BioRuby の使い方

塩基・アミノ酸配列を処理する (Bio::Sequence クラス)

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の番号を指定したりする事ができます。

```
#!/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 (Sequence::NA オブジェクト)
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 (Float)
p seq.composition                       # 全塩基組成 (Hash)

puts seq.translate                      # 翻訳配列 (Sequence::AA オブジェクト)
puts seq.translate(2)                   # 2文字目から翻訳（普通は1から）
puts seq.translate(1,11)                # 11番目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
```

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。

Bio::Sequence 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 subseq(from,to) の他に、String の [] メソッドも使うことができます。た だし、Ruby の文字列は 1 文字目を 0 番目として数えますので、塩基配列や アミノ酸配列は通常 1 ずらして考えないといけないため注意が必要です （subseq メソッドは、内部で 0 base から 1 base への変換をしていて、 from, to のどちらかでも 0 以下の場合は nil を返すようになっています）。

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。 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。例えば、

- 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
end
```

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（元の ftp://ftp.ncbi.nih.gov/genbank/ の .seq ファ イルでも、サブセットでもよい）が手元にあるとして、gb2fasta コマンドの真 似をして、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 してみます。ちなみに gets で使われている 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。また、 名前が長いので RS (record separator) という別名もつけてあります。

```
#!/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
```

この後、フラットファイルを扱うラッパークラス Bio::FlatFile が実装され たので、次のように書き直すことができます。

```
#!/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
```

逆に、FASTA フォーマットのファイルを読み込むには、

```
#!/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
```

などとすることができます。

このとき、Bio::FlatFile.new の最初の引数にデータベースファイルのフォー マットを BioRuby のクラス名で指定しています。これについて詳しくは次の セクションを参照してください。

さらに、各 Bio::DB クラスの open メソッドで同様のことができるようになっ たので、ファイル名を指定する場合は

```
#!/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。

次に、GenBank の複雑な FEATURES の中もパースして、遺伝子ごとの塩基配列と アミノ酸配列を取り出してみます。

```
#!/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|

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

  gb.features.each do |feature|      # FEATURES の要素を一つずつ処理
    position = feature.position
    hash = feature.assoc             # レガシーだが簡単のためハッシュに直す

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

    # 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(', ')

    # 塩基配列
    puts ">NA splicing('#{position}') : #{gene_info}"
    puts gb.naseq.splicing(position)

    # アミノ酸配列（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= のもの）
    puts ">AA original translation"
    puts hash['translation']
  end
end
```

- 注：上記のように assoc メソッドで Feature オブジェクトからハッシュを生成 すると qualifier をキーとしてデータを取り出すことができるので便利ですが、 キーが同一の複数の qualifier が 1 つの feature 中に存在する場合、情報が 失われます（これを防ぐためにデフォルトではデータを配列で持たせています）。

ここで、splicing は GenBank フォーマットの position 表記を元に、塩基配 列から exon 部分を切り出したりする強力なメソッドです。もし遺伝子の切り 出しやアミノ酸への翻訳に BioRuby のバグがあれば、最後の 2 行で表示され るアミノ酸配列が異なる事になります。

この splicing メソッドの引数には GenBank の position 文字列の他に Bio::Locations オブジェクトを渡すこともできるようになっています。 position のフォーマットや Bio::Locations について詳しく知りたい場合は bio/location.rb を見てください。

- GenBank などの feature 行から持ってきた position の例

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

- あらかじめ Locations オブジェクトにしてから渡してもＯＫ

```
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 に対しても splicing メソッド で部分配列を取り出すことができるようになっています。

- アミノ酸配列のシグナルペプチドを切り出すとか

```
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やり方は同じで、 データベースの１エントリ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)
```

が、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
```

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。

FASTA による相同性検索を行う（Bio::Fasta クラス）

問い合わせ配列が FASTA 形式で入った query.pep がある時、ローカルとリモー トで FASTA 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（コマンド名が fasta34 でパ スが通っている場合の例で説明します）、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 形式で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、

```
#!/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
```

というスクリプトを f_search.rb という名前で作ったとすると、

```
% ./f_search.rb query.pep target.pep > f_search.out
```

のように実行すれば検索することができます。

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。上の例では Fasta オブジェクトの query メソッドを使って検索し ていますが、逆に問い合わせ配列に対し

```
seq = ">test seq¥nYQVLEEIGRGSFGSVRKVIHIPTKKLLVRKDIKYGHMNSKE"
seq.fasta(factory)
```

のように factory を渡して fasta メソッドを呼ぶ方法もあります。

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。ktup 値だけはメソッドで指定します。 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。特にヒットしたターゲットに 対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るようにしています。これらのメソッドの多くは後で 見るように Bio::Blast::Report と共通にしてあるのですが、FASTA 固有の値を 取り出すメソッドなどが必要な場合は、Bio::Fasta::Report クラスのドキュメ ントを参照してください。検索結果から様々な値をどのように取り出すかはスク リプト次第です。

さらに、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 のあとで factory オブジェクトの output メソッドを使えば 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) での検索を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 の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が決まります。

- 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'

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 と読み替えれば基本的には大丈夫です。

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 に対して検索する場合には Bio::Blast.remote を使います。 この引数で FASTA と異なるのは program です。

- 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'

をそれぞれ指定します。

ところで、Bio::Blast は、外部ライブラリに依存しないようにデフォルトでは -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ているので、-m 7 の XML 形式の出力を使うこと をお勧めします。Ruby の XMLParser か REXML ライブラリを別途インスト ールし ておけば、配列やアライメントを含む BLAST の全出力結果を使うことができます。 これらの XML ライブラリは処理速度が速い XMLParser, REXML の順で検索され、 インストールされていれば自動的に使われるようになります。

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
```

簡便のため、スコアなどいくつかの情報はベストの Hsp の値を Hit から参照し ています。

逆に、Hit の内部の Hsp オブジェクトを直接見ないと取れない値が必要な場合 や、各 Hsp を全部見たい場合、blastpgp で各 Iteration オブジェクト毎の値 が必要な場合などもあると思います。Bio::Blast::Report オブジェクトは実際 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 実行時のパラメータと統計情 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。ここで対応しているのは blastall -m 7 で実行した XML フォー マットの出力です。

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

require 'bio'

# XML 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

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、これらについて NCBI の意向で E-Utils というウェブプログラム 群を使うことが推奨されているので、今後は esearch, 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 とほぼ同じ処理をしていますが、 E-Utils の機能により、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、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（強調など文字の修飾はでき ないので実用には手直しが必要ですが）。

Bio::Reference クラスに合うように各データベースパーザが REFERENCE 行など を処理するのは少し大変なのですが、対応すれば BibTeX 形式などに変換できる のは便利ではないかと思います（人名など例外が多くて実際にはパーザを作るの はかなり面倒くさいです）。

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。

BioRuby のサンプルプログラムの使い方

BioRuby のパッケージには samples/ ディレクトリ以下にいくつかのサンプルプ ログラムが含まれています。古いものも混じっていますし、量もとても十分とは 言えないので、実用的で面白いサンプルの提供は歓迎です。

to be written...

OBDA

OBDA (Open Bio Database Access) とは、2002 年の１月と２月に Arizona と Cape Town の２回に分けて行われた BioHackathon において、BioPerl, BioJava, BioPython, BioRuby などの各プロジェクトを中心としたメンバー間で 合意された配列データベースへの共通アクセス方法です。

- 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

それぞれの詳細は <URL:http://obda.open-bio.org/> を参照してください。 各 spec は CVS で cvs.open-bio.org の obf-common/ 以下に置いてあります。

BioRegistry

設定ファイルを読み込んで、各データベースごとのエントリ取得方法を個人やサ イト毎のレベルで指定できるようにするものです。設定ファイルの検索は、

- 指定したファイル
- ~/.bioinformatics/seqdatabase.ini
- /etc/bioinformatics/seqdatabase.ini
- http://www.open-bio.org/registry/seqdatabase.ini

の順に行われます。BioRuby の実装では最初に見つかった設定が優先ですが、後 のファイルにしか書かれていない情報も追加されるようになっています。従って システム管理者が /etc/bioinformatics/ に置いた

設定ファイルのうち個人的に 変更したいものだけ ~/.bioinformatics/ で上書きするといった使い方ができる ようになっています。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取りに行きます。サンプルの seqdatabase.ini ファ イルが bioruby のソースパッケージに入っ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
```

のようなエントリを単位として必要なだけ定義します。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提案されていますが、今のところ BioRuby で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） さらにオプションが必要な場合があります。ここで protocol には

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

が指定できますが、今のところ開発者のマンパワー不足により BioRuby で扱え るのは index-flat, index-berkleydb, biofetch と biosql だけです。

BioRegistry を使うには、まず

```
reg = Bio::Registry.new
```

として設定ファイルを読み込みます。

```
# 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。外部ライブラリに依存しないシンプルな index-flat イン デックスと Berkeley DB (bdb) を使った index-berkeleydb インデックスの作 成を行うことができます。インデックスの作成には bioruby パッケージに附属 する bioflat コマンドを使って、

```
% bioflat --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。クラス名は BioRuby での各データベースのパーザ名前になり ますが、フォーマットの自動認識機能を内蔵しているので省略可能です。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 モジュールを別途インストールしてある場合は 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 オプションに bdb をつけます。

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 BioHackathon の間に 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、bioruby.org で運用してい ます。また、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 のものと EBI の ２ヶ所しかありません。

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 と一緒にインストールされる biofetch コマンドを用いる方法

   ```
   % biofetch 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')
   ```

最後の BioRegistry を使う場合は 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（この記述によって BioRegistry で生成さ れた Bio::Fetch サーバに対してはサーバの URL とデータベース名の指定が済 んでいることになります）。

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。）

BioSQL

to be written...

KEGG API

別ファイルの KEGG_API.rd.ja や

- <URL:http://www.genome.jp/kegg/soap/>

を参照してください。

APPENDIX

必要なライブラリのインストール

to be written...