Ruby 逆引きハンドブック

るびきち◆著

Rubyのすべてを集大成した
逆引きリファレンスの
決定版!

Ruby
1.8.6/1.8.7/1.9
各バージョンに
対応!

圧倒的な情報量で詳細に解説。
手元に置いて、いつでも使える!

24時間無料でサンプルデータをダウンロードできます。

C&R研究所

Ruby 逆引きハンドブック

るびきち◆著

Ruby 1.8.6/1.8.7/1.9 各バージョンに対応!

C&R研究所

■権利について

- 本書に記述されている社名·製品名などは、一般に各社の商標または登録商標です。なお、本書では™、©、®は割愛しています。

■本書の内容について

- 本書は著者·編集者が実際に操作した結果を慎重に検討し、著述·編集しています。ただし、本書の記述内容に関わる運用結果にまつわるあらゆる損害・障害につきましては、責任を負いませんのであらかじめご了承ください。
- Rubyおよび拡張ライブラリなどは、仕様が変更になる場合もあります。本書で解説している場合と動作が異なったり、サンプルコードが動作しなくなる場合がありますので、あらかじめご了承ください。

■サンプルについて

- 本書で紹介しているサンプルは、C&R研究所のホームページ(http://www.c-r.com)からダウンロードすることができます。ダウンロード方法については、7ページを参照してください。
- サンプルデータの動作などについては、著者・編集者が慎重に確認しております。ただし、サンプルデータの運用結果にまつわるあらゆる損害・障害につきましては、責任を負いませんのであらかじめご了承ください。
- サンプルデータの著作権は、著者及びC&R研究所が所有します。許可なく配布・販売することは堅く禁止します。

●本書の内容についてのお問い合わせについて

この度は C&R 研究所の書籍をお買いあげいただきましてありがとうございます。本書の内容に関するお問い合わせは、FAX または郵送で「書名」「該当するページ番号」「返信先」を必ず明記の上、次の宛先までお送りください。お電話や電子メール、または本書の内容とは直接的に関係のない事柄に関するご質問にはお答えできませんので、あらかじめご了承ください。

〒 950-3122 新潟県新潟市北区西名目所 4083-6　株式会社 C&R 研究所　編集部
FAX 025-258-2801
「Ruby 逆引きハンドブック」 サポート係

PROLOGUE

Ruby on Railsの大ブレークで日本発のオブジェクト指向スクリプト言語「Ruby」は世界的に注目されています。もちろんRubyはWeb専用言語ではなくて、システム管理やテキスト処理から数値計算まで、あらゆる用途に適しています。Rubyの活躍する領域があまりにも広範囲に及ぶため、エッセンスを詰め込んだだけでもこんなにぶ厚い本になりました。

本書は「Rubyで○○するにはどう書けばいいのか?」という問いに答える逆引き本です。項目を開くと、そこにはRubyらしい答えが書いてあります。さまざまな条件に対応できるように、1つの項目につき、サンプルをたくさん用意しました。また、関連する項目を簡単に参照できるようになっています。困ったときにはとりあえず机の横に置いてある本書を参照するような、辞書的な使い方ができることを狙いにしています。

本書はRuby初めての人からベテランに至るまで、Rubyに関わるすべての人のために書きました。プログラミングが初めての人は入門書と併用するといいでしょう。プログラミング経験者はCHAPTER 01 ～ 04を流し読みするとRubyの概要がわかると思います。

本書はRubyの基礎から奥義まで紹介しています。文字列・正規表現・配列・ハッシュは、基本中の基本なので多くのページ数を割きました。Rubyでは組み込みのメソッドをしっかりマスターすることが大切だからです。後半の章から実践的な項目になっていき、最後の方はRubyの柔軟性に直に迫ります。Rubyは特に今流行りのDSL(ドメイン特化言語)の作成に向いています。DSLの作成についても解説しています。

RailsでRubyを知った人は、まずRubyの基礎を固めてください。基礎がしっかりしていないうちにRailsという応用をやろうとしても、すぐにつまづいてしまいます。これからRailsを始める初心者にも、Rubyをよりよく使おうと思っている中・上級者にも、本書が役立てば幸いです。

一般に、書籍のサンプルとなってるアプリケーションのソースコードは、スペースおよび説明の都合上、どうしても無理やりこしらえたおもちゃのような例になってしまいます。そこで本書のサンプルは、もっとミクロな視点で「このテクニックを使うと何が起きるのか」がはっきりわかるような説明的なコードにしました。

本書はリファレンスマニュアルを置き換えるものではありません。幸い、RubyにはReFe2というリファレンスマニュアル閲覧ツールがあります。詳しい情報を手早くReFe2で参照できるように、メソッド名はReFe2が解釈する形で表記しています。たとえば、組み込み関数「print」は「Kernel#print」と表記しています。

本書はRuby 1.8.6/1.8.7/1.9の各バージョンに対応しています。

最後に、Ruby 1.9の情報を追い掛けているMauricio Fernandez氏には感謝しています。現場では当分Ruby 1.8の時代が続くと思いますが、長年の議論でより使いやすくパワーアップしたRubyをお楽しみください。

2009年4月

るびきち

本書について

本書の表記方法

本書の表記についての注意点は、次のようになります。

▶サンプルの注釈方法について

サンプルには実際の使用例に加え、注釈記号による注釈を付けました。注釈記号は3つあります。注釈記号「# =>」の後ろには、その行の式の値を記しています。厳密にいうと「式の値に『inspect』メソッドを適用して人間が読みやすい文字列に変換したもの」を記しています。Rubyはオブジェクト指向で値ベースの言語であるため、式の値が何になるかが一番大切です。式の値を表示するとその値が文字列化されるため、値のクラスがあやふやになってしまいます。そのため、通常のプログラミング解説書のような「ソースコードと実行結果」形式ではなく、注釈方式を採用しました。irbでサンプルの式を1行1行、打ち込んで試してみてください。

例外（エラー）が発生するコードは「例外が発生するコード rescue $!　# => #<例外クラス:エラーメッセージ>」のように注釈しています。irbに打ち込むときはrescue以下は不要です。

注釈記号「# >>」は標準出力への出力を記しています。サンプルをRubyインタプリタで実行すると、そのように表示されることを意味します。仕様上の制限のため、出力はサンプルの末尾にまとめられますが、どの部分で出力されたのかがはっきりわかるように配慮しました。

注釈記号「# !>」はその行で発生した警告を記しています。仕組みの解説の上でやむを得ず警告が出るコードを示すことはありますが、実際のプログラミングでは警告の出るコードは記述すべきではありません。

次に注釈の例を示します。注釈方式による解説は、Ruby界でしばしば使われるので慣れてください。

```
1+2                          # => 3
["ab", "cd"]                 # => ["ab", "cd"]
Thread.main                  # => #<Thread:0xb7c08660 run>
raise "an error" rescue $!   # => #<RuntimeError: an error>
puts 1
puts 2
# >> 1
# >> 2
```

式「1+2」の評価結果が「3」であることを意味します。

例外が発生するコードです。

出力内容はまとめて最後尾になります。

▶クラスの表記方法について

Rubyのデータは文字列や配列も含め、すべて何らかのクラスに属したオブジェクトです。目次ページでは初心者が逆引きできるように「文字列」など、すべて日本語で表記してありますが、解説ページでは厳密にするためにクラス名で表記することがあります。日本語名とクラス名の対応は、右ページの表のようになります。

日本語名	クラス名
整数	Integer
小さい整数	Fixnum
大きい整数	Bignum
浮動小数点数	Float
数値	Numeric
文字列	String
正規表現	Regexp
配列	Array
ハッシュ	Hash
時刻	Time
範囲	Range
ペア	2要素の配列

▶インスタンスの表記方法について

本書ではクラス名をクラスのインスタンスという意味で使っています。たとえば「Stringを返す」は「Stringクラスのインスタンスを返す」、すなわち「文字列を返す」を意味します。また、Rubyではクラスそのものもオブジェクトなので、クラスをオブジェクトとして扱う場合は「Stringクラスオブジェクト」と明記します。

▶インデックスの表記方法について

文字列や配列のインデックスは1からではなくて0から始まります。たとえば配列「a = ["a", "b", "c"]」において「a[0] == "a"」「a[1] == "b"」「a[2] == "c"」となります。それに対して、通常の日本語では1番目から数えます。そのため、プログラミングの文脈で「N番目」という言葉は文脈によって0始点か1始点かが異なります。

本書ではインデックスを「インデックスN」と表記し、「N番目」は通常の日本語の意味(1始点)で使います。

なお、メソッドの引数については最初の引数が「第1引数」です。「第0引数」ではありません。

▶サンプルコードの中の「\」(バックスラッシュ)と「¥」(円記号)について

Windows環境では、「\」(バックスラッシュ)は「¥」(円記号)で表示されます。本書のサンプルコードでは「\」で表記していますので、Windows環境をお使いの方は読み替えてください。

▶サンプルコードの中の▼について

本書に記載したサンプルコードは、誌面の都合上、1つのサンプルコードがページをまたがって記載されていることがあります。その場合は▼の記号で、1つのコードであることを表しています。

本書で使用しているシェルコマンドについて

プログラムの実行結果ではいろいろなコマンドが使われています。登場するコマンドはGNU/Linuxなど、Unix系OSのコマンドなのでユーザには見慣れないかもしれません。本書で使われているコマンドについて解説します。

- 「cat」はファイルの内容を標準出力に表示するコマンドです。Windowsでは「type」コマンドに相当します。スクリプトの入力ファイルの内容を示す場合にも使用しています。
- 「echo」は引数をそのまま標準出力に表示するコマンドです。パイプと併用して短い文字列をコマンドの標準入力に渡すときに使用しています。Windowsでも同名です。
- 「sudo」はUnix系OSでは管理者権限でコマンドを実行します。主にライブラリなどのインストール時に使用しています。

シェルにはリダイレクト機能が存在します。通常、標準入力は端末からのキーボード入力で、標準出力は端末への表示です。それらをファイルにリダイレクトすると入出力がファイルに置き換えられます。Unix系OS・Windows共通です。

- 「>」は標準出力の内容をファイルに書き込みます。
- 「<」はファイルの内容を標準入力へ送ります。
- 「|」はパイプで、前のコマンドの標準出力を次のコマンドの標準入力へ送ります。

サンプルの動作について

サンプルの中には、Unix系OSのコマンドや環境などに依存しているため、Windows環境では動作しないサンプルがあります。また、お使いの環境によっては、本書の注釈と異なる結果が表示されたり、動作しない場合もあります。なお、Windows環境の場合、バイナリがない拡張ライブラリはインストールできなかったり、正しく動作しないことがあります。あらかじめ、ご了承ください。

サンプルファイルのダウンロードについて

本書のサンプルデータは、C&R研究所のホームページからダウンロードすることができます。

本書のサンプルを入手するには、次のように操作します。

❶「http://www.c-r.com/」にアクセスします。

❷トップページ左上の「商品検索」欄に「022-4」と入力し、[検索]ボタンをクリックします。

❸検索結果が表示されるので、本書の書名のリンクをクリックします。

❹書籍詳細ページが表示されるので、[サンプルデータダウンロード]ボタンをクリックします。

❺下記の「ユーザー名」と「パスワード」を入力し、ダウンロードページにアクセスします。

❻「サンプルデータ」のリンク先のファイルをダウンロードし、保存します。

サンプルのダウンロードに必要な
ユーザー名とパスワード

ユーザー名 **rubygk**
パスワード **6b8pu**

※ユーザー名・パスワードは、半角英数字で入力してください。また、「J」と「j」や「K」と「k」などの大文字と小文字の違いもありますので、よく確認して入力してください。

サンプルファイルは、CHAPTERごとのフォルダの中に項目番号のフォルダに分かれています。サンプルはZIP形式で圧縮してありますので、解凍してお使いください。

CONTENTS

CHAPTER 01 Rubyの基礎知識

CHAPTER 02 基本的なツール

CHAPTER 03 Rubyの文法

CHAPTER 05 文字列と正規表現

CHAPTER 08 数値と範囲

CHAPTER 09 時刻と日付

CHAPTER 10 入出力とファイルの扱い

CHAPTER 16 マルチスレッドと分散Ruby

CHAPTER 17 ドメイン特化言語(DSL)の構築

CHAPTER 01

Rubyの基礎知識

Rubyとは

Rubyの概要

Rubyは、まつもとゆきひろ氏が作成した、手軽に使えるオブジェクト指向スクリプト言語です。簡単に使え、使っていて「楽しい」言語です。しかもフリーソフトウェアです。言語おたくの作者がいろいろな言語を渡り歩いて、それぞれの良さを集結した究極の言語です。

▶日本発!

Rubyの作者は日本人で、日本語の使用は最初から考慮されています。日本語が文字化けする外国産ソフトウェアもある中、非常にありがたいことです。また、日本語の情報源が多いのも嬉しいことです。作者に日本語で質問できるのは助かります。

▶オブジェクト指向スクリプト言語

Rubyは、PerlやPythonと同様に、スクリプト言語です。そのため、プログラムを記述したらコンパイルしないですぐに実行できます。

Rubyは設計当初からオブジェクト指向を念頭に置いた言語です。オブジェクト指向と聞いて怖がる必要はありません。オブジェクト指向は、本来、人間にとって自然な考え方であり、Rubyならば自然な形でオブジェクト指向プログラミングすることができます。他言語のオブジェクト指向に挫折した人も、プログラム初心者も、Rubyを試してみる価値が大いにあります。

▶すぐに使えるクラスが豊富

オブジェクト指向プログラミングというと、すぐにクラス定義や継承が思い付く人もいるでしょう。肩の力を抜いてください。Rubyにはすでに多くのクラスが用意してあって、すぐに使えます。クラスを使うだけならば、とても簡単です。そして、多くの雑用はクラスを使うだけの手続き型プログラミングで済みます。

▶すぐに使える変数

変数はいちいち宣言する必要がありません。最初の代入が変数宣言を兼ねています。

ローカル変数、グローバル変数、インスタンス変数、クラス変数、定数と、いろいろな種類の変数が存在しますが、変数名を見ただけで、どの種類の変数かがわかります。

▶動的型付

変数には型がありません。その代わり、オブジェクトが自分の型を知っています。そのため、自分が考えている処理をすぐに記述できます。数値が代入してある変数に文字列を代入することさえ可能です。

▶すべてがオブジェクト

Rubyでは、整数や文字列など、基本的なデータ型もオブジェクトであり、特定のクラスに属しています。クラスさえもオブジェクトです。そのため、すべてのデータを差別することなく、統一的に扱えます。

▶特異メソッド

特定のオブジェクトに専用のメソッドを定義したり、上書きしたりすることができます。たとえば、引数に合ったメソッドを動的にくっつけるということも可能です。

▶ブロック

ブロックは、Ruby的高階関数です。ブロックを使うことで、コードをより抽象化することができます。しかも、わかりやすい文法です。

▶テキスト処理が得意

強力な文字列操作と正規表現は、Perlから受け継ぎました。テキスト処理に強いので、ネットワークプログラミングも得意です。

▶多倍長整数

組み込みの多倍長整数があります。無量大数（10の68乗）でも表現できます。それこそメモリが許す限り、どんなに大きい整数でも表すことができます。

▶拡張ライブラリが簡単に作れる

多くのスクリプト言語はC言語などで拡張することができます。Rubyの拡張ライブラリAPIは使いやすいよう工夫されているので、作成は簡単です。

▶温かいコミュニティ

まつもと氏は気さくで話しやすい人です。その人柄に触発されたのか、Rubyコミュニティは温かい雰囲気があります。疑問があったら怖がらずにメーリングリストで質問してみましょう。誰でも最初は初心者なのです。初心者の質問でも優しく答えてくれる人が、メーリングリストにはたくさんいます。

筆者も昔はくだらない質問をしては、先輩rubyistたちに助けられました。だからこそ今があります。初心者にも優しいコミュニティというのは意外に見過されがちですが、言語の発展において重要な要素だと筆者は考えています。

▶「楽しい」

Rubyプログラミングは楽しいです。これが一番大切です。楽しいプログラミングになるように、Rubyは慎重に設計されています。楽しいと必然的にやる気が出て、作業効率も上がります。ここまで「楽しい」を全面に押し出している言語は、他に知りません。

Rubyの入手方法について

Rubyのダウンロード

Rubyはフリーソフトウェアです。つまり、誰でも自由に無料で使えます。

Windows環境では、ActiveScriptRubyを使うのが無難です。ActiveScriptRubyには、Ruby本体に加え、RubyGemsやWindows専用のライブラリなども付いてきます。

Unix系OSならば、ソースからインストールしましょう。もしも使っているOSにパッケージシステムがあるならば、それを使うと楽です。

具体的なインストール方法については、Rubyインストールガイドに詳細な説明があるので、参考にしてください。

ソースからインストールする場合、なるべく安定版を選びましょう。最新版はあくまで開発版なので、仕様変更の恐れがあります。冒険したい人はどうぞ。

内　容	URL
Ruby公式サイト	http://www.ruby-lang.org/ja/
ActiveScriptRuby	http://arton.hp.infoseek.co.jp/indexj.html
Rubyソースコード	http://www.ruby-lang.org/ja/downloads/
Rubyインストールガイド	http://www.ruby-lang.org/ja/install.cgi

◉ Ruby公式サイト

Rubyの基本的な記述方法について

Rubyスクリプトファイルの作成

Rubyスクリプトは、テキストエディタかRuby統合開発環境（IDE）を使って作成することができます。Windows環境の場合はメモ帳でも、一応、作成することはできますが、より高機能なテキストエディタを使うことをお勧めします。最近のテキストエディタはRubyのソースコードにも対応していて、色を付けたり、括弧の対応を取ったりすることができます。**エディタのサポートは開発効率に大きく影響します。**

Rubyスクリプトの拡張子は、基本的に「.rb」です。CGIスクリプトとして実行する場合は「.cgi」にするなど、一部の例外はあります。実行スクリプトの場合は、拡張子がないことも多くあります。

Rubyの式の評価結果をすぐ知る方法

Rubyは制御構造も含め、すべての「文」に値を持つ言語です。そのため、以後「Rubyの式」という表現に統一します。

Rubyの式の評価結果を知りたいのならば、わざわざスクリプトファイルを作成せずに、インストールしたらすぐに使える「irb」というRuby電卓（REPL:Read-Eval-Print-Loop）を使いましょう。

Rubyスクリプトの基本的な記述

Rubyスクリプトの基本的な記述は、次のようになります。

▶「#!」行

実行用のRubyスクリプトの1行目は「#!」から始めます。Unix系OSを使っている人やCGIスクリプトを設置したことのある人にはおなじみのshebangです。shebangには「#!/usr/bin/env ruby」や「#!/usr/bin/ruby -Ke」などと記述します。前者は「env」コマンドでRubyインタプリタを環境変数「PATH」から探して実行する指定、後者は「/usr/bin/ruby」を「-Ke」オプション付きで実行する指定です。Unix系OSはshebangを読み取ることで、実行許可されたスクリプトをそのまま実行することができます。

Windowsだと「#!」の行に「ruby」という文字列が含まれる場合、その右にある「-」から始まる文字はRubyインタプリタのオプションとして認識されます。Unix系OSのshebangとは異なり、あくまでオプションの指定のみに使われます。

▶インデントは2

Rubyも他言語と同様に、制御が深くなっていくとインデントします。Rubyのインデントは、ほぼすべて2です。他言語の4 ～ 8インデントに目が慣れている人には浅い気がするかもしれませんが、そのうち慣れます。

なお、タブでインデントするのは、お勧めできません。タブの幅はエディタの設定で可変なのでソースのインデントがぐちゃぐちゃになりかねません。ほんのわずかなバイト数を節約すること

はできますが、大容量時代である今日では、弊害の方がずっと大きくなります。タブでのインデントは避けましょう。

COLUMN エディタはパートナー

プログラミング上達のためには、エディタの習熟は欠かせません。Windowsのメモ帳を使っている人は、今すぐに使用を中止して別のエディタに乗り換えましょう。動作実績があり、拡張性の高いエディタがいいでしょう。エディタの扱いに慣れている人は、よりよい使い方を考えてみましょう。

プログラミング向きのエディタには、次に解説する機能が重要です。たとえ現時点でない機能であっても、エディタの拡張言語で実装できればそれで構いません。

まず自動インデント機能は必須です。間違っても手でスペース連打してはいけません。正しいインデントをするのはエディタの仕事です。自動インデントがおかしくなったら、大抵、どこかに間違いがあるのです。もっとも、Rubyはパーサ泣かせな言語なので、自動インデントが狂うことはありますが……。

ソースコードの視覚効果は馬鹿になりません。コメントに色が付いていると、そこがコメントだとひと目でわかります。文字列リテラルに色が付くと、閉じ忘れやエスケープ忘れを未然に回避できます。また、目視で括弧の対応を取るのはとても目が疲れるので、閉括弧を打ったら対応する括弧をハイライトする機能も必要です。

エディタの中でシェルコマンドを実行する機能は重要です。ドキュメントを書いたり、メールで動作結果を送ったり、ブログで発表したりするなど、テキストの中にスクリプトの実行結果を埋め込む必要性が必ず出てきます。

シェルコマンドを実行するだけでなく、スクリプトのエラーが発生したとき、その場所へジャンプする機能があると非常に快適です。

筆者は10年以上にわたりすべての編集作業をEmacsのみで行っています。最初は特異なキーバインドに悪戦苦闘しましたが、慣れると指が勝手に動くようになります。Ruby本体にはRubyプログラミングを快適にするEmacs Lispが含まれています。ある作業をしていて、面倒だなと感じたらすぐにEmacsのドキュメントを参照して、よりよい編集方法を探します。見つからなかった場合はとりあえずネットで検索して、それでも見つからなかった場合はさくさくとEmacs Lispで欲しい機能を実装します。確かにエディタの勉強・拡張の時間は多少取られますが、先行投資と考えています。実装した暁には効率的に編集できるので、長い目で見ると時間は節約できるでしょう。

エディタは手の延長線であると同時にあなたのパートナーです。エディタにいい仕事をさせるには、あなたもエディタも成長しなければなりません。

関連項目 ▶▶▶

- スクリプトの文字コードを設定する …… p.38
- 手軽な実験環境 …… p.64

Rubyの実行について

Rubyの実行方法

Rubyスクリプトを実行するには、コマンドラインでRubyの式を実行する方法（ワンライナー）と、irbを使う方法、スクリプトファイル名を指定して実行する方法があります。

ワンライナー

ワンライナーを実行するには、「ruby -e 'Rubyの式'」とします。わざわざファイルを作成するまでもないような小さいRubyスクリプトを実行するのに使います。シェルとの親和性もあります。ワンライナーには独自のテクニックがあるので、《ワンライナーを極める》(p.517)で解説します。

irbを使う方法

irbは入力したRubyの式やスクリプトファイルを、1行ずつ実行してその値を表示します。手軽にRubyの使い方を学習できます。

Rubyスクリプトファイルを実行する

Rubyスクリプトファイルを実行するには、「ruby オプション スクリプトファイル名 コマンドライン引数」とします。オプションとコマンドライン引数は省略可能です。日本語を含むスクリプトを実行するときは、スクリプトに文字コードを設定してください。

Unix系OSを使っている開発者は、shebang機能を使ってスクリプトをコマンドとして実行できるように、環境変数「PATH」の通ったディレクトリにスクリプトファイルを置いています。shebang機能が使えないWindowsは、そのままでは実行できません。代わりに「ruby -S スクリプト」と実行してください。

Rubyはアプリケーションに組み込まれていたり、CGIスクリプトとして実行することができたりと、実行方法は多岐に及びます。

Rubyインタプリタのオプション

Rubyインタプリタには、マニアックなものも含め、多様なオプションがあります。すべてを知るためには、「ruby -h」を実行してください。

通常の使用で重要となるのは、次のオプションになります。

オプション	内　容
-Kkcode	スクリプトの文字コードkcodeを指定する（Ruby 1.8）
-S	実行するスクリプトを環境変数「RUBYPATH」と「PATH」から探す
-Idirectory	ライブラリのロードパスの先頭にdirectoryを加える
-rlibrary	スクリプト実行前に「require library」を実行する

関連項目 ▶▶▶
- Rubyの基本的な記述方法について …… p.35
- 手軽な実験環境 …… p.64
- コマンドライン引数を読む …… p.494
- ワンライナーを極める …… p.517

スクリプトの文字コードを設定する

Ruby 1.8では「-K」オプションを指定する

日本語文字列、日本語正規表現という日本語を含むスクリプトを記述するときには、文字コードのオプションを指定します。

▶ 文字コードオプションを付けてRubyを起動する

Rubyインタプリタには、スクリプトの文字コードを指定する「-K」オプションが存在します。「-K」の後ろに「e」「s」「u」「n」の引数を受け付けます（下表参照）。

「-K」オプションの引数	指定方法	内　容
e	-Ke	EUC-JP
s	-Ks	Shift_JIS
u	-Ku	UTF-8
n	-Kn	日本語を認識しない（デフォルト）

スクリプトに日本語を含む場合は、「-K」オプションを付ける必要があります。「-K」オプションを付けないと正規表現が日本語文字を認識してくれなくなり、場合によっては字句解析中にエラーになることがあります。

字句解析中のエラーが起きる有名な例として、Shift_JISの「表」という文字があります。とんでもないことに、この文字の2バイト目は「\」となっています。そのためShift_JISを認識しないで「'表'」というコードを記述した場合、閉じる「'」がエスケープされてしまい、「文字列が閉じてないよ」と怒られてしまいます。他にも「能」という文字も該当します。

この問題を解決するために「'表\'」と記述するのは、日本語を認識しない言語処理系でよく行う対処方法です。当然、Rubyでもその方法で対処できますが、さすがは日本発の言語だけあって、文字の内部表現のことを考えずに解決することができます。Shift_JISで記述されたスクリプトを実行するには「-Ks」を指定するだけです。

▶ 「#!」にオプションを付ける

コマンドラインに、毎回、「-K」オプションを付けるのは面倒です。スクリプトの最初の行に「#!ruby -Ks」と記述すれば、Rubyインタプリタは、あたかも「-Ks」オプションを付けたかのように振る舞います。「#!」のことをshebangといいます。

Unix系OSの場合、#!の後ろにRubyインタプリタのフルパスを記述してスクリプトに実行許可を与えると、スクリプトから、直接、実行することができます。

```
$ cat sjis.rb
#!ruby -Ks
puts '表'
$ ruby sjis.rb
表
$
```

Ruby 1.9では必ず「magic comment」を付ける

Ruby 1.9では、「magic comment」という、より進んだ方法を採用しています。「magic comment」は文字コード（エンコーディングと呼ぶ）を指定する「魔法のコメント」のことです。「magic comment」を導入することで、異なるエンコーディングのスクリプトをシームレスに読み込ませることができるようになりました。たとえば、EUC-JPで記述されたスクリプトからShift_JISで記述されたスクリプトをロードすることがあるとします。Ruby 1.8だとロードする前に「$KCODE='s'」を設定して、ロード後に「$KCODE='e'」に戻す必要がありますが、Ruby 1.9ではその必要がなくなります。スクリプトごとにエンコーディングを認識します（スクリプトエンコーディング）。

「magic comment」は基本的にスクリプトの最初の行に付けますが、最初の行がshebangの場合は、次の行に「magic comment」を付けます。

「magic comment」は基本的に「# coding: エンコーディング名」と記述します。実際はもっと柔軟にできていて、「coding」という文字列に区切り文字（「：」か「=」）とエンコーディング名が続いていれば、「magic comment」とみなされます。区切り文字の前後に空白が含まれていても構いません。また、「coding」の前やエンコーディング名の後ろに余計な文字列があっても構いません。エンコーディング名は、大文字と小文字を区別しません。

「magic comment」の形式が柔軟な理由は、各種エディタのエンコーディング指定に対応させるためです。

「# -*- coding: エンコーディング名 -*-」の形式は、Emacsでのファイルエンコーディング指定と合致しています。Emacsでは「euc-jp-unix」のように、エンコーディング名に「-dos」「-unix」「-mac」という接尾辞を指定することができます。それに対応するように、Rubyでは接尾辞を認識します。

「# vim: set fileencoding=utf-8：」の形式は、Vimでのファイルエンコーディング指定と合致しています。

エンコーディング	magic commentの例
UTF-8	# -*- coding: utf-8 -*-
EUC-JP	# -*- coding: euc-jp -*-
	# -*- coding: euc-jp-unix -*-
	# coding: euc-jp
	# vim: set fileencoding=euc-jp：
Shift_JIS	# -*- coding: shift_jis -*-
	# -*- coding: shift_jis-dos -*-
	# -*- coding: sjis -*-
Windows-31J	# coding: windows-31j
	# -*- coding: cp932 -*-

Ruby 1.9でもしばらくの間は「-K」オプションは存続していますが、あくまで互換性のためなのでdeprecated（非推奨）です。Ruby 1.9で日本語を含むスクリプトを実行する場合は、**必ず「magic comment」を付けましょう。**

Ruby 1.8では、「magic comment」は普通のコメントとして読み飛ばされます。それでも移

植性を考えるとshebangでの「-K」オプションと、「magic comment」の双方を指定しておくべきです。たとえば、エンコーディングがEUC-JPの場合は、次のようになります。スクリプトエンコーディングは疑似変数「__ENCODING__」で参照できます。

```
#!/usr/local/bin/ruby -Ke
# -*- coding: euc-jp -*-
s = "EUCですよ。"
__ENCODING__  # => #<Encoding:EUC-JP>
```

関連項目 ▶▶▶

- Ruby 1.9のエンコーディングについて …… p.60
- Rubyでの日本語の扱いについて …… p.192
- Ruby 1.9のIOエンコーディングについて …… p.451

スクリプトを探索する順序について

スクリプト探索パス

カレントディレクトリ以外のスクリプトを指定するときに、常にフルパスで指定しないといけないとしたら、長くなって打ち込むのが面倒です。おまけに呼び出されるスクリプトを他のディレクトリに移動したら、呼び出すスクリプトのパスも書き換えないといけません。それを解消するために、スクリプトの探索パスがあります。スクリプトのファイル名のみを指定した場合、探索パスの優先順位の高い方から順に探索し、見つかったスクリプトを適用します。

探索パスには、次のように、相対パスを設定することができます。そのときは、カレントディレクトリを基準とした相対パスになります。

▶探索パスの設定方法

探索パスが1つの場合は1つのディレクトリ名を、複数の場合はディレクトリ名を区切り文字で区切ります。区切り文字は、Windowsは「;」(セミコロン)で、Unix系OSが「:」(コロン)です。

▶実行スクリプトの探索パス

Rubyインタプリタに「-S」オプションを付けた場合、スクリプトは実行スクリプト探索パスから探索されます。環境変数「RUBYPATH」が設定されている場合は、そのディレクトリから探索します。その次に、環境変数「PATH」に設定されているディレクトリから探索します。

▶ライブラリの探索パス(ロードパス)

Rubyでは、「load」や「require」でスクリプトをライブラリとして読み込めます。ライブラリの探索パスは「$LOAD_PATH」(特殊変数なら「$:」)にディレクトリの配列で設定されています。ロードパスを知るには、「ruby -e 'puts $:'」を実行します。

環境変数「RUBYLIB」を設定すると、ロードパスの先頭に設定したディレクトリが追加されます。

さらに「-I」オプションでディレクトリを指定すると、環境変数「RUBYLIB」よりも優先して探索します。

次は、Unixシェル上におけるロードパスの表示例です。

```
$ RUBYLIB= ruby -e 'puts $:'
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/vendor_ruby/1.8
/usr/local/lib/ruby/vendor_ruby/1.8/i686-linux
/usr/local/lib/ruby/vendor_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
$ RUBYLIB=/opt/ruby:/opt/ruby/1.8 ruby -e 'puts $:'
```

```
/opt/ruby
/opt/ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/vendor_ruby/1.8
/usr/local/lib/ruby/vendor_ruby/1.8/i686-linux
/usr/local/lib/ruby/vendor_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
$ RUBYLIB=/opt/ruby:/opt/ruby/1.8 ruby -Ilib:bin -e 'puts $:'
lib
bin
/opt/ruby
/opt/ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8
/usr/local/lib/ruby/site_ruby/1.8/i686-linux
/usr/local/lib/ruby/site_ruby
/usr/local/lib/ruby/vendor_ruby/1.8
/usr/local/lib/ruby/vendor_ruby/1.8/i686-linux
/usr/local/lib/ruby/vendor_ruby
/usr/local/lib/ruby/1.8
/usr/local/lib/ruby/1.8/i686-linux
.
```

関連項目 ▶▶▶

- ライブラリを読み込む …… p.43
- ライブラリが意図通りに動かない原因について …… p.47

ライブラリを読み込む

ライブラリの使い方

Rubyの組み込みクラスは充実していますが、それだけでは実用的なプログラミングはできません。標準のRubyにないクラスやメソッドを使用する場合は、ライブラリをロードする必要があります。

▶ライブラリの種類

ライブラリとは、標準のRubyにないクラスやメソッドを定義するものです。ライブラリには、Rubyスクリプトと拡張ライブラリがあります。

Rubyスクリプトによるライブラリの拡張子は「.rb」です。

拡張ライブラリは、C言語などのコンパイル言語で記述されたライブラリです。C言語のライブラリをRubyで使用する場合や、Rubyスクリプトでは遅すぎる場合は、拡張ライブラリを使うことになります。拡張子は「.so」や「.dll」などプラットフォームによって異なります。拡張ライブラリを使用するにはコンパイルする必要があるため、コンパイラがない環境の場合は、あらかじめコンパイルされたバイナリを探してくる必要があります。

▶ライブラリをロードする

ライブラリをロードするには、「Kernel#require」を使用します。ロードとは、ライブラリを読み込んでトップレベルでその内容を評価（実行）することです。「require ライブラリ名」という書式になります。

ライブラリ名は、絶対パス（フルパス）でも相対パスでも記述できます。絶対パスの場合は、そのライブラリがロードされます。相対パスの場合はロードパス（「$LOAD_PATH」「$:」変数）から順次ライブラリを探します。たとえば、ロードパスに「/usr/local/lib/ruby/1.8」が含まれている場合に「require 'cgi/session'」すると「/usr/local/lib/ruby/1.8/cgi/session.rb」がロードされます。

ライブラリの拡張子は、省略するのが普通です。省略した場合、「Rubyスクリプト→拡張ライブラリ」の順に検索されます。

「require」は、同じライブラリを、複数回、ロードしません。すでにロードされている場合は、無視されます。ロードしたかどうかは、すでにロードしたライブラリを「$"」変数に追加することで判別します。

▶任意のRubyスクリプトをロードする

任意のファイルをRubyスクリプトとしてロードするには、「Kernel#load」を使用します。すでにロードされたライブラリを再ロードすることもできます。拡張ライブラリには使用できません。

拡張子は「.rb」でなくても構いませんが、その代わり、拡張子まで指定する必要があります。Rubyスクリプトによる設定ファイルをロードするには、「load」を使用してください。再読み込みできるのが強味です。

COLUMN ファイル間でローカル変数は共有できない

「require」や「load」では、グローバル変数とインスタンス変数、クラス変数、定数は共有できますが、ローカル変数は共有できません。ローカル変数の影響範囲を局所化するために、そういう仕様になっています。

次に実例を示します。

```
# vars.rb
$gvar = 1
@ivar = 1
class X
  @@cvar = 1
end
CONST = 1
lvar = 1
lvar2 = 1
```

```
require 'vars'
$gvar       # => 1
@ivar       # => 1
class X
  @@cvar    # => 1
end
CONST       # => 1
lvar rescue $!
# => #<NameError: undefined local variable or method `lvar' for #<Object:0xb7bac644 @ivar=1>>
```

このようにvars.rbにあるローカル変数を読み取ろうとしても「NameError」になります。

すでに宣言されているローカル変数lvar2を共有するには「Kernel#eval」と「IO.read」で可能ですが、強引な方法なのでお勧めできません。それでも、宣言されていないローカル変数（lvar）を外部から与えることはできません。

```
lvar2 = 0
eval File.read("vars.rb")
lvar rescue $!
# => #<NameError: undefined local variable or method `lvar' for #<Object:0xb7bac644 @ivar=1>>
lvar2                                   # => 1
```

COLUMN ローカル変数の代わりに無引数メソッドを使用する

Rubyでは、ローカル変数の参照と無引数メソッド呼び出しは、字面上、同じになっています。そのため、無引数メソッドをローカル変数の代わりにすることができます。前ページのCOLUMNのコードと見比べてみてください。lvarに「代入」すると、今度はメソッド呼び出しではなく、ローカル変数「lvar」が宣言されます。

```
# vars2.rb
def lvar() 1 end
```

```
require 'vars2'
lvar                            # => 1
defined? lvar                   # => "method"
lvar = 2
lvar                            # => 2
defined? lvar                   # => "local-variable"
```

モジュールやメソッド定義など、ローカル変数が見えないスコープでは、「lvar」メソッドが見えるようになります。

```
module Foo
  lvar                          # => 1
  defined? lvar                 # => "method"
end
```

「lvar」メソッドを再定義しても、ローカル変数「lvar」が見えるのでローカル変数が優先されます。ローカル変数が見えないところでは再定義が反映されます。

```
def lvar() 3 end # !> method redefined; discarding old lvar
lvar                            # => 2
module Foo
  lvar                          # => 3
end
```

ローカル変数が見えていても、「()」を付ければ強制的にメソッド呼び出しになります。

```
lvar()                          # => 3
```

COLUMN グローバルな名前空間を汚染しないでロードする

「Kernel#load」の省略可能な第2引数にtrueを指定した場合、グローバルな名前空間を汚染しません。内部で生成される無名モジュールがトップレベルとなるからです。これは、複数の独立したRubyスクリプトを1つのプロセスで動かす場合に使います。ただし、既存のクラスを開くには、「::」を前置する必要があります。

```
# anonymous.rb
def hoge
  p :hoge
 end
class ::Object       # Objectクラスを開くには、こうする必要がある
  def fuga
    p :fuga
  end
end
hoge
```

```
load "anonymous.rb", true
1.fuga               # うまくいく
hoge                 # エラー！
# ~> -:3:in `<main>': undefined local variable or method `hoge' for main:Object (NameError)
# >> :hoge
# >> :fuga
```

関連項目 ▶▶▶

● スクリプトを探索する順序について …… p.41

ライブラリが意図通りに動かない原因について

ロードパス問題

ライブラリが意図通り動かない原因のうち、典型的な原因はライブラリのロードパスの優先順位にまつわる問題です。Rubyに限らず、あらゆる言語において起こることです。

▶ロードパスの優先順位に気を付けよう

ライブラリが意図したように動かない場合、まずは他のライブラリがロードされている可能性を疑ってください。ライブラリ探索の優先順位は、ロードパスの先頭にあるほど高くなります。そのため、本来、読み込まれるべきディレクトリよりも前にあるディレクトリにライブラリが見つかった場合は、そちらの方が読み込まれます。

ロードパス問題を説明するために、筆者のロードパスを例にします。

```
puts $:
# >> /m/home/rubikitch/ruby
# >> /usr/local/lib/ruby/site_ruby/1.8
# >> /usr/local/lib/ruby/site_ruby/1.8/i686-linux
# >> /usr/local/lib/ruby/site_ruby
# >> /usr/local/lib/ruby/1.8
# >> /usr/local/lib/ruby/1.8/i686-linux
# >> .
```

ロードパスの最優先になるディレクトリは「/m/home/rubikitch/ruby」で、自作のライブラリを置いています。そして「/usr/local/lib/ruby/site_ruby」以下に標準添付されていないライブラリを置いています。その次に「/usr/local/lib/ruby/1.8」以下に標準ライブラリを置いています。最後にカレントディレクトリからライブラリを探索します。まとめると「自作ライブラリ→インストールしたライブラリ→標準ライブラリ→カレントディレクトリのライブラリ」の順になります。

Rubyは、デフォルトでは自作のライブラリのディレクトリを設定していません。筆者の環境では、環境変数「RUBYLIB」に「/m/home/rubikitch/ruby」を設定したため、ロードパスの先頭に加わっています。

▶既存ライブラリと同名の自作ライブラリを作ってはいけない

たとえば、筆者の環境で、「uri.rb」は「/usr/local/lib/ruby/1.8/uri.rb」にありますが、「/m/home/rubikitch/ruby」に「uri.rb」を置いた場合は、「/m/home/rubikitch/ruby/uri.rb」が読み込まれてしまいます。標準ライブラリの存在を知らずに同名の自作ライブラリを自分用のディレクトリに置いてしまうと、標準ライブラリを使用するスクリプトが自作ライブラリを読み込んでしまい、誤動作してしまいます。

この悲劇を防ぐためには、あらかじめ、どのような標準ライブラリがあるのかを知っておく必要があります。そして、標準ライブラリと同じにならないファイル名にしましょう。また、標準ライブラリだけでなく、インストールしたライブラリとも同じにならないようにしましょう。

▶ライブラリの存在チェック

既存のライブラリと同じファイル名になっていないかどうかをチェックする簡単な方法があります。シェルから「ruby -rチェックするライブラリ名 -e ''」というコマンドを実行するのです。これはライブラリを読み込んですぐに終了するワンライナーです。既存のライブラリ名の場合は何も表示せずに終了しますが、存在しないライブラリの場合は「LoadError」が発生します。「LoadError」が発生すれば、同じファイル名のライブラリはないので、安心してそのライブラリ名を使えます。

次の例では、「cgi.rb」と「newlib.rb」の存在をチェックしています。cgi.rbは標準ライブラリとして存在するので何も表示せずに終了しましたが、newlib.rbは存在しないライブラリなので「LoadError」を起こしました。

```
$ ruby -rcgi -e ''
$ ruby -rnewlib -e ''
ruby: no such file to load -- newlib (LoadError)
```

環境変数「RUBYLIB」を設定している人は、環境変数「RUBYLIB」を設定しない状態で上記のコマンドを実行すると、自作ライブラリかどうかを判別することができます。「LoadError」が発生すれば、自作ライブラリか存在しないライブラリです。

▶インストールしたライブラリと標準ライブラリの衝突

自作ライブラリとの衝突は簡単に判別できるため、それほど深刻な問題ではありません。しかし、めったに起こらないものの、インストールしたライブラリと標準ライブラリの衝突は、かなりタチの悪い問題です。次のストーリーを考えてください。

- 自分の環境にはRuby 1.8.1がインストールしてある。
- ライブラリxxのバージョン0.9をインストールしてある。
- (長い年月の末)Ruby 1.8.6をインストールする。
- 1.8.6にはxxのバージョン1.0が標準添付されている。

この場合、「/usr/local/lib/ruby/site_ruby/1.8/xx.rb」(0.9)と「/usr/local/lib/ruby/1.8/xx.rb」(1.0)が共存している状態にあります。ロードパスは標準ライブラリよりもインストールしたライブラリを優先するため、「require 'xx'」したら古いバージョン0.9が読み込まれることになります。

ライブラリはなるべく互換性が保たれる状態で開発されるため、古いバージョンでも動くものと動かないものが出てきます。そして、気付くのが遅くなりがちです。そういう意味でかなりやっかいです。こうなると、ライブラリのファイル名を知る必要があります。

関連項目 ▶▶▶
● ライブラリのファイル名を得る ……p.753

Rubyを制御する環境変数について

Rubyインタプリタが参照する環境変数

いくつかの環境変数を設定することで、Rubyインタプリタの挙動をカスタマイズすることができます。

▶環境変数の指定方法

環境変数の指定方法は、シェルによって異なります。お使いのシェルのドキュメントを参照してください。なお、Windowsについては、バッチファイルでの設定方法です。

シェル	指定方法
sh系	RUBYLIB=$HOME/ruby:$HOME/ruby2
	export RUBYLIB
csh系	setenv RUBYLIB $HOME/ruby:$HOME/ruby2
Windows	set RUBYLIB=%HOME%/ruby;%HOME%/ruby2

▶環境変数「RUBYOPT」

環境変数「RUBYOPT」には、Rubyインタプリタにデフォルトで渡すオプションを指定します。たとえば、環境変数「RUBYOPT」に「-Ke」を渡すと、コマンドラインや「#!」で「-Ke」を指定しなくてもEUC-JPを認識します。一見すると便利そうですが、筆者は利用をお勧めしません。なぜなら、次の危険性があるからです。実際に筆者もはまりました。

- スクリプトが予期せぬ挙動をしてしまう恐れがある。
- その挙動がRubyインタプリタのデフォルトの挙動だと思い込んでしまう。
- いつの間にか環境変数「RUBYOPT」の存在すら忘れてしまう。

スクリプト探索パスの設定は、他の環境変数が受け持ちます。

▶環境変数「RUBYLIB」

環境変数「RUBYLIB」を設定すると、この環境変数の値をRubyライブラリの探索パスの先頭に加えます。環境変数「RUBYLIB」には、自作のRubyライブラリを格納するディレクトリを指定しておくとよいでしょう。設定例は、上記の表の通りです。

▶環境変数「PATH」

環境変数「PATH」は、シェル・Rubyインタプリタ内外を問わず、コマンドを実行するときに検索するパスです。「-S」オプションを指定したときは、環境変数「PATH」で指定されたディレクトリからスクリプトを探索します。

▶環境変数「RUBYPATH」

環境変数「RUBYPATH」は、「-S」オプションを指定したときに、環境変数「PATH」よりも優先的にスクリプトを探索するパスです。実行用Rubyスクリプトのみのディレクトリを作成している人は設定しましょう。

Unix系OSの場合、スクリプトでもバイナリでも実行許可属性が付いていたら、そのファイルを、直接、実行できます。そのため、これは事実上、非Unix系OSのためのオプションです。

▶環境変数「RUBYSHELL」

環境変数「RUBYSHELL」は、Rubyインタプリタ内でコマンドを実行するときに使用するシェルを指定します。指定されていない場合はCOMSPECの値を使います。これはOS2版、mswin32版、mingw32版のみに有効な環境変数です。

関連項目 ▶▶▶

- Rubyの実行について …… p.37
- スクリプトを探索する順序について …… p.41

オブジェクト指向について

Rubyは純粋なオブジェクト指向プログラミング言語

Rubyは設計当初からオブジェクト指向を念頭に置いている言語です。また、Rubyのデータは、**すべてがクラスに属したオブジェクト**です。そのため、整数であろうがユーザ定義クラスのオブジェクトだろうが、分け隔てなく、メソッド呼び出しができます。

▶オブジェクト・クラス・メソッド

Rubyのオブジェクトは、特定のクラスに属しています。クラスとは、データがどういう種類のもので、どういう振る舞いをするかを決定付ける枠組です。オブジェクトの振る舞いは、メソッドで定義されます。メソッドはオブジェクトと結び付いたいわゆる関数のことで、実際にC++ではメンバ関数と呼ばれています。

たとえば、文字列"abcd"の長さを知るには「"abcd".length」と記述します。Stringオブジェクト"abcd"に「length」というメソッドを呼び出すという意味です。C言語だと「strlen("abcd");」となりますが、Rubyの場合は"abcd"が前に来ます。あくまで"abcd"が「主役」なのです。日本語や英語でも主語が前に来るように、(多くの)オブジェクト指向言語は、主体となるオブジェクトが前に来ます。この主体となるオブジェクトを「レシーバ」と呼びます。

また、クラスが異なれば、同じ名前のメソッドを持つことができます。たとえば、配列(つまりArrayオブジェクト)["a","b","c","d"]の長さを知るには、「["a","b","c","d"].length」と記述します。同じ長さを求めるメソッドでも、文字列の場合とはコンピュータから見て別の処理です。一方、人間から見て「長さを求める」というのは、オブジェクトが異なるだけで同じ意味です。そのため、同じメソッド名になっています。そう考えると、オブジェクト指向というのは、より人間的な考え方といえます。

▶ポリモーフィズム

逆にobjの長さを求めるために「obj.length」と記述した場合、objが文字列ならば「String#length」が、配列ならば「Array#length」が呼び出されます。オブジェクトの種類によって自動的に最適な処理(メソッド)が選ばれます。これをポリモーフィズムや多態と呼びます。そのおかげで、何がやりたいのかが明確なプログラムになります。また、別なクラスに同名のメソッドを用意することで、そのクラスにも対応できるようになります。

▶カプセル化

オブジェクトは内部構造をユーザから隠し、メソッド経由でしか操作できないようになっています。そのため、ユーザは内部構造を知らなくてもオブジェクトを使えます。また、開発者は振る舞いを保持したまま内部構造を自由に変更できます。

▶インスタンス

インスタンスとオブジェクトは同じ意味です。ただ、インスタンスには、クラスに属していることを強調するニュアンスがあります。たとえば、配列[1, 2]のことを「配列オブジェクト」とも「配列クラスのインスタンス」ともいいます。

クラスとインスタンスの関係を明確にしてください。「配列」がクラスで、[1, 2]がインスタンスです。[3, 4]もインスタンスです。このように、インスタンスはたくさん作成できます。

▶複数のインスタンス

複数のインスタンスを作成できることは大きなメリットです。テキストエディタを作成することを想像してみてください。1つのファイルのみしか編集できないエディタならば、「ファイルから読み込む」関数と「セーブする」関数を用意して、ファイルを読み込むバッファをグローバル変数にするかもしれません。オブジェクト指向プログラミングではバッファをクラスにして、「ファイルからバッファに読み込む」メソッド、「バッファの内容をセーブする」メソッドを作成します。複数のインスタンスを作成できるので、複数個のファイルを扱うのは容易です。それぞれのバッファはみな独立した状態を持っていて、特定のバッファに対してメソッドを呼び出せば、そのバッファのみに作用します。グローバル変数にアクセスする関数よりも、メソッドの方がはるかに強力であることは想像に難しくないでしょう。

▶継承とMix-in

クラスの継承とは、あるクラスに機能を追加したクラスを作ることです。Rubyでは、継承の元となるクラスを「スーパークラス」といいます。親クラス・基本クラス・基底クラスなどと表現している言語もあります。Rubyでは、継承されたクラスを「サブクラス」といいます。子クラス・派生クラス・導出クラスなどと表現している言語もあります。

サブクラスはスーパークラスのメソッドをすべて受け継ぎます。また、スーパークラスのメソッドをサブクラスで書き換えることもできます。

スーパークラスが複数ある継承のことを多重継承といいますが、混乱の元となるのでRubyでは意図的にサポートしていません。その代わり、モジュールから機能を追加する「Mix-in」があります。**クラス階層を表現する継承と、機能を共有するためのMix-inをうまく使い分ける**のが、Rubyにおける設計で重要なことです。Mix-inで多重継承の機能をほとんどをサポートできるので、「単一継承+Mix-in」がRuby流の多重継承といえるでしょう。

たとえば、「数値」クラスがあると、それを継承した「実数」クラスを作ることができます。「比較可能なオブジェクト」はモジュールとして表現しておいて「実数」クラスにMix-inし、「複素数」クラスにはMix-inしないでおきます（実際は絶対値で比較可能になっている）。「比較可能なオブジェクト」はクラス階層を横断して、「文字列」にMix-inすることもできます。

▶クラスもオブジェクト

Rubyではクラスさえもオブジェクトです。たとえば、文字列クラスを表すStringは、Classクラスのインスタンスです。Classクラスは、クラス全体の振る舞い方を定義しています。たとえば、オブジェクトは「new」メソッドで作成しますが、それはClassクラスで定義しているのです。「クラスもオブジェクト」なのは事実ですが、オブジェクト指向の初心者には混乱の元となるので、慣れないうちは頭の片隅に置いておく程度で構いません。

▶関数

厳密にいうと、Rubyには関数は存在しません。ただし、「func(1,2,3)」のようにレシーバを指定しないでメソッド呼び出しができます。どうしても関数にしか見えないので、それを関数的

メソッドとか、いっそのこと「関数」と呼んでいます。本書でもトップレベルで定義されたメソッドを関数と呼んでいます。

▶インスタンスメソッドとクラスメソッド

インスタンスメソッドとは、インスタンスと結び付いたメソッドです。「"abcd".length」は、Stringクラスのインスタンスメソッド「length」(「String#length」)を呼び出すという意味です。まずインスタンスありきです。なお、単に「メソッド」と表現した場合はインスタンスメソッドです。

クラスメソッドは、クラスと結び付いたメソッドです。クラスメソッドの場合はインスタンスを生成しなくてもいきなり使えます。たとえば、Fileクラスのクラスメソッド「join」(「File.join」)はパスを「/」でつなぐメソッドで、Fileオブジェクトとは無関係で使えます。クラスメソッドは、そのクラスと関係のある関数的メソッドを定義するのに使えます。もちろん、インスタンスを生成する「new」メソッドは、クラスメソッドです。クラスメソッドは、複数のコンストラクタを定義する場合にも使えます。

COLUMN 特異メソッド

人間社会には枠にとらわれない人間がいるように、Rubyのオブジェクトもクラスという枠を超えて個性を出すことができます。Rubyは、個々のオブジェクト専用のメソッドを定義できます。そういうメソッドを特異メソッドと呼びます。特異メソッドはクラスで定義されていないメソッドを定義したり、クラスで定義されているメソッドを上書きしたりすることができます。

クラスが振る舞いを定義していても特異メソッドで振る舞いを追加・変更できるので、Rubyの場合は他言語ほどクラスの権力は強くありません。

COLUMN クラスメソッド

クラスの特異メソッドはクラスメソッドです。しかし、クラスオブジェクトはClassクラスとModuleクラスのメソッド(「Module#attr」など)も受け付けます。それらはクラスの特異メソッドと混用できるので、クラスメソッドだという考え方もあります。

それでも「クラスメソッド」という言葉が出てくるときは、クラスの特異メソッドに焦点を絞っていることが多いので、本書でも「クラスメソッド=クラスの特異メソッド」という定義にします。

クラス階層について

すべてのデータは何らかのクラスに属している

Rubyのすべてのデータは、何らかのクラスに属したオブジェクトです。整数や文字列という基本的なデータも例外ではありません。

▶ Objectクラスはすべてのオブジェクトの源

Rubyはスーパークラスを1つしか持たない単一継承なので、クラス階層は単純な木構造になります。その始点がObjectクラスです。どのクラスでもスーパークラスをたどっていけば、必ずObjectクラスに到達します。Objectクラスはすべてのオブジェクトの源といえるでしょう。

Objectクラスは、Rubyのオブジェクトの一般的な振る舞いを定義しています。そのため、Objectのインスタンスメソッドは、どんなクラスのインスタンスにも適用できます。たとえば、オブジェクトごとに一意的に割り当てられているIDを得る「object_id」メソッドは、Objectクラスで定義されています。

▶ 組み込みクラスのクラス階層

Ruby 1.8のクラス階層は、次のようになります。インクルードしているモジュール(Mix-in)は「<>」で付記しています。Ruby 1.9ではRubyで記述されたクラスが組み込みになったり、Ruby VM(YARV)関係のクラスが追加されたりしています。

```
Object
 ├─Array <Enumerable>
 ├─Binding
 ├─Continuation
 ├─Data
 │  └─NameError::message
 ├─Dir <Enumerable>
 ├─Enumerator <Enumerable>
 ├─Exception
 ├─FalseClass
 ├─File::Stat <Comparable>
 ├─Hash <Enumerable>
 ├─IO <File::Constants, Enumerable>
 │  └─File <File::Constants, Enumerable>
 ├─MatchData
 ├─Method
 ├─Module
 │  └─Class
```

```
├NilClass
├Numeric <Comparable>
│ ├Float <Precision, Comparable>
│ └Integer <Precision, Comparable>
│   ├Bignum <Precision, Comparable>
│   └Fixnum <Precision, Comparable>
├Proc
├Process::Status
├Range <Enumerable>
├Regexp
├String <Enumerable, Comparable>(Ruby 1.9のStringはEnumerableではない)
├Struct <Enumerable>
│ └Struct::Tms <Enumerable>
├Symbol
├Thread
├ThreadGroup
├Time <Comparable>
├TrueClass
└UnboundMethod
```

クラス階層の詳細

それでは上記のクラス階層図を眺めてみましょう。

ClassはModuleでもある

任意の「クラス」はClassというクラスのインスタンスであることは、特筆すべき点の1つです。つまり、クラスそのものもオブジェクトとして扱うことができるということです。

さらに、Classのスーパークラスは Moduleです。そのため、Moduleで定義されているメソッドは、Classでも適用することができます。たとえば、インスタンス変数へアクセスするメソッドを作成する「Module#attr_accessor」は、モジュール定義でもクラス定義でも使用可能です。

FileはIOでもある

Fileのスーパークラスは IOです。IOは一般的な入出力を扱うクラスで、Fileはファイル入出力に特化したクラスです。「read」や「print」などの読み書きメソッドはIOで定義されていますが、当然、Fileでも適用することができます。

インクルードしているモジュール(Mix-in)

多くのクラスでは、EnumerableやComparableがインクルードされています。モジュールをインクルードすると、そのモジュールのメソッドも使用可能になります。

「each」メソッドが定義されているクラスにはEnumerableが、「<=>」メソッドが定義されているクラスにはComparableがインクルードされています。これらは、特定のメソッドが定義され

ているクラスを使いやすくパワーアップするモジュールです。また、クラス階層を横断してメソッドを追加します。

COLUMN Ruby 1.9ではBasicObjectがObjectのスーパークラス

Objectクラスがすべてのオブジェクトの源と述べましたが、実際は少し違っている部分があります。実は、Ruby 1.9では、ObjectクラスにBasicObjectというスーパークラスが存在します。BasicObjectクラスはごくわずかなメソッドしか定義されていない、いわば半人前のオブジェクトが属するクラスです。それを継承して、Rubyのオブジェクトとして使いものになるようにしたのがObjectクラスです。それでも、あくまで内部構造が少し変わっただけなので、通常のプログラミングではObjectがすべてのオブジェクトの源と考えて問題ありません。BasicObjectは、特殊な用途に使用することができます。

COLUMN 卵が先か鶏が先か

Classのスーパークラスは Moduleで、Moduleのスーパークラスは Objectであることはクラス階層により明らかです。一方、Objectクラスオブジェクトは Classのインスタンスです。

つまり、ObjectとClassは「卵が先か鶏が先か」という関係となっています。頭が混乱しそうな構造ですが、Rubyインタプリタは、この問題にうまく対処しています。興味がある方はRubyインタプリタのソースコード「object.c」の「Init_Object」関数を読んでみてください。

動的型付について

変数ではなくてオブジェクト自身が型を知っている

Rubyの変数には、どんなオブジェクトでも代入できます。

▶変数は名札

Rubyの変数には型がありません。その代わり、Rubyオブジェクトが自分自身の型を知っています。そのため、Rubyの変数には、どんなオブジェクトでも代入できます。代入という言葉は箱に入れるようなイメージを思い浮かべますが、実際は**貼ってはがせる名札を貼り付ける**ようなイメージです。

ただし、どんなオブジェクトでも代入できるからといって、何を代入してもそのプログラムが動くわけではありません。「型」に合ったオブジェクトを代入しないと、後でそのオブジェクトを使うときにエラーが出ます。

▶クラスの権力はさほど大きくない

Rubyにおける「型」を考えてみます。たとえば、「io.print("abcd\n")」というプログラムをエラーなく動かすことを考えてみます。これを見て、変数「io」には、IOオブジェクトやFileオブジェクトが入らないといけないと思う先入観は捨ててください。実は、変数「io」には、**1つの引数を取る「print」メソッドを持つオブジェクトならば、何でも代入できる**のです。それがRubyにおける「型」です。クラスではなく、振る舞いで型が決まる型付けを、「動的型付」や「duck typing」といいます。

そのため、IOオブジェクトやFileオブジェクト以外にも、StringIOオブジェクトも代入可能です。StringIOクラスはIOクラスのサブクラスではありません。単にIOクラスの振る舞いをまねしている文字列のクラスです。振る舞いをまねするとは、対象クラスのメソッドをすべて持ち、同じ引数で動作するようなクラスを作成することです。それどころか、文字列に特異メソッド「print」を定義したオブジェクトも条件を満たします。「同じインターフェイスを持つ」という表現もしますが、Javaのようにinterface宣言があるわけではありません。呼び出すメソッドに反応するかしないか、それがすべてなのです。

このようにRubyにおいて、クラスの力は絶対ではありません。それでは、なぜ、クラスを用意しているかというと、クラスはオブジェクト指向プログラミングにおける大切な思考のツールだからです。

▶動的型付の利点と欠点

動的型付は便利ですが、欠点がないわけでもありません。静的型付言語のようにコンパイル時の型チェックがなくなります。そのため、実行してみるまで変数に正しい型が代入されているのかがわかりません。

静的型付言語と動的型付言語は、個人の経験や背景の違いにより好みが分かれます。静的型付言語で変数にいちいち型を記述しないといけないのは面倒ですが、コンパイル時に型チェックが効く上に、型がわかる分、コードを読むのが楽です。反面、動的型付言語は型

を記述しなくていい分、素早くプログラムを記述することができますが、実行時でないと型チェックできず、巨大なプログラムを読むときは型を想像しないといけません。

Rubyはあくまでスクリプト言語なので、素早くコーディングできることが求められます。そのため、動的型付を選びました。型チェックが弱い欠点は、自動テスト（ユニットテストなど）でカバーしましょう。型がわからない問題は、変数名をわかりやすくするなどの工夫をしましょう。

COLUMN Rubyの学習方法

Rubyを学ぶ目的は、Ruby on Railsで格好いいWebアプリケーションを作りたい、Webからの情報収集を自動化したい、日常的な作業を効率化したい、システム管理をしたいなど、人それぞれだと思いますが、Rubyを実践で使うためには基礎をしっかり学ばないといけません。しっかりした基礎があれば、Rubyはあなたの大親友（公式サイトのキャッチコピー）になってくれます。

プログラミング言語を学習するはじめの一歩は文法です。手始めにリテラル、演算子、代入、変数、条件分岐、ループ、メソッド呼び出し、メソッド定義を学ぶと、Rubyでちょっと遊ぶことはできます。Rubyの感触をつかんだら、クラス定義、ブロック、Mix-in、特異メソッド、例外処理などの高度な文法を学びましょう。

Rubyの考え方については『初めてのRuby』（オライリー・ジャパン刊）という良書があります。考え方がしっかりしていれば、Rubyの世界にすんなり溶け込めます。

文法について理解したら、文字列、正規表現、配列、ハッシュという基本的なクラスを学びましょう。メソッドの使い方はirbやxmpfilterでサンプルをそのまま実行したり、改変したりして手になじむようにしましょう。大切なのは手を動かすことです。ただ説明を眺めるよりも、実際に実行して初めて身に付きます。教材は本書とリファレンスマニュアルです。それぞれ別の視点で書かれているので、双方のサンプルを実行してみるとよいでしょう。

クラスが異なっても同じ名前のメソッドがたくさん登場することに気付くでしょう。たとえば、文字列の長さを求めるメソッドと、配列の長さを求めるメソッドはともに「length」メソッドです。別々の名前ではなく、「長さを求めるのは『length』メソッド」とさえ覚えておけばいいので、覚えることが少なくて済みます。

ここまで学べば、ちょっとしたプログラムくらいはすぐに書けるようになります。途中で難解な文法やメソッドに出会うと思いますが、わからないところは無理せずに飛ばしてしまって構いません。難解なものは登場したときに覚えればいいです。どうしてもわからなければ、怖がらずにruby-listで質問してみましょう。

ブロックはRubyの花形機能なので《**ブロックの使用例**》(p.119)でブロックのありがたみについて解説しています。ブロックを愛するようになれば、Rubyプログラミングがぐっと楽しくなることうけあいです。

正規表現はテキスト処理には欠かせない大切な技術です。Rubyの学習と並行して正規表現も学習しておくと、新境地を開けます。他言語やエディタでも使われている上、今後、何十年と使われるであろう技術なので、正規表現の知識は大きな財産となります。

メソッドの表記方法について

メソッドの表記方法

Rubyのメソッドは、「インスタンスメソッド」と「特異メソッド」の2つに分けられます。クラスオブジェクトの特異メソッドを「クラスメソッド」といいます。Rubyでは、事実上、標準となっているメソッドの表記方法が存在します。ReFeやRIでドキュメントを参照する場合にも使用されるので、覚えておいてください。

▶インスタンスメソッドの表記方法

インスタンスメソッドは「クラス名#メソッド名」と表記します。たとえば、Arrayクラスの「each」インスタンスメソッドは「Array#each」となります。また、単に「メソッド」という場合は、インスタンスメソッドを指します。

▶クラスメソッドの表記方法

クラスメソッド、すなわちクラスオブジェクトの特異メソッドは、「クラス名.メソッド名」か「クラス名::メソッド名」と表記します。たとえば、Timeクラスの「now」クラスメソッドは、「Time.now」または「Time::now」となります。インスタンスメソッドの表記方法とは異なり、実際のコードでのクラスメソッドの呼び方のままなので覚えやすいと思います。

▶モジュール関数の表記方法

モジュール関数とは、モジュールのインスタンスメソッドであると同時に特異メソッドでもあるメソッドのことです。これについては特に表記方法が定まっていないようです。そのため、Mathモジュールの「sin」モジュール関数は、「Math.sin」とも「Math::sin」とも「Math#sin」とも表記されます。本書ではモジュール関数と明記した上で、「Math.sin」とクラスメソッドと同じ表記にしました。

▶組み込み関数の表記方法

組み込み関数は、実際はKernelのモジュール関数です。たとえば、組み込み関数「print」は「Kernel.print」とも「Kernel::print」とも「Kernel#print」とも表記されます。しかし、組み込み関数はプライベートメソッドでレシーバを付けることができないため、「Kernel#print」と表記されることが多くなります。本書でも「Kernel#print」と表記することにしました。

Ruby 1.9のエンコーディングについて

Ruby 1.9は多言語対応

Ruby 1.8からRuby 1.9への最大の変更点は、多言語化です。文字列、正規表現、シンボル、IOは個々にエンコーディング情報（文字コード）を保持しています。そのおかげで、複数の文字コードを正しく共存させることができます。

▶ エンコーディングはどこから来るのか

エンコーディングは、「IO」「文字列操作」から設定されます。

リテラルはそもそもスクリプトに記述されるものなので、「magic comment」でスクリプトのエンコーディングを正しく設定しておけば問題ありません。《スクリプトの文字コードを設定する》(p.38)を参照してください。

文字列操作によるエンコーディング付加はRubyが行ってくれるので、これも問題ありません。

IOは基本的にロケールエンコーディングですが、少しばかりややこしくなります。《Ruby 1.9のIOエンコーディングについて》(p.451)を参照してください。

▶ 使えるエンコーディング

Ruby 1.9で使えるエンコーディングは、日本語以外にもたくさんあります。「ruby -e 'puts encoding.name_list'」をシェルから実行すると、エンコーディングの正式名が出てきます。「ruby19 -e 'puts Encoding.aliases'」を実行すると別名が出てきます。

スクリプトで日本語を記述する場合は、EUC-JP、UTF-8、Shift_JIS、Windows-31Jを使います。これらはASCII互換エンコーディングです。Windows-31JはWindowsで使われているShift_JIS拡張なので、Windowsで日本語を含むスクリプトを記述するならばShift_JISよりもWindows-31Jを指定するべきです。

UTF-16BE、UTF-16LE、UTF-32BE、UTF-32LEはASCII互換エンコーディングではないため、スクリプトエンコーディングには使えません。それでも文字列操作やIOに使えます。

UTF-7、ISO-2022-JPはIOには使えるものの、文字列操作はできません（バイト列として扱われる）。これらはダミーエンコーディングと呼ばれています。ASCII互換とはみなされません。

他にもいくつか特別なエンコーディングがあります。US-ASCIIはASCII文字からなる文字列のエンコーディングです。デフォルトのスクリプトエンコーディングはUS-ASCIIです。ASCII-8BITはバイナリ文字列のエンコーディングです。日本語文字列と結合するためには、エンコーディングを設定する必要があります。

デフォルトの外部エンコーディング（default_external）、デフォルトの内部エンコーディング（default_internal）、ロケールエンコーディングは、それぞれ、「external」「internal」「locale」という名前で参照できます。

▶ エンコーディング変換

文字列を別のエンコーディングに変換した文字列を作成するには、「String#encode」を使

用します。変換後の文字列に置き換える「String#encode!」もあります。

```
# -*- coding: euc-jp -*-
s = "新機能"
s.encoding               # => #<Encoding:EUC-JP>
s.encode "Shift_JIS"     # Shift_JISに変換
s.encode "Windows-31J"   # Windows-31Jに変換
s.encode "UTF-8"         # UTF-8に変換
s.encode "iso-2022-jp"   # ISO-2022-JPに変換
# EUC-JPはASCII文字のみでは表現できないのでエラー
s.encode "US-ASCII" rescue $!
# => #<Encoding::UndefinedConversionError: "\xE6\x96\xB0" from UTF-8 to US-ASCII>
s.encode! "ISO-2022-JP"  # 破壊的に変換
s.encoding               # => #<Encoding:ISO-2022-JP (dummy)>
```

「String#encode」はバイト列も変更されますが、バイト列はそのままでエンコーディング情報のみを変更したいことがたまにあります。たとえば、ネットワークアクセスで後からcharsetに基づくエンコーディングを付加するケースです。その場合は、「String#force_encoding」を使用します。これは破壊的メソッドです。「force_encoding」は虚偽のエンコーディングも指定できてしまうため、使用には注意が必要です。なお、エンコーディング情報が正しいかどうかは、「String#valid_encoding?」で調べます。

```
# -*- coding: euc-jp -*-
ai = "あい"
ai.encoding                              # => #<Encoding:EUC-JP>
ai.valid_encoding?                       # => true
# バイナリとして扱う
ai.force_encoding "ASCII-8BIT"           # => "\xA4\xA2\xA4\xA4"
ai.encoding                              # => #<Encoding:ASCII-8BIT>
ai.valid_encoding?                       # => true
# 虚偽のエンコーディングを指定する(危険)
ai.force_encoding "UTF-8"
ai.valid_encoding?                       # => false
# バイナリ文字列からエンコーディングを設定する
"\xA4\xA2\xA4\xA4".force_encoding "EUC-JP" # => "あい"
```

▶ 文字列処理は基本的にエンコーディングが一致しないとだめ

結合・比較・正規表現マッチという文字列処理は、基本的に両者のエンコーディングが一致しないと「Encoding::CompatibilityError」例外が発生します。異なるエンコーディングの文字列を処理する場合は、「String#encode」などで明示的にエンコーディングを揃えてください。Rubyは暗黙の型変換をあまりしない主義なので、こういうスタンスを採りました。

```
# -*- coding: euc-jp -*-
euc  = "あ"
sjis = "い".encode "Shift_JIS"
```

```
euc + sjis rescue $!
# => #<Encoding::CompatibilityError: incompatible character encodings: EUC-JP and Shift_JIS>
# エンコーディングを揃える
euc + sjis.encode("EUC-JP")      # => "あい"
euc + sjis.encode(euc.encoding)  # => "あい"
euc + sjis.encode(__ENCODING__)  # => "あい"
```

▶ASCII文字だけからなる文字列は特別扱い

異なるエンコーディングの文字列処理はエラーになりますが、例外的にASCII文字だけで構成された文字列は処理可能です。それでも、ダミーエンコーディングの場合はエラーになります。

```
# -*- coding: euc-jp -*-
x = "ab".encode("Shift_JIS") + "cd".encode("UTF-8")  # => "abcd"
x.encoding                                           # => #<Encoding:Shift_JIS>
"ab".encode("ISO-2022-JP") + "cd" rescue $!
# => #<Encoding::CompatibilityError: incompatible character encodings: ISO-2022-JP and EUC-JP>
```

関連項目 ▶▶▶

- スクリプトの文字コードを設定する …… p.38
- Rubyでの日本語の扱いについて …… p.192
- 文字コードを変換する …… p.238
- Ruby 1.9のIOエンコーディングについて …… p.451

CHAPTER 02

基本的なツール

手軽な実験環境

Ruby電卓「irb」について

物事を始めるには準備が大切です。プログラミングにおいて、ツールの整備は欠かせません。まずは「irb」というRuby電卓から紹介します。

▶irbは電卓

Rubyをインストールしたら、irbが使えるようになっています。コマンドプロンプトから「irb」と打ってみてください。「irb(main):001:0>」というプロンプトが出てきます。これは電卓です。適当にRubyの式を入力してください。

```
$ irb -f
irb(main):001:0> 1
=> 1
irb(main):002:0> 1+3
=> 4
irb(main):003:0> "string"
=> "string"
irb(main):004:0> "12345".length
=> 5
irb(main):005:0> Math.cos(0)
=> 1.0
irb(main):006:0>
```

数値はそのまま

四則演算は普通にできる

「"」で囲むと文字列

文字列に「length」というメッセージを送ると長さが返る

三角関数の計算もできる

一目瞭然です。使い方を少し試すくらいならば、わざわざファイルにスクリプトを記述する必要はありません。次は変数に代入して式を計算する例です。

```
irb(main):006:0> a=1
=> 1
irb(main):007:0> b=7
=> 7
irb(main):008:0> c=10
=> 10
irb(main):009:0> a+b+c
=> 18
irb(main):010:0> a+b*c
=> 71
irb(main):012:0> a*b*c
=> 70
irb(main):013:0>
```

HINT

システムにGNU Readlineライブラリがインストールされているのならば、Ctrl+pを押すことで前に入力した式を取り出すことができます。

COLUMN irbでスクリプトを実行

実は、irbには、スクリプトファイル名を指定し、ファイルに書いてある行を1行ずつ実行する機能があります。本書ではスペースの都合上、xmpfilterを使うことにしますが、「irb --prompt simple」を実行しても、irbで手入力しても構いません。

```
$ cat irb-input.rb
1+2
"abcdef".upcase
exit
$ irb --prompt simple irb-input.rb
>> 1+2
=> 3
>> "abcdef".upcase
=> "ABCDEF"
>> exit
$
```

COLUMN 「Kernel#p」で式の値を表示する

「Kernel#p」は式の値を表示します。irbの最初の例で「Kernel#p」を使ったら、次のようになります。

```
p 1
p 1+3
p "string"
p "12345".length
p Math.cos(0)
# >> 1
# >> 4
# >> "string"
# >> 5
# >> 1.0
```

しかし、多数の「Kernel#p」を使うと、どの式の値か、ひと目ではわからなくなってしまいます。この問題を解決するのが、「xmpfilter」による注釈付けです（次ページのCOLUMN参照）。

COLUMN xmpfilterとエディタを使えば自動で再計算

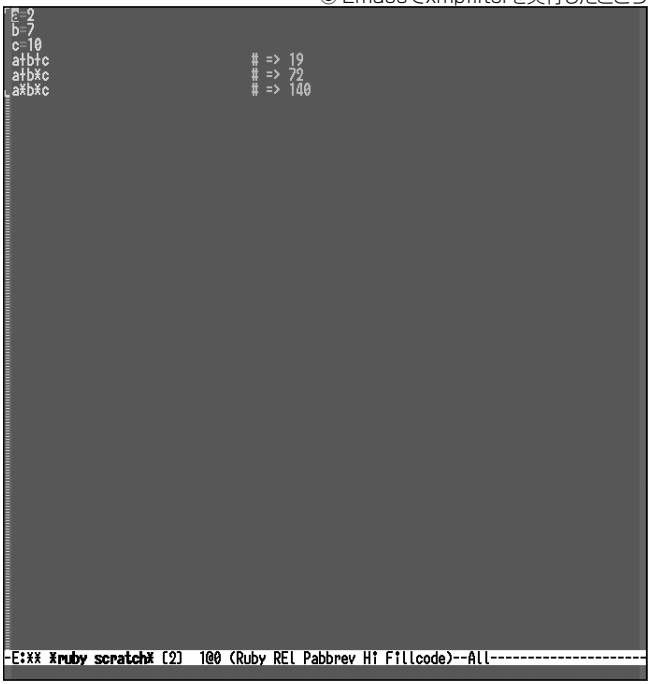
64ページの変数に代入して式を計算する例で、たとえば、aを2にして再計算する場合を考えてください。再び「a+b+c」などの3つの計算式を手作業で再実行する必要があり、面倒です。

この問題を解決するために、Mauricio Fernandez氏と筆者で「xmpfilter」というツールを開発しました。xmpfilterを利用すると、計算マーク「# =>」を付けたRubyスクリプトをxmpfilterに通すと、計算マークの後ろに計算結果が出てきます。実行するたびに再計算され、計算結果は更新されます。「http://raa.ruby-lang.org/list.rhtml?name=rcodetools」からダウンロードし、展開してsetup.rbを実行してください。RubyGemsからも「sudo gem install rcodetools」でインストールできます。本書に登場するRubyスクリプトはxmpfilterで注釈を付けました。

```
$ tar xzvf rcodetools-0.8.1.tar.gz
$ cd rcodetools-0.8.1
$ sudo ruby setup.rb
```

xmpfilterはエディタと併用することで真価を発揮します。そのため、EmacsとVimインターフェイスも付属しています。普通のフィルタプログラムなので、他のエディタに対応させるのは容易です。

◉ Emacsでxmpfilterを実行したところ



拡張パッケージを手軽にインストール

setup.rb

Rubyプログラム・ライブラリのインストールは、多くの場合、非常に簡単です。開発者のサイトからアーカイブをダウンロードすると、「setup.rb」が入っていることが多いでしょう。setup.rbは、「ruby setup.rb」を実行するだけでパッケージに拡張ライブラリがあればコンパイルし、実行スクリプトのshebangを書き換え、置かれるべき場所にファイルをインストールするインストーラです。Unix系OSの場合はroot権限が必要でしょうから、「sudo ruby setup.rb」と実行してください。

開発者にとって、Rubyスクリプトをリリースするときにsetup.rbを添付するのは、いわば礼儀となっています。

```
$ sudo ruby setup.rb
---> bin
<--- bin
---> lib
---> lib/fastri
<--- lib/fastri
<--- lib
---> bin
<--- bin
---> lib
---> lib/fastri
<--- lib/fastri
<--- lib
rm -f InstalledFiles
---> bin
mkdir -p /usr/local/bin/
install fastri-server /usr/local/bin/
install fri /usr/local/bin/
install ri-emacs /usr/local/bin/
install orig.diff /usr/local/bin/
install qri /usr/local/bin/
<--- bin
---> lib
mkdir -p /usr/local/lib/ruby/site_ruby/1.8/
---> lib/fastri
mkdir -p /usr/local/lib/ruby/site_ruby/1.8/fastri
install full_text_index.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
install util.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
install full_text_indexer.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
install version.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
install name_descriptor.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
```

```
install ri_service.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
install ri_index.rb /usr/local/lib/ruby/site_ruby/1.8/fastri
<--- lib/fastri
<--- lib
$
```

▶古いsetup.rbの場合

もし、添付されているsetup.rbが古い場合は、次の3つのコマンドを実行する必要があります。

```
$ ruby setup.rb config
$ ruby setup.rb setup
$ sudo ruby setup.rb install
```

RubyGems

RubyGemsはRuby版CPANです。世界中の人が公開しているRubyプログラム・ライブラリをコマンド一発でインストールすることができます。ユーザにとって、わざわざ開発者のサイトに行かなくて済む上、依存パッケージも同時にインストールしてくれるので、setup.rbよりも楽です。

▶Ruby 1.9には標準添付

RubyGemsはあまりにも多くの人に使われるようになったため、Ruby 1.9でついに標準添付されました。そのため、Ruby 1.9を使っている場合は、RubyGemsのインストールは不要です。

▶RubyGemsのインストール

「RubyGemsダウンロード」から最新版の「.tgz」か「.zip」をダウンロードして展開し、「sudo ruby setup.rb」を実行するだけです。詳細は「RubyGemsのインストール」を参照してください。

内　容	URL
RubyGemsダウンロード	http://rubyforge.org/frs/?group_id=126
RubyGemsユーザーズマニュアル	http://docs.rubygems.org/read/book/1
RubyGemsのインストール	http://docs.rubygems.org/read/chapter/3

▶gemのインストール

RubyGems用パッケージを「gem」といいます。拡張子も「.gem」です。たとえば、rakeをインストールしてみましょう。「sudo gem install rake」を実行します。

```
$ sudo gem install rake
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed rake-0.7.3
1 gem installed
Installing ri documentation for rake-0.7.3...
Installing RDoc documentation for rake-0.7.3...
$
```

さらに、gemはgithubでもよくリリースされているので「gem sources -a http://gems.github.com」も実行しておきましょう。これで自動的にgithubも探しに行きます。

▶どんなgemがあるのかを調べる

現在では、たくさんのgemがrubyforgeに登録されています。どんなgemがあるのか、調べてみたくなるものです。「gem query -bd」コマンドを実行すると、すべてのgemの名前、バージョン、概要が出力されます。とてつもなく長いので、ファイルにリダイレクトしてから見るようにしましょう。正規表現にマッチしたgemのみを出力することも可能ですが、gemコマンド自身が非常に重いのでファイルに落とすのが賢明です。

▶gemを使う

gemをインストールしたら、早速、使ってみましょう。

rakeのようにコマンドとして提供されているものは、そのままコマンドとして使えます。

ライブラリとして提供されているgemの場合、Ruby 1.8では使う前に「require 'rubygems'」が必要です。その後に提供されているライブラリをrequireしましょう。Ruby 1.9では、あらかじめ、RubyGemsの機能が組み込みになっているため、「require 'rubygems'」は不要です

▶RubyGemsのアップデート

一度、RubyGemsをインストールしたら、RubyGemsそのもののアップグレードは、非常に簡単です。「sudo gem update --system」のコマンド一発です。

COLUMN　RubyGemsの欠点

便利なRubyGemsですが、大きな欠点が2つあります。

1つは、gemを使うのにいちいちスクリプトで「require 'rubygems'」を加えないといけないことです。かなり面倒です。Rubyインタプリタに「-rubygems」というオプションを付ければ、スクリプト実行前にRubyGemsをロードしてくれます。この欠点はRuby 1.9で修正されました。

もう1つは、内部に複雑なカラクリがあるので「require 'rubygems'」自体にかなり時間がかかることです。fastRIなど、一瞬で実行が終了するスクリプトの場合、このオーバーヘッドが体感速度に影響します。起動時間を切り詰めないといけないCGIスクリプトでは使いものになりません。

RubyGems開発者たちもその欠点は痛いほどわかっていて、改善されつつあります。

COLUMN　gemコマンドの他の使い方

gemコマンドは多機能です。「gem help」を実行すると、ここでは紹介していない使い方がいろいろ出てきます。詳しくは「RubyGemsのユーザーズマニュアル」を参考にしてください。これは、非常に詳しいマニュアルです。

関連項目 ▶▶▶
● ワンライナーを極める……p.517

日本語でドキュメント引き

リファレンスマニュアルとReFe2

Rubyは日本語のマニュアルが充実しています。旧来の「Rubyリファレンスマニュアル」に加えて「Rubyリファレンスマニュアル刷新計画」があります。

▶ Rubyリファレンスマニュアル

旧来のRubyリファレンスマニュアルは「http://www.ruby-lang.org/ja/man/」にてオンラインで閲覧することができます。

◉ Rubyリファレンスマニュアル

▶ Rubyリファレンスマニュアル刷新計画

旧来のリファレンスマニュアルでは、システム上、メンテナンスするのが困難になってきたため、新しいシステムでマニュアルを再構成しています。それがRubyリファレンスマニュアル刷新計画、通称「るりま」です。

るりまではリファレンスマニュアルにアクセスするために、専用のWebサーバとコマンドライン検索ツールが用意されています。

現在のるりまでは、ライブラリのドキュメントに焦点を絞っているので、Rubyの概要や文法については旧来のRubyリファレンスマニュアルを参照してください。

▶ るりまのインストール

るりまをインストールするには、「http://www.ruby-lang.org/ja/man/archive/snapshot/ruby-refm-1.9.1-dynamic-snapshot.tar.bz2」からダウンロードして、適当なディレクトリに展開します。ここでは「~/compile」(WindowsではC:¥compile)以下に展開することにします。

そうすると、「~/compile/ruby-refm-1.9.1-dynamic-snapshot」というディレクトリが作成されます。このディレクトリをドキュメントディレクトリと呼ぶことにします。

```
$ cd ~/compile
$ wget http://www.ruby-lang.org/ja/man/archive/snapshot/ruby-refm-1.9.1-dynamic-snapshot.tar.bz2
$ tar xjf ruby-refm-1.9.1-dynamic-snapshot.tar.bz2
```

▶るりまWebインターフェイス

ドキュメントディレクトリにはserver.rbがあります。これはるりま専用Webサーバで、実行するとブラウザからるりまにアクセスすることができます。bitclustディレクトリ以下にあるserver.exeはserver.rbをWindows実行ファイルに変換したものなので、Windowsの人はserver.exeを実行しても構いません。ただし、執筆時では、server.exeを1つ上のディレクトリ(ドキュメントディレクトリ)に移動しないと動作しません。

server.rbが動かない場合は、ポート番号10080番が使用中です。そのときは「ruby server.rb --port=33432」のようにポート番号を変更してください。server.exeの場合も同様に、コマンドプロンプトから「server.exe --port=33432」としてください。

```
$ ruby server.rb
[2008-04-29 01:06:52] INFO  WEBrick 1.3.1
[2008-04-29 01:06:52] INFO  ruby 1.8.6 (2008-03-03) [i686-linux]
[2008-04-29 01:06:52] DEBUG TCPServer.new(127.0.0.1, 10080)
```

Webサーバを立ち上げたら、「http://127.0.0.1:10080」をブラウザでアクセスします。なお、Windowsの場合は自動でブラウザが立ち上がります。

◉るりまWebインターフェース

▶るりまを検索する準備

しかし、特定のメソッドの使い方を調べたい場合に、いちいちブラウザを開くのは面倒です。直接、「○○クラスの××メソッドの説明を出せ」と命令する方が楽です。

ドキュメントディレクトリの「bitclust」ディレクトリ以下に、ReFe2というリファレンスマニュアル検索ツールがあります。ReFe2はリファレンスマニュアルデータベースから特定の項目を閲覧するツールです。データベースはドキュメントディレクトリに、「db-1_8_7」「db-1_9_1」という名前で存在します（執筆時点）。

ただし、そのままだと使いにくいので、シェルスクリプトやバッチファイルを作成しましょう。当然、環境変数「PATH」の通ったディレクトリに作成します。「-C」ではドキュメントディレクトリ直下の「bitclust」ディレクトリを、「-d」ではリファレンスマニュアルデータベースを指定します。

Unix系OSでは、次の内容でシェルスクリプト「refe」を作成します。

```
#!/bin/sh
ruby -C ~/compile/ruby-refm-1.9.1-dynamic-snapshot/bitclust -Ke -Ilib bin/refe.rb \
-d ../db-1_9_1 "$@"
```

Windowsでは、次の内容で「refe.bat」を作成します。

```
@ruby -C C:¥compile¥ruby-refm-1.9.1-dynamic-snapshot¥bitclust -Ke -Ilib bin¥refe.rb ^
-d ..¥db-1_9_1 -e sjis %*
```

▶refeを使う

refeコマンドを作成したら、シェル（コマンドプロンプト）から自由にリファレンスマニュアルを検索することができます。refeコマンドの引数には、調べたいメソッド、クラス、組み込み変数を入力します。メソッドは、Ruby慣例の記法でクラス込みとともに指定することができます。つまり、クラスメソッドの場合は「.」で、インスタンスメソッドの場合は「#」でクラス名とメソッド名を区切ります。

なお、シェルによってはクォートやエスケープが必要なことがあります。たとえば、ほとんどのUnix系OSのシェルでは「refe $_」は「refe '$_'」と入力する必要があります。zshでは「refe File#read」は「refe File\#read」か「refe 'File#read'」と入力してください。

本書の解説はrefeコマンドでリファレンスマニュアルを調べられるように、Ruby慣例の記法でメソッド名を表記しています。本書では典型的な使い方のみしか解説していませんので、完全な解説はrefeでリファレンスマニュアルを引いてください。

```
$ refe Kernel.p
Kernel.p
--- p(*arg) -> nil

引数を人間に読みやすい形に整形して改行と順番に標準出力 [[m:$stdout]] に出力します。主にデバッグに使用します。

引数の inspect メソッドの返り値と改行を順番に出力します。つまり以下のコードと同じです。
```

```
  print arg[0].inspect, "\n", arg[1].inspect, "\n", ...

整形に用いられる[[m:Object#inspect]]は普通に文字列に変換すると
区別が付かなくなるようなクラス間の差異も表現できるように工夫されています。

p に引数を与えずに呼び出した場合は特に何もしません。

@param arg 出力するオブジェクトを任意個指定します。
@raise IOError 標準出力が書き込み用にオープンされていなければ発生します。
@raise Errno::EXXX 出力に失敗した場合に発生します。
@return nil を返します。

  puts "" #=> (空行)
  p "" #=> ""

  puts 50,"50"
  #=> 50
  #=> 50
  p 50,"50"
  #=> 50
  #=> "50"

@see [[m:Object#inspect]],[[m:Kernel.#puts]],[[m:Kernel.#print]]
```

コマンドの例を下表に示します。

コマンドの例	表示されるドキュメント
refe Kernel.p	組み込み関数p
refe IO.read	IOのreadクラスメソッド
refe Class	Classクラス
refe $_	組み込み変数$_

英語でドキュメント引き（高速版）

fastRIでドキュメント引き

Rubyをインストールすると、RIというドキュメント検索ツールが付いてきます。しかし、あまりにも遅くて使いものにならないので高速版が開発されました。それがfastRIです。fastRIは、執筆時点では、Ruby 1.9に対応していません。

▶インストール

「sudo gem install fastri」か、ホームページからパッケージをダウンロードして「sudo ruby setup.rb」を実行してください。RubyGems版はRubyGemsのオーバーヘッドが馬鹿にならないので、setup.rbをお勧めします。

- fastRIホームページ

  URL http://eigenclass.org/hiki.rb?fastri

▶準備

インストール後、「fastri-server -b」を実行してください。RIのデータベースが集められます。新たなパッケージをインストールしたときに、そのドキュメントを参照したい場合にもこのコマンドを実行する必要があります。

▶使い方

ドキュメントを検索するには、「qri」コマンドを使います。引数に調べたいメソッド名を入力します。クラス名込みで指定することができます。指定方法は、Rubyの慣例に従います。つまり、クラスメソッドの場合は「.」か「::」で、インスタンスメソッドの場合は「#」でクラス名とメソッド名を区切ります。

```
$ qri IO.readlines
------------------------------------------------------------ IO::readlines
     IO.readlines(name, sep_string=$/)   => array
------------------------------------------------------------------------
     Reads the entire file specified by name as individual lines, and
     returns those lines in an array. Lines are separated by sep_string.

        a = IO.readlines("testfile")
        a[0]   #=> "This is line one\n"
$ qri "IO#readlines"
------------------------------------------------------------ IO#readlines
     ios.readlines(sep_string=$/)  =>   array
------------------------------------------------------------------------
     Reads all of the lines in ios, and returns them in anArray. Lines
     are separated by the optional sep_string. If sep_string is nil,
     the rest of the stream is returned as a single record. The stream
     must be opened for reading or an IOError will be raised.
```

```
        f = File.new("testfile")
        f.readlines[0]   #=> "This is line one\n"
$
```

メソッド名のみ指定すると候補が出てきます。また、それらの部分文字列を指定することもできます。

```
$ qri readlin
------------------------------------------------------ Multiple choices:

     Buffering#readline, Buffering#readlines, CSV.readlines,
     DEBUGGER__::Context#readline, File.readlink, IO#readline,
     IO#readlines, IO.readlines, IRB::Locale#readline, Kernel#readline,
     Kernel#readlines, Net::Socket#readline, Pathname#readlines,
     Pathname#readlink, RAAInstall::StringIO#readline,
     RAAInstall::StringIO#readlines, StringIO#readline,
     StringIO#readlines, WWWsrv::Reader#readline,
     WWWsrv::Reader#readlines, Zlib::GzipReader#readline,
     Zlib::GzipReader#readlines
$
```

COLUMN その他の機能

fastRIの機能は、他にもいろいろあります。たとえば、dRubyサーバを使える人は「fastri-server」コマンドでfastRIを常駐させて、「qri」の代わりに「fri」コマンドでより高速に参照することができます。他にも、全文検索機能があります。詳しくはパッケージの説明を読んでください。

関連項目 ▶▶▶

● メソッドの表記方法について …… p.59