体を図4に示します。

この構造体の本質は、strm_queue_task構造体のリンクリストです。キューの追加と取り出しの処理を図示すると図5のようになります。ただし、図5では基本的な処理だけ抜粋してあります。

キューの基本的構造はstrm_queue構造体のfo (first out) メンバーから連なるリンクリストです(図6)。取り出しは、foから一つタスクを取り出

```
struct strm_queue_task {
    strm_stream *strm;
    strm_func func;
    strm_value data;
    struct strm_queue_task *next;
};

struct strm_queue {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    struct strm_queue_task *fi, *hi, *fo;
};
```

図4 Streamのキュー構造体

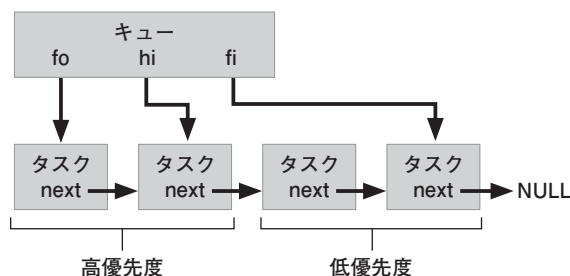


図6 リンクリストによるキュー

●追加

```
void
strm_queue_push(strm_queue *q,
                struct strm_queue_task *t)
{
    if (q->fi) {
        q->fi->next = t;
    }
    q->fi = t;
    if (!q->fo) {
        q->fo = t;
    }
}
```

●取り出し (実行)

```
int
strm_queue_exec(strm_queue *q)
{
    struct strm_queue_task *t;
    strm_stream *strm;
    strm_func func;
    strm_value data;

    t = q->fo;
    q->fo = t->next;
    if (!q->fo) {
        q->fi = NULL;
    }

    strm = t->strm;
    func = t->func;
    data = t->data;
    free(t);

    (*func)(strm, data);
    return 1;
}
```

図5 キューへの追加と取り出し (抜粋)

した上で、foが次のタスクを指すように更新します。追加はリンクリスト末尾のタスクのnextが新しいタスクを指すようにリンクします。毎回リンクリストをたぐって末尾を探すのは非効率なので、fi(first in) メンバーで末尾を指しておきます。

優先度の実装

次にキューに優先度を実装します。これはfiとfoの間を指すhi (high priority input) というポインタを用意します。優先度が低い(生産者)タスクを追加するときには、今まで通りfiにタスクを追加します。

それ以外のタスクのときには、hiが指す「高優先度タスクの末尾」にタスクを追加します。こうすると、fiからつながるリンクリスト全体としては、高優先度→低優先度の順にタスクが並ぶことになります。タスクからキューを取り出すときには、優先度を気にせず先頭から取り出せばよいことになります。

排他制御

このタスクキューにタスクを追加するスレッドとタスクを取り出し実行するスレッドは、ほとんどの場合異なるスレッドになるでしょう。

ということは、あるスレッドがキューを書き換えているときに、別のスレッドも書き換えようとする、リンクリストの整合性が壊れる危険性があります。

この危険性を回避するために、複数スレッドが同時に更新を試みる可能性があるデータ構造は実際の更新が独立に行われるように排他制御をする必要があります。このために用いられるのが、mutexです。図4のstrm_queue_task構造体にはpthread_mutex_t型のmutexというメンバーが用意してあります。

mutexの使い方は簡単です(図7)。最初に初期化します。そして排他制御をする必要がある「危険領域」をlockとunlockで囲みます。

キューの実装では、q->fi、q->hi、q->foを更新する部分が「危険領域」になりますから、これらの操作をする部分をlockとunlockで囲むことになります。これだけで、lockとunlockで囲まれた領域を実行するスレッドを一つだけに制限できます。

気を付けなくてはいけないのは、mutexは明示的な排他制御しかしらないことです。データを更新する処理をlockとunlockで囲むことを忘れてしまった場合、面倒なバグが発生することになるでしょう。

```
(1) 初期化
pthread_mutex_init(&mutex, NULL);

(2) ロック
pthread_mutex_lock(&mutex);

(3) アンロック
pthread_mutex_unlock(&q->mutex);
```

図7 pthread_mutex_tの使い方

キューが空のときの処理

もう一つ気になる点が、キューが空のときにどのように対処するか、です。実行すべきタスクがキューに存在しないときには何もすることがありません。キューにタスクが追加されるまで待つ必要があります。

一番簡単な実装は、ループしながらキューを頻繁に見に行って、「まだ来ないかな」とチェックするやり方です。このような手法を「ビジーループ」と呼びます。しかし、ビジーループはタスクが来るまでループを回り続けるわけですから、無駄にCPUパワーを消費することになります。このようなやり方は、待ち時間が短いことに確信が持てるときにしか使えません。

今回採用したのはPOSIX threadライブラリの「条件変数」という機能です。条件変数は、相互排他ロックのmutexと組み合わせて使います。例えば、図8のプログラム(a)の部分のようにmutexでロックしている間にpthread_cond_wait()を呼び出すと、ほかのスレッドで呼び出されるまで「待ち」に入ります。

(a) 条件変数での待ち方

```
pthread_mutex_lock(&mutex);
while (!fo) { // foがセットされていないかも
    pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);
```

(b) 条件変数の起こし方

```
pthread_mutex_lock(&mutex);
// 条件が成立する処理
// 条件が成立したら「起こす」
pthread_cond_signal(&cond, &mutex);
pthread_mutex_unlock(&mutex);

// 複数の待ちがあったときsignalはどれか一つ起こす
// pthread_cond_broadcastは全部一度に起こす
```

図8 条件変数の使い方

今回は、キューのタスクを実行しようとして、キューが空だったらpthread_cond_wait()を呼び出し、キューにタスクが追加されるまで待つことにしました。排他制御と条件変数による「待ち」を加えたバージョンを図9に示します。図5のものと比較してください（図9ではq->hiに対する処理も追加しています）。


```

int
strm_queue_exec(strm_queue *q)
{
    struct strm_queue_task *t;
    strm_stream *strm;
    strm_func func;
    strm_value data;

    pthread_mutex_lock(&q->mutex);
    while (!q->fo) {
        pthread_cond_wait(&q->cond, &q->mutex);
    }
    t = q->fo;
    q->fo = t->next;
    if (t == q->hi) {
        q->hi = NULL;
    }
    if (!q->fo) {
        q->fi = NULL;
    }
    pthread_mutex_unlock(&q->mutex);

    strm = t->strm;
    func = t->func;
    data = t->data;
    free(t);

    (*func)(strm, data);
    return 1;
}

```

図9 キューからのタスク実行

重要なところは、pthread_cond_wait()を囲む条件部分がifではなくwhileである点です。実は当初のコードはifでチェックをしていたのですが、pthread_cond_wait()は条件が成立していない場合にもなんらかの原因で戻ってきってしまうことがあるそうで、そのチェックが必要であると指摘されました。スレッドプログラミングは難しい。

タスクの追加についてもこれと同様で、キューのデータを操作する部分をmutexのlockとunlockで囲み、その末尾でpthread_cond_signal()を呼んでいます。

マルチスレッド処理のデバッグ

というような感じでプログラムを開発しているのですが、毎回、記事で書く内容に合わせて

Stream 処理系のどの部分を開発しようか考えていたりします。結果として「記事を書かねばならない」という使命感によって、歩みは遅いものの着実に開発が進んでいます。

今回、解説している部分にしても、記事を書きながらデバッグしてたりします。もちろん記事を書く時点でのプログラムは期待した通りに動作することを確認してから記事を書いています。しかし最初から完璧なプログラムなど書けるわけもなく、何度もデバッグを繰り返すことになります。

しかし、マルチスレッドプログラムのデバッグは大変です。今回の変更によって、Stream 処理系のマルチスレッド化が進んだので、ますます大変です。

シングルスレッドプログラムであれば gdb などのデバッガーを活用するのですが、gdb のスレッド対応は私にはあまり使いやすいとは思えませんでした。知らないだけかもしれませんが、デバッガーを活用したスレッドプログラミングのデバッグに精通の方がおられたら、ぜひご教授いただきたく。

では、どうしたかという、恥ずかしい話ではありますが、原始的な printf デバッグを活用しました。このとき注意すべき点がいくつかあります。

第一に、デバッグ出力の出力先を stderr (標準エラー出力) にする点です。これにより、プログラムの通常の出力とデバッグ出力を切り分けることができます。例えば、

```
% a.out | wc
```

とすれば、標準出力の行数と文字数、単語数を数えることができますが、デバッグ出力はコンソールに出力されます。デバッグ出力を含めてスクロールさせてみるのであれば、

```
% a.out |& lv
```

とすることで、標準出力と標準エラー出力を混在させて眺めることができます。

第二に、デバッガーで調べたいような値を出力する fprintf を仕込む必要があります。例えば構造体のメンバーがちゃんと初期化されているとか、正しい順序で処理が実行されているとかを確認することになります。このときに便利なのは fprintf の %p 指定子です。%p はポインタのアドレスを表示してくれるフォーマット指定子です。通常のアドレスは 16 進表記に、NULL のときには「(nil)」と表記してくれるのがありがたいです。

第三にコミットするときには、git diff などを利用してデバッグ出力のための fprintf がすべて消しであることを確認することです。デバッグ出力を付けたままで全世界に公開すると、ちょっと恥ずかしいことになります。

構造体でオブジェクトを表現

これまで Stream のデータは構造体や文字列を void* にキャストして渡していました。型の違いは自分で気を付ける必要があり、間違えるとプログラムがクラッシュします。

しかし、言語処理系としてはどのようなオブジェクトを指しているか型情報を管理したいところで。Streamでは図10のような構造体を使ってオブジェクトを表現することにしました。

Streamのオブジェクトを表現するstrm_valueはシンプルな構造体で、データを表現するunion（共用体）と型を表現するtypeメンバーからなります。Cのデータとstrm_valueの相互変換には図11に示す関数を用います。これらの関数の使い方については2-6以降により詳しく解説するつもりです。

オブジェクトの表現方法としては、今回採用した構造体を用いる方式以外にも、Pythonで採用している数値も含めてすべてポインタで表現する方式や、LuaJITで採用している浮動小数点数に埋め込んでしまう「NaN Boxing」方式や、Rubyで採用している整数だけはポインタに埋め込む「タグ付きポインタ」方式などがあります。これらも興味深いテクニックなので、後でしっかり解説しましょう。

```
enum strm_value_type {
    STRM_VALUE_BOOL,
    STRM_VALUE_INT,
    STRM_VALUE_FLT,
    STRM_VALUE_PTR,
};

typedef struct strm_value {
    enum strm_value_type type;
    union {
        long i;
        void *p;
        double f;
    } val;
} strm_value;
```

図10 Streamのオブジェクト表現

```
strm_value strm_ptr_value(void*);
strm_value strm_bool_value(int);
strm_value strm_int_value(long);
strm_value strm_flt_value(double);

#define strm_null_value() strm_ptr_value(NULL)

void *strm_value_ptr(strm_value);
long strm_value_int(strm_value);
int strm_value_bool(strm_value);
double strm_value_flt(strm_value);
```

図11 strm_value変換関数

ガーベージコレクション

オブジェクトが作れるようになると、今度は使わなくなったオブジェクトを回収できるようにしなくてはなりません。ガーベージコレクション（GC）が必要です。

ガーベージコレクションの実現の仕方にはいくつかありますが、今回はその中でも最も簡単な方法であるlibgcを使おうと思います。

libgcは正式名称を「Boehm-Demers-Weiser's GC」と呼ぶ、ライブラリ型のガーベージコレクターです。原則的にCやC++のプログラムにこのライブラリをリンクするだけで、mallocで割り当てたメモリー領域が使われなくなったら、勝手に回収してくれる魔法のようなライブラリです。スレッドにも対応しているのでStreamにもびったりです。

libgcを使うには、まずライブラリをインストールする必要があります。apt-getなどのパッケージマネージャーを使って、libgcパッケージをインストールします。パッケージ名はDebian系Linux(Ubuntuなど)ではlibgc-dev、Red Hat系(Fedoraなど)ではlibgc-develです。

```
% apt-get install libgc-dev
```

インストール後は、プログラム実行の先頭で

```
GC_INIT();
```

を呼び出し、表1に従ってプログラムを修正します。

calloc()とfree()については補足が必要でしょう。calloc()は割り当てられたメモリー領域をゼロクリアする関数ですが、libgcには対応する関数がありません。しかし、GC_MALLOC()は割り当てたメモリー領域がクリアされていることが保証されているので、引数の調整だけが必要になります。free()についてはガーベージコレクションを行うlibgcの性質上、本来は不要ですから基本的な対応はfree()の呼び出しを削除することになります。

ただし、なんらかの事情で強制的に解放したい場合にはGC_FREE()を用いることになります。その場合は、使用中のメモリー領域をうっかり開放してしまうというバグを導入しないように利用者が注意する必要があります。

プログラム修正後は、Makefileをいじってlibgcをリンクするようにすればお終いです。なんと簡単なことでしょう。

少なくとも当面は、ガーベージコレクションにlibgcを使うことにします。しかしlibgcも万能ではありません。ポインタの値を加工するテクニックを使った場合には正しくGCできませんし、内部的にアセンブラコードを利用している関係上、すべてのプラットフォームに対応することは困難です。

また、Streamのオブジェクトがイミュータブルである特性は、リファレンスカウント方式のGCや世代別GCの実装に有利です。この点を利用すると、汎用のGCであるlibgcよりも効率の良いGCが実装できるかもしれません。この辺の考察は将来への課題としましょう。

元の関数	libgc対応
malloc(size)	GC_MALLOC(size)
realloc(p,size)	GC_REALLOC(p,size)
calloc(n,size)	GC_MALLOC(n*size)
free(p)	GC_FREE(p) または 削る

表1 libgc修正方法

“放置”していた構文解析も前進

このように最近イベントループの開発にかかりきりだったわけですが、その間、構文解析の部分は放置していました。

しかし、私が放置していたからといって、進歩しなかったわけではありません。Streem開発初期にGoで独自に処理系（Streem）を開発された Mattn さん（Yasuhiro Matsumoto さん）を中心にプルリクエストを送ってくださる方がいて、以下のようなことができました。

- 構文木の生成
- 簡単なインタープリタ

私が主導しない状態で、密接なコミュニケーションもなしに開発がどんどん進むのは驚きです。Streem の開発は、私のオープンソースに対する見識をしばしば書き換えてくれます。

まとめ

Streem はコア実装もマルチスレッドに対応し、次第に完成度が高まってきました。しかし、まだ言語としては使いものになりません。そろそろ構文解析部とコア部を組み合わせ、言語処理系として動作させることができるようにしたいです。今後にご期待を。

ガーベージコレクションでミスがあった

2015年4月号掲載分です。イベントループによるタスク処理のマルチスレッド化を実装しています。今回はmutexを使った優先度付きキューによってマルチスレッド化を実現しています。実はこの部分はこの後ロックフリーアルゴリズムの導入などによって完全に置き換えられてしまうのですが、試行錯誤の記録としてそのまま掲載することにします。

しかし、今になって原稿を読み返すとこの頃デバッグに苦労したことを思い起こします。愚痴にしかありませんが、マルチスレッドプログラミングは本当にデバッグが大変です。バグの発生がタイミングに依存するので、どんな状態で問題が発生したのか突き止めるのが非常に困難だからです。

プログラムの状態を知るためにデバッガーを使ったり、あるいはプログラムにprintfを挿入するなどすると、実行のタイミングがずれてバグが発生しなくなったりするのはザラですから、泣きそうになります。この後もイベントループ実装にはいろいろと手を入れています、正直なことを言うと、現在でもStreamのイベントループにバグがないことを確信できていないのです。

libgcの利用についても補足しておく必要があります。雑誌掲載時には表1のようなプログラムの修正について記述してありませんでした。ものすごく古いバージョンのlibgcはmalloc()などの関数を置き換えて、libgcをリンクするだけでガーベージコレクションを可能にしていたような記憶があって、きちんとした検証もせずに原稿を書いてしまいました。

サンプルプログラムが動作していたので気づきませんでしたが、これはガーベージコレクションをしていたのではなく、メモリーを割り当てる一方で全く開放していなかったのです。サンプルプログラムでは処理するデータがあまり大きくなかったので問題が発生していなかっただけでした。恥ずかしい限りです。

2-6 キャッシュとシンボル

今回は、Streamの実装を通じてメモリアクセスの詳細について学びます。マルチコアの環境では、キャッシュの有効活用が特に重要になります。さらに言語設計を通じて、シンボルというデータ型について解説しましょう。

2-5ではマルチコアを有効に活用するため、マルチスレッドに対応したのですが、一つ気が付いたことがありました。2-5の実装では、パイプラインにタスクが並んでいるときに、複数コアを最大限活用するように各タスクを別のスレッドに割り当てていました。しかし冷静に考えると、これはあまり良くない戦略のように思えてきました。タスクの数が少ないうちはさほど影響はないのですが、タスクの数が増えてくると意外な点が実行性能に影響を与えそうなことに気が付いたのです。

今後の改善に向けて、今回はまずこの問題について考察します。

■ キャッシュの有効活用

ここで検討するのは、メモリアクセス速度の問題です。私たちは普段プログラムを書いているときに、メモリアクセスにどれくらい時間がかかるかなどは考えないことが多いでしょう。そもそも変数がメモリーに割り当てられているのか、CPUレジスタに割り当てられているのかさえ、多くの場合には関心を払いません。しかし、CPUにとってはデータを取り出すときに、どこから取り出すのかは天と地ほど違いがあるのです。

CPUはレジスタからは1クロックでデータを取り出せます。2GHzのCPUなら1クロックは0.5ナノ秒（20億分の1秒）です。ところが、同じデータがメインメモリー上にあった場合には、外部バスにアクセスして、遅いが大容量のメモリーからデータを転送する必要があります。このため、はるかに長い（何百倍もの）時間がかかります。

その間、CPUは必要なデータが手に入らないので待つ必要があります。ということは、毎回メインメモリーからデータを取り出しては、CPUは本来の速度の何百分の1の性能しか発揮できないわけです。

キャッシュで高速化

それではCPU性能がもったいないので、かなり前からCPUでは「キャッシュ」という仕組みでこの問題に対処しています。

「キャッシュ」のスペルは「cache」です。現金は「cash」でちょっと違いますね。「cache」というのは元々はフランス語で「隠し場所」「貯蔵庫」というような意味があるそうです。転じて、コンピュータの世界では「一度取り出したデータ（あるいは頻繁にアクセスするデータ）を一時的に格納しておく記憶領域」という意味で用いられます。

具体的にはCPUに直結した小容量だが高速の記憶領域を用意して、メモリーから取り出したデータをこの「キャッシュ領域」にも保存します。もう一度同じアドレスへのアクセスが発生した場合には、遅いメインメモリーにはアクセスせずにキャッシュ上のデータを用います。これによってメモリーアクセス待ちによる性能低下を回避しようとしています。

キャッシュを多段にする

しかし、高速なキャッシュはそれほど容量を大きくすることができません。現在、私の手元にあるコンピュータ(CPU: Intel Core i7 2620M) ではデータ用と命令用にそれぞれ64K バイトずつしか搭載されていません。この容量は2015年時点の最新CPUでも変化していません。

そこでキャッシュに乗っていないデータへのアクセスをできるだけ減らすため、現代のCPUはキャッシュを多段に搭載しています(図1)。例えば、Core i7 の場合、64K バイトというのはレベル1 (L1) キャッシュのことで、これにL1 キャッシュほどは高速ではないが、それよりも大容量のレベル2 (L2) キャッシュが512K バイト、さらに大容量のレベル3 (L3) キャッシュが4M バイト搭載されています。

多くのプログラムでは、同じアドレスのデータへ頻繁にアクセスする傾向がありますから、4M バイトのキャッシュがあれば、かなりの確率で遅いメインメモリーにアクセスせずに済みます。

データの書き換えに課題あり

マルチコア環境では、L1 キャッシュは各コアが独立に持っていて、L2 と L3 のキャッシュは複数コア間で共有することが多いようです。

で、キャッシュというのはあくまでも一時的なデータの保存場所ですから、データの書き換えが発

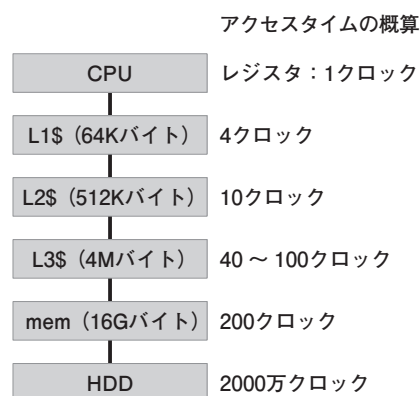


図1 多段キャッシュ

このクロック比はあくまでも概算。アクセスタイムはCPUごとに違い、クロック以外の要素もあるので正確な数値は示しづらい。「L1\$」の「\$」は「キャッシュ」の略で、cache (キャッシュ) と cash (現金) をかけたダジャレが由来だ。

生したときには、後からそれを参照できるようにデータをメインメモリーに書き戻す必要があります。また、ほかのコアがメモリー上のデータを書き換えた場合、手元のキャッシュ上のデータは古い意味のないものになってしまいます。古いキャッシュは無効化して、改めてデータを読み込んだりする必要があります。

これは実は大変複雑な問題で、プログラミングに関することわざに「コンピュータサイエンスにおける難題が二つある。キャッシュの無効化と名前付けだ」(Phil Karlton氏)というものがあるくらいです。ここではその難題の詳細には立ち入りません。一見簡単そうに見えるメモリーアクセスも、CPU内部では大変複雑なことが行われているということだけ紹介しておきます。

パイプラインを再考

さて、話を元に戻しましょう。パイプラインにタスクが並んでいるときに、複数コアを最大限活用するようにそれぞれのタスクを別のスレッドに割り当てることにしたと冒頭で言いました。そうすると、パイプラインを流れるデータは異なるスレッドからアクセスされることになります。これでは高速なL1キャッシュへのアクセスを有効活用できる場面が少なくなると考えられます。

つまり、かなり厳しい制限のあるキャッシュ容量を有効に活用するためには、同じデータは極力同じコアからアクセスすることが望ましいのです。同じデータに複数のコアからアクセスすると、貴重なL1キャッシュに重複するデータが格納され、古いデータが掃き出されてしまうことにつながります。これはちょっともったいない。

この現象は複数のパイプラインが動作するようになると一層顕著になると考えられます。現在、Streamで実行できるような非常にシンプルなパイプラインではさほど問題にならない「キャッシュの無駄遣い」が、パイプラインが混み合ってきたら顕在化すると思われます。

同一コアで動かす方が高速

例えば図2(1)のようなパイプラインを考えてみましょう。最初のタスクでは標準入力から1行分の文字列を読み込みます。読み込んだ文字列データはL1→L2→L3キャッシュを経由して、最終的にメインメモリーに書き込まれます(図2(2))。

文字列を大文字に変える次のタスクが、別のスレッド、つまり恐らくは別のコアで実行されるとそのデータは当該コアのL1キャッシュには乗っていないでしょうから、L2キャッシュを探しに行くことにな

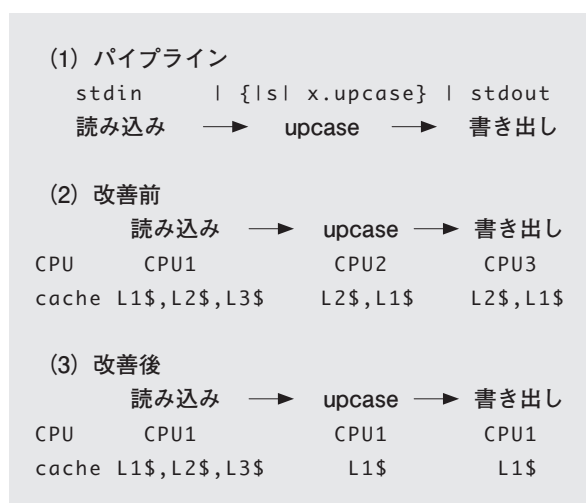


図2 パイプライン上のタスクとキャッシュ

ります。コア間で共有されることが多いL2キャッシュにはそのデータが残っていることが期待できません。その場合はL2キャッシュからの読み出しで済むでしょう。あるいは別のパイプラインが同時に実行されていた場合などには、L2キャッシュもあふれているかもしれません。その場合にはL3キャッシュ、そこもあふれていたらメインメモリーへのアクセスになります。

一方、同一パイプラインのタスクをできるだけ同じコアに割り当てるようにした場合、キャッシュへのアクセスは図2(3)のようになります。同じコアでタスクが実行される場合、データがL1キャッシュに残っていることが期待できますから、キャッシュがあふれていなければ、高速なL1キャッシュへのアクセスだけで済む可能性があります。

キャッシュを意識する

このように現代のCPU上で実行されるソフトウェアの実行性能にはキャッシュの活用が大きな影響を与えます。ソフトウェアにとってキャッシュへのアクセスは「透過的」であり、性能以外の点ではその存在は感じられないようになっています。しかし、性能においては大きな違いを生みます。キャッシュを有効に活用しない場合、何十倍も遅くなることがあります。21世紀の高速ソフトウェアの開発にはキャッシュを意識することが欠かせません。

とはいうものの、目に見えないだけにキャッシュの活用はなかなか困難です。さらにスレッドが絡むとキャッシュの状態を理解することは人間に可能な領域を超えるのではないかと思います。当て推量でプログラムをいじってもなかなか性能向上につながりません。

そんなときには、測定です。正しい測定こそが性能改善の最善手です。Linux上でキャッシュの働きを測定するツールとしては、oprofileやcachegrindのようなものがあります。これらのツールは単にどの関数でどれだけ時間を消費しているかを測定するだけでなく、命令ごとのキャッシュミスによる遅延などもレポートしてくれます。これらのツールの使い方については、将来Streamの性能改善について解説するときに一緒に説明しましょう。

■ シンボルの扱い

さて、次は言語仕様の話です。

Rubyには「シンボル」というデータ型があります。これは変数名や識別子などを表す型で、名前を持ちます。そういう意味では文字列によく似ているのですが、文字列とは以下の点で異なります。

- 同じ名前を持つシンボルは一つしか存在しない
- その結果、一致判定が高速（内容をチェックする必要がない）
- Rubyの文字列のように内容を変更できない

Rubyではこの比較の高速性を利用して、メソッド名・変数名の指定やキーワード引数、マップのキーなどにシンボルを広く使っています。

RubyのシンボルはLispから受け継いだものです。元々Ruby開発前からLispの強い影響を受けていた私としては、シンボルの導入はごく当然のことだと考えていました。

Lispのシンボル

1958年生まれのLispでは、シンボルはリストと並んでLisp基本データ型の一つです。最初のバージョンのLispにもシンボルは存在していたそうです。その時代にはむしろ文字列の方が存在しなかったとか。現代の我々が文字列を利用するような局面では、シンボルを代わりに使っていたと聞きます。ですから、シンボルは文字列よりも古いデータ型なのです。

Lispでは、シンボルはさまざまなところに使われます。Lispではプログラムそのものもリスト構造で表現されますが、そこに登場する変数名や関数名など名前に関するものはすべてシンボルで表現されます(図3)。そういうわけでシンボルのないLispなど考えられないのです。

```
; Lispによる階乗プログラム
(defun fact (n)
  (if (eq n 1)
      1
      (* n (fact (- n 1)))))
; 網掛けされた部分はシンボル
```

図3 Lispプログラム中のシンボル

初心者の戸惑い

そこでRubyにもシンボルを導入したのですが、Lispをご存じない方でRubyを学ぶ人の中には、なぜ文字列とシンボルが異なるのか理解するのが難しい人が現れました。

私にとってはシンボルと文字列が違うのはあまりにも当たり前で、最初何に不満を持っているのか理解できませんでした。しかし何度も同じような文句を聞いて、不満の理由と背景が次第に分かってきました。

つまり、内部のデータの持ち方はともかく、外側から見ると文字列とシンボルは同じ「文字の列として表現される内容」を持つ存在に見えます。そういう視点からはシンボルとは単に変更できない文字列なのに、なぜか適用できる操作(メソッド)が圧倒的に少ない不便なものに見えるわけです。「不便だから文字列と同じメソッドを用意しよう」とか「文字列とシンボルを統一しよう」というリクエストを何度も受けました。それでようやく「ああ、文字列とシンボルに対する捉え方が私と根本的に異なるのだ」ということが理解できるようになりました。

このような「誤解」あるいは「認知のズレ」が発生するのは、Rubyほど強くLispの影響を受けていない言語にはシンボルのようなものは登場しないからでしょう。とはいえ、内部的にはシンボルのようなものが不要なわけではないのです。

他言語におけるシンボル相当

では、シンボルのない言語ではシンボル相当のことをどうやって実現しているのでしょうか。

一番簡単な方法は、すべて通常の文字列で代替することです。内部的な識別子（名前）の管理にも通常の文字列を使うことは、それほど困難なことではありません。性能を気にしなければ。

通常の文字列を識別子に用いることの最大の欠点は、比較に文字列の長さに比例した時間がかかることです。文字列"abcd"と"abce"の違いを判定するためには、1文字ずつ比較して4文字目までチェックする必要があります。識別子のように頻繁に比較されるものに通常の文字列を用いると性能問題を起こしやすくなります。

そこでPythonなど、いくつかの言語では、シンボルを導入する代わりに文字列に特別の工夫をして、性能問題を回避しています。

具体的には、一定の条件を満たす文字列については、同じ内容を持つものは同一のオブジェクトを返すようにする（インターン化）ことで、内容を見ずに一致検査ができるようにします。

一定の条件とは、Pythonの場合、以下の二つのいずれかです。

- 一定の長さ（デフォルトの場合20文字）以下の文字列リテラル
- 明示的に「intern」した文字列

一方、Luaの場合は原則的にすべての文字列がシンボル化されます。つまり同じ内容を持つ文字列は、すべて同じオブジェクトになります。

これらが可能なのは、PythonとLuaにおいて文字列が不変、つまり内容を変更することができないからです。Rubyのような文字列の内容が変更できる言語では、このような選択は取れません。難しいトレードオフです。

Streamのシンボル（相当）

さて、Streamの話に戻しましょう。Streamのような言語にもシンボル相当のものは必要です。これをどうするのかは言語設計上の重要な判断になります。

Rubyのような独立したシンボルというデータ型を導入するか、あるいはPythonやLuaのように文字列を使った上で、シンボリックの使い方を支援するような仕組みを取り入れるか、どちらが良いのでしょうか。

実はStreamの設計ポリシーとして「特に思い入れがなく、明らかな技術的メリットもない場合、Rubyとは異なる選択をする」というものがあります。Streamのオブジェクトが原則変更不可である点や、ブロック構造の表現にプレース（{ }）を用いているのもこのポリシーによるものです。

ここから考えると、シンボルについてはRubyとは異なったやり方を採用しようと思います。幸い、Streamでは文字列を変更することはできない仕様にしたので、Pythonなどと同様に文字列にシンボルとしての役割も持たせることにします。

文字列生成関数を修正

具体的にどうするかというと、Stroom の文字列生成関数に手を入れて、同じ内容の文字列は同じ文字列になるように変更します。

新しい文字列生成の手順は以下のようになります。

1. 文字列オブジェクトを登録するハッシュテーブルを用意する。ハッシュテーブルはポインタ(`const char*`)と長さ(`size_t`)をキーにして、文字列オブジェクト(`struct strm_string*`)をバリューとする。
2. 文字列生成のために渡されたデータ(ポインタと長さ)からハッシュテーブルを検索し、既に登録されていれば、その文字列オブジェクトを返す。
3. 登録されていないければ、文字列オブジェクトを生成し、ハッシュテーブルに登録する。

ここまでは、それほど難しいことはありません。実際にこの手順に従って文字列オブジェクトを生成する関数 `strm_str_new()` を図4に示します。

注意すべき点は、ハッシュテーブルに `khash` というライブラリを使っている点です。この `khash` では、ヘッダーファイルをインクルードするだけでハッシュテーブルが使えるようになります。マクロを活用してテンプレート型のようなことも実現し、任意の型をキーやバリューにできます。

スレッド問題

さて、これで Lua と同様に文字列をシンボルのように扱うことができるようになりました。同じ内容を持つ文字列は同じオブジェクトとなることが保証されるようになりました。

しかし、実はこれは不完全で、少なくとも二つの問題が残っています。

最初の問題はスレッド対応です。図4のプログラムはシンボルテーブルへの参照と登録を含みますが、複数スレッドが同時にシンボルテーブルにアクセスすれば、最悪の場合データが破壊されます。排他制御をするなど、なんらかの方法でスレッドに対応する必要があります。

考え付いたアイデアは二つです。一つはシンボルテーブルへのアクセス(具体的には `kh_put` の呼び出し)を `mutex` で囲むことです。`lock/unlock` には毎回最低でも 100ns はかかりますが、まあ、誤差の範囲内でしょう。こちらのやり方は数行の追加で済むので圧倒的に簡単に対応できます。

別のアイデアは、シンボルテーブルへの登録をイベントループが始まる前、まだマルチスレッド実行に入る前にだけ行うことです。マルチスレッド実行中はシンボルテーブルへ登録しません。こちらのやり方は排他制御が不要な分、マルチスレッド環境での性能低下がない(または少ない)ことが期待できます。

実際問題、シンボルになるような文字列が作られるのはプログラムの初期化フェーズであることが予想できるので、これでうまくいきそうな気がします。ただし、実装がやや複雑になることと、同

じ文字列でもシンボルテーブルに登録されるものとされないものが存在することになるので、それについて対応する必要が出てきます。

二つのアイデアを比較すると、前者のものが圧倒的に優れているようです。そのため、今回の実装ではmutexを使うものを採用しますが、次に述べる問題への対処を考えると、将来的にはほかのアイデア、特に後者についても検討する必要がありそうです。

```
#include "khash.h"

/* ハッシュテーブルの定義 */
KHASH_INIT(sym, struct sym_key,
    struct strm_string*, 1, sym_hash, sym_eq);

/* シンボルに登録するハッシュテーブル */
static khash_t(sym) *sym_table;

/* 文字列オブジェクトの割り当て */
static struct strm_string*
strm_str_alloc(const char *p, size_t len)
{
    struct strm_string *str
        = malloc(sizeof(struct strm_string));

    str->ptr = p;
    str->len = len;
    str->type = STRM_OBJ_STRING;

    return str;
}

/* 文字列オブジェクト生成関数 */
struct strm_string*
strm_str_new(const char *p, size_t len)
{
    khiter_t k;
    struct sym_key key;
    int ret;

    /* シンボルテーブルの初期化 */
    if (!sym_table) {
        sym_table = kh_init(sym);
    }

    /* シンボルテーブルに載っているか */
    key.ptr = p;
    key.len = len;
    k = kh_put(sym, sym_table, key, &ret);
    /* 存在する: ret == 0 */
    /* 存在しない: kの位置に挿入可能 */

    if (ret == 0) {
        /* found */
        /* 見つかったらそれを返す */
        return kh_value(sym_table, k);
    }
    else {
        /* なければオブジェクトを割り当てる */
        struct strm_string *str;

        /* allocate strm_string */
        if (readonly_data_p(p)) {
            /* リードオンリー領域ならコピー不要 */
            str = strm_str_alloc(p, len);
        }
        else {
            /* 文字列データをコピーする */
            char *buf = malloc(len);
            if (p) {
                memcpy(buf, p, len);
            }
            else {
                memset(buf, 0, len);
            }
            str = strm_str_alloc(buf, len);
        }
        /* 生成したオブジェクトをテーブルに登録 */
        kh_value(sym_table, k) = str;
        return str;
    }
}
```

図4 シンボルに対応したstrm_str_new()

シンボルごみ問題

二つ目の問題は、私が「シンボルごみ問題」と呼んでいるものです。ここでは文字列を処理するたびに、すべての文字列をシンボルテーブルに登録します。さまざまな種類の文字列が登場すると、それだけシンボルテーブルに登録される文字列が増えて、メモリーを圧迫する危険性があります。

文字列とシンボルが分離していて、そのような危険性が少ないはずの Ruby でさえ、外部入力からシンボルを生成することで、メモリーを大量に消費してプログラムの実行を阻害させる攻撃を許す脆弱性が発見されています。そのため、Ruby では 2014 年 12 月にリリースされた Ruby 2.2 においてシンボルもガーベージコレクション (GC) の対象になりました。Ruby 2.2 以降では使われていないシンボルは、自動的に回収されます。

もちろん、Stroom においても同様の問題が発生する危険性があります。将来的にはなんらかの対策が求められるでしょう。

とはいえこの問題は、忘れてはいけないが急いで対処する必要もないタイプです。だから、ここでは考察だけしておいて、今後の課題としましょう。

シンボルの GC

では、「シンボルごみ問題」に将来対処するとして、どのような対策が考えられるでしょうか。

一つには、Ruby のようにシンボルもガーベージコレクションの対象にすることがあります。この場合、シンボルテーブルから文字列への参照は「弱い参照」(参照はしているがガーベージコレクション保護の対象にはならない参照)として、文字列オブジェクトが回収されるタイミングでシンボルテーブルから取り除くことになるでしょう。

もう一つは、Python のように条件を満たすものだけシンボルテーブルに登録することで、テーブルの肥大化を避けることです。前述のシンボルテーブルに登録する文字列とそうでない文字列が登場する方式ですね。

しかし、こちらの方式にはいくつか欠点があります。まず、シンボルテーブルに登録されていない文字列が存在するということは、結局は文字列の内容による比較が必要になる局面が多発する危険性があります。それでは、せっかくのシンボルの良さが失われてしまいます。

さらに、このやり方だけでは、結局はシンボルテーブルの肥大化を完全には避けられません。実際、Python でも明示的に登録した文字列はガーベージコレクションの対象にしています。

このような点を考慮すると、将来の「シンボルごみ問題」への対処は、シンボルのガーベージコレクションで対応すべきであろうことが分かります。

ただし、現在の Stroom はメモリー管理に libgc (Boehm-Demers-Weiser's GC) を使っているため、動的に割り当てた (malloc した) 領域の解放について心配する必要はありません。しかし、前述の弱い参照などを実現することは困難です。

シンボルのガーベージコレクションを実現するためには、より細かい制御を行うため、libgc を捨

てて自前のガーベージコレクションを実装する必要がありそうです。

まとめ

今回はキャッシュとメモリアクセスのコストについて、それからプログラミング言語におけるシンボルの設計についての二本立てで解説しました。

メモリアクセスのような自明に思えることでさえ、普段見えないキャッシュのような仕組みによって支えられていること、また、そのキャッシュの振る舞いによってソフトウェアの性能が数倍あるいは数十倍も変化する可能性があるというのは興味深い点です。

興味深いといえば、プログラミング言語における識別子（名前）の扱いをどのようにするかというように一見些細に見えるようなことでさえ、考慮すべきことがたくさんあります。これも、あまり知られていない言語設計の面白い点です。

タイムマシンコラム

世代違いによる認知の偏り

2015年5月号掲載分です。現代のコンピュータは我々の想像以上に複雑になっていて、時々予想に反した動作をします。キャッシュの存在による実効性の変化はその最たるものの一つです。

我々が用いているプログラミングモデルでは、データは二次記憶領域にあるファイルと、一次記憶領域であるヒープまたは変数しか見えませんが、実際には、変数はレジスタに割り当てられているものとメモリーに割り当てられているものがあり、外見上は全く同じなのにアクセス時間は段違いです。また、メモリーに割り当てられている変数やヒープへのアクセスもキャッシュに乗っているかどうかでアクセス時間が全く異なります。この辺に気を付けたプログラミングができるかどうか、ソフトウェア性能に大きく影響します。

とはいえ、現時点でのStreamの実装は効率を全く考慮していないこともあり、性能という観点からはキャッシュを意識する段階までとても到達していません。Streamのためというよりは、ソフトウェア性能に関する読み物として捉えていただければと思います。

もう一つのテーマ、シンボルについても少し語っておきましょう。本文にも書きましたが、元々Lispの強い影響を受けていた私にはシンボルと文字列を区別するというのはあまりにも自明で全く疑問に思ったことがありませんでした。しかし、Lispのことをあまり知らない「若い」世代にはこの区別が無意味なものに感じられたというのは、非常に興味深いと思いました。私が自明と思っていたことが、バックグラウンドに依存する「認知の偏り」の典型的な例であったことを示していたからです。時代や背景が変化すると常識も変化するということを実感しました。

2-7

AST (抽象構文木) に変換

今回はStreemの構文解析器に手を入れて、まがりなりにも言語として動くところまで持っていくます。構文解析をして、その結果を抽象構文木に変換します。それをさらに仮想マシンのマシン語に変換すれば実行効率を高められますが、まずは「動くこと」を優先します。

Streem内部の仕組みは少しずつ出来上がってきましたが、まだ言語としては十分に動作しません。そこで今回は構文解析器に手を入れて、まがりなりにも言語として動くところまで持っていこうと思います。

その前に2-6のフォローアップを。2-6では「シンボル」に対応するため、文字列すべてをテーブルに登録し、同じ内容の文字列は同じオブジェクトになるように実装しました。

ところがその後、いくつか実験を繰り返したところ、これはあまり良くないことが分かってきました。すべての文字列をテーブルに登録すると、読み込むデータが大きくなるにつれて、消費するメモリーが予想以上に大きくなってしまいます。

2-6でも解説したようにLuaがこのような手法を採用しているので、あまり問題ないのかなとなんとなく考えていました。しかし、やはり思い込みはせずに自分で測定することが大切です。

シンボル対応の新手法

ということで、新たに以下のような対応を入れることにしました。

まずイベントループに入る前、シングルスレッドのときには排他制御について心配する必要がありませんから、文字列生成の時点でシンボルテーブルに登録することにします。ただし、Pythonのやり方を参考に、一定の長さ以上（とりあえずは64文字以上）の文字列はシンボルテーブルへの登録をしません。

イベントループが始まって、マルチスレッドモードに入ったら、競合を避けるため、通常の文字列生成ではシンボルテーブルにアクセスしないで個別に割り当てます。この場合、同じ内容の文字列でも異なるオブジェクトになります。

ただし、何らかの理由でシンボル化した文字列が欲しいこともあります。このため明示的にシンボル化した文字列を得るための新しい関数「strm_str_intern()」を作りました。この関数を呼び出すとシンボルテーブルを排他制御しながらアクセスし、シンボル化された文字列を返します。

この一連の変更で、すべての文字列がシンボル化されているわけではなくなったので、文字列の一致判定の部分も変更する必要があります。これまではオブジェクトのアドレス比較だけで一致判定ができましたが、今後はシンボル化されていない文字列は内容を比較するようになりました（図1）。キーになるのは、シンボル化された文字列はSTRM_STR_INTERNEDというフラグが設定されているということです。

```
int
strm_str_eq(strm_string *a, strm_string *b)
{
    /* アドレスが同じなら一致 */
    if (a == b) return TRUE;
    /* 両方の文字列がシンボル化されていれば */
    if (a->flags & b->flags & STRM_STR_INTERNED) {
        /* アドレスが一致しないということは違うオブジェクト */
        return FALSE;
    }
    /* ここからは内容による比較 */
    /* 長さが違っていれば違う文字列 */
    if (a->len != b->len) return FALSE;
    /* 内容进行比较して一致していれば同じ文字列とする */
    if (memcmp(a->ptr, b->ptr, a->len) == 0) return TRUE;
    /* 一致しなかった */
    return FALSE;
}
```

図1 文字列比較関数

構文解析アクション

それでは言語処理の実装に戻しましょう。

これまでに「Yacc」を用いて文法を定義する方法について解説してきました。Yaccの定義をツールに与えると構文解析をする関数を生成してくれます。

文法定義に「アクション」を追加することで、文法に応じた処理を実行することができます。「アクション」とは、ルールに合致したときに実行されるコードです。

後ほど出てきますが、アクション部では「\$\$」がそのルールが生成する値、「\$1」などが文法のn番目の要素が生成した値になります。注意しないといけないのは、アクションは文法ルールがマッチした途端に実行されるので、予想とは異なる順序で実行される可能性があることです。イベントドリブンのプログラムだと思えばよいかもしれません。

抽象構文木に変換する

ということは、言語処理の本質はアクション部に何を書くかということになります。ある程度複雑な言語を扱おうと思えば、それなりのアクションを記述する必要があります。

多くの言語処理系では、このアクション部分で、プログラムを木構造に変換します。元のプログラムは単なるテキストでしかありませんが、それでは扱いにくいからです。

このようにプログラムを木構造に変換したものを「抽象構文木」(Abstract Syntax Tree - AST)と呼びます。例えば、Streamで「Hello World」に相当する「標準入力から読み込んだ文字列を標準出力に書き出す」という簡単なプログラムに対する抽象構文木は図2のようになります。

演算子を持つ式(op)は、「node_op」というノード(節)で表現され、それには三つの枝があります。一つは演算子名でopという枝に格納されます。今回は「|」が演算子名です。引数は2番目(lhs)と3番目(rhs)の枝に格納されます。lhsとrhsはそれぞれ「left hand side (左辺)」「right hand side (右辺)」の略です。

演算子以外の式や文についても相当するノードを用意しています。例えば関数呼び出しは「node_call」、if文は「node_if」のような形です。

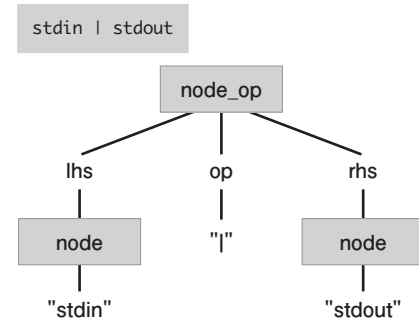


図2 抽象構文木の例
lhsとrhsはそれぞれ「left handside (左辺)」「right hand side (右辺)」の略。

構文木ノードを構造体で表現

この抽象構文木を表現する構造体の記述は図3のようになります。

抽象構文木のノードを構成する構造体は先頭に「node_type type」(NODE_HEADER)という共通のメンバーを持っています。プログラム中では、node構造体へのポインタとして扱っておいて、必要に応じてtypeの値を見てキャストしています。型安全性という意味では行儀の悪いプログラムですが、動的型を持つ言語では一般的に用いられるテクニックです。

後は、構文に応じたノードを作る関数を定義して(図4)、それをアクションから呼び出すだけです(図5)。

実際のソースコードは、streamリポジトリ(<https://github.com/matz/stream>)のsrcディレクトリにあるparse.y(構文解析部のyaccソースコード)とnode.c(ノード生成部)を参照してください。

これで、Yaccが生成したyyparse関数を実行すると、抽象構文木が得られる仕組みができました。

```

typedef enum {
    NODE_ARGS,
    NODE_PAIR,
    NODE_VALUE,
    NODE_CFUNC,
    NODE_BLOCK,
    NODE_IDENT,
    NODE_LET,
    NODE_IF,
    NODE_EMIT,
    NODE_RETURN,
    NODE_BREAK,
    NODE_VAR,
    NODE_CONST,
    NODE_OP,
    NODE_CALL,
    NODE_ARRAY,
    NODE_MAP,
} node_type;

#define NODE_HEADER node_type type

typedef struct {
    NODE_HEADER;
    node_value value;
} node;

typedef struct {
    NODE_HEADER;
    node* recv;
    node* ident;
    node* args;
    node* blk;
} node_call;

// 以下、構造体定義が続く

```

図3 抽象構文木の構造体

```

extern node* node_array_new();
extern node* node_pair_new(node*, node*);
extern node* node_map_new();
extern node* node_let_new(node*, node*);
extern node* node_op_new(const char*, node*, node*);
extern node* node_block_new(node*, node*);
extern node* node_call_new(node*, node*, node*, node*);
extern node* node_int_new(long);
extern node* node_double_new(double);
extern node* node_string_new(const char*, size_t);
extern node* node_if_new(node*, node*, node*);
extern node* node_emit_new(node*);
extern node* node_return_new(node*);
extern node* node_break_new();
extern node* node_ident_new(node_id);
extern node* node_ident_str(node_id);
extern node* node_nil();
extern node* node_true();
extern node* node_false();

// if文に相当するノード
node*
node_if_new(node* cond, node* then, node* opt_else)

```

```

{
    node_if* nif = malloc(sizeof(node_if));
    nif->type = NODE_IF;
    nif->cond = cond;
    nif->then = then;
    nif->opt_else = opt_else;
    return (node*)nif;
}

// 整数リテラルに相当するノード
node*
node_int_new(long i)
{
    node* np = malloc(sizeof(node));

    np->type = NODE_VALUE;
    np->value.t = NODE_VALUE_INT;
    np->value.v.i = i;
    return np;
}

// 同様にノードを生成する関数定義が続く

```

図4 ノード生成関数（抜粋）

```

program    : compstmt
            { /* 生成されたノードを */
              /* parser_state pに格納する */
              p->lval = $1;
            }
            ;

/* 中略 */

primary0   : lit_number /* lex.l中でノードが生成される */
            | lit_string /* 同上 */
            | identifier
            {
                $$ = node_ident_new($1);
            }
            | '(' expr ')'
            {
                $$ = $2;
            }

```



```

| '[' args ']'      /* リスト */
{
  $$ = node_array_of($2);
}
| '[' ']'           /* 空リスト */
{
  $$ = node_array_of(NULL);
}
| '[' map_args ']' /* マップ */
{
  $$ = node_map_of($2);
}
| '[' ':' ']'       /* 空マップ */
{
  $$ = node_map_of(NULL);
}
| keyword_if condition '{' compstmt '}' opt_else
{
  $$ = node_if_new($2, $4, $6);
}
| keyword_nil
{
  $$ = node_nil();
}
| keyword_true
{
  $$ = node_true();
}
| keyword_false
{
  $$ = node_false();
}
;

```

図5 抽象構文木を作るアクション (抜粋)

構文木のまま実行する

文法を解析した結果の抽象構文木ができたら、後はこれを処理することになります。処理の手順はいろいろあって、言語処理系ごとに異なっています(表1)。

表にしてみると、抽象構文木から仮想マシン(VM)のマシン語(慣習的にバイトコードと呼ぶことが多い)を生成する処理系が圧倒的に多いですね。

言語処理系	処理
Ruby 1.8	抽象構文木をそのまま実行
Ruby 1.9以後	仮想マシン語を生成してからVMで実行
mruby	仮想マシン語を生成する。直接VM実行も可能
Python	仮想マシン語を生成してからVMで実行
Java	仮想マシン語を生成する

表1 言語処理系ごとの抽象構文木に対する処理

これには理由があって、メモリーアクセスの効率などの面で、抽象構文木のリンクをたどって実行するよりも、一度バイトコードを生成してVMで実行する方が、実行効率が高いことが多いからです。Rubyがバージョン1.9以後で性能が劇的に改善したのも、この仮想マシンの導入が大きな理由です。

では、Streamではどうするのかというと、もちろん最終的にはなんらかのVMを導入するつもりですが、VMを実装するためにはそれなりの手間と時間がかかります。できるだけ早く「動いている」という状態を達成するために、とりあえずRuby1.8と同様の抽象構文木を直接たどる実行関数を作ることにします。

無駄なことだと思われるかもしれませんが、実際に動かしてみないとイメージがつかめないことが多いものです。言語仕様について試行錯誤をするためにも、できるだけ早く動く状態に持つていくことは大切です。また、何よりも自分の書いたプログラムを実際に動かして試せるのは、プログラマにとってモチベーションの源泉です。思えば、20年を越えるRubyの開発でも、一番辛かったのは、最初に動作するところに持つていくまでの半年間だったような気がします。

抽象構文木のトラバース

トラバース (traverse) とは「たどる」という意味です。構文解析器はプログラムのテキストを変換して、抽象構文木を表現する構造体のリンク構造にします。プログラムを解釈するには、このリンク構造をたどる必要があります。

Streamの抽象構文木をトラバースしながら実行する関数はexec.cにあるexec_expr()関数です。この関数は一つの巨大なswitch文で、ノードの種別ごとにそれに合わせた処理をします。exec_expr関数はかなり大きいので(133行)、図6にその抜粋を載せておきます。

```
/* 抽象構文木を実行する関数 */
/* ctx: コンテキスト */
/* np: 抽象構文木 */
/* val: 実行結果 */
/* 戻り値: 0 - 成功、 1 - 失敗 */
static int
exec_expr(node_ctx* ctx, node* np, strm_value* val)
{
    int n;

    /* 抽象構文木がNULLなら失敗 */
    if (np == NULL) {
        return 1;
    }
}
```

```
/* 抽象構文木のタイプで分岐 */
switch (np->type) {
/* 変数アクセス */
case NODE_IDENT:
/* 変数名から値を取り出す */
    *val = strm_var_get(np->value.v.id);
    return 0;
/* if文 */
case NODE_IF:
{
    strm_value v;
/* npをnode_ifにキャストする */
    node_if* nif = (node_if*)np;
/* 条件部を評価 (exec_exprを再帰呼び出し) */
    n = exec_expr(ctx, nif->cond, &v);
/* 条件部評価が失敗していたら失敗 */
    if (n) return n;
/* 条件部が真ならば */
    if (strm_value_bool(v)) {
/* then部を評価 */
        return exec_expr(ctx, nif->then, val);
    }
    else if (nif->opt_else != NULL) {
/* else部を評価 */
        return exec_expr(ctx, nif->opt_else, val);
    }
    else {
/* else部がなければnull */
        *val = strm_nil_value();
        return 0;
    }
}
break;
/* 演算子式 */
case NODE_OP:
{
/* node_opにキャスト */
    node_op* nop = (node_op*)np;
    strm_value args[2];
    int i=0;

/* 左辺を評価 */
    if (nop->lhs) {
        n = exec_expr(ctx, nop->lhs, &args[i++]);
        if (n) return n;
    }
}
```

```

/* 右辺を評価 */
if (nop->rhs) {
    n = exec_expr(ctx, nop->rhs, &args[i++]);
    if (n) return n;
}
/* 関数呼び出し（「|」という名前の関数を呼び出す） */
return exec_call(ctx, nop->op, i, args, val);
}
break;
/* 関数呼び出し */
case NODE_CALL:
    ...
}
}

```

図6 exec_expr関数（抜粋）

再帰呼び出しの活用

このような木構造をトラバースする関数の実現には、再帰呼び出しを使うのが一般的です。例えば、演算子式は以下の手順で評価します。

1. 左辺を再帰呼び出しで評価する
2. 右辺を再帰呼び出しで評価する
3. 両辺の評価結果を引数として、演算子に相当する手続きを呼び出す

そのほかの式や文についても同様です。ノードのタイプごとに処理を記述する必要があるのですが、どうしても行数が長くなってしまいますが、やっていることは同じようなことの繰り返しなので、さほど複雑ではありません。

同じように木構造を再帰呼び出しでトラバースしているのが、main.cのdump_node()関数です。こちらはデバッグ用にインデントを使って木構造を表示しています。構成はexec_expr()と似たようなものです。dump_node()関数の抜粋を図7に示します。p.138 図2の木構造をdump_node()によってダンプしたものが図8です。

```

stdin | stdout

STMTS:
OP:
  op: |
  IDENT: stdin
  IDENT: stdout

```

図8 抽象構文木のダンプ出力

```
/* 抽象構文木をダンプする関数 */
/* np: 抽象構文木 */
/* indent: インデントレベル */
static void
dump_node(node* np, int indent) {
    int i;

    /* 指定されたレベル分インデントする */
    for (i = 0; i < indent; i++)
        putchar(' ');

    /* NULLならばNILを出力 */
    if (!np) {
        printf("NIL\n");
        return;
    }

    /* 抽象構文木のタイプで分岐 */
    switch (np->type) {
    /* if文 */
    case NODE_IF:
        {
            /* タイプを出力 */
            printf("IF:\n");
            /* 条件部を出力 */
            dump_node(((node_if*)np)->cond, indent+1);
            for (i = 0; i < indent; i++)
                putchar(' ');
            printf("THEN:\n");
            /* then部を出力 */
            dump_node(((node_if*)np)->then, indent+1);
            node* opt_else = ((node_if*)np)->opt_else;
            /* (あれば) else部を出力 */
            if (opt_else != NULL) {
                for (i = 0; i < indent; i++)
                    putchar(' ');
                printf("ELSE:\n");
                dump_node(opt_else, indent+1);
            }
        }
        break;
    /* 演算子式 */
    case NODE_OP:
        /* タイプを出力 */
        printf("OP:\n");
        for (i = 0; i < indent+1; i++)
```

```

    putchar(' ');
    /* 演算子名を出力 */
    print_id("op: ", ((node_op*) np)->op);
    /* 左辺を出力 */
    dump_node(((node_op*) np)->lhs, indent+1);
    /* 右辺を出力 */
    dump_node(((node_op*) np)->rhs, indent+1);
    break;
    /* 中略 */
    ....
default:
    /* 知らないタイプ (エラー) */
    printf("UNKNOWN(%d)\n", np->type);
    break;
}
}

```

図7 dump_node()関数 (抜粋)

オープンソース流開発

今回紹介した抽象構文木の生成と実行をする部分は、mattnさん (Yasuhiro Matsumotoさん) が送ってくださったプルリクエストがベースになっています。私が2-6まで解説していたイベントループの部分に取り組んでいる間に、基本的な部分を実装してくださっていました。これもオープンソースの力です。

しかし、送ってくださったものをそのまま採用したわけではありません。関数や構造体の名前変更をはじめとして、大規模な変更をしています。通常の (オープンソース以外の) ソフトウェア開発の経験をお持ちの方からは、奇異に思われるかもしれません。多くの場合、一つの箇所を複数の人が変更することは「避けるべきこと」とされますし、人によっては「失礼」と感じることもあります。

確かに自分で書いたコードに愛着を感じたり、ほかの人に触られたくないと思ったりするのは自然なことかもしれません。しかし、多くのオープンソースプロジェクトでそうした感情はあまり重要視されません。

オープンソースプロジェクトでは、修正を一度送った後いなくなってしまう人も多く、「所有意識」や「担当者意識」を尊重しすぎると、進歩が止まってしまう。そもそもオープンソースに特徴的な、第三者が自由に修正を送り、作者がそれを受け入れるという行為そのものが、オリジナル作者の所有意識が強ければできないことです。そのような意識の人たちの集まりであるオープンソースプロジェクトでは、所有意識のような「自意識」は重要視されないことの方が自然なのかもしれません。

理想の実行系

Ruby 1.8を見れば分かるように、今回解説した抽象構文木をトラバースする処理系は、簡単に実装できるものの、性能が出ないのが欠点です。その最大の理由は2-6で解説したメモリーキャッシュです。

構造体のリンクをたどる場合、メモリー空間に飛び飛びに存在する構造体を次々にアクセスすることになります。このためキャッシュ効率的には、ほぼ最悪といってもよいでしょう。

メモリーキャッシュを有効活用するためには、一度のアクセスは狭い領域でできるかぎり連続的に行うのが理想的です。そのために、多くの処理系では抽象構文木を仮想マシンの命令列に変換し、実際の実行はこの命令列を解釈する方法を選択しています。

将来的には、Streamでもこのような方式を採用するつもりです。Streamの適用分野を考えると、性能は無視できない要素になることが予想できるからです。さらに先には、実行時にマシン語を生成して、そちらを実行するJIT (Just-in-time) コンパイラについてもぜひ検討したいです。

しかし、現時点では言語デザインに集中することの方が大切です。プログラマの格言にあるように「早過ぎる最適化はすべての悪の根源である」からです。

今後の計画

これで、ようやくStreamプログラムを実行している気分を感じられるようになりました。とはいっても、まだ関数定義もできなければ、パイプラインの操作もできないので、感じられるのは気分だけです。

そこで、次は関数実行部を実装して、「ちゃんとした」Streamプログラムの実行を可能にしようと思います。また、実行中に発生するエラーに対処する例外処理についても検討するつもりです。

まとめ

さて、ここまでのparse.y、node.c、exec.cで定義されているような関数を組み合わせると「とりあえず動く」レベルの言語を簡単に作ることができます。今回解説したソースコードには201506というタグを打っておきます。皆さんもこのソースコードを参考に「自分の言語」を作ってみてはいかがでしょうか。

また言語のデザインについていろいろ考えてみると、今、自分が使っている言語がなぜそういう風になっているのか、言語設計者が何を考えていたのか、その理由が想像できるようになってなかなか楽しいです。

構文木の実装方法は一つではない

2015年6月号掲載分です。今回は構文解析器（フロントエンド）とコード生成部（バックエンド）をつなぐ構文木の実装について解説しています。

構文木の実装というか、フロントエンドとバックエンドの連携の方法は一つではありません（し、木構造を使うとも限りません）。中にはフロントエンドとバックエンドを分離しないで構文解析器の中で直接コードを生成するコンパイラも存在しています。しかし、構文解析器の中での直接生成は最適化が難しくなるのであまりお勧めではありません。

そこで多くのコンパイラでは、構文解析したプログラムの構造を反映したなんらかのデータ構造をコード生成部に引き渡すことが行われます。そしてプログラムを表現する構造としては木構造が一般的なので、構文を反映した木構造、つまり構文木がしばしば用いられます。

構文木の実装はさまざまです。私が関わった言語処理系でも、今回解説した Stream ではノードの種別ごとに異なる構造体を利用していますし、mruby では Lisp 類似の cons セルのリンクによって木構造を作っています。また、CRuby ではノードを表現する構造体で union を用いることによってノード種別を表現しています。どれか一つが正解ということではないのです。

ちなみに本格的なコンパイラである gcc や clang では、gcc においては RTL (Register Transfer Language)、clang においては LLVM IR (Intermediate Representation) という中間表現を用いてフロントエンドとバックエンドをつないでいます。これらはそれぞれ構文木データではないのは興味深いところです。

2-8

ローカル変数と例外処理

言語として動き始めたStreamに、今回は二つの機能を追加します。一つ目のローカル変数では、ネストを許すか、「クロージャール」をどう実現するかといった設計上の課題があります。二つ目の例外処理では、エラーの無視による処理の継続について検討します。

2-7でようやくプログラミング言語としての形をなしてきたStreamですが、今回は「ローカル変数」と「例外処理」を追加します。

■ ローカル変数

まずはローカル変数について考えてみましょう。現代のプログラミング言語にとっては常識中の常識といえるローカル変数ですが、一昔前にはそうでもありませんでした。

30年前にタイムスリップ

例えば30年前にタイムスリップしたとしましょう。当時、ホビープログラマにとって一般的だった言語はBASICで、当時のBASICにはローカル変数がありませんでした。ローカル変数が存在しないプログラミングが想像できるでしょうか。すべての変数はどこからでも変更され得るので、値がどこで変更されたのか把握するのは非常に困難です。

当時の若いプログラマたちは、みんなBASICでプログラミングしていたわけですが、ゲームなど結構な規模のプログラムを開発することもありました。今考えてみるとよくデバッグできたなと感心します。

ローカル変数のない世界

BASICではサブルーチンを

```
gosub 4000
```

のような行番号で呼び出します。引数や戻り値のようなものはありませんから、グローバル変数を使って値を受け渡すことになります。ある変数に値を格納してサブルーチンを呼び出すと、計算結果が別の変数に格納される、というような形です。当然、情報隠蔽のような高級なことは不可能です。手続きを関数にまとめて抽象化することもできません。

さらに言えば関数がないので関数の再帰呼び出しなどもできません。

ローカル変数の導入

ローカル変数を最初に発明したプログラミング言語がなんなのか、残念ながら正確な答えは調べられなかったのですが、最初期に導入した言語がAlgolであったことは間違いのないでしょう。AlgolはFORTRANのようにその言語そのものが生き残ることはありませんでしたが、そこで導入されたアイデアが後の多くの言語に影響を与えています。

ローカル変数があると、一連の処理を隠蔽して関数として提供することができます。また再帰呼び出しもできるようになります(図1)。いや、配列を利用して自前でスタックを用意するなど無理をすれば、グローバル変数を用いて再帰を行うこともできないことはないでしょう。事実、ローカル変数なかった頃のFORTRANではそのような方法でプログラミングしていたと聞いたことがあります。しかし、正直自分でやるのは勘弁してほしいですね。

```
def fact(n)
  if n == 1
    1
  else
    n * fact(n-1)
  end
end
```

図1 再帰呼び出し

ローカル変数の実装

ローカル変数の実装はさほど難しいものではありません。実装手法としては、例えば関数のコンパイル時にローカル変数ごとに個別のインデックスを割り当てておきます。関数実行時には配列を用意して、そのインデックスでオフセットした位置に値を格納する方法などが考えられます。この関数実行ごとに用いる配列を一般的にスタックと呼びます。

気を付けるべき点は、関数の実行ごとに使用するローカル変数の数を把握しておいて、スタックがあふれないようにすることです。スタックオーバーフローはセキュリティ問題にも通じる重大な欠陥です。

では、Stroom処理系にローカル変数機能を追加してみましょう。まずは、構文解析の部分で変数への代入と参照をする部分を変更します。

Stroomに実装する

現在のStroomの構文解析器は代入の部分ではNODE_LETというノードが、参照の部分ではNODE_IDENTというノードが生成されます。

今まではNODE_LETは無視、NODE_IDENTはグローバル変数へアクセスしていましたが、ここに手を入れます。まずはNODE_LETでは、ローカル変数を初期化します。先ほどはローカル

変数ごとにインデックスを割り当てると説明しましたが、このバージョンのインタープリタは効率を全く考慮しませんので、ローカル変数にもハッシュ表を用いることにします。今後、バーチャルマシン (VM) を導入するときに効率についていろいろ考えることにしましょう。

NODE_LETに対する作業は、(もし行われていなければ)ハッシュ表の初期化と代入になります。Streamではローカル変数であっても一度代入した値を変更することはできません。このため、既に代入されたローカル変数への再代入は実行時エラーにします。もちろん、これも将来のVM導入時にはコンパイルエラーにすべきです。

NODE_IDENTは、ローカル変数テーブルをアクセスし、あればその値を、なければグローバル変数テーブルをアクセスし、グローバル変数としても定義されていなければエラーにします。

Streamでは、関数実行時にはnode_ctxという構造体が渡されます。この構造体は実行時のコンテキスト(文脈)を保存します。ローカル変数の実装のためのローカル変数テーブル(ハッシュ)は、この構造体にメンバーとして追加します。

node_ctxは、あくまでも現在のノードをたどって実行するための構造体ですから、将来のVM化の際には構造体の名前が変更になると思います。

ネストしたローカル変数

言語設計上の判断項目として「ネストしたローカル変数を許すか」というものがあり、言語によって判断が異なっています。例えばCやJavaではネストしたローカル変数が許されており、ブレースで囲んだスコープの間に宣言されたローカル変数はスコープの範囲内だけで有効です。異なるスコープで同じ名前の変数を宣言してもそれは別の変数であると見なされます(図2)。

内側のスコープで外のスコープに属する変数と同じ名前の変数を宣言しても構いません。これらの「同名だけど別の変数」は文字通り別の変数ですから同じ名前なのに型が違って構いません。コードの理解しやすさの観点からは混乱を避けるためにやめた方がよいと思います³。

```
void
func()
{
    int i = 10; /* iのスコープはfunc全体 */

    while (i--) {
        int j = 5; /* jのスコープはwhileの中だけ */

        printf("i:%d j:%d\n", i, j);
    }

    /* ブレースでスコープを導入 */
    {
        double j = 1.5; /* 上のjとは異なる変数 */

        printf("new j:%g\n", j);
    }
}
```

図2 Cのネストしたスコープ

一方、Rubyではスコープはクラス定義やメソッド定義でだけ導入され、Cのプレースのような一時的なスコープを導入する構文は（後で紹介する例外を除けば）用意されていません。

上でも説明したようにネストしたスコープによって導入され得る、同じ名前だけど別の変数というのは、それがなければならない事態というものが考えられません（単に別の名前の変数にすればよい）。その割に、混乱を招くだけなのでわざわざ導入する必要はないだろうというのがその理由です。特にRubyのような動的型付けの言語では型の不整合による間違い検出が期待できませんから、無駄な混乱は避けるべきです。

この点についてはStreamでも同様にしようと思います。

例外としてのネストしたスコープ

さて上の方でRubyには「例外を除いて」ネストしたスコープはないと述べました。その例外について解説しておきましょう。

確かにRubyはネストしたスコープを避けるように設計されています。しかし例外としてクロージャはネストしたスコープになっています。つまり、ブロックや無名関数の中で登場したローカル変数のスコープは、クラスやメソッドの定義本体ではなく、そのブロックや無名関数の範囲内だけがスコープになります（図3）。

```
# doからendまでのブロックがスコープになる
[1,2,3].each do |i| # iはブロック内だけ有効
  sq = i * i        # sqもブロック内だけ有効
  p [i, sq]
end

# 無名関数もスコープを導入する
f = ->(x) { x * x } # xはブロック内だけ有効
```

図3 Rubyのネストしたスコープ

pは引数のオブジェクトを分かりやすい文字列にして標準出力に出力するメソッド。

考えてみればこれは当然で、Rubyのブロックや無名関数は、関数ですからその関数にスコープを限定したローカル変数が必要です。でなければ、グローバル変数しかないサブルーチンの時代に逆戻りです。

混乱を減らすため、せめてもの工夫として、CやJavaのようなネストしたスコープで外側のスコープと同じ名前の変数は警告するようにしています。しかし、ローカル変数の有効範囲が一目で分からなくなるという欠点もあり、最善とはいえないのが現状です。ここではRubyの変数宣言のない仕様が裏目に出ていますね（図4）。

個人的にはここはRubyの設計の中でも気に入らない点です。過去、この点を直すためにいろ

いろいろなアイデアを検討しました。図4の警告などはそれを反映したものです。本当は単なる警告だけではなく、言語のスコープ設計によってなんとかしたかったのですが、互換性の問題とか、言語仕様を必要以上に複雑にってしまうなどの理由で採用できませんでした。

採用しなかったルールには、ローカル変数の伝搬のようなものがあります(図5)。

```
# 偶然外のスコープと変数名がかぶった
e = 10
[1,2,3].each do |i|
  p i
end

[1,2,3].each do |e|
  p e
end
# -vを指定すると
# warning: shadowing outer local variable - e
# と警告される

# スコープからの変数の持ち出しが面倒

even = nil # この行を忘れると痛い目にあう
[1,2,3].each do |i|
  if i % 2 == 0
    even = i
  end
end
p even # 事前に初期化しないとevenにアクセスできない
# -vを指定すると
# warning: assigned but unused variable - even
# と警告される
```

図4 Rubyのローカル変数の欠点

```
[1,2,3].each do |i|
  if i % 2 == 0
    # ここで初期化された変数が
    even = i
  end
end
# スコープの外でアクセスされたら、スコープを引き上げる
# つまり、外のスコープに属するローカル変数と見なす
p even
```

図5 ローカル変数の伝搬

良いアイデアだと思ったが、実装の面倒さと、解決する以上の混乱を導入する危険性にちゅうちょした。

クロージャー（関数閉包）

ブロックによるネストしたローカル変数があるということは、ブロックや無名関数から外側のスコープの変数を参照できるということです。そして、無名関数はスコープの外に出てしまった後も生き続けることがあります。

例えば図6のプログラムを見てみましょう。関数 `incdec` の中で作られた二つの無名関数はそれぞれ外側のローカル変数 `acc` を参照しています。`incdec` の実行が終わった後、通常であれば消えてしまうローカル変数ですが、無名関数から参照されているために消えません。このような状態のことを関数オブジェクトに外側のスコープの変数が「閉じ込められている」ことから、クロージャーまたは関数閉包と呼びます。

クロージャーの実装

このようなクロージャーの実装はなかなか面倒です。Ruby の処理系では関数オブジェクトにローカル変数のネスト関係を「環境」として保存しています。スコープの外側の変数にアクセスするときには、外側の環境を参照することになります。

`mruby` での関数オブジェクト (`proc`) と環境 (`env`) の定義を図7に示します。

`mruby` VM にはスコープの外側を参照する命令、`OP_GETUPVAR` と `OP_SETUPVAR` があります。両方ともいくつ上の環境にある何番目の変数を参照するかを指定するオペランドを取ります。

環境は独立した Ruby のオブジェクトで、関数オブジェクトから参照されている間は生き続けます。誰からも参照されなくなったら、ガーベジコレクターに回収されます。

```
def incdec
  acc = 0 # 「閉じ込められる」変数
  inc = ->() {
    acc += 1
    acc
  }
  dec = ->() {
    acc -= 1
    acc
  }
  return [inc, dec]
end

inc, dec = incdec()
p inc.call # => 1  accが一つ増えた
p inc.call # => 2  accが一つ増えた
p dec.call # => 1  accが一つ減った
p dec.call # => 0  accが一つ減った
```

図6 クロージャー

```
struct REnv {
  MRB_OBJECT_HEADER;
  mrb_value *stack;
  mrb_sym mid;
  ptrdiff_t cioff;
};

struct RProc {
  MRB_OBJECT_HEADER;
  union {
    mrb_irep *irep;
    mrb_func_t func;
  } body;
  struct RClass *target_class;
  struct REnv *env;
};
```

図7 `mruby` のクロージャー実装

Streamのクロージャー

しかし、Stream には Ruby とは異なる点があつて、クロージャーの実装をよりシンプルにできます。それはローカル変数であっても書き換えられないという点です。もちろん、これにより図6のような状態を持つクロージャーを作ることではできませんが、状態と副作用は関数型プログラミングでは避けるべきものと考えられていますから、それほど悪いことではないでしょう。

Stream における関数オブジェクトの定義を図8に示します。先ほど述べたようにStreamではローカル変数が書き換わる心配がないので、クロージャーは変数の値をそのままコピーしてくればよいはずです。しかし、現在の実装では簡単のため外側の環境へのリンクを保持していて、参照のたびにリンクをたどっています。VM版では性能向上のために、関数オブジェクトを生成するときに変数の値をコピーすることになります。

```
typedef struct strm_lambda {
    STRM_OBJ_HEADER;
    /* 関数の実体 */
    struct node_lambda* body;
    /* スコープを保持するコンテキスト */
    struct node_ctx* ctx;
} strm_lambda;
```

図8 Streamのクロージャー実装

コンパイル時チェック

さて、今回の実装では既に存在しているローカル変数への再代入、存在しないローカル変数への参照は実行時エラーにしています。しかし、ローカル変数への代入や参照がエラーかどうかはプログラムの字面だけを見て判断できることであり、本来はコンパイルエラーにすべきです。

実行時エラーは「実行されなければ検出されない」という不安が付きまといますが、コンパイルエラーにそんな心配は不要です。昔の言語には文法エラーでさえ実行時に行われるものもありましたが、より多くコンパイル時にチェックするというのはプログラミング言語の進化の歴史です。近いうちにそのような改善をするつもりです。

■ 例 外 的 事 象

さて、ローカル変数の次には例外処理について考えてみましょう。

さまざまな処理をするときに、なんらかの理由で意図した通りに進まない可能性はゼロではありません。処理の本筋に集中したいのはやまやまですが、それでもこのような例外的なケースを無視できません。

例えば「ファイルをオープンする」というシンプルな処理を考えてみましょう。実行する処理は「ファイル名を指定して、ファイルをオープンする」というそれだけです。LinuxなどUNIX系OSでは、この処理はopenというシステムコールが請け負います。

openシステムコールの型を図9に示します。flagsでオープンモード（読み込み、書き込み、追加のいずれか）を指定しますが、新しくファイルを作ることになる場合には、第3引数としてファイルモード、つまりファイルのアクセス権（パーミッション）を指定します。

```
int open(const char* path, int flags);
int open(const char* path, int flags, mode_t mode);
```

図9 openシステムコール

さて、このようにシンプルなファイルのオープン処理ではありますが、それでも例外的事態は発生します。openシステムコールが発生し得るエラーを表1にまとめます。EPERMのようなLinux固有のものもありますが、実に24種類もの例外的事象が発生し得ることが分かります。

もちろんほとんどの場合、ファイルは無事オープンできるでしょうが、だからといって例外的事象を無視してよいわけではありません。逆に例外的事象が発生した場合に、プログラムが異常終了するのは困ることもあるでしょう。エディタを使っている、ファイル名を打ち間違えて、存在しないファイルを指定しただけでエディタ全体が異常終了してしまったら泣けてきます。

つまりプログラムの実行には、異常事態（例外的事象の発生）が付きもので、それらを適切に処理する必要があるというわけです。

しかし、一方で例外的事象はあくまでも例外であるので、できるだけ書きたくない、読みたくないという欲求もあります。例外的事象の処理に埋もれて、プログラムのロジックの本質が読み取りにくくなるのは望ましくありません。

プログラミング言語においては、この例外的事象をどう扱うかが設計上の重要な関心事になります。

エラー名	内容
ENAMETOOLONG	ファイル名が長すぎる
EACCES	ファイルアクセス権がない
EDQUOT	disk quota（容量制限）に到達
EEXIST	ファイルが存在する
EFAULT	pathが不正なアドレス
EINTR	割込がかかった
EINVAL	不正なflagsを指定した
EISDIR	ディレクトリーを指定した
ELOOP	シンボリックリンクがループ
EMFILE	プロセスがファイルを開きすぎ
ENAMETOOLONG	ファイル名が長すぎる
ENFILE	システムがファイルを開きすぎ
ENODEV	デバイスがない
ENOENT	ファイルがない
ENOMEM	（カーネル内の）メモリーが足りない
ENOSPC	ディスクが一杯
ENOTDIR	パスがディレクトリーでなかった
ENXIO	FIFOで相手がいない
EOPNOTSUPP	tmpfileがサポートされてない
EOVERFLOW	ファイルが大きすぎる
EPERM	O_NOATIMEの権限がない
EROFS	読み込み専用ディスクに書き込もうとした
ETXTBSY	実行中のプログラムへの書き込み権限がない
EWOULDBLOCK	O_NONBLOCK指定時にブロックしそう

表1 openで発生するエラー

明示的エラーチェック

Cや、最近だとGoは明示的なエラーチェックをすることになっています。例えば、失敗する可能性がある関数を呼んだら、その戻り値をチェックして成功したかどうかを確認するということです。

この方法のメリットは言語仕様としての支援が一切不要なので非常にシンプルな点です。また、後で紹介する「例外」は暗黙のうちに実行を中断してしまうので、タイミングによって予想外の事態を引き起こす危険性があります。例えば実行が途中で中断したため、データの不整合が発生したり、メモリの解放がスキップされてメモリーリークが起きたりするような事態です。

このような問題が起きないことを、（やはり例外がある言語である）C++では「例外安全」と呼んでいます。しかし少し調べれば分かるように、C++においても例外安全を保証することは大変に困難です。明示的なエラーチェックではそのような困難さとは無縁です。

一方、明示的なチェックには例外的ケースの処理がロジックに入り込んでしまって、正常時の処理が埋没してしまい分かりにくくなるデメリットがあります。さらに明示的なチェックを忘れると、前提条件が成立しないままプログラムが突き進んでしまい、異常終了を起こす、あるいは異常終了だけではなくセキュリティ問題まで引き起こしてしまう危険性もあります。

Goでは関数が複数の値を返せることを利用して、明示的なチェックを忘れる危険性を下げますが、それでも煩雑であることには違いありません。

「例外」を発生させる

やや紛らわしいのですが、多くのプログラミング言語は、例外的な事象が発生した場合、「例外」というものを発生させてプログラムを中断させる機能を提供しています。例外機能を提供する言語としては、C++、Java、Rubyなどがあります。最近のプログラミング言語ではむしろ明示的なエラーチェックよりも例外機能の方が一般的だと思います。

例外の最大の利点は、例外的事象により前提条件が成立しなかった場合（例えばファイルが存在しないので開けなかったなど）、自動的に実行が中断されることです。このため前提条件が成立しないまま実行される危険性がなく、その分安全なことがあります。

例外のデメリットは、既に述べたように、異常事態に対して自動的に中断されるので、例外安全性の実現が困難であることです。ただし、ガーベージコレクションのあるRubyのような言語では、手動でリソース管理をするC++のような言語よりも例外安全性を維持しやすく、困難さの度合いは言語によります。

SwiftのOptional

米Apple社が発表したSwiftには、最近の言語にしては珍しく例外処理の機能がありません。その代わり、失敗する可能性がある関数はその型がOptional<T>という型になっています。Optionalはある型の任意の値かまたはnilという値を取れる型で、多くの関数は失敗したときにnil

が返るように設計されています。SwiftではOptional<T>をT?と省略表記できます。

ある型がOptionalである場合、その型の値は直接参照できません。

```
var i: Int? = 10;
println(i + 2) // エラー
```

Optional型から実際の値を取り出したい場合、その操作をSwiftではunwrapと呼んでいます。変数名の後ろに「!」を付けると値を取り出せますが、値がnilであった場合には実行時エラーになります。

```
println(i! + 2)
```

nil時に取る値も指定できます。

```
println((i??5) + 2)
```

あるいはletとifを組み合わせて明示的なnilチェックができます。

```
if let i2 = i {
    println(i2 + 2)
}
```

このように書いた場合、i2の型はOptionalではなく、Intになるのでこれ以上unwrapする必要はありません。

最後に「.」の代わりに「?.」を使ってメソッドを呼び出す方法を紹介します。「?.」を使う場合、値があるときにはその値のメソッドを実行し、nilの場合には何も行わないでnilを返せます。

```
var dog? = Dog()
dog!.bark() // nilならエラー
dog?.bark() // nilならnil
```

このようにSwiftでは例外機能を用いずに、Optional型によって例外的状況（エラー）に対応しています。このOptional型はHaskellのMaybe型やOCamlのOption型を参考にしたものと思われま。静的型をうまく利用してエラー処理をする良いアイデアだと思います。

エラーの無視

では、Streamの例外処理はどうするかについて考えてみましょう。ほかの言語と異なり、Streamには明確に二つの実行フェーズがあります。つまりパイプラインを準備する初期化フェーズと、パイプラインにデータが流れていくパイプラインフェーズです。初期化フェーズで例外的状況が発生すると、それ以降の実行の継続が困難ですから、通常の例外機能を用います。

一方、パイプラインフェーズでは大量のデータが流れていくことになります。例えば10Gバイトのデータファイルを読み込んでいるとき、そのうちの1行がなんらかの理由で壊れているからという理由で処理全体が失敗するのは必ずしも望ましくありません。

そこで、パイプラインフェーズでは明示的に指定しない限り、エラーが発生したデータに対する実行を中断するだけで、パイプラインは次のデータ処理を継続します。このようなエラーの無視による処理の継続は米Google社が開発したクラウド上のデータ処理言語Sawzallでも採用されています。

Streamでの例外処理の実装

Streamのメソッドを実装するC関数は図10のようなプロトタイプを持っており、argcとargvが引数、retがメソッドからの戻り値を意味します。で、関数からの戻り値がメソッド実行の成功か失敗を意味していて、成功のとき0を、失敗のときにそれ以外を返します。

```
int exec_plus(node_ctx* ctx, int argc, strm_value* args, strm_value* ret);
```

図10 Streamメソッドを実装する関数

まとめ

ローカル変数にしても例外にしても現代のプログラミング言語としては常識といってもよい機能です。しかし言語設計という観点から見てみると、こんな当たり前の機能でさえ、設計上のトレードオフや課題が見いだせるものです。

言語設計とはこのような重箱の隅のような点について、詳細に検討し続ける行為なのです。

長いようで短いプログラミング言語の歴史

2015年7月号掲載分です。今回は比較的独立したローカル変数と例外処理の2本立てで、お送りしました。

現代のプログラミングにおいてローカル変数は常識で存在を疑うこともありません。しかし、ほんの30数年前にはローカル変数の存在しないプログラミング言語が現役で生きていたことは驚きです。実際にその時代を経験した私すら、すっかり忘れていて、改めて驚いたくらいですから、最初から「ちゃんとした」言語でプログラミングを学んだ人にとっては驚愕の事実かもしれません。

歴史の浅いプログラミングの世界では、そういうことって時々ありますよね。はるかな昔のことと思ったらほんの数十年前の事だったり、歴史上の人物と思ってたら存命だったり。

例外処理についても触れておきましょう。例外機能自身はおそらくLispやその周辺で誕生したので、40年以上の長い歴史があります。しかし、Javaが一般化するまではあまり広く利用されていませんでした。この辺りの経緯はガーベージコレクションと同じですね。

例外機能によって明示的なエラー処理の記述が減ってプログラムの見通しが良くなるメリットがあります。しかし、メリットばかりではなく、プログラムが予想しないタイミングで中断する可能性があるのも、予想外の挙動が発生する可能性もあります。

Goとかはコンカレントプログラミングと相性が悪いという理由で、例外機能を導入せず、必ずエラーチェックをするというスタイルを採用しています。Streamの例外機能は原則的に「エラーが起きたら、そのデータを捨てる」というもので、ストリーミング処理の性質をうまく利用しています。ただし、予想外のデータによるエラーはそれでいいのですが、プログラムのバグによるエラーが無視されてデバッグが困難になるケースもあるので、ここは今後の工夫が必要です。

第3章

オブジェクト指向機能を実装する

3-1

さまざまなオブジェクト指向

「オブジェクト指向プログラミング」とは何か、についてはさまざまな意見があり、明確な定義があるとはいえません。この辺りが20世紀の終わり頃まで意見の対立があった原因でしょう。今回は歴史を踏まえたオブジェクト指向の話をしします(うんちく成分多め)。

1993年頃、Rubyを作る直前のことですが、私は会社の同僚の石塚圭樹さん(後のRubyの名付け親)と一緒に、書籍を出す企画を考えていました。石塚さんは既に「オブジェクト指向プログラミング」という書籍をアスキーから出しておられて、その続編となるような本について構想していたのです。

その企画を一言で表現するならば「(オブジェクト指向言語を)作りながら学ぶオブジェクト指向プログラミング」というものです。オブジェクト指向プログラミングのコンセプトを言語処理系を設計・実装する過程を通じて学べば、より本質的な理解ができるのではないだろうかという試みです。

残念ながら、この企画は「とても売れそうにない」という理由でボツになってしまい、日の目を見ることはありませんでした。ただ、このときに例題として作ろうとしていた言語が、後のRubyにつながったことはあまり知られていない事実です。

そもそも各種オブジェクト指向言語は、それぞれ設計者の意図や思想を反映して開発されています。だから、言語を通じてオブジェクト指向プログラミングを学ぶという構想そのものは、そんなに悪くないかもしれません。

せっかくの機会ですから、わずかなページ数ではありますが、20数年の時を経て、「言語を通じて学ぶオブジェクト指向」というテーマに再チャレンジしてみましょう。まずは「最初のオブジェクト指向言語」からです。

Simulaのオブジェクト指向

世界最初のオブジェクト指向言語と呼ばれているのは1967年発表のSimulaです。Simulaは名前から想起されるように、シミュレーション用の言語としてOle-Johan Dahl氏とKristen Nygaard氏が開発しました。当時大学などで広く用いられていたAlgolをベースに、シミュレーションに登場するエンティティを表現するためにクラスやオブジェクトを導入しました。ここでいうエンティティとは、例えば交通シミュレーションであれば信号とか自動車とかそういうシミュレーション対象にな

るもののことです。

Simulaは世界最初のオブジェクト指向言語と呼ばれますが、それは現在から振り返っていわれていることです。「オブジェクト指向プログラミング」という用語は、この後紹介するSmalltalkの開発者Alan Kay氏が「発明」したとされています。実はSimulaの時点では厳密に言うとオブジェクト指向プログラミングという呼称はまだ存在していないのです。

しかし、Simulaはクラス、継承、オブジェクト、動的結合、コルーチン、ガーベージコレクションなど、現代のオブジェクト指向言語の多くが備えている機能を初めから備えていました。ということは、まだ名付けられてはいなかったものの、その名前が存在する前から「オブジェクト指向プログラミング」の概念は確かに存在していたことになります。私たちがオブジェクト指向言語を使うとき、直接または間接的にSimulaの影響を全く受けていないものは存在しないと断言してよいでしょう。今から50年近く前、オブジェクト指向という用語より前に既に完成していたオブジェクト指向言語であるSimulaは、ある意味、プログラミング言語界のオーパーツ* といってもよいかもしれません。

開発者はきさくな方

Dahl氏とNygaard氏は2001年に「プログラミング言語Simula IとSimula67の設計を通してオブジェクト指向プログラミングの基本的なアイデアを生み出したことに対して」コンピュータサイエンスにおける最高の賞であるチューリング章を受賞しています。お二人とも2002年に亡くなっております。

私は2001年、チューリング賞受賞の2カ月前に、デンマークで開催されたJAOOというカンファレンスでKristen Nygaard教授にお目にかかったことがあります。大変きさくな方で、懇親会でいろいろな話をしてくださいました。

私に向かって「何?言語を設計している?それはオブジェクト指向言語かね。それは素晴らしい。すべてのオブジェクト指向言語は私の孫みたいなものだ。わっはは」と笑っていらっしゃいました。

Smalltalkのオブジェクト指向

さて、Simulaが世界最初のオブジェクト指向言語であるならば、最も有名で影響力の大きいオブジェクト指向言語は恐らくSmalltalkでしょう。もっとも最近のプログラマはSmalltalkに直接触れることはなく、オブジェクト指向言語といえばJavaと思っているかもしれませんが。

Smalltalkは1970年代初頭からXerox Palo Alto Research Center (PARC) で開発されたオブジェクト指向言語です。オブジェクト指向プログラミングという単語そのものが、このSmalltalk開発プロジェクトの中で生まれました。

SmalltalkはSimulaの影響を受けて誕生しましたが、最も重視した点は「ダイナブック、すなわ

【オーパーツ】 その時代の技術では加工できるはずがないなど、常識ではありえない出土品などを呼ぶ言葉。

ち子供でも使える未来のコンピュータにおける言語とはどのようなものか」ということでした。

そこで、彼らは子供でも理解しやすく、直接的に操作できる「オブジェクト」に注目し、また教育用言語として当時台頭してきていたLOGOの影響も受けつつ、「オブジェクトにメッセージを送って操作する」というモデルを中心に据えた言語を設計したのです。「ペンを上げる」「100歩前に進む」「右に120度」などの命令で構築されるLOGOのタートルグラフィックスが、「タートル」というオブジェクトと、それに対する命令というモデルで再構築された瞬間です。

LOGOをSmalltalkで表現

タートルグラフィックスを行うLOGOのプログラムを図1に、それをSmalltalkで表現したものを図2に示します。

```
FORWARD 100  
RIGHT 120  
FORWARD 100  
RIGHT 120  
FORWARD 100  
RIGHT 120  
FORWARD 100
```

図1 LOGOによるタートルグラフィックス

```
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120  
Turtle go: 100.
```

図2 Smalltalkによるタートルグラフィックス

Smalltalkの文法については少し説明が必要でしょう。「Turtle home」の部分は、「Turtleオブジェクトにhomeというメッセージを送る」という意味です。Turtleオブジェクトはそのメッセージに応じて、カーソルの位置をホームポジションに移動します。

引数があるメッセージには後ろに「:」が付きます。ちょっと変わっているのは複数の引数がある場合には、それぞれに引数の前にメッセージが付くところです。例えば、仮にgoと同時に色を指定できるメッセージがあるとする、距離と同時に色を指定する必要があり、そのメッセージは「go:color:」のようになるでしょう。実際の呼び出しは

```
obj go: 100 color: #red
```

のような感じになります、「#red」はSmalltalkのシンボルの表記です。これはほかの言語のキーワード引数に似ていますが、省略可能ではありませんし、順序を変えることもできません。「go:color:」という一つのメッセージが分割されて表記されていると考えるべきです。Smalltalkでは、制御構造を含めたほとんどすべてがこのメッセージ送信によって実現されているのが特徴です。

RubyはSmalltalk似？

Smalltalkは、発表された年度に応じて、Smalltalk-72、Smalltalk-76、Smalltalk-80という三つのバージョンがあります。現在 Smalltalk といえば、最後のバージョンである Smalltalk-80(およびそこから派生)を意味しますが、バージョンが進むごとに、大人が使いこなすプログラミング言語と環境に成長しています。「子供のため」とされた絵文字の使用などは影を潜めています。

何年も前に Smalltalk 開発リーダーだった Alan Kay 氏と昼食を一緒にしたときに、「Smalltalk はなんだか Lisp の影響が強くなってきて、当初の構想とはズレてきたんだよねえ」とおっしゃっていました。また、「Ruby は Smalltalk-76 の頃にちょっと似てるよ」とも。Smalltalk-76 は Xerox PARC の外にはあまり公開されていなかったこともあって、あまり資料が残っていません。どの辺りが Ruby に似ているのか、私にも正確には把握できていないのですが、なかなか興味深い発言です。

Actorのオブジェクト指向

Simula のシミュレーション指向、Smalltalk のオブジェクト指向の影響を受けて、米マサチューセッツ工科大学 (MIT) の Carl Hewitt 氏が 1973 年に考えついたのがアクターモデルです。

アクターモデルは、個々のオブジェクトが主体的に計算をして、オブジェクト間の通信はメッセージによって行われるというものです。Smalltalk のメッセージ送信は、結果が帰ってくるのを待つので「同期的」です。一方のアクターの場合は非同期で、メッセージは一方的に送りっ放しになり、結果は別のメッセージとして送り返されることになります。

アクターモデルは「数百・数千のマイクロプロセッサから構成され、個々にローカルメモリーを持ち、高性能通信ネットワークで通信する並列コンピュータが近い将来登場するとの予測」から誕生しました。実際には 1973 年から見て「近い将来」にそのようなコンピュータが誕生することはなく、アクターモデルはすぐには普及しませんでした。しかし、それから 40 年以上後の現在では、マルチコアやメニーコア、クラウドなどによって、Hewitt 氏の予測がようやく現実に近い近づいています。

Erlangもアクターモデル

アクターモデルを提供する言語も増えてきていて、例えば Erlang などはアクターモデルを設計の中心に据えた言語といってもよいでしょう。アクターモデル自身はオブジェクト指向の影響を強く受けて発案されたのにもかかわらず、Erlang の作者である Joe Armstrong 氏は以前「オブジェクト指向なんてダメだ」と発言していました。しかし最近になって「Erlang のプロセスはオブジェクトであり、Erlangこそが真のオブジェクト指向であることに気が付いた」と 40 年ぶりに Hewitt 氏のアクターモデルを再発見したような発言をしているのが興味深いです。

CLOSのオブジェクト指向

Lispは非常に柔軟性の高い言語で、プログラミングにおける新しい機能を実験するのに最適です。オブジェクト指向プログラミングについても例外ではなく、SimulaやSmalltalkによってオブジェクト指向プログラミングという概念が発明された直後から、さまざまなオブジェクトシステムが各種Lisp処理系の上で実験されました。それらのオブジェクトシステムの集大成に当たるのがCommonLisp Object System (CLOS) です。

ほかのオブジェクト指向言語と比較したときのCLOSの特徴は、

- 多重継承
- 特異メソッド
- マルチメソッド
- メソッド結合

です。

オブジェクト指向プログラミングにおける継承とは、既存のクラスから機能を受け継ぎ、機能を追加・改変することによって新しいクラスを作ることです。

RubyやJavaを含む多くの言語では継承で機能を受け継ぐクラスを一つしか指定できません。これを単一継承（または単純継承）と呼びます。

「多重継承」とは、一つではなく複数のクラスから継承することで、CLOSやC++で可能です。継承先を一つから二つ以上にするのは自然な拡張といえます。しかし、実際にはそんなに簡単なことではなく、二つ以上のクラスから継承すると、メソッドの名前が衝突したり、クラス階層が単純な木構造からネットワークになったりと課題が山積みです。JavaやRubyはそういうのを嫌って単一継承を採用しているわけです。

一方CLOSでは、複数のクラスから継承したときに問題が起きにくいようなやり方として「Mix-in」を考案したり、矛盾を解消する仕組みを導入したりしています。CLOSのMix-inは多重継承の使い方における「紳士協定」のようなものでしたが、Rubyではモジュールのincludeという形で、言語機能として特別扱いされています。

「特異メソッド」とは、クラス単位ではなくある特定のオブジェクトに対して定義されるメソッドのことです。CLOSでは引数のクラス名の代わりに

(eql 値)

と指定することで、そのオブジェクト(値)に固有のメソッド(ここではeqlという名前)を定義できます。

クラスとは独立したメソッド

「マルチメソッド」は、複数のクラスに所属するメソッドです。多くのオブジェクト指向言語では、メソッドはクラスに所属して、そのクラスのオブジェクトに対してメソッドを呼び出す形になっています。ところが、CLOSではメソッドは関数に所属して、その引数すべてのクラスに応じて適切なメソッドが選択されます。

これは実例を見ないと分かりにくいでしょう。まず、CLOSのメソッド呼び出しは通常の関数呼び出しと全く同じ外見を持っています。

```
(length obj)
```

とすると、lengthという名前の関数（複数のメソッドをまとめた関数なので総称関数と呼ばれます）に所属する複数のメソッドのうち、objのクラスに合わせたものが実行されます。ほかのオブジェクト指向言語で言えば

```
obj.length()
```

という呼び出しと同じですが、順序が違うというわけです。ところが複数の引数を取ると、変わったことになってきます。例えば

```
(plus obj1 obj2)
```

という、足し算（plus）の機能を持った総称関数の呼び出しがあったとすると、どのメソッドが呼ばれるかということは、すべての引数のクラスによって決まります。obj1とobj2が整数、浮動小数点それぞれのすべての組み合わせごとに別々のメソッド定義を持てるのです。これにより引数の型によって分岐する必要がなくなり、より適切な手続きを選べます。C++やJavaのメソッドオーバーロードにちょっと似ていますが、マルチメソッドによる選択はあくまでも通常の（第1引数による）メソッド選択と同様に動的に行われます。

これはコペルニクスの転回で、オブジェクト指向言語にしばしば見られる、クラスがあってそれに属するメソッドがある、という構造が全くなってしまう。その代わりに、クラスがあり、それとは全く独立にメソッドがある、という構造になります。これをオブジェクト指向と呼べるのかという議論もあったようですが、マルチメソッドには引数による分岐を自動化できるし、既存のLispとの整合性も高いということで、Lisp界では受け入れられたようです。しかし、Lisp以外の言語でマルチメソッドを採用したものはほかにほとんどありません。例外的に採用しているのは2015年12月によりやく正式リリースされたPerl6です。

大掛かりなメソッド結合

CLOSの特徴で最後のものが「メソッド結合」です。これは同じ名前で適用可能なメソッドが複数あるとき、それらをどのように組み合わせて呼び出すかという仕組みです。

多くのオブジェクト指向言語では、メソッド中に `super` のような名前を使ってスーパークラス（継承元）のメソッドを呼び出せます。しかし、CLOSでは多重継承の問題もあって、そんなに簡単に解決できることばかりではありません。そこでCLOSはメソッドの組み合わせ方さえ自由に定義できるようにしてしまいました。なんという柔軟性。

例えば、標準のメソッド結合方式では、メソッド呼び出しは次の順序で行われます。

まず、呼ばれた総称関数に所属するメソッドのうち引数に対して適用可能なものを優先順位ごとにソートします。この場合の優先順位とは、1は「整数」であり、「整数」は「数」のサブクラスであるとしたら、整数を取るメソッドの方が数を取るメソッドよりも優先度が高い（Lisp仕様の「より特定のである」）と見なして計算します。

そして、まず「`:around`」というタグが付いているメソッドを（存在するならば）特定のなものから順に呼び出します。図示すると図3のような順序になります。aroundメソッド内部では

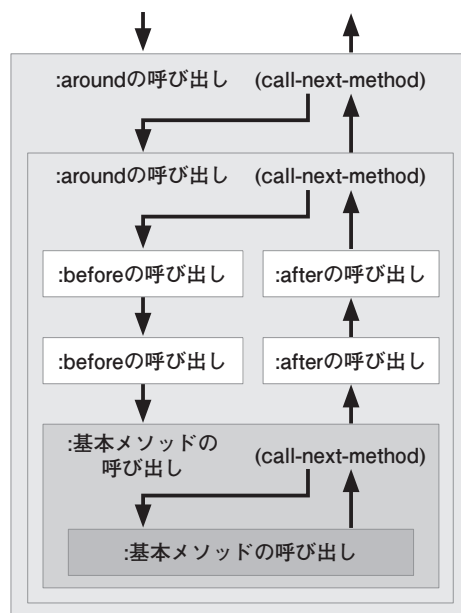


図3 CLOSの標準メソッド結合

```
(call-next-method)
```

という形式で優先順位が次のメソッドを呼び出せます。それ以上 `around` メソッドが存在しなければ次のステップに進みます。

次に「`:before`」というタグの付いているメソッドを特定のなものから順に呼び出します（戻り値は捨てる）。それからタグの付いていないメソッドで最も特定のものを呼び出します。`call-next-method`で次に特定のメソッドを呼び出せるのは `around` メソッドと同様です。

最後に「`:after`」というタグが付いているメソッドを特定のでないものから順に実行します（戻り値は捨てる）。

`after` メソッドの実行が終わると、`around` メソッドの実行に戻り最後まで実行されます。最終的な戻り値は最も特定の `around` メソッドのものになります。

このメソッド結合はアスペクト指向プログラミングの基礎になりました。実際、Javaに対するアス

ペクト指向プログラミング処理系であるAspectJの開発者は、CLOSの設計者（の一人）であるGregor Kiczale氏と同一人物です。

Rubyにも機能を取り入れた

CLOSのオブジェクト指向機能は、正直言うとかなり大掛かりでほとんどの場合にはオーバースペックで、すべての機能を使う機会はほとんどないでしょう。しかし、オブジェクト指向プログラミングの歴史の中でもこれほど深く考慮され、大胆に設計された言語はほとんどありません。また、CLOSが提供する、ほかのオブジェクト指向言語には取り入れられなかった「一風変わった」オブジェクト指向機能は、ある意味ロストテクノロジーとして、今後のオブジェクト指向言語の設計に大いに参考になるものもたくさんあります。事実、Rubyが提供しているMix-inや特異メソッド、またModule#prependなどの機能はCLOSを参考に設計しています。

現代ではCLOSはあまり広く使われていませんが、今後人気が上昇する可能性はゼロではありません。そうならなくても、そこで培われた機能は新たな言語に取り入れられていくかもしれません。

C++のオブジェクト指向

さて、同じオブジェクト指向言語と名乗っていても、かなり異なった背景を持っているのがC++です。C++はCに対してオブジェクト指向機能を追加した言語で、オブジェクト指向言語の多くが影響を受けているSmalltalkの影響が驚くほど見られません。例えば用語一つとってもC++ではスーパークラスとは呼ばず基底クラス（base class）と呼び、サブクラスではなく導出クラス（derived class）です。何か文化の違いのようなものを感じます。

私は2004年に、C++の設計者であるBjarne Stroustrup氏と直接話す機会がありました。2003年にデンマークで開催されたJAOOカンファレンス^{*1}でお目にかかって「私の学生はC++ばかりでRubyのことを知らないから、ちょっと話してくれ」と誘われました。それで彼が教授を勤める米テキサスA&M大学で講義することになったのです。

そのときにC++の起源についてお伺いしました。いわく、「英ケンブリッジ大大学院生のときに論文のためのシミュレーションを使うのに、学部時代に使ったSimulaが使いたかったが、当時は遅くて使いものにならず、BCPL（Cの前身）を使った。後に英AT&Tベル研究所に就職してから、使いものになるSimulaを目指してC with Classという言語を作り、これが後のC++になった」ということでした。

つまり、C++はSmalltalkの影響をあまり受けないSimulaの直系の子孫であり、それだからSmalltalkの影響を受けた用語をわざと避けたということがあるようです。また、柔軟性のために

^{*1} JAOOカンファレンスは2001年と2003年の2回出席しましたが、2001年は本文中で紹介したNygaard教授との出会いもあり、さらにちょうど会期中に9.11のテロ事件があって記憶に残るカンファレンスでした。また、Ruby普及の立役者であるDave Thomas氏に初めて会ったのもこのときです。2003年のときにはStroustrup氏にお会いして、大学に招待されたこと、MVCモデルの生みの親であるTrygve Reenskaug氏にお会いしたのがハイライトでした。

性能を犠牲にしがちなオブジェクト指向言語が多い中、性能に徹底的にこだわっているところも、Stroustrup氏の「Simulaは遅かった」という経験に起因していると考えられます。

当初のC++には、例外もなく、多重継承もなく、テンプレートもなく非常にシンプルな言語で、静的型を持つオブジェクト指向言語としてはやや実用性に欠けるところがありました。しかし、それ以降の進歩はめざましく、今では単なるオブジェクト指向言語を越えて、テンプレートを活用したジェネリック指向プログラミング言語と呼んでもよいような存在になってきました。ただ、たくさんの機能を持つ複雑な言語という側面もあります。

C++はベースとなったCが目指した構造化プログラミングの延長線上にオブジェクト指向を定義した言語です。オブジェクトという実体とメッセージ送信というモデルでプログラミングを表現しようとしたSmalltalkとは双璧をなす存在です。かつてのオブジェクト指向論争は、この二つの視点がお互いを十分に理解しなかったことから発生したのではないかと考えます。

Javaのオブジェクト指向

Simula、Smalltalk、CLOS、C++について説明してしまったので、私個人としてはもう満足なのですが、一応そのほかの有名なオブジェクト指向言語についても簡単に解説しておきます。

JavaはC++的オブジェクト指向をほんの少しSmalltalk寄りにしたようなポジションのプログラミング言語です。C++のように積極的にSmalltalk用語を避けるのではなく、普通にスーパークラスとかサブクラスとか呼んでいますし。

オブジェクト指向プログラミング言語として見た場合の特徴は、C++に似ていながら、Lispで見られる（そしてC++が性能上の理由から避けていた）ガーベージコレクションを積極的に導入したことが挙げられます。またインタフェースによって、「仕様の多重継承」を許しながら、「実装の多重継承」は許さないストイックな姿勢も印象的です。

Rubyのオブジェクト指向

我らのRubyについても少しだけ。思想的にはRubyもJava同様C++とSmalltalkの中間に位置する言語です。とはいえ、JavaよりもはるかにSmalltalk寄りです。RubyではC++よりもメッセージ指向の色合いが濃くなっていて、例えば、Rubyの持っているmethod_missingという機能は、SmalltalkのDoesNotUnderstandメッセージで実現されているもののコピーです。

RubyはSmalltalkだけでなくCLOSからも影響を受けていて、特異メソッドやMix-inなどの機能を受け継いでいます。しかし、CLOSの持つ機能でもマルチメソッドやメソッド結合のような大胆で大規模な機能は取り込んでいません。というのも、Rubyは実験的な言語ではなく、実用的なオブジェクト指向言語を目指したため、ユーザーが混乱するような機能はできるだけ避けようという保守的な設計判断をしたためです。

オブジェクト指向も「当たり前」になった？

2015年11月号掲載分です。今回は完全な脱線回で、歴史を踏まえた上でさまざまな視点からオブジェクト指向について語っています。その一方でStream言語は全く登場しません。

オブジェクト指向については、その概念についてかなり混乱があって、多くの人が厳密な定義の合意なしに議論するものですから、議論が発散したり、荒れたりしがちです。まあ、個別の言語と違って、オブジェクト指向というのは単なる考え方の指針程度のものでし、それもそれぞれの言語によって微妙に異なっていたりするので、背景が異なる人とはなかなか話が合わないのも当然かもしれません。

そうは言っても、議論が進展しないのは不毛ですから、たまにはこうして俯瞰して考えるのも良いことなのではないかと思います。今回の解説も、できるだけどれか一つの考えに固執することなく、平等に扱うように心がけました。もっとも、取り上げている言語のうちの一つを私自身が設計しているという点で完全な平等は不可能でしょうが、それでも、言語設計者としてよりも言語オタクとしての立場をとって執筆したつもりです。

一方で、ここ何年かはオブジェクト指向についての議論を目撃していないのも事実です。最近ではむしろ関数型プログラミングとオブジェクト指向プログラミングとの対比などの議論を見かけることの方が多くなりました。

これもオブジェクト指向プログラミングが「当たり前」になってきたからではないかと思います。数十年前に「構造化プログラミング」が大いに話題になった時期もありましたが、今となっては常識になってしまい、全く話題にならなくなりました。それと同じことがオブジェクト指向プログラミングにも起きつつあるのでしょうか。長年のオブジェクト指向プログラミングファンとしては寂しい気持ちもあります。

3-2

Stroom のオブジェクト指向

3-1では歴史を踏まえて各種プログラミング言語のオブジェクト指向機能を見てみました。今回は、いよいよStroomのオブジェクト指向機能を設計します。適用するデータの種類によって適切な処理を選択する「動的結合」を実現しましょう。

さて、いよいよStroomのオブジェクト指向機能を設計します。ただ、関数型言語の影響の強いStroomでは、そのままほかの言語のオブジェクト指向機能を持ってくるにしても使いやすくなりそうにありません。まずは、Stroomの特徴を復習し、それにふさわしいオブジェクト指向機能について考えてみましょう。

Stroomでは動的結合に価値

Stroomの最も重要な特徴は、ほとんどの「オブジェクト」がイミュータブル、つまり更新不可であるということです。ほとんどのオブジェクト指向プログラミングでは、オブジェクトの属性（インスタンス変数など）を書き換えることによって、計算処理を行います。つまり、状態をオブジェクトに閉じ込めることによって扱いやすくしようという意図です。3-1で解説した最古のオブジェクト指向言語「Simula」でも、シミュレーションの状態を管理するためにオブジェクトを導入し、複数の同種オブジェクトの振る舞いをまとめて定義するためにクラスを導入していました。

ところが、関数型言語の影響を受けたStroomでは値を後から変更することはできません。ということは「状態の管理」がそもそも発生しないわけです。この状態管理の不在、または副作用が存在しないという条件があるなら、伝統的なオブジェクト指向の必要性は高くないことになります。

では、Stroomのような言語ではオブジェクト指向プログラミングはうれしくないのでしょうか。

そんなことはありません。オブジェクト指向プログラミングのいくつかの特徴のうち、「動的結合」は副作用の存在に関係なく、うれしいものです。

動的結合とは、適用するデータの種類によって適切な処理が選択されることです。例えば、aという変数に格納されているデータが、文字列であっても、配列であっても「length」というメソッドを呼び出すと、その長さを求められます。文字列の長さを求める処理と、配列の長さを求める処理は、内部実装が全く異なるでしょう。しかしプログラマはその実装を気にすることなく「長さを求めたい」という抽象的なレベルで考えることができます。Stroomにオブジェクト指向機能を導入す

るのであれば、何よりもこの動的結合こそが必要なものです。

総称関数

そこで、関数型言語的な性質と相性が良い形で動的結合を導入するため、総称関数を導入することにします。総称関数とは(3-1でも解説しましたが)、CommonLispなどで採用されている、引数のデータ型に応じて内部処理を選択する関数のことです。つまり、データの長さを求めたいときには

```
length(a)
```

とすれば、aのデータ型に応じて適切な長さを求める処理(メソッド)が選択されます。CommonLispでは最初の引数だけでなく、すべての引数の型に応じてメソッドが選択されることになっていますが、Streamでは(手を抜いて)当面は最初の引数の型だけでメソッドを確定させようと考えています。

総称関数は、関数呼び出しという形式を維持したまま、動的結合を実現できる優れたアイデアです。

メソッド定義の方は、

```
def length(a:array) { ... }
```

という風に引数で型を指定します。これによって、もしlengthという総称関数が存在していなければ、それを作成し、そしてその総称関数にarray型に対応するメソッドを登録することになります。型を省略した場合はどんな型でも受け付けるという意味になります。

メソッド定義自身も副作用になり得るので、Streamではトップレベルでしかメソッド(や、後述するネームスペース)を定義できないことにします。Rubyのできる条件付きメソッド定義のようなことは許しません。

クラス機能追加の副作用を限定

Rubyではクラスに後から機能を追加することが可能です。そのことを利用して既存のクラスの機能を拡張する「モンキーパッチ」と呼ばれるテクニックが広く使われています。「Ruby on Rails」を使っている人がお世話になっている「ActiveSupport」というライブラリは、このモンキーパッチを使って、既存のクラスにさまざまな便利機能を追加しています。これにより、

```
2.days.ago
```

といった素のRubyとはかなり異なったプログラミングが可能になっています。

しかしモンキーパッチは便利な反面、副作用もあります。クラスに後から無制限にメソッドを足していくと、同じ名前で作動が異なるメソッドを複数のライブラリで追加することがあり得ます。そうすると、両方のライブラリを使おうとした途端に思わぬ問題が発生します。

そのような事態を避けるための仕組みがRuby2.0で導入したRefinementという機能です。Refinementとは、要するにある特定のスコープでだけクラスにメソッドを追加する機能です。

図1にRefinementの使い方を示します。

お分かりでしょうか。usingで指定されたスコープの外では、クラスは変更されことなく元の振る舞いを保っています。しかし、usingで指定されたスコープ(usingが登場してからそのファイルの終わりまで)では、Refinementが有効になり、追加された機能が見えるようになります。この切り替えは静的なものであり、同じオブジェクトでも、スコープの内外では異なる振る舞いをするようになります。これで、名前の衝突やメソッド追加の副作用などについて悩まずに済みます。

実装が難しかったRefinement

まだまだ荒削りで今後の機能拡張が予想されるRefinementですが、今後、モンキーパッチを置き換えていくことになるでしょう。Refinementという機能はまだまだマイナーで、類似の機能を提供している言語はほとんどありません。JavaにClassboxという名

```
# 元のクラス fooというメソッドだけを持つ
class Foo
  def foo
    p :foo
  end
end

# Refinementの単位となるモジュール
module FooRefine
  # Fooクラスをrefine (拡張)
  refine Foo do
    # fooメソッドを拡張する
    def foo
      p :foo_refine
      # superで元のメソッドが呼べる
      super
    end

    # 新しいメソッドを追加
    def bar
      p :bar
    end
  end
end

# Fooクラスのオブジェクトを作る
f = Foo.new

# Fooクラスのfooメソッドを呼ぶ
f.foo # => :foo

# Fooクラスにはbarメソッドは定義されてない
# f.foo.bar # Error! メソッドがないから

# usingでRefinementが追加される
using FooRefine

# ここからは拡張されたfooとbarが有効
f.foo # => :foo_refine\n:foo
f.bar # => :bar
```

図1 RubyのRefinement

前で類似の機能を拡張した処理系と、ある種の Smalltalk 方言が Selector Namespace という名前で同じような機能を提供しているだけです。

加えて、Objective-C の「カテゴリー機能」や、C# や Swift の「クラス拡張」は、Refinement に似ているといえないこともないでしょう。

いざ、この Refinement を実装しようとする、なかなか困難が付きまといます。静的型の情報を使って実現できる C# や Swift と異なり、Ruby ではすべてを実行時に解決する必要があります。とはいえ Refinement を導入することを理由に性能が低下することは許されません。実際、Refinement を Ruby に入れたいと発言してから実際に導入されるまで 10 年近くかかったのは、効率的な実装が思い付かなかったからです。前田修吾さんが効率良い実装を考え付いてくださるまでそれだけの時間が必要でした。

総称関数と Refinement

ところが、総称関数を採用すると Refinement と同じことが意外なほどシンプルに実現できます。

総称関数は外側から見ると単なる関数として扱えます。ならば Refinement 相当は、スコープによって同じ名前でも別の関数を呼び出せることに相当します。これは関数呼び出しがある多くの言語で「スコープ」とか「ネームスペース」とかいう名前で普通に実現できていることです。

そこで、Stream にもネームスペースという概念を導入して、Refinement 相当を実現します。Stream のネームスペースの文法は図 2 のようになります。

```
namespace文  = namespace <名前> {
                文...
            }
import文     = import <名前>
def文       = def <名前>(<名前>[:<名前>],...) {
                <文>...
            }
名前        = [A-Za-z_][A-Za-z0-9_]*
```

図2 Streamのネームスペース文法

```
# namespace文でtest_nsというnamespaceを定義
namespace test_ns {
  # importでほかのネームスペースを取り込める
  import development_ns
  # development_nsで提供される変数・関数が見える

  # 関数定義
  def print(message) {
    puts(stderr, message)
  }
}
```

図3 Streamのネームスペース例

文法定義をいきなり見せられても分かりづらいでしょう。実際のプログラム例を図 3 に示します。期待される動作を注釈で記しました。

ネームスペースを使えば、ある特定のスコープだけで見える関数を作るのは簡単です。それはつ

まり、Refinementのような実装が難しい機能を導入しなくても、そのスコープだけで有効なメソッドの定義（SwiftやC#のクラス拡張で可能なこと）と、そのスコープに限ったメソッドの振る舞いの一時的な変更（Refinementなら可能なこと）の両方ができるということでもあります（図4）。

```
# グローバルな関数foo
def foo(a) {
  # 引数を文字列化して出力
  puts(to_s(a))
}

# 「クラス拡張」用のネームスペース
namespace extend {
  # 既存の関数をオーバーライド
  def foo(a:string) {
    puts("foo\n")
    # オーバーライドした関数を呼ぶ
    super(a)
  }
}

# このネームスペースでだけ有効な関数
def bar(a) {
  puts("bar\n")
}

import extend
foo("a") # extendのfooが呼ばれる
bar("b") # extendのbarが呼ばれる
```

図4 総称関数とネームスペースによるクラス拡張

名前の衝突

import文を使うと、あるネームスペースにほかのネームスペースから機能を取り込むことができます。しかし、複数のネームスペースをimportしたとき、それぞれのネームスペースが同じ名前を保持していると名前の衝突が起こります。衝突が起きると、どちらを参照してよいのか分からないので矛盾が発生します。

この矛盾を解消する方法はいくつかあります。最もシンプルな方法はエラーを出すことです。つまり衝突が起きるということは、機能的にかぶっていることが多く危険なので、そもそも組み合わせて使うべきではないという立場です。

それから、import時に重複する名前をリネームすることで衝突を避けるやり方があります。分かりやすいといえばその通りですが、名称変更された関数を元のネームスペースで呼び出していた場合にどのように処理するかなど考えるべき点も多く、実装が複雑になります。

そのほかにも、衝突した場合にはネームスペースを明示的に指定して呼び出すやり方もあります。リネームをするほど複雑な実装にならないかもしれませんが、簡単とまではいえそうにありません。

この点をどうするかについては、だいたい悩みましたが、当面はシンプルさと分かりやすさを優先して、衝突したらエラーとすることにしました。Rubyの経験から考えても複数のネームスペース（Rubyの場合はモジュール）から引き継いだ名前が衝突して、それを解消すべき状況はそんなに頻繁には発生しないと考えられるので。今後、エラー以外の名称衝突解消の方法が絶対に必要にならないと考えているわけではありません。衝突を解消することがどうしても必要な具体的な事

案が発生してから考えることにします。

Streamのオブジェクト

さて、オブジェクト指向プログラミングであれば、クラスを定義したり、オブジェクトを作ったり、それに対してメソッドを呼び出したりしたいものです。

メソッド呼び出しは総称関数で実現することにしたましたが、クラスやオブジェクトについてはどうしましょう。これまでの解説で述べてきたように、似たようなデータ構造を複数導入したくないので、できれば新たに「オブジェクト型」のようなものの導入は避けたいところです。

ここで、ほかの言語を参考にしましょう。Luaという言語では、テーブルというデータ構造があり、これをオブジェクトの代わりにします。Perlもハッシュをオブジェクトとして活用しています。また、JavaScriptにはオブジェクトというデータ型がありますが、その実体は単なるハッシュテーブルです。

ということは、既存のデータ構造にちょっと工夫をすることで、オブジェクト（相当）を表現できるということです。Streamには配列があり、これが普通の言語におけるハッシュテーブルの役割もカバーしています。この配列をオブジェクトとして取り扱えそうです。

Luaではテーブルをオブジェクトして扱うために、メタテーブルというものを設定します。メタテーブルはオブジェクトに対する各種操作を格納したテーブルで、いわばクラスの役割を果たしています。

Perlでは、スカラー型のデータにパッケージをblessすることでオブジェクトとして扱えます。つまり、そのオブジェクトに対する操作はパッケージに含まれる関数が引き受けるということです。

さて、Streamではどうしようかと悩んだあげく、Perlのやり方にヒントを得て、配列をオブジェクト相当として活用することになりました。つまり、こういうことになります。

```
new <名前> [値...]
```

として呼び出すと、<名前>で指定されたネームスペースを「クラス」として関連付けられた配列が与えられます。つまり、

```
new Foo [1,2,3]
```

は3要素の配列にFooという名前のネームスペースを関連付けたものを返すということです。この配列を第1引数とする関数呼び出しを特別扱いすることでオブジェクト指向機能を実現します。

ネームスペースをクラスとして活用することでオブジェクト指向機能を実現したいわけですが、そのために、関数呼び出しの処理を拡張します。拡張した関数呼び出しの処理を図5のフローチャートに示します。これにより、関数の第1引数がnewで作られた配列だった場合、そのネームスペースの関数が呼び出されます。

これで総称関数とネームスペースを使った比較的シンプルなオブジェクト指向システムを作ることができました。

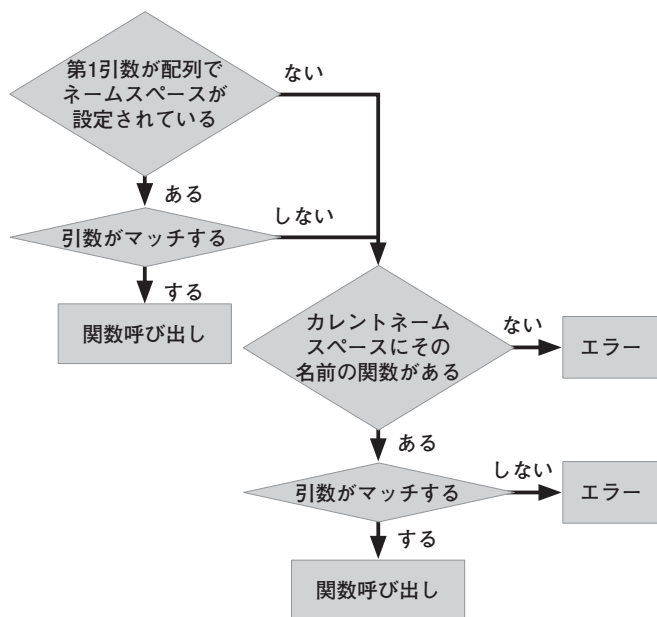


図5 関数呼び出しフローチャート
「引数がマッチする」とは引数の数と型が関数
定義と一致すること。

メソッドチェーン

とはいうものの、関数スタイルがいつも最適であるとは限りません。複数の関数呼び出しを連ねていくスタイルでは、関数呼び出しの順番とプログラム上登場する順番が逆になってしまうことがあります。

例えば、配列要素を条件（偶数である）によって選別し、それをソートして、重複した要素を取り除くという処理を関数呼び出しスタイルで記述すると図6 (a) のようになります。行いたい処理の順番（選別→ソート→重複削除）に対して関数の登場順が逆になっています。

(a) 関数スタイル
`uniq(sort(filter(ary, {x -> x % 2 == 0})))`

(b) メソッドスタイル
`ary.filter{x-> x % 2 == 0}.sort.uniq`

図6 関数スタイルとメソッドスタイル

これがRubyのようなメソッド呼び出しスタイルだと図6 (b) のようになります。メソッドが登場する順序が実際の順序と一致しており、分かりやすくなっています。また、日本語における「選別して、ソートして、重複削除して」というような動作を連ねる表現にも合致します。

ほかの言語でもそのように感じることは多かったのか、いくつかの言語では関数呼び出しを連鎖できるような記法を提供しています。例えば、Haskellでは「\$」、Elixirでは「|>」を使います。こ

れらを使うと図7 (b) のようになります。

どうでしょう? メソッド呼び出しスタイルとほぼ同じような感じでプログラミングできるようになりました。これは記法として大変便利なので、Streamでも導入します。Streamでは「\$」の代わりに「.」を使います。そうすると、なんと見かけ上は普通のオブジェクト指向言語のコードのように見えるようになります (図7 (c))。

```
(a) 関数スタイル
uniq(sort(filter(ary, {x -> x % 2 == 0})))

(b) $記法
ary $ filter{x-> x % 2 == 0} $ sort $ uniq

(c) .記法 (Streamで採用)
ary.filter{x-> x % 2 == 0}.sort.uniq
```

図7 関数スタイルと、関数呼び出しの連鎖記法

Lisp-1とLisp-2

関数型言語の先祖といってもよいLispには、関数の扱いにおいて2種類の流儀があり、それぞれLisp-1とLisp-2と呼ばれます。

Lisp-1はSchemeと呼ばれるLisp方言で採用されている考え方で、関数の名前空間と変数の名前空間に区別がなく、名前空間が一つしかない (だからLisp-1) ものです。Lisp-1では、

```
(print "hello")
```

は「printという名前の変数に格納してある関数を"hello"を引数として呼び出す」という意味に解釈され、

```
print
```

は関数への参照になります。

一方、Lisp-2はCommonLispなどで採用されている考え方で、関数と変数で名前空間が分離されているので、二つ (以上) の名前空間があるものです。Lisp-2では

```
(print "hello")
```

は「printという名前の関数を"hello"を引数にして呼び出す」ので、表面上の動作はLisp-1と同じに見えます。しかしprintの処理をする関数は

```
print
```

という変数を参照しても取り出すことはできなくて、特別な形式を用いる必要があります。
CommonLispの場合には

```
(function print)
```

またはその省略形である

```
#`print
```

という記法を用います。

Lisp-1とLisp-2の違いは些細なように思えるかもしれませんが、実は一長一短があります。
Lisp-1の利点は関数を値として取り出しやすく、関数型プログラミングを実践しやすいという点です。
Lisp-1なら

```
((func args) arg)
```

のように関数を返す関数からの戻り値に引数を渡して呼び出すことも簡単で、これを使ってオブジェクト指向システムを作るのも容易です。
Lisp-2ではこうはいきません。

一方Lisp-1では、関数呼び出しに必ず関数への参照が伴うので最適化をしにくいというのは欠点とも呼べるでしょう。
Lisp-2はそのちょうど裏返しになります。

この考え方はLisp以外にも当てはめられます。実際JavaScriptやPythonはLisp-1であり、RubyやSmalltalkはLisp-2です。
Lisp-2を採用する理由は、メソッド呼び出しの最適化のためであり、その分、メソッドをオブジェクトとして取り出すのは若干手間がかかります(図8)。

```
# Python (Lisp-1)
obj.foo(1)          # メソッド呼び出し
f = obj.foo          # メソッド取得
f(1)                # メソッド間接呼び出し

# Ruby (Lisp-2)
obj.foo(1)          # メソッド呼び出し
obj.foo              # メソッド呼び出し (引数なし)
f = obj.method(:foo) # メソッド取得
f.call(1)           # メソッド間接呼び出し
```

図8 PythonとRubyのメソッド取得

注意すべき点は、Lisp-2であるRubyではメソッドを取り出すためにmethodメソッドを使う点と、取り出したメソッドオブジェクトを呼び出すときにcallメソッドを使う必要がある点です。

図8のプログラムだけ見ると、PythonのLisp-1的やり方の方がシンプルで優れているように見えます。しかしRubyの方にも、メソッド呼び出しの最適化の余地が大きく、実際メソッドキャッシュ

などが実装されている点と、引数のない属性参照に見える形のメソッド呼び出しが可能である点にメリットがあります。

で、Streamはどうするかというと、Lisp-1とLisp-2の中間のような、いわばLisp-1.5^{*1}とでも呼ぶべきやり方にしようと思います。つまり、関数名を直接指定したときには関数への参照を得ることができ、「.」を使ったメソッドチェーン形式では引数のないメソッド呼び出しとみなします(表1)。

書式例	意味
func	関数funcの参照
func()	関数funcの呼び出し
a.b()	関数bの呼び出し(b(a))
a.b	関数bの呼び出し(b(a))
&func	funcと同じ意味
&func(1)	関数funcの部分適用
&a.b	関数bの部分適用(&b(a))
&a.b(c)	(&a.b)(c) すなわち b(a,c)

表1 Streamの関数呼び出し

さらに「&」記号を用いることで関数の部分適用を得られます。部分適用とは、関数引数の一部を埋めた関数のことです。といっても、言葉では少々分かりにくいかもしれませんので、実例を見ながら解説しましょう。

例えば、二つの数値を足し合わせるplusという関数があったとしましょう。

```
plus(1,2) # => 3
```

は1と2を足した3を返します。さて、ここで、

```
plus5 = &plus(5)
```

とすると、plus5に代入されるのは「plusの最初の引数を5を固定した関数」です。このplus5に引数を渡すとその数値と5を加えた値を返します。

```
plus5(1) # => 6
plus5(9) # => 14
```

ここでは引数を一つだけ指定しましたが、もちろん二つ以上指定することも可能です。

```
plus12 = &plus(1,2)
plus12() # => 3
plus12(3) # => ERROR! 引数多い
plus123 = &plus(1,2,3)
# ERRRO! 引数多い
```

*1 本当にこれをLisp-1.5と呼ぶと、初期のLisp処理系と名前がかぶるのでよくはないのですが。

この形式はドット記法にも有効です。「f = &a.b」は「f = &b(a)」の省略記法であり、「f(c)」と呼び出すと「a.b(c)」、言い換えると「b(a,c)」という関数が呼び出されます。

ま と め

今回は Stream のオブジェクト指向機能を設計しました。CommonLisp の総称関数や Haskell の \$ 記法などを既存の言語のアイデアを拝借しつつも、それをうまく組み合わせることでユニークなものを設計できたのではないのでしょうか。また、「言語設計ってこうやってやるんだ」という点においても（もしかしたら）参考になるかもしれません。

今回解説した部分のソースコードは「<https://github.com/matz/stream>」で「201512」というタグを打っておきます。ただし、実装の難易度から「&」による関数の部分適用をはじめとしていくつかの部分は手付かずです。ほかの部分もまだまだ未完成ですが、参考までに。

タイムマシンコラム

もっと良いデザインがあるはず

2015年12月号掲載分です。3-1で解説したオブジェクト指向の概念と歴史を踏まえた上で、Streamという関数型の影響を受けた言語におけるオブジェクト指向機能とはどのようなものであるべきかを考察しています。オブジェクト指向プログラミングはもはや常識となってはいますが、Streamのような言語にとって最適なオブジェクト指向機能のデザインは自明ではありません。

今回は総称関数とネームスペースを組み合わせたオブジェクト指向機能をデザインしてみました。とりあえず使い物にはなるという点では素晴らしいのですが、完全に満足しているわけではありません。「&」による部分適用の実装も不完全です。

今回のデザインは関数型プログラミングとオブジェクト指向プログラミングをうまくつないでいるとは思いますが、それでもなおCLOSの総称関数のように名前空間制御によってrefinementと同様のことが実現できるわけではありませんし、完全な融合と呼べるほどではありません。まだ不満が残るので将来もっと良いデザインが実現できたらな、と考えています。

3-3

Strem 文法再訪

いろいろ考えながら設計したStremの文法もしばらく時間をおいて改めて眺めてみると、「とりあえず」と思って設計した文法に不満が出てきました。文法を整備するならば、ほとんど誰もまだStremを使っていない今がチャンスです。Stremの文法を再検討してみましょう。

この原稿のためにサンプルプログラムを書く必要があるのですが、恐らく私は世界で一番Stremでプログラムを記述していると思います。というか、少なくとも現時点ではStremのプログラムを書いている人は、ほかにはいないのではないかと思います。まだまだ実用的なレベルには到達していませんから。

そうやってStremでプログラムを書いていると若干気に入らない点が出てきました。最初に文法を設計したときにはあまり考えていなかったことに気が付くというのはよくあることです。

shift/reduce conflict

まず気に入らない点は、文法の曖昧性です。Stremの文法はYACCというツールのための記法(YACC記述と呼んでいます)で書かれていて、GNUのYACC互換ツールであるbisonでCにコンパイルされます。

本原稿執筆直前のStremの文法定義をbisonにかけると、

```
8 shift/reduce conflicts
```

という警告が表示されます。これは「文法が曖昧なルールが8カ所ある」という意味です。「conflict」(衝突)とは、構文解析時のある文脈で、ある記号が来たときに、遷移すべき状態が複数ある状態を指します。そのルールが終了する(reduce)のか、まだ続く(shift)のか分からない場合に

```
shift/reduce conflict
```

と呼びます。いずれにしてもルールは終了するのだけれど、遷移すべき状態が複数あるときには

```
reduce/reduce conflict
```

というエラーが発生します。例えば、

```
expr + expr * expr
```

という式が存在した場合、この式を

```
(expr + expr) * expr
```

と解釈するのか、あるいは

```
expr + (expr * expr)
```

と解釈するのかは、自明ではありません。そこでyaccは「どちらか分かりませんよ」という意味でshift/reduce conflictという警告を発します。しかし実際の式では、算術二項演算子は左結合であり、

```
1 + 1 + 1
```

は

```
(1 + 1) + 1
```

と解釈します。また演算子には優先順位があり、「積算演算子(*)は加算演算子(+)よりも優先度が高い」というルールがあるので、本当は曖昧性はありません。そこで、YACC記述の中でそのルールを教えてやると、このconflictを消せます。

まあ、bisonは賢いので多少の曖昧性があってもよしなに解釈してくれる構文解析器を生成してくれます。しかし文法に曖昧性があると、人間も混乱する可能性があります。

そこで、今回のconflictがどこで起きていて、文法のどこに曖昧性があるのか調べてみることにしましょう。yaccに-vオプションを付けると文法をどのように解釈したかというログを「y.output」という名前のファイルに出力してくれます。

```
$ LANG=C yacc -v parse.y
```

このマークで改行

ここでyaccというのはbisonの別名なのですが、bisonはyaccという名前で起動すると「yacc互換モード」で動作します。bisonモードでは構文解析器のソースと出力ファイルの名前が表1のようになります。一つのプログラムで複数の構文解析器を扱うようなことがなければ、yaccモードの方が便利だと思うので、そちらを使っています。

「LANG=C」は出力されるログの日本語表記をやめるために指定します。ログを眺めるときには日本語が便利ですが、ログの中身を検索するときには、いちいち「状態」など入力するよりも、「State」の方が入力しやすいので、むしろ日本語でない方が便利です。もちろん個人の感想ですが。

このようにして生成したy.outputファイルを眺めてみると、先頭の方に図1のような警告が表示されます。そこで例えばState 11を眺めてみると図2のようになっています。ちなみに状態を検索するためには、正規表現で「^state 11」のようになると便利です。

図2を見ると「\n」または「;」でconflictが起きていることが分かります。実はプログラムの構造として、宣言が最初に来て、それから実行文が次に来るような文法にしていたのですが、その間の区切りとなる「\n」や「;」がどちらのルールで解釈すべきかということで曖昧性が発生していたのです。まあ、所詮は区切り文字に過ぎませんし、どちらに所属していたとしても、大差ないので実害がないといってもよいでしょう。しかし、やはり気分が良くないので、ルールを変更することにします。

宣言と実行文

これまでの文法では、宣言文に続いて、実行文が来るという昔のCのようなルールになっていました。しかし、そのような分離にこだわる理由はさほどありません。そこで、これまで宣言とされてきた

モード	bisonモード ^{*a}	yacc互換モード
構文解析器ソース	parse.tab.c	y.tab.c
ログ出力ファイル	parse.output	y.output

*a ソースファイルがparse.yの場合。

表1 bisonモードとyacc互換モード

```
State 0 conflicts: 2 shift/reduce
State 11 conflicts: 2 shift/reduce
State 14 conflicts: 2 shift/reduce
State 51 conflicts: 2 shift/reduce
```

図1 y.outputの警告

```
State 11

4 decls: decl_list . opt_terms
6 decl_list: decl_list . terms decl

';' shift, and go to state 6
'\n' shift, and go to state 7

';' [reduce using rule 106
(opt_terms)]
'\n' [reduce using rule 106
(opt_terms)]
$default reduce using rule 106
(opt_terms)

opt_terms go to state 50
terms go to state 51
term go to state 15
```

図2 conflictのあるstate

- メソッド定義
- ネームスペース定義

を含むものをトップレベル文として、トップレベルであればどこでも置けることにします。これはメソッド定義などを条件文の中や、メソッド定義の中に置けなくなることを意味しますが、それは望ましいことでしょう。

新しく導入したトップレベル文のためのルールを図3に示します（アクションを除く）。

関数を定義するdef文は、結局は関数オブジェクトを変数に代入しているだけなので、トップレベル文ではなくて、通常の文の方に置いています。

それからdef文もmethod文も定義部が一つの式で表現できるときには、「= 式」という形で書けるようにしました。簡潔に書きたいですからね。本当は「=」もなくしたかったのですが、謎のコンフリクトに悩まされたあげくの苦肉の策です。

```
topstmts : topstmt
         | topstmts terms topstmt
         ;

topstmt  : keyword_namespace identifier '{' topstmts '}'
         | keyword_class identifier '{' topstmts '}'
         | keyword_import identifier
         | keyword_method identifier '(' opt_f_args ')' '{' stmts '}'
         | keyword_method identifier '(' opt_f_args ')' '=' expr
         | stmt
         ;
```

図3 トップレベル文のルール

break文の削除

さらに文法を整理しましょう。現在の文法には、関数の実行を中断するbreakとskipという二つの予約語がありますが、重複しているので一つ削ります。breakは元々Cのループを中断する文ですが、Streamにはそもそもループがありません。そこでbreakの方を削ることにします。

字句解析器であるlex.lと構文解析器のparse.yからbreakに関する部分を削るだけです。元々break文に関する処理はまともに動いていませんでしたから、削ってもなんの問題もありません。

ついでにskip文がちゃんと動くようにしておきましょう。skipが呼ばれたら例外が発生して実行を中断するようにします。

if文の変更

この機会に文法で気になる点はまとめて直してしましましょう。

気になっていたことの一つはif文です。Streamの数少ない制御構造の一つであるif文ですが、式を多用するプログラムでは若干見栄えが悪い点になります。例えば、有名なFizzBuzzプログラムを旧文法で記述すると図4のようになります。

```
seq(100) | map{x ->
  if x % 15 == 0 {"FizzBuzz"}
  else if x % 3 == 0 {"Fizz"}
  else if x % 5 == 0 {"Buzz"}
  else {x}} | stdout
```

図4 旧文法でのFizzBuzz

気になる点は二つで、一つはブレース ({ }) が必須である点、もう一つは条件文を囲むカッコが存在しない点です。

旧文法でブレースが必須なのには訳があります。制御構造でブレースを用いる言語、例えばCで発生する問題に「dangling else」というものがあります。これは、

```
if (cond1)
  if (cond2)
    statement2
  else
    statement3
```

という文があったとき、これを

```
if (cond1) {
  if (cond2)
    statement2
  else
    statement3
}
```

と解釈するか

```
if (cond1) {
  if (cond2)
    statement2
}
else
  statement3
```

と解釈するかが自明でなく、コンフリクトが生じることです。yaccはコンフリクトが生じてでもshiftを優先することで「elseは近い方のifに所属する」というルールを実現しているのですが、曖昧性があるのは確かです。

Rubyではこれを嫌って「櫛くし型構文」を導入しています。つまり、そもそも一つしか文がないときと複数文があるときでスタイルが異なるのが問題なので、ブレースなどは使わずif文は対応するendまでと決めてしまうわけです。そうすると、if文全体が

```
if cond1
  statement1
elsif cond2
  statement2
else
  statement3
end
```

というスタイルになり、ifやelseなどが櫛の歯のように飛び出すことから櫛型構文と呼んでいます。

一方、Perlではブレースの省略を許さないことでこの問題に対処しています。当初はStreamもこのやり方をまねてブレースの省略を許さないことにしたのです。

ところが、Streamが多用する関数型プログラムでは、if文が式として値を返すことがしばしばあります。その際には、むしろブレースはない方が式として自然に記述できます。

そこで、Cのスタイルに戻って、単文（または単一の式）ではブレースなし、複数の文をまとめるときにはブレースで囲むという文法に変更することしました。

それに伴い、条件式の周りをカッコで囲むようにしました。旧文法で条件式の周りのカッコを不要にしたのはGoをまねたのですが、関数呼び出しの後ろにブロックが付けられるStreamでは、そのために文法が複雑になってしまっていました（条件式での関数呼び出しではブロックを付けられないように式のために2種類の独立したルールを持っていました）。そのときには頑張ってルールを書いたのですが、しばらくたって冷静になってみるとここまで頑張ってルールを複雑にしても得られるものは多くないなと思うようになりました。この機会にシンプルにしようと思います。

で、if文の変更後の文法でFizzBuzzを記述したものが図5になります。図4と比べると、わずかですがすっきりとした印象があるかもしれません。

```
seq(100) | map{x->
  if (x % 15 == 0)    "FizzBuzz"
  else if (x % 3 == 0) "Fizz"
  else if (x % 5 == 0) "Buzz"
  else                x
} | stdout
```

図5 新文法でのFizzBuzz

若干の曖昧性

今回の変更によって、機械にとっては曖昧でなくても人間には間違いやすい組み合わせが発生しました。例えば

```
if (cond) {print(x)}
```

というif文があった場合、この

```
{print(x)}
```

はprint(x)という関数呼び出しをブレースで囲んだものにも見えますし、定義部分としてprint(x)を持つ関数オブジェクト、つまり

```
{->print(x)}
```

の引数部分が省略されたものであるようにも見えます。このようなケースでは、Streamはif文の本体部分のブレースを文を囲むものとして解釈します。

実はこれは実装が結構面倒で、普通に文法を書く人間同様に構文解析器も混乱してコンフリクトを起こしてしまいます。そこで、Streamの実装では、とりあえず一度関数オブジェクトであるとみなして構文解析をした上で、ifの本体部分であればそれを単なるブレースとして構文木を組み直しています。

右方向代入の追加

気になる点はまだあります。

Streamのプログラムを書いていて、時タイラックとするのは、パイプラインを代入するときです。どういうことかというと、Streamのパイプラインは「|」でつないで左から右へ流れます。これを分岐などのために代入すると、左から右へパイプラインを作っておいて、それを代入するときに左に戻って代入することになり、視線と意識の流れが逆流します。

実例を見ながら考えましょう。図6にあるのは、(p.314 ~の5-5で解説する) 偏差値の計算プログラムです。

ここで注目してほしいのは、図6で(a)のマークが付いた部分です。inputからaverage()関数を通して計算する平均値(のストリーム)をavsという変数に代入しています。意識はinput、average()、変数avsと流れているのに、プログラムでの登場順序はavs、input、average()となっています。(b)のマークが付いた部分も同様です。この意識の流れと記述順の食い違いは微妙な

ストレスとなって生産性を低下させます。

そこで、「=>」という記号を使って左から右への代入を導入することにします。これを使えば、図6の(a)は

```
input | average() => avs
```

となり、意識の流れと登場順序が一致します。

プログラミング言語において代入が右から左なのは当たり前ですし、意識の流れと字面上の登場順序の食い違いをそんなに気にするのは「なんと大げさな」と感じる人も多いでしょう。しかし、このようなささいなことの積み重ねが言語の使い勝手の良さにつながるのです。

```
input = fread("result.csv") | map{x->number(x)} # (c)
avs = input | average() # 平均 (a)
sts = input | stdev() # 標準偏差 (b)
zip(avs, sts) | each{x->
  avg = x(0); std = x(1)
  fread("result.csv") | map{x->number(x)} | each {score->
    ss = (score-avg)*10 / std + 50
    print("得点: ", score, "偏差値:", ss)
  }
}
```

図6 偏差値の計算

関数呼び出しの変更

最後に手を付けたのが関数呼び出しの部分です。

前にも述べたように、Streamは、できるだけRubyとは異なる設計判断をしようと考えています。

また、最初からオブジェクト指向言語として設計したRubyとは異なり、Streamではむしろ関数型プログラミングを重視しようと考えました。このためメソッド呼び出しを中心に置くRubyに対して、Streamではあくまでも関数呼び出しを中心にしようと考えていました。

しかし、そうは言っても、オブジェクト指向の便利さ、特にポリモルフィズムは欲しかったので、関数呼び出しの第1引数の所属するネームスペースによって呼び出される関数が決定する仕組みも導入しました。

それぞれ理由のある設計判断ですが、これにより言語における関数の位置付けが若干ぶれてしまったのも事実です。

例えば通常の関数型言語ならば、

```
number(x)
```

という式は「numberという名前が付いている関数（オブジェクト）を引数xで呼び出す」という動作が期待されるでしょう。

一方、現状の Stream では

```
number(x)
```

の動作は「xがなんらかのネームスペースを持ち、そのネームスペースにnumberという関数が登録されていればそれを、そうでなければ、現在のスコープでnumberという名前で参照できるオブジェクトを関数として呼び出す」という動作をします。

まあ、若干呼び出しルールが複雑ではありますが、慣れてしまえばどうということはありません。困ることは、この動作の場合、呼び出される関数を総称的に取り扱うことができないということです。

図6の(c)の部分を見てください。ここではmapを使ってストリームのエレメントを数値に変換しています。mapは関数を引数として取りますから、本来ならばこの部分は

```
map(number)
```

と呼び出せるはずですが、しかし、現在の Stream ではnumber(x)という関数呼び出し（のように見えるもの）で、データを変換できるのは、それぞれのネームスペースに登録された関数が行っていることです。numberという名前で得ることができる値（関数）の働きとは限りません。その結果、上記のようにnumberという名前から「各種データから数値への変換を行う関数」を取り出すことはできず、

```
map(number)
```

のような呼び出しもできないことになります。

これではうれしくありません。そこで「そのコンテキストでの『関数呼び出し』と同じ働きをするオブジェクト」を導入することにします。ネームスペースに分散した関数を総称的に扱うことができるので、このオブジェクトを「総称関数 (generic function) オブジェクト」と呼ぶことにします。

総称関数オブジェクトを識別子の前に「&」を付けて取り出します。よって、

```
map(&number)
```

は上記の

```
map{x->number(x)}
```

と同じ働きをします。

関数の直接呼び出し

もう一つ、関数名と変数名が偶然に衝突した場合に対処するために、ポリモルフィズムを無視して、第1引数のネームスペースに関わらず、関数オブジェクトをそのまま呼び出す方法を用意しておきます。

```
func.(x)
```

のように引数のカッコの前に「.」を置くと、確実に関数オブジェクトの呼び出しになります。必ず識別子で指定しなければならない関数呼び出しと違って、こちらの形式ではfuncの部分は呼び出し可能なオブジェクトを返す任意の式を記述できます。

Lisp-1とLisp-2

このような関数呼び出しのあり方についての議論は、Lisp コミュニティーにおいて古くから議論されていて、(3-2でも述べたように)大きく分けると「Lisp-1」と「Lisp-2」と呼ばれています。

Lisp-1は関数と変数の名前空間が分離されていない方式です。名前空間が一つしかないからLisp-1なんですね。

Lisp-1を採用している言語としてはSchemeやJavaScriptやPythonがあります。Lisp以外のほかの言語にも適用な可能な設計方針の名前なのに、いまだにLispの名前を付けて呼ばれるのは変な感じですね。

これらの言語では、

```
print(x)
```

という関数呼び出しはprintという式を評価し、その結果である関数（または呼び出し可能なオブジェクト）を引数xで呼び出すという意味を持ちます。関数名に相当するのは単なる変数の参照ですから、任意の式を置くことができます。つまり、

```
((complex_expr)(args1))(args2)
```

のような式があると

- `complex_expr` を評価する
- その結果の関数を `args1` を引数として呼び出す
- その結果の関数を `args2` を引数として呼び出す

というような動作をすることになります。シンプルなルールで非常に複雑なことでも実現できることが分かります。

Lisp-1 は仕組みがシンプルであり、その上にいろいろな仕組みを組み立てるのに便利です。シンプルさを追求した Lisp 方言である Scheme で採用されたのも納得することができます。

Lisp-1 は関数オブジェクトの取り出しが基本的な操作ですから、関数を頻繁に取り扱う関数型プログラミングと相性が良く、最近の言語でよく採用されています。

Lisp-2 の利点と欠点

一方、関数と変数の名前空間が分離されている方式を Lisp-2 と呼びます。Lisp-2 方式を採用している言語は、CommonLisp、Ruby、Smalltalk などがあります。

Lisp-2 を採用している言語では、

```
print(x)
```

という関数呼び出しは、なんらかの関数テーブルから `print` という名前の関数を探し、その関数を引数 `x` で呼び出すという意味を持ちます。現在のスコープに `print` という名前の変数があっても、関数と名前空間とは独立しているので無関係です。

Lisp-2 は「関数を名前で呼び出す」という考え方がオブジェクト指向の「メッセージにより機能（メソッド）を呼び出す」という考え方にしっくりきます。元々オブジェクト指向言語として設計された Smalltalk や Ruby が Lisp-2 方式であるのも納得です。関数の探索が言語ランタイムの中に完全に隠蔽されているので、メソッドキャッシュのような関数呼び出し高速化のための仕組みを組み込むのが比較的簡単であるのもメリットです。

デメリットとして「この呼び出しによって呼ばれ得る関数」を総称的に扱うのがあまり得意ではないことが挙げられます。このため関数型プログラミングをするときに若干手間が増えてしまいます。

Lisp-1 であれば

```
map(number)
```

で済むものが

```
map({x->number(x)})
```

と書かなくてはならないのです。もっとも、先に紹介したような「総称関数オブジェクト」の導入によって、

```
map(&number)
```

程度に簡略化ができるので、さほど大きなデメリットではないと考えることもできます。

StreemとLisp-2

すでに述べたように、Streemでは当初、関数型プログラミングでの扱いやすさを期待してLisp-1方式を採用しようと考えていました。しかし元々オブジェクト指向プログラミングが好きなものですから、なんとかポリモルフィズムを導入しようといういろいろ考える間にLisp-2的挙動を導入してしまいました。

今回の改善で導入した総称関数は、そのLisp-2方式でありながらLisp-1的に動作させるための工夫です。いわばLisp-1.5とでも呼ぶべきでしょうか^{*1}。

まとめ

今回はStreem言語の文法を見直してみました。意外といろいろと直すべき点が見つかるものです。

しかし、ユーザーがいないといういろいろ試行錯誤できてよいですね。Rubyみたいに何万人もユーザーがいたら、ほんのささいな変更でも非互換性があると大問題になりますから。

今後の文法拡張としては、関数型言語でよく見られる「パターンマッチ」という機能について検討しようと考えています。

*1 初期のLisp処理系にLisp 1.5というものがあってかけているのですが、説明が必要な冗談は面白くありません。

言語デザイナーはどう文法を改善するのか

2016年8月号掲載分です。連載ではだいぶ終わり間近になって文法に大きく手を入れたのですが、書籍版では最終版の文法に合わせてサンプルプログラムなどを書き換えています。

このため今回の文法改善については、「このように変更した背景」という程度の意味しかありません。似たような記述は2-4にもありますね。連載が進むにつれて少しずつ文法が進化していったのがうかがえます。

この辺が雑誌連載の限界ですね。元の連載記事をあまり大きく書き換えるようにはしなかったもので、今回はそのまま掲載します。言語デザイナーが文法を設計したり、改善したりするときに、どんなことを考えているのかということの参考にしていただけるとよいのではないかと思います。

今回の記事には雑誌連載の限界を感じることももう一つあります。3-2と3-3で「Lisp-1とLisp-2」のネタが完全にかぶっていることです。元々この話題は私の大好きなネタではありますが、3-2は2015年12月号掲載で、この3-3は2016年8月号掲載で半年以上間が開いていました。このためネタの重複について全く意識していませんでした。この時点では書籍化など少しも考えていませんでしたから。書籍化に合わせて内容に応じて並べ替えたら、同じネタが並ぶことになるとは。自業自得ではありますが。

3-4 パターンマッチ



今回は関数型言語によく見られる機能である「パターンマッチ」について解説します。パターンマッチがあると、ある種のデータ構造の扱いがとて簡単になります。試行錯誤の末、ようやくStreamに実装できました。Rubyにも欲しいんですけどねえ。

パターンマッチといえはまず正規表現を思い浮かべますが、関数型言語におけるパターンマッチもやることは似たようなものです。つまりは、値とパターンとがマッチするかどうかチェックすることと、マッチした値から一部分を取り出すことです。

Erlangのパターンマッチ

ただし関数型言語では、マッチの対象になるのは文字列ではなく、データ構造です。例えばパターンマッチを備える関数型言語であるErlangの例を見てみましょう(図1)。整数の合計を次々と足し合わせていく「フィボナッチ数列」を求めるプログラムです。

Erlangではcase文を使ってパターンマッチをします(本当は関数定義部でもパターンマッチができるのですが、そちらはErlangの教科書にお任せします)。

case文はNで与えられる式とパターンとのマッチをして、マッチが成功すると「->」の後ろの文を実行します。パターンの中に変数があると、その変数への代入もします。

ただ、図1の例はあまりにもシンプルなので普通の条件分岐との違いやうれしさが分からないかもしれません。もうちょっとだけ複雑なパターンを考えてみましょう。リストの長さを求めるlen()関数をErlangでパターンマッチを使って定義すると図2のようになります。

図2のcase文を見ると、最初のパターンは[], つまり空リストです。これはLが空リストとマッチするとき、リストの長さはゼロであるということです。

```
fib(N) ->
case N of
  1 -> 1;
  2 -> 2;
  _ -> fib(N-1) + fib(N-2)
end.
```

図1 Erlangのパターンマッチ (1)

```
len(L) ->
case L of
  [] -> 0;
  [_|T] -> 1 + len(T)
end.
```

図2 Erlangのパターンマッチ (2)

次のパターンはちょっと変わっています。

`[_|T]`

は、リストの先頭要素が `_` であり、残りが `T` のとき、という意味です。リストが空でないとき、こちらのパターンにマッチします。Erlang のパターンマッチにおける「`_`」はなんにでもマッチする変数であり、中身には興味がないことを示します。これでリストの先頭と末尾の要素を別々に取り出せます。先頭を除

いたリスト `T` の長さを `len()` で求めた上に 1 を足すとこのリストの長さが求められるというわけです。これを実行すると図 3 のように計算が進みます。

```
len([1,2,3,4]) = len([1|[2,3,4]])
                = 1 + len([2|[3,4]])
                = 1 + 1 + len([3|[4]])
                = 1 + 1 + 1 + len([4|[]])
                = 1 + 1 + 1 + 1 + len([])
                = 1 + 1 + 1 + 1 + 0
                = 4
```

図3 lenの計算

再帰的定義とのセットが便利

長さとは先頭要素の長さ (1) に残りのリストの長さを加えたものであるという再帰的定義にとまどうかもしれませんが、慣れてしまえばどうということはありません。そして、パターンマッチ (と関数の再帰呼び出しの組み合わせ) はこのような再帰構造を持ったデータ構造の処理に大変便利なのです。それこそが多くの関数型言語がパターンマッチを備える理由です。

パターンマッチを備える関数型言語としては、今回紹介した Erlang のほかにも

- Haskell
- OCaml
- Scala

などたくさんあります。実際問題、明示的に関数型言語と名乗る言語はそのほとんどがパターンマッチを備えています。例外は古くからある (最近はあまり関数型言語として扱われない) Lisp と、その直系の子孫である Clojure くらいでしょうか。

「末尾再帰化」で最適化

さて、余談ではありますが、図 2 の再帰呼び出し関数はあまりリストが長いと呼び出しスタックを使い切ってしまいます。この問題を解決するためには関数を末尾再帰化するというテクニックが使えます。

末尾再帰とは関数実行の最後が自分自身への関数呼び出しであることです。この形の再帰関数

呼び出しは簡単にループに展開できることが知られており、ループにすれば呼び出しスタックはムダに消費しないし、関数呼び出しも省略できます。しかし、再帰呼び出しをしても図2のように

```
1 + Len(T)
```

のような構造をしていると、最後の実行は「+」関数なので末尾再帰になりません。

そこで図4のように書き換えると、末尾再帰にできてありがたいです。図4のプログラムは、Erlangにおいては引数の数が異なると違う関数として扱われることを利用しています。しかし、ほかの言語でも2引数の方の関数の名前を変えればよいだけです。再帰関数を末尾再帰に変えるために簡単に使えるテクニックになります。

```
len(L) -> Len(L, 0).  
Len(L, Acc) ->  
  case L of  
    [] -> Acc;  
    [_|T] -> len(T, Acc+1)  
  end.
```

図4 Erlangのパターンマッチ（末尾再帰化）

■ S t r e e m の パ タ ー ン マ ッ チ

では、Streamのパターンマッチをどのように実装するのかということについて考えなければなりません。

途中さんざん試行錯誤した点についてはお伝えしても退屈かもしれませんが、2016年5月頃に一瞬だけ関数の引数が

```
{x -> print(x)}
```

から

```
{|x| print(x)}
```

となっていたのは、この試行錯誤の影響です。手元での変更をついっかりチェックインして全世界に公開してしまいました。あと、東京Ruby会議の発表でもこちらの文法を使って説明してしまいました。

さて、そのような試行錯誤を経て決定したStreamのパターンマッチの文法は以下の通りです。

case、ifによるパターン

まず、関数オブジェクトの引数部分にパターンを書くことができるようにしました。パターンを置くためには、先頭に予約語「case」を置きます。

case パターン -> 実行文

また図5のように、caseを複数並べることで、複数パターンを指定することもできます。さらにどれにも当てはまらないケースのためにelseを置けます。

パターンマッチを含む関数の呼び出しは通常の関数と同じです。図6のように、呼び出されると上からパターンマッチをして、マッチしたら「->」の後ろの文を実行します。

並べられたパターンが複数マッチする場合でも、最初にマッチしたものだけが選ばれます。マッチするパターンが存在しなければ例外が発生します。

caseにはifによる条件式を付けることもできます(図7)。この条件式のことを「ガード」と呼びます。指定されたパターンにマッチしてもifで指定したガードが成立しなければ、成功したとはみなされません。

matchによるパターンマッチ

関数呼び出しではなく、直接パターンマッチをするにはmatch関数を使います。match関数を使うとパターンマッチは図8のようになります。match関数の実態は引数として与えられた関数オブジェクトを呼び出しているだけですが、Streamの持つ文法とうまくマッチしています。ほかの言語でパターンマッチする構文と似たように見えますし、実際ほぼ同じ働きをします。

```
foo = {
  case 1,x -> print(x+1)
  case 2,x -> print(x*2)
  else      -> print("else")
}
```

図5 複数のcaseとelse

```
foo(1,2) # => 3 (2+1だから)
foo(2,1) # => 2 (1*2だから)
foo(1)   # => else (マッチしない)
```

図6 パターンマッチの実行例

```
sign = {
  case 0 -> print("0")
  case x if x > 0 -> print("+")
  case x if x < 0 -> print("-")
}
sign(0) # => "0"
sign(10) # => "+"
sign(-1) # => "-"
```

図7 ifで「ガード」を付ける

```
match(1,2) {
  case 1,x -> print(x+1)
  case 2,x -> print(x*2)
  else      -> print("else")
}
```

図8 match関数

Streamのパターンマッチ構文を使って図1のプログラムを書き換えると図9のようになります。

```
fib = {  
  case 1 -> 1  
  case 2 -> 2  
  case n -> fib(n-1) + fib(n-2)  
}
```

図9 図1のパターンマッチをStreamで記述したもの

■ パターンマッチ構文

さてパターンマッチの概略は上に述べた通りなので、その詳細について考えてみましょう。

パターンマッチは値とパターンとを比較してマッチしているかどうかを判定するものです。そしてパターンには表1に示す種類があります。

種類	例	成功時の動作
変数	foo	変数への束縛
文字列	foo	文字列の一致
数値	42	数値の一致
配列	[1,a]	配列の各要素のマッチ
構造体	[kw:a]	構造体(名前付配列)のマッチ

表1 パターンの種類

変数パターン

まず、最初のパターンは「変数」です。変数はStreamの識別子で、英文字（アルファベット）から始まり、英数字が続くものです。ただし、Streamの予約語は変数になれません。変数は任意の式にマッチします。

変数は「束縛された状態」と「非束縛の状態」があります。代入やパターンマッチによって値が決定した変数のことを「束縛された状態」と呼び、まだ値が決まっていない変数を「非束縛の状態」と呼びます。非束縛の変数に対するパターンマッチは、マッチを試みた値との束縛が発生して常に成功します。束縛された変数に対するマッチはマッチを試みる値と束縛されている値とが一致するときに成功します。

実例を見ないと分かりにくいかもしれませんが、図10はパターンマッチを使った一致判定関数です。

same関数の中のパターンマッチで変数xが2回登場しているので、この二つの値がイコールのときにマッチします。

```
same = {  
  case x,x -> true # comparison in match  
  case _,_ -> false # fallback  
}  
  
print(same(1,1)) # => true  
print(same(1,2)) # => false  
print(same([1],[1])) # => true  
print(same([1],[2])) # => false
```

図10 パターンマッチによる一致判定

ワイルドカードパターン

変数マッチには例外があって、「_」という変数は決して束縛状態になりません。ということは、同じ変数を何度でもマッチさせられるということです。これにより、何か値があるはずなのでマッチはさせたいが、その中身には関心がないときにとりあえず置く変数として「_」を使えます。この働きから、「_」のことを「プレースホルダー変数」とか、なんにでもマッチすることから「ワイルドカード変数」と呼ぶこともあります。

図10のsame関数の定義でも

```
--
```

というパターンが登場しますが、具体的な変数と異なり、「_」はなんにでもマッチしますから、1番目と2番目に対応する値が異なってもマッチします。

束縛状態にならないということは、値も取り出せないということです。たとえマッチしていても、「_」変数の値は参照できません。未定義変数と同じ扱いになり、エラーになります。

リテラルパターン

変数が並んでいるだけなら、パターンマッチは通常の間数引数とほとんど差はありません。違いは変数だけでなく「リテラル」を直接並べられることです。

リテラルとは具体的な値です。パターンの中に文字列や数値が登場すると、その値自体とマッチしたときに成功します。Streamのパターンマッチに登場するリテラルは、文字列、数値、true、false、nilです。

配列は要素としてパターンを含むのでリテラルとは異なる扱いになります。

配列パターン

配列マッチは配列に対してマッチするパターンです。基本の配列パターンは

```
[パターン, パターン, ...]
```

のようにブラケット([])の間にコンマ区切りのパターンを並べたものです。配列マッチは、以下の条件が成立するときにマッチが成功します。

1. マッチに対応する値が配列
2. パターンの数と配列要素数がマッチ
3. すべてのパターンマッチが成功

配列パターンマッチの例を図11に示します。

```
match([1,2,3]) {
  case [a]      -> print(1)    # 配列[1,2,3]に対するマッチ
  case [a,b]    -> print(2)    # 1要素の配列にマッチ
  case [1.5,b,c] -> print(1.5) # 2要素の配列にマッチ
  case [a,"b",c] -> print("b") # 先頭要素が1.5の3要素配列にマッチ
  case [a,b,c]  -> print(3)    # 2番目の要素が"b"の3要素配列にマッチ
  case [a,b,c]  -> print(3)    # 任意の3要素配列にマッチ
  case _        -> print("any") # 配列でない場合も含めてマッチ
}
```

図11 配列パターンマッチ

配列マッチは再帰的な構造なので、配列の配列などのパターンも書けます。例えば

```
[[a,b],[c,d]]
```

は、二つの要素がそれぞれ2要素の配列である配列にマッチするパターンです。このマッチが成功すると、それぞれの要素の値がa、b、c、dという変数に束縛されます。

このパターンを

```
[[a,b],[a,b]]
```

と書き換えると、二つの配列の各要素が等しい配列、例えば

```
[[1,2],[1,2]]
```

のような配列にだけマッチするようになります。

可変長配列パターン

パターンを要素として持つ配列マッチに加えて、長さの分からない配列から、その一部を取り出す方法も提供しています。配列要素のパターンとして、「*変数」というものを加えると、マッチの残りの要素を配列として変数に束縛できます。つまり図12のようになります。

```
match([1,2,3]) {
  case [a,*b] ->
    print(a) # => 1 (最初の要素)
    print(b) # => [2,3] (残り)
}
```

図12 可変長配列パターン

可変長配列「*変数」は、配列パターンのどこか任意の場所に1回だけ登場できます。つまり、先ほどの例と同じように末尾に来てでもいいですし、

```
[*a,b]
```

のように先頭に来てても、

```
[a,*b,c]
```

のように真ん中に来てても構いません。しかし、

```
[a,*b,c,*d]
```

のように同じレベルに可変長配列パターンが複数回登場するとマッチする範囲が分からなくなりますから、文法エラーにしています。

最初の方に出てきた Erlang の

```
[H|T]
```

というパターンは Stream では

```
[H,*T]
```

になります。実際には Stream では実装が配列であり、Erlang ではリンクリストなので、内部処理は異なるのですが、先頭の要素と残りの要素にパターンマッチするという振る舞いは同じです。

構造体パターン

Stream では配列の要素にそれぞれ名前を付けられます。いわば「ラベル付き配列」ですが、ここではそれを「構造体」と呼びます。

構造体もパターンマッチの対象になります。構造体パターンは以下のような文法を持ちます。

```
[ラベル:パターン, ...]
```

構造体パターンは指定したラベルのパターンがすべて該当する値にマッチするときに全体がマッチします。

基本的なルールは簡単ですが、実装のためにはコーナーケースについて考慮する必要があります。

最初に考慮しなければならないのは、構造体の実装は配列要素にラベルを付けたものですから、明確な順序があります。パターンに指定したラベルと配列のラベルの順番が異なった場合にマッチすべきかどうかを決定しなければなりません。

使い勝手を考えると、順番は考慮しない方がよさそうです。構造体として用いるデータ構造で各要素の順番を考慮することはまれですし、むしろ順番を考えたくないからこそラベルを付けるわけですから。

次に考慮すべき点は、構造体をマッチするために、すべてのラベルが一致する必要があるか、それとも一部のラベルが一致しただけでマッチを成功したと見なすかどうかです。

これも使い勝手から考えてみましょう。

完全マッチを強制すると、構造体に要素が追加されたときにすべてのパターンを書き換える必要が発生します。さらに不完全マッチを許せば、構造体から一部の要素だけを取り出すためにパターンマッチを用いることができます。ここは不完全マッチを許す方がよさそうです。

最後に、構造体のラベルは実は重複を許します。同じ構造体に複数の同じラベルが付いていた場合、パターンマッチはどのようにマッチすればよいでしょうか。例えば

```
[a,1,b:2,a:3]
```

という構造体があったとして、aというラベルに対応するパターンは1にマッチすればよいでしょうか。それとも3とマッチすればよいでしょうか。

私が思いついたのは以下の二つの案です。

1. 常に最初のラベルにマッチする
2. ラベルの登場順にマッチする

1案はかなりシンプルで、実装も簡単で挙動も予想しやすいのが利点です。ただし、複数の値のうち先頭のもののしかパターンマッチでは取り出せません。

2案は

```
[a:x, a:y]
```

のようなパターンがあれば、aというラベルが付いた最初のパターンは初めのaの値(1)にマッチし、次のaのラベルが付いたパターンは2番目のaの値(3)にマッチするというものです。

2案の「いたれりつくせり」感は、おもてなし度が高いですが、今回はシンプルな1案を採用します。過去の経験からいうと、2案は望ましくない結果をもたらすことが多いからです。一つの構造体に同じ名前のラベルが複数使われるような例外的な事象にあまりコストをかけても報われな

いばかりか、挙動の予測困難さにユーザーをびっくりさせてしまうことがあります。これらの仕様を踏まえつつ、構造体のパターンマッチの例を図13に示します。この構造体パターンマッチを使えば、CSVから読み込んできたデータから一部を抽出して表示することが図14のように簡単にできます。

```
match([a:1,b:2,c:3,a:4]) { # 構造体（ラベル付き配列）でマッチ
  case [a:a, c:c, a:x] -> # bは使われてなくてもマッチ
                        # aは最初のものだけ使われるのでxも1になる
                        # ラベルの順番は考慮されない
    print([a,c,x])      # => [1,3,1]
}
```

図13 構造体パターンマッチ

```
# 以下のようなCSVファイル（voters.csv）がある
# name,address,age
# まつもとゆきひろ,島根県松江市,51
# まつもとたくと,島根県松江市,19
# まつもとポチ,島根県松江市,4
# まつもとタマ,島根県松江市,1

# voters.csvから選挙権のある人を選択・表示
fread("voters.csv")|csv()|each{
  # 2016年から18歳以上が選挙権を持つ
  case name:name, age:age if age>=18 ->
    print(name) # 有権者名を表示
  else ->      # 選挙権なければ何もしない
}
```

図14 CSVの読み込みと抽出

構造体パターンにも可変長配列が適用可能です。

ただし、構造体パターンは要素の順番を考慮しないので、「*変数」が登場できるのは末尾だけになります。その場合、「*変数」には、すでにラベルで指定された以外の値を集めた構造体（ラベル付き配列）が代入されます。

では、例題を見ながら考えてみましょう。

```
match([a:1,b:2,c:3]) {  
  case [a:x,*z] -> print(x,z)  
}
```

というプログラムは何を出力するでしょうか。

答えは、

```
1 [b:2,c:3]
```

です。

つまり、`a:x`というパターンにより、変数`x`にラベル`a`に対応した値である`1`が、そしてラベル`a`以外の要素を集めた構造体が`z`に代入されるというわけです。

では、次の問題です。構造体から可変長配列を取り出すとき、同じ名前のラベルが複数存在していたらどのように振る舞うべきでしょうか。重複したラベルが指定されているときと、されていないときそれぞれで考えてみてください。

具体的なプログラムだと、

重複しているラベルが指定されているケース

```
match([a:1,b:2,a:3]) {  
  case [a:x,*z] -> print(z)  
}
```

重複したラベルが指定されていないケース

```
match([a:1,b:2,a:3]) {  
  case [b:x,*z] -> print(z)  
}
```

で、それぞれ何が出力されるかということですね。

せっかくの機会なので、メモにパターンを列挙しながら、どのような挙動だとどううれしいかについて考えてみてください。使い勝手や実装の複雑さなど考えるべき要素はいくつもあります。

自分なりの考えがまとまったら、最新の *Streem* をダウンロード・コンパイルして上記のプログラムがどのような結果になるか見てみましょう。みなさんの結論と同じ動作をしましたか？

もし奇特にも、単にこの記事を読むだけでなく、考える時間を取ってくださった読者の方がいらっしゃったのなら、今、みなさんが行った活動こそが言語デザイナーが日常している言語デザイン活動そのものなのです。

いつか言語のデザインをやりたいと考える読者が一人でもいらっしゃるといいなと思っています。

ネームスペースパターン

Stream のオブジェクト指向機能の回 (3-2) で解説したように、Stream のあらゆる値はその値に適用可能な関数を保持する「ネームスペース」を保持しています。Stream のネームスペースは、ほかの言語におけるクラスのようなもので、その値の「型」を表現していると考えられます。

その型を確認するために使えるのがネームスペースパターンです。ネームスペースパターンはほかのパターンの後ろに「@ネームスペース名」という形で表記します。

例えば

```
str@string
```

というパターンは対応する値が string ネームスペースを持つ（つまり文字列の）ときにマッチが成功して、変数 str にその文字列を代入します。

ネームスペースパターンにはもう一つの表記法があります。それは

```
[@ネームスペース名 値,...]
```

です。配列（あるいは構造体）表記の最初に「@ネームスペース名」を置くわけですね。意味としては

```
[値,...]@ネームスペース名
```

と変わりません。

こちらはその配列がネームスペースが適用された特別な配列であることを明示するための表記です。

```
new namespace [配列]
```

で作られたオブジェクトは

```
[@namespace 配列パターン]
```

でマッチすると覚えてください。

まとめ

今回は Stream に新たに導入したパターンマッチについて解説しました。また、パターンマッチの挙動をデザインする過程で言語デザイナーの思考の一端も紹介できたのではないかと思います。

タイムマシンコラム

Stream に最適なパターンマッチ機能

2016 年 9 月号掲載分です。今回は関数型言語の多くが搭載しているパターンマッチ機能を設計・実装しました。パターンマッチが欲しかったんですね。特にパイプラインを流れてきたデータをパターンによって分別して処理するというのは Stream が推奨するストリーミングプログラムに最適な処理だと思います。Stream におけるパターンマッチの活躍がとても楽しみです。

本当は Ruby にもパターンマッチ機能を導入したいのですが、既存の文法との衝突や機能的な制限などがあり、今のところ実現できていません。Ruby では、過去にも欲しいと思っていた機能が長年の試行錯誤の上、とうとう導入された例がいくつもありますから（例：Refinement）、パターンマッチも将来導入できるとよいのですが。

連載と書籍版との掲載順序の違いで、今回当たり前のように登場している CSV の読み込み機能は書籍版ではまだ解説していません。これが登場するのは 5-3 になります。

第4章

Streamオブジェクトを 実装する

4-1

ソケットプログラミング

ここまでの開発によって、Streemは言語として「とりあえず動く」程度にまで成長しました。今回は言語処理系を離れて、ネットワークプログラミングについて学ぼうと思います。ソケットを使ったネットワーク通信の機能をStreemに追加します。

いまやネットワークは空気のような存在です。最近ではコンピュータを単体で利用することは少なくなっていて、多くのアプリケーションがネットワークの存在を前提にしています。出張などで飛行機に乗り、ネットワークから隔離されてしまうと、日常的に利用しているアプリケーションのうち、どれほど多くがネットワークに依存しているかということを改めて考えさせてくれます。

だからというわけでもありませんが、Streemにもネットワークを経由して通信できる機能を付与してみましょう。Streemの機能拡張の良い例題にもなりそうです。

StreemソケットAPI

まず、Streemでソケットを用いるプログラムを見てみましょう。図1がStreemによるネットワークサーバーの実装例で、図2がクライアントの実装例になります。

それぞれあまりに簡単なので拍子抜けしてしまいそうです。これらのプログラムの挙動を行ごとに解説してみましょう。

```
01 # simple echo server on port 8007
02 tcp_server(8007) | {s -> s | s}
```

図1 Streemによるネットワークサーバー実装例

```
01 s = tcp_socket("localhost", 8007)
02 stdin | s
03 s | stdout
```

図2 Streemによるネットワーククライアント実装例

Streemネットワークサーバー

では、まず図1のプログラムから見てみましょう。1行目は単なるコメントです。「ポート8007で待つechoサーバー」と書いてあります。プログラム全体で2行しかありませんから、ネットワークサーバーが実質1行で記述できていることになります。

ネットワーク接続は「ホスト名」と「ポート(番号またはサービス名)」によって指定します。このサーバーを起動すると、指定したホスト名とポート番号(8007)でこのサーバーに接続できます。

「tcp_server(8007)」というのがサーバーを作る関数です。この関数はサーバーソケットを作り、待ち受けをして、クライアントの接続を受け付けるとそれに相対するソケットを作って、パイプラインに流すという働きをします。

次の「{s -> s | s}」というのは関数リテラルです。Streamでは関数をパイプラインにつなぐと、パイプラインを流れる各要素を引数にして、その関数を呼び出します。今回の関数は、クライアントに接続しているソケットを引数にして、それをパイプラインでつないでいます。「s | s」というのは一見何をやっているのか分からないかもしれませんが、「クライアントソケットからの入力をそのまま送り返す」という意味になります。ソケットは読み込みと書き込みの両方ができる全二重の接続であることに注目してください。

一般的なサーバーでは、読み込んだ情報を加工して返すような、もうちょっと複雑な処理になるでしょう。

Streamネットワーククライアント

図2のクライアントプログラムもシンプルです。1行目の「tcp_socket("localhost", 8007)」で、ホストlocalhostのポート8007番に接続するソケットを生成して、変数sに代入しています。

2行目では、標準入力(stdin)から受け取った入力をソケットに流しています。これでソケットを経由してネットワークサーバーにデータを送ったことになります。

3行目では、ネットワークサーバーから受け取ったデータを標準出力(stdout)に流しています。

このプログラムを図1のechoサーバーにつなぐと、ネットワーク通信をしつつ、キーボードから入力した文字列をそのまま返す(ということはネット通信を考えなければcatを引数なしで実行したのと同じ)動作をします。

ソケットをつないだネットワークプログラミングも、Streamを使えば非常にシンプルになることが見て取れたのではないかと思います。

Streamの機能拡張

では、今度はこのソケット機能がStreamでどのように実装されているのかを見てみましょう。まずはStreamに関数を追加するところからです。

図1と図2のプログラムで見たように、ソケット機能は二つの関数をStreamに追加することで実装されています。Streamに関数定義を追加するには図3のようにします(説明の都合上、実際のソースを若干加工しています)。今回解説する実際のCコードはStreamのソースコードのsocket.cファイルで見られます。

Streamでは、Cで実装する(グローバル)関数を次のような手順で定義します。

まず、C関数ポインタから関数オブジェクトを作ります。それにはstrm_cfunc_value()関数を用

います。strm_cfunc_value()で登録する関数は、戻り値がintで、四つの引数を取るものでなければなりません。最初の引数(strm_state*)がメソッド呼び出しのコンテキスト、2番目(int)がStream関数に与えられた引数の数、3番目が引数を格納した配列、4番目がStream関数からの戻り値を格納する場所です。この関数自身からの戻り値は、関数実行の成功か失敗を表すもので、関数が成功したときはSTRM_OK(=0)、失敗したときはSTRM_NG(=1)を返します。

次にその関数オブジェクトをグローバル変数に代入します。グローバル変数の定義にはstrm_var_def()関数を用います。

初期化用の関数strm_socket_init()は、インタープリタ初期化関数node_init()から呼ばれています。新しい機能を追加したら、その初期化用関数をnode_init()から呼び出す必要があります。

```
int tcp_server(strm_state*, int, strm_value*, strm_value*);
int tcp_socket(strm_state*, int, strm_value*, strm_value*);

void
strm_socket_init(strm_state* state)
{
    strm_var_def("tcp_server", strm_cfunc_value(tcp_server));
    strm_var_def("tcp_socket", strm_cfunc_value(tcp_socket));
}
```

図3 Streamへの関数定義追加

ソケットとは

そもそもUNIX系OSにおけるソケットとはどのようなものなのでしょう。

ソケットはネットワーク接続の通路となるための「OSオブジェクト」です。ソケットが発明されたのは4.2BSDだったとされています。

ソケットはファイルディスクリプタと呼ぶ整数で識別します。これはopenシステムコールでオープンしたファイルなどと同じ識別子です。そしてファイルディスクリプタに対しては共通の操作として、read(読み込み)、write(書き込み)、select(待ち受け)、close(終了)などの処理ができます。

しかし、考えてみればディスク上のファイルにつながっているファイルディスクリプタと、ネットワーク接続しているソケットにつながっているファイルディスクリプタで、読み書きなどの処理が全く共通であるはずがありません。つまり、システムコールは内部的にファイルディスクリプタの種類に応じて適切な処理を自動的に選択していることになります。

このような対象の種別に応じた自動的な分岐はある種のオブジェクト指向であり、そういう意味で実はUNIXはオブジェクト指向OSだったのです。驚き。

クライアントソケット

ソケットの使い方はそれほど難しくはありません。が、Cにおけるソケットプログラミングは(きめ細かな指定ができる分だけ)手順が多くなります。まずは大まかな手順から説明します。まずはよりシンプルなクライアント側からです。

クライアントからのソケット通信手順は、以下の通りです。

1. 接続先情報取得
2. ソケット生成
3. 接続
4. 入出力

まずは接続先を指定します。ネットワーク接続先はホストとポートの組み合わせによって指定します。TCP/IP 接続ではホストはIPアドレスによって表現されます。時々見かける「192.168.0.1」とか「127.0.0.1」のような四つの(255以下の)数の組み合わせで表現されているものがIPアドレスです。元々のTCP/IPアドレスは32ビット(4バイト)で表現され、IPアドレスはその各バイトを10進数で表現したものでした。

しかし、IPv4アドレスが枯渇しつつある現在、アドレスに128ビット用いるIPv6が登場しています。

IPv4の頃は、ホスト名からIPアドレスを得るためには`gethostbyname()`関数を用いていましたが、最近ではIPv4とIPv6の使い分けができる`getaddrinfo()`関数を使うことが望ましいとされています。

`getaddrinfo()`関数を用いるとアドレス、ポート、ソケットタイプの情報が得られます。あとはそれを用いて、`socket()`システムコールでソケットを生成してから、`connect()`システムコールでサーバーに接続するだけです。

接続後は通常のファイルディスクリプタとして動作しますから、`read()`や`write()`を用いてデータを読み書きすることになります。ソケットのデータの読み書きには`recv`(読み込み)、`send`(書き込み)という別のシステムコールもありますが、今回は使わないのでこれらの説明はまた別の機会にしましょう。

ソケットの使い方(クライアント)

では、Streamのソケット機能の実装を見ながら、Cにおけるソケットプログラミングの実際を見てみましょう。

図4がStreamの`tcp_socket`関数の実装です。説明のためWin32対応のコードは削っています。

```

static int
tcp_socket(strm_state* state, int argc, strm_value* args, strm_value *ret)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sock, s;
    const char *service;
    char buf[12];
    strm_string* host;

    if (argc != 2) {
        return STRM_NG;
    }
    host = strm_value_str(args[0]);
    if (strm_int_p(args[1])) {
        /* 文字列化したポート番号を指定 */
        sprintf(buf, "%d", (int)strm_value_int(args[1]));
        service = buf;
    }
    else {
        strm_string* str = strm_value_str(args[1]);
        service = str->ptr;
    }

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */ ← ヒントの指定
    hints.ai_socktype = SOCK_STREAM;
    s = getaddrinfo(host->ptr, service, &hints, &result); ← getaddrinfo()関数

    if (s != 0) { /* getaddrinfo失敗 */
        /* エラーの理由をgai_strerror()で調べる */
        node_raise(state, gai_strerror(s));
        return STRM_NG;
    }

    /* addrinfoをたぐって接続してみる */
    for (rp = result; rp != NULL; rp = rp->ai_next) {
        sock = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        if (sock == -1) continue; /* 失敗したら次にチャレンジ */
        /* connectしてみる */
        if (connect(sock, rp->ai_addr, rp->ai_addrlen) != -1) ← connect()関数
            break; /* 成功したらループ脱出 */
        close(sock); /* ソケットクローズして再挑戦*/
    }

```

```

}
/* 結果のaddrinfoを解放 */
freeaddrinfo(result);

if (rp == NULL) { /* リンク全部失敗した */
    node_raise(state, "socket error: connect");
    return STRM_NG;
}
/* socketをstrm_ioオブジェクトにラップして返す */
*ret = strm_ptr_value(strm_io_new(sock, STRM_IO_READ|STRM_IO_WRITE|STRM_IO_
FLUSH));
return STRM_OK;
}

```

図4 tcp_socket関数

まず手順1として、接続先の情報を取得します。情報取得にはgetaddrinfo()関数を用います。getaddrinfo()はホスト、サービス、ヒント、結果の四つの引数を取ります。

最初の引数である「ホスト」にはホスト名を渡します。このとき、「127.0.0.1」(IPv4)とか「:::1」(IPv6)とかのIPアドレスの文字列表現を渡すこともできます。

「サービス」にはサービス名を渡しますが、名前が与えられたときには「/etc/services」に登録されているサービス名からポート番号を検索します。「8007」のような数字からなる文字列が渡されたときには、その番号のポートを利用します。ホストとサービスはいずれかがNULLであっても構いませんが、両方ともNULLであれば情報を取得できないのでエラーになります。

「ヒント」は情報を取得するためのヒントになるaddrinfo構造体を渡します。特に指定することがなければNULLを渡しても構いませんが、今回はIPv4でもIPv6でも構わないという意味でai_familyにAF_UNSPECを、UDP接続ではなくTCP接続を検索するという意味でai_socktypeにSOCK_STREAMを指定しています。

sockaddinfo()はデータ取得に成功すると0でない値を返します。なんらかの理由でエラーだった場合、gai_strerror()関数でそのエラーの理由を示す文字列を取得できます。

getaddrinfo()が成功すると「結果」でポインタを渡した変数にaddrinfo構造体が返ってきます。検索結果が複数ある場合にはaddrinfo構造体がリンクリストになっているので、先頭から順番に試すことになります。

結果として返ってきたaddrinfo構造体はfreeaddrinfo()関数を使って解放してやる必要があります。

情報が取得できれば、あとのソケット接続は簡単です。socket()システムコールでソケットを作り、connect()システムコールでサーバーに接続します。socket()もconnect()も呼び出しに必要な情報はすべてaddrinfo構造体に含まれています。

connect システムコールが成功したら、サーバーとの接続は確立されています。あとは通常のファイルディスクリプタとして read や write を使って読み書きできます。注意すべき点は、ソケットは同じファイルディスクリプタで読み込みも書き込みもできることです。このような双方向通信ができる特性を「全二重」と呼びます。

全二重通信をしている場合には、通信終了時に下手に close を呼んでしまうと読み込みの通信路も書き込みの通信路も同時に閉じてしまうことになります。このような全二重のファイルディスクリプタを、片方の通信路だけ閉じるために shutdown システムコールがあります。

Stream でもソケットに対応するため、I/O の取り扱いを少し変更して、双方向通信の場合には、まず shutdown システムコールを呼ぶようにしています。もちろん全二重でないファイルディスクリプタに対して shutdown システムコールを呼ぶと、エラーになってしまいます。しかし、特に害はないのでそのまま無視できます。

サーバーソケット

今度はサーバー側のソケットの使い方を見てみましょう。サーバー側のソケット通信手順は以下の通りです。

1. 接続先情報取得
2. ソケット生成
3. listen/bind
4. accept
5. 入出力

「接続先情報取得」の部分はクライアントソケットと同様に getaddrinfo() を用います。ただし、IPv6 と IPv4 の両対応などを考えなければ、getaddrinfo() を用いなくて、直接 socket() システムコールと bind() システムコールを呼んでも問題はありません。

サーバーとして待ち受けをするためには、listen システムコールで待ちキューの長さを指定し、bind システムコールでサーバーとして登録します。

登録したサーバーソケットに accept システムコールを呼び出すと、クライアントとつながった新しいソケット(ファイルディスクリプタ)を返します。注意すべき点は、クライアント側と異なり、サーバーソケットは入出力の対象とならず、クライアントからの接続待ちをするソケットであるという点です。

クライアントと接続しているソケットは、通常の読み書きできるソケットですから、普通に read/write で通信することができます。

ソケットの使い方(サーバー)

サーバーソケットは接続手順がクライアントソケットとは若干異なるため、サーバーを提供する

Stream関数tcp_serverの実装もtcp_clientとはかなり異なっています。

そういえば、Streamのタスクの作り方もまだちゃんと説明していませんでしたから、これについても解説します。

図5はtcp_server関数の実装です。この関数はソケットを生成してから、そのソケットで接続を待つタスクを生成するまでを担当します。

tcp_server関数の実装は図4のtcp_socket関数の実装とさほど変わりません。違いは以下の点でしょう。

```
struct socket_data {
    int sock;
    strm_state *state;
};

static int
tcp_server(strm_state* state, int argc, strm_value* args, strm_value *ret)
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sock, s;
    const char *service;
    char buf[12];
    struct socket_data *sd;
    strm_task *task;

    if (argc != 1) {
        return STRM_NG;
    }
    if (strm_int_p(args[0])) {
        sprintf(buf, "%d", (int)strm_value_int(args[0]));
        service = buf;
    }
    else {
        volatile strm_string* str = strm_value_str(args[0]);
        service = str->ptr;
    }

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_STREAM; /* Datagram socket */
    hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address */
    hints.ai_protocol = 0;          /* Any protocol */

```



```

s = getaddrinfo(NULL, service, &hints, &result); ← getaddrinfo()関数
if (s != 0) {
    node_raise(state, gai_strerror(s));
    return STRM_NG;
}

for (rp = result; rp != NULL; rp = rp->ai_next) {
    sock = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
    if (sock == -1) continue;

    if (bind(sock, rp->ai_addr, rp->ai_addrlen) == 0) ← bind()関数
        break;
        /* Success */
    close(sock);
}

if (rp == NULL) {
    node_raise(state, "socket error: bind");
    return STRM_NG;
}
freeaddrinfo(result);

if (listen(sock, 5) < 0) { ← listen()関数
    close(sock);
    node_raise(state, "socket error: listen");
    return STRM_NG;
}

/* タスクの生成 */
/* タスクデータの割り当てと初期化 */
sd = malloc(sizeof(struct socket_data));
sd->sock = sock;
sd->state = state;
/* strm_task_newでタスク生成 */
/* strm_task_new(タスクタイプ, タスク関数, 終了関数、データ) */
task = strm_task_new(strm_producer, server_accept, server_close, (void*)sd);
/* 戻り値としてタスクオブジェクトを作って代入 */
*ret = strm_task_value(task);
return STRM_OK;
}

```

図5 tcp_server関数

まず、接続先ホストを指定するクライアント側と比較して、サーバー側は任意のホストからの接続を受け付けるため、ホスト指定がありません。また、ヒントとしてAI_PASSIVEを指定して、どのアドレスからの接続も受け付けることを明示的に教えています。

サーバーソケットではconnectシステムコールの代わりにbindシステムコールを用います。connectが「接続に行く」という司令であったのに対して、bindは「接続を待つ」という意味になります。

またbind後にはlistenシステムコールを呼び出しています。listenの引数である待ちキューの長さですが、昔聞いた「とりあえずlistenには5を指定しておけ」という教えに従って今回は5を指定しています。が、20年以上前に聞いた話であり根拠があるわけでもないようですし、現代のようなハイトラフィックが予想される環境では、もっと大きな数を指定した方がよいのかもしれない。

タスクの生成

tcp_serverの末尾ではタスクを生成しています。まず、タスクが利用するデータを割り当て、それを初期化しています。

そして、strm_task_new関数を呼び出して新しいタスクを作っています。strm_task_new関数の引数は四つ。それぞれ、タスク種別、タスク関数、終了関数、そしてタスクデータです。タスク種別は表1の

種別	意味
strm_producer	生産者(入力なし、出力あり)
strm_filter	加工者(入力あり、出力あり)
strm_consumer	消費者(入力あり、出力なし)

表1 タスク種別

3種類のいずれかを指定します。今回はacceptしたクライアントソケットを「作り出す」タスクですから、生産者扱いになり、strm_producerを指定します。

タスク関数は実際のタスクを実行する関数で、終了関数はタスク終了時に呼ばれる関数です。今回はserver_acceptとserver_closeを指定していますが、これらの中身については後で説明します。

サーバーソケットのacceptから後を担当するのが、server_accept関数（とそこから呼ばれるaccept_cb関数）です（図6）。サーバーソケットにクライアントから接続が来ると、サーバーソケットは「読み込み待ち」の状態になります。そこでread()の代わりにaccept()を呼べば、クライアントと接続されたソケットが得られるというわけです。

server_accept関数ではタスクデータを取り出し、strm_io_start_read関数を呼び出して、ソケットファイルディスクリプタの読み込みに対して待ち受けをします。クライアントソケットからの接続要求が来ると、コールバックとして指定したaccept_cb関数が呼ばれます。

accept_cb関数は、acceptシステムコールを呼び、得られたソケットをstrm_io構造体にラップして、emitしています。これでパイプラインの次のタスクにソケットを引き渡します。strm_io_emit関数は、第3引数で指定したファイルディスクリプタに対して入力があれば、第4引数で指定したコールバックを呼び出します。ここではコールバックにaccept_cb自身を指定しているので、全体として「サーバーソケットに入力があるたびaccept_cbが呼ばれるループ」が構成されていることになります。

```

static void
accept_cb(strm_task* task, strm_value data)
{
    struct socket_data *sd = task->data;
    struct sockaddr_in writer_addr;
    socklen_t writer_len;
    int sock;

    writer_len = sizeof(writer_addr);
    sock = accept(sd->sock, (struct sockaddr *)&writer_addr, &writer_len);
    if (sock < 0) {
        close(sock);
        if (sd->state->task)
            strm_task_close(sd->state->task);
        node_raise(sd->state, "socket error: listen");
        return;
    }

#ifdef _WIN32
    sock = _open_osfhandle(sock, 0);
#endif
    strm_io_emit(task, strm_ptr_value(strm_io_new(sock, STRM_IO_READ|STRM_IO_
WRITE|STRM_IO_FLUSH)),
                sd->sock, accept_cb);
}

static void
server_accept(strm_task* task, strm_value data)
{
    struct socket_data *sd = task->data;

    strm_io_start_read(task, sd->sock, accept_cb);
}

```

図6 server_accept関数

まとめ

今回はStreemにソケット通信機能を追加しました。ネットワークを通じて通信できることでStreemの応用範囲が広がりそうです。

今回のソケット対応もmattn (Yasuhiro Matsumoto) さんが送ってくださったプルリクエストがベースになっています。いつも感謝です。

Streem開発のコミュニティにぜひ参加を

2015年8月号掲載分です。Streemでもソケット通信ができるようになりました。

まあ、ネットワーク通信はできて悪いことはないのですが、今回の解説はソケットの使い方にフォーカスを置いたものではなく、ソケットをStreemにどのように実装したかという解説になっています。ということは、ソケットプログラミングの学習にはあまり役に立たないような気がしますね。書いてたときには「良い解説を書いた」という気になっていたのですが。

しかし、気を取り直してみると、今後、誰かがStreemに機能を追加しようとしたときに、StreemのC APIを利用することについては参考になりそうな気がします。今後、Streemの開発に積極的に参加してくれる人が出てくれるとよいのですが。コミュニティとして協力して開発することこそがオープンソースソフトウェア開発の醍醐味であり、一番の楽しみなのでから。

4-2

基本データ構造



プログラミングにおいてデータ構造は重要です。プログラミング言語は初めからいくつかのデータ構造を提供することで、プログラミングを支援します。今回は、各種プログラミング言語のデータ構造について調べて、Streamの設計に反映します。

ほとんどのプログラミング言語は、組み込みのデータ構造を持っています。ここでいう組み込みとは、言語（処理系）に最初から組み込まれていて、ライブラリなどをロードすることなく利用できるという意味です。

どのようなデータ構造を組み込みにするかは、言語設計者や処理系作成者の判断になるわけですが、その判断は言語の性格をかなり強く反映します。今回は、いくつかの言語の組み込みデータ構造と、そこからうかがえる設計判断について見てみます。それから、今回設計している言語Streamの組み込みデータ構造をどうすべきかについて考察します。

■ C の 基 本 デ ー タ 構 造

まずはCの基本データ構造について見えます。Cを選んだ理由は二つあります。一つは、現代で広く使われているプログラミング言語の中で、C(とC++)は基本データ構造がそれ以外の多く言語とかなり異なっていて非常に特徴的であることです。もう一つは、解説している私自身が最も長い時間使っている言語がCだからです。

Cの基本データ構造を表1に示します。Cの基本データ構造は、大きく四つのグループに分けられます。

型	種別	解説	グループ
char	整数	文字(8ビット整数)	整数
short	整数	短整数	
int	整数	整数	
long	整数	長整数	
long long	整数	長々整数	
enum	列挙型	実体は整数	浮動小数点数
float	浮動小数点数	単精度	
double	浮動小数点数	倍精度	
*	ポインタ	実体はアドレス	アドレス
[]	配列	実体はアドレス	
struct	構造体	—	構造体
union	ユニオン	—	

表1 Cの基本データ構造

最初のグループは整数属です。Cの整数はそのサイズごとに各種そろっています。Cの整数は仕様上サイズが固定されておらず、

```
char ≤ short ≤ int ≤ long ≤ long long
```

ということしか決まっています。特殊なコンピュータでは、すべての整数型のサイズが64ビット固定というものもかつて存在していたそうです。現代広く使われているコンピュータ(とOS)でよくある組み合わせは表2の通りです。アーキテクチャーが32ビットと64ビットの場合で分かれていて、64ビットでは2種類をよく見かけます。

アーキテクチャー	32ビット	64ビット	
char	8	8	8
short	16	16	16
int	32	32	32
long	32	32	64
long long	32	64	64

表2 Cの整数のサイズ

整数属の「おまけ」的存在として列挙型のenumがあります。これは「名前」を表現するデータ型ですが、その実体は単なる整数です。

ポインタで演算ができる

次のグループは浮動小数点数属です。Cの規格では決められていませんが、現在実用化されているほぼすべてのコンピュータではIEEE754という浮動小数点フォーマットが用いられており、単精度では32ビット、倍精度では64ビットのサイズになっています。

第3のグループはアドレス属です。メモリー上のある地点を表現するのがアドレスで、データの並びを表現するのが配列です。配列はメモリー割り当て用で、ポインタはメモリー操作用であると考えてほぼ間違いありません。ポインタが必要とされるところに、配列を渡すと自動的にポインタに変換されます。

Cのポインタのユニークな点は、整数と似たような演算が可能な点です。Cプログラマには当然に思えますが、ポインタに整数を加算したり、ポインタの差を取ったりすることができるのは、ほかの言語にはあまりない特徴です。

Cの基本データ構造で最後のグループが構造体属です。構造体はstructで定義される「データの塊」です。配列は同じ種類のデータの並びですが、構造体は任意のデータの並びになります。構造体に含まれる各データ(メンバー)には名前が付けられています。プログラム上では見えませんが、メモリーアクセスの都合上、データとデータの間には隙間(パディング)が入れられる可能性があります。

変幻自在の「ユニオン」

今回の分類ではユニオンも構造体属に含めました。ユニオンはunionで定義されます。unionの定義は構造体によく似ていますが、構造体がデータの並びを定義するものであるのに対して、ユニオンは同じメモリー領域に対する複数の型での解釈を定義するものです。

ユニオンはあまり頻繁に用いられるものではないので、そのようなものをなんに使うのか疑問に持つ人も多いでしょう。ユニオンの使われ方はいくつかありますが、典型的なものは以下の3種類です。

- 最大サイズの確保
- 条件付き構造体定義
- メモリー解釈の操作

「最大サイズの確保」とは、複数のデータ型が存在するときに、そのいずれであっても格納できる十分な大きさを持った領域を確保することです。

例えばCRubyでは、各種オブジェクトを格納できるメモリー管理のための配列に、「オブジェクトを表現する構造体のユニオンの配列」というものを使っています。実際に利用するときにはオブジェクトの型に合わせて、配列要素へのポインタをそのオブジェクトを表現する構造体へのポインタにキャスト（型変換）します。

「条件付き構造体定義」とは、条件によって構造体の定義が変化することです。これは具体例を見ないと分からないかもしれません。またCRubyの話になりますが、CRubyの文字列(String)は、メモリー消費量を削減するため工夫をしています。文字列長が一定サイズ以下の場合、構造体内部に文字列情報を埋め込み、それ以上の場合には別に割り当てたメモリー領域に文字列情報を持つようにしています。

つまり、文字列を表現する構造体(struct RString)の定義が、文字列の長さという条件で変化することを意味します。これを実現するためにstruct RStringの定義は、図1のようになっています（単純にしてあります）。

```
#define RSTRING_EMBED_LEN_MAX ((int)((sizeof(VALUE)*3)/sizeof(char)-1))
struct RString {
    struct RBasic basic;
    union {
        struct {
            long len;
            char *ptr;
            long capa;
        } heap;
        char ary[RSTRING_EMBED_LEN_MAX + 1];
    } as;
};
```

図1 struct RString

文字列情報が埋め込んであるかどうかは `basic.flags` の部分を見ると判別できるようになっています。埋め込まれていた場合には、`as.ary` にアクセス、それ以外の場合には `as.heap` にアクセスすることによって条件付き定義を実現しています。

最後の「メモリー解釈の操作」は型情報を回避して実際のメモリー上のデータに直接アクセスすることです。CPUが複数バイトからなる整数を格納するバイト順のことを「バイトオーダー」とか「エンディアン」と呼びます。例えば、32ビットの整数を構成する4バイトを、先頭から `a`、`b`、`c`、`d` と名付けたとして、「`a`、`b`、`c`、`d`」という順序で格納する方式を「ビッグエンディアン」と呼び、「`d`、`c`、`b`、`a`」という順序の方式を「リトルエンディアン」と呼びます。

普通に考えるとビッグエンディアンの方が自然に思えますが、実際のCPUが採用しているバイトオーダーはリトルエンディアンの方が圧倒的に多数派です。リトルエンディアンを採用している代表的なCPUはインテルx86で、ビッグエンディアンを採用する代表的なCPUはSPARCです。図2のプログラムはユニオンを使って、実行中のCPUのバイトオーダーを判別しています^{*1}。

```
#include <stdio.h>
#include <stdint.h>

int
little_endian()
{
    union {
        /* 正確にサイズを合わせるため
         char, intではなく
         uint32_t(32ビット),
         uint8_t(8ビット)を使う */
        uint32_t i;
        uint8_t c[4];
    } u;

    u.i = 0xa0b0c0d0;
    return u.c[0] == 0xd0;
}

int
main()
{
    if (little_endian())
        puts("little endian");
    else
        puts("big endian");
    return 0;
}
```

図2 エンディアン判定プログラム

データ構造から見るCの性格

さて、Cの基本データを一通り見てきましたが、なにか気が付いた点はありませんか？

Cの基本データ構造の特徴の一つは、整数を表現する型が1バイト(8ビット)から8バイト(64ビット)まで、各種サイズの整数型が取りそろえてある点です。さらに同サイズでも符号付きと符号なしが使えます。もう一つの特徴は、既に述べたようにポインタ(アドレス)に対する整数同様の演算が許されている点です。

これらの特徴を含めて、Cの基本データ構造は、生のCPUの機能を反映しています。ほとんどのCPUは各種サイズの整数演算をする命令を備えていますし、浮動小数点数を扱う機能も備えて

*1 このプログラムではリトルエンディアンでないものはビッグエンディアンであると判定していますが、すごく古いCPUには「`b`、`a`、`d`、`c`」と並ぶ「ミドルエンディアン」という形式もあるので、厳密には間違っています。ただしビッグエンディアンさえ絶滅寸前の現在、気にする必要はないと思います。

います。

(ほとんどの) CPUでは、データに型はなく、どの命令を使うかによって、そのデータ(バイトまたはバイト列)がどのような意味を持つかが決まります。ポインタもCPUレベルではアドレスを表現する整数にすぎませんし、整数であれば整数演算できるのも不思議ではありません。

ユニオンについても、メモリー領域に対する解釈を変更するだけですから、一見複雑に見える処理も、結局は生のCPUの動作が想像できれば自然なことです。

これらのことから分かるのは、プログラミング言語としてのCは、ある程度の移植性と、型チェックを加えた上でCPUの振る舞いを直接操作できるような言語であるということです。

Cの出自は、当時はハードウェアごとにアセンブラで開発されていたOSを、移植可能な高級言語でほとんど記述するための言語です。この性質は基本データ構造にも反映されているということです。

■ Ruby の 基 本 デ ー タ 構 造

次はRubyの基本データ構造を調べてみましょう。Cと比較するとRubyに組み込まれているデータ構造はかなり多いので、代表的なものだけ抜粋します。表3に示すのがRubyの基本データ構造です。

Cと比較すると、生のCPUが取り扱うデータ構造は姿を潜め、より抽象度の高いデータ構造が目立ちます。この辺りが言語の性格の違いといえるでしょう。

充実した機能を持つ文字列と正規表現が基本データに含まれている点は、Rubyが元々テキスト処理を主眼とするスクリプト言語として誕生したことを反映しています。最近RubyはWebアプリ開発言語として認知されることが多く、正規表現をはじめとするテキスト処理に対する要求は下がっています。誕生以来20数年の間に言語の使われ方が変化してきたことは興味深いです。

整数型は一つにすべきだった

Rubyの基本データ構造の中で、個人的に反省している点はFixnumとBignumの区別です。

これらのデータ構造は両方とも整数を表現しています。Fixnumはポインタサイズの整数で表現できる整数、Bignumはそれを越えるサイズの整数を意味しています。実装的には、Fixnumはリファレンス(ポインタ)に直接埋め込まれた整数であり、オブジェクト割り当てのためのメモリーを消費しないというメリットがあります。一方、Bignumはヒープに割り当てられたオブジェクトであり、メモリー

クラス	種別	解説
Fixnum	整数	31ビットまたは63ビット整数
Bignum	整数	多倍長整数
Float	浮動小数点数	倍精度のみ
String	文字列	
Regexp	正規表現	/abc\$/など
Array	配列	
Hash	ハッシュ	連想配列とも呼ぶ
Range	範囲	1..2など
Proc	クロージャー	ブロックとも呼ぶ
Object	オブジェクト	インスタンス変数を持つ
Symbol	シンボル	識別子を表現する型

表3 Rubyの基本データ構造(抜粋)

を消費しますが、表現できる整数の範囲に制限はありません（図3）。

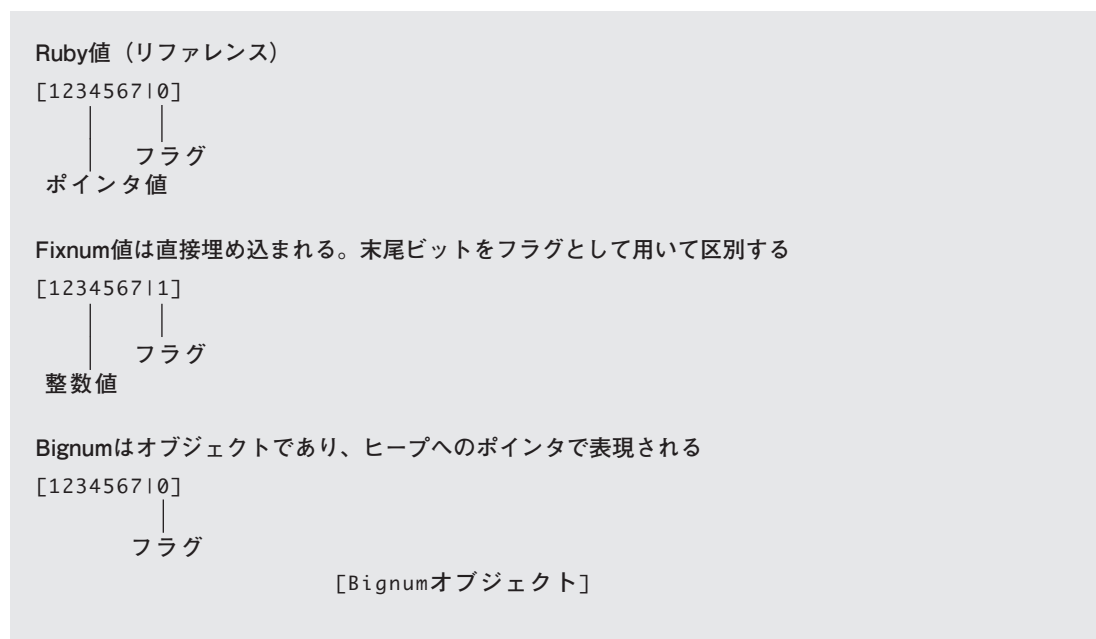


図3 FixnumとBignum

Rubyの値はCのポインタで実装される。ほとんどのOSではポインタ値は4または8の倍数なので末尾2、3ビットは0である。末尾ビットはフラグとして用い、0のときはポインタである。

CRubyではRubyの値の表現を、タグ付きポインタという手法で実装しています。通常のオブジェクトはヒープ上に割り当てられた構造体で表現され、値はその構造体へのポインタとして実装されます。しかし、CRubyを実装しているC言語ではポインタとして用いられるアドレスと整数が相互変換可能であることを利用して、小さな整数をポインタ値に直接埋め込みます。

具体的には、CPUのメモリアクセスの関係でポインタ値が4または8の倍数であり、整数に変換したときに末尾2、3ビットが必ず0であることを利用します。その末尾1ビットをフラグとして用い、残りのビットに整数値を格納します。結果として、ポインタと同じサイズの整数よりも1ビット小さいサイズの整数（32ビットアーキテクチャーであれば31ビット、64ビットアーキテクチャーであれば63ビット）はFixnumとして値に直接埋め込みます。

Fixnumの範囲を超える整数は、ヒープにBignumオブジェクトを割り当てて、そちらに格納します。

厳密にいうとBignumはあらゆるサイズの整数を表現できるのですが、演算結果がFixnumで表現できる範囲内であれば、自動的にFixnumに変換するようになっていますから、結果的に値の範囲で分担していることになります。

ということは、意味の観点から言うとFixnumもBignumも表現しているものは整数という存在

です。ただ単に実装の都合で、ある一定の範囲以内のものをFixnumで、それを越えるものをBignumで表現しているということです。

この整数の分担は、元々Lispで実現されていたもので、クラス名も含めてLispから継承しています。しかし、実装の都合による分類というのは、実装者にとっては大変意味のあることですが、言語の利用者にとってはあまり重要ではありません。重要でない区別が言語仕様に「漏れ出している」はあまり格好のよいものではありません。

浮動小数点は一つになっている

例えば、Ruby2.0で64ビットアーキテクチャー上では、Floatのうち一定範囲のものをFixnum同様にリファレンスに埋め込む最適化がされました。つまり、同じFloatクラスであっても、あるものは埋め込み値であり、あるものはヒープに割り当てられたオブジェクトになるわけです。しかし、この違いは内部的に処理され、ユーザーが気にすることはありません。今思えば、FixnumとBignumの区別も同様に、整数を表現するクラスだけを用意して、実装の切り替えはユーザーに見えない形で処理すべきだったと思います。

より抽象度が高いという点では、Luaなどいくつかの言語では整数と浮動小数点数との区別さえやめていて、Numberという一つの型で対応しています。浮動小数点数には誤差がつきものである点を考えると、躊躇するところがないわけではありませんが、一つの方向性ではあると思います。

もう一つ、Lispから継承したシンボルについても若干の反省があります。2-6で既に述べたように、シンボルと文字列の使い分けは、今となっては古いやり方です。効率や実装の都合で似ているけど異なるデータ構造を導入するのは、「今風」ではないと感じています。

このような「ちゃんと動くなら使い分けが必要ない方がよい」というのは、Rubyの設計思想の一つで、「大クラス主義」と呼ばれることもあります。

OCamlの基本データ構造

このような「使い分け」を求める言語設計の例としてOCamlを取り上げてみましょう。OCamlは関数型として有名な言語で、最近ではその開発効率と性能の高さから（海外の）ファイナンスの分野で利用されています。同じ関数型言語に分類されるHaskellと比べると、静的型や強力な型推論の点では似ていますが、暗黙の遅延評価はありませんし、やろうと思えば副作用のある操作も可能である点で、よりカジュアルであるといえるでしょう。

個人的にはHaskellよりもOCamlの方が好みだったりしますが、これはOCamlの方が優れているということではなく、あくまでも個人的な嗜好です。

さてこのOCamlですが、Rubyとは逆に積極的に使い分けを奨励するために、似たようで少し違うデータ構造を複数提供しています（表4）。

型	意味	解説
'a list	リスト	線形リスト
'a array	配列	O(1)アクセス、書き換え可能
'a * 'b	タプル	複数値をまとめたもの
{name: 'a}	レコード	構造体のようなもの

表4 OCamlの類似データ構造

OCaml のリストは、

```
[値; 値]
```

と表現します。リストの先頭に値を追加するには、`::` 演算子を使います。

```
0 :: [1; 2]
=> [0; 1; 2]
```

リストは線形リストというデータ構造で実装されています。要素へのアクセスは長さに比例した時間がかかり、先頭への要素への追加は低コストでできます (図4)。

一方、配列は値の並びです。配列 n 番目の要素へのアクセスは、 n の大きさにかかわらず一定時間でできます。しかし要素を追加するには配列全体をコピーする必要があります。

明示的な型宣言のない Ruby などとは違い、静的型の言語である OCaml では、配列やリストのすべての要素はコンパイル時に定まる共通の型を持つ必要があります。タプル (やレコード) は複数の型を持つ値をまとめるためにあります。簡単に言うと、それぞれ別の型を持つ複数の型を集めたものがタプル、それぞれの値に名前を付けたものがレコードになります。

アクセスコストや型によって複数のデータ構造を使い分けるのが OCaml 流ということができでしょう。Ruby の大クラス主義とは異なりますが、静的型のメリットを享受し、効率の良いプログラムを書くためには有効なやり方だと考えたのでしょう。また OCaml の誕生時期 (1996 年、基になった Caml は 1985 年) を考えると、これはこれで自然なことだと思います。

■ Stream の基本データ構造

さて、これらのことを踏まえて、Stream の基本データ構造について考えてみましょう。

まずは数についてです。Stream はシステムプログラミング言語としては設計されていませんから、C のように CPU が取り扱う各種サイズの整数を直接扱う必要はありません。むしろ、使い分けについて考えなくても済むように、積極的にまとめた方がよいでしょう。そこで、数はすべて `Number` という型で表現し、効率のために内部的には整数と浮動小数点数を使い分けるようにしましょう。

Cでは文字列は文字型（8ビット整数）の配列によって扱われましたが、文字列が重要なデータ型になると考えられるStreemではRuby同様専用の文字列型を導入します。Ruby同様の文字列操作のメソッドも（今後段階的に）整備します。ただし、より関数型言語の影響を受けたStreemでは、文字列型はイミュータブル（更新不可）なので、文字列を書き換えるような機能は提供しません。シンボルと文字列についても、既に述べたように使い分けをしません。内部的には専用テーブルに登録された（インターン化された）文字列をシンボルとして利用することになります。

悩ましいのは配列です。複数の値をまとめるデータ構造は、Rubyでも配列、ハッシュ、オブジェクトなどいくつも提供しています。OCamlはさらに多くのデータ構造を持っています。もちろん、使用する局面によって使い分けるためなのですが、これまでの流れからいうとユーザーが明示的に「使い分け」をしなければならない事態はできるだけ避けたいところです。

Streemのリストは「不要」

複数の値をまとめるデータ構造には、アクセスパターンによる区別（整数インデックスによる配列と名前をキーにするハッシュ）、アクセスコストによる区別（ $O(n)$ のリストと $O(1)$ の配列・ハッシュ）、型による区別（単一型のリスト・配列と複数型を持てるタプル）があります（表5）。

データ構造	アクセスコスト	型 ^{*a}	解説
リスト	$O(n)$	単一	先頭の要素追加は $O(1)$
配列	$O(1)$	単一	要素追加は複製のため $O(n)$
ハッシュ	$O(1)$	単一	副作用を前提としたデータ構造
レコード	$O(1)$	複数	構造体相当（Rubyではオブジェクト）
タプル	$O(1)$	複数	実装的には配列相当

*a 静的型言語の場合

表5 複数値をまとめるデータ構造

「使い分け」を減らしたいStreemとしては、このうちどれを採用して、どれを不採用にすべきでしょうか？

まず、ハッシュですが、よく考えるとハッシュというデータ構造は更新がないと意味がありません。ですから、原則的に副作用のないStreemにハッシュは不要です。ハッシュ相当は要素にラベルを付ける機能で代替します。ラベルの付いた配列はレコード（Rubyにおけるオブジェクト）の代替にもなります。

残るは配列とリストの違いです。配列とリストは、複数の値をまとめるという振る舞いは同じですが、内部実装が異なるためアクセスコストが異なります。しかし、実装の都合で使い分けを要求するのは避けるという設計方針からいうと、ここでその区別を残したくはありません。

設計方針は二つあり得ます。一つはどちらかを諦めて、データ構造も統一してしまう方法、もう一つは、内部的にはデータ構造を使い分け、ユーザーにはできる限りそれを見せない方法です。

ここでRubyの事例を考えてみましょう。Rubyはその開発の最初期（公開前）を除くと配列しか持たず、基本データ構造としてリストを提供していません。しかし、Rubyにリストがないからという

理由で実行コストが大きくて困るとかいう話は聞いたことはありません。いや、もちろんかつてアクセスコストの問題が発生したことが全くないわけではないでしょうが、重大問題にまで発展することはほとんどないと考えてもよいでしょう。

実は、将来必要になりそうだとということで、Streamにリストを実装しかけていましたが、今回きちんと考察した結果、どうやらリストは不要のようだという結論になりました。ソースコードの単純化のため、この辺りのコードを整理することにしましょう。

Streamのその他のデータ構造

数値、文字列、配列については既に述べました。Streamにはその他のデータ構造として、真偽値、クロージャール、I/O、タスクがあります。恐らくは今後も必要に応じて追加していくでしょうが、その場合でもできるだけ「使い分け」が不要になるように、一つの働きには一つのデータ構造（型）という方針を徹底したいと考えています。

今回の変更点

今回はデータ構造についての解説と考察が主眼でしたので、Stream処理系に手を入れた部分はさほど多くありません。

変更したのは、整数と浮動小数点数の統一化、開発途上で放置していたリスト実装の整理、配列の文字列化ルーチンの改善程度です。今回解説している時点のソースコードに「201510」というタグを打っておきます。

まとめ

今回は言語に初めから組み込まれている基本データ構造から、その言語の性質を読み解くことを試みました。CやRubyでは基本データ構造が驚くほど直接的に当初の設計目的を反映していて興味深いことです。また、言語の基本データ構造において使い分けが少ない方が優れているという知見が得られました。

また、この考察に基づいてStreamの基本データ構造について見直しました。これでより使い分けの少ない使いやすい言語に近づいたと思います。今後も引き続きStreamの改善に努めるつもりです。

Rubyにも「間違い」があった

2015年10月号掲載分です。説明の都合上、書籍では2015年9月号掲載分よりもこちらを先に解説した方がよいと考えました。

今回は各種言語（C、Ruby、OCaml）が備える基本的データ型について眺めた後、それを参考に Stream の基本的データ型を設計するという流れで解説しています。

プログラミング言語を構成する要素として、最も目立つのは文法ですが、実際には言語の性質はデータ型やライブラリによって大きく影響されます。どんなに優れた文法を持つ言語でも、データ型やライブラリが優れていなければ生き残ることはできないのです。

今回の解説では、それぞれ異なるフォーカスを持つ言語（システムプログラミング言語としてのC、スクリプト言語としてのRuby、関数型プログラミング言語としてのOCaml）のそれぞれの基本データ型がどのような思想を反映しているのか、ということをはほんの少しだけ解説しています。言語デザインの参考になるかもしれません。

今回はまたRubyのFixnumとBignumの区別についても記述しました。原稿を執筆してから1年以上経過した2016年12月にリリースされる（本コラム執筆時点では未リリース）Ruby 2.4では、とうとうFixnumとBignumが統一されてIntegerになりました。Rubyくらい世界中で広く使われていると、互換性の問題もあって、過去のデザインに「間違い」があってもなかなか直すことができません。しかし、この「Integer統合」は比較的うまくいったのではないかと思います。

4-3

オブジェクト表現と NaN Boxing

4-2のオブジェクト指向機能の設計に続いて、今回は言語処理系のデータの持ち方を改善するテクニックについて学びましょう。V7というJavaScript処理系を参考にして「NaN Boxing」と呼ぶテクニックを実装します。

ある日、V7というJavaScriptの処理系と出会いました。V8といえば、Google Chromeに内蔵されている（そしてnode.jsのコアになっている）JavaScript処理系ですが、V7は初耳です。調べてみると、組み込み向けのコンパクトなJavaScript処理系で（実装は1ファイルで17000行程度）、実行速度も高速（JITを持たない処理系としては最速を目指す）なのだそうです。

言語処理系の実装を読むのが趣味の私としては、ちょっと時間を取って調べてみることにしました。いろいろと面白かったのですが、一番興味を引かれたのはそのオブジェクト表現の実装でした。「NaN Boxing」と呼ばれるテクニックは、実はmrubyでもコンパイルオプションで利用可能なのですが、V7ではmrubyより若干洗練されたものを採用しているようです。これはもうちょっと調べてみる必要がありそうです。

クリーンルーム設計

ところでV7は、GPL2と商用ライセンスのデュアルライセンスになっています。GPL2で都合が悪いときには、作者に連絡して（有償で）商用ライセンスを取得せよということのようです。オープンソース化はしたいが、ただ乗りは避けたいという意図でしょうし、理解はできます。また、V7が目指しているような組み込み領域では、ライセンスの不安なく利用するために商用ライセンスを取得したい人（企業）もそれなりにいそうです。

しかし今回はV7を組み込みたいのではなく、Streamの参考にしたいだけなので商用ライセンスまでは取得したくありません。とはいえGPL2とStreamのMITライセンスでは矛盾が発生するので、そのままコピーするわけにはいきません。

そこで、（えせ）クリーンルーム設計を使って開発することにします。クリーンルーム設計とは、ソフトウェアのリバースエンジニアリング手法の一つです。解析チームと実装チームを隔離することで、著作権や企業秘密に抵触することなく別実装を作ります。今回はGPLが伝搬することを避けるため、V7のソースコードを解析しながら解説（本記事）を執筆し、その情報を基にStreamを開発す

るという手法を採ります。とはいえ、一人で作業するので完全な隔離は不可能ですから、あくまでもえせクリーンルーム設計です。

リファレンスの表現手法

まず、V7やStreemのような言語処理系で、オブジェクトがどのように表現されているかを紹介しておきましょう。

CPython (Cで実装したPython) のように、いくつかの処理系ではオブジェクトへの参照は構造体へのポインタで表現します。この単純ポインタ法では、構造体アクセスの速度が最速ですし、メモリーの無駄ありません。ただし、整数などのような頻繁に登場する値に対してもすべて構造体を割り当てる必要があり、大量のオブジェクト割り当てが問題になり得ます。

そこで、その点を改良する手法として「Tagged Pointer法」があります。これはポインタを整数に変換したときに、最下位2から3ビットが常にゼロである（ように多くのOSが実装されている）ことを利用する方法です。そのビットに型情報を詰め込み、整数など一部の値を参照に直接埋め込みます（図1）。処理系としてはEmacs LispやCRubyがこれを採用しています。

ポインタのビット表現		意味
[...0000 0000]	→	false
[...0000 0100]	→	nil
[...0000 0010]	→	true
[...0000 0110]	→	undef
[...xxxx xxx1]	→	整数
[...xxxx x000]	→	通常のポインタ (8バイト境界)

図1 Tagged Pointer法（Rubyの場合）

Tagged Pointer法により、ポインタだけを使うのと同じメモリー効率で、頻繁に登場する整数や真偽値などを詰め込むことで全体のメモリー効率がさらに向上します。逆に詰め込まれた値を取り出すために若干のビット演算が必要ですが、このコストは大したものではないでしょう。

現状のStreamでは構造体で表現

mruby (のデフォルト)、Streem、Luaなどではオブジェクト参照の表現に構造体を使っています。現状のStreemでは「strm_value」という型を用意しています。この実体は構造体で、図2のように定義してあります。この構造体に、ポインタや整数、浮動小数点数を格納できます。ちなみにC言語のunionでは、複数の型を一つのフィールドに詰め込みます。

```
typedef struct {
    enum strm_value_type type;
    union {
        long i;
        void *p;
        double f;
    } val;
} strm_value;
```

図2 オブジェクト参照であるstrm_valueの定義

構造体法の最大のメリットは、実装のシンプルさと移植性の高さです。構造体法は（単純ポインタ法と同様に）CPUやOSになんの仮定も置きません。Cコンパイラが提供される環境であればど

こでも実装できます。かつ、単純ポインタ法の持つ整数オブジェクトの大量割り当ての問題を回避できます。

デメリットはメモリー効率です。mrubyで用意しているmrb_value構造体は、64ビットCPUでのサイズが16バイト、32ビットCPUでは12バイトです。オブジェクト表現にポインタを使えば、そのサイズはそれぞれ64ビットCPUで8バイト、32ビットCPUで4バイトですから、無駄は明らかです。実装のシンプルさと移植性以外にメリットは見当たりません。しかしある種の実装、例えば組み込みを含めたあらゆる環境への移植性を目指すmrubyのような実装においては、移植性は譲れない条件になります。

オブジェクト表現の最後の手法がV7で採用しているNaN Boxingです。これは浮動小数点数の構造を利用して、64ビットのサイズでオブジェクトを表現するテクニックです。

IEEE754

Cの規格では浮動小数点数の表現については何も規定していません。しかし、現在のほとんどすべてのコンピュータでは、浮動小数点数の表現としてIEEE754と呼ぶフォーマットを採用しています。私の知っている範囲内でIEEE754を採用していなかったコンピュータにはVAXというミニコンピュータがありますが、かなり古いマシンですし、さすがに現役で動作しているところはないでしょう。あと、古い汎用機でも独自の浮動小数点数表現を用いていたと聞いたことがあります。

これから説明するNaN Boxingは、IEEE754のフォーマットを利用（悪用）して実現しています。まずそのフォーマットから解説します。

IEEE754で定義されている浮動小数点数は、その精度によって複数種類定義されています（表1）。各型はデータを格納するビット幅が異なるだけでほぼ同じ構造をしています。今回のテーマNaN Boxingで用いるdoubleを中心に解説します。

精度	型名	サイズ	符号部/指数部/仮数部ビット長
単精度	float	4バイト	1/8/23
倍精度	double	8バイト	1/11/52
四倍精度	long double	16バイト	1/15/112

表1 IEEE浮動小数点数の種類

浮動小数点数は、符号部、指数部、仮数部という三つの部分から構成されています（図3）。最上位ビットから、それぞれのビット幅分を切り出して符号なし整数と解釈したものをa、b、cとしたとき、浮動小数点数は以下のような意味になります（doubleの場合）。

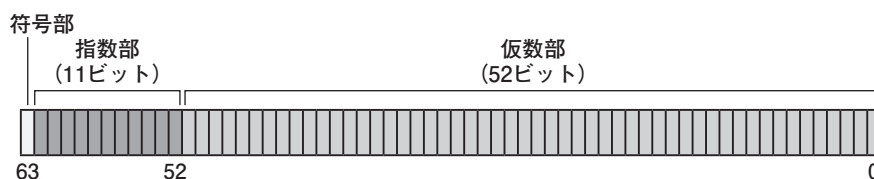


図3 IEEE浮動小数点数フォーマット

$$(-1)^a * 2^{(b-1023)} * (1+c/2^{51})$$

つまり、この式の意味は符号部が1のときは負、0のときは正であり、仮数部は1023オフセットを足して符号付き整数を表現し、指数部は小数点以下の数を2進で表現したものであるということです。

この式に従い、2.5をIEEE754 doubleで表現すると、正数だから符号部は0になり、2.5を整数部が1になるように変換して

$$1.25 * 2^1$$

となり、仮数は1.25、指数は1となります。仮数部は小数点より上の1を取り除き0.25となり、ビット表現は2進表現で

01000000（以下44ビット0が続く）

となり、指数部は1023のオフセットを加えて1024、ビット表現は

100000000000

となります。これを最上位ビットから符号部、指数部、仮数部の順に並べると、16進表現で

0x4004000000000000

となります。この値を出力するプログラムを図4に示します。前述のunionは同じビットパターンを異なる型で解釈することを許すので、このようなプログラムにぴったりです。

```
#include <stdio.h>
#include <stdint.h>

int
main()
{
    union {
        double f;
        uint64_t i;
    } u;
    u.f = 2.5;
    printf("0x%lx\n", u.i);
    return 0;
}
```

図4 IEEE754 doubleの16進出力プログラム

特殊な浮動小数点数

IEEE754には二つの特殊な値^{*1}があります。一つは無限大(Infinity)で、もう一つは非数(Not a Number、NaN)です。無限大は非ゼロの浮動小数点数を0で割ったときの結果など、数学上の無限大を示す値で、正負の二つの値があります。無限大のIEEE754の表現は、指数部が2047で仮数部がゼロになります。正負は符号ビットで決まります。

NaNは例えば0/0や $\infty + (-\infty)$ のように数学的には意味のない計算結果や負数の平方根のように実数の範囲では表現できない値を表現するために用いられます。NaNのIEEE754の表現は、指数部は ∞ 同様に2047(ビット表現で1111111111)で、仮数部がゼロ以外の場合です。

NaN Boxing

このNaNを活用して、浮動小数点数をオブジェクト表現に流用するのがNaN Boxingです。

既に述べたようにNaNは、指数部のビットがすべて1で、仮数部がゼロでないものですから、 $2^{52}-1$ 通り、つまり4503599627370495通りのビットパターンがあります。これだけの広さの空間があれば、いろいろな値を詰め込めます。NaNの隙間に詰め込む(Box)からNaN Boxingですね。

しかし、V7でオブジェクト表現に使う型は、意外にも浮動小数点数ではなく64ビット整数(uint64_t)です。これを「v7_val_t」という名前でtypedefしています。

ベンチマークを取っていないので推測ですが、浮動小数点数ではなく、整数を使う理由は恐らくその方が高速だからでしょう。関数呼び出しの引数や戻り値として値を引き渡すとき、浮動小数点数は特別扱いされるCPUがあり、そのような環境では整数の方が若干効率が良い可能性があります。NaN Boxingではビットパターンさえ渡せば、浮動小数点数そのものをオブジェクト表現に用いる必要はありません。

V7では64ビットを符号部(1ビット)、指数部(11ビット)、仮数部(52ビット)のそれぞれを以下のように解釈します。

まず符号部ですが、V7をはじめとして多くのNaN Boxingの実装では符号部を使いません。V7では常に1にしています。仮数部はNaNの定義によりすべて1になります。実際の値は残りの52ビットの領域に詰め込むことになります。

そこで仮数部52ビットのうち、先頭の4ビットをオブジェクトの種別を示すフラグとして使います(表2)。残りの48ビット(6バイト)を使って、

種別	フラグ	解説
OBJECT	0b1111	オブジェクト
FOREIGN	0x1110	外部ポインタ
UNDEFINED	0x1101	JavaScriptのundefined
BOOLEAN	0x1100	真偽値
NAN	0x1011	NaN(浮動小数点数)
STRING_I	0x1010	インライン文字列(長さ<5)
STRING_5	0x1001	インライン文字列(長さ=5)
STRING_O	0x1000	文字列(GC対象)
STRING_F	0x0111	外部文字列
STRING_C	0x0110	文字列チャンク
FUNCTION	0x0101	JavaScript関数
CFUNCTION	0x0100	C関数
GETSETTER	0x0011	getter+setter
REGEXP	0x0010	正規表現
NOVALUE	0x0001	配列用の未初期化値
INFINITY	0x0000	無限大(浮動小数点数)

表2 V7の値の種別

*1 正確には「非正規化数」というもう一つの特別な値(グループ)も存在しますが、今回は扱いません。

各種のオブジェクトを表現することになります。

まとめると、v7_val_t の64ビットのうち、符号部+指数部+仮数部の先頭4ビットの合計16ビットがオブジェクトの種別を表すタグとなり、残りの48ビットにほかの表現したい値を詰め込みます。

真偽値など詰め方

では、具体的にどのように値を詰め込むのでしょうか。

最も簡単な真偽値から見てみます。V7で真偽値 (true または false) を作る関数 v7_create_boolean() の定義は図5のようになっています。

```
v7_val_t v7_create_boolean(int v) {  
    return (!!v) | V7_TAG_BOOLEAN;  
} ▲ !!vはvが非ゼロのとき1、ゼロのとき0
```

図5 v7_create_boolean()

V7_TAG_BOOLEAN は BOOLEAN (真偽値) を示すタグ (符号部+指数部+フラグ) です。上で述べたように値部は48ビットありますが、ここでは true は1、false は0とぜいたくに使っています。

JavaScriptには存在しませんが、例えばRubyのシンボルのような値を表現するのにも同じような方法を使います。つまり、シンボルに対応する整数 (48ビット以内) とそのタグを組み合わせるオブジェクト表現を作るわけです。

整数の詰め込み方

実はJavaScriptには整数はない (数はすべて浮動小数点数で表現する) のですが、とりあえず一般論として NaN Boxing における整数の詰め込み方を解説しておきます。型表現として用いられるのが符号部、指数部合わせて16ビット、値の表現に用いるのが48ビットですから、この48ビットに整数を収めればよいわけです。

一番簡単な方法は、32ビット整数を採用することです。48ビット中16ビットは無駄になりますが、まあ、32ビットCPU時代にはほとんどの計算が32ビットで行われてきたわけですから、実害はほとんどないでしょう。多くのCPUでは48ビットという中途半端なサイズよりも32ビット整数の演算の方がはるかに効率が良いので、その点もメリットになります。

別のアイデアとしては48ビットをフルに整数表現に活用することも可能です。しかし、その場合計算は64ビット整数ですることになるので、オーバーフローに気を付ける必要がありますし、若干扱いが煩雑になる可能性があります。

浮動小数点数の場合

あ、念のためですが、NaN Boxing は浮動小数点数の使われていないビットパターンに値を詰め込むテクニックですから、浮動小数点数は加工する必要はありません。ただし、浮動小数点数を表現する型は double、オブジェクト表現の実体は64ビット符号なし整数ですから、その変換だけは必要になります。やり方は図4のときと同じで、浮動小数点数を union に格納してから、整数として取り出します。

ポインタの詰め方

さて、いろいろな値を詰め込む上で一番問題になるのはポインタです。32ビットアーキテクチャー上ではポインタのサイズも32ビットなので48ビット以内に収まるので何の問題ありません。しかし64ビットアーキテクチャーでは、ポインタのサイズも64ビットなので48ビットを超えてしまうのです。

ところが、幸いにもほとんどのOSではポインタに64ビットをフルに使うことはなくて、48ビット程度に収まる範囲の値しか使っていないのです。考えてみれば、48ビットあれば、256TBのメモリー空間にアクセスできますから当分はこれで困ることはなさそうです。

残念ながら一部のOS (Solarisとか) では、NaN Boxingがタグに使っている上位16ビットの領域もポインタが使うので、このテクニックは使えません。まあ、昔と違ってSolarisを使っている人も少なくなっているのだから問題にはならないかもしれませんが、NaN Boxingはやはり移植性が最大のネックになりますね。

ポインタが48ビットの範囲に収まっているのであれば、あとは真偽値などと同様に、TAGと組み合わせるだけで詰め込むことができます。

文字列の詰め方

V7は文字列の表現にかなり工夫があります。文字列はかなり頻繁に登場するオブジェクトですから、効率を真剣に考えたのかもしれませんが。表2を見ると、全部で16種類しかない値の種別の内、5種類も文字列のために使われているのが分かります。

まずはSTRING_IとSTRING_5です。値を表現するためのビットは48ビットもありますから、長さが6バイトの範囲の収まる文字列は値の中にそのまま詰め込めます。JavaScriptは文字列を変更できませんから、こういう手法が使えるわけです。Rubyでは文字列が変更可能ですから、この手法はそのままでは使えませんね。一方、Streamは文字列も含めてオブジェクトは変更不可(immutable) ですから、使えそうです。

(a) 1~4バイトの場合

```
v7_val_t  
[V7_TAG_STRING_I | 長さ | 1バイト目 | 2バイト目 | 3バイト目 | 4バイト目 | NUL ]  
      16ビット      8ビット  8ビット  8ビット  8ビット  8ビット  8ビット
```

(b) 5バイトの場合

```
v7_val_t  
[V7_TAG_STRING_5 | 1バイト目 | 2バイト目 | 3バイト目 | 4バイト目 | 5バイト目 | NUL ]  
      16ビット      8ビット  8ビット  8ビット  8ビット  8ビット  8ビット
```

図6 インライン文字列

C言語の文字列との整合性からか、V7では文字列の末尾にかならずNUL ('\0')を付けています。すると埋め込める最大のバイト数は5です。

1～4バイトまでの文字列はSTRING_Iで表現します(図6(a))。5バイトではバイト数を格納する余裕がありませんから、特別のタグを用意します(図6(b))。

残りの3種類の内、STRING_Fは外から与えられたV7のGC(ガーベージコレクション)では管理されない文字列です。例えばCの文字列をラップしたものがSTRING_Fになります。これらの文字列はGCの対象ではありませんから、そのままポインタとして扱うことになります。

文字列のGC

STRING_OはownedつまりV7がメモリー領域を管理する文字列です。文字列は、VMが管理する領域に割り当てられます。メモリー領域が足りなくなるとGCが始まり、文字列用のメモリー領域が回収されます。

V7は文字列についてスライド圧縮を採用しています。つまり、回収された文字列がいた隙間の領域を後側からスライドさせて詰めることで空間を有効活用します(図7)。しかし、スライドによる移動前の文字列への参照(アドレス)をスライド後のアドレスに書き換えないとデータが壊れてしまいます。

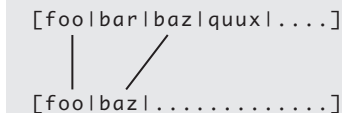
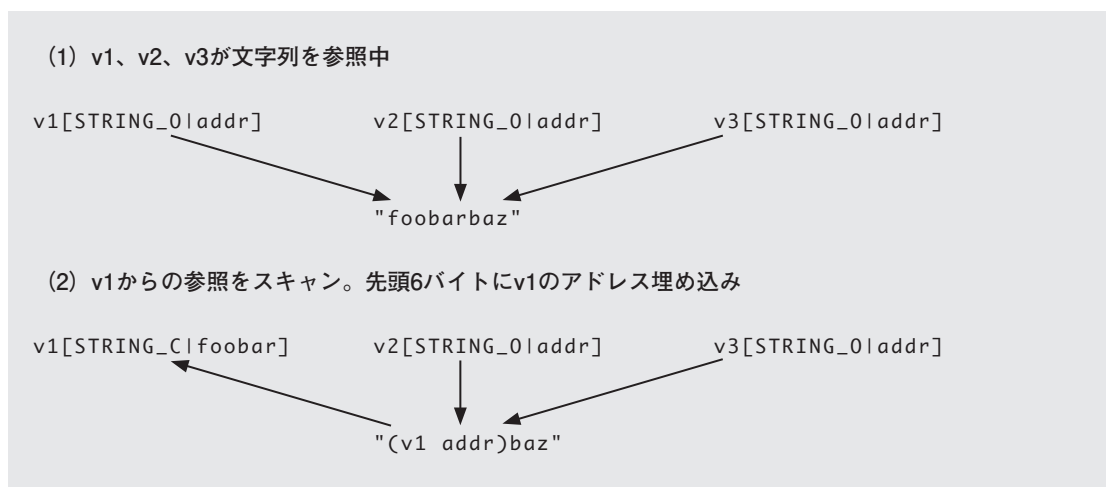


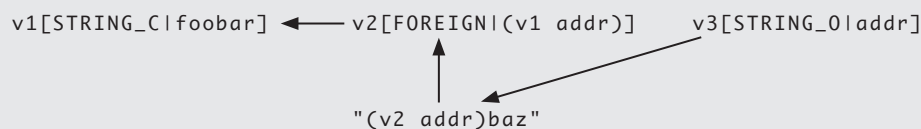
図7 スライド圧縮
"foo"と"baz"だけが「生きている」とする。

V7ではそのためにSTRING_Cを使っています。ひと目では何をやっているのか分からないくらい複雑なのですが、概略を示すと以下の手順になります(図8)。

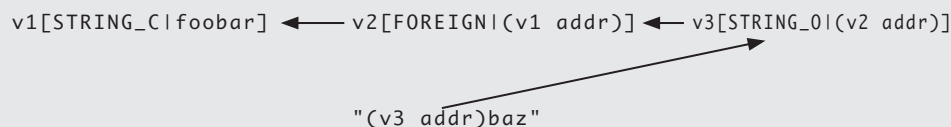
まず、GCで「生きている」文字列を判別するマークフェーズで、まだマークされていない文字列への参照vを見つけると、



(3) v2からの参照をスキャン。リンクをつなげる



(4) v3からの参照をスキャン。さらにリンクをつなげる



(5) スキャン終了後、文字列をスライドさせる。アドレスが変わる

(6) リンクをたどりアドレスを書き換える

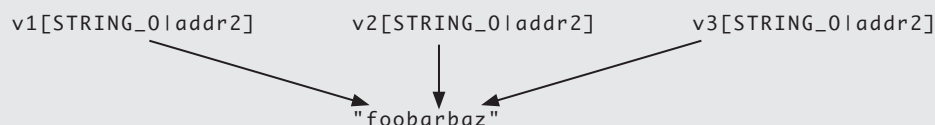


図8 文字列のGC手順

1. 文字列にマークする
2. 文字列の先頭6バイトをvに埋め込む
3. vのタグをSTRING_Cにする
4. vのアドレスを文字列に書き込む

という処理をします。

マークされた文字列への参照v2を見つけると

1. マークされた文字列に埋め込まれたアドレスをv2に埋め込む
2. v2のタグをFOREIGNにする
3. v2のアドレスを文字列に埋め込む

という処理をします。これを繰り返すと、マークフェーズが終了した時点で、「生きている」すべての文字列は

- マークされている
- そのオブジェクトへのすべての参照がリンクリストになっている
- リンクリストの末尾にアドレスで置き換えられた部分の文字列情報がある

という状態になっています。

V7のソースコードを読んでいると、この辺りの処理にCPUがリトルエンディアン*でないとトラブルが起きそうなコードを見かけました。最近はx86もARMもほとんどリトルエンディアンだからあまり気にしなくてもよいのかな。

後は文字列領域を順番にスキャンして、文字列がマークされていたらスライドさせて領域を詰め、リンクをたどってすべての参照のアドレスを更新する手順を繰り返します。

なんだか面倒臭い処理をしていますが、これには理由があります。実装のシンプルさを考えるとスライド圧縮など考えず、文字列領域をそのままmalloc(割り当て)し、使い終わればfree(解放)すればよさそうなものですが、一般にmallocは小さな領域をたくさん割り当てるとメモリーが無駄に消費されることが多いことが知られています。また、文字列のアドレスが分散することから、ワーキングセット(アクセスするアドレスの範囲)が大きくなって、キャッシュが効きにくくなることも考えられます。実行効率を考えると、この複雑さは割に合うとV7の作者は考えたのでしょう。

StroomへのNaN Boxingの採用

ここでまとめたV7のNaN Boxing実装を参考にStroomにもNaN Boxingを導入してみます。

まず、オブジェクト表現のための型strm_valueをuint64_tに置き換えます。また、オブジェクトの種別を表現するため、表3のタグを用意します。現時点では12種類しかありませんが(最大15種類)、今後必要があれば増やすかもしれません。

文字列はV7同様のテクニックを利用して、6バイト以内の文字列はstrm_valueに埋め込みます。Stroomは元から末尾にNULを置くことにしていませんから、48ビット(6バイト)を最大限活用できます。

ただ、文字列のスライド圧縮は当面導入しないことにします。ちゃんとベンチマークを取って、そこがボトルネックであると分かってから作業したいからです。

種別	解説
BOOL	真偽値
INT	整数
ARRAY	配列
STRUCT	メンバー名付き配列
OBJECT	newされた配列
FOREIGN	外部ポインタ
STRING_I	文字列(1~5バイト)
STRING_6	文字列(6バイト)
STRING_O	文字列(GC管理)
STRING_F	文字列(GC管理外)
CFUNC	C関数

表3 Stroomの値種別

.....
 【リトルエンディアン】 2バイト以上の数値データを記録するときに最下位バイトから記録する方式。最上位バイトから記録する方式はビッグエンディアンと呼ぶ。

GCの実装

これまでStreamでは、Boehm GCライブラリというC/C++向けのGCライブラリを使って独自のGC実装をサボってきました。

NaN Boxingを採用すると、ポインタ値がそのまま見えなくなるのでBoehm GCが動作しなくなります。Stream独自のGCを導入する必要があります。今回はマークアンドスイープ方式の非常にシンプルなGCを用意しました。Streamの言語の特性を生かしたGCについては、今後、設計・開発することにしてしまおう。

まとめ

今回は、オブジェクト表現のテクニックであるNaN Boxingについて、高速をうたうJavaScript処理系V7のコードリーディングをしながら学びました。またそのテクニックをStreamに導入してみました。この知識は読者が独自言語を設計し、その処理系を開発するときにも役に立つでしょう。Rubyに続く世界的な言語が読者の中から生まれることがあれば、これ以上ない喜びです。

タイムマシンコラム

GCの実装にはまだ「やる気」が足りない…

2016年1月号掲載分です。今回はコンパクトなJavaScript実装であるV7の実装からNaN Boxingの実装法を学んでいます。浮動小数点数にポインタをはじめとする値を詰め込むNaN Boxingは大変興味深い「ハック」です。今回の解説は、NaN Boxingの実装に関する資料としてそれなりに有意義なものだと思います。

ここで告白しなければならないことがあります。本文では「マークアンドスイープ方式の非常にシンプルなGCを用意しました」と書いていますが、実際には用意できていませんでした。もちろんウソをつくつもりは全くなく、原稿を書いている時点では実装するつもりでいたのですが、時間(とやる気)の関係で実装できませんでした。

以前、mrubyのGCを実装したときには、退屈な会議の間の内職で数時間で実装できたので、きっとできると思ったのですが、ちょっとやる気が足りませんでした。結局、このコラムを書いている今でもStreamにはまともなGCが実装されないままです。本当に申し訳ない。

4-4 ガーベージコレクション

4-3ではStreamのオブジェクトの表現方法を改善して「NaN Boxing」を採用しました。変更に伴い、メモリー管理の方法も新しくする必要があります。この機会にメモリー管理、特にGC(ガーベージコレクション)のアルゴリズムについて解説し、Streamでの実装について考察します。

JavaやRubyのような言語では、プログラムの実行中にオブジェクトをたくさん作ります。コンピュータから見ると、オブジェクトはデータが格納されているメモリー領域にすぎません。それがプログラム言語からはオブジェクトに見えているわけです。

同じオブジェクト指向言語でもC++のような言語では、このオブジェクトが占めるメモリー領域を人間が手動で管理しています。Cのようなオブジェクト指向以前の言語でも手動でメモリー管理する点は同様です。

Cではmalloc()という関数を使ってメモリー領域を直接割り当てられますし、C++ではnewを使ってオブジェクトをヒープ領域に割り当てます。これらの手続きは、OSにある程度の塊としてメモリー領域を切り出すように要求し、それを分割して返します。毎回OSから切り出してくるのでは効率が悪過ぎるからです。

これらの手続きによって割り当てられたメモリー領域が不要になれば、free(C)とかdelete(C++)を使って、その領域がもう不要になったことをプログラマがシステムに教えてやります。システムはあるまとまった単位で、使わないメモリー領域をOSに返すことになるでしょう。しかし、この「もう使わなくなった」というのがさまざまなトラブルのもとでした。

メモリー領域を自動的に解放

まだ利用中のメモリー領域をうっかり返却してしまうと、返してしまった領域はそのうち別の目的に再利用されてしまいます。後でアクセスしたときに、そのメモリー領域のデータは書き換えられてしまっている可能性があります。するとプログラムは誤動作したり、異常終了したりします。

逆に「また使うかも」と思ってメモリー領域をシステムに返却しないでいると、あるいは、使い終わっても返却するのを忘れていると、それも問題になります。実際にはアクセスされない領域がいつまでも残って、メモリーがムダに消費されます。そして、最終的にはパフォーマンスの低下や異常終了を引き起こします。そもそも大量に割り当てられた細かい資源をいちいち管理することは、人に

は非常に苦手なことなのです。

このメモリー管理、特にメモリー領域の解放を自動化しようというのがガーベージコレクション (GC) です。GCは実は古くからある技術で、1960年代から研究されていて論文がたくさん書かれています。大学の研究室レベルでは長らく使われていましたが、一般のプログラマが普通に使うようになったのは、1990年代のJavaの登場からといってもよいでしょう。それ以前は、知る人ぞ知る技術という感じだったようです。

この自動化というのがくせ者でした。まだGCという技術が一般的になる前、Javaが登場してやっと注目を集めるようになったばかりの頃には否定的な見方を多かったのです。「GCは信頼できない」とか「割り当てたメモリー領域は人間が明示的に（心をこめて）解放するべきものだ」とか「GCは人間が手作業でメモリー管理するよりも遅いので採用できない」などの意見が数多く聞かれました。

しかし時間がたつにつれ、Javaの普及も伴って、そんな声もあまり聞こえなくなりました。GCの技術も進歩して、人間が直接メモリー管理するよりも間違いが少ない上、多くの場合で性能が高いことも実証されてきたからだと思われます。

もちろん、デバイス組み込みのハードリアルタイムシステムなど、GC技術の適用が難しい領域はまだ残っています。しかし、そういう例外を除けば、GCはもはや一般化したといってもよいでしょう。

トレース法とリファレンスカウント法

GCには2大手法としてトレース法とリファレンスカウント法というものがあります。それぞれ両極端に当たります。

トレース法は、ルートと呼ばれる起点から始めて再帰的に参照されているオブジェクトをたどっていく(トレースする)方法です。トレース法の中には、トレースしながら「生きている」オブジェクトにマークしていき、最後にマークされていない(死んでいる)オブジェクトを回収する「マークアンドスイープ」法や、トレースによって判別した「生きている」オブジェクトを別領域にコピーして、取り残されたオブジェクトを旧領域ごとまとめて削除してしまう「コピー」法などがあります。

トレース法の良いところはルートから間接的に参照されている「生きている」オブジェクトを確実に検出できることです。逆に欠点はオブジェクトが多くなると、GCに必要な処理時間が長くなる点です。

マークアンドスイープ法

「マークアンドスイープ」法は最初期に開発されたアルゴリズムです。原理は非常にシンプルで、ルートから参照可能なオブジェクトに再帰的にマークを付け、その後、マークが付いてないオブジェクトをガーベージとして回収します。

図1にマークアンドスイープアルゴリズムの概略を示します。

まずプログラムの実行に伴い、オブジェクトが割り当てられます(図1(1))。オブジェクトはほか

のオブジェクトを参照することがあります。

GCが始まると、ルートから参照可能なオブジェクトに「マーク」が付けられます(図1(2))。マークはオブジェクト内部のフラグとして実装されることが多いです。ここではマークされたオブジェクトを黒塗りにしています。

マークされたオブジェクトから参照されているオブジェクトにもマークを付けます(図1(3))。これを繰り返すことにより、ルートから間接的に参照できるすべてのオブジェクトにマークが付きます。ここまです「マークフェーズ」呼びます。マークフェーズが終了した時点でマークが付いているオブジェクトは「生きている」と見なされます。

すべてのオブジェクトを順にスキャンして、マークが付いていなければ回収します(図1(4))。これを「スイープフェーズ」呼びます。次のGCのために、スキャンしながら、生きているオブジェクトに付いているマークもクリアしておきます。

マークアンドスイープの変種として、スイープの代わりに、「生きている」オブジェクトを詰めていく「マークアンドコンパクト」法というアルゴリズムもあります。

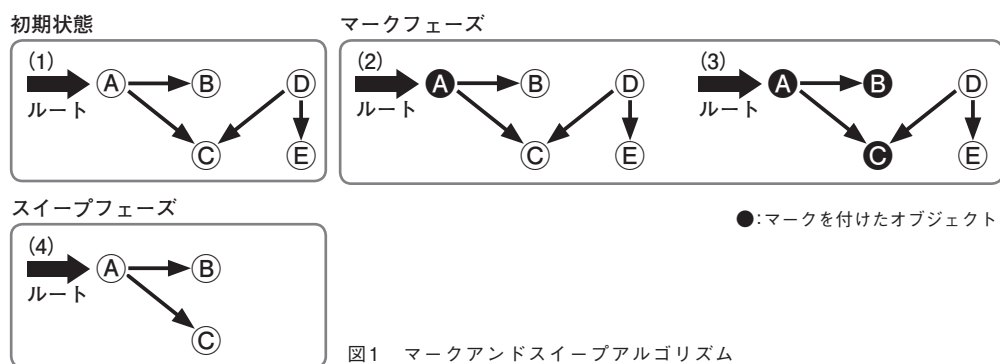


図1 マークアンドスイープアルゴリズム

マークアンドスイープ法とその変種では、処理時間は「生きているオブジェクト数」と「全オブジェクト数」の和に比例します。その結果、大量のオブジェクトが割り当てられ、そのうちのごく一部だけが生き残るような場合、スイープフェーズで大量の死んだオブジェクトをスキャンする必要があるため、必要以上に時間がかかる欠点があります。

コピー法

「コピー」法は、このような欠点を克服することを目指したアルゴリズムです。

コピー法では、ルートから参照されているオブジェクトを別空間にコピーします。そして、そのコピーされたオブジェクトから参照されているオブジェクトを再帰的にコピーしていきます。

図2(1)はGC開始前のメモリーの状態です。これは図1(1)と同じです。

次に、古いオブジェクトが存在する空間(旧空間と呼ぶ)とは別のメモリー領域(新空間)を用意します。そしてルートから参照可能なオブジェクトを新空間にコピーします(図2(2))。

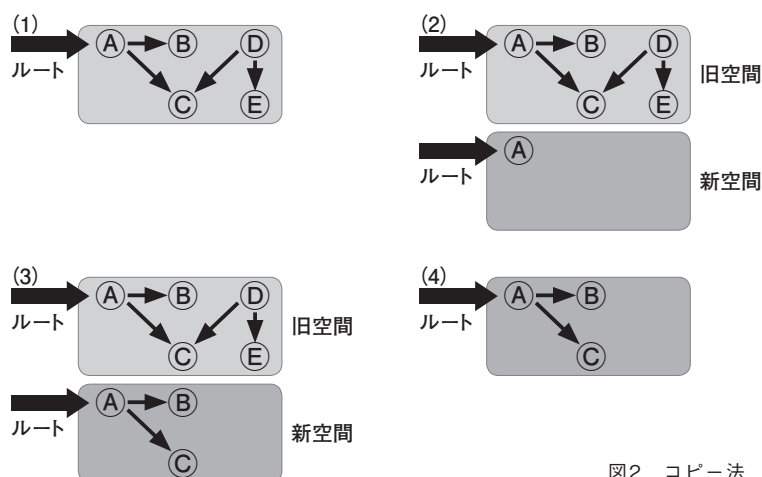


図2 コピー法

コピーされたオブジェクトから参照されているオブジェクトも芋づる式に新空間にコピーしていきます(図2(3))。コピーが終了すると、「生きている」オブジェクトはすべて新空間に移動し、「死んでいる」オブジェクトは旧空間に残されています。

ここで旧空間を破棄すると、死んでいるオブジェクトが占有していた領域は一気に解放されます(図2(4))。個別のオブジェクトのスキャンは不要です。次回のGCではこの新空間が、次の旧空間になります。

図2を見ると、コピー法には、マークアンドスイープのスイープフェーズ相当が存在しないことが分かります。マークアンドスイープ法において、大量のオブジェクトを割り当てて、そのほとんどがすぐに「死んでしまう」ようなケースでは、スイープフェーズのコストがバカになりません。コピー法では、そのコストがありません。しかし、マークを付けるよりも、オブジェクトをコピーする方がコストが高いため、逆に「生きている」オブジェクトの割合が多い場合には若干不利になります。

このアルゴリズムのもう一つのメリットには「局所性」があります。コピー法では参照されているオブジェクトから順に新しい空間にコピーしますから、関係が近いオブジェクトがメモリー空間的にも近くに配置される可能性が高まります。これを局所性と呼びます。局所性が高い場合、メモリーキャッシュなどが有効に働きやすくなるので、プログラムの実行性能が向上することが期待されます。

コピー法の欠点は、メモリー効率の悪さです。コピー中には一時的とはいえ、新旧二つの同サイズの領域を用意する必要があり、最大メモリー消費量の半分しか有効活用できません。このメモリー空間の無駄を軽減するために、領域をより細かく分割するコピー法の変種も存在します。

GCの性能指標

トレース法の基本的なアルゴリズムは大きく分けて上記のマークアンドスイープ法かコピー法のいずれかか、その変種です。

GCはプログラムの処理の本質とは無関係ですから、そのために消費される時間は短ければ短

いほどよいことになります。しかし、上記の基本的アルゴリズムは性能という点では若干の課題が残ります。

GC の性能は二つの指標で測られます。「GC 実行時間」と「停止時間」です。

GC 実行時間は、GC 処理そのものの性能になります。つまり、アプリケーションの実行時間全体のうち、GC によって消費される時間のことを GC 実行時間と定義します。

一方の停止時間は、アプリケーションが本来の処理を中断して GC を行うとき、処理が中断される時間のことです。特に最悪のケースでの停止時間（最大停止時間）が重要な指標になります。

停止時間が重要なのは、長くなるとアプリケーションの応答性に問題が発生するからです。例えば、Web サービスで 1000 人からアクセスがあったとして、999 人には 100 ミリ秒で結果を返すが、運の悪い 1 人だけは、GC のため、結果を受け取るまで 10 分かかるのはあまり望ましい状況ではありません。あるいは、ロボットを制御するソフトウェアで、ロボットが歩いている最中に GC のため 1 秒制御が停止すると、恐らくロボットは転倒してしまうでしょう。

補助的 GC 手法

そこで GC の性能を改善するために、これらの基本となるアルゴリズムと組み合わせるタイプの技法もいくつか知られています。今回はその中でも代表的な「世代別 GC」（Generational GC）と「インクリメンタル GC」について解説します。これらの技法は複数を組み合わせる場合もあり得ます。

まずは GC 技法としては最も重要だと思われる世代別 GC から解説しましょう。

世代別 GC

世代別 GC はプログラム実行時間中で GC のために消費される時間を短縮することを目指した技法です。

世代別 GC の基本的なアイデアは、一般的なプログラムでは「オブジェクトのほとんどは比較的短時間でゴミになり、ある程度長い時間生き残ったオブジェクトはより長い寿命を持つ」ことを利用することです。寿命が長いものが生き残りやすく、短いものがより早く不要になるのだとしたら、割り当てられてからあまり時間がたっていない「若い」オブジェクトを重点的にスキャンしてやります。これで全オブジェクトをスキャンしなくても多くのガーベージを回収することが期待できます。

世代別 GC では、生成されてから間もない若い「新世代」と、長生きしている「旧世代」にオブジェクトを分類します。実装によっては複数の世代を用いることもあります。

すぐに「死んでしまう」可能性が高い新世代オブジェクトだけをスキャンする回収をマイナー GC と呼びます。マイナー GC の具体的な回収手順は以下ようになります。

まずルートから普通にスキャンを始め、「生きている」オブジェクトを探します。このアルゴリズムはマークアンドスイープでもコピーでも構いませんが、多くの世代別 GC の実装ではコピーが採用されています。注目すべき点は、スキャンの途中で旧世代領域に属するオブジェクトが登場すると、その先はスキャンしない点です。このことにより、スキャンするオブジェクトの数を大幅に減らせます。

生き残ったオブジェクトは旧世代に所属させます。具体的には、コピー法の場合にはコピー先を旧世代空間にするでしょうし、マークアンドスイープ法では、オブジェクトになんらかのフラグを付けることが多いようです。

旧世代からの参照を記録

このときに問題になるのが、旧世代領域から新世代領域への参照です。新世代領域しかスキャンしないと、そのままでは旧世代領域から新世代領域への参照はチェックされません。このため旧世代領域からしか参照されていない若いオブジェクトは間違えて「死んでいる」と見なされてしまいます。そこで世代別GCでは、オブジェクトの更新を監視します。旧世代領域から新世代領域への参照が発生すると、リメンバードセット (remembered set) と呼ばれるテーブルに登録します。マイナー GC では、このリメンバードセットもルートに含めます。

世代別GCが正しく動作するためにはリメンバードセットの内容を常に最新に更新しておく必要があります。そこで、旧世代領域から新世代領域への参照が発生した瞬間にその参照を記録するルーチンを、オブジェクトを更新する場所すべてに埋め込みます。この参照を記録するルーチンをライトバリア (write barrier) と呼びます。

旧世代領域に属するオブジェクトは一般的に寿命が長い傾向があるとはいえ、決して「死なない」わけではありません。プログラムの実行に伴って旧世代領域に所属している「死んでいる」オブジェクトも増えていきます。この死んでいる旧世代オブジェクトがムダにメモリーを占有している状況を回避するために、たまに旧世代領域も含むすべての領域をスキャンするGCを行います。このすべての領域を対象にするGCをフルGCまたはメジャーGCと呼びます。

世代別GCはGCのためにスキャンするオブジェクトの数を減らし、GC実行時間を短くする効果があります。しかしメジャーGCがありますので、最大停止時間は改善されません。

インクリメンタルGC

さきほどのロボットの例でも分かるように、リアルタイム性の高いプログラムではGCの性能よりも最大停止時間が短いことが重視されます。

そのようなリアルタイム性の高いプログラムではGCによる中断時間が予測可能である必要があります。例えば、最悪でも10ミリ秒以内に完了するというような条件が付けられるわけです。

通常のGCアルゴリズムではこのような保証は不可能です。GCによる停止時間はオブジェクトの数や状態に依存するからです。そこでリアルタイム性を維持するにはGCが完全に終わるのを待たず、処理を細切れにして少しずつ実行するようにします。これをインクリメンタルGCと呼びます。

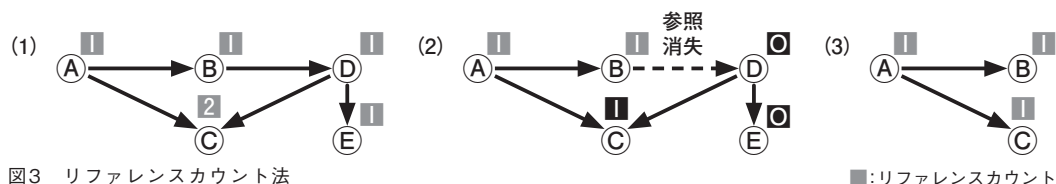
インクリメンタルGCでは、GC処理が少しずつ進むため、処理の最中にプログラム本体の実行が進行し参照が書き換わってしまいます。既にスキャンが終わり、マークが終了したオブジェクトが書き換わって、新しいオブジェクトを参照するようになった場合、その新しいオブジェクトはマークの対象にならず、まだ「生きている」のに回収されてしまいます。

インクリメンタルGCではそのような問題を避けるために世代別GCと同様にライトバリアを用います。既にマークされたオブジェクトから参照が書き換わった場合、ライトバリアによって、新たに参照されるようになったオブジェクトが、スキャンの始点となるように登録されます。

インクリメンタルGCは、処理を細切れに実行するため、中断時間を一定以内に抑える働きをします。一方、処理の中断のためのコストが必要なので、GCにかかる時間の総和は大きくなる傾向があります。これはトレードオフの関係です。

リファレンスカウント法

トレース法に対する、GCのもう一方の大分類、リファレンスカウント法は、各オブジェクトに自分に対するリファレンスカウント（被参照数）を記録しておくものです。参照が増減するたびにその数を書き換えます（図3）。



被参照数を増減させるタイミングは、変数への代入や、オブジェクトの内容の更新、関数の終了（ローカル変数からの参照がなくなる）などです。そして、被参照数がゼロになったオブジェクトはもうどこからも参照されないのが明らかなので、そのメモリー領域を解放します。

リファレンスカウント法の最大のメリットは、オブジェクトの解放を局所的に判断できることでしょう。ルートから全部のオブジェクトをたどらないとオブジェクトの生死が判断できないトレース法と違って、参照がなくなったその瞬間にオブジェクトの死亡が判定できます。また、個別のオブジェクト単位で解放するので、ほかのアルゴリズムと比較して、GCのための停止時間が短い（ことが多い）のもメリットです。

一方、デメリットもあります。リファレンスカウントの最大のデメリットは、循環参照を持つオブジェクトを解放できない点です。お互いに参照し合っているオブジェクトは、そのグループ全体がどこからも参照されずガーベージになっていても被参照数がゼロになりません。結果的にこれらのオブジェクトは永遠に解放されません。

別のデメリットとして、参照の増減のたびに正確にリファレンスカウントも増減させる必要があり、それをうっかり忘れると非常に原因を発見しづらい困難なメモリートラブルを引き起こします。

最後に、リファレンスカウントの管理は並列処理と相性が悪いというデメリットがあります。同時に複数のスレッドがリファレンスカウントを増減すると、リファレンスカウントの値に不整合が発生する（結果としてメモリートラブルの原因となる）可能性があります。それを避けるためリファレンスカウントは排他的に操作する必要がありますが、ひんぱんに発生する参照の操作のたびにロックをかけ

るといったコストは馬鹿になりません。

StreamにおけるGC

これまでStreamはBoehm GCというCやC++のプログラムに自動的にGCを追加する便利なライブラリを利用してきました。これは普通にmallocやnewで割り当てたオブジェクト(メモリー領域)を、プログラマがいちいちfree/deleteしなくても、使われなくなったことを自動的に検出して、勝手に回収してくれるという優れものです。しかし残念ながらBoehm GCには、ポインタを加工してはいけないという制限があり、4-3で導入したNaN Boxingとは両立できません。

そこでBoehm GCを諦めて独自のGCを実装する必要が出てきました。独自に実装するなら、GCの実装に大きな影響を与えるStreamの特徴を生かしたいところです。

一つ特徴は、汎用言語とは異なり、Streamではイベントループ開始後は実行がタスク単位で独立であるという点です。タスクの実行中に生成されたデータは、明示的にemitされたもの(そこから再帰的に参照されているもの)以外はほかのタスクから参照されることはありません。このことにより、emitでほかのタスクに「輸出」されたデータさえマークすれば、タスク終了時にこれまで作ったデータを一気に削除することで、タスク単位でのGCが可能になります。つまり、世代別GCと同様に、すべてのオブジェクトをスキャンしなくてもよいので、GC実行時間が短くなることが期待されます。

旧から新への参照がない

もう一つの特徴はStreamのほとんどのデータ構造が更新不能である点です。データ構造が更新不能であると、GCにとってありがたいことがいくつもあります。

まず、オブジェクトが更新不能であるということは、そのオブジェクトから参照されるものは、オブジェクト生成時には既に存在していなければなりません。ということは、古いオブジェクトはより新しいオブジェクトを参照できないので、循環参照は発生しません。既に述べたように、循環参照に対応できないことがリファレンスカウント法の欠点ですが、更新不能オブジェクトはその欠点を克服できます。

また、古いオブジェクトがより新しいオブジェクトを参照しないのなら、世代別GCを実装する場合にもライトバリアが不要になります。

さらに、コピー法を実装する場合にもメリットがあります。オブジェクトが更新可能であるとき、オブジェクトとコピーは厳密に区別されます。あるオブジェクトを更新しても、コピーは更新されないからです。ところが更新が禁止されている場合、(明示的にポインタ値などで区別しない限り)オリジナルとコピーの区別は必要ありません。通常、コピー法のGCでは、すべての参照をコピーされた新しいオブジェクトを指すために更新する必要があります。しかしオブジェクトが更新不能である場合には、既存のリファレンスは書き換えず、コピーを作るだけでコピー法を実現できます(コピー数が多くない場合に限る)。

GC 実装

そうはいつても、一度に複雑なものを作ろうとすると失敗するので、今回は以下のような戦略で実装します。

Streamの実行は基本的にタスク単位に細切れに行われます。この一つのタスク実行のたびにGCを行います。

タスク実行中にタスクからタスクへデータが渡されるとき（つまりemitされるとき）、受け渡されるデータはこのタスクの外から参照可能になるので、このデータに「生きている」マークを付けます。データが配列でほかのオブジェクトを参照していれば、それらにも（再帰的に）マークを付けます。それからグローバル変数から参照されるデータにも（再帰的に）マークを付けます。Streamではグローバル変数も書き換え不能ですから、一度参照されたオブジェクトは、プログラム終了まで「生き続ける」ことになります。

一つのタスクの実行が終了すると、そのタスク実行中に割り当てられたデータのうち、「生きている」マークが付いていないものを一気に削除します。次のタスクを始める前に「生きている」マークはクリアしておきます。

GC 実装の将来

さて、今回実装したGC機能は基本的なもので、まだまだ改善の余地があります。ここでは、今後の改善の構想について解説しておきます。

Streamの各タスクは別スレッドで動作していますが、複数スレッドから一つのデータ構造にアクセスする場合にはロックが必要となるなど、問題が発生しやすく、実行性能的にもペナルティーがあります。そこでスレッドごとにあらかじめ一定のメモリー領域を割り当てておき、データ割り当てはこのメモリー領域から切り出します。この領域はほかのスレッドからアクセスされませんから、排他制御は不要です。

また、emitでデータを渡すタスクが別のスレッドで動作しているときには、データを再帰的にそのスレッドのメモリー領域にコピーします。既に述べたように、Streamのようにデータが更新不能で空間移動するオブジェクトが限定的である場合には、簡単にコピー法を実装できます。

結果として、このGC実装は、マークアンドスイープ法とコピー法を組み合わせたものになります。また、タスク処理が終了するごとに処理中に生成されたデータに対してGCを行うことは、世代別GCの一種であるとも考えることもできます。

まだ構想中ではありますが、Streamの言語仕様の特質を利用すると、従来の汎用言語よりも効率の良いGCが実装できそうです。引き続き研究を続けたいと思います。

まとめ

GCの理論や実装の詳細については「ガベージコレクションのアルゴリズムと実装」（秀和システ

ム)や「The Garbage Collection Handbook」(洋書、Chapman & Hall)などの書籍を参照してください。新たな世界が開けるかもしれません。

Streamの言語仕様がGCに与える影響というのは興味深い点です。元々は並列実行についてだけ考えていて、GCについてはまったく考慮せずに設計した言語仕様なのですが、思わぬところで影響が出るものです。

タイムマシンコラム

ガーベージコレクションには言語の特質が影響する

2016年2月号掲載分です。今回のテーマはガーベージコレクションです。前半部分はガーベージコレクションという技術そのものの概要解説で、ページ分量の割には網羅的な解説ができています。相変わらずの自画自賛ですね。

しかし、懺悔しなければならないこともあります。後半の「StreamにおけるGC」の部分では、Streamの言語仕様がガーベージコレクションにどのように影響を与えるかを考察して、そのメリットを生かすようなガーベージコレクションの実装について解説しています。この考察そのものは間違いではないのですが、このStream専用ガーベージコレクションは、実際には実装されていません。

4-3のタイムマシンコラムでも語りましたが、この回もやる気と時間が不足していました。今回は、実際の実装の解説よりも、ある言語の特質がガーベージコレクションのような機能の実装にどのように影響を与えるかを考察したその思考過程を参考していただきたいところです。Streamのガーベージコレクションはなんとかしなければならぬのですが、このコラムを執筆している時点でもまだ手付かずです。みなさんが、この本が読む頃までには実装できているとよいのですが。

4-5 ロックフリーアルゴリズム

コンカレントプログラミングにおいて正しくない結果が出ることを避けるために、排他処理は非常に重要です。しかし、ロックをすることで実行性能を下げってしまう欠点があります。性能に響くロックをしないデータ構造とアルゴリズムを「ロックフリー」と呼びます。今回はロックフリーとは何かというところから始めて、その実装に至るまでを踏み込んで解説しようと思います。

まずコンカレント実行環境においても意図通りに動作すること（スレッド安全またはスレッドセーフ）の重要性について解説しましょう。

例として、図1のような簡単なキューの実装を題材にします。仕組みは簡単です。struct queue というデータ構造はheadとtailという二つのリンクを持ち、データの追加はtailに行い、取り出しはheadから行うことで、先に入れたものが最初に取り出される（FIFO）キューを実装しています。

```
#include <stdio.h>
#include <stdlib.h>

struct queue_node {
    void* v;
    struct queue_node* next;
};

struct queue {
    struct queue_node* head;
    struct queue_node* tail;
    //a
};

struct queue*
queue_new()
{
    struct queue* q;

    q = (struct queue*)malloc(sizeof(struct queue));
```

```

    if (q == NULL) {
        return NULL;
    }
    /* Sentinel node */
    q->head = (struct queue_node*)malloc(sizeof(struct queue_node));
    q->tail = q->head;
    q->head->next = NULL;
    //b
    return q;
}

int
queue_add(struct queue* q, void* v)
{
    struct queue_node *n;
    struct queue_node *node = (struct queue_node*)malloc(sizeof(struct queue_node));

    node->v = v;
    node->next = NULL;
    //c
    q->tail->next = node;    ← (1)
    q->tail = node;         ← (2)
    //c
    return 1;
}

void*
queue_get(struct queue* q)
{
    struct queue_node *n;
    void *val;

    n = q->head;
    if (n->next == NULL) {
        return NULL;
    }
    //c
    q->head = n->next;
    val = (void*)n->next->v;
    //c
    free(n);
    return val;
}

```

図1 キューの実装 (第1版)

提供するAPIは表1に示す三つで、これはこれから解説する後の版でも同じです。

API	機能
struct queue* queue_new()	キューの作成
int queue_add(struct queue* q, void* v)	キューへの追加
void* queue_get(struct queue* q)	キューからの取り出し

表1 キューのAPI

コンカレント実行の罠

さて、このように実装したシンプルなキューですが、複数スレッドを用いたコンカレント環境で実行すると思わぬ問題が発生します。

コンカレントに実行が行われると、同じデータ構造へ同時に参照・更新が実行される可能性があります。そうするとデータ構造が破壊されたり、データが失われたりする可能性があります。例えば以下のようなシナリオです。図1のプログラムを見ながら考えてみてください。

1. スレッドAとスレッドBが同時にキューに書き込む
2. スレッドAがキュー末尾 (q->tail) のノードの next に新しいノードをリンクし、q->tail を新しいノードを指すように書き換える (図1の「(1)」と「(2)」の部分)
3. このタイミングでスレッドBも同時にノードを追加しようとすると、運が悪いとスレッドAが書き換えた q->tail->next をスレッドBが上書きする。
4. この後、スレッドA、Bの順に q->tail の書き換えが起きると、Aが追加したノードが失われる。スレッドB、Aの順であれば、リンク不整合が発生して、head から tail へのリンクが切れ、それ以降キューに追加した要素を取り出すことができなくなる。

これは一例ですが、そのほかにもいろいろと問題が発生する可能性があります。しかも微妙なタイミングに依存して問題が発生するので、100回試して1回だけ失敗するというような大変見つけにくいバグになりがちです。

このようなバグをプログラマたちは、量子力学の不確定性原理の提唱者であるハイゼンベルグになぞらえて「ハイゼンバグ」と呼んでいます。メモリーバグと並んで、原因を発見しにくいバグとして忌み嫌っています。

排他制御の導入

このような問題を避けるための最も簡単な方法はロックの導入です。POSIXで定義されている pthread ライブラリには、排他制御をするためのロックである

```
pthread_mutex_t
```

というデータ型が定義されています。mutexとは相互排他 (mutual exclusive) の略です。この mutex を使うとキュー操作を排他制御できます。修正は簡単です。

まずstruct queue構造体にpthread_mutex_tのメンバーlockを追加します(図1//aの位置)。キューを初期化するqueue_new()関数においては、

```
pthread_mutex_init(&lock, NULL);
```

というロックの初期化処理を追加します(図1//bの位置)。

queue_add()とqueue_get()の関数のうち、参照や更新を排他的に実行する必要がある部分を

```
pthread_mutex_lock(&lock)
pthread_mutex_unlock(&lock)
```

で囲みます(図1//cの位置)。途中でreturnなどで抜ける場合にもアンロックを忘れないようにします。

それ以外の部分は図1のプログラムと共通なのでプログラム全体の掲載はしません。

このように一貫性の維持のために、同時にアクセスされることを避けるべき領域をmutexによって保護することで、データ構造をスレッドセーフにできます。いったんロックがかかると、ほかのスレッドが同じ領域を実行するためにロックをかけようとした時点で実行が停止され、アンロックされるまで待たされることになります。よってロックとアンロックで囲まれた領域では実行できるスレッドは一度に一つだけであり、同時実行のトラブルは発生しないことになります。

ロックの問題

さてmutexロックを使った排他制御は元のプログラムに対してあまり大規模な変更をせずにスレッドに対応できる利点があります。

しかし、あるスレッドがロックをかけている間は、ほかのスレッドはロックが解除されるのを待つしかありません^{*1}。マルチスレッドプログラミングを用いてコンカレント実行する目的は、できるだけ並列に動作して性能を上げたいことでしょうから、待ちによる停止はあまりうれしくありません。特に頻繁にアクセスされるデータ構造では排他制御の待ちによるロスが危惧されます。

そこで待ちがない(wait free)ようにロック無しでコンカレント実行できる仕組みがロックフリーです。

ロックフリーとは

ロックフリー(lock free)とは、文字通りロックを使わないということです。排他制御のためのロ

*1 正確にはpthread_mutex_trylock()というロックがかかっているかどうかチェックする関数は存在します。が、それだけでロックフリー操作ができるわけではないので、使い勝手はよくありません。

クを使わずに、複数スレッドからのアクセスを矛盾なく処理できるのだろうか」と疑問に思う方もいらっしゃるでしょう。

ロックフリーなデータ構造は「アトミック」操作という魔法によってコンカレント実行を実現します。

アトミックは原子力という意味ではありません。「アトム」に「分割できないもの」という意味があることから、処理中に割り込みなどによって中途半端な処理状態が観測されないことが保証されていることをアトミックと呼びます。

アトミックな処理は、途中で割り込みなどが入ることもなく、処理全体が成功するか、あるいは前提条件が成立せずに失敗するかのいずれかです。というか、コンカレント処理に慣れていない人（私もそれほど慣れているわけではありません）には、こんなことさえも保証されていないのかと驚く限りです。

コンピュータにおける分割できない操作というのは意外に多くありません。CPUの1命令はアトミックに実行されそうな気がしますが、実は最近のCPUでは機械語の1命令も内部的には複数の「 μ op」という小さな命令群に分割され、多くは最適化されてから実行されています。つまり機械語1命令であっても、厳密にはアトミックであるとは限りません。ということは、普通の方法ではどうやってもアトミックな操作はできないわけです。

CPUがアトミックな命令を持つ

しかし、アトミック性はコンカレント環境において非常に重要です。スレッド操作ライブラリなどでも（例えばmutexの実装に）かならずアトミックな操作が使われています。そのため、現代のほとんどのCPUは、アトミックであることを保証する命令を用意しています。このような命令の代表格にcompare and swap (CAS) というものがあります。

CASは三つの引数を取ります。

```
CAS(a, b, c)
```

aはアドレス、bとcは整数値です。これは「aのアドレスがbであるときにはcに置き換えて真を返す、そうでなければ偽を返す」という働きをします。つまり、CASを使えば、あるアドレスのデータを読み込んでから、その値を加工し、新しい値を書き戻すまでに、そのアドレスがほかのスレッドによって更新されていないことを確認できます。

元々C言語にはそのような命令はありませんが、最近のGCC (ver4.1以降) は、拡張命令として

```
__sync_bool_compare_and_swap()
```

という命令が追加されていて、CASを使うことができます。

そして、このCAS命令を使うことでロックフリーデータ構造を構築できます。

ロックフリーキュー

さて、このCAS命令を使ったロックフリーデータ構造の実装を見るために、図1のプログラムとAPI互換なロックフリーキューの実装を図2に示します。

```
#include <stdio.h>
#include <stdlib.h>

struct queue_node {
    void* v;
    struct queue_node* next;
};

struct queue {
    struct queue_node* head;
    struct queue_node* tail;
};

struct queue*
queue_new()
{
    struct queue* q;

    q = (struct queue*)malloc(sizeof(struct queue));
    if (q == NULL) {
        return NULL;
    }
    /* Sentinel node */
    q->head = (struct queue_node*)malloc(sizeof(struct queue_node));
    q->tail = q->head;
    q->head->next = NULL;
    return q;
}

int
queue_add(struct queue* q, void* v)
{
    struct queue_node *n;
    struct queue_node *node = (struct queue_node*)malloc(sizeof(struct queue_node));

    node->v = v;
    node->next = NULL;
```



```

while (1) {
    /* tailを更新する */
    n = q->tail;
    /* tail->nextはNULLであるはず。であればnodeを追加して次へ */
    if (__sync_bool_compare_and_swap(&(n->next), NULL, node)) {
        break;
    }
    /* 失敗した（ほかのスレッドが更新した）のでq->tailの書き換えを試みる */
    else {
        __sync_bool_compare_and_swap(&(q->tail), n, n->next);
    }
}
/* q->tailを更新する */
__sync_bool_compare_and_swap(&(q->tail), n, node);
return 1;
}

void*
queue_get(struct queue* q)
{
    struct queue_node *n;
    void *val;

    while (1) {
        /* q->head->nextを取り出す */
        n = q->head;
        /* キューが空ならNULLを返す */
        if (n->next == NULL) {
            return NULL;
        }
        /* q->headの更新 */
        if (__sync_bool_compare_and_swap(&(q->head), n, n->next)) {
            break;
        }
    }
    /* 値の取り出し(先頭は番兵) */
    val = (void *) n->next->v;
    /* 使わなくなったノード(元番兵)の解放 */
    free(n);
    return val;
}

```

図2 キューの実装（第3版）
ロックフリーデータ構造を利用した例。

図2のプログラムを見ていて興味深い点がいくつかあります。一つは構造体の定義が図1と図2で全く変わらない点です。ロックフリーキューでは、`pthread_mutex`のようなロックの追加などは不要です。

もう一つは構造体メンバーの更新にCASを用いていることです。すでに説明したようにCASでは更新すべきアドレスの「現在あるはずの値」と「新しい値」を指定します。もし「現在あるはずの値」と実際のアドレスの内容が異なっていれば、それはほかのスレッドが値を書き換えてしまっているということなので、更新をやり直すことになります。

実際に`queue_add()`や`queue_get()`の定義を比較してみましょう。図1のプログラムでは(途中でほかのスレッドによって更新されることを考慮していないので)、単純に代入によって更新しています。一方の図2のプログラムでは、CASによって更新が成功するまで繰り返すループになっています。

処理順序の保証

連載時の前回(本書では5-2)では、「順序はあまり重要でないので、全体で一つのキューを用意し、そこにデータを処理するタスクを突っ込む」という解説をしました。

しかし、実際にそのような仕組みを作って動作させてみたところ、非常にシンプルなケースでは正しく動作するのですが、ある程度複雑になってくると期待と異なる動作が目立つようになってきました。

一つは暗黙に順序を期待している処理が思ったよりも多かった点です。例えば、しばらく前に解説したCSV(comma separated values)処理ですが、「最初の行が文字列のフィールドばかりだった場合には、それらを各フィールドの名前とみなす」という機能を持っています。しかし、順序が保証されないということは、「最初の行」が正しく与えられるとは限らないわけです。それ以降の行であれば順序の入れ替えは問題にならないのですが。

またファイル入出力においても、多くの場合、行の順序が変化しないことを期待しています。例えば、読み込んだ文字列を大文字化するようなフィルターをStreamで記述したとして、読み込んだファイルの行が前後するのはあまり望ましくないでしょう。

そこで順序を保証するために、キューの持ち方を変更することにしました。具体的には順序が保存されるデータキューをストリームごとに持ち、タスクの実行を予約するストリームを保持するキューをグローバルに用意しました。

`emit`をすると、データとそれを処理する関数をストリームごとのキューに、実行待ちをしているストリームをグローバルキューに追加します(図3)。

まとめると、`strm_emit()`は、以下の手順でデータを次のストリームに対してemitします。

1. 次のストリームのキューにタスク(関数、データの組)を追加する
2. 次のストリームをグローバルキューに追加し、タスク実行を予約する
3. `func`が指定された場合、自ストリームのキューにタスクを追加し、低優先度キューにストリームを追加し、タスク実行を予約する

```

void
strm_emit(strm_stream* strm, strm_value data, strm_callback func)
{
    /* dataがnilでなければqueueに追加 */
    if (!strm_nil_p(data)) {
        /* dst:下流のストリーム */
        strm_stream* dst = strm->dst;
        /* 下流ストリームのキューに関数とdataを保持するstrm_taskを追加 */
        strm_queue_add(dst->queue, strm_task_new(dst->start_func, data));
        /* 下流ストリームでのタスク実行を予約するためキューに追加 */
        strm_queue_add(queue, dst);
    }
    /* funcが指定されるとこれもキューに追加 */
    if (func) {
        /* dataに意味はないのでnilを指定 */
        strm_queue_add(strm->queue, strm_task_new(func, strm_nil_value()));
        /* 低優先度キュー (prod_queue) にストリームを追加 */
        strm_queue_add(prod_queue, strm);
    }
}

```

図3 新しいキューを使うemit

低優先度キューが必要な理由は、プロデューサーの実行があまりに頻繁だとキューが長くなり過ぎてしまい、無駄な空間を消費してしまうため、プロデューサー実行の優先度を下げたいからです。

将来的にはもうちょっと別の方法で流量制御をしたいと思っています。

排他制御

もう一つ、実際にやってみて発生した問題は排他制御です。同じストリームに属するタスクが複数同時に実行されると、データ競合が起きる可能性があります。個々のタスクはスレッドセーフになっていないからです。

これまでの実装ではストリームが実行されるワーカーレッドを固定することで、同じストリームのタスクが同時実行されないようにしていました。しかし、CPU コアの有効活用という観点からは、ワーカーレッドの固定はあまり望ましい方法ではありません。

そこで、ワーカーレッドごとの処理ループの実装を大きく変更することにしました (図4)。

```
static void*
task_loop(void *data)
{
    strm_stream* strm;

    for (;;) {
        /* キューからストリームを取り出す */
        strm = strm_queue_get(queue);
        /* キューが空なら低優先度キューから取り出す */
        if (!strm) {
            strm = strm_queue_get(prod_queue);
        }
        if (strm) {
            /* strm->exclをフラグに排他制御 */
            if (strm_atomic_cas(strm->excl, 0, 1)) {
                struct strm_task* t;

                /* ストリームごとのキューにあるタスクを全部処理 */
                while ((t = strm_queue_get(strm->queue)) != NULL) {
                    /* タスク実行 */
                    task_exec(strm, t);
                }
                /* フラグを下げる */
                strm_atomic_cas(strm->excl, 1, 0);
            }
        }
        /* ストリームが全部閉じられたらループ終了 */
        if (stream_count == 0) {
            break;
        }
    }
    return NULL;
}
```

図4 修正したワーカースレッドループ

ループ処理の手順は、次のようになります。

1. キューからストリームを取り出す
2. ストリームがタスクを実行中でなければフラグ (strm->excl) を立ててから、現時点でたまっているすべてのタスクを実行
3. ストリームがタスクを実行中だったら (フラグが立っていれば)、そのままスキップ

複数のタスクがキューに登録されるタイミングでは、フラグが立っていることで無視されるストリームが発生することになりますが、こちらはどうせタスク実行をスケジュールすることだけが目的なので問題ありません。フラグが立っているということは、そのストリームに対する処理が実行中ということなので、実行中であればキューにたまっているタスクをすべて実行するというのですから。

とはいえ、効率が悪いことは確かなので、もうちょっとマシな実装はないか今後検討することにししょう。

図4の関数で

```
strm_atomic_cas()
```

という関数を呼び出していますが、これはGCCの

```
__sync_bool_compare_and_swap()
```

を呼び出すマクロです。私自身がこの拡張命令の名前をどうにも覚えられないので苦肉の策です。

図4でstrm_atomic_cas()を使っている部分は、スレッドセーフなフラグの使い方の典型例になります。

ロックフリー演算

GCCがバージョン4.1以降で提供しているアトミック演算はCASだけではありません。Streamではそのような演算をまとめたatomic.hというヘッダーを用意しました(図5)。現状ではGCCの拡張命令を用いたマクロだけを用意しています。将来別のコンパイラ(例えばVisual C++)をサポートするときには、このヘッダーを変更することで対応することができるはずです。

```
#define strm_atomic_cas(a,b,c) __sync_bool_compare_and_swap(&(a),(b),(c))
#define strm_atomic_add(a,b) __sync_fetch_and_add(&(a),(b))
#define strm_atomic_sub(a,b) __sync_fetch_and_sub(&(a),(b))
#define strm_atomic_inc(a) __sync_fetch_and_add(&(a),1)
#define strm_atomic_dec(a) __sync_fetch_and_sub(&(a),1)
#define strm_atomic_or(a,b) __sync_fetch_and_or(&(a),(b))
#define strm_atomic_and(a,b) __sync_fetch_and_and(&(a),(b))
```

図5 atomic.h

せっかく使えるようになったロックフリー演算ですから、Stream処理系の何カ所かでも利用することにしました。

まずはストリーム処理の終了検出のためにアクティブなストリームの数をカウントする変数 `stream_count` の増減に `strm_atomic_inc()` と `strm_atomic_dec()` を使います。通常のインクリメント演算子ではアトミック性の保証がないので、タイミングによって正しくカウントできない可能性があります。`strm_atomic` シリーズであればその心配はありません。

`stream_count` がゼロになったということは、もう処理すべきストリームが存在しないということです。安心して、イベントループを終了できます。

ストリームの生存管理のための被参照数（リファレンスカウント）の維持にも、同様に `strm_atomic_inc` と `dec` を用いています。「混合」などでストリームが複数の上流ストリームから参照されている場合には、上流すべてのストリームが終了した場合に、下流のストリームを「閉じる」ことができます。そのため、ストリームごとに何カ所から参照されているかどうかをカウントしています。ストリームが結合されるたびにリファレンスカウントを増やし、上流ストリームが閉じるたびにリファレンスカウントを減らします。ゼロになったら「閉じる」処理をします。

後は、ストリームの要素数を数える `count()` にも使えそうな気がします。しかし当面タスク処理は排他制御が行われているので、あえてアトミック性の保証は要らないかなと思ってまだ手を付けていません。

まとめ

アトミック命令とそれらに基づいたデータ構造は、マルチコアを有効に活用するために効果があります。今回はロックフリーデータ構造の原理と実装について解説しました。

ロックフリーアルゴリズムの実装にも問題があった

2016年5月号掲載分です。今回も懺悔回です。もちろんロックフリーアルゴリズムの説明そのものは間違っていないのですが、今回解説したコードにはいくつかの問題があります。ので、このコードをコピーしてみさんのソフトウェアに流用するのはお勧めできません。

最初の問題は、ABA問題です。今回はロックフリーアルゴリズムは、操作中データが変更されていないことを確認するために次の戦略を採っています。CASを使って今回の操作前の値と一致しなければ、ほかのスレッドが割り込んで操作したとみなして処理をやり直すという方法です。以下のようなイメージです。

1. スレッド1:古い値を保存し、更新処理を開始
2. スレッド2:割り込んで更新をして、処理が完了
3. スレッド1:古い値が変化しているかをチェックし、変化があればやり直し

ところが、スレッド2が書き換えた古い値がfree()後、malloc()で再利用されるなどの理由で、更新した新しいはずの値がたまたま「古い値」と同じ値である可能性は低いもののゼロではありません。このようなケースでは、ロックフリーアルゴリズムは割り込みを検出できません。これがABA問題です。ABA問題が発生すると他スレッドでの更新が検出できず、データの一貫性が破壊されることがあります。ABA問題はロックフリースタックで頻発しますが、キューでも発生しないわけではありません。

この問題を解決するための方法はいくつかあって、ひとつはハザードポインタというデータ構造を使うことです。ハザードポインタはまだ利用されている値の解放を遅延するためのデータ構造で、値が参照されている間はアドレスが再利用されないで値が衝突することはありません。別の方法としてはガーベージコレクションを使うことです。ガーベージコレクションでも利用中のポインタは再利用されませんからABA問題を回避することができます。

このABA問題のせいなのか現時点では明らかになっていませんが、今回紹介したコードを高負荷で運用するとデータの取りこぼしなどが起きることが分かっています。これもこのコードの流用がお勧めできない理由です。

第5章

ストリーミング
を強化する

5-1

パイプラインプログラミング

しばらく言語処理系の実装の話を続けてきましたが、今回からはライブラリ強化の話をしましょう。今回はStreamでCSVデータの処理やブロック崩しゲームの実装を考えます。そこで必要になった状態を管理するための「組み込みデータベース」も実装することにしました。

Streamの特徴であるパイプラインプログラミングですが、通常のプログラミングスタイルとはかなり異なります。パイプラインプログラミングのスタイルについて、以前簡単には解説しました。今回はもうちょっと具体的に、さまざまなタイプの処理をパイプラインならどう書くかについて考察してみましょう。パイプラインプログラミングの意外な世界が見えるかもしれません。

以前、タスクのパイプラインを構成するときの代表的なパターンについて解説しました。紹介したのは以下のようなものです。

- 供給者・消費者パターン
- ラウンドロビンパターン
- 放送パターン
- 集約パターン
- 要求・応答パターン

しかし、そのときには具体的なプログラムについては深く話さなかったもので、あまりイメージできなかったかもしれません。今回はもうちょっと踏み込むことにしましょう。

データ集計

Streamの最も得意そうな処理といえば、集計処理でしょう。データを読み込む、条件に合うものを選択する、データを加工する、データをカウントするといった処理はどれも、とても簡単にパイプラインで組み立てられそうです。

ここでは例としてCSVデータを対象に、いくつかの簡単な集計処理をしてみましょう。

まず、図1に示すようなCSVデータがあったとしましょう。これはプログラミング言語とそのリリース年、作者名のリストです。

このリストから21世紀にリリースされた言語を選んでみます。2001年以降にリリースされた言語を表示するStreemプログラムは図2のようになります。

このリストの中で21世紀生まれはClojure、Scala、Elixirの三つになりますか。気持ちとしてはStreemも加えたいところですが、まだちゃんと使い物になるレベルに到達していないのでおこがましいというところでしょう。

では、20世紀生まれの言語はいくつあるでしょう。このリストは小さいので人間が数えても大差ないのですが、もっとずっと大きなCSVデータであれば意味があるでしょう。要素の数を数えるためにはcount()関数を使います(図3)。

```
year,name,designer
1995,Ruby,Yukihiro Matsumoto
1987,Perl,Larry Wall
1991,Python,Guido van Rossum
1995,PHP,Rasmus Lerdorf
1959,LISP,John McCarthy
1972,Smalltalk,Alan Kay
1990,Haskell,Simon Peyton Jones
1995,JavaScript,Brendan Eich
1995,Java,James Gosling
1993,Lua,Roberto Ierusalimschy
1987,Erlang,Joe Armstrong
2007,Clojure,Rich Hickey
2003,Scala,Martin Odersky
2012,Elixir,José Valim
```

図1 プログラミング言語リスト

```
fread("lang.csv")|csv()|filter{x->x.year>2000}|stdout
```

図2 21世紀生まれの言語を選ぶStreemプログラム

```
fread("lang.csv")|csv()|filter{x->x.year<2001}|count()|stdout
```

図3 20世紀生まれの言語の数を数えるStreemプログラム

このリストの中で20世紀生まれの言語は11個です。やはり、このような処理は簡単ですね。

Webサービスとパイプライン

しかし、Streemは別にデータ集計専用の言語というわけではありません。CSVの集計以上のことにもパイプラインプログラミングは適用可能です。例えばWebテクノロジーへの適用について考えてみましょう。

現代のソフトウェアの多くはWebテクノロジーを使って構築されるようになりました。CDやDVDなどのメディアから手元のPCにインストールするソフトウェアはめっきり少なくなって、ブラウザーからアクセスするようなものばかりです。ビジネス向けのソフトウェアでさえ、Webテクノロジーを使って開発されています。

一方、スマートフォンではブラウザーでソフトウェアを使う行為はあまり一般的ではありません^{*1}。アプリをインストールするのが日常です。しかし、そのアプリでさえ、HTTPを使ってサーバーサイドのAPIから情報を取得するのが一般的です。

つまり、手元にインストールされるアプリであろうが、ブラウザーからアクセスされるサーバーサイドのソフトウェアであろうが、現代のソフトウェアのほとんどはWebテクノロジーと無縁ではあり得ません。

HTTPのソフトウェア構成

HTTP (Hyper Text Transfer Protocol) を中心にしたソフトウェアの構成は図4のようになります。

まず、クライアントはサーバーに要求（リクエスト）を送ります。サーバーはその要求に従って処理をして、その結果（レスポンス）をクライアントに送り返します。厳密な話をするといろいろと例外的なパターンも存在しますが、HTTPの基本はこのリクエスト、レスポンスというシンプルなパターンです。

HTTPリクエストには表1のような種別があり、要求の種別を示すことができます。

HTTPを経由したAPIアクセスを考える場合には、サーバーをデータベースとみなして表2のような対応にすることが望ましいといわれています。このようなデータベースに対するCREATE、READ、UPDATE、DELETEという操作を頭文字を取ってCRUDと呼びます。

サーバーサイドアーキテクチャー

図4のHTTPを構成するソフトウェアのうち、サーバーサイドに注目してみましょう。サーバーサイドのソフトウェア構成は図5のようになります。クライアントの接続管理、HTTPリクエストの構文解析、プロセス管理などは主にHTTPサーバーが引き受けます。サーバーサイドソフトウェアは、その解析済みの情報を受け取り、処理をして、その結果をHTTPサーバーに返します。そうすると、HTTPサーバーの方でHTTPレス

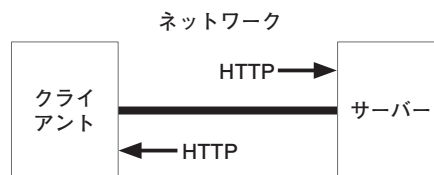


図4 HTTPのソフトウェア構成

種別	意味	使い方
GET	取得	ページ情報の取得
PUT	更新	ファイルのアップロードなど
POST	作成	ボディー部にフォーム情報あり
DELETE	削除	リソースの削除
HEAD	ヘッダー	ヘッダー情報のみの取得

表1 HTTPリクエストタイプ

DBアクセス	意味	リクエストタイプ
CREATE	作成	POST
READ	取得	GET
UPDATE	更新	PUT
DELETE	削除	DELETE

表2 DBアクセスとしてのHTTPリクエスト

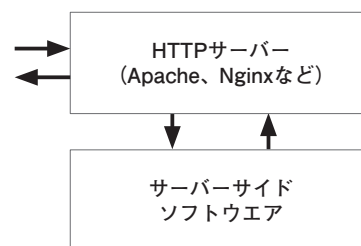


図5 サーバーサイドソフトウェア構成

^{*1} 最初期のiPhoneにはアプリというものがなく、すべてブラウザー経由でアクセスすることになっていました。アプリの登場によってPCとは逆の変化が起きたことは興味深いですね。

ポンスを組み立て、クライアントに送り返します。

つまり、サーバーサイドプログラムのレベルで見ると、HTTPリクエストの情報を受け取り、加工して、その結果をHTTPレスポンスとして返すという処理になります。この処理はパイプラインで記述できます。

具体的な例として、簡単なToDoアプリを考えてみましょう。認証などはHTTPサーバーで実行されるとして、サーバーサイドソフトウェアは「表示」「作成」「編集」「削除」という四つの処理を受け付けます。

この処理は前述のCRUDに従い、それぞれGET、POST、PUT、DELETEの各リクエストタイプで受け付けます。コードは図6のようになります。図6はStreamのコードですが、まだHTTPをサポートするライブラリなどは完成していないので、あくまでも将来予想図です。

図6ではクライアントに送るデータを作る手続きrender()の内容は示していません。しかし状態（成功または失敗）とユーザーIDから、表示すべき内容のHTML（APIの場合ならJSONかもしれない）を構築する手続きであると考えてください。

render()が作った結果（表示用データ）がhttp_response()に渡されて、そこからHTTPサーバーに送られます。

図6のプログラムはあくまでもイメージですが*2、この例によって通常のプログラミングをパイプラインで表現する方法について、少しはイメージできたのではないかと思います。

ビデオゲームの例

もう一つ、パイプラインで記述するのが難しそうな例を考えてみましょう。ブロック崩しのようなシ

```
db = kvs()
http_request() | map{req ->
  if (req.type == "GET") {           # 表示
    emit :ok
  }
  else if (req.type == "POST") {     # 作成
    db.put(req.todo_id, [req.title, req.due])
    emit :ok
  }
  else if (req.type == "PUT") {      # 編集
    db.put(req.todo_id, [req.title, req.due])
    emit :ok
  }
  else if (req.type == "DELETE") {  # 削除
    db.put(req.todo_id, nil)
    emit [:ok, req.user]
  }
  else {                             # その他（エラー）
    emit :error
  }
} | render() | http_response()
```

図6 Streamによるサーバーサイドソフトウェア

*2 Streamの周辺ライブラリが貧相で全く実用的でないのはお恥ずかしい限りです。

ンプルなビデオゲームを作るとします。

残念ながら現状の Stream でビデオゲームが書けるというわけではありません。ここでは、仮にリアルタイムキー入力とグラフィックス出力のライブラリが存在すると仮定して話を進めます。

ビデオゲームをパイプラインで記述するためにはどうしたらよいでしょう。方法はいろいろ考えられそうですが、今回は図7に示すように、三つのパイプラインを用意する方法を考えました。

第1のパイプラインは、ユーザーからの入力を受け付けて、自機（パドル）の位置を更新するものです。

第2のパイプラインは、定期的に呼び出されて、それぞれの物体（ボールの位置やブロックの状態）を更新するものです。

第3のパイプラインは更新した結果のグラフィックス出力をします。

```
# 更新頻度（毎秒30回）
fps = 30
# 更新間隔
tick = 1/fps*1000
# 盤面を表すメモリーデータベース
board = kvs()
kvs.put(:paddle_x, 0)
kvs.put(:ball_x, 0)
kvs.put(:ball_y, 0)
kvs.put(:ball_x_vec, 1)
kvs.put(:ball_y_vec, 1)
kvs.put(:num_balls, 5)

key_event() | each{x ->
  if (x == "LEFT") {
    board.update(:paddle_x, {x -> x-1})
  }
  else (if x == "RIGHT") {
    board.update(:paddle_x, {x -> x+1})
  }
}

timer_tick(tick) | each{x ->
  # ボール位置の更新
  x_vec = board.get(:ball_x_vec)
  y_vec = board.get(:ball_y_vec)
  x = board.update(:ball_x, {x -> x + x_vec})
  y = board.update(:ball_y, {y -> y + y_vec})
}
```

```

# 衝突判定
if (x == 0) { # ボールが一番下にきた
    if (ball_hit_paddle()) {
        board.update(:ball_x_vec, {x -> -x})
        board.update(:ball_y_vec, {y -> -y})
    }
    else { # 落ちた
        n = board.update(:num_balls, {n -> n-1})
        if n == 0 {
            game_over()
        }
    }
}
else if (ball_hit_block()) {
    erase_block()
}
}

timer_tick(tick) | each{x ->
    # グラフィックス出力
    display_board()
}

```

図7 ブロック崩し

このようにパイプラインを分割することで、一つひとつの処理をシンプルに保ち、理解しやすく、また保守しやすく記述可能になります。

この実装の鍵になるのは、処理を複数のパイプラインに分割することと、盤面の状態をインメモリーのデータベースに格納する点です。

イミュータビリティと状態

さて、Streamの大きな特徴はデータ構造がイミュータブル（更新不能）なことですが、プログラムを書くとするとこれはこれで大変です。

純粋関数型言語で基本的に副作用を持たないHaskellでは「モナド」という仕組みを使って状態変更のような副作用のある処理を管理しています。しかし正直、使いやすいとは思えません。

同じデータ構造がイミュータブルの言語でもErlangというActorベースの言語では、状態を2種類の方法で管理しています。一つは独立して動作するプロセスに状態を封じ込めることで、状態の変更や取り出しはそのプロセスへのメッセージのやり取りで行います。このやり方は巧妙ですが、Streamのような明示的なプロセス（ヤスレッド）を持たない言語では採用しづらいものです。

もう一つの方法は、ETSやMnesiaというようなErlangの言語処理系に組み込まれたデータベー

スを使うことです。つまり、通常のデータ構造はイミュータブルでも、データベースのような状態を保存するための専用のデータ構造だけは更新を許すというアプローチです。

これは通常のプログラミングに慣れている人にも大変分かりやすいです。実際、これまでの例題でもToDoやブロック崩しプログラムでは、データベースを使って変更可能な状態を実現していました。

組み込みDBのkvsを導入

このことを考えるとStreamには組み込みデータベースが必要です。Clojureという言語も原則的にデータ構造はイミュータブルですが、STM (Software Transactional Memory) という方式を用いて、更新可能な状態を実現しています。トランザクションはデータベースの一貫性を維持する方法ですが、Clojureはこれをデータ構造の一貫性維持のために用いています。

Stream処理系に組み込むデータベースの具体的な仕様についてはまだまだ検討の余地はありますが、とりあえずは簡単なものとして「kvs」という名前のデータベースを作ってみました。

kvsは簡単なキーバリューストアです。仕様は表3の通りです。

updateの挙動だけは説明が必要かもしれません。updateはキーと関数を引数に取ります。関数は古い値を引数として受け取り、その戻り値が新しい値になります。

それからtxn関数は、トランザクションを開始して、引数として受け取った関数を実行します。この関数はトランザクションを表すオブジェクトを受け取ります。トランザクションはdbと同じ振る舞いをし、それに対する更新はトランザクション終了時にデータベースに反映されます。トランザクション開始以後にデータベースが更新されていて、矛盾が発生する場合、それまでの変更は破棄され、トランザクション関数が最初から実行されます。トランザクション関数実行中にエラーが発生した場合には、トランザクション中でのそれまでの変更は破棄された後、例外が再発生します。

kvsの実装

さて、では実際にkvsを実装してみましょう。何度も繰り返していることですが、まず動作する実装が重要です。ここでは性能などのことは考えず、できるだけ早く仕様を満たす実装が求められます。もちろん、その実装は、実際に利用する中で性能上の不満や改善すべき点が見つかることでしょう。そのときには、きちんと測定し、何が問題かを明らかにしてから改善すべきです。デザイン・実装・測定・改善はソフトウェア開発の鉄則です。

kvsの実装に当たっても、実装においては最短を目指します。幸い、Streamの実装にはすでにメモリー上のキーバリューストア（というか単なるハッシュテーブル）であるkhashを採用しています。当面はこれを利用してkvsを実装しましょう。

API	働き
db = kvs ()	メモリーDBをオープン
db.put (key, val)	データ設定
db.get (key)	データ取得
db.update (key, f)	データ更新
db.txn (f)	トランザクション
db.close ()	DBを終了

表3 kvs仕様

khash

khashはC言語用のハッシュテーブルライブラリです。MITライセンスで提供されています。khashの最大の特徴はヘッダーファイルだけからなることです。khashを使うにはヘッダーファイル「khash.h」をインクルードして、利用するハッシュテーブルをマクロを利用して宣言します。アクセス用の関数などもマクロを通じて定義されます。

例えば、kvsで使うStream文字列から、任意のStreamデータへのハッシュテーブルは、図8のように定義します。マクロKHASH_INITがハッシュテーブルを定義するマクロです。六つも引数を取るマクロですが、その意味は表4の通りです。

```
#include "strm.h"
#include "khash.h"

KHASH_INIT(kvs, strm_string, strm_value, 1,
kh_int64_hash_func, kh_int64_hash_equal);
```

図8 kvsのハッシュテーブル定義

順序	名前	意味
1	name	ハッシュテーブル名
2	khkey_t	キーの型
3	khval_t	バリューの型
4	kh_is_map	マップ(1)、セット(0)
5	hash_func	キーのハッシュ関数
6	hash_equal	キーの比較関数

表4 KHASH_INITの引数

ハッシュテーブル名はハッシュテーブル構造体の名前に使います。ハッシュテーブルは「khash_t(名前)」という型でアクセスします。図8で定義したハッシュテーブルにアクセスするプログラムは図9のようになります。

```
int
main()
{
    int ret, is_missing;
    khiter_t k;
    strm_string key = strm_str_intern("foo", 3);
    strm_string key2 = strm_str_intern("bar", 3);
    khash_t(kvs) *h = kh_init(kvs);

    /* kh_putで保存位置を取得 */
    k = kh_put(kvs, h, key, &ret);
    /* kh_value()への代入で実際に保存 */
    kh_value(h, k) = strm_int_value(10);
    /* kh_getで参照位置を取得 */
    k = kh_get(kvs, h, key2);
    /* 値がなければkはkh_end(最終位置)に */
```

```

/* あればkh_value()で値を取り出す */
is_missing = (k == kh_end(h));
/* kh_begin()からkh_end()まで繰り返すことで */
/* 全要素を探することができる。空の位置もあるので */
/* 要素アクセス前にkh_exist()で要チェック */
for (k = kh_begin(h); k != kh_end(h); k++)
    if (kh_exist(h, k))
        kh_value(h, k) = strm_int_value(1);
/* ハッシュテーブルの破棄はkh_destroy()で */
kh_destroy(kvs, h);
return 0;
}

```

図9 ハッシュテーブルへのアクセス

khashはヘッダーファイルだけで使えて大変便利ですので、Cでプログラムを開発する機会があれば、お薦めのライブラリです。APIが独特なので若干慣れが必要ですが。

khashによるkvs実装

ここまでくれば（最もシンプルなものであれば）キーバリュースタックを作るのは簡単です。kvsの最初の実装のうち、作りが分かる部分を図10に抜粋します。

nsメンバーにstrm_state構造体で表現されるネームスペースが格納されていて、オブジェクト指向言語におけるクラス代わりになっている点に注意さえすれば、さほど難しい内容ではないでしょう。

```

struct strm_kvs {
    STRM_PTR_HEADER;
    khash_t(kvs) *kv;
};

static khash_t(kvs)*
get_kvs(int argc, strm_value* args)
{
    struct strm_kvs *k;

    if (argc == 0) return NULL;
    k = (struct strm_kvs*)strm_value_ptr(args[0], STRM_PTR_KVS);
    return k->kv;
}

static int
kvs_get(strm_task* task, int argc, strm_value* args, strm_value* ret) {

```

```

khash_t(kvs)* kv = get_kvs(argc, args);
strm_string key = strm_str_intern_str(strm_to_str(args[1]));
khtiter_t i;

i = kh_get(kvs, kv, key);
if (i == kh_end(kv)) {
    *ret = strm_nil_value();
}
else {
    *ret = kh_value(kv, i);
}
return STRM_OK;
}

static int
kvs_close(strm_task* task, int argc, strm_value* args, strm_value* ret) {
    khash_t(kvs)* kv = get_kvs(argc, args);
    kh_destroy(kvs, kv);
    return STRM_OK;
}

static strm_state* kvs_ns;

static int
kvs_new(strm_task* task, int argc, strm_value* args, strm_value* ret) {
    struct strm_kvs *k = malloc(sizeof(struct strm_kvs));

    if (!k) return STRM_NG;
    k->ns = kvs_ns;
    k->type = STRM_PTR_KVS;
    k->kv = kh_init(kvs);
    *ret = strm_ptr_value(k);
    return STRM_OK;
}

void
strm_kvs_init(strm_state* state) {
    kvs_ns = strm_ns_new(NULL);
    strm_var_def(kvs_ns, "get", strm_cfunc_value(kvs_get));
    strm_var_def(kvs_ns, "put", strm_cfunc_value(kvs_put));
    strm_var_def(kvs_ns, "update", strm_cfunc_value(kvs_update));
    strm_var_def(kvs_ns, "txn", strm_cfunc_value(kvs_txn));
    strm_var_def(kvs_ns, "close", strm_cfunc_value(kvs_close));
    strm_var_def(state, "kvs", strm_cfunc_value(kvs_new));
}

```

図10 kvsの初期実装（抜粋）

排他制御

しかし、このシンプルな実装は実用的ではありません。マルチスレッド、マルチタスクな言語である Stream では必須の排他制御が行われていないからです。

例えばハッシュテーブルを書き換えている最中に、ほかのスレッドが読み込みをしようとしたら、ありもしないデータを読み出してしまいかもしれません。書き込みの競合があればデータ構造が壊れてしまう危険性もあります。

このような事態を避けるため、なんらかの排他制御が必要になります。そのための手段はいくつかあります。今回使うのは一つがロック、もう一つがトランザクションです。

ロックによる排他制御

ロックによる排他制御は比較的簡単です。ハッシュテーブルごとに `pthread_mutex_t` というロックを用意しておき、共有されるデータ構造（今回はハッシュテーブル）へのアクセス前後に

```
pthread_mutex_lock()
pthread_mutex_unlock()
```

という関数で囲むだけです。

これで少なくともデータを破壊してしまうような事態は避けられるはずです。

トランザクションの実現

しかし、ハッシュテーブルのような一種のデータベースの場合、ロックによる保護でデータ破壊を避けるだけでは十分でない場合があります。

複数のスレッドが同時に書き込んだ場合、データベースそのものが壊れなくても、データの一貫性が維持できないことがあるからです。例えば、銀行の口座 A から口座 B へ振り込みをするときに、途中で別の振り込みが行われたために振り込んだはずのお金が消えてしまったりしたら大問題です。

このようなことを避けるため、データベースではトランザクションという手法を採用しています。トランザクションは、成功して一連の状態を変更するか、衝突してやり直すか、失敗して状態をトランザクション開始前の状態を保存するかのいずれかの結果を保証します。

kvs では、`txn` 関数を用いてトランザクションを実現します。`txn` 関数はトランザクションを開始し、引数として渡された関数にトランザクションオブジェクトを引数にして渡します。トランザクションオブジェクトは通常の kvs データベースと同じように動作し、トランザクション終了時に（操作が成功すれば）データベースに書き込みます。

トランザクションの実装はかなり複雑なので紙面の関係で解説できそうにありません。github.

com/matz/streemにあるソースコードsrc/kvs.cを参照してください。

まとめ

今回はStreemのパイプラインプログラミングについて学びました。また、状態変化の実現に重要なkvsデータベースを実装しました。こうやって、少しずつStreemを実用に近づけていくのも連載に連動した言語開発の醍醐味です。

タイムマシンコラム

当たり前にあるべき機能はやはり必要

2016年3月号掲載分です。今回はStreemの特徴であるパイプラインプログラミングスタイルで、具体的にどうやってプログラムを記述するかについて考察してみました。

Streemが大きく影響を受けている関数型プログラミングでもそうですが、それを使ってエレガントに記述できる領域は確かにあります。解説書などではその領域の問題を取り上げて、「こんなにエレガントに解決できるんですよ」と紹介してくれます。

しかし、私たちがプログラマとして日常的に取り組んでいる問題の多くも同じようにエレガントに解決できるとは限りません。いや、もちろん解決できるんでしょうけど、エレガントかどうかは別問題です。慣れていない身には多くの問題は直接的に解決できないと感ぜられるでしょう。とはいえ、対象領域ごとに言語を切り換えるのも（主に精神的）コストの高い話です。

Streemは、どちらかというと汎用言語ではなくパイプラインプログラミング専用言語なので、その心配はやや低いです。しかしそれでも、これ一つでできる範囲は広い方がよいに決まっています。

そこでStreemで多様なプログラムを開発することを想定して、いろいろ考察してみました。その考察結果がインメモリーデータベースであるkvsです。言語仕様に書き換え可能な変数があれば簡単に実現できることのためにデータベースを導入するのは大げさな気もしました。しかし、Clojureの試みなどを見ていると、これはこれでアリなのではないかと思います。今回は性能を考えないプロトタイプですが、コンセプトの有効性は示すことができたのではないかと思います。

今回のサンプルプログラムに登場する関数群ですが、kvsを除くと、http_request()とかkey_event()とかtimer_tick()とか、まだ実際には提供されていない関数ばかりです。Streemで実用的なプログラムを記述するためには、このような便利関数をまだまだたくさん用意する必要がありますということに気がきました。この後数回の連載ではそのような関数の検討と実装が続くことになります。

5-2

パイプライン構成要素

5-1では、パイプラインプログラミングの実践について解説し、その重要な構成要素としてkvsを開発しました。今回もパイプラインプログラミングの実践に必要な構成要素について検討を加えつつ、必要なツールをそろえていこうと思います。概念を整理して実装を大幅に改造します。

これまでStreemを開発してきて、「とりあえず」のつもりで付けてきた名前が実態に合っていないことが気になってきました。

例えば、strm_taskという重要な構造体がありますが、Streemの言語レベルには「タスク」という概念は登場しません。この「タスク」が何を表現するのか、自分でもいまいちピンときていませんでした。実装上と言語上の概念に解離が起きているような気がして、どうにも違和感があります。

ここ数カ月、ずっとこの違和感について考えてきました。ここでパイプラインプログラミングを突き詰めるために、一度概念を整理する必要があるような気がしてきました。

パイプラインの構成要素

そこでStreemのパイプラインに登場する構成要素を再構成してみます。重要になるのは以下の四つの概念です。

- パイプライン
- ストリーム
- タスク
- ワーカー

この中で最も重要な概念はストリームです。ストリームはデータの流れを構成する「処理」です。ストリームには、データを生成する「プロデューサー」、入力として受け取ったデータを加工する「フィルター」、データを受け取り消費する「コンシューマー」の3種類があります。プログラミングモデルとしては、ストリームを結合したのもストリームになるのですが、処理系レベルでは、strm_streamという構造体で一つの処理を表現します。

プロデューサーからコンシューマーまでひとつながりになったストリームのことをパイプラインと呼び

ます。もっともStreemの処理系でパイプラインを扱う部分は(まだ)存在しないので、あくまでも概念上だけの存在です。

ストリームを流れるデータ一つ分の処理を担うのがタスクです。Streem 処理系はタスクによる処理はそれぞれ一つのC関数で表現されています。ストリームがデータを次段にemitするたびにタスク構造体を実行キューに追加して、実行を予約することになっています。

ワーカーはタスクを実行するスレッドのことです。Streem 処理系では搭載されているCPUの数だけワーカーを生成します。ワーカーはキューの先頭から一つずつタスクを取り出しては実行します。

global renaming

さて、概念を整理できたので、それに対応してソースコードの方も変更することにしましょう。Streemの実装の構造体名、関数名、変数名などに対して表1のような変更を加えます。

種別	元名称	変更した名称
型	strm_task	strm_stream
変数	task	strm
構造体	strm_thread	strm_worker

表1 名称変更

実はcore.cやqueue.cで実装されているイベント処理内では、strm_taskとは似た名前ではありますが、実際には無関係のstrm_queue_taskという名前の構造体が使われており、こちらが今回の概念整理後のタスクに該当するものです。こちらはこちらで大幅に書き換えたのですが、その点については後で解説することになります。

一般にソースコードに登場する名称を大規模に変更すると、互換性を失います。このため特にオープンソースソフトウェアの場合、なかなか困難が伴います。

Rubyでも開発の初期に大規模に名前を変更しました。これは、ほかのライブラリなどと名称が重複しないために「rb_」というプリフィックスを付けるためのものでした。もう20年近く前のこととなりますが、まだユーザー数が少なかったにもかかわらず、動かなくなるライブラリが続出したりでなかなか大変な思いをしました。

Streemの場合、まだ実用性に乏しくほとんど誰も使っていないことと、そもそも拡張用APIが提供されていないことがあって、まだまだ気軽に変更できます。恐らくは今後もより良い実装のために大胆な変更が行われると思います。

デバッグ用のstrm_p()を追加

ついでにもう一つ機能を拡張しておきましょう。

先日のNaN Boxingの導入以来、Streemのオブジェクトはすべて64ビット整数で表現されるようになりました。NaN Boxingは実行効率は良いのだと思いますが、中身が見えないのでデバッグするときに面倒です。そこでデバッガーからオブジェクトの中身を表示する関数strm_p()を新設しました。

strm_p()の実装は簡単です(図1)。引数として与えられたオブジェクトをstrm_to_str()関数で文字列オブジェクトに変換し、strm_str_cstr()関数でCの文字列ポインタを取り出し、fputs()で標準出力に表示するだけです。

```

strm_value
strm_p(strm_value val)
{
    char buf[7];
    strm_string str = strm_to_str(val);
    const char* p = strm_str_cstr(str, buf);
    fputs(p, stdout);
    fputs("\n", stdout);
    return val;
}

```

図1 オブジェクトの中身を表示する関数strm_p()

strm_str_cstr() 関数の buf 引数については少々説明が必要かもしれません。Stream の NaN Boxing では 6 文字以下の短い文字列を strm_value の中に直接埋め込んでいるので、文字列ポインタを取り出すことができません。そこでそのような文字列内容をコピーするために buf 領域が必要です。埋め込まれる文字列は最大 6 文字 (バイト) なので、末尾の NUL 文字分を足して、buf のサイズは最低 7 バイト必要です。

さらにオブジェクトごとに文字列表現をカスタマイズできる仕組みも用意しました。namespace が設定されたオブジェクトが to_str() 関数を持っていた場合、その戻り値を文字列表現として使うようにしました。

パイプラインの構成部品

さて、5-1 でしたパイプラインプログラミングのパターンは以下の 3 種類でした。

- プロデューサーからのデータをフィルターで加工して、コンシューマーで出力
- プロデューサーからのデータをフィルターで加工して、データベース (kvs) に書き込む
- 一定時間ごとに処理を起動して、データベースから読み込んだデータを出力

名前	働き
stdin	標準入力
fread()	ファイル読み込み
tcp_server()	ソケット接続
tcp_socket()	ソケット読み込み
seq()	一定範囲の整数
rand()	乱数列
tick()	一定時間ごとにイベント

表2 Streamのプロデューサー

名前	働き
stdout	標準出力
stderr	標準エラー出力
fwrite()	ファイル書き出し
tcp_socket()	ソケット書き出し
each()	繰り返し

表3 Streamのコンシューマー

関数名	働き
map()	関数適用結果で置き換え
filter()	条件に合うものを選別
count()	要素数を与える
sum()	要素を合計する
csv()	csv文字列を配列に変換
flatMap()	配列展開するmap (新設)
split()	文字列の分割 (新設)
reduce()	畳み込み (新設)
reduce_by_key()	キーごとの畳み込み (新設)

表4 Streamのフィルター

これまでに登場したプロデューサーは表2、コンシューマーは表3の通りです。

あと、登場済みと今回新設したフィルターを表4に示します。もちろんこれだけのフィルターでできることは限られていますので、フィルターは今後さらに追加していきます。

新設したフィルターに、`reduce()` があります。`reduce()` はストリームの要素を畳み込む関数で、

```
reduce(b){x,y->...}
```

というような形で呼び出します。上流のストリームから要素 (`e1`, `e2`, ...) が与えられると、まずは関数が `b` を第1引数、`e1` を第2引数として呼び出されます。その結果を `r1` とすると、次の要素 (`e2`) に対しては、`r1` が第1引数、`e2` が第2引数として関数が呼び出されます。以後同様に繰り返されて、最後の結果が出力として、下流のストリームに渡されます。

`reduce()` を使うと階乗の計算は、図2のようになります。ほかの言語でよく見る再帰やループを使ったものに比べると、だいぶ外見が異なります。

また、既出の `sum()` 関数は `reduce()` を使って、図3のように定義できます。このように定義された `sum()` は上流から受け取ったデータの合計を下流に渡すフィルターストリームを返します。

畳み込みで単語カウント

`reduce` の亜種として、キーについて集計する `reduce_by_key()` も必要になるでしょう。これはパイプラインプログラミングにおける「Hello World」といっても過言ではない単語カウントに用いられます。Stream による単語カウントを図4に示します。

図4では `stdin` から入力された各行を

```
flatMap{s->s.split(" ")}
```

によって単語ごとに分割しています。文字列に `split` 関数を適用すると、引数として与えられた区切り文字で分割した配列が返ってきます。`flatMap` は関数から与えられた配列をストリーム要素に展開します。その結果、上流から

```
this is my pen
my name is yukihiro
```

```
seq(6)|reduce{x,y->x*y}|stdout
```

図2 `reduce`による階乗の計算

```
def sum() {
  reduce(0) {x,y -> x+y}
}
```

図3 `reduce()`を使った`sum()`の定義

```
stdin
| flatmap{s->s.split(" ")}
| map{x->[x, 1]}
| reduce_by_key{k,x,y->x+y}
| stdout
```

図4 Streamによる単語カウント

という2行のデータが流れてきたら、

```
this  
is  
my  
pen  
my  
name  
is  
yukihiro
```

という8要素のデータを下流に流すことになります。なお、図4ではflat_mapの説明をしたかったため、使っていませんが簡便性のために、

```
split(" ")
```

だけで

```
flatmap{s->s.split(" ")}
```

とほぼ同じ働きをするsplit()関数も用意しています。

次の段のmapは単語の列を

```
[単語, 1]
```

という配列の列に変換します。

最後にreduce_by_key()は2要素の配列のストリームを取り、第1要素ごとに第2要素をreduceします。その結果、元の2行の入力は

```
[this,1]  
[is,2]  
[my,2]  
[pen,1]  
[name,1]  
[yukihiro,1]
```

に変換されることになります。これをstdoutに流すと単語カウントになるわけです。

ストリームの分岐・合流

これまで紹介してきた例のほとんどでは、ストリームで構成されるパイプラインはプロデューサーからフィルターを経由してコンシューマーに至る一本道でした。

しかし、ストリームの構成パターンはそのような一本道ばかりではありません（図5）。

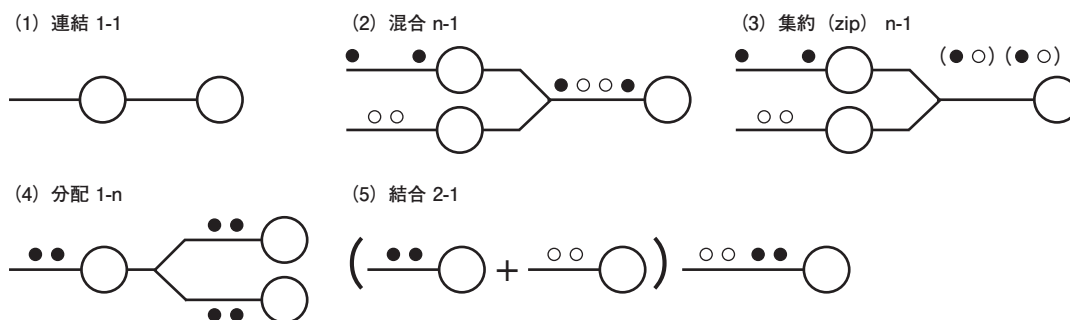


図5 ストリームの構成パターン

図5 (1) の連結は、「|」演算子を使って二つのストリームの入力と出力をつなげることです。これまで登場してきた基本的なストリーム構成ですね。

混合は、複数のストリームの内容を一つにまとめることです。データは来た順に「混ぜて」下流のストリームに流されます。具体的にはaとbという二つのストリームがあったとして、これらの内容をstdoutに流す場合には、

```
a | stdout
b | stdout
```

とすることで、図5 (2) の状態に結合したストリームが構築されます。

図5 (3) の集約は、複数のストリームの要素をまとめる点では混合と同じですが、動作が異なります。それぞれのストリームからの要素をそのまま流すのではなく、それらを組み合わせて配列にして流します。

集約はジッパー（ファスナー）のかみ合わせのように見えることからzipという関数を用います。

seq()で1から始まるストリーム、fread(path)でファイルの内容のストリームを得た場合、この二つのストリームを集約したストリームを得るためにはzip関数を使って

```
zip(seq(), fread(path))
```

とします。すると、各行に

[数値, 行]

というデータが与えられて行番号と行の組み合わせを得られます。ちょうど「cat -n」コマンド相当ですね。

seq() 関数が整数を順番に生成するのと、ファイルの読み込みとでは実行されるペースに違いがあるはずです。それでも各ストリームの要素が順番に組み合わせられます。

図5(4)の分配は、上流から来たデータを複数のストリームに届けることです。aというストリームから来たデータをbとcというストリームに届ける場合には

```
a | b
a | c
```

とします。この場合、bとcには同じデータが届けられます。

最後の図5(5)の結合は、二つのストリームを順番につなぐことです。aとbの二つのストリームを結合すると、aのストリームの全データが下流に流れた後、bというストリームからのデータを流します。ストリームを結合する場合には「+」演算子を使います。例えば

```
(a + b) | stdout
```

とすると、aの内容に続いてbの内容が出力されます。UNIXのcatコマンドが、複数並べたファイルの内容を続けて出力してくれるのと同じようなものだと考えることもできるでしょう。

フロー制御が必要

さて、ここで難しいことが一つあります。例えばzipの場合、入力となるストリームが複数あって、それぞれデータを生成するペースが異なります。seq()やrand()のような軽い計算だけのプロデューサーが上流であれば次々とデータを送ってくるでしょう。一方ソケットのように、いつデータが送られてくるのかは接続先次第というプロデューサーもあります。

しかし、zip自身は上流からのすべてのデータがそろわないと処理を始められません。このためペースの違う上流を組み合わせると、ペースが速い上流からのデータが積み上がってしまう(≡メモリーを消費してしまう)ことになります。このような事態を避けるためには、データの流量制御が必要になります。

プロデューサーを非優先

流量制御の実現方法はいくつも考えられると思いますが、今回もこれまでと同じくプロデューサー

ストリームの優先順位を下げることで実現しようと思います。

パイプラインを流れるデータが多過ぎるので下流の処理が追いつかず、流量制御が必要になるのです。だから、下流処理の優先順位をプロデューサーよりも上げることで、流量制御をしようという試みです。ただ、非同期なプログラムの挙動を予想するのは困難です。zipのような合流を伴うストリームに対して、この優先順位方式が本当にうまく動作するのか若干の不安があります。そこでまずは実際に動作させてみて、結果を測定してから問題があるかどうか確認してみようと思います。

5-1までのStreamの実装では実行キューに優先度があって、プロデューサーの優先順位を下げていました。今回はプロデューサー専用のキューを用意します。キューからの取り出し順序を通常キュー→プロデューサーキューにする方法で、プロデューサーの優先度を下げます。

分配時の滞留は考えない

実は同じ問題は下流側の事情でも発生します。分配によって複数の下流にデータを届けるとき、それぞれの下流ストリームのデータ処理速度が違っていると、遅いストリームの前にデータが積み上がってしまいます。しかし、分配の上流ストリームがフィルターであってプロデューサーではない場合には、上で説明したような優先順位による流量制御は効果がありません。

いろいろ検討した結果、下流側の問題はさほど深刻にならないだろうと考えて、今回は対応を見送ることにしました。

もっとも、20年間のRubyの開発で学んだ知恵の一つに、言語処理系の場合、利用される局面を限定することは困難という点があります。発生する可能性がある問題は必ず発生し、しかも想定したよりも深刻な事態になるものです。Streamでもいつかこの問題に対処しなければならない日が来ることでしょう。そのときまでに賢い解決策を考えておく必要がありますね。

従来のキューはメモリーを配慮

流量制御を実装するために、Streamのイベント処理の根幹部分であるイベントループの部分を大々的に置き換えることにします。

これまでの構成を図6に示します。この構成の特徴は、ワーカースレッドごとにタスクキューを持つことで、同じパイプラインの処理が同じワーカースレッドで実行されることを狙っていることです。これにはいくつかメリットがあって、まずは、ストリームに属するタスクが複

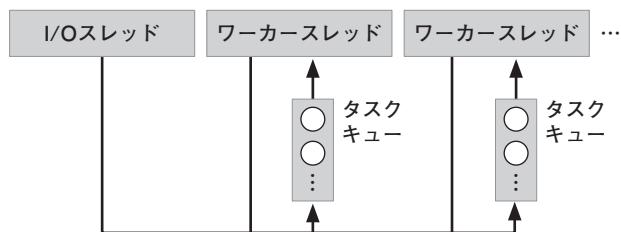


図6 これまでのイベント処理アーキテクチャ

数のスレッドで実行される心配がないため、排他制御のことを気にしなくて済みます。さらに、タスクが順番に処理されるため、処理速度のムラによってデータの順番が前後することもないのもメリットです。最後に一連のタスクが同じスレッドで実行されることで、キャッシュが共有化され、メモリー

アクセスの効率が低いことが期待できます。

しかし一方で、パイプラインの数が少ないときには、メニーコア環境であっても結局複数コアを活用できず、性能が伸びない危険性もあります。

つまり、キャッシュの活用による性能上のメリットと、複数コアを活用しきれない性能上のデメリットの両方が存在するわけです。これを総合的に考えるとデメリットの方が上回りそうです。また順序の保証についても、ストリーム処理において順序は問題にならない場合が多く、それほどのメリットにはなりません。

コア活用を優先するよう変更

そこで図7のような新しい構成を考えました。キューを共有することで手が空いたワーカースレッドが仕事を引き受けることになり、コアの活用率が上がります。同じパイプラインに属するタスクが別のスレッドで実行されると1次キャッシュを共有できない可能性があります。

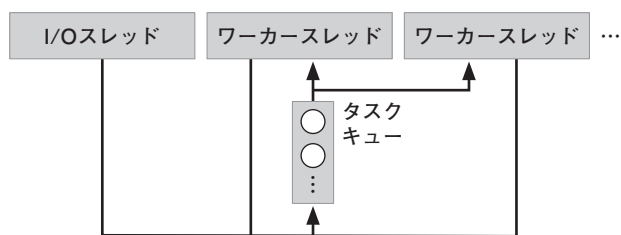


図7 新しいイベント処理アーキテクチャ

しかし2次キャッシュや3次キャッシュに乗っている可能性は十分に高いので、ここでは気にしないことにします。なお図7では、流量制御のためのプロデューサーキューについては省略しています。

ただし、図7の構成そのままでは若干問題があります。まず、同一ストリームに属するタスク間で排他制御が必要になるケースはかなり頻繁に存在します。これまでに紹介したフィルターの中でも、ストリームに状態が存在するcount()、sum()、reduce()などは排他制御が必要になります。

このようなケースのために、各ワーカースレッドにも個別のキューを用意しておきます。キューから取り出してきたタスクの属するストリームが、排他制御の必要なものなら、同時実行されて一貫性を失わないために、現在実行中のタスクが終了してから、続くタスクが実行できるようにします。つまりタスクを実行中のワーカーの個別キューに、続くタスクを追加します。

タスクキュー実装の改善

イベント処理の効率を高めるために、さらにキューの実装も置き換えます。これまでの実装はロックと条件変数を組み合わせたものです。これは正しく動作することは保証されるものの、性能についてはかなり問題があります。

マルチスレッド環境でのデータ構造は、うまくやらないと正しい処理ができません。例えば、あるスレッドがデータを書き換えている最中に別のスレッドがデータを読み込んだ場合、実際に読み出せるのは、

- 運良く正しいデータ

- 書き換え途中の一貫性のないデータ
- 書き換えたデータの無関係のゴミ

のいずれかです。しかも、たちが悪いことに、タイミングによってどれが発生するのか予想できません。大体うまく動くけど、時々失敗するというのは、バグとしては最悪の部類です。

このような事態を避けるため通常、マルチスレッド環境で共有されるデータ構造は排他制御をする必要があります。具体的には、スレッド間で共有されるデータをアクセスする場合には、その直前でロックをかけ、アクセスが終わればロックを解除します。ほかのスレッドがデータをアクセスするときにもロックをかけようとしませんが、別のスレッドがアクセス中でロックがかかっている場合には、ロックが解除されるまで実行が停止します。

実行が停止するということは、処理が滞るということです。Stream のようにコアの数だけワーカースレッドを立ち上げるアーキテクチャーでは、せっかくのマルチコアが十分に活用できないということでもあります。これは実にもったいない。

そのようなことを避けるために、ロックによる保護を用いないロックフリーなデータ構造が考案されています。ロックフリーとは複数のスレッドがどのタイミングでアクセスしても破綻しないようなアルゴリズムを採用したデータ構造です。

自分で考えても実装できそうにないので、既存のものを利用することにします。GitHub で検索した結果、

```
github:supermartian/lockfree-queue
```

を見つけました。このロックフリーキュー（を大改造して）使用することにします。

Compare and Swap

スレッドを考慮しない「普通」のプログラミングでは、データを更新するときに、まず読み込んで、それを加工して、書き戻すという手順を踏みます。しかし、読み込んでから加工するまでの間にほかのスレッドがそのデータにアクセスしたり、書き換えたりしたら不整合が起きる可能性があります。そんなことが起きないようにデータの参照と更新を不可分に行うところがロックフリーアルゴリズムの基本になります。

このような不可分の更新操作を「Compare and Swap」（略して CAS）と呼びます。CAS は、メモリー位置、値 1、値 2 の三つの引数を取ります。そのメモリー位置の内容と値 1 を比較し、等しければそのメモリー位置に値 2 を格納する操作です。CAS は CPU の 1 命令で実現され、その操作中にほかのスレッドが実行されることはないことが保証されます。このような不可分の操作を「アトミック」と呼びます。

元々 C 言語には、CAS を実現する命令は存在しませんから、その命令を呼び出すアセンブラ記

述が必要でした。しかし、幸い GCC 4.1 からは CAS を実現する拡張機能

```
__sync_bool_compare_and_swap()
```

が提供されるようになりました。

まとめ

今回はパイプライン処理の概念整理と実装の改善をしました。マルチスレッド環境でのプログラミングは考慮すべき点が多く、なかなか悩ましいです。次回（本書では4-5、下のタイムマシンコラム参照）は、この実装改善についてもう少し解説しようと思います。

タイムマシンコラム

タイミングに依存するバグは本当に大変

2016年4月号掲載分です。パイプラインの構成要素として、reduce、reduce_by_key、zipなどの関数を新設しています。また、ロックフリーキューについてもちょっとだけ触れています。

雑誌では、4-5の前号に今回を掲載していたので、本書ではちょっとだけ違和感を持つ人もいらっしゃるかもしれません。しかし、4-5のタイムマシンコラムでも書いたように、このロックフリーキューは高負荷時に発生するバグが解決できず、最終的には不採用（コードは残っているが、コンパイル時フラグでロックを利用するキューを使うようにしている）になっています。

タイミングに依存する非決定的バグは本当にデバッグが大変です。だいたい一生懸命挑戦したのですが、時間と労力の限界が来て諦めてしまいました。いつか、再挑戦したいものです。

5-3

CSV 処理機能

▶ **Stream**のようなパイプライン処理をする場合に、入力元になるデータ形式として最も多く使われるのは、恐らく**CSV**でしょう。そこで**CSV**によるデータ入力機能について開発の順序を追いつながら解説しましょう。「標準」がない**CSV**の処理は結構やっかいです。

CSV (Comma-Separated Values) は表形式のデータを表現するために広く使われている形式です。特にExcelなどの表計算ソフトからのデータをほかのソフトウェアに持ち出すためには、**CSV**のようなシンプルなテキストフォーマットが最も確実で信頼できます。

しかし、一方で**CSV**には各ソフトがそれぞれに対応しているだけで定まった標準がありません。一応、IETF (Internet Engineering Task Force) がRFC4180として文書化していますが、あくまでも「Informational (情報提供)」という位置付けで、厳密な規格というわけではありません。さらにRFC4180に従っていない**CSV**データが大量にある現状では、準拠していないからといって無視できないのが現実です。

今回はこの**CSV**によるデータ入力機能を**Stream**に追加します。

RFC4180

RFC4180で定義されている**CSV**データフォーマットはだいたい以下のようなルールに従います。

まず、ファイルは一つ以上のレコードからなります。レコードはCRLF (Carriage Return/Line Feed) で区切られる「行」です。CRLFをC言語的に表記すれば「`\r\n`」になります。

レコードは一つ以上のフィールドからなります。フィールドはコンマ (,) で区切られます。最後のフィールドの後ろにはコンマを付けません。RFC4180では各レコードは同じ数のフィールドを含むと定義しています。

フィールドはダブルクォート (") で囲めます。ただし、フィールドがコンマ、ダブルクォート、改行を含む場合には必ずダブルクォートで囲む必要があります。ダブルクォートで囲まれたフィールド中ではダブルクォートを二つ続ける (") ことで、ダブルクォート自身を表現します。

CSVはオプションとしてヘッダーを持てます。ヘッダーがあるかどうかは外部から指定し、存在する場合には先頭のレコードを構成する各フィールドの文字列が相当するフィールドの名前になります。

CSVバリエーション

既に述べたように、あくまでInformationalなRFC4180は「CSVの規格」ではなくて、「最低限このような定義をCSVとします」というような緩い合意のようなものです。

また、RFC4180が策定されるより前からCSVを解釈するソフトウェアは数多く開発されていますから、CSVの解釈にも数多くのバリエーションがあります。例えば、以下のようなものです。

- レコード区切りは、CRLFだけでなくLFも許容するものもある。
- フィールド区切りは、コンマだけでなくタブやスペースによる区切りを許容するものもある。
- ダブルクォートのエスケープは2重に重ねることになっているが、バックスラッシュ(\)を前置するスタイルのエスケープを許すものもある。
- フィールド数はRFC4180ではすべてのレコードで同じ数となっているが、異なっていた場合の対応は処理系ごとに異なる。エラーにするものもあれば、多ければ無視し、足りなければ末尾に空文字のフィールドを足す処理系もある。
- 空行がCSVファイルに入っていた場合、その行を無視する処理系もあれば、フィールド数ゼロのレコードとする処理系もある。
- CSVデータ中にコメントを許す処理系もある。
- RFC4180にフィールドのデータ型に関する記載はない(すべて文字列として扱う)が、処理系によってはすべて数字で構成されるフィールドを数値に自動変換する。

CSVフォーマットのデータが大量に存在するので、これらのバリエーションを無視するわけにもいかず、多くのCSV関数は大量のオプションを受け付けるものが多いです。

ただ、今回はあまりオプションにこだわることはせず、「ほとんどの場合にうまくいく」程度のものを目指します。今後、必要に応じて機能強化していこうと考えています。

GitHub探求

自分でゼロからCSVを解釈する関数を書いてもよいのですが、せっかくオープンソースソフトウェアを開発しているのですから、他人の手を借りることにしましょう。

そこでインターネットで公開されているCSV解釈ルーチンを検索し、条件に合うものを探すことにします。見つからなければ諦めて自分で開発すればよいことです。

CSVルーチンの検索のための条件は以下のようなものです。

- Streamに組み込みやすいようコードがCで記述されている。
- 後で手を入れやすいよう見通しの良い実装。
- グローバル変数を使わないなどスレッドセーフ性を持つ。

● Stream と組み合わせて使うのに適したライセンス。できれば MIT。

うまくこんな条件を満たすライブラリが存在するのでしょうか。まずは、GitHub で検索してみましょう。最近はソースコードが GitHub に集約されていて楽になりましたね。

GitHub の検索窓で、以下のように入力します。

```
csv language:C
```

「language:C」の部分は実装言語の指定です。この部分はユーザーが指定したものではなく、GitHub がソースの情報から推測したものですから、たまに間違っていることがあります。しかし間違いがあるのはだいたい複数の言語で実装されているプロジェクトで、今回探しているような単一機能のものではありません。

この条件で検索すると174個のリポジトリがヒットします(2015年7月上旬時点)。かなりの数ですね。中にはライセンスが明示されていないものも含まれていますから、とりあえずそれらは除外します。ほかに条件の合うものがなければ、作者にメッセージを送って交渉することになるでしょうが、最初からライセンスが明示されているものの方がいろいろと楽です。

それらを上記の条件に合うかどうかを基準に絞り込んでいきます。一番多いのが GPL で提供されている libcsv のラッパーです。今回は libcsv ほどの機能が必要というわけではないですし、これらは除外します。次にグローバル変数が多用されていて、スレッド実行が基本である Stream と相性が悪いものも諦めます。

そうやって残ったものの中で、最も都合が良かったものが「semitrivial/csv_parser」というリポジトリでした。グローバル変数を使っておらず、非常にシンプルで無駄なことをしていません。また見通しが良いので改造も簡単そうです。さらにライセンスが Stream 本体と同じ MIT なのもありがたいです。これをベースに開発を進めることにします。

ライセンス

GitHub にはライセンスが明示されていないリポジトリもたくさんあります。今回は MIT ライセンスのソースコードが見つかりましたが、ライセンスが明示されていないときはどうすればよいのでしょうか。

私がお勧めしたいのは、直接作者に問い合わせを試みることです。GitHub のリポジトリの issue で質問してもよいですし、ほとんどの場合、作者のメールアドレスも登録されています。

「自分のソフトウェアにあなたのコードを使いたいが、ライセンスが明示されていないので不安がある。ライセンスを決めてもらえないだろうか。私としては MIT ライセンスがありがたい。その場合、あなたへのクレジット(貢献の表示)はこんな感じでどうだろうか」などという内容を丁寧にメールしましょう。あまりにも自分勝手になければ前向きの反応をもらえる可能性は高いと思います。

GitHub にソースを公開するような人で、自分のコードに興味を持った人からの連絡を嫌がる人

はあまりいません。もっとも既にユーザーがいるソフトウェアのライセンスを自分の都合で変更しろなどという要求はさすがに通らないと思います。その辺は相手の気持ちになって考える必要があります。そういう意味では下手にライセンスが明示されているものよりも、明示されていないものにライセンスを新たに付けてもらう方が（反応をもらえないというリスクはあるにしても）、見込みがあるのかもしれない。

実は今回採用した「semitrivial/csv_parser」ですが、この原稿の執筆中にリポジトリが削除されてしまいました。既にコードは手元にコピーしてあるので作業に問題はないのですが、オリジナルに敬意を払わないのは後々トラブルを起こす可能性がゼロではありません。

そこで作者にメールしてみたところ、すぐに返事が返ってきました。「そんなに重要なコードだと思わなかったので深く考えずに削除したが、使うというなら復旧する。興味を持ってくれてありがとう」ということでした。直後に復旧したリポジトリに感謝の気持ちを込めて、フォークした上で、スターも付けておきました。

このような交流もオープンソース開発の醍醐味の一つではあります。

csv_parser

オリジナルの csv_parser は図1に示すたった二つの関数からなっています。parse_csv 関数で文字列の CSV レコードを解析して、フィールドに分割した文字列の配列を返します。この配列を使い終わったら、free_csv_line 関数で解放します。

```
char **parse_csv(const char *line);  
void free_csv_line(char **parsed);
```

図1 csv_parser が提供する関数

実装は空行も含めてわずか145行です。コードの見通しも良く改造しやすいといえるでしょう。

ただし Stream に組み込むにはいくつかの点を変更する必要があります。最初の点は文字列の表現です。csv_parser は末尾に NUL (\0) を置く C の文字列を受け取り、C 文字列の配列を返します。しかし Stream は独自のオブジェクトを持つので、Stream 文字列 (strm_string) を受け取り、Stream 文字列を含む Stream 配列 (strm_array) を返すべきです。

まずはここから変更しましょう。NUL 文字が途中に含まれていても動作する必要があります。

まず、NUL 終端文字列である C 文字列 (char*) でデータを受け取っている部分をすべて Stream 文字列 (strm_string*) に置き換えます。また、「文字列が NUL でない間」という条件のループを、「文字列の長さだけ」という条件に置き換えます。

データを返す部分でも、malloc でメモリー空間を割り当てている部分を Stream のオブジェクト割り当て (配列は strm_ary_new と文字列は strm_str_value) に置き換えます。

ついでにオリジナルは末尾の改行に対応していなかったなので、末尾に改行 (CR または LF) が付いていた場合、それを取り除くようにします。

メモリー管理については Stream の持つガーベージコレクション機能に任せます。図1の free_

csv_line 関数は削除します。

Stream 配列

これで Stream に（とりあえずは）CSV 機能が追加され、図 2 のようなコードが動くようになりました。

しかし現時点では配列の内容を出力する機能がないので、CSV を読み込んで stdout に出力しても、配列に変換されたレコードが

```
fread("sample.csv")|csv()|stdout
```

図2 CSVを読み込むStreamサンプル

```
[...]
```

としか表示されません。これでは配列が与えられたことは分かって、中身の情報が全くないので、正常に動作しているかどうかさえ確認できません。まあ、今まで手抜きをしてきたのが悪いのですが。

これではまずいのでこの機会に配列を表示する機能を実装します。

配列の表示は実は結構面倒なのです。配列が再帰的なデータ構造で、配列自身を含むほかのデータを要素として持つこともそうですし、文字列の場合、配列の要素であるかどうかで表現方法が変化します。

ということかということ、"abc\n" という文字列があったとして、これを出力すると、a, b, c, そして改行を順に出力することになります。これが配列要素になった場合、出力は

```
["abc\n"]
```

のようにすべきです。つまり配列要素としての文字列はクォートで囲んだり、特殊文字をエスケープ表記に変換したりする必要があります。

コードについては value.c にある、strm_inspect 関数を参照していただきたいのですが、処理はだいたい以下のような手順になっています。

- 従来の文字列化関数 (strm_to_str 関数) とは別に、人間に読みやすく加工する文字列化関数 (strm_inspect 関数) を用意する。
- 配列の文字列化は、strm_inspect 関数を使って行う。strm_inspect 関数は各データ型について、次のような文字列化処理を実行する。
- 整数、浮動小数点数については、そのまま文字列化する。文字列についてはクォートで囲み、特殊文字 (\t, \r, \n, \e) はエスケープ表記、コントロール文字は数値表記する。
- 配列については、 "[" の後ろに各要素を strm_inspect で文字列化したものを "," で区切って並び、末尾に "]" を置く。

その結果、人間が読める形式で配列を出力できるようになり、例えば

```
まつもと,男性,50\r\n
```

というような行を読み込ませると

```
["まつもと", "男性", "50"]
```

というような出力が得られるようになります。

CSV仕様

これでとりあえずCSVデータを読み込めるようになりましたが、まだ完璧とはいえません。

上述のようなCSVの仕様の曖昧さに対して、StroomのCSV解析機能がどのように振る舞うべきかを検討し、きちんと決定する必要があります。

もう一度CSVの仕様の曖昧さについてまとめてみましょう。CSVの仕様には以下の点について曖昧性があります。

- レコード区切り
- フィールド区切り
- フィールド数
- クォートエスケープ
- コメント
- フィールド型
- ヘッダー

これらについては、オプションを許すのではなく表1のようにすることにします。オプションによって、さまざまな状況に対応する（ことによってコードが複雑化する）ことよりも、シンプルさと明快さを優先することを選んでわけです。

機能	曖昧さ	対応
レコード区切り	LF、CRLF	LF（末尾のCRは削除）
フィールド区切り	コンマかタブか	コンマのみ
フィールド数	エラーか数合わせか	エラー
クォートエスケープ	2重クォート、\前置	2重クォート
コメント	＃、//など	なし
フィールド型	全部文字列か数値、日付	数値のみ自動変換
ヘッダー	外部から指定	先頭行が全部文字列ならヘッダー

表1 StroomのCSV機能対応

将来的にはこれらの一部はオプションとして対応する必要が出てくるのかもしれませんが、当面はこれでいくつもりです。

CSVのタスク化

さて、ここからオリジナルの csv_parser からの大改造が始まります。

まずは下準備として、csv() 関数が専用のタスクを返すようにします。イメージとしては、4-1 のソケットの解説におけるサーバーソケットと同じような実装になります。

まず、タスク間で共有されるデータを保持するための構造体を作ります。これを csv_data 構造体と名付けます (図3)。構造体のメンバーの意味は、この後で解説します。

そして、csv 関数の実体は図4のようになります。csv_data 構造体を初期化して、タスクを作っているだけです。

これで、パイプラインの上流からデータが流れてくるたびに、csv_accept 関数が実行されることになります。この辺も4-1の tcp_server 関数と全く同じです。csv_accept 関数の定義の外枠は図5のようになります。上流から与えられたデータが第2引数になります。また、先ほど初期化していた csv_data 構造体は「task->data」という形で得られます。ただし、これは「void*」という形で格納されていますので、キャストで元の構造体へのポインタへ戻してやる必要があります。

```
struct csv_data {
    strm_array *headers;
    enum csv_type *types;
    strm_string *prev;
    int n;
};
```

図3 csv_data構造体

```
static int
csv(strm_state* state, int argc,
    strm_value* args, strm_value* ret) {
    strm_task *task;
    struct csv_data *cd = malloc(sizeof(struct csv_data));

    if (!cd) return STRM_NG;
    cd->headers = NULL;
    cd->types = NULL;
    cd->prev = NULL;
    cd->n = 0;

    task = strm_task_new(strm_filter,
        csv_accept, NULL, (void*)cd);
    *ret = strm_task_value(task);
    return STRM_OK;
}
```

図4 csv関数の実体

```
static void
csv_accept(strm_task* task, strm_value data) {
    strm_string *line = strm_value_str(data);
    struct csv_data *cd = task->data;
    ...
}
```

図5 csv_accept関数の外枠

フィールド数のチェック

さて、ここまできたらあとは表1にあるような仕様を順番に追加していただくだけです。

まずは、フィールド数のチェックから実装しましょう。CSVの各レコードが持つフィールド数が異なっていた場合、エラーにします。以前解説したように、Streamのパイプライン処理におけるエラーとそのデータを無視することですから、条件を満たさないときには、そこで単にreturnしてしまいます。例えば、フィールド数チェックのときには図6のようなコードを追加します。

```
if (cd->n > 0 && fieldcnt != cd->n)
    return;
cd->n = fieldcnt;
```

図6 csv_accept関数フィールド数チェック

「cd->n」はゼロで初期化されていますから、csv_acceptの初回実行時には0になっています。その初回実行時にcd->nを今回のフィールド数で初期化して、次回のチェックに備えるということになります。これで、最初のレコードと異なるフィールド数のレコードはすべて無視されることになります。

よく考えて「これが便利ははず」として決めた、この「単に無視」というポリシーですが、実用上は問題ありません。しかしプログラム開発中にバグがあったときに、どこに問題があるのか分からないのは予想以上にづらいものでした。このため少なくとも開発モードではエラーメッセージを得られるようにした方がよいなと思いました。近いうちにまたエラー処理について変更を加えようと思います。

行 継 続

CSVの仕様では文字列の中には特別な意味を持つ文字を含めても構わないことになっています。例えばコンマや（エスケープした）ダブルクォートや改行です。

しかし現在StreamのCSV処理では、ファイルから読み込んでくる時点で既に行に分割されたデータが上流から渡されてきます。このため文字列に改行を含んでいた場合、一つのレコードのはずのデータが、複数行に分割して渡されてしまいます。

そこで、文字列がクォートの中で終わっていたら（つまり、入力がクォート中に改行が来ている文字列だったら）、それをcsv_data構造体の中に保存しておき、次回の実行時に渡されてきた次のデータと結合して一つのレコードにしてから解釈するというコードを付け加えます（図7）。これには文字列がクォートの中で終わっていたら、フィールド数を数えるcount_fields関数が-1を返すことを利用します。

```
if (cd->prev) {
    strm_string *str = strm_str_new(NULL, cd->prev->len+line->len+1);

    tmp = (char*)str->ptr;
    memcpy(tmp, cd->prev->ptr, cd->prev->len);
    *(tmp+cd->prev->len) = '\n';
    memcpy(tmp+cd->prev->len+1, line->ptr, line->len);
    line = str;
    cd->prev = NULL;
}
fieldcnt = count_fields(line);
if (fieldcnt == -1) {
    cd->prev = line;
    return;
}
```

図7 行の継続処理

前の行 (cd->prev) が存在していた場合、まず前の行と現在の行とを結合します。この結合処理の部分が美しくないのですが、これは将来なんとかしたいところです。

フィールドの型

現在フィールドはすべて文字列として扱っていますが、明らかに数値を表現するものについては自動で数値に変換した方が便利なのが多いでしょう。そこでフィールドを構成する文字列が、すべて数字であった場合には整数、"."を間に挟む二つの数字列であった場合には浮動小数点数に自動変換することにします。これは切り出したフィールドを文字列化する部分を csv_value 関数に切り出し、内容に応じて整数または浮動小数点数に変換する処理を追加することで実現します。

また、型情報を csv_data の types メンバーに覚えておいて、フィールドの型が合わない場合にはエラーとしてそのレコードをスキップすることになります。

ヘッダー処理

世の中にある多くの CSV データは、先頭行にヘッダーが付いています。このヘッダーを適切に処理する機能を追加しましょう。

R という言語の data.table ライブラリには fread という CSV データを読み込む関数がありますが、これはファイルの先頭行の各フィールドが全部文字列であるときに、それはヘッダーであると見なしていました。これにならい、同じ条件が成立するときには先頭行はヘッダーと見なしてスキップすることになります。

しかし、ただスキップするだけでは情報が失われてしまうだけなのでもったいない気がします。

そこで、ヘッダーが与えられた CSV データを解析した結果は、ヘッダーで指定されたフィールド名を持つマップを返すことにします。

元々 Stroom の文法には Ruby などのハッシュに似たマップというデータ型が組み込まれていました。しかしデータがイミュータブル(更新不可)という性質からハッシュにはあまり意味がないなと思って、文法的には受け付けるけど実装はしていませんでした。

しかし、今回のことをきっかけにマップを「配列要素に名前が付いた配列」として再定義しようと考えました。これは別に珍しいデータ型ではなく、Python にも「名前付きタプル」として導入されていますし、R でも似たような機能があります。

そこで、配列を表す構造体 (strm_array*) に headers というメンバーを追加して、この構造体に名前情報があるときにはそれを strm_inspect 関数で要素ごとに表示するようにしました。アクセスも名前で行いたいところですが、それは時間の関係で後回しです。

この結果、ヘッダーのある CSV ファイルは図 8 のよう解析されます。

■入力

名前, 出身
matz, 鳥取
junko, 山口

■出力

[名前:"matz" 出身:"鳥取"]
[名前:"junko", 出身:"山口"]

図8 CSVのヘッダー処理

誤動作

型やヘッダーをサポートする機能を入れて、csv 解釈をちょっと賢くしてみました。が、実際に CSV データを与えてみると、あまりうれしくないケースがいくつかありました。

一つは型の不整合によるデータのスキップです。特に CSV の機能を確認するためのテストデータでは各フィールドの型がそろっていないものがあり、それらの型が合わないレコードは自動的にスキップされてしまいます。仕様通りとはいえ、忘れているとかなりびっくりします。もっとも実用的な CSV データはほとんど型がそろっているでしょうから、それほど問題にはならないかもしれません。

もう一つはヘッダーです。CSV データにはヘッダーがなく、かつすべてのフィールドが文字列であるようなものもそれなりあることが分かりました。現在の実装では、正当なレコードをヘッダーと見なして最初の行を失ってしまいます。これはかなり深刻な問題なので、オプションでヘッダーの有無の指定を許すなり、あるいはさらに賢いヘッダー判定条件を考えるなりしないと実用にならないかもしれません。

今回実装してみて初めて分かったのですが、CSV の解析はなかなか奥が深く、今回で完成したとはとてもいえません。特に上記の問題はかなり深刻なので、もう少し手を入れる必要がありそうです。

まとめ

今回の解説はソースコードを見ながらでないと理解しづらいかもしれません。「<https://github.com/matz/streem.git>」にあるソースコードを手元に置いて、src/csv.cを見ながら読むと理解の助けになるでしょう。今回解説している時点のソースコードには「201509」というタグを打っておきます。

タイムマシンコラム

CSV 処理はだいぶ前に実装していた

2015年9月号掲載分です。Streemの言語として基本的な部分はほぼ完成し、以後は機能の追加（と完成度と性能の改善）が主眼になります。手始めとして、データ処理言語としては非常に基本的な機能であるCSVの読み込み機能を実装してみました。

お気づきかもしれませんが、この回は周囲の回に比べて掲載時期が古いです。書籍構成上この位置に掲載しましたが、本来はずっと前に登場していたのです。この回で解説している配列とマップの同一視は、（時系列的には後になる）3-2で解説しているStreemのオブジェクト指向機能のベースになっています。もし本書を読み返すことがあったら時系列順に読み返すのも面白いかもしれません。

5-4 時間表現

今回は時間の表現とその操作の実装について解説します。国際規格に基づいて時刻を表現することにしましょう。タイムゾーンにも対応し、時刻の足し算や引き算もできるようにします。UNIX標準のAPIが充実していないため、設計と実装はなかなか大変でした。

今回は、時間の表現とその操作の実装について考えてみましょう。ストリーム型データ処理においても、時間の操作はそれなりに発生するものです。

例えば、東京の降水量データからグラフを描くというタスクを考えてみましょう。何年何月何日に何ミリ雨が降ったというCSVデータを読み込み、それをグラフ化するといったことになるでしょう。あるいは月ごと、年ごとの平均を取るなどの処理が必要になるかもしれません。そうすると「時間」というものをデータとして扱うが必要になってきます。

時間（または時刻）は、多くの場合には、

2016-06-01

のような文字列表現が用いられます。しかし文字列のままでは、大小比較だけではできないものの、経過日数を計算したりするのは容易ではありません。

プログラミングにおいて、時間を時間として取り扱いたくなるのは自然の欲求でしょう。

時間と時刻

ところで、「時間」という言葉はいろいろな意味で用いられます。時の長さ（例：この原稿を書くのにかった時間は2週間だ）にも用いられますし、ある瞬間の時間的位置を示すこともあります（例：今の時間は2016年5月1日午前10時だ）。このような曖昧な表現は、時々混乱を招きます。そこで本記事では今後ある特定の時間的位置のことを「時刻」と呼ぶことにします。英語で表現するならば、「time stamp」ですかね。

一方、時間の長さのことは「duration」と呼びます。適切な日本語が思い付かないのですが、あえて呼ぶならば「時間隔」とか「継続時間」とかですかね。あえて、こちらの意味に限定して「時間」という言葉を使う人もいます。 「時」の「間」なので、用語としては適切なのかもしれませんが、

場合によっては、これも混乱を招きそうな気がします。

時刻の文字列表現

人間は時刻を表現するのにさまざまな方法を用います。特に日付の表現は文化に強く依存します。例えば日本では、

平成28年5月1日

というような表現を使いますが、アメリカでは

May 1 2016

と書くことが多いでしょう。しかし、ヨーロッパでは

1 May 2016

という順になります。なんで文化ごとにこんなに違うんでしょう。

これでは混乱のもとなので日付と時刻の表現方法を規定する国際規格があります。それがISO8601です。

ISO8601では、

20160501（基本形式）

または

2016-05-01（拡張形式）

という形で日付を表現します。年→月→日の順番は、間接的に日本式が合理的であることの証明になっている気がしますね。経過時間（duration）も含める場合、

20160501T100000+0900

または

2016-05-01T10:00:00+09:00

という表現になります。基本形式は数値と区別がつかないので、実際の時刻表現には拡張形式の方が多く用いられているような気がします。

ISO8601はJISにも翻訳されていて、それはJIS X 0301になっています。興味深いことにJISでは、ISO8601に対していくつかの拡張をしています。

- 日付区切りの「-」の代わりに「.」を使える
- 和暦の元号をサポートしているので「H28.05.01」または「平28.05.01」などと書ける

時刻の表現方法

ISO8601を使えば時刻を文字列で表現できるようになるわけですが、文字列表現のままではプログラムとしてはどうにも使いにくいです。そこで、時刻型を導入することを考えるわけですが、時刻をどのように表現するのが適切でしょうか。

時間というものは過去から未来へ1次元に流れるものですから、数値で表現するのが適切そうです。多くのシステムでは時刻は、原点となるある特定の時刻（エポックと呼びます）からの経過時間によって時刻を表現します。

Linuxを含めた多くのUNIXシステムでは、1970年1月1日00:00 (UTC) をエポックとします。経過時間は秒によって表現します。つまり、2016-05-01T10:00Zは、エポックである1970-01-01T00:00Zからの経過秒数である1462096800で表現します。例えば、現在時刻を取り出すシステムコールtime(2)は、以下のようなAPIです。

```
time_t t = time(NULL);
```

tにはエポックからの経過秒数が整数で与えられます。

しかし、いつも秒単位で話が済むとは限らず、1秒以下の情報を取り出す必要があるかもしれません。UNIXでもそう思ったのか、より新しいシステムコールgettimeofday(2)が追加されています。こちらは秒以下の時刻がマイクロ秒単位で与えられます。

```
struct timeval tv;  
gettimeofday(&tv, NULL);  
tv.tv_sec; // => 秒数  
tv.tv_usec; // => マイクロ秒
```

マイクロ秒なのにusecなのはおかしいと思われるかもしれません。これはマイクロ(100万分の1)を意味するギリシャ文字 μ (ミュー)とアルファベットのuが似ているからという理由なんだそうです。

その後、さらにさらに細かい時刻分解能が必要なこともありえると考えたのか、POSIX.1-2008

ではより新しいシステムコール `clock_gettime(2)` が追加されています。

```
struct timespec tp;
clock_gettime(CLOCK_REALTIME,&tp)
tp.tv_sec; // => 秒数
tp.tv_nsec; // => ナノ秒
```

`clock_gettime(2)` の 1 秒未満の部分は、マイクロ秒ではなくナノ秒単位です。

UNIX では 1970 年 1 月 1 日からの秒数で時刻を表現しますが、すべてのシステムで同じ表現を用いているわけではありません。例えば Windows NT では、エポックを 1601 年 1 月 1 日に、経過時間を秒単位ではなく 100 ナノ秒単位で表現します。

時刻型の構造体

では、早速時刻を表現するデータ型を Stream に追加しましょう。以前、kvs の実装などで行ったのと同基本的な仕組みは同じです。

まずは、時刻を表現する構造体を定義します (図 1)。メソッドを持つ構造体の先頭には

```
struct strm_time {
    STRM_AUX_HEADER;
    struct timeval tv;
    int utc_offset;
};
```

図 1 時刻型構造体

```
STRM_AUX_HEADER;
```

というマクロを配置します。実際の時刻を表現する型として「struct timeval」を、時差を表現する整数として「utc_offset」を用意します。struct timeval は図 2 のような定義の構造体で、マイクロ秒の分解能で時刻を表現します。

struct timeval の採用に当たって、man ページには図 3 のような気になる記述がありました。「非推奨」とは穏やかではありません。obsolete は「古くなった」というような意味で「廃止予定」に近いニュアンスがあります。

```
struct timeval {
    time_t      tv_sec;        /* seconds */
    suseconds_t tv_usec;      /* microseconds */
};
```

図 2 struct timeval の定義

POSIX.1-2008 marks gettimeofday() as obsolete, recommending the use of clock_gettime(2) instead.

訳：POSIX.1-2008によればgettimeofday()は非推奨であり、代わりにclock_gettime(2)の利用が推奨される。

図 3 gettimeofday の man ページの記述

これはclock_gettime()の採用を真剣に考えないといけないなと思って調べてみたところ、MacOSではいまだにclock_gettime()が未実装なのだそうです。POSIX.1-2008という何年も前の規格で定義されているものが未実装なのは実際どうなのよ、と思わないでもないですが、移植性を考えるとないものは仕方ありません。信頼と実績のgettimeofday()を使うことにしましょう。

時刻のUTCからの時差を秒単位で表現するのがutc_offsetです。例えば日本はUTCよりも9時間進んでいるので、日本時間を表す時刻においてutc_offsetは

```
9×60×60 = 32400
```

になります。

原理主義的な立場に立てば、時刻型に時差情報は不要です。ある一瞬を表現する時刻にとって時差は意味がありません。日本での午後9時は、UTCの正午であり、表現こそ違えども、同じ時刻には違いありません。

時差情報は、それを文字列表現に変換するときに必要なものです。ある時刻が9時であるという情報は、そのタイムゾーンの情報なしには意味がありません。

しかし時刻には出自があり例えば

```
2016-05-01T10:00:00+09:00
```

という表現から作られた時刻は、デフォルトでは元のタイムゾーンで表示されてほしいと思うのは当然のことです。そこで、タイムゾーンを指定して生成された時刻データには時差情報を埋め込み、表示のデフォルトにすることにしました。

UTC

ここまでUTCという言葉の説明なしに使ってきましたが、一度きちんと説明しておきましょう。UTC (Coordinated Universal Time) は時刻の原点です。頭文字を取ったCUTでないのには深い理由があるそうです。規格制定時にこの用語の正式名称について、英語のCoordinated Universal Timeとフランス語のTemps Universel Coordonneの間の激しい綱引きがあり、結局どちらでもないUTCが採用されたという経緯があると聞いたことがあります。

昔は時刻の原点はイギリスのグリニッジ天文台にちなんでGMT (Greenwich Mean Time) と呼ばれていました。天文計測によって求められるGMTに対して、より正確さを追求するUTCはセシウム133が91億9263万1770回振動する時間を1秒として原子時計で計測して求められます。

地球の自転速度のふらつきにより、GMTとUTCの間には微妙なズレが生じます。それを補正するため、ごくまれに「うるう秒」が挿入されます。

うるう秒はやっかいな問題で、例えば2012年7月1日にはうるう秒挿入に伴う広範囲の障害が

発生しました。

時刻型データの生成

では、時刻型データの生成を考えてみましょう。図4は現在時刻を生成する`now()`関数の実装です。`now()`は省略可能な引数があり、指定されるとそれがタイムゾーンになります(表1)。

表記	意味
Z	UTC
+09:00	UTC+9時間
+0900	UTC+9時間(省略形)
+900	UTC+9時間(さらに省略形)
-09:00	UTC-9時間
-0900	UTC-9時間(省略形)
-900	UTC-9時間(さらに省略形)

表1 タイムゾーン指定表記

```
static int
time_now(strm_stream* strm, int argc, strm_value* args, strm_value* ret)
{
    struct timeval tv;
    int utc_offset;

    switch (argc) {
    case 0:
        utc_offset = time_localoffset();
        break;
    case 1: /* timezone */
    {
        strm_string str = strm_value_str(args[0]);
        utc_offset = parse_tz(strm_str_ptr(str), strm_str_len(str));
        if (utc_offset == TZ_FAIL) {
            strm_raise(strm, "wrong timezeone");
            return STRM_NG;
        }
    }
    break;
    default:
        strm_raise(strm, "wrong # of arguments");
        return STRM_NG;
    }
    gettimeofday(&tv, NULL);
    return time_alloc(&tv, utc_offset, ret);
}
```

図4 `now()`の実装

タイムゾーンの指定方法には、例えば日本標準時として「JST」のような省略形を用いることや、「Asia/Tokyo」のような都市名を用いることもあります。しかしJapan Standard TimeとJamaica Standard Time(というものがあるとして)の混同など曖昧性が発生したり、世界各地の都市名のテーブルが必要になったりします。このため今回は対応していません。

`time_now()` がやっていることは単に引数を処理して、`gettimeofday(2)` で現在時刻を取得し、`time_alloc()` を呼び出しているだけです。

図5に示すように、`time_alloc()` の実装も全然難しくありません。通常の `Stream` オブジェクトの初期化 (`type` と `ns` の設定) をした後は、与えられた `struct timeval` の `tv_usec` を正規化して、負の値や1秒を越える値を持たないように調整しているだけです。

```
static int
time_alloc(struct timeval* tv, int utc_offset, strm_value* ret)
{
    struct strm_time* t = malloc(sizeof(struct strm_time));

    if (!t) return STRM_NG;
    t->type = STRM_PTR_AUX;
    t->ns = time_ns;
    while (tv->tv_usec < 0) {
        tv->tv_sec--;
        tv->tv_usec += 1000000;
    }
    while (tv->tv_usec >= 1000000) {
        tv->tv_sec++;
        tv->tv_usec -= 1000000;
    }
    memcpy(&t->tv, tv, sizeof(struct timeval));
    t->utc_offset = utc_offset;
    *ret = strm_ptr_value(t);
    return STRM_OK;
}
```

図5 `time_alloc()` の実装

時差の求め方

実際問題として、実装に苦労したのは上で挙げたような部分ではなく、ローカルタイムとUTCの時差を求める関数 `time_localoffset()` の実装でした。

最初はローカルタイムで「1970-01-01T00:00:00」に対応する時刻を求めれば、時差が求められるのではないかというアイデアを使っていました。しかし、Twitterで「1970年と現在で時差が異なる場合がある」との指摘をもらいました^{*1}。

*1 <https://twitter.com/nalsh/status/717021969758040064>

確かに日本では通常タイムゾーンの変化がないので忘れがちですが、夏時間が導入されている国は多いですし、最近だと北朝鮮が国のタイムゾーンを+09:00から+08:30にずらした事例があります。実は日本でも1948年から1952年までは夏時間が導入されていたことがあったそうです。1970年1月時点の時差では対応できないケースがたくさんありましたね。

ではどうすればいいのだろうか悩んでいると、これまたTwitterで、gmtime(3)とmktime(3)を使えば簡単にできることを教えてもらいました^{*2}。140文字以内のコードで書いてしまうとは驚きです。

教えていただいたコードをベースに若干修正したものが図6になります。厳密に言うと図6のコードではプログラム実行中にタイムゾーンが変化した場合に対応できませんが、ここでは気にしないことにしましょう。将来、世界中を飛び回る人のデバイスでStreamを動かしたいというニーズが発生したら考えることにします。

と思っていたのですが、国によっては冬時間と夏時間の切り替えは年に2回発生するわけで、そのタイミングでプログラムが実行中であることは十分にあり得る事です。やはり対応しないといけませんね。

このコードのキモは、gmtime(3)の使い方にあります。gmtime(3)はtime_tで表現される秒単位の現在時刻を、UTCにおける日付や時間表現に分割したstruct tmに変換する関数です。ローカルタイムにおけるstruct tmに変換する関数はlocaltime(3)です(_rがついたものはそのスレッドセーフ版)。それからmktime(3)はstruct tmからtime_tへ変換するlocaltime(3)の逆を行う関数です。

さて、gmtime(3)を使ってUTCで表現した日付をmktime(3)を使ってローカルタイムでtime_tに変換すると、時差分だけずれた時刻が得られます。後はdifftime(3)を使って差分を取れば秒単位の時差が得られるというわけです。

UNIXの時間関数は秒単位のもの、マイクロ秒単位のもの、ナノ秒単位のものなどが混在していたり、プラットフォームによって使えたり使えなかったりする上に、時差の扱いがあまり充実していません。このため正直あまり使いやすくないのですが、こうやって知恵を絞って機能を組み合わせることで、意外となんとかなるものです。

```
static int
time_localoffset()
{
    static int localoffset = 1;

    if (localoffset == 1) {
        time_t now;
        struct tm gm;
        double d;

        now = time(NULL);
        gmtime_r(&now, &gm);
        d = difftime(now, mktime(&gm));
        localoffset = d;
    }
    return localoffset;
}
```

図6 ローカルタイムの時差の求め方

*2 <https://twitter.com/unak/status/717026294337122304>

時刻操作の実装

後は、時刻型のメソッドを定義します。とりあえず定義するメソッドは表2の四つです。

名前	機能	備考
+	時刻の加算	時刻+数値(秒数)→時刻
-	時刻の減算	「時刻-時刻」または「時刻-数値」
number	浮動小数点数への変換	エポックからの秒数を浮動小数点数で得る
string	文字列への変換	省略可能引数としてタイムゾーンを取る

表2 時刻型のメソッド

注意すべきは減算メソッドの型です。時刻から時刻を減算すると経過時間 (duration) が求められます。これをどの型で表現するかは悩ましい問題です。案としては、durationを表現するデータ型を導入する方式と、経過秒数を数値 (浮動小数点数) で表現する方法が考えられます。

浮動小数点数を採用した場合の懸念は、時刻情報を表現しきれないかもしれないことです。struct timevalのサイズは64ビット、浮動小数点数もサイズは64ビットですが、浮動小数点数で実質的に数値表現に使えるのは仮数部52ビットだけです。もっともマイクロ秒部は最大でも100万未満であり、20ビットで表現できます。このため秒部が32ビットで表現できる2038年まではこれで行なうとかなりそうです。

シンプルさを重んじて、今回は時刻同士の減算結果を浮動小数点数で表現することにします。

あと注意すべき点は、減算は時刻同士だけでなく、時刻と数値とでも可能であること、文字列変換関数string()は省略可能な引数としてフォーマットを取り、時刻を任意のタイムゾーンで表現可能であることでしょうか。

任意のタイムゾーンで時刻表示

任意のタイムゾーンでの時刻表示にもちょっとした工夫が必要だったので紹介しておきます。ある時刻から任意のタイムゾーンにおける時間表現 (struct tm) を得る方法は自明ではありません。UNIXの時間関数は、UTCかローカルタイムのいずれかを扱うようにしか設計されていないからです。

しかし一見難しそうなこの処理も、ちょっとした工夫で実現できます。秒数単位である時刻を示すtime_tと、UTCからの(秒単位の)時差utc_offsetから、そのタイムゾーンでのstruct tmを得る関数get_tmの実装を図7に示します。

```
static void
get_tm(time_t t, int utc_offset, struct tm* tm)
{
    t += utc_offset;
    gmtime_r(&t, tm);
}
```

図7 任意のタイムゾーンのtmを得る関数

実装があまりにシンプルなので拍子抜けしたかもしれません。任意のタイムゾーンのUTCとの時差分だけずらした時刻に対して`gmtime_r(3)`を適用することで、そのタイムゾーンでの`struct tm`が得られるというものです。考えてみれば当たり前でも、私にとってはちょっとした驚きでした。

時刻リテラル

さて、このようにしてStreamに時刻型を導入しましたが、この機会に時刻リテラル（数値や文字列を直接記述した定数）も導入することにしましょう。時刻リテラルを持つプログラミング言語はそれほど多くはありませんが、データ処理における時刻の扱いの重要性を考えると、リテラルがあっても決しておかしいことではありません。

そこで、時刻リテラルをどのような表記にするかですが、可能であればISO8601を使いたいところです。しかし、残念ながら

```
2016-05-01
```

のような表現は、整数の引き算に見えてしまうのでこのままでは使えそうにありません。ううむ。そこで、JIS X 0301を参考に日付の間を「.」で区切ることにしましょう。つまり、Streamの時刻表現は以下のようになります。

```
2016.05.01
2016.05.01T00:00:00Z
2016.05.01T00:00:00.342Z
2016.05.01T00:00:00+09:00
```

これは便利だと思ったのですが、時刻表記の中にタイムゾーンを表すプラスやマイナスが含まれていて、若干混乱を招きそうな気がします。Rubyなどは時刻リテラルを持たず、以下のような形式のメソッド呼び出しで時刻オブジェクトを生成しています。

```
Time.new(2016,5,1,0,0,0)
```

これでもいいかなと思う気持ちもあって悩むところです。まあ、時刻リテラルはせっかく実装したのでこのまま残しますが、しばらく使ってみて具合が悪ければ、後で削除するかもしれません。この辺がまだまだユーザーのいない言語の気楽なところですね。

時刻リテラルの実装

時刻リテラルの実装はさほど難しくはありません。構文解析器である`lex.l`に時刻リテラルを解釈

する正規表現を追加した上で、時刻リテラルを表現するノードを追加するだけですみます。

実際のlex.lの変更点は図8のようになります。

後はstrptime(3)などを使って文字列表現をstruct timevalに変換するだけです。今まで用意してきたルーチンを活用すればなんの困難もありません。

```
DATE    [0-9]+\.[0-9]+\.[0-9]+
TIME    [0-9]+":"[0-9]+(":[0-9]+)?(}\.[0-9]+)?
TZZONE  "Z"|["+-"][0-9]+(":[0-9]+)?
%%

{DATE}("T"{TIME}{TZZONE}?)? {
    lval->nd = node_time_new(yytext, yyleng);
    LEX_RETURN(lit_time);
};
```

図8 lex.lの変更点

CSVの時刻対応

時刻リテラルよりも大切なことがあります。それはCSVの時刻対応です。予想されるStroomのユースケースで日付や時刻のデータの入力元として最有力なのはCSVファイルでしょう。

現在はCSVファイルの各フィールドで文字列か数値かを自動で判別するようにしていますが、これに時刻データへの対応を追加します。フィールドの値がISO8601形式またはStroomの時刻リテラルの形式であれば、それは時刻データであると見なして、時刻オブジェクトに変換します。

簡単だと思って手を付けたら、これまでの浮動小数点数の対応に大きなバグがあって、むしろその修復に手間取りました。

今回の一連の開発でも気付いていなかった重大なバグが散見されたので、そろそろ言語仕様へのテストを導入する必要があるそうです。

まとめ

今回はStroomに時刻データ対応を追加しました。長年のUNIXプログラマとして、普段はUNIXが提供する機能とAPIには満足しているのですが、時間関連のAPIについては不完全さが目に付きます。特に今回のようにUTCかローカルタイムでないタイムゾーンを扱おうとすると、とたんに難しくなります。

もっとも時間と暦の扱いは、元々相当に複雑で、どこでもAPIの設計には苦勞しています。Rubyも例外ではなく、Rubyの時刻を扱うTimeクラスや日付を扱うDateクラスの設計もさまざまな事情を反映して相当複雑になっています。

この辺りのAPI設計事情については、田中哲著『APIデザインケーススタディ』に詳しいです。この本ではまるまる1章を割いて時間にまつわるデザインの難しさについて解説しています。デザインの経緯を知るものとしては涙なくしては読めません。そうでない人には知らない世界をのぞく楽しい読み物として読めるでしょう。

タイムマシンコラム

複数のタイムゾーンに見事対応できた

2016年6月号掲載分です。私たちが時間というものを考えるときに、過去から未来に流れる非常にシンプルなものをイメージします。また、物理の基本的値として頻繁に登場するために数学的な意味での値だと捉えがちです。

しかし、いざプログラミングで時刻と時間を扱おうと思うと、意外にも文化や政治に関連する要素がたくさん含まれていることを発見します。例えば、時刻をどのように表現するかは文化に依存します。タイムゾーンと時差は、国や地域ごとに政治によって決定されます。うるう秒がいつ挿入されるかも観測によるズレに基づいて会議で決定されています。

元々 UNIX やそれを標準規格化した POSIX は時刻と時間の処理に関してあまり深い関心がなかったようで、API があまり充実していません。今回は POSIX の関数だけを使って複数のタイムゾーンを扱う難しさに真っ向から挑戦してみました。かなり苦悩しましたが、事前に期待した以上にうまくいったのは大変満足です。

5-5

統計基礎の基礎

「ビッグデータ」というフレーズが話題になっていますが、数学は昔から大きなデータを扱うために努力しており、それは「統計」と呼ばれています。今回は統計の基礎の基礎の部分をStreamでどのように実現するかについて解説します。

あまり大きな声では言えないのですが、学生時代から私は数学が苦手でした。一般的にプログラミングは理系の活動だと思われており、理系の人は数学が得意ということになっています。なので、私が数学苦手だということかなり意外に思われるようです。あるいは苦手といっても、さほどでもないだろうと思われているようです。

しかし、実際問題として、数学が苦手なのは本当に本当で、高校時代には数学Ⅲの成績が「1」（しかも10段階で）とか、高校3年生1学期での定期試験の平均点が16点とか悲惨な有様でした。大学入試でも、入学してからの数学でも大変に苦労したものです。

考えてみると、数学や算数に伴う手計算に対して（コンピュータにやらせればよいと）モチベーションが上がらなかったことで、興味が持てなかったことがつまずきのきっかけになったのではないかと思います。間違いやすい人間に正確さを必要とする計算をさせてはいけないのではないかと今でも本気で思います。

もちろん、コンピュータサイエンスは数学の上に成り立っていますし、数学とは切っても切れない関係にあります。しかしプログラミングのすべてに数学が必要というわけではありません。プログラミング活動の多くはユーザーの要求を把握することで占められますから、数学との関係はそれほど強くありません。特に私が昔から興味があったプログラミング言語のデザインやユーザーインタフェースといった分野では、数学を用いることはあまりありません。

以前は「だから数学なんてそんなに必要ない」とうそぶいていたのですが、Rubyを開発するにはやはり数学が重要な局面がしばしばありました。コミュニティのメンバーに助けて（間違いを指摘して）いただきながら、少しずつ頑張っています。とはいえ、いまだに苦手意識は抜けません。

さて、長々と余談を続けましたが、ストリーミングプログラミングを実現するStreamにおいて、恐らく主要な適用分野になりそうなのが、データ処理です。典型的な例としては、テストの成績データをCSVで読み込んで、成績処理をするような使い方です。そうすると、数学が苦手とは言ってられません。今回は統計的データ処理の基礎の基礎について一緒に学びましょう。

合計と平均

さて、まずは小学生レベルの平均から考えましょう。小学生の娘に聞いたところ、平均は小学校5年生で習うそうです。算数の教科書も貸してもらいました。教科書には平均の定義は

平均 = 合計 ÷ 個数

と書いてありました。平均はそれぞれ異なる値をならすとどれくらいになるかを示す値です。中間試験が14点で、期末試験が18点だと平均は

$(14 + 18) \div 2 = 16$

で16点ですね。5-4までの時点でStreamにはストリームのデータ個数を求めるcount()とストリームの合計を求めるsum()を提供しているので、これらを組み合わせれば平均を求めるのは簡単です。average()関数の実装は図1のようになるでしょう。

```
struct avg_data {
    double sum;
    strm_int num;
};

static int
iter_avg(strm_stream* strm, strm_value data) {
    struct avg_data* d = strm->data;
    d->sum += strm_value_flt(data);
    d->num++;
    return STRM_OK;
}

static int
avg_finish(strm_stream* strm, strm_value data) {
    struct avg_data* d = strm->data;

    strm_emit(strm, strm_flt_value(d->sum/d->num), NULL);
    return STRM_OK;
}

static int
```



```

exec_avg(strm_stream* strm, int argc, strm_value* args, strm_value* ret, int
avg) {
    struct avg_data* d;

    if (argc != 0) {
        strm_raise(strm, "wrong number of arguments");
        return STRM_NG;
    }
    d = malloc(sizeof(struct avg_data));
    if (!d) return STRM_NG;
    d->sum = 0;
    d->num = 0;
    *ret = strm_stream_value(strm_stream_new(strm_filter, iter_avg, avg_finish,
(void*)d));
    return strm_ok;
}

```

図1 average()の実装

exec_avg関数でタスクを作り、要素ごとにiter_avg関数を実行して合計を求め、最後にavg_finish関数で合計を個数で割って平均値を求めます。処理がストリームを構成するタスクに分割されているので少々分かりにくいですが、やっていることはシンプルな平均値の計算です。

合計の罫

平均の計算なんて簡単だって思いますよね。まあ、小学校5年生レベルなので当然です。しかし、リアルワールドは恐ろしいところです。こんなに簡単に見える処理にも罫が隠されているのです。

先に述べた通り、平均の求め方は、合計を取って個数で割ります。しかし、この合計の求め方に罫があるのです。それは誤差です。

コンピュータは正確な実数を表現することができないので、近似として浮動小数点数を用いて計算しますが、これには近似につきものの誤差が発生します。そして、浮動小数点数には誤差に関する二つの「罫」があります。

一つは人間にとってはキリの良い値であっても、コンピュータにとってはそうでないことがあるということです。例えば、0.1は非常にシンプルな値ですがその意味は「1を10で割ったもの」なので、浮動小数点数が採用している2進数では割り切れません。つまり、どこかで打ち切らなければならず、そこに誤差が発生します。

もう一つの罫は、浮動小数点同士で計算を繰り返すと誤差が蓄積しやすい性質があることです。数個の値を合計するくらいであれば問題にはならないでしょうが、これが数万個、あるいは数千万個の要素の合計だと誤差が無視できないレベルになる可能性があります。例えば図2のプログラム

は同じ数の1000万個の平均を取ります。repeat() は第1引数で指定された値を第2引数で指定した回数だけ生成するストリームを作る関数です。

```
repeat(0.15,10000000) | average() | stdout # 実行結果  
# 0.1499999999834609
```

図2 誤差の出る平均

同じ数の平均を取るわけですから、結果は同じになるはずですが、実際には誤差の蓄積により微妙に差が発生しています。

合計なんて単純な足し算だと思ってしまうかもしれませんが、誤差のことを考えると結構面倒なものなのです。

Kahanのアルゴリズム

もちろん、コンピュータサイエンスはこのような事態を回避する方法についても考えています。

誤差を抑えて浮動小数点数を合計するアルゴリズムとしては、Kahanのアルゴリズムが知られています。Kahanのアルゴリズムでは、加算によって失われる下位ビットの情報を次の計算に繰り入れることで誤差を補償するものです。ウィキペディアにある擬似コードは図3のようになっています。

```
function kahanSum(input)  
  var sum = 0.0  
  var c = 0.0      ← 処理中に失われる下位ビット群の補償用変数  
  for i = 1 to input.length do  
    y = input[i] - c  ← 問題なければ、cはゼロ  
    t = sum + y      ← sumが大きくyは小さいとすると、yの下位ビット群が失われる  
    c = (t - sum) - y ← (t - sum) はyの上位ビット群に相当するのでyを引くと下位ビット群  
                      が得られる (符号は逆転)  
    sum = t          ← 数学的にはcは常にゼロのはず。積極的な最適化に注意  
  next i            ← 次の繰り返しでyの失われた下位ビット群が考慮される  
  return sum
```

図3 Kahanのアルゴリズム

出典は「<https://ja.wikipedia.org/wiki/カハンの加算アルゴリズム>」。

このアルゴリズムを使って、図1の実装を書き換えると図4のようになります。変更が必要なのはstruct avg_dataとiter_avg関数だけです。

これによって若干計算量は増えますが、誤差を抑えることができます。この改善をした後、図2

のプログラムを実行すると、繰り返し回数に関わらず正しい（元と同じ）値が得られます。

この件もそうですが、浮動小数点を含む計算には誤差が付きものです。アルゴリズムを選択する場合も可能であれば誤差を考慮したものを採用するべきです。

```
struct avg_data {
    double sum;
    double c;
    strm_int num;
};

static int
iter_avg(strm_stream* strm, strm_value data) {
    struct avg_data* d = strm->data;
    double y = strm_value_flt(data) - d->c;
    double t = d->sum + y;
    d->c = (t - d->sum) - y;
    d->sum = t;
    d->num++;
    return STRM_OK;
}
```

図4 average()の改善
Kahanのアルゴリズムを使った。

平均と分散（標準偏差）

平均によって複数の値に対する全体の傾向を知ることができますが、あくまでも全体を「ならした」値なので、どうしても情報が失われます。例えば1クラスが20人で、二つのクラスA、Bが100点満点の試験を受けたとき、Aクラスでは全員が50点、Bクラスでは10人が100点、残り10人が0点だった場合、どちらのクラスも平均は50点になります。

AクラスとBクラスでは成績の傾向が全く異なりますが、平均からは区別が付きません。このような違いを検出するには、平均とは別に値のバラつき具合も把握する必要があります。値のバラつき具合は「標準偏差」によって表現します。X1,

X2, …Xiの平均を μ としたとき、標準偏差は図5の式によって定義される「分散」の正の平方根 σ です。

あまり数学に慣れていない（私のような）人には難しげな式ですが、要するに、個々の値と平均との差を2乗して合計した値を個数で割ったものです。

これを先ほどの2クラスの成績で計算すると、Aクラスは全員が同じ得点でバラつきがないので

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

図5 分散（標準偏差の2乗）の定義

標準偏差は0、Bクラスは半分が満点、半分が0点ですから標準偏差は約51.3になります。同じ平均50点でもかなり性質が違ってくるのが分かりますね。

ストリーミングアルゴリズム

さて図5の定義を見ると、標準偏差を求めるためには、まず平均を求めてから、個々の値との差を計算する必要があります。つまり、何も考えずにそのまま実装すると平均の計算のために全部の値を読み取り、その後、標準偏差のためにもう一度同じ値を最初から読み取る必要があります。例えば図6のプログラムはループを2回使って標準偏差を計算します（奥村晴彦氏著の『C言語による最新アルゴリズム事典』p.254から引用）。

```
int i, n;
float x, s1, s2;
static float a[NMAX];

s1 = s2 = n = 0;
while (scanf("%f" &x) == 1) { /* 1回目のループ */
    if (n >= NMAX) return EXIT_FAILURE;
    a[n++] = x; s1 += x;
}
s1 /= n; /* 平均 */
for (i=0; i<n; i++) { /* 2回目のループ */
    x = a[i] - s1; s2 += x * x;
}
s2 = sqrt(s2/(n-1)); /* 標準偏差 */
printf("個数:%d 平均:%d 標準偏差: %g", n, s1, s2);
```

図6 標準偏差の計算

しかし、同じデータを何度も読むのは無駄ですし、特にストリーミング処理の場合には処理途中の（超巨大かもしれない）データをどこかに保存しておくことが必要になりそうです。

こんな無駄を避けるために、データの一つずつ読み込むだけで処理できるアルゴリズムを「ストリーミングアルゴリズム」と呼びます。調べてみると標準偏差の計算にもストリーミングアルゴリズムが存在していました。

同じく『C言語による最新アルゴリズム事典』によると、図7のプログラムのように計算すると、データを一度通読するだけで、かつ誤差を比較的小さく抑えながら標準偏差を計算できるそうです。

このアルゴリズムを使って、分散と標準偏差を計算する関数 (stdev() と variance()) を Stream に追加します。誌面の都合上、ここではコードは明示しませんが、仕組みとしては図1の average() 関数と同じようなものです。ソースコードでは stat.c の exec_stdev() 関数の周辺を眺めていただけるとよいでしょう。

```

int i, n;
float x, s1, s2;

s1 = s2 = n = 0;
while (scanf("%f" &x) == 1) {
    n++;
    x -= s1;
    s1 += x / n;
    s2 += (n-1) * x * x / n;
}
s2 = sqrt(s2/(n-1));
printf("個数:%d 平均:%d 標準偏差: %g", n, s1, s2);

```

図7 標準偏差のストリーミングアルゴリズム

Streamでの標準偏差の計算

では、この新しく定義したstdev()関数を使って先ほどの例題の標準偏差を計算してみましょう。まずは計算のベースになる全員が50点を取ったクラスAの成績を生成する方法から考えましょう。

20人全員が50点を取ったのですから、50が20個連続する値を生成すればよいわけですね。このためには上でも説明したrepeat()を使います。

```
repeat(50,20)|stdout
```

とすると、20個の50が標準出力に書き出されます。平均を取るためには

```
repeat(50,20)|average()|stdout
```

標準偏差を取るためには

```
repeat(50,20)|stdev()|stdout
```

とします。実に簡単ですね。

クラスBの方は、100点が10人、0点が10人ですから、二つのストリームを結合することにししょう。100を10個、続けて0を10個並べたストリームを得るためにはrepeat()とconcat()を組み合わせ、図8のように記述できます。

とはいえ、例題のクラスのような全員同じ点とか、半分満点で半分0点のようなことは現実にはまず起こりません。テストの得点のような作為的でない値は、平均周辺が最も多く、平均から離れ

るに従って次第に少なくなる傾向があり、我々は経験的にそのことを知っています。

```
concat(repeat(100,10),repeat(0,10))|stdev()|stdout
# 51.29891760425771
```

図8 クラスBの成績の生成と標準偏差

偏差値

さて、平均と標準偏差はストリーミングアルゴリズムで計算できましたが、どうしてもストリーミングアルゴリズムでは計算できない指標もあります。

例えば、成績処理につきものの順位と偏差値はいずれもストリーミングでは計算できません。順位の計算のためには成績順のソートが必要ですし、偏差値の計算のためにはあらかじめ平均と標準偏差を計算しておく必要があります。

ここでは例題として偏差値を計算してみましょう。偏差値の定義は

$$\text{偏差値} = (\text{得点} - \text{平均}) \times 10 / \text{標準偏差} + 50$$

です。これで偏差値を計算するプログラムを図9に示します。

```
input = fread("result.csv")|map{x->number(x)}
avs = input | average() # 平均
sts = input | stdev()   # 標準偏差
zip(avs, sts) | each{ x ->
  avg = x(0); std = x(1)
  fread("result.csv") | map{x->number(x)} | each { score ->
    ss = (score-avg)*10 / std + 50
    print("得点: ", score, "偏差値:", ss)
  }
}
```

図9 偏差値の計算

zipを使っているところはちょっと分かりにくいと思うので解説しておきます。同じ入力ストリームから平均と標準偏差を求めました。average()もstdev()も1要素を与えるストリームを返します。この二つのストリームにzip()関数を適用すると「要素を取り出して配列にまとめる」という働きをします。

ですから続く`each()`はループではなく、取り出してきた値に対して処理を実行するだけの作用を持ちます。

一度データを読み込んで平均と標準偏差を求めた後で、またデータを読み込み直す必要があるのがどうにも格好悪いですね。実際に読み込むのは避けられないにしても、もうちょっとマシな指定方法がないのか少し考えてみましょう。`future`とか`promise`を使えばなんとかなりそうな気がします。ちょっと宿題にさせてください。

ソート

データをその値の大小によって並び替えるソートはコンピュータサイエンスでは重要なトピックで、高速化のためにさまざまなアルゴリズムが考案されています。現時点で「最も高速」と考えられているのは、その名も「quick sort」というアルゴリズムです。

Cでは標準ライブラリに`qsort(3)`という関数が提供されていて、メモリー上のデータを簡単にソートできます。それは良いのですが、一つ重要な問題があります。それはメモリーに収まる程度の大きさのデータしかソートできないという点です。

この問題は「外部マージソート」という手法によって一応解決できます。「外部マージソート」はメモリーに格納できる程度のデータを分割して読み込んだ上で、それぞれをソートした上でファイルに書き出します。そして個別のソートされたファイルを先頭から読み出し、つなぎ合わせることで全体をソートします。手順は以下の通りです。

1. 元データセットからメインメモリーに収まるサイズのデータを読み込む
2. 読み込んだデータをソートし、ファイルに書き出す
3. すべてのデータを処理するまで1と2を繰り返す
4. 書き出した複数のファイルから、それぞれの1要素を読み込む
5. 最も小さい要素を結果ファイルに書き出す
6. 書き出した要素を取り込んだファイルからまた1要素を読み込み、全データを結果ファイルに書き出すまで繰り返す

UNIXの`sort`コマンドもこのアルゴリズムを使ってソートをしています。

大規模ソートは難しい

ただ、この「外部マージソート」にもいくつかの課題もあります。最初の課題は作業ファイルを作るのでディスクを消費してしまう点です。データ規模が大きくなればディスク容量についても心配になります。外部マージソートが必要になるということは、必然的に大きなデータを扱っているわけですから、心配はなおさらです。これは適切なディスク配置を心がける以外に対応策はないように思えます^{*1}。

もう一つの課題は、Streamでは任意のデータをストリームに流すことができ、ソートの対象になり得るという点です。つまり、単なる数値や文字列であれば作業ファイルへの書き出しは簡単ですが、構造のあるデータは情報を失わずにファイルに書き出すことがより難しくなります。この点についてはJSONやMessagePackのような構造データを表現できる記法で書き出すことである程度対応できるでしょう。

このようにデータ規模が大きくなれば、いろいろと考慮しなければならないことも増えることが分かります。

と、ここまで偉そうなことを書いてきましたが、とりあえず手早く作業を進めたいので、今回はメモリー中でソートすることにします。外部マージソートによる大規模データへの対応は今後の課題ということにしましょう（肩すかしですが）。

```
input | sort() | output
```

というパイプラインでデータをソートします。現在の実装ではinputからのデータを一度全部メモリーにため込んでからソートし、その結果を一度にoutputに流します。

ソートするデータがすべて数値であれば自然にソートできますが、例えば各データが配列で、そのn番目の要素でソートしたいというようなニーズもあることでしょう。その場合には関数を指定します。

つまり、配列の1番目の要素（インデックスは0から始まるので実際には2番目）でソートするためには、

```
sort{x,y->cmp(x(1),y(1))}
```

のように比較関数を指定します。

ソートの応用

データのソートができれば、統計的に意味のある値を取ることができます。一番分かりやすいのは順位でしょう。順位を得るということは成績順にソートすることと同じことです。

また、ソートすれば「中央値」を取することもできます。これはソートしたデータにおける中央の要素の値です。データ総数が偶数のときはちょうど中央の値がないので、中央に近い二つの値の平均を中央値とします。中央値はmedian()で求めます。

中央値は平均値に似ていますが、「外れ値」による影響が小さい点が優れています。平均はすべ

*1 しかし、英語版ウィキペディアでは外部マージソートをin-placeで行うことにより、必要なディスク容量を元のデータと同程度に抑えることができると書いてあります。ただし、その部分には[要出典]マークが付けられていますが。

ての値の影響を受けるので、測定ミスなどで外れ値（ほかの値より著しく異なる値）があった場合、誤差が大きくなってしまう危険性があります。しかし中央値は外れ値にはほとんど影響を受けません。

サンプリング

「ビッグデータ」という言葉が流行していますが、実際にあまりに規模の大きなデータを扱うのは困難です。データの処理コストもそうですが、そもそもデータを集めることさえ難しい場合もしばしばあります。

元々「統計」という学問は、実データを集めることが困難な「ビッグデータ」を推計するための手段として誕生しました。

母集団がある程度大きいと意外なほど少ないサンプルで全体の傾向が把握できます。上下5%に誤差を許容するならば、母集団が10万人であった場合、傾向を把握するために必要なのは、わずか383人です。

Streamでもデータを扱いやすくするためにサンプリングをする関数を実装しましょう。サンプリングのためのストリーミングアルゴリズムとしては「レザボアサンプリング (Reservoir Sampling)」があります。

レザボアサンプリングは以下のような手順でサンプリングをします。N個のサンプルを取るためには、

1. 最初のN個のサンプルを配列に登録する
2. それ以降のi番目の要素に対して、0からi-1までの乱数rを生成する
3. 乱数rがNよりも小さいとき、テーブルのr番目をi番目の要素に置き換える

「レザボア (reservoir)」とは貯水池のことです。最初のN個の要素で貯水池をいっぱいにし、上流から要素が流れてくるときにcounter/sizeの確率でランダムに一部を置き換えていきます。貯水池の要素が新しい要素に置き換わっていくことで、最終的に母集団全体からサンプルを取ることができます。

言葉で説明するよりもコードで見た方が分かりやすい人のために、図10にレザボアサンプリングをRubyで記述したものを記述します。

このアルゴリズムを利用してサンプリングをするsample()関数を実装しました。

```
def reservoir_sampling(seq, k)
  e = seq.to_enum
  reservoir = e.take(k)
  n = k
  e.each do |item|
    r = rand(n)
    n += 1
    if r < k
      reservoir[r] = item
    end
  end
  return reservoir
end

# 呼び出し
print reservoir_sampling(0..1000000, 10)
```

図10 Rubyによるレザボアサンプリング

例えば、パイプラインに「sample(100)」を挟むとストリーム全体から100個の要素をランダムに選り出して下流に流します。母集団が大き過ぎるときに、全体の傾向を保ったまま要素を絞り込むことができます。

先にも述べたように、5%程度の誤差を許容するならば、大きな母集団であっても驚くほど小さな数のサンプルから傾向を知ることができます。とはいえ、母集団の大きさが分からない段階であまり絞りこむと正しい結果が得られないことは注意する必要があります。

まとめ

「ビッグデータ」が注目される現代、統計はますます重要になることでしょう。Stream がいつか Excel くらい簡単に使える統計分析のツールにまで成長するとよいなと思います。

タイムマシンコラム

苦手な分野は人に任せたいが…

2016年7月号掲載分です。いくつかの統計関数を導入しました。

本文中でも告白しましたが、私は数学に対して非常に苦手意識を持っています。ですから、今回の原稿の執筆も大変苦労しました。小学生の娘に教科書を借りてまで復習しながら、泣きそうな気持ちで執筆しました。

コンカレントプログラミングといい数学処理といい、自分が苦労したくないからよくできたツールが欲しいのです。しかし残念ながら私の欲しいツールが存在しないので、自分で作るしかなく、自分で作るためにはまさに苦手な問題に真正面から対峙しないといけないという矛盾があります。多分、そういうのが得意な人とチームを組みながら開発できるとよいのですが、なかなかそんな風に補い合える仲間には簡単に巡り会えないものです。私がコミュ障なのがいけないのでしょうか。

5-6 乱数

サイコロで得られるような乱数は、ゲームなどで頻繁に使われますが、Streamが扱うようなデータ処理でも活躍します。今回は乱数の実装と応用について基礎的な部分を学習します。

乱数とは規則性がなく、どんな数が得られるか分からない（ランダムな）数のことです。例えばサイコロを転がすと1から6までの整数が得られますが、事前に何が出るのかは分かりません。実際のサイコロはどの面も平等に上になるように、穴の深さが慎重に調整してあるそうです。ですから、サイコロを十分に大きな回数転がすと、すべての面について上になる確率が等しくなるように作られています。もっとも安物のサイコロの場合には、そこまで気を使って作られていないので、確率にズレがあることもあるそうです。

コンピュータでは乱数はいろいろな局面で使われます。例えば、ほとんどのゲームにはなんらかの形で乱数が用いられています。ゲーム以外にも、例えばsshやhttpsの通信における暗号化にも使われています。

データ処理の領域で乱数を利用する代表例としては「モンテカルロ法」が挙げられます。モンテカルロ法は乱数を利用して計算をする手法で、一例としては乱数によって円周率を求められます（図1）。

正方形の中に乱数で座標を決めてたくさんの点を置くと、その中には1/4円内に含まれる点と外側に置かれる点ができることになります。そこで円の中に含まれる点の数を全体の数で割ると、その値はおおむね $\pi/4$ になり、点の数が多くなればなるほど正確になります。これがモンテカルロ法による円周率の求め方です。

モンテカルロ法のほかにも、5-5で紹介した「レザボアサンプリング (Reservoir Sampling)」では乱数を用いて多数のデータから偏りなくサンプリングをします。

また、乱数を利用して効率を高める「乱択アルゴリズム」のようなものもあります。乱択アルゴリズムには、誤差を許容することによって速度を稼ぐ「ブルームフィルター」のようなものがあります。

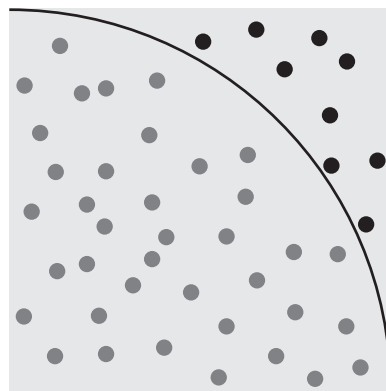


図1 モンテカルロ法の例

真の乱数と擬似乱数

そもそもコンピュータではサイコロを転がすような乱数（真の乱数）を得る方法はありません。しかし真の乱数でなくても、なんらかの計算によって「乱数っぽい数」を得ることはできます。

一番簡単な方法としては、時刻情報を使う方法があります。現在の秒の値や、あるいは利用できる時刻情報によってはマイクロ秒やナノ秒単位の時刻の情報を利用して、ランダムな数値を得るわけです。昔々のマイコンZ80では、メモリーをリフレッシュするタイミング情報を保持していたR(リフレッシュ)レジスタの値(1～127)を乱数として利用していたと聞いています。

擬似乱数は計算によっても得られます。擬似乱数を得るアルゴリズムはいくつも知られていますが、代表的なものには以下のようなものがあります。

- 線形合同法
- メルセンヌツイスター
- Xorshift

計算によって求める擬似乱数の特徴は、再現性があることです。つまり、同じ初期値から計算を始めると、全く同じ乱数列が得られるのです。

再現性があるということは、乱数の「規則性がなく、どんな数を得られるか分からない数」という性質とは矛盾しています。しかし、これはこれで便利な場合があるのです。例えば、乱数を使ってなんらかのシミュレーションをしたとしましょう。擬似乱数を用いて、同じ初期値から同じ乱数列が得られるということは、全く同じシミュレーション結果を再現できるということでもあります。シミュレーション結果を追試する場合などでは、この再現性がとても「ありがたいこと」があり得ます。

擬似乱数の評価

擬似乱数列を生成するアルゴリズムが複数考案されていることはすでに述べました。それらのアルゴリズムがどのように異なっているのか、どのように評価すればよいのでしょうか。

擬似暗号アルゴリズムの評価基準には、以下のようなものがあります。

- 偏り
- 周期
- 速度(計算量)
- 暗号論的安全性

「偏り」は、そのアルゴリズムがどのくらい「真の乱数」から離れているかを示します。アルゴリズムによっては生成される乱数に偏りが生じて、例えばある特定の数値の倍数が登場しやすいなど

の現象が発生することがあります。

計算によって求められる擬似乱数列は、いずれ同じパターンを繰り返すことになりがちです。その場合、同じパターンが生じるまでの長さが「周期」です。周期が短い擬似乱数アルゴリズムは、次の値が予想しやすかったり、偏りが大きくなったりします。

偏りと周期はその暗号アルゴリズムに固有の性質です。個別の暗号アルゴリズムがなぜそのような性質を持つかは数学的に決まるのですが、それを解説するのは、正直なところ私の手に余ります。ここでは「そういうものだ」と理解してください。

「速度」は、次の乱数を計算するのに必要な計算量を意味します。統計やシミュレーションなどの分野では大量の乱数を必要とします。その場合、あまり計算量の多い擬似乱数アルゴリズムでは、暗号を求める時間が処理全体に対するボトルネックになってしまいます。

「暗号論的安全性」はそのアルゴリズムを暗号の分野に利用しても安全かどうかを示すものです。暗号の分野では鍵生成やワンタイムパッドなど乱数を必要とする場面が数多くあります。ここで安易な擬似乱数を用いると、そこが穴になって暗号が破られてしまう危険性があります。

暗号論的安全性を実現するためには、生成される乱数が偏りのないだけでなく、途中の状態がある程度明らかになっても、暗号が破られない性質を持つ必要があります。これはかなり難しい条件ですが、それでも例えば、「Blum-Blum-Shub」のような暗号論的安全な擬似乱数アルゴリズムは存在します。また、通常の擬似乱数アルゴリズムで生成した乱数に暗号で用いられる「一方向関数」を適用することで、暗号論的に安全にできます。しかし、いずれにしても安全性の実現のためにそれなりのコストがかかるので、通常の（例えば統計のような）利用方法にはオーバースペックになります。

今回紹介する擬似乱数アルゴリズムは、いずれも（そのままでは）暗号論的には安全ではありません。ここでは、暗号目的に乱数を使う場合に、安易に通常の擬似乱数アルゴリズムを使うのは危険であることだけ覚えておいてください。

線形合同法

では、（暗号論的に安全ではない）擬似乱数アルゴリズムのうち、代表的なものについて解説しましょう。

最初に取り上げるのは、広く実用的に使われた擬似乱数アルゴリズムとしては最古ではないかといわれる「線形合同法」です。線形合同法による乱数列は、図2の漸化式によって定義されます。

$$X_{n+1} = (A \times X_n + B) \bmod M$$

図2 線形合同法の漸化式

ただし、A、B、Mは定数で、 $M > A$ 、 $M > B$ 、 $A > 0$ 、 $B \geq 0$ になります。この定数と初期値 X_0 の選び方によって、乱数列の性質が決まります。

線形合同法の周期は最大でもMになります。しかし、その性質は定数のとり方で左右されるので、不適切な定数を選ぶとMよりもずっと短い周期になったり、偏りが大きくなったりします。

数式があまり得意でない私のような人のために、Cによる線形合同法を用いた乱数列発生プログラムを図3に示します。このプログラムでは、定数として

```
A=1566083941
B=1
M=232
```

```
uint32_t
rand(void)
{
    static uint32_t seed = 1;
    seed = seed * 1566083941UL + 1;
    return seed;
}
```

図3 Cによる線形合同法を用いた乱数列発生プログラム

を選択しています。これは比較的马ジな定数の組み合わせだといわれています。

線形合同法はさほど計算量の多いアルゴリズムではありませんが、乱数として優れているとはいえません。特に注意すべき点は、下位ビットのランダム性が低い点です。例えば、線形合同法で得られた32ビット乱数から0から7までの乱数を作りたければ、32ビット乱数をrとして、「r > 29」のように上位ビットを取り出すべきです。「r%8」とか「r&0xf」のような操作で、下位4ビットを取ってはけません。

また線形合同法は周期が比較的に短いのと、漸化式の定義から明らかなようにある乱数が得られるとその次の乱数が一意に決まるという性質があります。そのため例えばモンテカルロ法などに用いると、点が一様にならず格子状になる現象が発生することがあります。ゲームなどならともかく、統計やシミュレーションの分野でこのアルゴリズムを利用するのは注意する必要があります。今では線形合同法よりも性質の良い擬似乱数アルゴリズムがいくつも知られていますから、そちらを利用の方が望ましいといえます。

Cの標準ライブラリは、乱数を得るための関数rand()を提供していますが、多くの場合、線形合同法を利用しています(CのISO規格ではrand()の乱数計算に線形合同法の利用を規定していません。ただ、参考として掲載されているアルゴリズムは線形合同法です)。

つまり、上記の線形合同法の注意点はそのままrand()にも当てはまる可能性が高いです。場合によってはシステムが提供するrand()に頼らず独自に擬似乱数アルゴリズムを用意する必要が出てくるかもしれません。

メルセンヌツイスター

線形合同法に比べて比較的新しい擬似暗号アルゴリズムが「メルセンヌツイスター」(Mersenne Twister)です。メルセンヌツイスターは、1996年に広島大学(当時)の松本眞氏と西村拓士氏が発表しました。

メルセンヌツイスターの最大の特徴はその長い周期性です。線形合同法の解説でも少し触れましたが、周期が短いと大量の乱数を用いたシミュレーションなどで偏りが発生します。メルセンヌツイスターの周期は $2^{19937}-1$ という非常に長いものです。これは10進で表記すると6000桁を超える

大変大きな数です。この $2^{19937}-1$ はメルセンヌ素数と呼ばれるタイプの素数であり、このアルゴリズムの名前の由来にもなっています。

周期が長いだけでなく、連続する乱数の間の相関関係が小さいこと（「高次元に均等分布する」という言い方をするそうです）、またそれ以前の擬似乱数アルゴリズムよりも高速であることなど非常に優れたアルゴリズムです。

実際、その優れた点が評価されて、Ruby や Python をはじめとする多くのプログラミング言語処理系で標準の乱数生成アルゴリズムとして採用されています。

このように優秀なメルセンヌツイスターですが、欠点がないわけではないです。一つは内部的な状態ベクトルが大きいことです。線形合同法では整数一つしか内部状態を持ちませんでしたが、メルセンヌツイスターは623個の32ビット整数を内部状態として保持します。これは、途中の状態を保存して乱数列を再現させる場合に若干扱いが面倒になります。それより何より、この623個の状態ベクトルを初期化するときに、十分な注意を払わないと、乱数の質が低下してしまいます。

これらの点を改善したSFMT (SIMD-oriented Fast Mersenne Twister) というアルゴリズムも存在しています。オリジナルのメルセンヌツイスターよりも2倍速いという触れ込みなのですが、登場したのが2006年と比較的新しいせいか（といってももう10年にもなりますが）、あまり使われているのを見かけません。私自身も今回の原稿のために調べていて初めて知ったくらいです。今後はこちらを利用してみようかと考えています。

Xorshift

メルセンヌツイスターよりもさらに新しい擬似乱数アルゴリズムがXorshiftです。XorshiftはGeorge Marsaglia氏が2003年に発表しました。

Xorshiftはその名の通り、xor（排他的論理和）演算とビットシフト演算だけを使って高速に擬似乱数を得るアルゴリズムです。その結果、高速に乱数を計算することができます。

Xorshiftは、周期の大きさについてメルセンヌツイスターよりも短いものの（状態ベクトルのサイズによって $2^{32}-1$ から $2^{128}-1$ まで変化する）、線形合同法に比べるとはるかに優れたランダム性を示します。それでありながら、実装は非常にシンプルです。

```
uint32_t xorshift(void) {
    static uint64_t x = 88172645463325252ULL;
    x = x ^ (x << 13); x = x ^ (x >> 7);
    return x = x ^ (x << 17);
}
```

図4 Xorshiftの実装

Xorshiftの最もシンプルな実装（状態ベクトルが64ビットのもの）を図4に示します。図4のコードによる乱数の周期は $2^{64}-1$ です。

こんなにシンプルな計算で優れた乱数を生成することができ、また、その方法が21世紀になるまで見つかっていなかったというのは驚きです。

Xorshiftには、よりランダム性を高める派生系であるXorshift*とXorshift+があります。

擬似乱数の初期値

さて、ここまで解説してきた通り、擬似乱数アルゴリズムは計算によってランダム性が高い（ように見える）数列を生成するものです。しかし、あくまで計算結果によるものですから、真の乱数ではありません。

コンピュータは基本的に決定的に動作する、つまり同じ状態から始めると同じ結果が得られるものですから、真の乱数を扱うことは困難です。

擬似乱数アルゴリズムの初期値として、できるだけ予測しづらい値を導入することで、擬似乱数をより乱数らしく扱うことが可能になります。

典型的な初期値には時刻が使われます。OSから取得した現在時刻の小さな部分（マイクロ秒単位やナノ秒単位）を利用することにより、実行ごとに異なるランダムな初期値を得ることができます。

ただ注意しなければならないことは、ハードウェアが実際持っている時刻機能は、OSが提供するマイクロ秒やナノ秒単位の分解能を持っていないことです。マイクロ秒単位の時刻を返すシステムコールがあっても、その時刻が本当にマイクロ秒単位で正確であるというわけではないことに注意してください。

このように当てにならない時刻よりも、OSがよりランダムな数値を得る方法を提供している場合があります。

`/dev/random`

外部から与えられるユーザーからの入力などは、基本的に予測不可能ですから、乱数の基になる「エントロピー（乱雑さ）」を備えています。

例えば、Linuxではドライバなどからこの外部情報に基づいたエントロピーを集めています。そして「`/dev/random`」というデバイスファイルを読み出すことで、集めたエントロピーを消費して「真の乱数」を得ることができます。

しかし、デバイスなどから集められるエントロピーには限りがあります。このため、あまりたくさんの乱数を`/dev/random`から読み出すと集めたエントロピーを使い切ってしまうます。`/dev/random`はそのような場合、読み出しをブロックして、エントロピーがたまるまで待ちます。

ただ単に乱数情報が欲しいだけなのに、ブロックしてしまっは困る場合もあるので、別のデバイスファイル「`/dev/urandom`」というものもあって、こちらはブロックしません。`/dev/urandom`はエントロピーを使い切った場合、過去のエントロピーを初期値として暗号論的に安全な擬似乱数アルゴリズムを使って乱数値を返します。

ほかのOSでも似たような機能を提供しています。例えばFreeBSDも、「`/dev/random`」を提供しています。乱数を返すという挙動は共通ですが、こちらは最初から（暗号論的に安全な）擬似乱数アルゴリズムを使っており、ブロックしません。そういう意味では、FreeBSDの`/dev/random`

はLinuxの/dev/urandom相当だといえるでしょう。

/dev/randomから得られる乱数は、時刻よりも予想しづらく、良い初期値となります。実際、Rubyでは（利用可能であれば）/dev/urandomを使って乱数列を初期化しています。

一方、mrubyやStreamでは乱数の初期化に時刻情報を使っています。/dev/urandomはLinuxなど一部のOSでしか利用できないため、移植性のことも考えてこうなっています。ただ、あまり移植性を考慮していないStreamでは、時刻よりも/dev/urandomを使った方がよいのかもしれない。

では、乱数生成にはいつもデバイスファイルから読み込めばよいと思う人もいるかもしれませんが。しかし、これらは性能の点から擬似乱数アルゴリズムの代替にはなりません。

擬似乱数のベンチマーク

プログラマとしては、実際にコードを書いて比較したいところです。そこで今回紹介した擬似乱数アルゴリズムをそれぞれ評価してみましょう。といっても、偏りや周期などは（実装に間違いさえなければ）理論的に定まるので、今回はとりあえず性能について測定します。

まず、今回紹介した線形合同法、メルセンヌツイスター、Xorshiftの3種類のアルゴリズムを使って1億個の乱数を生成し、その実行時間を比較します。

実際のベンチマークプログラムを図5に、その出力結果を図6に示します。測定に用いたマシンのスペックは表1の通りです。

```
#include <stdio.h>
#include <inttypes.h>
#include <sys/time.h>

/* linear congruential method */
uint32_t
lcm_rand(void)
{
    static uint32_t seed = 1;
    seed = seed * 1566083941UL + 1;
    return seed;
}

/* merseene twister */
#define N 624
#define M 397
#define M 397
#define MATRIX_A 0x9908b0dfUL /* constant vector a */
```

```

#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static uint32_t mt[N]; /* the array for the state vector */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

void
mtw_init(uint32_t s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] = (1812433253UL*(mt[mti-1]^(mt[mti-1]>>30))+mti);
        mt[mti] &= 0xffffffffUL;
    }
}

uint32_t
mtw_rand(void)
{
    uint32_t y;
    static const uint32_t mag01[2]={0x0UL, MATRIX_A};

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1) /* if mtw_init() has not been called, */
            mtw_init(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
        mti = 0;
    }

    y = mt[mti++];

```

```

/* Tempering */
y ^= (y >> 11);
y ^= (y << 7) & 0x9d2c5680UL;
y ^= (y << 15) & 0xefc60000UL;
y ^= (y >> 18);

return y;
}

uint32_t
xor_rand(void) {
    static uint64_t x = 88172645463325252ULL;
    x = x ^ (x << 13); x = x ^ (x >> 7);
    return x = x ^ (x << 17);
}

#define TIMES 100000000
#define BENCH(name) do {\
    struct timeval tv, tv2, tv3;\
    int i;\
    char *f;\
    name ## _rand(); /* rehearsal */\
    f = #name;\
    gettimeofday(&tv, NULL);\
    for (i=0; i<TIMES; i++) {\
        name ## _rand();\
    }\
    gettimeofday(&tv2, NULL);\
    timersub(&tv2, &tv, &tv3);\
    printf("func %s: %ld.%06ldsec\n", f, tv3.tv_sec, tv3.tv_usec);\
} while (0)

int
main()
{
    printf("benchmark repeats %d times\n", TIMES);
    BENCH(lcm);
    BENCH(mtw);
    BENCH(xor);
}

```

図5 乱数生成ベンチマークプログラム

```
benchmark repeats 100000000 times  
func lcm: 0.305532sec ← 線形合同法  
func mtw: 0.963983sec ← メルセンヌツイスター  
func xor: 0.733444sec ← Xorshift
```

図6 乱数生成ベンチマーク結果

機種	Thinkpad E450
CPU	Core i7-5500U
CLOCK	2.40GHz
MEM	16GB
OS	Ubuntu 16.04
GCC	gcc 5.4.0

表1 ベンチマークマシンのスペック

このベンチマークを見る限り、最も高速なのが線形合同法で1億回乱数を生成するのに約0.3秒、次に高速なのがXorshiftで約0.73秒、最も時間がかかったのがメルセンヌツイスターで約0.96秒でした。

ただし、線形合同法は乱数の品質から論外であることを考えると、周期の長さや性能のトレードオフでメルセンヌツイスターとXorshiftを使い分けるのがよいのかもしれません。あるいは2倍高速という触れ込みのSFMTの採用を検討すべきかもしれません。

実際、SFMTもベンチマークを取ろうと考えたのですが、SSE2などのSIMD命令を利用することもあって、ソースコードが意外に複雑でした。このため、時間切れで今回のベンチマークプログラムに追加できるほど簡潔な形でまとめられませんでした。大変残念です。

Xorshiftはメルセンヌツイスターの2倍も高速ではないので、SFMTの性能が額面通りであれば、長周期と性能を両立させることができるわけで最強ではないでしょうか。現時点では私も理解が及ばず使いこなせていないので、今後も研究を続けたいと思います。

Streamの乱数機能

現時点でのStreamの乱数に関連する機能は、乱数のストリームを生成するrand()関数と、乱数を用いてストリームのサンプリングをするsample()関数の二つです。

rand()関数は乱数を一つずつ渡すストリームです。例えば、

```
rand() | stdout
```

を実行すると、(割り込みをかけて中断するまで)ずっと乱数を表示し続けます。

sample()関数は上流のストリームの要素を引数で指定した数だけサンプリングします。

```
fread("data.csv") | sample(100) | stdout
```

とすると、data.csvに含まれている行から100行だけサンプリングして標準出力に表示します。data.csvがたとえ何万行であろうとも均等にサンプリングしてくれるのがsample()関数の、つまりはそれが用いている「レザボアサンプリング」の魅力です。

これらの関数は擬似乱数アルゴリズムとしてXorshiftの修正版Xorshift64*（周期 $2^{64}-1$ ）を採用しています。オリジナルのXorshiftとの違いは、乱数生成の最後の過程に乗算を用いている点です。これにより若干計算速度は落ちますが、ランダム性は向上し、Die Hardテストという擬似乱数アルゴリズムのテストにすべて合格するようになります。

ただ、現在のXorshift64*はどうにも周期が短いので、実際の統計処理などに使うのには品質が不足するかもしれません。今後、メルセンヌツイスターやSFMTに置き換えることも検討する必要がありますでしょう。

乱数のさまざまな種類

これまで解説してきた乱数は一定の範囲内の数が均等な確率で登場する「一様乱数」でした。しかし、統計やシミュレーションに必要とされる乱数は、一様乱数ばかりではありません。

統計解析を主なターゲットとする言語であるRには乱数生成関数がたくさん用意されています（表2）。

命令ルールは、「r」＋分布の名前（多くは省略形）になっています。例えば一様乱数は一様分布（Uniform Distribution）する乱数ですからrunifですし、正規分布（Normal Distribution）する乱数はrnormになります。

Streamではrand()がrunif相当です。ほかの乱数生成関数は必要になってから実装すればよいと考えていますが、使い道が明確でシミュレーションなどにすぐに使いそうな標準正規乱数だけは、早々に作ろうと考えています。名前が悩ましいところですがNormal Distributionから取ってnrnd()か、rand_normal()あたりはどうでしょうか。前者の方がコンパクトですが、今後さまざまな分布を取る乱数生成関数を提供することを考えると、あまり省略し過ぎない方がよいかもしれません（結局rand_norm()にしました）。

名前	意味	解説
runif	一様乱数	いわゆる乱数
rnorm	標準正規乱数	正規分布する乱数
rbinom	二項乱数	二項分布する乱数
rpois	ポアソン乱数	ポアソン分布する乱数
rexp	指数乱数	指数分布する乱数
rgamma	ガンマ乱数	ガンマ分布する乱数

表2 Rの乱数生成関数

まとめ

今回は乱数を生成する擬似乱数アルゴリズムとその応用について解説しました。特にStreamがターゲットにするようなデータ処理には乱数は重要な役割を果たします。本記事が実際にお手元に届くまでには、Streamの乱数機能の実装も進めておくつもりです。

新しいアルゴリズムに挑戦した

2016年10月号掲載分です。乱数生成についての解説ですが、5-5の統計基礎に引き続き数学っぽい話題で泣きそうです。一応、各種アルゴリズムとその評価方法について頑張って解説しました。

伝統的な乱数生成アルゴリズムとしては線形合同法、比較的新しいアルゴリズムとしてはメルセンヌツイスターが有名ですが、新しいことを試してみたいという気持ちから今回はXorshiftを採用してみました。比較的シンプルで高速で乱数の品質もよいというXorshiftですが、派生がたくさんある上にネットの情報がやや錯綜しているのが困りました。数学力の低さから、自力で正しさを検証できないので、どの情報を信じたらよいのか困りました。

とりあえず、自分が正しいと信じる実装を用意したのですが、これが本当に正しいのか実は確信が持てないでいるのです。

5-7 ストリームグラフ

今回は、ストリーム処理をする言語であるStreamにふさわしい、ストリーム入力からグラフを出力する手続きについて解説します。GUIライブラリはやはり廃りが激しいので、確実に長く使えるCUIベースでグラフを出力しましょう。見た目を改善していく方法も紹介します。

エンジニアの多くはそうでないかと信じているのですが、私はなんでも測定するのが大好きです。毎晩体重計に乗りますし、病気で熱が出たときには15分くらいごとに体温を測って変化の傾向を調べようとします。また、PCの画面にもメモリー消費量、CPU使用率、ネットワーク転送量などのグラフを表示させています。宇宙戦艦ヤマトの艦内にやたらとメーターがあったのも、似たような心理ではないかとにらんでいます。

ですから、Streamにも入力されたデータをグラフ化できる機能があるといいなと考えていました。しかし、グラフィックスはなかなか扱いが面倒です。GUIはプラットフォームごとにAPIが異なります。なんらかのGUIライブラリを利用するにしても、その解説だけで誌面が尽きてしまうでしょう。

それにGUIライブラリの寿命は一般にプログラミング言語よりかはるかに短いです。Rubyの初期に大変人気のあったGUIライブラリであるTkが、今やほとんど見かけなくなったのを見ても明らかでしょう。

そこで、GUIのような見栄えに関する部分は後回しにして、グラフを表示するという本質だけについて解説することにしましょう。

GUIとCUI(とCLI)

CUIはGUI(Graphical User Interface)と対になっている用語で、Character User Interfaceの略です。もっとも海外ではCUIという言葉を書くことはもうほとんどないので、日本独自の用語になってしまっているようです。海外ではどちらかというと、CLI(Command Line Interface)という言葉の方をよく聞くようです。

CUIはキャラクターで画面表示を行うので、もう何十年も使われてきているターミナルの中で動作できます。今後もターミナルがなくなることは心配しなくてもよさそうです。今回はこちらのCUIを使ってグラフを表示する機能を作ります。

stag

そこで参考にできるツールはないかといろいろと探したところ、stag^{*1}というツールを見つけました。

stagは標準入力から数値データを読み込み、それに対して棒グラフを出力するツールです。例えば、図1のようにStreamで生成した乱数列からグラフを出力できます。出力結果は図2のようになります。

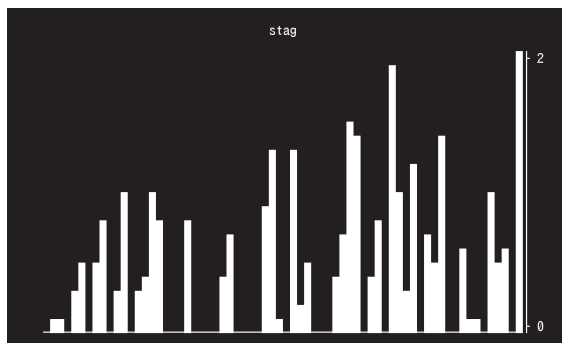


図2 stagを使った乱数グラフ出力結果

```
stream -e 'rand_norm()|take(100)|stdout' | stag
```

図1 stagを使った乱数グラフ出力

これはこれで便利なツールですが、Streamからデータをほかのプロセスに送ることなく、直接グラフ化できればもっと便利だと思います。そこで今回はstagと同じような働きをする関数をStreamに追加しましょう。

画面構成

まず、stagの表示は画面を図3のように分割しています。

これをキャラクターで出力すればよいわけです。幸い、ほとんどのターミナルではエスケープシーケンスを使って、出力する文字に色を付けたり、カーソルを移動させたりできます。その機能を使えば、画面の部分書き換えもできます。

stagはCUIのためのライブラリncursesを使っていますが、今回の開発ではそれほど多くの機能を必要としないため、エスケープシーケンスを直接利用して描画することになります。



図3 stagの画面分割

エスケープシーケンス

最近では、ターミナルという言葉聞いても、「シェルのコマンドを入力するウィンドウね」くらいの印象しかありません。しかし昔々はターミナル（端末）といえば、コンピュータの入出力をするための機械でした。現在、私たちが使っている「ターミナル」はそのターミナルという機械の機能をソ

*1 <https://github.com/seenaburns/stag>

フトウェアで実現するものであり、正確にはターミナルエミュレーターと呼ぶべきものです。

機械としてのターミナルが現役だった頃、大きな（といっても現在の視点からはとても貧弱な）コンピュータにいくつかのターミナルをつなげて操作するのが当たり前でした。パーソナルなコンピュータが登場するよりも前の時代のことです。

この時代のターミナルは、テキストを表示する画面と入力をするキーボードが付いているだけで、コンピュータとしての処理能力はありませんでした。入力された情報をすべてコンピュータ（当時はホストと呼ばれました）へ送り、戻ってきた情報を画面に表示する、それだけの機能しかありませんでした。

しかし、それではあまりにも表示がさみしいので、次第にターミナルに対して、エスケープ文字に続いていくつかの文字を送ることで、画面を操作するさまざまな機能呼び出すことができるように進化しました。このような文字列をエスケープシーケンスと呼びます。例えば、

```
ESC [ 2 J
```

で画面を消去できます。

これらのエスケープシーケンスは当初端末のメーカーや機種ごとに異なっていました。しかし当時、大変普及したDECのVT100という機種のエスケープシーケンスがおおむねデファクトスタンダードになりました。

今回グラフ機能の実装で使うエスケープシーケンスを表1に示します。

シーケンス	意味
ESC [x; y H	(x, y) へカーソル移動
ESC [2 J	画面クリア
ESC [1 K	カーソルより前の行クリア
ESC [3x m	文字色指定 (黒赤緑黄青紫水白)
ESC [0 m	色リセット
ESC [6 n	カーソル位置の取得
ESC [? 25 l	カーソルの消去
ESC [? 25 h	カーソルの表示

表1 代表的なエスケープシーケンス

ウィンドウサイズ取得

最初に、現在のターミナルウィンドウのサイズを取得します。ウィンドウサイズを取得する方法はいろいろありますが、今回は、`ioctl`を使います。

`ioctl`はファイルディスクリプターを対象に入出力を制御するためのシステムコールです。

```
ioctl(fd, request, ...);
```

という形式で呼び出し、`request`に応じた制御をカーネルが実施します。同じ`request`に対してもディスクリプターの先にいるデバイスによって処理が異なるかもしれませんが、そこは気にする必要はありません。ある種のオブジェクト指向だと考えることもできるでしょう。このファイルディスクリプターを通じて、さまざまなオブジェクトをオブジェクト指向的に扱えるのがUNIXの良いところであり、登場時には大変斬新だったところです。

`ioctl`でカーネルにウィンドウサイズを問い合わせるリクエストは、`TIOCSWINSZ`です。引数とし

てはウィンドウサイズを保持する構造体 `struct winsize` のポインタを渡します (図4)。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <termios.h>

static int
get_winsize(int* row, int* col)
{
    struct winsize w;
    int n;

    n = ioctl(1, TIOCGWINSZ, &w);
    if (n < 0 || w.ws_col == 0) {
        return -1;
    }
    *row = w.ws_row;
    *col = w.ws_col;
    return 0;
}

int
main()
{
    int row, col;
    int n;

    n = get_winsize(&row, &col);
    if (n < 0 || col == 0) {
        printf("WINSZ failed\n");
        exit(1);
    }
    printf("WINSZ (%d, %d)\n", w.ws_col, w.ws_row);
}
```

図4 ioctlによるウィンドウサイズの取得

カーソルの移動

カーソルの移動にはエスケープシーケンスを用います。出力先のファイルディスクリプターがターミナルに向いていれば、エスケープシーケンスを送ることで、カーソルを自由に移動できます。座標 (x,y) の位置にカーソルを移動させるためには、

```
ESC [ x; y H
```

というエスケープシーケンスを送ります。移動は左上隅が原点になりますが、原点の位置が(0, 0)ではなく、(1, 1)であることに注意が必要です。

図5のプログラムは画面消去後、カーソルを移動させながらHello Worldを出力するプログラムです。実行すると図6のような出力になります。

画面サイズが取得でき、画面消去とカーソル移動ができれば、後はそれらの組み合わせでグラフを出力できるようになります。

タイトル 描画

タイトルの描画は簡単です。タイトルが指定されていれば、画面の先頭行に移動し、タイトルを描画します。画面中央にタイトルを描画するために、画面サイズと文字列長さを考慮してカーソルを移動させる必要があります。

タイトル表示のプログラム（のmain部分）を図7に示します。図7のプログラムは単体ではコンパイルできず図4と図5のプログラムで定義されている関数を利用していますが、関数の使い方は理解できるでしょう。

グラフ 描画

グラフの描画は比較的簡単です。グラフデータ、その最大値、ウィンドウサイズから、各行に先頭から1文字ずつ、その位置（カラム）にグラフが表示されるかどうかを判定し、グラフが表示される場合には色を指定します。各行の右端にはy座標を表示させます。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void
clear()
{
    printf("\x1b[2J");
}

static void
move_cursor(int row, int col)
{
    printf("\x1b[%d;%dH", row, col);
}

int
main()
{
    int i;

    clear();
    for (i=1; i<10; i++) {
        move_cursor(i, i*2);
        printf("%d:Hello World\n", i);
    }
}
```

図5 カーソル移動サンプル

```
1:Hello World
2:Hello World
3:Hello World
4:Hello World
5:Hello World
6:Hello World
7:Hello World
8:Hello World
9:Hello World
```

図6 カーソル移動サンプル出力

```
int
main(int argc, char **argv)
{
    int i, row, col;
    char* title;
    int tlen;
    int start;

    if (argc != 2) exit(1);
    title = argv[1];
    tlen = strlen(title);
    // 画面サイズ取得
    if (getwinsize(&row, &col) < 0) exit(1);
    start = (col - tlen) / 2;
    clear();                // 画面消去
    move_cursor(start, 1);  // カーソル移動
    write(1, title, tlen);  // タイトル表示
    move_cursor(row-1, 1);  // 最終行に移動
    return 0;
}
```

図7 タイトル表示

graph_bar()関数

さて、これまで解説してきたことを組み合わせたgraph_bar()関数は図8のようになります。図8で利用しているget_winsize()、move_cursor()、clear()などの関数は図4と図5で定義されているものと同じです。

処理手順としては、まず初期化として

- 画面サイズ取得
- 画面クリア
- タイトル描画

をします。そして上流から数値データを受け取るたびに、

- データの格納
- 最大値の計算
- グラフ描画

をします。グラフの描画では

- y 軸の描画
- 左端から棒グラフの表示

をします。まあ、エスケープシーケンスを正しく使えばそんなに難しいことはありませんね。

```
struct bar_data {
    const char *title;
    strn_int tlen;
    strn_int col, row;
    strn_int dlen, llen;
    strn_int offset;
    strn_int max;
    double* data;
};

static void
show_title(struct bar_data* d)
{
    int start;

    clear();
    if (d->tlen == 0) return;
    start = (d->col - d->tlen) / 2;
    move_cursor(1, start);
    fwrite(d->title, d->tlen, 1, stdout);
}

static void
show_yaxis(struct bar_data* d)
{
    move_cursor(1,2);
    printf("\x1b[0m");    /* 色を戻す */
    for (int i=0; i<d->llen; i++) {
        move_cursor(i+3, d->dlen+1);
        if (i == 0) {
            printf("├ %d    ", d->max);
        }
        else if (i == d->llen-1) {
            printf("├ 0");
        }
        else {
```

```

        printf(" | ");
    }
}

static void
show_bar(struct bar_data* d, int i, int n) {
    double f = d->data[i] / d->max * d->llen;

    for (int line=0; line<d->llen; line++) {
        move_cursor(d->llen+2-line, n);
        if (line < f) {
            printf("\x1b[7m "); /* 色反転 */
        }
        else if (line == 0) {
            printf("\x1b[0m-"); /* 色を戻してベースライン描画 */
        }
        else {
            printf("\x1b[0m "); /* 色を戻して空白描画 */
        }
    }
}

static void
show_graph(struct bar_data* d)
{
    int n = 1;

    show_yaxis(d);
    for (int i=d->offset; i<d->dlen; i++) {
        show_bar(d, i, n++);
    }
    for (int i=0; i<d->offset; i++) {
        show_bar(d, i, n++);
    }
}

static int
init_bar(struct bar_data* d)
{
    if (getwinsize(&d->row, &d->col))
        return STRM_NG;
    d->max = 1;
}

```

```

    d->offset = 0;
    d->dlen = d->col-6;
    d->llen = d->row-5;
    d->data = malloc((d->dlen)*sizeof(double));
    for (int i=0;i<d->dlen;i++) {
        d->data[i] = 0;
    }
    show_title(d);
    return STRM_OK;
}

static int
iter_bar(strm_stream* strm, strm_value data) {
    struct bar_data* d = strm->data;
    double f, max = 1.0;

    if (!strm_number_p(data)) {
        strm_raise(strm, "invalid data");
        return STRM_NG;
    }

    f = strm_value_float(data);
    if (f < 0) f = 0;
    d->data[d->offset++] = f;
    max = 1.0;
    for (int i=0; i<d->dlen; i++) {
        f = d->data[i];
        if (f > max) max = f;
    }
    d->max = max;
    if (d->offset == d->dlen) {
        d->offset = 0;
    }
    show_graph(d);
    return STRM_OK;
}

static int
fin_bar(strm_stream* strm, strm_value data) {
    struct bar_data* d = strm->data;

    move_cursor(d->row-2, 1);
    if (d->title) free((void*)d->title);

```

```

    free(d->data);
    free(d);
    return STRM_OK;
}

static int
exec_bar(strm_stream* strm, int argc, strm_value* args,
        strm_value* ret) {
    struct bar_data* d;
    char* title = NULL;
    strm_int tlen = 0;

    strm_get_args(strm, argc, args, "ls", &title, &tlen);
    d = malloc(sizeof(struct bar_data));
    if (!d) return STRM_NG;
    d->title = malloc(tlen);
    memcpy((void*)d->title, title, tlen);
    d->tlen = tlen;
    if (init_bar(d) == STRM_NG) return STRM_NG;
    *ret = strm_stream_value(strm_stream_new(strm_consume
        r, iter_bar, fin_bar, (void*)d));
    return STRM_OK;
}

```

図8 graph_bar関数の実装

グラフ記述で工夫した点は、グラフ表示用のバッファの使い方です。bar_data構造体には表示できるグラフの数だけデータを保持するdataという配列があり、offsetで指定される位置に入力されたデータを書き込みます。offsetは、配列の長さdlenを超えた時点で0に戻されますから、これは一種のリングバッファです。

表示時にはoffsetから始めてバッファの終わりまでデータを表示して、それから今度は先頭からoffsetの直前までのデータを表示します。これにより、入力された時系列順に左からグラフとして表示することになります。

グラフの描画はカーソルを移動しながら、値がある高さまでスペースを反転色で表示し、残りは通常色のスペースで埋めます。

グラフが書けるようになったら、

```
seq(100)|graph_bar()
```

とか (図9)、

```
rand_norm()|take(100)|graph_bar()
```

とか (図10) を表示させてみましょう。



図9 seq(100)|graph_bar()の表示結果

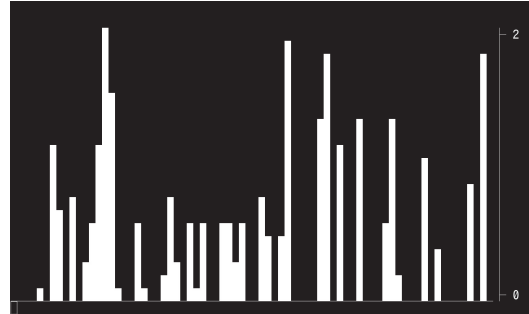


図10 rand_norm()|take(100)|graph_bar()の表示結果

ウィンドウサイズの調整

昔の機械端末であれば画面サイズは変化しないでしょうが、現代のターミナルエミュレーターでは、ウィンドウのリサイズにより画面サイズが変化することがあり得るでしょう。

CUIでもウィンドウサイズ変更に対応することができます。今回はソースコードをコンパクトにするために実際にはウィンドウサイズの変更には対応しませんが、やり方については解説しておきます。

出力しているターミナルのサイズが変わると、プロセスにSIGWINCHというシグナルが送られます。シグナルが送られたら、グラフの初期化をやり直して、再描画すればよいわけです。

シグナルを受け付けるにはsingal関数を使います。

```
signal(シグナル番号, ハンドラー関数)
```

で指定すると、シグナルが届いたときにハンドラー関数で指定した割り込みハンドラーが呼び出されます。例えば

```
signal(SIGWINCH, winch_handler);
```

のような感じですね。割り込みはいつどのようなタイミングで起きるか分からないので、割り込みハンドラー中で実行できない処理もあります。そこで、ハンドラーの中では変数の代入だけにしておいて、メインルーチンで割り込みの有無をチェックするのが定石です。

しかし、Stroomに組み込む関数として考えるとちょっと難しい点もあります。一つのプロセスの中では一つのシグナルに対して一つのハンドラー関数しか登録できません。このためプログラムの別の場所でシグナルを使おうと思うと、いずれかのハンドラーが呼び出されなくなってしまいます。

別の考え方としては、どうせ大したコストではないので、毎回ウィンドウサイズを取得し、今までのサイズと異なっていたら、初期化をやり直すというアイデアもあり得るでしょう。こちらはシグナルに頼らなくて済む分、処理系のほかの部分でのシグナル処理を気にする必要がありません。

カーソルの後処理

シグナルといえば、もう一つ気になることがあります。現在のgraph_barの実装ではグラフ描画のときにカーソルがちらつくのを避けるため、描画中はカーソルを消しています。

問題は[Ctrl + C]キーでキーボード割り込みをかけてプログラムが中断したときに、カーソルが消えたままになることです。これを解決するためには、キーボード割り込みによって送られるSIGINTシグナルに対してハンドラーを設定する必要があります。

しかし、SIGWINCHのところでも解説したように、シグナルハンドラーの設定ではプログラムのほかの部分で同じシグナルに対して別々のハンドラーを指定できないという問題があります。通常、全体を協調して開発するアプリケーションソフトウェアではあまり問題にならないのですが、プログラミング言語処理系のような各機能が独立している場合にはこの点が問題になります。

困った。

しかし、考えてみると一つのシグナルに一つしかハンドラーを設定できないのが問題の根源です。そこでStroom処理系の中でのシグナルハンドラー設定用の関数を用意することにしました。新関数

```
strm_signal(sig, func, arg)
```

でsigで指定した番号のシグナルに対してハンドラー関数funcを指定します。複数ハンドラーを指定しても上書きされず、すべて呼ばれます。argはvoid*型の引数でハンドラー関数に引数として渡されます。

これを使えば、Stroom処理系の別の場所で同じシグナルをハンドルしたいという状況でもハンドラーの衝突を心配する必要はありません。

今後の課題

さて、ここまででとりあえず数値ストリームから棒グラフを出力できるようになりました。しかしキャラクターでドットを表現するのは21世紀の観点からはいかにも貧弱です。もうちょっとマシな出力はできないものかと考えるのは当然でしょう。もっとドットを。

Sixelグラフィックス

とはいえ、最初に書いた理由からGUIライブラリを使うのもできれば避けたいところです。そんなときにぴったりの技術がSixelグラフィックスです。Sixelは「Six Pixels」の略で、ターミナル上の一つのキャラクターを六つの「ピクセル」に分解した上に、ピクセルごとに256色をエスケープシーケンスで指定するという技術です。1980年代にDECのVT200シリーズで導入されたこれまた比較的古い技術ですが、これを解釈するターミナルエミュレーターでは、拡張機能を用いて最大1600万色まで表示できます。また、エスケープシーケンスを用いた非効率なグラフィックス表示でありながら、現代のマシンではGIFアニメーションをスムーズに表示できます。

Sixelグラフィックスの欠点はすべてのターミナルがSixelグラフィックスに対応していない点です。例えば、私の手元ではxtermとmltermは対応していましたが、gnome-terminalとrostermは対応していませんでした。

まとめ

今回は数値ストリームを入力として、キャラクターグラフィックスで棒グラフを出力する関数を作りました。正直しょぼいグラフですが、Sixelを使ったピクセル単位の表示など今後の改善に期待です。

タイムマシンコラム

Streemの開発は今後も続く

2016年11月号掲載分です。今回はキャラクターベースのシンプルなグラフ表示機能の実装です。データを処理してグラフを表示させたいときは、たびたびありますが、そのためにデータをExcelに取り込んでグラフ機能を使うのは面倒だという私のような人向けの機能です。

とはいえ、今回実装できたのは目が粗い棒グラフを表示するのが精一杯でした。とてもグラフ機能ができましたとはいえないレベルなのですが、まあ、不可能ではないことを示すことだけできたのではないのでしょうか。

ここまで、何回かかけてStreemに機能を追加してきましたが、ここで一段落です。もちろん、Streemは完成にはほど遠いので、開発をやめてしまうわけではありませんが、無限に連載を続けるわけにもいきません。本書もここで一段落、今後はコミュニティーベースのOSSとしての開発を継続することにしましょう。

おわりに

本文にも書きましたが、私は統計を含む数学全般が苦手です。それからコンカレントプログラミングも得意ではありません。しかし、ビッグデータやデータサイエンスなどの言葉が流行し、コンピュータのマルチコア化が進む現代において、私の苦手なそれらの分野の重要性は増すばかりです。

私たち（の多く）はエンジニアです。問題があれば技術でそれを解決しようというのがエンジニア魂なのではないかと思います。そこでこれらの問題に対する私なりの解答がStreemなのでした。もっとも、ひらめきはあっても実力が伴わない領域での戦いは困難を極めました。本書でも何度か懺悔している通り、実力が及ばずバグを取り切れなかったところや、やる気と時間が不足して完成度が低いまま取り残されているところがあります。しかし、それらの「欠点」は、Streemという言葉そのものの価値を減じるものではないだろうと考えています。

実際、私はこのStreemの実行モデルを相当気に入っていて、2015年のRubyKaigiやRuby Confのキーノートでは、Rubyの将来のバージョン（Ruby3）にStreemのコンカレンシーモデルをベースにした機能を取り込もうという提案をしています。割と本気だったのですが、その後の検討で最終的なRuby3には、どうやら笹田耕一さんが提案したGuild（ギルド）というモデルの方が実際に採用されそうです。こちらの方が表現力や互換性に優れているという判断です。

Streemの抽象度が高いコンカレンシーモデルは、きめ細かな処理記述ができない点が利点でもあり、欠点でもあります。Ruby3に取り込むのは欠点の部分が無視できませんでした。

まあ、私の考えたStreemのモデルを取り込む方式は、RubyとStreemで異なる実行モデルを一つにするために、「一つの言語に二つのモデル」が存在する、別の言い方をすると「よく似た二つの言語が混じり合っている」ことになっているので、ユーザーにとっての混乱が大きくなる危惧がありました。採用できなかったのは仕方がないことだと自分でも思いますが。

できなかったこと

本文中でも懺悔したように、Streamはまだまだ未完成です。特にガーベージコレクション（GC）がろくに実装できていないことは大問題です。現在のStreamの処理系はほとんどメモリーを解放していません。ですから、大量のデータを扱うと必然的にメモリーを食いつぶすことになります。

というのも、4-4では、スレッドごとに別の空間にオブジェクトを割り当てて、GCもスレッドごとに行うことで効率を上げると解説しましたが、実際に実装してみるとこれが予想以上に困難でした。パイプラインでデータを受け渡すと、そのオブジェクトは次のスレッドに引き渡されることになります。また、このオブジェクトが配列のような、ほかのオブジェクトへの参照を含む場合、そこから参照されているオブジェクトを再帰的に次のスレッドのメモリー空間にコピーする必要があります。これが意外に面倒で、ちょっと後回しにしてみました。

実用的な処理系を目指すためにも、この点はなんとかしたいと考えています。最初のステップとしては、NaN Boxingと呼ばれるオブジェクト表現を変更した上で、libgcを利用しようと考えています。

libgcはリンクすることでC言語で記述されたアプリケーションにGCを追加するライブラリです。ただし、「ポインタの値を加工してはいけない」という制約条件があって、現状のStreamが採用しているNaN Boxingではこの制約条件を満たしません。

現代のほとんどすべてのコンピュータが採用している浮動小数点表現であるIEEE754では、倍精度浮動小数点数64ビットのうち、符号部に1ビット、仮数部に11ビット、仮数部に52ビットを使用しています。

また、計算結果が未定義の場合の演算結果（たとえば0で割るとか）を示す値としてNaN（Not a Number）という値が定義されています。「指数部がすべて1」の値がNaN値であると定められています。NaNを示す値は一つだけあればよいのに、実際には指数部がすべて1であればすべてNaNです。このため、うまくやればNaNで解釈される値域に52ビット分の整数値を埋め込むことができます。これがNaN Boxingです。実際には52ビットのうち、4ビットを値の種別を示すタグに、残りの48ビットを実際の値を格納するのに使っています。

48ビットしかなければ64ビットOSでのポインタを格納できないような気がします。しかし実際には、一部の例外を除き、Linuxをはじめとする多くのOSでは64ビットマシンでもポインタ値として48ビットしか使用していませんので問題になることはありません。実に巧妙です。

一方で、このやり方では、ポインタの値が浮動小数点数に変換されてしまうので、libgcはポインタを見つけることができず、GCできません。そこでNaN Boxingの部分に手を入れて、ポインタ値をポインタのままにするフォーマットに変更します。

発想としては簡単です。64ビットポインタ値のうち、実際にアドレスを表現するのは48ビットで、残りの16ビットはゼロになっています。NaN Boxingではこの部分にタグを入れてNaN値に見せかけています。ということは、ポインタ値を表現する場合にタグがゼロになるようにタグの値を調整してやればよいでしょう。もちろん、この調整した値は正当な浮動小数点数にはなりません。このため浮動小数点数を取り出すときに値を加工する必要がありますが、それほど大きな手間ではありません。

このようなNaN Boxingの変種をFavor Pointersというそうです。

NaN BoxingをFavor Pointersにしてしまった後は簡単です。現状malloc()を呼んでメモリーを割り当てている部分をGC_malloc()で置き換えるだけで、GCが実現できます。libgcは4-4で解説したようなアプリケーション固有の知識を用いた効率化はできませんが、スレッド対応した世代別GCを提供しているので、それなりの効率でGCしてくれます。

さよならとこれから

Ruby3に取り込まれないからといって、Streemの価値がなくなるわけではありません。Streemは独立した言語として今後も（ほそぼそと）開発を継続しようと考えています。

Streemに近い対象領域を持つ言語・ツールはほかにも存在します。例えば、tab^{*1}やdatamash^{*2}です。これらの言語・ツールのライバルになれるべく、今後もStreemを成長させようと思います。

まつもと先生の次回作にご期待ください！

2016年12月
まつもとゆきひろ

*1 tab <http://tkatchev.bitbucket.org/tab/>

*2 datamash <http://www.gnu.org/software/datamash>

まつもと ゆきひろ

1965年生まれ。鳥取県米子市出身。筑波大学第三学群情報学類卒業。プログラミング言語Rubyの生みの親。プログラミング言語デザインの第一人者。おそらく日本唯一のプログラミング言語デザイナー。株式会社ネットワーク応用通信研究所フェロー、一般財団法人Rubyアソシエーション理事長、Heroku Chief Architectなど、肩書多数。三女一男犬猫一匹ずつの父でもある。温泉好き。島根県在住。牡羊座。O型。

まつもとゆきひろ 言語のしくみ

2016年12月27日 第1版第1刷発行

著者	まつもと ゆきひろ
編集	日経Linux
発行人	寺山 正一
発行	日経BP社
発売	日経BPマーケティング
	〒108-8646 東京都港区白金1丁目17番3号

ブックデザイン	小口翔平+上坊菜々子+山之口正和 (tobufune)
制作	マップス
印刷・製本	図書印刷

ISBN 978-4-8222-3917-6

©まつもとゆきひろ 2016

- 本書の無断複写・複製（コピー等）は著作権法上の例外を除き、禁じられています。
購入者以外の第三者による電子データ化および電子書籍化は、私的使用を含め一切認められておりません。
- 本書の内容に訂正がある場合は、日経Linuxのサイトに掲載します。URLは、
<http://itpro.nikkeibp.co.jp/atcl/mag/14/236760/120400027/>
(短縮URL:<http://nkbp.jp/2gOuTFh>)です。