実例で学ぶプログラミング言語
Julia 入門

v0.5 の新機能編

yomichi　著

コミックマーケット **90 (2016** 夏**)** 版　　**EREX** 工房 発行

まえがき

本書について

Julia は Jeff Bezanson, Stefan Karpinski, Viral B. Shah の 3 氏らによって開発された技術計算向けプログラミング言語で、2012 年に発表されました。Julia は高速で表現力が高く、また組み込み型と同等のユーザー定義型の追加やマクロによる構文の追加など、非常に強力な言語となっています。その一方で学習コストは低く、特にライブラリを組み合わせて使うだけならかなり単純・素直にプログラムを書くことが出来る言語です。日本でも 2014 年辺りから、主に R や Python などで統計処理や機械学習を行っていた人たちを中心として注目されてきています。

2016 年 7 月 26 日に次期安定バージョン候補である v0.5.0-rc0 が、8 月 4 日にその改訂版である v0.5.0-rc1 がリリースされました。過去バージョン (v0.4 や v0.3) のリリースタイミングから見るに、8 月下旬から 9 月頭にかけて、v0.5.0 がリリースされると考えられます。

さて、この v0.5 ですが、過去の例に負けず劣らず多彩な新機能追加や構文変更、標準ライブラリの変更が存在します。本書ではこれらの変更を概観していき、読者を次世代 Julian へと導いていければと思います。文字列、配列、関数の 3 つに関する大きな変更・追加点を特に見ていきます。Julia の文法などは、前著[*1] などを利用してください。

なお、節タイトルに付いている (S1.1, p1) などは、前著で関係のあるセクション番号およびページ数を表します。

[*1] 「実例で学ぶプログラミング言語 Julia 入門 v0.4.2 対応版」。PDF 版は次の URL からダウンロード可能です。http://yomichi.hateblo.jp/entry/2016/08/11/120654

おやくそく

本書の内容に基づく運用結果について、筆者は何ら責任を負いません。

連絡先

Copyright : Yuichi Motoyama 2016
E-mail : yomichi@tsg.jp
twitter : @yomichi_137

目次

第 1 章

文字列

1.1 型の統合 (S2.9, p27)

これまでは ASCII 文字列を表す型と Unicode 文字列を表す型が分かれていて、さらに Unicode は 3 種類のエンコードが用意されていましたが（ASCIIString, UTF8String, UTF16String, UTF32String）。v0.5 では、UTF16String と UTF32String がなくなり、さらに残った ASCIIString と UTF8String とが、String というひとつの型に統合されます[*1]。

抽象型 AbstractString はそのまま残り、String を含むので、関数の引数としてこれを使っていた場合はそのまま使い続けることができます。

```
julia-0.4.6> typeof("Julia")
ASCIIString
julia-0.4.6> typeof("Julia 言語")
UTF8String
julia-0.4.6> ASCIIString <: AbstractString
true
julia-0.4.6> UTF8String <: AbstractString
true
```

[*1] ASCIIString と UTF8String は v0.5 では deprecated 扱いとなります。

```
julia-0.5.0-rc1+0> typeof("Julia")
String
julia-0.5.0-rc1+0> typeof("Julia 言語")
String
julia-0.5.0-rc1+0> String <: AbstractString
true
```

UTF16String, UTF32String については、主な使用目的は外部ファイルや外部ライブラリの使用だと思われますが、LegacyStrings.jl で引き続き提供されます。

1.2 Unicode 9

Unicode 9.0 に対応しました。

第 2 章

配列

2.1 配列の配列

配列を並べた配列リテラルを書くと、(たぶん) 大多数の予想に反して自動的に結合されてしまいましたが、v0.5 からは配列の配列になります。結合したい場合は vcat 関数を使うか [;] を使ってください。

```
julia-0.4.6> a = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
julia-0.4.6> [a,a]
WARNING: [a,b] concatenation is deprecated; use [a;b] instead
 in depwarn at ./deprecated.jl:73
 in oldstyle_vcat_warning at
    /Users/yomichi/work/julia/julia-v0.4.6/lib/julia/sys.dylib
 in vect at ./abstractarray.jl:32
while loading no file, in expression starting on line 0
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3
julia-0.4.6> [a;a]
6-element Array{Int64,1}:
```

```
 1
 2
 3
 1
 2
 3
```

```
julia-0.5.0-rc1+0> a = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
julia-0.5.0-rc1+0> [a,a]
2-element Array{Array{Int64,1},1}:
 [1,2,3]
 [1,2,3]
julia-0.5.0-rc1+0> [a;a]
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3
```

2.2 reshape (S3.3.6, p46)

reshape 関数で生成した配列は、元のものとメモリを共有したりしていなかったりしていました。

```
# 共有しているケース
julia-0.4.6> A = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
```

```
 3
 4
julia-0.4.6> B = reshape(A,2,2)
2x2 Array{Int64,2}:
 1  3
 2  4
julia-0.4.6> B[1] = 42
42
julia-0.4.6> A
4-element Array{Int64,1}:
 42
  2
  3
  4

# 共有していないケース
# 1:4 は immutable なのでコピーを返す
julia-0.4.6> C = 1:4
1:4
julia-0.4.6> D = reshape(C, 2,2)
2x2 Array{Int64,2}:
 1  3
 2  4
julia-0.4.6> D[1] = 137
137
julia-0.4.6> C
1:4
```

v0.5 からは、ReshapedArray というビューオブジェクトを返すことで、常に元とメモリを共有するようになっています。コピーを返したいときは、collect や copy を明示的に使ってください。

```
julia-0.5.0-rc1+0> A = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
 3
 4
julia-0.5.0-rc1+0> B = reshape(A, 2, 2)
2 × 2 Array{Int64,2}:
 1  3
 2  4
```

```
julia-0.5.0-rc1+0> B[1] = 42
42
julia-0.5.0-rc1+0> A
4-element Array{Int64,1}:
 42
  2
  3
  4

# 1:4 は immutable なので、D の中身を変えようとするとエラーを吐く
julia-0.5.0-rc1+0> C = 1:4
1:4
julia-0.5.0-rc1+0> D = reshape(C, 2, 2)
2 × 2 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  3
 2  4
julia-0.5.0-rc1+0> D[1] = 137
ERROR: indexed assignment fails for a reshaped range;
       consider calling collect
 in setindex!(::Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}},
              ::Int64, ::Int64)
 at ./reshapedarray.jl:151
julia-0.5.0-rc1+0> D = collect(reshape(C, 2, 2))
2 × 2 Array{Int64,2}:
 1  3
 2  4
julia-0.5.0-rc1+0> D[1] = 137
137
julia-0.5.0-rc1+0> C
1:4
```

2.3 切り出し (S2.8, p25)

多重配列から部分配列を切り出すとき、コロンや添字の配列などではなく、単一の整数で指定された次元は、すべて部分配列から削減されます。この挙動が嫌な場合は、コロンを使って `A[:,:,1:1]` などとしてください。

```
julia-0.4.6> A = reshape(1:8, 2,2,2)
2x2x2 Array{Int64,3}:
```

```
[:, :, 1] =
 1  3
 2  4
[:, :, 2] =
 5  7
 6  8
julia-0.4.6> A[1,:,:]
1x2x2 Array{Int64,3}:
[:, :, 1] =
 1  3
[:, :, 2] =
 5  7
julia-0.4.6> A[:,1,:]
2x1x2 Array{Int64,3}:
[:, :, 1] =
 1
 2
[:, :, 2] =
 5
 6
julia-0.4.6> A[:,:,1]
2x2 Array{Int64,2}:
 1  3
 2  4
```

```
julia-0.5.0-rc1+0> A = reshape(1:8, 2,2,2)
2 × 2 × 2 Base.ReshapedArray{Int64,3,UnitRange{Int64},Tuple{}}:
[:, :, 1] =
 1  3
 2  4
[:, :, 2] =
 5  7
 6  8
julia-0.5.0-rc1+0> A[1,:,:]
2 × 2 Array{Int64,2}:
 1  5
 3  7
julia-0.5.0-rc1+0> A[:,1,:]
2 × 2 Array{Int64,2}:
 1  5
 2  6
julia-0.5.0-rc1+0> A[:,:,1]
```

```
2 × 2 Array{Int64,2}:
 1  3
 2  4
```

2.4 リスト内包表記 (S3.3.5, p46)

2.4.1 多次元配列

リスト内包表記で出来上がる配列の次元は、以前は一次元となっていましたが、v0.5 からはリスト内包表記の中で用いたものと同じになります。なお、多重 for に関しては以前と同じく一次元配列が得られます。

```
julia-0.4.6> A = rand(2,2)
2x2 Array{Float64,2}:
 0.861165  0.0713677
 0.162392  0.309633
julia-0.4.6> B = [x for x in A]
4-element Array{Any,1}:
 0.861165
 0.162392
 0.0713677
 0.309633
```

```
julia-0.5.0-rc1+0> A = rand(2,2)
2 × 2 Array{Float64,2}:
 0.00934896  0.745626
 0.982493    0.537303
julia-0.5.0-rc1+0> B = [x for x in A]
2 × 2 Array{Float64,2}:
 0.00934896  0.745626
 0.982493    0.537303
julia-0.5.0-rc1+0> C = [i+j for i in 10:10:30 for j in 1:3]
9-element Array{Int64,1}:
 11
```

```
12
13
21
22
23
31
32
33
```

2.4.2 型推論

リスト内包表記の要素の型は、以前は関数から推論されていましたが、v0.5 では実際に生成された要素の型から推論されます。その結果として、以前よりも具体的な型が得られます。

```
julia-0.4.6> A = [complex(1.0)]
1-element Array{Complex{Float64},1}:
 1.0+0.0im
julia-0.4.6> B = [sqrt(x) for x in A]
1-element Array{Any,1}:
 1.0+0.0im
```

```
julia-0.5.0-rc1+0> A = [complex(1.0)]
1-element Array{Complex{Float64},1}:
 1.0+0.0im
julia-0.5.0-rc1+0> B = [sqrt(x) for x in A]
1-element Array{Complex{Float64},1}:
 1.0+0.0im
```

2.4.3 if によるフィルタリング

リスト内包表記の for 文の直後に if cond とおくことで、cond が真の時だけを通すようなフィルタリングをすることができます。多重 for の時は、フィルタリングしたい for ループの直後に if が必要です。なお、フィルタリングをつけると、多次元配列から作ったリスト内包表記でも一次元配列が返ってきます[*1]。

```
julia-0.5.0-rc1+0> [x for x in 1:10 if iseven(x)]
5-element Array{Int64,1}:
  2
  4
  6
  8
 10
julia-0.5.0-rc1+0> [10x+y for x in 1:5 if iseven(x) for y in 1:5 if isodd(y)]
6-element Array{Int64,1}:
 21
 23
 25
 41
 43
 45
julia-0.5.0-rc1+0> [x for x in randn(3,3) if x>0]
3-element Array{Float64,1}:
 1.19474
 0.577494
 0.948306
```

2.4.4 ジェネレータ式

リスト内包表記で [] の代わりに () を使うことで、配列の代わりにイテレータを返す、いわゆるジェネレータ式が使えるようになりました[*2]。

[*1] 要素数が減りうるため。cond == true なる要素をゼロ埋めしたい場合は、ifelse(cond, x, zero(x)) という具合に明示的にゼロを埋めましょう。

[*2] この節で紹介したリスト内包表記の新機能は、そのままジェネレータ式でも有効です。

非常に長い配列を扱うが、一度に全部保持する必要が無い場合は、ジェネレータ式を使うことで計算時間と使用メモリとをともに抑えることができるかもしれません。なお、関数の引数として直接渡す場合は、() を省略可能です。

```
# list
julia-0.5.0-rc1+0> @time sum([2x for x in 1:(1<<24)])
  0.145616 seconds (12.02 k allocations: 128.517 MB, 42.18% gc time)
281474993487872
# generator
julia-0.5.0-rc1+0> @time sum( 2x for x in 1:(1<<24) )
  0.020334 seconds (16.85 k allocations: 707.449 KB)
281474993487872
```

ジェネレータ式ができたことで、辞書型の内包表記はジェネレータ式を用いた `Dict(x=>f(x) for x in xs)` という形式になりました。

第 3 章

関数

3.1 ラムダ式、クロージャの強化 (S3.4.1, S3.4.2, p50-53)

3.1.1 型の変更

これまではすべての関数、無名関数は単一の型 Function の値だったのですが、v0.5 からはすべて固有の型をもち、Function はそのすべての型の抽象型となります。クロージャの環境は型のフィールドとして保存されます。

```
julia-0.5.0-rc1+0> bind1st(f,a) = (xs...) -> f(a,xs...)
bind1st (generic function with 1 method)
julia-0.5.0-rc1+0> p42 = bind1st(+,42)
(::#9) (generic function with 1 method)
julia-0.5.0-rc1+0> p42.
a f
julia-0.5.0-rc1+0> p42.a
42
julia-0.5.0-rc1+0> p42.f
+ (generic function with 163 methods)
julia-0.5.0-rc1+0> p42(1,2,3)
48
julia-0.5.0-rc1+0> typeof(f)
##7#8
julia-0.5.0-rc1+0> supertype(typeof(f))
Function
```

3.1.2　無名関数の速度向上

無名関数に固有の型がついた結果、JIT コンパイルが働くようになり、速度が向上します。do 構文や高階関数は、書くのは楽だけれど（普通の関数と比べて）速度上のオーバーヘッドが大きくて使いづらかったのですが、これからは気兼ねなく使っていけるようになります。

```
julia-0.4.6> function foo()
       a = randn(2000,2000)
       @time 2a
       @time map(x->2x, a)
       end
foo (generic function with 1 method)
# JIT コンパイル
julia-0.4.6> foo();
  0.040393 seconds (2 allocations: 30.518 MB, 62.67% gc time)
  0.307757 seconds (8.12 M allocations: 159.413 MB, 19.38% gc time)
julia-0.4.6> foo();
  0.020330 seconds (2 allocations: 30.518 MB, 15.97% gc time)
  0.183909 seconds (8.00 M allocations: 152.588 MB, 26.11% gc time)
```

```
julia-0.5.0-rc1+0> function foo()
       a = randn(2000,2000)
       @time 2a
       @time map(x->2x, a)
       end
foo (generic function with 1 method)
# JIT コンパイル
julia-0.5.0-rc1+0> foo();
  0.073091 seconds (2 allocations: 30.518 MB, 74.73% gc time)
  0.089694 seconds (4 allocations: 30.518 MB, 80.71% gc time)
julia-0.5.0-rc1+0> foo();
  0.010962 seconds (2 allocations: 30.518 MB, 2.12% gc time)
  0.010005 seconds (4 allocations: 30.518 MB, 2.27% gc time)
```

3.1.3 キーワード引数

無名関数にもキーワード引数を使えるようになります。

```
julia-0.4.6> f = (x;kwd=42) -> x+kwd
ERROR: syntax: "begin
    x
    kwd=42
end" is not a valid function argument name
```

```
julia-0.5.0-rc1+0> bind1st(f,a) = (xs...) -> f(a,xs...)
bind1st (generic function with 1 method)
julia-0.5.0-rc1+0> p42 = bind1st(+,42)
(::#9) (generic function with 1 method)
julia-0.5.0-rc1+0> p42.
a f
julia-0.5.0-rc1+0> p42.a
42
julia-0.5.0-rc1+0> p42.f
+ (generic function with 163 methods)
julia-0.5.0-rc1+0> p42(1,2,3)
48
julia-0.5.0-rc1+0> supertype(typeof(f))
Function
julia-0.5.0-rc1+0> typeof(f)
##7#8
```

3.2 返り値の型指定

関数そのものに型指定 `::T` をつけると、返り値の型指定をすることができます。返り値 `ret` に対して自動的に型変換関数 `convert(::Type{T}, ret)` が呼ばれます。

caption

```
julia-0.5.0-rc1+0> import Base.convert
julia-0.5.0-rc1+0> type A
       x::Int
       end
julia-0.5.0-rc1+0> convert(::Type{Int}, a::A) = a.x
convert (generic function with 596 methods)
julia-0.5.0-rc1+0> a = A(42)
A(42)
julia-0.5.0-rc1+0> Int(a)
42
julia-0.5.0-rc1+0> foo(a::A)::Int = a
foo (generic function with 1 method)
julia-0.5.0-rc1+0> foo(a)
42
```

3.3 関数呼び出し

ユーザ定義型の値を関数として扱うためのインターフェースが `call{a::Type{A}, xs...) = ...` から `(a::A)(xs...) = ...` に変更になりました。

```
julia-0.4.6> import Base.call
julia-0.4.6> type PlusA y end
julia-0.4.6> call(a::PlusA, x) = x + a.y
call (generic function with 1036 methods)
julia-0.4.6> a = PlusA(42)
PlusA(42)
julia-0.4.6> a(137)
179
```

```
julia-0.5.0-rc1+0> type PlusA y end
julia-0.5.0-rc1+0> (a::PlusA)(x) = x+a.y
julia-0.5.0-rc1+0> a = PlusA(42)
PlusA(42)
```

```
julia-0.5.0-rc1+0> a(137)
179
```

3.4 可変長引数の長さ取得 (S3.3.6, p46)

Vararg{T,N} を引数として使ってパラメトリック関数を作ることで、可変長引数の長さ N を用いた関数多重ディスパッチをすることができます。次の例では、foo は以前の書き方[*1]で、bar は新しい書き方をしています。

foo は引数を何個与えても同じメソッド (39322) が呼ばれますが、bar の方は引数の数を変えるとそれぞれ別メソッドが呼ばれます (69331, 69334, 69447)。また、メソッド内部で分岐がなくなっています。ただ、多重ディスパッチで関数本体は最適化されますが、コンパイル時間やデータ容量、ディスパッチのオーバーヘッドなどのトレードオフに負けるかもしれない、という可能性は頭に入れておいても良いでしょう[*2]。

```
julia-0.5.0-rc1+0> foo{T}(xs::Vararg{T}) = length(xs) < 3 ? 0 : 1
foo (generic function with 1 method)

julia-0.5.0-rc1+0> @code_llvm foo(1)
define %jl_value_t* @julia_foo_69322(%jl_value_t*, %jl_value_t**, i32) #0 {
top:
  %3 = alloca %jl_value_t**, align 8
  store volatile %jl_value_t** %1, %jl_value_t*** %3, align 8
  %4 = icmp sgt i32 %2, 2
  %. = select i1 %4, %jl_value_t* inttoptr (i64 4553719968 to %jl_value_t*),
       %jl_value_t* inttoptr (i64 4553719904 to %jl_value_t*)
  ret %jl_value_t* %.
}

julia-0.5.0-rc1+0> @code_llvm foo(1,2)
define %jl_value_t* @julia_foo_69322(%jl_value_t*, %jl_value_t**, i32) #0 {
top:
```

[*1] foo{T}(xs::T...) = ... と同じです。

[*2] 実際測ったわけでもないし、さらにほとんどのケースでは気にするまでもないとも思いますが……。

```
  %3 = alloca %jl_value_t**, align 8
  store volatile %jl_value_t** %1, %jl_value_t*** %3, align 8
  %4 = icmp sgt i32 %2, 2
  %. = select i1 %4, %jl_value_t* inttoptr (i64 4553719968 to %jl_value_t*),
       %jl_value_t* inttoptr (i64 4553719904 to %jl_value_t*)
  ret %jl_value_t* %.
}


julia-0.5.0-rc1+0> bar{T,N}(::Vararg{T,N}) = N < 3 ? 0 : 1
bar (generic function with 1 method)

julia-0.5.0-rc1+0> @code_llvm bar(1,2)
define %jl_value_t* @julia_bar_69331(%jl_value_t*, %jl_value_t**, i32) #0 {
top:
  %3 = alloca %jl_value_t**, align 8
  store volatile %jl_value_t** %1, %jl_value_t*** %3, align 8
  ret %jl_value_t* inttoptr (i64 4553719904 to %jl_value_t*)
}

julia-0.5.0-rc1+0> @code_llvm bar(1,2,3)
define %jl_value_t* @julia_bar_69334(%jl_value_t*, %jl_value_t**, i32) #0 {
top:
  %3 = alloca %jl_value_t**, align 8
  store volatile %jl_value_t** %1, %jl_value_t*** %3, align 8
  ret %jl_value_t* inttoptr (i64 4553719968 to %jl_value_t*)
}

julia-0.5.0-rc1+0> @code_llvm bar(1,2,3,4)
define %jl_value_t* @julia_bar_69337(%jl_value_t*, %jl_value_t**, i32) #0 {
top:
  %3 = alloca %jl_value_t**, align 8
  store volatile %jl_value_t** %1, %jl_value_t*** %3, align 8
  ret %jl_value_t* inttoptr (i64 4553719968 to %jl_value_t*)
}
```

第 4 章

その他

4.1　スレッド並列の導入

まだ実験段階ですが、スレッド並列機能が導入されました。for ループのみがスレッド並列化できます。静的なブロック分割のみがサポートされています。なお、標準入出力への print や sleep などの幾つかの関数（ライブラリ呼び出し）は、スレッドセーフではない関係上セグメンテーション違反で強制終了します。

この機能を使うためには、例えば最大 4 スレッド 使いたい時には、Julia のビルド時に Make.user というファイルを作り、JULIA_THREADS = 4 と書き込んでからビルドしてください。

```
julia-0.5.0-rc1+0> Threads.nthreads()
4
julia-0.5.0-rc1+0> xs = zeros(Int, 10);
julia-0.5.0-rc1+0> Threads.@threads for i in 1:10
       xs[i] = Threads.threadid()
       end
julia-0.5.0-rc1+0> xs
10-element Array{Int64,1}:
 1
 1
 1
 2
 2
 2
 3
 3
```

```
 4
 4

julia-0.5.0-rc1+0> @time Libc.systemsleep(1)
  1.001654 seconds (133 allocations: 7.703 KB)
0

julia-0.5.0-rc1+0> @time Threads.@threads for i in 1:10
       Libc.systemsleep(1)
       end
  3.015814 seconds (9.71 k allocations: 392.098 KB)

julia-0.5.0-rc1+0> files = [open("$i.dat", "w") for i in 1:10];

julia-0.5.0-rc1+0> Threads.@threads for f in files
       println(f, Threads.threadid())
       end

shell> cat 1.dat
1
shell> cat 10.dat
4
```

4.2 辞書型に対する map

辞書型に対する map 系列の関数が受け取れる関数が、Pair 型から Pair 型への関数のみになりました。そのかわり、結果が配列ではなく辞書となります[*1]。

```
julia-0.4.6> d = Dict("Kagerou" => 1, "Shiranui" => 2)
Dict{ASCIIString,Int64} with 2 entries:
  "Kagerou"  => 1
  "Shiranui" => 2

julia-0.4.6> map(d) do kv
       kv[2] => kv[1]
       end
2-element Array{Any,1}:
```

[*1] 配列を得たい場合には、リスト内包表記を使うことになります。

```
 1=>"Kagerou"
 2=>"Shiranui"
```

```
julia-0.5.0-rc1+0> d = Dict("Kagerou" => 1, "Shiranui" => 2)
Dict{String,Int64} with 2 entries:
  "Kagerou"  => 1
  "Shiranui" => 2

julia-0.5.0-rc1+0> map(d) do kv
       kv[2] => kv[1]
       end
Dict{Int64,String} with 2 entries:
  2 => "Shiranui"
  1 => "Kagerou"

julia-0.5.0-rc1+0> map(d) do kv
       kv[1]
       end
ERROR: MethodError: no method matching similar(::Dict{String,Int64},
                                               ::Type{String})

julia-0.5.0-rc1+0> [kv[1] for kv in d]
2-element Array{String,1}:
 "Kagerou"
 "Shiranui"
```

4.3 関数名の変更・廃止

たくさんありますが、前著で触れた関数を紹介します。

4.3.1 BigFloat の精度まわり (S2.7, p20)

多倍長浮動小数点数の精度を取得および変更する関数 `get_bigfloat_precision()`, `set_bigfloat_precision(bit)`, `with_bigfloat_precision(f, bit)` が、それぞれ `precision(BigFloat)`,

setprecision(bit), setprecision(f, bit) に変更になりました。

4.3.2 ストリームからの読み出し (S3.4.4, p54)

ストリームからの読み出し関数のうち、すべてを単一文字列として読み込む readall と指定バイト読み込む readbytes とが、それぞれ readstring と read へと変更・統合されました。

4.3.3 基本型

与えた型の直接の基本型を返す super 関数[*2]は、supertype へと変更になりました。

4.4 トピック

その他の一般ユーザが触りそうな変更

- ユニットテストが大幅強化された
- マクロが多重ディスパッチに対応した
- using, import などが、システムによらず大文字小文字を区別するようになった
- 配列の要素ごとの演算やブロードキャスト演算の挙動や宣言が変更された
- nothing を返す map 関数としての foreach 関数が導入された
- PROGRAM_FILE グローバル変数でスクリプト名が取得可能になった
- Pkg.update() が引数を取れるようになり、個別の更新が可能になった

[*2] 前著で触れたと思っていましたが、意外にも扱ってませんでした。割とよく使う関数なので紹介しておきます。

あとがき

今までは LaTeX でベタ書きしていたんですが、なんとなく今回は Re:VIEW[*3] を使ってみました。情報がいろいろ散在していたりそもそもなかったり、せっかくあっても実は (少なくとも LaTeX では) 未実装だったりと、まだまだ使いこなしづらいところがありますが、使い込むとなかなか楽しそう＆今後が楽しみなツールだと思います。とりあえず LaTeX の数式が書けて、 LaTeX よりも手軽に章立てや図表、箇条書きなどが書けるのは素晴らしいですね。あとは索引機能なんかがうまく動くとかなり良さそう。

国内でもじわじわ利用者とか知名度が増えてきた感じで、でもやっぱりよくわからない Julia ですが、9 月 3 日には JuliaTokyo 6 が白金台で開催されるので、興味の出た方はどうぞ。参加者埋まっていますが、LT 発表すれば申し込めるはず……！

次回冬コミに受かったら、入門本のバージョンを上げるのと、発展トピックとしては並列化を扱えたらいいなと思います。シン・ゴジラでもスパコン出てきて海外と並列計算するとかいう謎の胸熱展開やりましたし！ 流石に通信速度とかスケーリングとかを考えると、大規模並列よりはグリッドコンピューティングとかパラメータ並列だとは思いますが。

[*3] http://reviewml.org/ja/, https://github.com/kmuto/review

実例で学ぶプログラミング言語 **Julia** 入門

2016 年 8 月 14 日　初版第 1 刷　発行
著　者　yomichi
発行所　EREX 工房
印刷所　セルフコピー機@キンコーズ