BioRuby の使い方

BioRuby は国産の高機能オブジェクト指向スクリプト言語 Ruby のための オープンソースなバイオインフォマティクス用ライブラリです。

Ruby 言語は Perl 言語ゆずりの強力なテキスト処理と、 シンプルで分かりやすい文法、クリアなオブジェクト指向機能により、 広く使われるようになりました。Ruby について詳しくは、ウェブサイト http://www.ruby-lang.org/ や市販の書籍等を参照してください。

はじめに

BioRuby を使用するには Ruby と BioRuby をインストールする必要があります。

Ruby のインストール

Ruby は Mac OS X や最近の UNIX には通常インストールされています。 Windows の場合も 1 クリックインストーラや ActiveScriptRuby などが 用意されています。まだインストールされていない場合は

- http://jp.rubyist.net/magazine/?0002-FirstProgramming
- http://jp.rubyist.net/magazine/?FirstStepRuby

などを参考にしてインストールしましょう。

あなたのコンピュータにどのバージョンの Ruby がインストールされているかを チェックするには

```
% ruby -v
```

とコマンドを入力してください。すると、たとえば

```
ruby 1.8.2 (2004-12-25) [powerpc-darwin7.7.0]
```

のような感じでバージョンが表示されます。バージョン 1.8.2 以降をお勧めします。

Ruby 標準装備のクラスやメソッドについては、Ruby のリファレンスマニュアルを 参照してください。

- http://www.ruby-lang.org/ja/man/

コマンドラインでヘルプを参照するには、Ruby 標準添付の ri コマンドや、 日本語版の refe コマンドが便利です。

- http://i.loveruby.net/ja/prog/refe.html

RubyGems のインストール

RubyGems のページから最新版をダウンロードします。

- http://rubyforge.org/projects/rubygems/

展開してインストールします。

```
% tar zxvf rubygems-x.x.x.tar.gz
% cd rubygems-x.x.x
% ruby setup.rb
```

BioRuby のインストール

BioRuby のインストール方法は http://bioruby.org/archive/ から 最新版を取得して以下のように行います。同梱されている README ファイルにも 目を通して頂きたいのですが、慣れないと 1 日がかりになる BioPerl と比べて BioRuby のインストールはすぐに終わるはずです。

```
% wget http://bioruby.org/archive/bioruby-x.x.x.tar.gz
% tar zxvf bioruby-x.x.x.tar.gz
% cd bioruby-x.x.x
% ruby install.rb config
% ruby install.rb setup
# ruby install.rb install
```

RubyGems が使える環境であれば

```
% gem install bio
```

だけでインストールできます。このあと README ファイルに書かれているように

```
bioruby-x.x.x/etc/bioinformatics/seqdatabase.ini
```

というファイルをホームディレクトリの ~/.bioinformatics にコピーして おくとよいでしょう。RubyGems の場合は

```
/usr/local/lib/ruby/gems/1.8/gems/bio-x.x.x/
```

などにあるはずです。

```
% mkdir ~/.bioinformatics
% cp bioruby-x.x.x/etc/bioinformatics/seqdatabase.ini ~/.bioinformatics
```

また、Emacs エディタを使う人は Ruby のソースに同梱されている misc/ruby-mode.el をインストールしておくとよいでしょう。

```
% mkdir -p ~/lib/lisp/ruby
% cp ruby-x.x.x/misc/ruby-mode.el ~/lib/lisp/ruby
```

などとしておいて、~/.emacs に以下の設定を書き足します。

```
; subdirs の設定
(let ((default-directory "~/lib/lisp"))
  (normal-top-level-add-subdirs-to-load-path)

; ruby-mode の設定
(autoload 'ruby-mode "ruby-mode" "Mode for editing ruby source files")
(add-to-list 'auto-mode-alist '("¥¥.rb$" . rd-mode))
(add-to-list 'interpeter-mode-alist '("ruby" . ruby-mode))
```

BioRuby シェル

BioRuby バージョン 0.7 以降では、簡単な操作は BioRuby と共にインストールされる bioruby コマンドで行うことができます。bioruby コマンドは Ruby に内蔵されている インタラクティブシェル irb を利用しており、Ruby と BioRuby にできることは全て 自由に実行することができます。

```
% bioruby project1
```

引数で指定した名前のディレクトリが作成され、その中で解析を行います。 上記の例の場合 project1 というディレクトリが作成され、さらに以下の サブディレクトリやファイルが作られます。

```
data/            ユーザの解析ファイルを置く場所
plugin/          必要に応じて追加のプラグインを置く場所
session/         設定やオブジェクト、ヒストリなどが保存される場所
session/config   ユーザの設定を保存したファイル
session/history ユーザの入力したコマンドのヒストリを保存したファイル
session/object   永続化されたオブジェクトの格納ファイル
```

このうち、data ディレクトリはユーザが自由に書き換えて構いません。 また、session/history ファイルを見ると、いつどのような操作を行ったかを 確認することができます。

２回目以降は、初回と同様に

```
% bioruby project1
```

として起動しても構いませんし、作成されたディレクトリに移動して

```
% cd project1
% bioruby
```

のように引数なしで起動することもできます。

この他、script コマンドで作成されるスクリプトファイルや、 web コマンドで作成される Rails のための設定ファイルなどがありますが、 それらについては必要に応じて後述します。

BioRuby シェルではデフォルトでいくつかの便利なライブラリを読み込んでいます。 例えば readline ライブラリが使える環境では Tab キーでメソッド名や変数名が 補完されるはずです。open-uri, pp, yaml なども最初から読み込まれています。

塩基, アミノ酸の配列を作る

seq(*str*)

seq コマンドを使って文字列から塩基配列やアミノ酸配列を作ることができます。 塩基とアミノ酸は ATGC の含量が 90% 以上かどうかで自動判定されます。 ここでは、できた塩基配列を dna という変数に代入します。

```
bioruby> dna = seq("atgcatgcaaaa")
```

変数の中身を確認するには Ruby の puts メソッドを使います。

```
bioruby> puts dna
atgcatgcaaaa
```

ファイル名を引数に与えると手元にあるファイルから配列を得ることもできます。 GenBank, EMBL, UniProt, FASTA など主要な配列フォーマットは自動判別されます （拡張子などのファイル名ではなくエントリの中身で判定します）。 以下は UniProt フォーマットのエントリをファイルから読み込んでいます。 この方法では、複数のエントリがある場合最初のエントリだけが読み込まれます。

```
bioruby> cdc2 = seq("p04551.sp")
bioruby> puts cdc2
MENYQKVEKIGEGTYGVVYKARHKLSGRIVAMKKIRLEDESEGVPSTAIREISLLKEVNDENNRSN...(略)
```

データベース名とエントリ名が分かっていれば、インターネットを通じて 配列を自動的に取得することができます。

```
bioruby> psaB = seq("genbank:AB044425")
bioruby> puts psaB
actgaccctgttcatattcgtcctattgctcacgcgatttgggatccgcactttggccaaccagca...(略)
```

どこのデータベースからどのような方法でエントリを取得するかは、BioPerl などと共通の OBDA 設定ファイル ~/.bioinformatics/seqdatabase.ini を用いてデータベースごとに指定することができます（後述）。 また、EMBOSS の seqret コマンドによる配列取得にも対応していますので、 EMBOSS の USA 表記でもエントリを取得できます。EMBOSS のマニュアルを参照し ~/.embossrc を適切に設定してください。

どの方法で取得した場合も、seq コマンドによって返される配列は、 DNA 配列のための Bio::Sequence::NA クラスか、アミノ酸配列のための Bio::sequence::AA クラスのどちらかのオブジェクトになります。

配列がどちらのクラスに属するかは Ruby の class メソッドを用いて

```
bioruby> p cdc2.class
Bio::Sequence::AA

bioruby> p psaB.class
Bio::Sequence::NA
```

のように調べることができます。自動判定が間違っている場合などには to_naseq, to_aaseq メソッドで強制的に変換できます。

```
bioruby> pep = dna.to_aaseq
bioruby> p pep.class
Bio::Sequence::AA
```

塩基配列やアミノ酸配列のクラスは Ruby の文字列クラスである String を 継承していますので、length で長さを調べたり、+ で足し合わせたり、* で 繰り返したりなど、Ruby の文字列に対して行える操作は全て利用可能です。 このような特徴はオブジェクト指向の強力な側面の一つと言えるでしょう。

```
bioruby> puts dna.length
12

bioruby> puts dna + dna
atgcatgcaaaaatgcatgcaaaa

bioruby> puts dna * 5
atgcatgcaaaaatgcatgcaaaaatgcatgcaaaaatgcatgcaaaaatgcatgcaaaa
```

complement

塩基配列の相補鎖配列を得るには塩基配列の complement メソッドを呼びます。

```
bioruby> puts dna.complement
ttttgcatgcat
```

translate

塩基配列をアミノ酸配列に翻訳するには translate メソッドを使います。 翻訳されたアミノ酸配列を pep という変数に代入してみます。

```
bioruby> pep = dna.translate
bioruby> puts pep
MHAK
```

フレームを変えて翻訳するには

```
bioruby> puts dna.translate(2)
CMQ
bioruby> puts dna.translate(3)
ACK
```

などとします。

molecular_weight

分子量は molecular_weight メソッドで表示されます。

```
bioruby> puts dna.molecular_weight
3718.66444

bioruby> puts pep.molecular_weight
485.605
```

seqstat(*seq*)

seqstat コマンドを使うと、組成などの情報も一度に表示されます。

```
bioruby> seqstat(dna)

* * * Sequence statistics * * *

5'->3' sequence   : atgcatgcaaaa
3'->5' sequence   : ttttgcatgcat
Translation   1   : MHAK
Translation   2   : CMQ
Translation   3   : ACK
Translation  -1   : FCMH
Translation  -2   : FAC
Translation  -3   : LHA
Length            : 12 bp
GC percent        : 33 %
Composition       : a -  6 ( 50.00 %)
                    c -  2 ( 16.67 %)
                    g -  2 ( 16.67 %)
                    t -  2 ( 16.67 %)
Codon usage       :

 *-----------------------------------------------*
 |       |               2nd               |      |
 |  1st  |---------------------------------| 3rd  |
 |       |  U    |  C    |  A    |  G    |      |
 |-------+-------+-------+-------+-------+------|
 | U   U |F  0.0%|S  0.0%|Y  0.0%|C  0.0%|  u   |
 | U   U |F  0.0%|S  0.0%|Y  0.0%|C  0.0%|  c   |
 | U   U |L  0.0%|S  0.0%|*  0.0%|*  0.0%|  a   |
 |  UUU  |L  0.0%|S  0.0%|*  0.0%|W  0.0%|  g   |
 |-------+-------+-------+-------+-------+------|
 |  CCCC |L  0.0%|P  0.0%|H 25.0%|R  0.0%|  u   |
 | C     |L  0.0%|P  0.0%|H  0.0%|R  0.0%|  c   |
 | C     |L  0.0%|P  0.0%|Q  0.0%|R  0.0%|  a   |
 |  CCCC |L  0.0%|P  0.0%|Q  0.0%|R  0.0%|  g   |
 |-------+-------+-------+-------+-------+------|
 |   A   |I  0.0%|T  0.0%|N  0.0%|S  0.0%|  u   |
 |  A A  |I  0.0%|T  0.0%|N  0.0%|S  0.0%|  c   |
 | AAAAA |I  0.0%|T  0.0%|K 25.0%|R  0.0%|  a   |
 | A   A |M 25.0%|T  0.0%|K  0.0%|R  0.0%|  g   |
 |-------+-------+-------+-------+-------+------|
 |  GGGG |V  0.0%|A  0.0%|D  0.0%|G  0.0%|  u   |
 | G     |V  0.0%|A  0.0%|D  0.0%|G  0.0%|  c   |
 | G GGG |V  0.0%|A 25.0%|E  0.0%|G  0.0%|  a   |
 |  GG G |V  0.0%|A  0.0%|E  0.0%|G  0.0%|  g   |
 *-----------------------------------------------*

Molecular weight  : 3718.66444
Protein weight    : 485.605
//
```

アミノ酸配列の場合は以下のようになります。

```
bioruby> seqstat(pep)

* * * Sequence statistics * * *

N->C sequence     : MHAK
Length            : 4 aa
Composition       : A Ala - 1 ( 25.00 %) alanine
                    H His - 1 ( 25.00 %) histidine
                    K Lys - 1 ( 25.00 %) lysine
                    M Met - 1 ( 25.00 %) methionine
```

```
Protein weight    : 485.605
//
```

composition

seqstat の中で表示されている組成は composition メソッドで得ることができます。 結果が文字列ではなく Hash で返されるので、とりあえず表示してみる場合には puts の代わりに p コマンドを使うと良いでしょう。

```
bioruby> p dna.composition
{"a"=>6, "c"=>2, "g"=>2, "t"=>2}
```

塩基配列、アミノ酸配列のその他のメソッド

他にも塩基配列、アミノ酸配列に対して行える操作は色々とあります。

subseq(from, to)

部分配列を取り出すには subseq メソッドを使います。

```
bioruby> puts dna.subseq(1, 3)
atg
```

Ruby など多くのプログラミング言語の文字列は 1 文字目を 0 から数えますが、 subseq メソッドは 1 から数えて切り出せるようになっています。

```
bioruby> puts dna[0, 3]
atg
```

Ruby の String クラスが持つ slice メソッド str[] と適宜使い分けると よいでしょう。

window_search(len, step)

window_search メソッドを使うと長い配列の部分配列毎の繰り返しを 簡単に行うことができます。DNA 配列をコドン毎に処理する場合、 ３文字ずつずらしながら３文字を切り出せばよいので以下のようになります。

```
bioruby> dna.window_search(3, 3) do |codon|
bioruby+   puts "#{codon}¥t#{codon.translate}"
bioruby+ end
atg     M
cat     H
gca     A
aaa     K
```

ゲノム配列を、末端 1000bp をオーバーラップさせながら 11000bp ごとに ブツ切りにし FASTA フォーマットに整形する場合は以下のようになります。

```
bioruby> seq.window_search(11000, 10000) do |subseq|
bioruby+   puts subseq.to_fasta
bioruby+ end
```

最後の 10000bp に満たない 3' 端の余り配列は返り値として得られるので、 必要な場合は別途受け取って表示します。

```
bioruby> i = 1
bioruby> remainder = seq.window_search(11000, 10000) do |subseq|
bioruby>   puts subseq.to_fasta("segment #{i*10000}", 60)
bioruby>   i += 1
bioruby> end
bioruby> puts remainder.to_fasta("segment #{i*10000}", 60)
```

splicing(position)

塩基配列の GenBank 等の position 文字列による切り出しは splicing メソッドで行います。

```
bioruby> puts dna
atgcatgcaaaa
bioruby> puts dna.splicing("join(1..3,7..9)")
atggca
```

randomize

randomize メソッドは、配列の組成を保存したままランダム配列を生成します。

```
bioruby> puts dna.randomize
agcaatagatac
```

to_re

to_re メソッドは、曖昧な塩基の表記を含む塩基配列を atgc だけの パターンからなる正規表現に変換します。

```
bioruby> ambiguous = seq("atgcyatgcatgcatgc")

bioruby> p ambiguous.to_re
/atgc[tc]atgcatgcatgc/

bioruby> puts ambiguous.to_re
(?-mix:atgc[tc]atgcatgcatgc)
```

seq メソッドは ATGC の含有量が 90% 以下だとアミノ酸配列とみなすので、 曖昧な塩基が多く含まれる配列の場合は to_naseq メソッドを使って 明示的に Bio::Sequence::NA オブジェクトに変換する必要があります。

```
bioruby> s = seq("atgcrywskmbvhdn").to_naseq
bioruby> p s.to_re
/atgc[ag][tc][at][gc][tg][ac][tgc][agc][atc][atg][atgc]/

bioruby> puts s.to_re
(?-mix:atgc[ag][tc][at][gc][tg][ac][tgc][agc][atc][atg][atgc])
```

names

あまり使うことはありませんが、配列を塩基名やアミノ酸名に変換する メソッドです。

```
bioruby> p dna.names
["adenine", "thymine", "guanine", "cytosine", "adenine", "thymine",
"guanine", "cytosine", "adenine", "adenine", "adenine", "adenine"]

bioruby> p pep.names
["methionine", "histidine", "alanine", "lysine"]
```

codes

アミノ酸配列を３文字コードに変換する names と似たメソッドです。

```
bioruby> p pep.codes
["Met", "His", "Ala", "Lys"]
```

gc_percent

塩基配列の GC 含量は gc_percent メソッドで得られます。

```
bioruby> p dna.gc_percent
33
```

to_fasta

FASTA フォーマットに変換するには to_fasta メソッドを使います。

```
bioruby> puts dna.to_fasta("dna sequence")
>dna sequence
aaccggttacgt
```

塩基やアミノ酸のコード、コドン表をあつかう

アミノ酸、塩基、コドンテーブルを得るための aminoacids, nucleicacids, codontables, codontable コマンドを紹介します。

aminoacids

アミノ酸の一覧は aminoacids コマンドで表示できます。

```
bioruby> aminoacids
?       Pyl     pyrrolysine
A       Ala     alanine
B       Asx     asparagine/aspartic acid
C       Cys     cysteine
D       Asp     aspartic acid
E       Glu     glutamic acid
F       Phe     phenylalanine
G       Gly     glycine
H       His     histidine
I       Ile     isoleucine
K       Lys     lysine
L       Leu     leucine
M       Met     methionine
N       Asn     asparagine
P       Pro     proline
Q       Gln     glutamine
R       Arg     arginine
S       Ser     serine
T       Thr     threonine
U       Sec     selenocysteine
V       Val     valine
W       Trp     tryptophan
Y       Tyr     tyrosine
Z       Glx     glutamine/glutamic acid
```

返り値は短い表記と対応する長い表記のハッシュになっています。

```
bioruby> aa = aminoacids
bioruby> puts aa["G"]
Gly
bioruby> puts aa["Gly"]
glycine
```

nucleicacids

塩基の一覧は nucleicacids コマンドで表示できます。

```
bioruby> nucleicacids
a       a       Adenine
t       t       Thymine
g       g       Guanine
c       c       Cytosine
u       u       Uracil
```

```
r       [ag]    puRine
y       [tc]    pYrimidine
w       [at]    Weak
s       [gc]    Strong
k       [tg]    Keto
m       [ac]    aroMatic
b       [tgc]   not A
v       [agc]   not T
h       [atc]   not G
d       [atg]   not C
n       [atgc]
```

返り値は塩基の 1 文字表記と該当する塩基のハッシュになっています。

```
bioruby> na = nucleicacids
bioruby> puts na["r"]
[ag]
```

codontables

コドンテーブルの一覧は codontables コマンドで表示できます。

```
bioruby> codontables
1       Standard (Eukaryote)
2       Vertebrate Mitochondrial
3       Yeast Mitochondorial
4       Mold, Protozoan, Coelenterate Mitochondrial and Mycoplasma/Spiroplasma
5       Invertebrate Mitochondrial
6       Ciliate Macronuclear and Dasycladacean
9       Echinoderm Mitochondrial
10      Euplotid Nuclear
11      Bacteria
12      Alternative Yeast Nuclear
13      Ascidian Mitochondrial
14      Flatworm Mitochondrial
15      Blepharisma Macronuclear
16      Chlorophycean Mitochondrial
21      Trematode Mitochondrial
22      Scenedesmus obliquus mitochondrial
23      Thraustochytrium Mitochondrial
```

返り値はテーブル番号と名前のハッシュになっています。

```
bioruby> ct = codontables
bioruby> puts ct[3]
Yeast Mitochondorial
```

codontable(*num*)

コドン表自体は codontable コマンドで表示できます。

```
bioruby> codontable(11)

 = Codon table 11 : Bacteria

   hydrophilic: H K R (basic), S T Y Q N S (polar), D E (acidic)
   hydrophobic: F L I M V P A C W G (nonpolar)

 *---------------------------------------------*
 |       |              2nd              |     |
 |  1st  |-------------------------------| 3rd |
 |       |  U    |  C    |  A    |  G    |     |
 |-------+-------+-------+-------+-------+-----|
 | U   U | Phe F | Ser S | Tyr Y | Cys C |  u  |
 | U   U | Phe F | Ser S | Tyr Y | Cys C |  c  |
```

```
  | U   U | Leu L | Ser S | STOP  | STOP  |  a  |
  |  UUU  | Leu L | Ser S | STOP  | Trp W |  g  |
  |-------+-------+-------+-------+-------+-----|
  |  CCCC | Leu L | Pro P | His H | Arg R |  u  |
  | C     | Leu L | Pro P | His H | Arg R |  c  |
  | C     | Leu L | Pro P | Gln Q | Arg R |  a  |
  |  CCCC | Leu L | Pro P | Gln Q | Arg R |  g  |
  |-------+-------+-------+-------+-------+-----|
  |   A   | Ile I | Thr T | Asn N | Ser S |  u  |
  |  A A  | Ile I | Thr T | Asn N | Ser S |  c  |
  | AAAAA | Ile I | Thr T | Lys K | Arg R |  a  |
  | A   A | Met M | Thr T | Lys K | Arg R |  g  |
  |-------+-------+-------+-------+-------+-----|
  |  GGGG | Val V | Ala A | Asp D | Gly G |  u  |
  | G     | Val V | Ala A | Asp D | Gly G |  c  |
  | G GGG | Val V | Ala A | Glu E | Gly G |  a  |
  |  GG G | Val V | Ala A | Glu E | Gly G |  g  |
 *-----------------------------------------------*
```

返り値は Bio::CodonTable クラスのオブジェクトで、コドンとアミノ酸の 変換ができるだけでなく、以下のようなデータも得ることができます。

```
bioruby> ct = codontable(2)
bioruby> p ct["atg"]
"M"
```

definition

コドン表の定義の説明

```
bioruby> puts ct.definition
Vertebrate Mitochondrial
```

start

開始コドン一覧

```
bioruby> p ct.start
["att", "atc", "ata", "atg", "gtg"]
```

stop

終止コドン一覧

```
bioruby> p ct.stop
["taa", "tag", "aga", "agg"]
```

revtrans

アミノ酸をコードするコドンを調べる

```
bioruby> p ct.revtrans("V")
["gtc", "gtg", "gtt", "gta"]
```

フラットファイルのエントリ

データベースのエントリと、フラットファイルそのものを扱う方法を紹介します。 GenBank データベースの中では、ファージのエントリが含まれる gbphg.seq の ファイルサイズが小さいので、このファイルを例として使います。

```
% wget ftp://ftp.hgc.jp/pub/mirror/ncbi/genbank/gbphg.seq.gz
% gunzip gbphg.seq.gz
```

ent(*str*)

seq コマンドは配列を取得しましたが、配列だけでなくエントリ全体を取得するには ent コマンドを使います。seq コマンド同様、ent コマンドでも OBDA, EMBOSS, KEGG API のデータベースが利用可能です。設定については seq コマンドの説明を 参照してください。

```
bioruby> entry = ent("genbank:AB044425")
bioruby> puts entry
LOCUS       AB044425                1494 bp    DNA     linear   PLN 28-APR-2001
DEFINITION  Volvox carteri f. kawasakiensis chloroplast psaB gene for
            photosystem I P700 chlorophyll a apoprotein A2,
            strain:NIES-732.
(略)
```

ent コマンドの引数には db:entry_id 形式の文字列、EMBOSS の USA、 ファイル、IO が与えられ、データベースの 1 エントリ分の文字列が返されます。 配列データベースに限らず、数多くのデータベースエントリに対応しています。

flatparse(*str*)

取得したエントリをパースして欲しいデータをとりだすには flatparse コマンドを使います。

```
bioruby> entry = ent("gbphg.seq")
bioruby> gb = flatparse(entry)
bioruby> puts gb.entry_id
AB000833
bioruby> puts gb.definition
Bacteriophage Mu DNA for ORF1, sheath protein gpL, ORF2, ORF3, complete cds.
bioruby> puts psaB.naseq
acggtcagacgtttggcccgaccaccgggatgaggctgacgcaggtcagaaatctttgtgacgacaaccgtatcaat
(略)
```

obj(*str*)

obj コマンドは、ent でエントリを文字列として取得し flatparse でパースした オブジェクトに変換するのと同じです。ent コマンドと同じ引数を受け付けます。 配列を取得する時は seq、エントリを取得する時は ent、パースしたオブジェクトを 取得する時は obj を使うことになります。

```
bioruby> gb = obj("gbphg.seq")
bioruby> puts gb.entry_id
AB000833
```

flatfile(*file*)

ent コマンドは 1 エントリしか扱えないため、ローカルのファイルを開いて 各エントリ毎に処理を行うには flatfile コマンドを使います。

```
bioruby> flatfile("gbphg.seq") do |entry|
bioruby+   # do something on entry
bioruby+ end
```

ブロックを指定しない場合は、ファイル中の最初のエントリを取得します。

```
bioruby> entry = flatfile("gbphg.seq")
bioruby> gb = flatparse(entry)
bioruby> puts gb.entry_id
```

flatauto(*file*)

各エントリを flatparse と同様にパースした状態で順番に処理するためには、 flatfile コマンドの代わりに flatauto コマンドを使います。

```
bioruby> flatauto("gbphg.seq") do |entry|
bioruby+   print entry.entry_id
bioruby+   puts  entry.definition
bioruby+ end
```

flatfile 同様、ブロックを指定しない場合は、ファイル中の最初のエントリを 取得し、パースしたオブジェクトを返します。

```
bioruby> gb = flatfile("gbphg.seq")
bioruby> puts gb.entry_id
```

フラットファイルのインデクシング

EMBOSS の dbiflat に似た機能として、BioRuby, BioPerl などに共通の BioFlat というインデックスを作成する仕組みがあります。一度インデックスを 作成しておくとエントリの取り出しが高速かつ容易に行えます。これにより自分専用のデータベースを手軽に作ることができます。

`flatindex(db_name, *source_file_list)`

GenBank のファージの配列ファイル gbphg.seq に入っているエントリに対して mydb というデータベース名でインデックスを作成します。

```
bioruby> flatindex("mydb", "gbphg.seq")
Creating BioFlat index (.bioruby/bioflat/mydb) ... done
```

`flatsearch(db_name, entry_id)`

作成した mydb データベースからエントリをとり出すには flatsearch コマンドを 使います。

```
bioruby> entry = flatsearch("mydb", "AB004561")
bioruby> puts entry
LOCUS       AB004561                2878 bp    DNA     linear   PHG 20-MAY-1998
DEFINITION  Bacteriophage phiU gene for integrase, complete cds, integration
            site.
ACCESSION   AB004561
(略)
```

様々な DB の配列を FASTA フォーマットに変換して保存

FASTA フォーマットは配列データで標準的に用いられているフォーマットです。 「>」記号ではじまる 1 行目に配列の説明があり、２行目以降に配列がつづきます。 配列中の空白文字は無視されます。

```
>entry_id definition ...
ACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGT
ACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGT
```

配列の説明行は、最初の単語が配列の ID になっていることが多いのですが、 NCBI の BLAST 用データベースではさらに高度な構造化がおこなわれています。

- ftp://ftp.ncbi.nih.gov/blast/documents/README.formatdb
- http://blast.wustl.edu/doc/FAQ-Indexing.html#Identifiers
- FASTA format (Wikipedia) http://en.wikipedia.org/wiki/Fasta_format

BioRuby のデータベースエントリのクラスにはエントリID、配列、定義について 共通のメソッドが用意されています。

- entry_id - エントリ ID を取得
- definition - 定義文を取得
- seq - 配列を取得

これらの共通メソッドを使うと、どんな配列データベースエントリでも FASTA フォーマットに変換できるプログラムが簡単に作れます。

```
entry.seq.to_fasta("#{entry.entry_id} #{entry.definition}", 60)
```

さらに、BioRuby では入力データベースの形式を自動判別できますので、 GenBank, UniProt など多くの主要な配列データベースでは ファイル名を指定するだけで FASTA フォーマットに変換できます。

flatfasta(*fasta_file*, **source_file_list*)

入力データベースのファイル名のリストから、指定した FASTA フォーマットの ファイルを生成するコマンドです。ここではいくつかの GenBank のファイルを FASTA フォーマットに変換し、myfasta.fa というファイルに保存しています。

```
bioruby> flatfasta("myfasta.fa", "gbphg.seq", "gbvrl1.seq", "gbvrl2.seq")
Saving fasta file (myfasta.fa) ...
  converting -- gbphg.gbk
  converting -- gbvrl1.gbk
  converting -- gbvrl2.gbk
done
```

KEGG API

BioRuby シェルでは KEGG API のウェブサービスも簡単に利用できます。

keggdbs

ゲノムネットで KEGG API を通じて利用可能なデータベースのリストを表示します。

```
bioruby> keggdbs
nt:     Non-redundant nucleic acid sequence database
aa:     Non-redundant protein sequence database
gb:     GenBank nucleic acid sequence database
(略)
```

keggorgs

KEGG に収録されている全生物種のリストを表示します。

```
bioruby> keggorgs
aae:    Aquifex aeolicus
aci:    Acinetobacter sp. ADP1
afu:    Archaeoglobus fulgidus
(略)
```

keggpathways

KEGG に収録されている全パスウェイのリストを表示します。

```
bioruby> keggpathways
path:map00010:  Glycolysis / Gluconeogenesis - Reference pathway
path:map00020:  Citrate cycle (TCA cycle) - Reference pathway
path:map00030:  Pentose phosphate pathway - Reference pathway
(略)
```

引数に３文字の KEGG 生物種記号をあたえると、その生物で利用できる パスウェイだけの一覧を返します。大腸菌 eco の場合以下のようになります。

```
bioruby> keggpathways("eco")
path:eco00010:  Glycolysis / Gluconeogenesis - Escherichia coli K-12 MG1655
path:eco00020:  Citrate cycle (TCA cycle) - Escherichia coli K-12 MG1655
path:eco00030:  Pentose phosphate pathway - Escherichia coli K-12 MG1655
(略)
```

keggapi

これら以外の KEGG API のメソッドは、keggapi に続けて呼び出すことで 利用できます。

```
bioruby> p keggapi.get_genes_by_pathway("path:eco00010")
["eco:b0114", "eco:b0115", "eco:b0116", "eco:b0356", "eco:b0688", (略)
```

利用可能なメソッドの一覧は KEGG API のマニュアルを参照してください。

- http://www.genome.jp/kegg/soap/doc/keggapi_manual_ja.html

DBGET

ゲノムネットの DBGET のコマンドである binfo, bfind, bget, btit, bconv は KEGG API を利用してそのまま実行できるようになっています。

binfo

```
bioruby> binfo
                        *** Last database updates ***
Date      Database              Release              #Entries      #Residues
```

```
--------  -------------   ------------------------  ------------  ----------------
 05/12/06  nr-nt                 05-12-04 (Dec 05)    63,078,043   111,609,773,616
 05/12/06  nr-aa                 05-12-05 (Dec 05)     2,682,790       890,953,839
 05/10/25  genbank                  150.0 (Oct 05)    49,152,445    53,655,236,500
 05/12/06  genbank-upd       150.0+/12-04 (Dec 05)     7,470,976     6,357,888,366
 (略)
```

binfo コマンドに続けてデータベース名を指定することでより詳細な情報が 表示されます。

```
bioruby> binfo "genbank"
genbank          GenBank nucleic acid sequence database
gb               Release 150.0, Oct 05
                 National Center for Biotechnology Information
                 49,152,445 entries, 53,655,236,500 bases
                 Last update:  05/10/25
                 <dbget> <fasta> <blast>
```

bfind(*keyword*)

bfind コマンドでデータベースに対するキーワードサーチを行うことができます。 データベース名と検索したいキーワードを文字列で渡します。

```
bioruby> list = bfind "genbank ebola human"
bioruby> puts list
gb:BD177378 [BD177378] A monoclonal antibody recognizing ebola virus.
gb:BD177379 [BD177379] A monoclonal antibody recognizing ebola virus.
(略)
```

bget(*entry_id*)

bget コマンドで指定した db:entry_id のデータベースエントリを取得できます。

```
bioruby> entry = bget "gb:BD177378"
bioruby> puts entry
LOCUS       BD177378                  24 bp    DNA     linear   PAT 16-APR-2003
DEFINITION  A monoclonal antibody recognizing ebola virus.
(略)
```

スクリプト生成

作業手順をスクリプト化して保存しておくこともできます。

```
bioruby> script
-- 8< -- 8< -- 8< --  Script  -- 8< -- 8< -- 8< --
bioruby> seq = seq("gbphg.seq")
bioruby> p seq
bioruby> p seq.translate
bioruby> script
-- >8 -- >8 -- >8 --  Script  -- >8 -- >8 -- >8 --
Saving script (script.rb) ... done
```

生成された script.rb は以下のようになります。

```
#!/usr/bin/env bioruby

seq = seq("gbphg.seq")
p seq
p seq.translate
```

このスクリプトは bioruby コマンドで実行することができます。

```
% bioruby script.rb
```

簡易シェル機能

cd(*dir*)

カレントディレクトリを変更します。

```
bioruby> cd "/tmp"
"/tmp"
```

ホームディレクトリに戻るには引数をつけずに cd を実行します。

```
bioruby> cd
"/home/k"
```

pwd

カレントディレクトリを表示します。

```
bioruby> pwd
"/home/k"
```

dir

カレントディレクトリのファイルを一覧表示します。

```
bioruby> dir
   UGO  Date                              Byte  File
------  ----------------------------  ----------  ------------
 40700  Tue Dec 06 07:07:35 JST 2005        1768  "Desktop"
 40755  Tue Nov 29 16:55:20 JST 2005        2176  "bin"
100644  Sat Oct 15 03:01:00 JST 2005    42599518  "gbphg.seq"
(略)

bioruby> dir "gbphg.seq"
   UGO  Date                              Byte  File
------  ----------------------------  ----------  ------------
100644  Sat Oct 15 03:01:00 JST 2005    42599518  "gbphg.seq"
```

head(*file, lines = 10*)

テキストファイルやオブジェクトの先頭 10 行を表示します。

```
bioruby> head "gbphg.seq"
GBPHG.SEQ            Genetic Sequence Data Bank
                          October 15 2005

                NCBI-GenBank Flat File Release 150.0

                          Phage Sequences

    2713 loci,    16892737 bases, from     2713 reported sequences
```

表示する行数を指定することもできます。

```
bioruby> head "gbphg.seq", 2
GBPHG.SEQ            Genetic Sequence Data Bank
                          October 15 2005
```

テキストの入っている変数の先頭を見ることもできます。

```
bioruby> entry = ent("gbphg.seq")
bioruby> head entry, 2
GBPHG.SEQ            Genetic Sequence Data Bank
                          October 15 2005
```

disp(*obj*)

テキストファイルやオブジェクトの中身をページャーで表示します。 ここで使用するページャーは pager コマンドで変更することができます（後述）。

```
bioruby> disp "gbphg.seq"
bioruby> disp entry
bioruby> disp [1, 2, 3] * 4
```

変数

ls

セッション中に作成した変数（オブジェクト）の一覧を表示します。

```
bioruby> ls
["entry", "seq"]

bioruby> a = 123
["a", "entry", "seq"]
```

rm(*symbol*)

変数を消去します。

```
bioruby> rm "a"

bioruby> ls
["entry", "seq"]
```

savefile(*filename*, *object*)

変数に保存されている内容をテキストファイルに保存します。

```
bioruby> savefile "testfile.txt", entry
Saving data (testfile.txt) ... done

bioruby> disp "testfile.txt"
```

各種設定

永続化の仕組みとして BioRuby シェル終了時に session ディレクトリ内に ヒストリ、オブジェクト、個人の設定が保存され、次回起動時に自動的に 読み込まれます。

config

BioRuby シェルの各種設定を表示します。

```
bioruby> config
message = "...BioRuby in the shell..."
marshal = [4, 8]
color   = false
pager   = nil
echo    = false
```

echo 表示するかどうかを切り替えます。on の場合は、puts や p などを つけなくても評価した値が画面に表示されます。 irb コマンドの場合は初期設定が on になっていますが、bioruby コマンドでは 長い配列やエントリなど長大な文字列を扱うことが多いため、初期設定では off にしています。

```
bioruby> config :echo
Echo on
  ==> nil

bioruby> config :echo
Echo off
```

コドン表など、可能な場合にカラー表示するかどうかを切り替えます。 カラー表示の場合、プロンプトにも色がつきますので判別できます。

```
bioruby> config :color
bioruby> codontable
(色付き)
```

実行するたびに設定が切り替わります。

```
bioruby> config :color
bioruby> codontable
(色なし)
```

BioRuby シェル起動時に表示されるスプラッシュメッセージを違う文字列に 変更します。何の解析プロジェクト用のディレクトリかを指定しておくのも よいでしょう。

```
bioruby> config :message, "Kumamushi genome project"

K u m a m u s h i   g e n o m e   p r o j e c t

  Version : BioRuby 0.8.0 / Ruby 1.8.4
```

デフォルトの文字列に戻すには、引数なしで実行します。

```
bioruby> config :message
```

BioRuby シェル起動時に表示されるスプラッシュメッセージを アニメーション表示するかどうかを切り替えます。 こちらも実行するたびに設定が切り替わります。

```
bioruby> config :splash
Splash on
```

pager(*command*)

disp コマンドで実際に利用するページャーを切り替えます。

```
bioruby> pager "lv"
Pager is set to 'lv'

bioruby> pager "less -S"
Pager is set to 'less -S'
```

ページャーを使用しない設定にする場合は引数なしで実行します。

```
bioruby> pager
Pager is set to 'off'
```

ページャーが off の時に引数なしで実行すると環境変数 PAGER の値を利用します。

```
bioruby> pager
Pager is set to 'less'
```

遺伝子アスキーアート

doublehelix(*sequence*)

DNA 配列をアスキーアートで表示するオマケ機能があります。 適当な塩基配列 seq を二重螺旋っぽく表示してみましょう。

```
bioruby> dna = seq("atgc" * 10).randomize
bioruby> doublehelix dna
     ta
    t--a
   a---t
  a----t
 a----t
t---a
g--c
 cg
 gc
a--t
g---c
 c----g
  c----g
(略)
```

遺伝子音楽

midifile(*midifile*, *sequence*)

DNA 配列を MIDI ファイルに変換するオマケ機能があります。 適当な塩基配列 seq を使って生成した midifile.mid を MIDI プレイヤーで演奏してみましょう。

```
bioruby> midifile("midifile.mid", seq)
Saving MIDI file (midifile.mid) ... done
```

以上で BioRuby シェルの解説を終わり、以下では BioRuby ライブラリ自体の 解説を行います。

塩基・アミノ酸配列を処理する (Bio::Sequence クラス)

Bio::Sequence クラスは、配列に対する様々な操作を行うことができます。 簡単な例として、短い塩基配列 atgcatgcaaaa を使って、相補配列への変換、 部分配列の切り出し、塩基組成の計算、アミノ酸への翻訳、分子量計算などを 行なってみます。アミノ酸への翻訳では、必要に応じて何塩基目から翻訳を開 始するかフレームを指定したり、codontable.rb で定義されているコドンテー ブルの中から使用するものを指定したりする事

ができます（コドンテーブルの 番号は http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi を参照）。

```
#!/usr/bin/env ruby

require 'bio'

seq = Bio::Sequence::NA.new("atgcatgcaaaa")

puts seq                              # 元の配列
puts seq.complement                   # 相補配列 (Bio::Sequence::NA)
puts seq.subseq(3,8)                  # 3 塩基目から 8 塩基目まで

p seq.gc_percent                      # GC 塩基の割合 (Integer)
p seq.composition                     # 全塩基組成 (Hash)

puts seq.translate                    # 翻訳配列 (Bio::Sequence::AA)
puts seq.translate(2)                 # ２文字目から翻訳（普通は１から）
puts seq.translate(1,9)               # ９番のコドンテーブルを使用

p seq.translate.codes                 # アミノ酸を３文字コードで表示 (Array)
p seq.translate.names                 # アミノ酸を名前で表示 (Array)
p seq.translate.composition           # アミノ酸組成 (Hash)
p seq.translate.molecular_weight      # 分子量を計算 (Float)

puts seq.complement.translate         # 相補配列の翻訳
```

print, puts, p は内容を画面に表示するための Ruby 標準メソッドです。 基本となる print と比べて、puts は改行を自動でつけてくれる、 p は文字列や数字以外のオブジェクトも人間が見やすいように表示してくれる、 という特徴がありますので適宜使い分けます。さらに、

```
require 'pp'
```

とすれば使えるようになる pp メソッドは、p よりも表示が見やすくなります。

塩基配列は Bio::Sequence::NA クラスの、アミノ酸配列は Bio::Sequence::AA クラスのオブジェクトになります。それぞれ Bio::Sequence クラスを継承し ているため、多くのメソッドは共通です。

さらに Bio::Sequence::NA, AA クラスは Ruby の String クラスを継承しているので String クラスが持つメソッドも使う事ができます。例えば部分配列を切り出すには Bio::Sequence クラスの subseq(from,to) メソッドの他に、String クラスの [] メソッドを使うこともできます。

Ruby の文字列は 1 文字目を 0 番目として数える点には注意が必要です。たとえば、

```
puts seq.subseq(1, 3)
puts seq[0, 3]
```

はどちらも seq の最初の３文字 atg を表示します。

このように、String のメソッドを使う場合は、生物学で普通使用される 1 文字目を 1 番目として数えた数字からは 1 を引く必要があります（subseq メソッドは これを内部でやっています。また、from, to のどちらかでも 0 以下の場合は 例外が発生するようになっています）。

ここまでの処理を BioRuby シェルで試すと以下のようになります。

```
# 次の行は seq = seq("atgcatgcaaaa") でもよい
bioruby> seq = Bio::Sequence::NA.new("atgcatgcaaaa")
# 生成した配列を表示
bioruby> puts seq
atgcatgcaaaa
# 相補配列を表示
bioruby> puts seq.complement
ttttgcatgcat
# 部分配列を表示（３塩基目から８塩基目まで）
bioruby> puts seq.subseq(3,8)
```

```
gcatgc
# 配列の GC% を表示
bioruby> p seq.gc_percent
33
# 配列の組成を表示
bioruby> p seq.composition
{"a"=>6, "c"=>2, "g"=>2, "t"=>2}
# アミノ酸配列への翻訳
bioruby> puts seq.translate
MHAK
# ２塩基を開始塩基として翻訳
bioruby> puts seq.translate(2)
CMQ
# ９番のコドンテーブルを使用して翻訳
bioruby> puts seq.translate(1,9)
MHAN
# 翻訳されたアミノ酸配列を３文字コードで表示
bioruby> p seq.translate.codes
["Met", "His", "Ala", "Lys"]
# 翻訳されたアミノ酸配列をアミノ酸の名前で表示
bioruby> p seq.translate.names
["methionine", "histidine", "alanine", "lysine"]
# 翻訳されたアミノ酸配列の組成を表示
bioruby> p seq.translate.composition
{"K"=>1, "A"=>1, "M"=>1, "H"=>1}
# 翻訳されたアミノ酸配列の分子量を表示
bioruby> p seq.translate.molecular_weight
485.605
# 相補配列を翻訳
bioruby> puts seq.complement.translate
FCMH
# 部分配列（１塩基目から３塩基目まで）
bioruby> puts seq.subseq(1, 3)
atg
# 部分配列（１塩基目から３塩基目まで）
bioruby> puts seq[0, 3]
atg
```

window_search(window_size, step_size) メソッドを使うと、配列に対してウィ ンドウをずらしながらそれぞれの部分配列に対する処理を行うことができます。 Ruby の特長のひとつである「ブロック」によって、「それぞれに対する処理」を 簡潔かつ明瞭に書くことが可能です。以下の例では、subseq という変数にそれぞれ 部分配列を代入しながらブロックを繰り返し実行することになります。

- 100 塩基ごとに（1塩基ずつずらしながら）平均 GC% を計算して表示する

```
seq.window_search(100) do |subseq|
  puts subseq.gc_percent
end
```

ブロックの中で受け取る部分配列も、元と同じ Bio::Sequence::NA または Bio::Sequence::AA クラスのオブジェクトなので、配列クラスの持つ全てのメ ソッドを実行することができます。

また、２番目の引数に移動幅を指定することが出来るようになっているので、

- コドン単位でずらしながら 15 塩基を 5 残基のペプチドに翻訳して表示する

```
seq.window_search(15, 3) do |subseq|
  puts subseq.translate
end
```

といったことができます。さらに移動幅に満たない右端の部分配列をメソッド 自体の返り値として戻すようになっているので、

- ゲノム配列を 10000bp ごとにブツ切りにして FASTA フォーマットに整形、 このとき末端 1000bp はオーバーラップさせ、10000bp に満たない 3' 端は 別途受け取って表示する

```
```

```
i = 1
remainder = seq.window_search(10000, 9000) do |subseq|
  puts subseq.to_fasta("segment #{i}", 60)
  i += 1
end
puts remainder.to_fasta("segment #{i}", 60)
```

のような事もわりと簡単にできます。

ウィンドウの幅と移動幅を同じにするとオーバーラップしないウィンドウサー チができるので、

- コドン頻度を数える

```
codon_usage = Hash.new(0)
seq.window_search(3, 3) do |subseq|
  codon_usage[subseq] += 1
end
```

- 10 残基ずつ分子量を計算

```
seq.window_search(10, 10) do |subseq|
  puts subseq.molecular_weight
end
```

といった応用も考えられます。

実際には Bio::Sequence::NA オブジェクトはファイルから読み込んだ文字列か ら生成したり、データベースから取得したものを使ったりします。たとえば、

```
#!/usr/bin/env ruby

require 'bio'

input_seq = ARGF.read       # 引数で与えられたファイルの全行を読み込む

my_naseq = Bio::Sequence::NA.new(input_seq)
my_aaseq = my_naseq.translate

puts my_aaseq
```

このプログラムを na2aa.rb として、以下の塩基配列

```
gtggcgatctttccgaaagcgatgactggagcgaagaaccaaagcagtgacatttgtctg
atgccgcacgtaggcctgataagacgcggacagcgtcgcatcaggcatcttgtgcaaatg
tcggatgcggcgtga
```

を書いたファイル my_naseq.txt を読み込んで翻訳すると

```
% ./na2aa.rb my_naseq.txt
VAIFPKAMTGAKNQSSDICLMPHVGLIRRGQRRIRHLVQMSDAA*
```

のようになります。ちなみに、このくらいの例なら短くすると 1 行で書けます。

```
% ruby -r bio -e 'p Bio::Sequence::NA.new($<.read).translate' my_naseq.txt
```

しかし、いちいちファイルを作るのも面倒なので、次はデータベースから必要な 情報を取得してみます。

GenBank のパース (Bio::GenBank クラス)

GenBank 形式のファイルを用意してください（手元にない場合は、 ftp://ftp.ncbi.nih.gov/genbank/ から .seq ファイルをダウンロードします）。

```
% wget ftp://ftp.hgc.jp/pub/mirror/ncbi/genbank/gbphg.seq.gz
% gunzip gbphg.seq.gz
```

まずは、各エントリから ID と説明文、配列を取り出して FASTA 形式に変換して みましょう。

Bio::GenBank::DELIMITER は GenBank クラスで定義されている定数で、 データベースごとに異なるエントリの区切り文字（たとえば GenBank の場合は //） を覚えていなくても良いようになっています。

```
#!/usr/bin/env ruby

require 'bio'

while entry = gets(Bio::GenBank::DELIMITER)
  gb = Bio::GenBank.new(entry)      # GenBank オブジェクト

  print ">#{gb.accession} "         # ACCESSION 番号
  puts gb.definition                # DEFINITION 行
  puts gb.naseq                     # 塩基配列（Sequence::NA オブジェクト）
end
```

しかし、この書き方では GenBank ファイルのデータ構造に依存しています。 ファイルからのデータ入力を扱うクラス Bio::FlatFile を使用することで、 以下のように区切り文字などを気にせず書くことができます。

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::FlatFile.new(Bio::GenBank, ARGF)
ff.each_entry do |gb|
  definition = "#{gb.accession} #{gb.definition}"
  puts gb.naseq.to_fasta(definition, 60)
end
```

形式の違うデータ、たとえばFASTAフォーマットのファイルを読み込むときでも、

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::FlatFile.new(Bio::FastaFormat, ARGF)
ff.each_entry do |f|
  puts "definition : " + f.definition
  puts "nalen      : " + f.nalen.to_s
  puts "naseq      : " + f.naseq
end
```

のように、同じような書き方で済ませられます。

さらに、各 Bio::DB クラスの open メソッドで同様のことができます。たとえば、

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::GenBank.open("gbvrl1.seq")
ff.each_entry do |gb|
  definition = "#{gb.accession} #{gb.definition}"
  puts gb.naseq.to_fasta(definition, 60)
end
```

などと書くことができます（ただし、この書き方はあまり使われていません)。

次に、GenBank の複雑な FEATURES の中をパースして必要な情報を取り出します。 まずは

/tranlation="アミノ酸配列" という Qualifier がある場合だけ アミノ酸配列を抽出して表示してみます。

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::FlatFile.new(Bio::GenBank, ARGF)

# GenBank の１エントリごとに
ff.each_entry do |gb|

  # FEATURES の要素を一つずつ処理
  gb.features.each do |feature|

    # Feature に含まれる Qualifier を全てハッシュに変換
    hash = feature.to_hash

    # Qualifier に translation がある場合だけ
    if hash['translation']
      # エントリのアクセッション番号と翻訳配列を表示
      puts ">#{gb.accession}
      puts hash['translation']
    end
  end
end
```

さらに、Feature のポジションに書かれている情報からエントリの塩基配列を スプライシングし、それを翻訳したものと /translation= に書かれていた配列を 両方表示して比べてみましょう。

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::FlatFile.new(Bio::GenBank, ARGF)

# GenBank の１エントリごとに
ff.each_entry do |gb|

  # ACCESSION 番号と生物種名を表示
  puts "### #{gb.accession} - #{gb.organism}"

  # FEATURES の要素を一つずつ処理
  gb.features.each do |feature|

    # Feature の position (join ...など) を取り出す
    position = feature.position

    # Feature に含まれる Qualifier を全てハッシュに変換
    hash = feature.to_hash

    # /translation= がなければスキップ
    next unless hash['translation']

    # /gene=, /product= などの Qualifier から遺伝子名などの情報を集める
    gene_info = [
      hash['gene'], hash['product'], hash['note'], hash['function']
    ].compact.join(', ')
    puts "## #{gene_info}"

    # 塩基配列（position の情報によってスプライシング）
    puts ">NA splicing('#{position}')"
    puts gb.naseq.splicing(position)

    # アミノ酸配列（スプライシングした塩基配列から翻訳）
    puts ">AA translated by splicing('#{position}').translate"
    puts gb.naseq.splicing(position).translate
```

```
    # アミノ酸配列（/translation= に書かれていたのもの）
    puts ">AA original translation"
    puts hash['translation']
  end
end
```

もし、使用されているコドンテーブルがデフォルト (universal) と違ったり、 最初のコドンが "atg" 以外だったり、セレノシステインが含まれていたり、 あるいは BioRuby にバグがあれば、上の例で表示される２つのアミノ酸配列は 異なる事になります。

この例で使用されている Bio::Sequence#splicing メソッドは、GenBank, EMBL, DDBJ フォーマットで使われている Location の表記を元に、塩基配列から 部分配列を切り出す強力なメソッドです。

この splicing メソッドの引数には GenBank 等の Location の文字列以外に BioRuby の Bio::Locations オブジェクトを渡すことも可能ですが、 通常は見慣れている Location 文字列の方が分かりやすいかも知れません。 Location 文字列のフォーマットや Bio::Locations について詳しく知りたい場合は BioRuby の bio/location.rb を見てください。

- GenBank 形式のデータの Feature で使われていた Location 文字列の例

```
naseq.splicing('join(2035..2050,complement(1775..1818),13..345')
```

- あらかじめ Locations オブジェクトに変換してから渡してもよい

```
locs = Bio::Locations.new('join((8298.8300)..10206,1..855)')
naseq.splicing(locs)
```

ちなみに、アミノ酸配列 (Bio::Sequence::AA) についても splicing メソッド を使用して部分配列を取り出すことが可能です。

- アミノ酸配列の部分配列を切り出す（シグナルペプチドなど）

```
aaseq.splicing('21..119')
```

GenBank 以外のデータベース

BioRuby では、GenBank 以外のデータベースについても基本的な扱い方は同じで、 データベースの 1 エントリ分の文字列を対応するデータベースのクラスに渡せば、 パースされた結果がオブジェクトになって返ってきます。

データベースのフラットファイルから 1 エントリずつ取り出してパースされた オブジェクトを取り出すには、先にも出てきた Bio::FlatFile を使います。 Bio::FlatFile.new の引数にはデータベースに対応する BioRuby でのクラス 名 (Bio::GenBank や Bio::KEGG::GENES など) を指定します。

```
ff = Bio::FlatFile.new(Bio::データベースクラス名, ARGF)
```

しかし、すばらしいことに、実は FlatFile クラスはデータベースの自動認識が できますので、

```
ff = Bio::FlatFile.auto(ARGF)
```

を使うのが一番簡単です。

```
#!/usr/bin/env ruby

require 'bio'

ff = Bio::FlatFile.auto(ARGF)

ff.each_entry do |entry|
  p entry.entry_id          # エントリの ID
  p entry.definition        # エントリの説明文
  p entry.seq               # 配列データベースの場合
```

```
end

ff.close
```

さらに、開いたデータベースの閉じ忘れをなくすためには Ruby のブロックを 活用して以下のように書くのがよいでしょう。

```
#!/usr/bin/env ruby

require 'bio'

Bio::FlatFile.auto(ARGF) do |ff|
  ff.each_entry do |entry|
    p entry.entry_id          # エントリの ID
    p entry.definition        # エントリの説明文
    p entry.seq               # 配列データベースの場合
  end
end
```

パースされたオブジェクトから、エントリ中のそれぞれの部分を取り出すための メソッドはデータベース毎に異なります。よくある項目については

- entry_id メソッド → エントリの ID 番号が返る
- definition メソッド → エントリの定義行が返る
- reference メソッド → リファレンスオブジェクトが返る
- organism メソッド → 生物種名
- seq や naseq や aaseq メソッド → 対応する配列オブジェクトが返る

などのように共通化しようとしていますが、全てのメソッドが実装されているわ けではありません（共通化の指針は bio/db.rb 参照）。また、細かい部分は各 データベースパーザ毎に異なるので、それぞれのドキュメントに従います。

原則として、メソッド名が複数形の場合は、オブジェクトが配列として返ります。 たとえば references メソッドを持つクラスは複数の Bio::Reference オブジェ クトを Array にして返しますが、別のクラスでは単数形の reference メソッド しかなく、１つの Bio::Reference オブジェクトだけを返す、といった感じです。

PDB のパース (Bio::PDB クラス)

Bio::PDB は、PDB 形式を読み込むためのクラスです。PDB データベースは PDB, mmCIF, XML (PDBML) の３種類のフォーマットで提供されていますが、 これらのうち BioRuby で対応しているのは PDB フォーマットです。

PDB フォーマットの仕様は、以下の Protein Data Bank Contents Guide を 参照してください。

- http://www.rcsb.org/pdb/file_formats/pdb/pdbguide2.2/guide2.2_frame.html

PDB データの読み込み

PDB の１エントリが 1bl8.pdb というファイルに格納されている場合は、 Ruby のファイル読み込み機能を使って

```
entry = File.read("1bl8.pdb")
```

のようにすることで、エントリの内容を文字列として entry という変数に 代入することができます。エントリの内容をパースするには

```
pdb = Bio::PDB.new(entry)
```

とします。これでエントリが Bio::PDB オブジェクトとなり、任意のデータを 取り出せるようになります。

PDB フォーマットは Bio::FlatFile による自動認識も可能ですが、現在は １ファイルに複数エントリを含む場合には対応していません。 Bio::FlatFile を使って１エントリ分だけ読み込むには、

```
pdb = Bio::FlatFile.auto("1bl8.pdb") { |ff| ff.next_entry }
```

とします。どちらの方法でも変数 pdb には同じ結果が得られます。

オブジェクトの階層構造

各 PDB エントリは、英数字４文字からなる ID が付けられています。 Bio::PDB オブジェクトから ID を取り出すには entry_id メソッドを使います。

```
p pdb.entry_id   # => "1BL8"
```

エントリの概要に関する情報も対応するメソッドで取り出すことができます。

```
p pdb.definition # => "POTASSIUM CHANNEL (KCSA) FROM STREPTOMYCES LIVIDANS"
p pdb.keywords   # => ["POTASSIUM CHANNEL", "INTEGRAL MEMBRANE PROTEIN"]
```

他に、登録者や文献、実験方法などの情報も取得できます（それぞれ authors, jrnl, method メソッド）。

PDB データは、基本的には１行が１つのレコードを形成しています。 １行に入りきらないデータを複数行に格納する continuation という 仕組みも用意されていますが、基本は１行１レコードです。

各行の先頭６文字がその行のデータの種類を示す名前（レコード）になります。 BioRuby では、HEADER レコードに対しては Bio::PDB::Record::HEADER クラス、 TITLE レコードに対しては Bio::PDB::Record::TITLE クラス、というように 基本的には各レコードに対応するクラスを１つ用意しています。 ただし、REMARK と JRNL レコードに関しては、それぞれ複数のフォーマットが 存在するため、複数のクラスを用意しています。

各レコードにアクセスするもっとも単純な方法は record メソッドです。

```
pdb.record("HELIX")
```

のようにすると、その PDB エントリに含まれる全ての HELIX レコードを Bio::PDB::Record::HELIX クラスのオブジェクトの配列として取得できます。

このことをふまえ、以下では、PDB エントリのメインな内容である立体構造に 関するデータ構造の扱い方を見ていきます。

原子: Bio::PDB::Record::ATOM, Bio::PDB::Record::HETATM クラス

PDB エントリは、タンパク質、核酸（DNA,RNA）やその他の分子の立体構造、 具体的には原子の３次元座標を含んでいます。

タンパク質または核酸の原子の座標は、ATOM レコードに格納されています。 対応するクラスは、Bio::PDB::Record::ATOM クラスです。

タンパク質・核酸以外の原子の座標は、HETATM レコードに格納されています。 対応するクラスは、Bio::PDB::Record::HETATM クラスです。

HETATM クラスは ATOM クラスを継承しているため、ATOM と HETATM の メソッドの使い方はまったく同じです。

アミノ酸残基（または塩基）: Bio::PDB::Residue クラス

１アミノ酸または１塩基単位で原子をまとめたのが Bio::PDB::Residue です。 Bio::PDB::Residue オブジェクトは、１個以上の Bio::PDB::Record::ATOM オブジェクトを含みます。

化合物: Bio::PDB::Heterogen クラス

タンパク質・核酸以外の分子の原子は、基本的には分子単位で Bio::PDB::Heterogen にまとめられています。 Bio::PDB::Heterogen オブジェクトは、１個以上の Bio::PDB::Record::HETATM オブジェクトを含みます。

鎖（チェイン）: Bio::PDB::Chain クラス

Bio::PDB::Chain は、複数の Bio::PDB::Residue オブジェクトからなる 1 個のタンパク質または核酸と、複数の Bio::PDB::Heterogen オブジェクト からなる 1 個以上のそれ以外の分子を格納するデータ構造です。

なお、大半の場合は、タンパク質・核酸（Bio::PDB::Residue）か、 それ以外の分子（Bio::PDB::Heterogen）のどちらか一種類しか持ちません。 Chain をひとつしか含まない PDB エントリでは両方持つ場合があるようです。

各 Chain には、英数字 1 文字の ID が付いています（Chain をひとつしか 含まない PDB エントリの場合は空白文字のときもあります）。

モデル: Bio::PDB::Model

1 個以上の Bio::PDB::Chain が集まったものが Bio::PDB::Model です。 X 線結晶構造の場合、Model は通常 1 個だけですが、NMR 構造の場合、 複数の Model が存在することがあります。 複数の Model が存在する場合、各 Model にはシリアル番号が付きます。

そして、 1 個以上の Model が集まったものが、Bio::PDB オブジェクトになります。

原子にアクセスするメソッド

Bio::PDB#each_atom は全ての ATOM を順番に 1 個ずつ辿るイテレータです。

```
pdb.each_atom do |atom|
  p atom.xyz
end
```

この each_atom メソッドは Model, Chain, Residue オブジェクトに対しても 使用することができ、それぞれ、その Model, Chain, Residue 内部のすべての ATOM をたどるイテレータとして働きます。

Bio::PDB#atoms は全ての ATOM を配列として返すメソッドです。

```
p pdb.atoms.size         # => 2820 個の ATOM が含まれることがわかる
```

each_atom と同様に atoms メソッドも Model, Chain, Residue オブジェクト に対して使用可能です。

```
pdb.chains.each do |chain|
  p chain.atoms.size     # => 各 Chain 毎の ATOM 数が表示される
end
```

Bio::PDB#each_hetatm は、全ての HETATM を順番に 1 個ずつ辿るイテレータです。

```
pdb.each_hetatm do |hetatm|
  p hetatm.xyz
end
```

Bio::PDB#hetatms 全ての HETATM を配列として返すのは hetatms メソッドです。

```
p pdb.hetatms.size
```

これらも atoms の場合と同様に、Model, Chain, Heterogen オブジェクトに 対して使用可能です。

Bio::PDB::Record::ATOM, Bio::PDB::Record::HETATM クラスの使い方

ATOM はタンパク質・核酸（DNA・RNA）を構成する原子、HETATM はそれ以外の 原子を格納するためのクラスですが、HETATM が ATOM クラスを継承しているため これらのクラスでメソッドの使い方はまったく同じです。

```
p atom.serial        # シリアル番号
```

```
p atom.name          # 名前
p atom.altLoc        # Alternate location indicator
p atom.resName       # アミノ酸・塩基名または化合物名
p atom.chainID       # Chain の ID
p atom.resSeq        # アミノ酸残基のシーケンス番号
p atom.iCode         # Code for insertion of residues
p atom.x             # X 座標
p atom.y             # Y 座標
p atom.z             # Z 座標
p atom.occupancy     # Occupancy
p atom.tempFactor    # Temperature factor
p atom.segID         # Segment identifier
p atom.element       # Element symbol
p atom.charge        # Charge on the atom
```

これらのメソッド名は、原則として Protein Data Bank Contents Guide の 記載に合わせています。メソッド名に resName や resSeq といった記名法 （CamelCase）を採用しているのはこのためです。 それぞれのメソッドの返すデータの意味は、仕様書を参考にしてください。

この他にも、いくつかの便利なメソッドを用意しています。 xyz メソッドは、座標を３次元のベクトルとして返すメソッドです。 このメソッドは、Ruby の Vector クラスを継承して３次元のベクトルに 特化させた Bio::PDB::Coordinate クラスのオブジェクトを返します （注: Vectorを継承したクラスを作成するのはあまり推奨されないようなので、 将来、Vectorクラスのオブジェクトを返すよう仕様変更するかもしれません）。

```
p atom.xyz
```

ベクトルなので、足し算、引き算、内積などを求めることができます。

```
# 原子間の距離を求める
p (atom1.xyz - atom2.xyz).r  # r はベクトルの絶対値を求めるメソッド

# 内積を求める
p atom1.xyz.inner_product(atom2.xyz)
```

他には、その原子に対応する TER, SIGATM, ANISOU レコードを取得する ter, sigatm, anisou メソッドも用意されています。

アミノ酸残基 (Residue) にアクセスするメソッド

Bio::PDB#each_residue は、全ての Residue を順番に辿るイテレータです。 each_residue メソッドは、Model, Chain オブジェクトに対しても 使用することができ、それぞれの Model, Chain に含まれる全ての Residue を辿るイテレータとして働きます。

```
pdb.each_residue do |residue|
  p residue.resName
end
```

Bio::PDB#residues は、全ての Residue を配列として返すメソッドです。 each_residue と同様に、Model, Chain オブジェクトに対しても使用可能です。

```
p pdb.residues.size
```

化合物 (Heterogen) にアクセスするメソッド

Bio::PDB#each_heterogen は全ての Heterogen を順番にたどるイテレータ、 Bio::PDB#heterogens は全ての Heterogen を配列として返すメソッドです。

```
pdb.each_heterogen do |heterogeon|
  p heterogen.resName
end

p pdb.heterogens.size
```

これらのメソッドも Residue と同様に Model, Chain オブジェクトに対しても 使用可能です。

Chain, Model にアクセスするメソッド

同様に、Bio::PDB#each_chain は全ての Chain を順番にたどるイテレータ、 Bio::PDB#chains は全ての Chain を配列として返すメソッドです。 これらのメソッドは Model オブジェクトに対しても使用可能です。

Bio::PDB#each_model は全ての Model を順番にたどるイテレータ、 Bio::PDB#models は全ての Model を配列として返すメソッドです。

PDB Chemical Component Dictionary のデータの読み込み

Bio::PDB::ChemicalComponent クラスは、PDB Chemical Component Dictionary （旧名称 HET Group Dictionary）のパーサです。

PDB Chemical Component Dictionary については以下のページを参照してください。

- http://deposit.pdb.org/cc_dict_tut.html

データは以下でダウンロードできます。

- http://deposit.pdb.org/het_dictionary.txt

このクラスは、RESIDUE から始まって空行で終わる 1 エントリをパースします （PDB フォーマットにのみ対応しています）。

Bio::FlatFile によるファイル形式自動判別に対応しています。 このクラス自体は ID から化合物を検索したりする機能は持っていません。 br_bioflat.rb によるインデックス作成には対応していますので、 必要ならそちらを使用してください。

```
Bio::FlatFile.auto("het_dictionary.txt") |ff|
  ff.each do |het|
    p het.entry_id  # ID
    p het.hetnam    # HETNAM レコード（化合物の名称）
    p het.hetsyn    # HETSYM レコード（化合物の別名の配列）
    p het.formul    # FORMUL レコード（化合物の組成式）
    p het.conect    # CONECT レコード
  end
end
```

最後の conect メソッドは、化合物の結合を Hash として返します。 たとえば、エタノールのエントリは次のようになりますが、

```
RESIDUE   EOH      9
CONECT      C1     4 C2   O   1H1  2H1
CONECT      C2     4 C1  1H2  2H2  3H2
CONECT      O      2 C1   HO
CONECT     1H1     1 C1
CONECT     2H1     1 C1
CONECT     1H2     1 C2
CONECT     2H2     1 C2
CONECT     3H2     1 C2
CONECT      HO     1 O
END
HET    EOH              9
HETNAM     EOH ETHANOL
FORMUL      EOH    C2 H6 O1
```

このエントリに対して conect メソッドを呼ぶと

```
{ "C1"  => [ "C2", "O", "1H1", "2H1" ],
  "C2"  => [ "C1", "1H2", "2H2", "3H2" ],
  "O"   => [ "C1", "HO" ],
  "1H1" => [ "C1" ],
```

```
  "1H2" => [ "C2" ],
  "2H1" => [ "C1" ],
  "2H2" => [ "C2" ],
  "3H2" => [ "C2" ],
  "HO"  => [ "O" ] }
```

という Hash を返します。

ここまでの処理を BioRuby シェルで試すと以下のようになります。

```
# PDB エントリ 1bl8 をネットワーク経由で取得
bioruby> ent_1bl8 = ent("pdb:1bl8")
# エントリの中身を確認
bioruby> head ent_1bl8
# エントリをファイルに保存
bioruby> savefile("1bl8.pdb", ent_1bl8)
# 保存されたファイルの中身を確認
bioruby> disp "data/1bl8.pdb"
# PDB エントリをパース
bioruby> pdb_1bl8 = flatparse(ent_1bl8)
# PDB のエントリ ID を表示
bioruby> pdb_1bl8.entry_id
# ent("pdb:1bl8") して flatparse する代わりに、以下でもOK
bioruby> obj_1bl8 = obj("pdb:1bl8")
bioruby> obj_1bl8.entry_id
# 各 HETEROGEN ごとに残基名を表示
bioruby> pdb_1bl8.each_heterogen { |heterogen| p heterogen.resName }

# PDB Chemical Component Dictionary を取得
bioruby> het_dic = open("http://deposit.pdb.org/het_dictionary.txt").read
# 取得したファイルのバイト数を確認
bioruby> het_dic.size
# 取得したファイルを保存
bioruby> savefile("data/het_dictionary.txt", het_dic)
# ファイルの中身を確認
bioruby> disp "data/het_dictionary.txt"
# 検索のためにインデックス化し het_dic というデータベースを作成
bioruby> flatindex("het_dic", "data/het_dictionary.txt")
# ID が EOH のエタノールのエントリを検索
bioruby> ethanol = flatsearch("het_dic", "EOH")
# 取得したエントリをパース
bioruby> osake = flatparse(ethanol)
# 原子間の結合テーブルを表示
bioruby> sake.conect
```

アライメント (Bio::Alignment クラス)

Bio::Alignment クラスは配列のアライメントを格納するためのコンテナです。 Ruby の Hash や Array に似た操作が可能で、BioPerl の Bio::SimpleAlign に 似た感じになっています。以下に簡単な使い方を示します。

```
require 'bio'

seqs = [ 'atgca', 'aagca', 'acgca', 'acgcg' ]
seqs = seqs.collect{ |x| Bio::Sequence::NA.new(x) }

# アライメントオブジェクトを作成
a = Bio::Alignment.new(seqs)

# コンセンサス配列を表示
p a.consensus             # ==> "a?gc?"

# IUPAC 標準の曖昧な塩基を使用したコンセンサス配列を表示
p a.consensus_iupac       # ==> "ahgcr"
```

```
# 各配列について繰り返す
a.each { |x| p x }
  # ==>
  #    "atgca"
  #    "aagca"
  #    "acgca"
  #    "acgcg"

# 各サイトについて繰り返す
a.each_site { |x| p x }
  # ==>
  #    ["a", "a", "a", "a"]
  #    ["t", "a", "c", "c"]
  #    ["g", "g", "g", "g"]
  #    ["c", "c", "c", "c"]
  #    ["a", "a", "a", "g"]

# Clustal W を使用してアライメントを行う。
# 'clustalw' コマンドがシステムにインストールされている必要がある。
factory = Bio::ClustalW.new
a2 = a.do_align(factory)
```

FASTA による相同性検索を行う（Bio::Fasta クラス）

FASTA 形式の配列ファイル query.pep に対して、自分のマシン(ローカル)あるいは インターネット上のサーバ(リモート)で FASTA による相同性検索を行う方法です。 ローカルの場合は SSEARCH なども同様に使うことができます。

ローカルの場合

FASTA がインストールされていることを確認してください。以下の例では、 コマンド名が fasta34 でパスが通ったディレクトリにインストール されている状況を仮定しています。

- ftp://ftp.virginia.edu/pub/fasta/

検索対象とする FASTA 形式のデータベースファイル target.pep と、FASTA 形式の問い合わせ配列がいくつか入ったファイル query.pep を準備します。

この例では、各問い合わせ配列ごとに FASTA 検索を実行し、ヒットした配列の evalue が 0.0001 以下のものだけを表示します。

```
#!/usr/bin/env ruby

require 'bio'

# FASTA を実行する環境オブジェクトを作る（ssearch などでも良い）
factory = Bio::Fasta.local('fasta34', ARGV.pop)

# フラットファイルを読み込み、FastaFormat オブジェクトのリストにする
ff = Bio::FlatFile.new(Bio::FastaFormat, ARGF)

# 1エントリずつの FastaFormat オブジェクトに対し
ff.each do |entry|
  # '>' で始まるコメント行の内容を進行状況がわりに標準エラー出力に表示
  $stderr.puts "Searching ... " + entry.definition

  # FASTA による相同性検索を実行、結果は Fasta::Report オブジェクト
  report = factory.query(entry)

  # ヒットしたものそれぞれに対し
  report.each do |hit|
    # evalue が 0.0001 以下の場合
    if hit.evalue < 0.0001
      # その evalue と、名前、オーバーラップ領域を表示
      print "#{hit.query_id} : evalue #{hit.evalue}¥t#{hit.target_id} at "
      p hit.lap_at
```

```
    end
  end
end
```

ここで factory は繰り返し FASTA を実行するために、あらかじめ作っておく 実行環境です。

上記のスクリプトを search.rb とすると、問い合わせ配列とデータベース配列の ファイル名を引数にして、以下のように実行します。

```
% ruby search.rb query.pep target.pep > search.out
```

FASTA コマンドにオプションを与えたい場合、３番目の引数に FASTA の コマンドラインオプションを書いて渡します。ただし、ktup 値だけは メソッドを使って指定することになっています。 たとえば ktup 値を 1 にして、トップ 10 位以内のヒットを得る場合の オプションは、以下のようになります。

```
factory = Bio::Fasta.local('fasta34', 'target.pep', '-b 10')
factory.ktup = 1
```

Bio::Fasta#query メソッドなどの返り値は Bio::Fasta::Report オブジェクト です。この Report オブジェクトから、様々なメソッドで FASTA の出力結果の ほぼ全てを自由に取り出せるようになっています。たとえば、ヒットに関する スコアなどの主な情報は、

```
report.each do |hit|
  puts hit.evalue           # E-value
  puts hit.sw               # Smith-Waterman スコア (*)
  puts hit.identity         # % identity
  puts hit.overlap          # オーバーラップしている領域の長さ
  puts hit.query_id         # 問い合わせ配列の ID
  puts hit.query_def        # 問い合わせ配列のコメント
  puts hit.query_len        # 問い合わせ配列の長さ
  puts hit.query_seq        # 問い合わせ配列
  puts hit.target_id        # ヒットした配列の ID
  puts hit.target_def       # ヒットした配列のコメント
  puts hit.target_len       # ヒットした配列の長さ
  puts hit.target_seq       # ヒットした配列
  puts hit.query_start      # 相同領域の問い合わせ配列での開始残基位置
  puts hit.query_end        # 相同領域の問い合わせ配列での終了残基位置
  puts hit.target_start     # 相同領域のターゲット配列での開始残基位置
  puts hit.target_end       # 相同領域のターゲット配列での終了残基位置
  puts hit.lap_at           # 上記４位置の数値の配列
end
```

などのメソッドで呼び出せます。これらのメソッドの多くは後で説明する Bio::Blast::Report クラスと共通にしてあります。上記以外のメソッドや FASTA 特有の値を取り出すメソッドが必要な場合は、Bio::Fasta::Report クラスのドキュメントを参照してください。

もし、パースする前の手を加えていない fasta コマンドの実行結果が必要な 場合には、

```
report = factory.query(entry)
puts factory.output
```

のように、query メソッドを実行した後で factory オブジェクトの output メソッドを使って取り出すことができます。

リモートの場合

今のところ GenomeNet (fasta.genome.jp) での検索のみサポートしています。 リモートの場合は使用可能な検索対象データベースが決まっていますが、それ以 外の点については Bio::Fasta.remote と Bio::Fasta.local は同じように使う ことができます。

GenomeNet で使用可能な検索対象データベース：

- アミノ酸配列データベース

    - nr-aa, genes, vgenes.pep, swissprot, swissprot-upd, pir, prf, pdbstr
- 塩基配列データベース
    - nr-nt, genbank-nonst, gbnonst-upd, dbest, dbgss, htgs, dbsts, embl-nonst, embnonst-upd, genes-nt, genome, vgenes.nuc

まず、この中から検索したいデータベースを選択します。問い合わせ配列の種類 と検索するデータベースの種類によってプログラムは決まります。

- 問い合わせ配列がアミノ酸のとき
    - 対象データベースがアミノ酸配列データベースの場合、program は 'fasta'
    - 対象データベースが核酸配列データベースの場合、program は 'tfasta'
- 問い合わせ配列が核酸配列のとき
    - 対象データベースが核酸配列データベースの場合、program は 'fasta'
    - (対象データベースがアミノ酸配列データベースの場合は検索不能?)

プログラムとデータベースの組み合せが決まったら

```
program = 'fasta'
database = 'genes'

factory = Bio::Fasta.remote(program, database)
```

としてファクトリーを作り、ローカルの場合と同じように factory.query など のメソッドで検索を実行します。

BLAST による相同性検索を行う（Bio::Blast クラス）

BLAST もローカルと GenomeNet (blast.genome.jp) での検索をサポートして います。できるだけ Bio::Fasta と API を共通にしていますので、上記の例を Bio::Blast と書き換えただけでも大丈夫な場合が多いです。

たとえば、先の f_search.rb は

```
# BLAST を実行する環境オブジェクトを作る
factory = Bio::Blast.local('blastp', ARGV.pop)
```

と変更するだけで同じように実行できます。

同様に、GenomeNet を使用してBLASTを行う場合には Bio::Blast.remote を使います。 この場合、programの指定内容が FASTA と異なります。

- 問い合わせ配列がアミノ酸のとき
    - 対象データベースがアミノ酸配列データベースの場合、program は 'blastp'
    - 対象データベースが核酸配列データベースの場合、program は 'tblastn'
- 問い合わせ配列が塩基配列のとき
    - 対象データベースがアミノ酸配列データベースの場合、program は 'blastx'
    - 対象データベースが塩基配列データベースの場合、program は 'blastn'
    - (問い合わせ・データベース共に6フレーム翻訳を行う場合は 'tblastx')

をそれぞれ指定します。

ところで、BLAST では "-m 7" オプションによる XML 出力フォーマットの方が 得られる情報が豊富なため、Bio::Blast は Ruby 用の XML ライブラリである XMLParser または REXML が使用可能な場合は、XML出力を利用します。 両方使用可能な場合、XMLParser のほうが高速なので優先的に使用されます。 なお、Ruby 1.8.0 以降では REXML は Ruby 本体に標準添付されています。 もし XML ライブラリがインストールされていない場合は "-m 8" のタブ区切りの 出力形式を扱うようにしています。しかし、このフォーマットでは得られる データが限られるので、"-m 7" の XML 形式の出力を使うことをお勧めします。

すでに見たように Bio::Fasta::Report と Bio::Blast::Report の Hit オブジェ クトはいくつか共通のメソッドを持っています。BLAST 固有のメソッドで良く使 いそうなものには bit_score や midline などがあります。

```
report.each do |hit|
  puts hit.bit_score        # bit スコア (*)
```

```
  puts hit.query_seq        # 問い合わせ配列
  puts hit.midline          # アライメントの midline 文字列 (*)
  puts hit.target_seq       # ヒットした配列

  puts hit.evalue           # E-value
  puts hit.identity         # % identity
  puts hit.overlap          # オーバーラップしている領域の長さ
  puts hit.query_id         # 問い合わせ配列の ID
  puts hit.query_def        # 問い合わせ配列のコメント
  puts hit.query_len        # 問い合わせ配列の長さ
  puts hit.target_id        # ヒットした配列の ID
  puts hit.target_def       # ヒットした配列のコメント
  puts hit.target_len       # ヒットした配列の長さ
  puts hit.query_start      # 相同領域の問い合わせ配列での開始残基位置
  puts hit.query_end        # 相同領域の問い合わせ配列での終了残基位置
  puts hit.target_start     # 相同領域のターゲット配列での開始残基位置
  puts hit.target_end       # 相同領域のターゲット配列での終了残基位置
  puts hit.lap_at           # 上記 4 位置の数値の配列
end
```

FASTAとのAPI共通化のためと簡便のため、スコアなどいくつかの情報は1番目の Hsp (High-scoring segment pair) の値をHitで返すようにしています。

Bio::Blast::Report オブジェクトは、以下に示すような、BLASTの結果出力の データ構造をそのまま反映した階層的なデータ構造を持っています。具体的には

- Bio::Blast::Report オブジェクトの @iteratinos に
  - Bio::Blast::Report::Iteration オブジェクトの Array が入っており
    Bio::Blast::Report::Iteration オブジェクトの @hits に
    - Bio::Blast::Report::Hits オブジェクトの Array が入っており
      Bio::Blast::Report::Hits オブジェクトの @hsps に
      - Bio::Blast::Report::Hsp オブジェクトの Array が入っている

という階層構造になっており、それぞれが内部の値を取り出すためのメソッドを 持っています。これらのメソッドの詳細や、BLAST 実行の統計情報などの値が 必要な場合には、 bio/appl/blast/*.rb 内のドキュメントやテストコードを 参照してください。

既存の BLAST 出力ファイルをパースする

BLAST を実行した結果ファイルがすでに保存してあって、これを解析したい場合 には（Bio::Blast オブジェクトを作らずに） Bio::Blast::Report オブジェク トを作りたい、ということになります。これには Bio::Blast.reports メソッド を使います。対応しているのは デフォルト出力フォーマット("-m 0") または "-m 7" オプションの XML フォーマット出力です。

```
#!/usr/bin/env ruby

require 'bio'

# BLAST出力を順にパースして Bio::Blast::Report オブジェクトを返す
Bio::Blast.reports(ARGF) do |report|
  puts "Hits for " + report.query_def + " against " + report.db
  report.each do |hit|
    print hit.target_id, "¥t", hit.evalue, "¥n" if hit.evalue < 0.001
  end
end
```

のようなスクリプト hits_under_0.001.rb を書いて、

```
% ./hits_under_0.001.rb *.xml
```

などと実行すれば、引数に与えた BLAST の結果ファイル *.xml を順番に処理で きます。

Blast のバージョンや OS などによって出力される XML の形式が異なる可能性 があり、時々 XML のパーザがうまく使えないことがあるようです。その場合は Blast 2.2.5 以降のバージョンをインストールするか -D や -m などのオプショ ンの組み合せを変えて試してみてください。

リモート検索サイトを追加するには

注: このセクションは上級ユーザ向けです。可能であれば SOAP などによる ウェブサービスを利用する方がよいでしょう。

Blast 検索は NCBI をはじめ様々なサイトでサービスされていますが、今のとこ ろ BioRuby では GenomeNet 以外には対応していません。これらのサイトは、

- CGI を呼び出す（コマンドラインオプションはそのサイト用に処理する）
- -m 8 など BioRuby がパーザを持っている出力フォーマットで blast の 出力を取り出す

ことさえできれば、query を受け取って検索結果を Bio::Blast::Report.new に 渡すようなメソッドを定義するだけで使えるようになります。具体的には、この メソッドを「exec_サイト名」のような名前で Bio::Blast の private メソッド として登録すると、４番目の引数に「サイト名」を指定して

```
factory = Bio::Blast.remote(program, db, option, 'サイト名')
```

のように呼び出せるようになっています。完成したら BioRuby プロジェクトま で送ってもらえれば取り込ませて頂きます。

PubMed を引いて引用文献リストを作る (Bio::PubMed クラス)

次は、NCBI の文献データベース PubMed を検索して引用文献リストを作成する例です。

```
#!/usr/bin/env ruby

require 'bio'

ARGV.each do |id|
  entry = Bio::PubMed.query(id)     # PubMed を取得するクラスメソッド
  medline = Bio::MEDLINE.new(entry) # Bio::MEDLINE オブジェクト
  reference = medline.reference     # Bio::Reference オブジェクト
  puts reference.bibtex             # BibTeX フォーマットで出力
end
```

このスクリプトを pmfetch.rb など好きな名前で保存し、

```
% ./pmfetch.rb 11024183 10592278 10592173
```

など引用したい論文の PubMed ID (PMID) を引数に並べると NCBI にアクセスし て MEDLINE フォーマットをパースし BibTeX フォーマットに変換して出力して くれるはずです。

他に、キーワードで検索する機能もあります。

```
#!/usr/bin/env ruby

require 'bio'

# コマンドラインで与えたキーワードのリストを 1 つの文字列にする
keywords = ARGV.join(' ')

# PubMed をキーワードで検索
entries = Bio::PubMed.search(keywords)

entries.each do |entry|
  medline = Bio::MEDLINE.new(entry) # Bio::MEDLINE オブジェクト
  reference = medline.reference     # Bio::Reference オブジェクト
  puts reference.bibtex             # BibTeX フォーマットで出力
end
```

このスクリプトを pmsearch.rb など好きな名前で保存し

```
% ./pmsearch.rb genome bioinformatics
```

など検索したいキーワードを引数に並べて実行すると、PubMed をキーワード 検索してヒットした論文のリストを BibTeX フォーマットで出力します。

最近では、NCBI は E-Utils というウェブアプリケーションを使うことが 推奨されているので、今後は Bio::PubMed.esearch メソッドおよび Bio::PubMed.efetch メソッドを使う方が良いでしょう。

```
#!/usr/bin/env ruby

require 'bio'

keywords = ARGV.join(' ')

options = {
  'maxdate' => '2003/05/31',
  'retmax' => 1000,
}

entries = Bio::PubMed.esearch(keywords, options)

Bio::PubMed.efetch(entries).each do |entry|
  medline = Bio::MEDLINE.new(entry)
  reference = medline.reference
  puts reference.bibtex
end
```

このスクリプトでは、上記の pmsearch.rb とほぼ同じように動きます。さらに、 NCBI E-Utils を活用することにより、検索対象の日付や最大ヒット件数などを 指定できるようになっているので、より高機能です。オプションに与えられる 引数については E-Utils のヘルプページ を参照してください。

ちなみに、ここでは bibtex メソッドで BibTeX フォーマットに変換しています が、後述のように bibitem メソッドも使える他、（強調やイタリックなど 文字の修飾はできませんが）nature メソッドや nar など、いくつかの雑誌の フォーマットにも対応しています。

BibTeX の使い方のメモ

上記の例で集めた BibTeX フォーマットのリストを TeX で使う方法を簡単にま とめておきます。引用しそうな文献を

```
% ./pmfetch.rb 10592173 >> genoinfo.bib
% ./pmsearch.rb genome bioinformatics >> genoinfo.bib
```

などとして genoinfo.bib ファイルに集めて保存しておき、

```
¥documentclass{jarticle}
¥begin{document}
¥bibliographystyle{plain}
ほにゃらら KEGG データベース~¥cite{PMID:10592173}はふがほげである。
¥bibliography{genoinfo}
¥end{document}
```

というファイル hoge.tex を書いて、

```
% platex hoge
% bibtex hoge   # → genoinfo.bib の処理
% platex hoge   # → 文献リストの作成
% platex hoge   # → 文献番号
```

とすると無事 hoge.dvi ができあがります。

bibitem の使い方のメモ

文献用に別の .bib ファイルを作りたくない場合は Reference#bibitem メソッ ドの出力を使います。上記の pmfetch.rb や pmsearch.rb の

```
puts reference.bibtex
```

の行を

```
puts reference.bibitem
```

に書き換えるなどして、出力結果を

```
¥documentclass{jarticle}
¥begin{document}
ほにゃらら KEGG データベース~¥cite{PMID:10592173}はふがほげである。

¥begin{thebibliography}{00}

¥bibitem{PMID:10592173}
Kanehisa, M., Goto, S.
KEGG: kyoto encyclopedia of genes and genomes.,
{¥em Nucleic Acids Res}, 28(1):27--30, 2000.

¥end{thebibliography}
¥end{document}
```

のように ¥begin{thebibliography} で囲みます。これを hoge.tex とすると

```
% platex hoge   # → 文献リストの作成
% platex hoge   # → 文献番号
```

と２回処理すればできあがりです。

OBDA

OBDA (Open Bio Database Access) とは、Open Bioinformatics Foundation によって制定された、配列データベースへの共通アクセス方法です。これは、 2002 年の1月と2月に Arizona と Cape Town にて開催された BioHackathon において、BioPerl, BioJava, BioPython, BioRuby などの各プロジェクトの メンバーが参加して作成されました。

- BioRegistry (Directory)
  - データベース毎に配列をどこにどのように取りに行くかを指定する仕組み
- BioFlat
  - フラットファイルの 2 分木または BDB を使ったインデックス作成
- BioFetch
  - HTTP 経由でデータベースからエントリを取得するサーバとクライアント
- BioSQL
  - MySQL や PostgreSQL などの関係データベースに配列データを格納する ための schema と、エントリを取り出すためのメソッド

詳細は http://obda.open-bio.org/ を参照してください。 それぞれの仕様書は cvs.open-bio.org の CVSレポジトリに置いてあります。 または、http://cvs.open-bio.org/cgi-bin/viewcvs/viewcvs.cgi/obda-specs/?cvsroot=obf-common から参照できます。

BioRegistry

BioRegistryとは、設定ファイルによって各データベースのエントリ取得方法を 指定することにより、どんな方法を使っているかをほとんど意識せずデータを 取得することを可能とするための仕組みです。 設定ファイルの優先順位は

- (メソッドのパラメータで)指定したファイル

- ~/.bioinformatics/seqdatabase.ini
- /etc/bioinformatics/seqdatabase.ini
- http://www.open-bio.org/registry/seqdatabase.ini

最後の open-bio.org の設定は、ローカルな設定ファイルが見つからない場合に だけ参照します。

BioRuby の現在の実装では、すべてのローカルな設定ファイルを読み込み、 同じ名前の設定が複数存在した場合は、最初に見つかった設定だけが使用されます。 これを利用すると、たとえば、システム管理者が /etc/bioinformatics/ に置いた 設定のうち個人的に変更したいものだけ ~/.bioinformatics/ で上書きすることが できます。サンプルの seqdatabase.ini ファイルが bioruby のソースに含まれて いますので参照してください。

設定ファイルの中身は stanza フォーマットと呼ばれる書式で記述します。

```
[データベース名]
protocol=プロトコル名
location=サーバ名
```

このようなエントリを各データベースについて記述することになります。 データベース名は、自分が使用するためのラベルなので分かりやすいものを つければ良く、実際のデータベースの名前と異なっていても構わないようです。 同じ名前のデータベースが複数あるときは最初に書かれているものから順に 接続を試すように仕様書では提案されていますが、今のところ BioRuby では それには対応していません。

また、プロトコルの種類によっては location 以外にも（MySQL のユーザ名など） 追加のオプションを記述する必要があります。現在のところ、仕様書で規定され ている protocol としては以下のものがあります。

- index-flat
- index-berkeleydb
- biofetch
- biosql
- bsane-corba
- xembl

今のところ BioRuby で使用可能なのは index-flat, index-berkleydb, biofetch と biosql だけです。また、BioRegistryや各プロトコルの仕様は変更されること がありますが、BioRubyはそれに追従できていないかもしれません。

BioRegistry を使うには、まず Bio::Registryオブジェクトを作成します。 すると、設定ファイルが読み込まれます。

```
reg = Bio::Registry.new

# 設定ファイルに書いたデータベース名でサーバへ接続
serv = reg.get_database('genbank')

# ID を指定してエントリを取得
entry = serv.get_by_id('AA2CG')
```

ここで serv は設定ファイルの [genbank] の欄で指定した protocol プロトコ ルに対応するサーバオブジェクトで、Bio::SQL や Bio::Fetch などのインスタ ンスが返っているはずです（データベース名が見つからなかった場合は nil）。

あとは OBDA 共通のエントリ取得メソッド get_by_id を呼んだり、サーバオ ブジェクト毎に固有のメソッドを呼ぶことになりますので、以下の BioFetch や BioSQL の解説を参照してください。

BioFlat

BioFlat はフラットファイルに対してインデックスを作成し、エントリを高速に 取り出す仕組みです。インデックスの種類は、RUbyの拡張ライブラリに依存しない index-flat と Berkeley DB (bdb) を使った index-berkeleydb の2種類が存在 します。なお、index-berkeleydb を使用するには、BDB という Ruby の拡張 ライブラリを別途インストールする必要があります。インデックスの作成には bioruby パッケージに付属する br_bioflat.rb コマンドを使って、

```
% br_bioflat.rb --makeindex データベース名 [--format クラス名] ファイル名
```

のようにします。BioRubyはデータフォーマットの自動認識機能を搭載している ので --format オプションは省略可能ですが、万一うまく認識しなかった場合は BioRuby の各データベースのクラス名を指定してください。検索は、

```
% bioflat データベース名 エントリID
```

とします。具体的に GenBank の gbbct*.seq ファイルにインデックスを作成し て検索する場合、

```
% bioflat --makeindex my_bctdb --format GenBank gbbct*.seq
% bioflat my_bctdb A16STM262
```

のような感じになります。

Ruby の bdb 拡張モジュール(詳細は http://raa.ruby-lang.org/project/bdb/ 参照) がインストールされている場合は Berkeley DB を利用してインデックスを作成する ことができます。この場合、

```
% bioflat --makeindex-bdb データベース名 [--format クラス名] ファイル名
```

のように "--makeindex" のかわりに "--makeindex-bdb" を指定します。

BioFetch

BioFetch は CGI を経由してサーバからデータベースのエントリを取得する仕様 で、サーバが受け取る CGI のオプション名、エラーコードなどが決められてい ます。クライアントは HTTP を使ってデータベース、ID、フォーマットなどを指 定し、エントリを取得します。

BioRuby プロジェクトでは GenomeNet の DBGET システムをバックエンドとした BioFetch サーバを実装しており、bioruby.org で運用しています。このサーバの ソースコードは BioRuby の sample/ ディレクトリに入っています。現在のところ BioFetch サーバはこの bioruby.org のものと EBI の二か所しかありません。

BioFetch を使ってエントリを取得するには、いくつかの方法があります。

1. ウェブブラウザから検索する方法（以下のページを開く）

   ```
   http://bioruby.org/cgi-bin/biofetch.rb
   ```

2. BioRuby付属の br_biofetch.rb コマンドを用いる方法

   ```
   % br_biofetch.rb db_name entry_id
   ```

3. スクリプトの中から Bio::Fetch クラスを直接使う方法

   ```
   serv = Bio::Fetch.new(server_url)
   entry = serv.fetch(db_name, entry_id)
   ```

4. スクリプトの中で BioRegistry 経由で Bio::Fetch クラスを間接的に使う方法

   ```
   reg = Bio::Registry.new
   serv = reg.get_database('genbank')
   entry = serv.get_by_id('AA2CG')
   ```

もし (4) を使いたい場合は seqdatabase.ini で

```
[genbank]
protocol=biofetch
location=http://bioruby.org/cgi-bin/biofetch.rb
biodbname=genbank
```

などと指定しておく必要があります。

BioFetch と Bio::KEGG::GENES, Bio::AAindex1 を組み合わせた例

次のプログラムは、BioFetch を使って KEGG の GENES データベースから古細菌 Halobacterium のバクテリアロドプシン遺伝子 (VNG1467G) を取ってきて、同じ ようにアミノ酸指標データベースである AAindex から取得したαヘリックスの 指標 (BURA740101) を使って、幅 15 残基のウィンドウサーチをする例です。

```
#!/usr/bin/env ruby

require 'bio'

entry = Bio::Fetch.query('hal', 'VNG1467G')
aaseq = Bio::KEGG::GENES.new(entry).aaseq

entry = Bio::Fetch.query('aax1', 'BURA740101')
helix = Bio::AAindex1.new(entry).index

position = 1
win_size = 15

aaseq.window_search(win_size) do |subseq|
  score = subseq.total(helix)
  puts [ position, score ].join("¥t")
  position += 1
end
```

ここで使っているクラスメソッド Bio::Fetch.query は暗黙に bioruby.org の BioFetch サーバを使う専用のショートカットです。（このサーバは内部的には ゲノムネットからデータを取得しています。KEGG/GENES データベースの hal や AAindex データベース aax1 のエントリは、他の BioFetch サーバでは取得でき ないこともあって、あえて query メソッドを使っています。）

さらなる情報

他のチュートリアル的なドキュメントとしては、BioRuby Wikiに置いてある BioRuby in Anger があります。

更新日時:Mon Feb 27 19:52:55 JST 2006