

チュートリアル講演

Julia 入門

Satoshi Terasaki (AtelierArith)

2023/07/10

この資料について

- [数学と物理におけるJuliaの活用](#) でのチュートリアル講演資料
- [piever/Remark.jl](#) を使って Markdown 原稿をスライドとして表示

更新日: 2023-07-10T03:59:43.607

本日の資料は [AtelierArith/julia_tutorial](#) にあります.

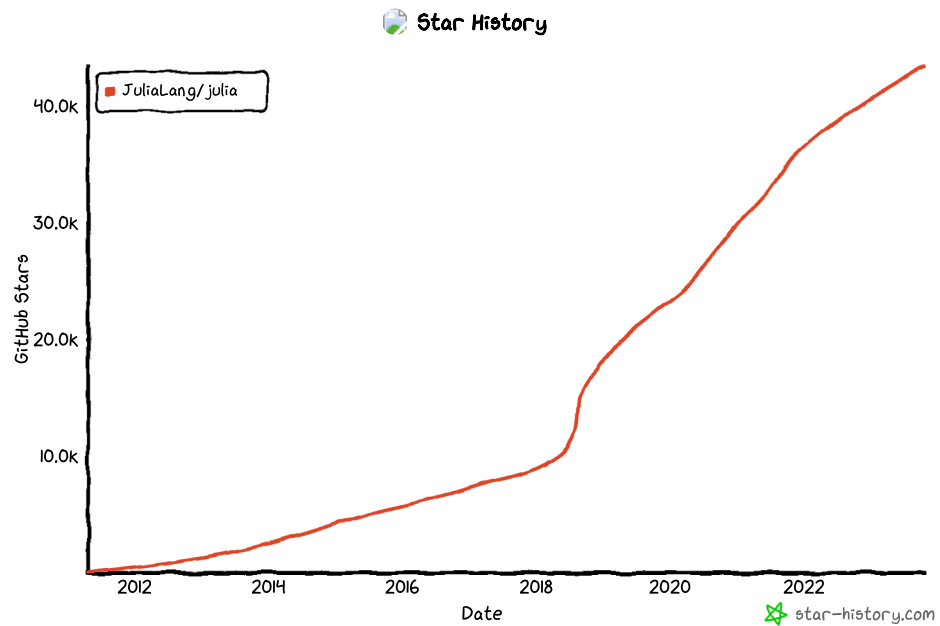
ここでの Julia とは

- プログラミング言語の一つ
- 2012 年 [Why We Created Julia](#) に発表. 開発は 2009 年ごろから
 - [Jeff Bezanson](#), [Stefan Karpinski](#), [Viral B. Shah](#), and [Alan Edelman](#)
 - 冒頭で In short, because we are greedy. とあるように既存のプログラミング言語の良いところを取り入れた言語
- 個人的に好きな箇所
 - We want the speed of C with the dynamism of Ruby
 - We want something as usable for general programming as Python
 - Something that is dirt simple to learn, yet keeps the most serious hackers happy.
 - (Did we mention it should be as fast as C?)
- 要するに
 - 高級言語のように使いやすく計算機の上で高速に動作する
 - 速く動くコードを早く書くことができる

Why We Use Julia, 10 Years Later

- 2022 年は 10 周年記念 🎉 [Why We Use Julia, 10 Years Later](#) が公開
 - [JuliaLang/www.julialang.org](https://github.com/JuliaLang/www.julialang.org) リポジトリで管理されている
 - [この原稿](#) にみんなが集まって書いたもの
 - 例: miguelraz さんの [Pull Request](#) など
- Julia というプログラミング言語に出会ったきっかけが紹介されている. 例えば以下のようなもの:
 - [Why We Created Julia](#) を読んでソースをビルドした
 - 同僚, 職場, 講義で出会った
 - C/C++/Fortran は難しい..., Python は簡単に使えるけれど遅い...
 - 気づいたら特定のパッケージのメンテナになっていた

Star History



直近の動き

- Juila 1.9 がリリース
 - TTFX 問題が改善される !!!
 - 可視化ツールの使用が捗る！
- デバッグまわりのツールが増えてきた
- 鈍器(褒め言葉)扱いの 実践Julia入門, Juliaプログラミング大全が登場してきた
- いろんな本が今年は出版されるらしい

時代は Julia なのでは？

そこでチュートリアル講演です

概要: ライトユースにも使いたいけれどもそれで高速性を犠牲にしたくない。総合的に開発できるものにしたい。超初心者にも習得は容易でありながらハッカーの満足にも応えられるものがほしい。」そういった願いからプログラミング言語 Julia が誕生しました。直近では v1.9 がリリースされ長年課題であったパッケージの読み込み時間の短縮など開発体験の改善が行われました。さらに和書の入門書も充実してきており学習を始めるには絶好のタイミングといえるでしょう。本チュートリアル講演では Julia の入門として

1. インストールと基本的な使い方 (<- このスライドの担当箇所)
2. 型と多重ディスパッチ
3. スレッド並列の基礎


3 つのトピックを解説します。この機会に Julia に入門し新しい体験を始めてみませんか。

Julia のインストール

Julia のインストール

- 要するに `julia` というコマンドが利用できれば良い

```
$ julia
```



```
Documentation: https://docs.julialang.org
Type "?" for help, "]"? for Pkg help.
Version 1.9.1 (2023-06-07)
Official https://julialang.org/ release

julia>
```

- すでに使える人はスキップする

素直な方法

- Julia の公式サイトから入手 <https://julialang.org/downloads/> 移動し 各自の環境に合わせて導入

それはそうだが

- 任意の人間が「各自の環境に合わせて導入してください」ができれば苦労しない
- 各自の環境が何とか, 環境変数とかパスを通すとか
- バージョンが上がる度に手動でインストールしたいか?

それはそうだが

- 任意の人間が「各自の環境に合わせて導入してください」ができれば苦労しない
- 各自の環境が何とか, 環境変数とかパスを通すとか
- バージョンが上がる度に手動でインストールしたいか?

そんなあなたに Juliaup

Juliaup - Julia version manager を使う方法 (1)

Windows

パワーシェルを開く.

```
PS> winget install julia -s msstore
```

[Windows アプリケーションから](#) も入手ができる

Juliaup - Julia version manager を使う方法 (2)

[juliaup/depoy/shellscript/_juliaup-init.sh](#) を叩いている.

Mac and Linux

```
$ curl -fsSL https://install.julialang.org | sh
$ source ~/.bashrc
$ julia --version
julia version 1.9.1
```

--yes オプションを使えば途中の対話操作を省略し進めることができる. [command-line-arguments](#) をみよ.

```
$ curl -fsSL https://install.julialang.org | sh -s -- --yes
```

動作確認

インストールができたと仮定して進める。 julia というコマンドを使うことができるか確認をする:

```
$ julia --version
julia version 1.9.1
```

何も指定しない場合 REPL(Read-Eval-Print Loop) が起動する:

```
$ julia

      _       _       _
     / \     / \     / \
    _/  \_   _/  \_   _/  \_
   /_    \_/  _/    \_/  _/    \_/
  /_      \_  /_      \_  /_      \_
 /_        \_/_        \_/_        \_
/_          \_/_          \_/_          \_

Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.9.1 (2023-06-07)
Official https://julialang.org/ release

julia>
```

Appendix: Julia 自体のアップデート

- [Julia v1.9.2 has been released](#) というアナウンスが出た
- Juliaup 経由で入れていると次のような通知が出る

```
$ julia
The latest version of Julia in the `1.9` channel is 1.9.2+0.x64.apple.darwin14. You currently have `1.9.1+0.x64.apple.darwin14`.

juliaup update

to install Julia 1.9.2+0.x64.apple.darwin14 and update the `1.9` channel to that version.
```

```
$ juliaup update

Installing Julia 1.9.2+0.x64.apple.darwin14
Downloading: [> ] 1.35 MiB/114.18 MiB eta: 14m
```

Appendix: Juliaup 以外の選択肢

- [abelsiqueira/jill](#)
 - jill - Julia Installer 4 Linux - Light
- [johnnychen94/jill.py](#)
 - `pip3 install jill && jill install`
- [Docker](#) を用いて `docker run -it --rm julia:1.9.1` のようにしてコンテナを起動する

参考資料

- 1.9 系が現時点で最新安定版. もし 1.10 がリリースされると 1.9 系のサポート（バグ修正など）はしなくなる.
- 特定のバージョンを長く使いたい場合は LTS (1.6 系) を使うと良い
- [Julia's Release Process](#)

Julia を動かす (REPL)

なぜ REPL で動かすのか?

もちろんファイルにコードを記述し次のように動かしても良い.

```
$ julia script.jl
```

- Julia は JIT コンパイルによって動作する言語
- `julia script.jl` することにパッケージのロード関数のコンパイルが行われるそのためのコストは無視できない
 - コンパイル結果を使い回し効率よく作業する必要がある
- 試行錯誤時は REPL の上で作業をするのがよく行われる
 - REPL の上でだけで学べることが多い
 - Python と異なりインデントに関してセンシティブではないのでコードをコピー&ペーストにより自由に実行させることができる
- もちろん VS Code, Jupyter, Pluto などのリッチな環境でも良い

VS Code を使いたい場合

- [VS Code のホームページ](#)
- [エクステンションの導入](#)
- [いくつか便利なショートカットキー](#)
 - ファイルにフォーカスを当てて Shift + Enter を押す
 - Alt-J Alt-O で REPL を開く

Pluto.jl 使いたい場合

- [Pluto.jl 入門](#)
 - JuliaTokai で話した勉強会資料

Julia を動かす (REPL)

`versioninfo()` を使うと詳細な情報を得ることができる。バグレポートなどに添付すると良い。

```
$ julia
```

```
(_) | Documentation: https://docs.julia-lang.org  
(_) |  
[ ] | Type "?" for help, "]"? for Pkg help.  
[ ] |  
[ ] | Version 1.9.1 (2023-06-07)  
[ ] | Official https://julia-lang.org/ release  
[ ] |  
[ ] |  
[ ] |
```

```
julia> versioninfo()  
Julia Version 1.9.1  
Commit 147bdf428cd (2023-06-07 08:27 UTC)  
Platform Info:  
OS: macOS (x86_64-apple-darwin22.4.0)  
CPU: 16 × Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz  
WORD_SIZE: 64  
LIBM: libopenlibm  
LLVM: libLLVM-14.0.6 (ORCJIT, skylake)  
Threads: 1 on 16 virtual cores  
Environment:
```

Julia を動かす (REPL)

REPL を使う (Julian mode)

```
julia> 1 + 1 # 簡単な算数
2

julia> println("Hello World")
Hello World

julia> print("Hello"); print(" "); print("World") # セミコロンで繋げることもできる
Hello World
julia>
```

julia> の部分はプロンプトと呼ばれる. julia> の部分も含めてコピペしても REPL 側が適切に処理する.

```
julia> 1 == 2
true # 実際は false なので実際に動かすと false になる
julia> println("Hello")
World # 実際は Hello が出るはず
```

- Python と異なりインデントに関してセンシティブではない
- コードを適当にコピペして自由に実行させることができる

Example

正の整数 a, b を選ぶ. $\gcd(a, b)$ が 1 となる確率が $1/\zeta(2) = 6/\pi^2$ となることを数値計算で確認する様子

```
julia> function calcn(N)
    cnt = 0
    for a ∈ 1:N # ∈ は \in + <tab> と入力する. `in` でも良い
        for b in 1:N # `in` の代わりに `=` と書いても良い
            if gcd(a, b) == 1 # gcd(a, b) == 1 && (cnt += 1) とすることもできる
                cnt += 1
            end
        end
    end
    prob = cnt / N / N
    return sqrt(6/prob) # sqrt は \sqrt + <tab>
end
calcn (generic function with 1 method)
julia> @time calcn(10^4)
1.791594 seconds
3.141534239016629 # 円周率に近い値
```

(上と同じコードを Python で書いてみると面白いかも. 少なくとも数十秒かかる.)

REPL を使う (Help mode)

- 例えば $x^{(1)}$, x_k , $\hat{\theta}$, \perp のような文字も使うことができる
- どう入力すべきなのか? REPL が教えてくれる!!!

```
julia> # ここで ? を押す  
help?> ? $\hat{\theta}$   
" $\hat{\theta}$ " can be typed by \theta<tab>\hat<tab>
```

通常の Julia mode で呼び出すこともできる.

```
julia> (Base.Docs.doc)((Base.Docs.Binding)(Main, : $\hat{\theta}$ ))
```

上記のコードの代わりに下記のようにして @doc マクロを使うこともできる :

```
julia> @doc  $\hat{\theta}$ 
```

寄り道(マクロについて)

マクロは呼ばれる式の断片を受け取り新しい式を生成する.

例えば `@doc` $\hat{\theta}$ は `(Base.Docs.doc)((Base.Docs.Binding)(Main, : $\hat{\theta}$))` を生成する. その様子は次のようにして確認できる:

```
julia> @macroexpand @doc  $\hat{\theta}$ 
:((Base.Docs.doc)((Base.Docs.Binding)(Main, : $\hat{\theta}$ )))
```

入力の仕方の他に, 関数の使い方を知ることができる:

```
help?> ndims
ndims(A::AbstractArray) -> Integer

Return the number of dimensions of A.

See also: size, axes.

Examples
=====

julia> A = fill(1, (3,4,5));

julia> ndims(A)
3
```

寄り道(マクロについて)

1 の原始 3 乗根 $\omega = \exp(2\pi\sqrt{-1}/3)$ をそのままコードに落とし込むと次のようになる:

```
julia> ω = exp(im * 2π/3) # im は虚数単位
-0.4999999999999998 + 0.8660254037844387im
julia> @show ω
ω = -0.4999999999999998 + 0.8660254037844387im
-0.4999999999999998 + 0.8660254037844387im
julia> ω ^ 3 # 3 乗するとほぼほぼ 1 になる.
0.9999999999999998 - 6.106226635438361e-16im
julia> @assert ω ^ 3 ≈ 1 # "≈" can be typed by \approx<tab>
```

マクロを使わないと次のようなコードを書いていることになる:

```
julia> ω = exp(im * 2π/3)
-0.4999999999999998 + 0.8660254037844387im
julia> println("ω = ", repr(begin dummyvariable = ω end)); dummyvariable
ω = -0.4999999999999998 + 0.8660254037844387im
-0.4999999999999998 + 0.8660254037844387im
julia> if ω ^ 3 ≈ 1
    nothing
else
    Base.throw(Base.AssertionError("ω ^ 3 ≈ 1"))
end
```

寄り道(フォントの問題)

文字化けする場合は入力に用いるフォントを変更するとよい。例えば次の候補がある:

- [cormullion/juliamono](#)
- [tohgarashi/JuGeM](#)
- [yuru7/juisee](#)
- [miiton/Cica](#)

例えば macOS の場合 ~/Library/Fonts/ 直下に .ttf 拡張子のファイルを配置する。

```
julia> using Plots; plot(sin, label="三角関数", title="日本語", fontfamily="JuGeM-Regular")
```

Pkg mode

パッケージマネージャが付属している。

```
julia> using Pkg; Pkg.add("Example") # JuliaLang/Example.jl をインストールする
julia> using Example; Example.hello("World")
julia> hello("Example")
julia> Pkg.rm("Example") # アンインストール
```

[Pkg mode](#) を用いて次のようにして書くこともできる:

```
julia> # ] を入力する
(@v1.9) pkg> add Example
(@v1.9) pkg> ^C # Ctrl と C を同時に押す
julia> # 元に戻る
```

寄り道(Example パッケージについて)

上記のコードは実際に動作する例. Example.jl の実装は概ね次のようになっている. 詳細は [こちら](#) を参照せよ.

```
module Example
export hello, domath

hello(who::String) = "Hello, $who"

domath(x::Number) = x + 5

end
```

export hello と宣言しているので using Example によって hello を即時使うことができる.

```
julia> using Example
julia> hello("World")
"Hello, World"
julia> domath(3)
8
```

名前空間の話

- `export name` によってユーザが `name` を使うことができる
- 便利である一方「この関数はどこで定義しているのかぱっと見わからない」問題がある

```
julia> using Example: hello
julia> hello("World") # hello がロードされていることがわかる
julia> domath(3) # これはできない
julia> Example.domath(3) # このように使う
```

```
julia> using Example: Example
julia> import Example
julia> Example.hello("World")
```

その他下記のようにしてヒントを得ることができる

```
julia> @doc hello("world")
julia> @which hello("world")
julia> @less hello("world")
```

TerminalClock.jl

Example.jl は簡単すぎるかもしれない？

```
julia> using TerminalClock; clock()
```

- [JuliaPackaging/Preferences.jl](#) によって時計の文字盤を制御することができる:

```
$ julia -q
julia> using TerminalClock
julia> tomlfile = joinpath(dirname(pathof(TerminalClock)), "dials", "UnicodeBox.toml");
julia> TerminalClock.set_dials(tomlfile)
julia> exit() # REPL を再起動
$ julia -q
julia> using TerminalClock; clock()
```


Search mode

- REPL で入力した過去の履歴を検索できる。 ~/.julia/logs/repl_history.jl に残っている
- 各々 Ctrl-R, Ctrl-S でサーチができる
- 入力途中の式 + Ctrl-P で以前入力したものを補完してくれる

[OhMyREPL.jl](#) を使う場合

[KristofferC/OhMyREPL.jl](#) を使うと直感的に探すことができる:

```
julia> using OhMyREPL
julia> 1 + 1 # シンタックスハイライトされる
julia> # Ctrl-R で直感的な入力履歴を参照できる
```

Julia で開発する際のワークフロー

Julia で開発する際のワークフロー

- REPL の機能を触れたので Julia での開発ワークフローについて触れる
- 知るとことで人生が豊かになる（かも）

REPL 周り学習リソース

2022 Workshop

Revise.jl の使い方v1.9 で追加された機能に関しての解説も

Project.toml/Manifest.toml

- 大抵のプログラムは何かしらのパッケージ(ライブラリ)に依存
 - ユーザは何をインストールすればいいかを知りたい
- Project.toml にて使用するパッケージを記述
 - Manifest.toml はより詳細な依存関係の情報を格納 (Project.toml から自動生成される)
- `Pkg.activate("path/to/Project.toml")`
 - プロジェクト(環境)をアクティベート(活性化)する.
- `Pkg.instantiate()`
 - Project.toml から Manifest.toml を作る.
 - Manifest.toml から依存関係を解決
- JuliaProject.toml と書くこともできる (混乱がなければ Project.toml でよい) .

例: 研究会の日程, 講演者のアブストを確認する (1)

```
$ cd julia_tutorial/playground/code/mdtable
```

table.jl を動かすための依存関係を解決

```
$ julia -e 'using Pkg; Pkg.activate("."); Pkg.instantiate()'
```

--project, または --project=@. で現在のディレクトリまたは親のディレクトリにある Project.toml から定まるプロジェクトをアクティベートする. [Stack Overflow](#) での解説も参考

```
$ julia --project=@. -e 'using Pkg; Pkg.instantiate()'
```

例: 研究会の日程, 講演者のアブストを確認する (2)

特定の環境で実行する際にも `--project` または `--project=@.` が必要.

```
$ julia --project=@. table.jl 0
```

`--project=@.` を指定するのは面倒なので環境変数を設定するファイル (`.bashrc` や `.zshrc`) のなかに

```
export JULIA_PROJECT=@.
```

を入れておくと幸せになれる.

試行錯誤の方法

```
script.jl を書く  
julia script.jl を実行する # (´・ω・`)起動に時間がかかる 😓  
script.jl を更新する  
julia script.jl を実行する # (´・ω・`)起動に時間がかかる 😓
```

JIT コンパイルが毎回走るので（人間にとって）効率が悪い。コンパイル結果を使い回す運用が必要。

mylib.jl 内部に main 関数があるとする。

```
julia> include("mylib.jl")  
julia> main()  
# 別のターミナルで作業して mylib.jl を編集  
julia> include("mylib.jl")  
julia> main()  
# 別のターミナルで作業して mylib.jl を編集
```

これでも良いが Revise.jl を使うと良い（次のページへ）

Revise.jl

mylib.jl 内部に main 関数があるとする.

```
julia> using Revise
julia> # includet は Revise から export されている
julia> includet("mylib.jl")
julia> main()
```

これで mylib.jl を変更すれば変更後の main() が実行できる. パッケージの開発でも同様.

using Revise; includet("mylib.jl") のようにセミコロン ; で繋げて書くと 1 行でセットアップができるので作業効率が良い.

- VS Code にて Alt-J Alt-O にて REPL を開くと Revise がすでにロードされている
 - Alt-J Alt-W でウォッチするとわかる

Debugger.jl

- 文字通りデバッガ
- 1行1行実行しその都度変数の状態を監視できる
- VS Code でもできるが REPL の上でもできる

```
julia> using Debugger  
julia> include("mylib.jl")  
julia> @enter main()
```

- [VS Code での使い方はこちら](#)

JET.jl

- 型不安定なコードや潜在的なエラーを検知ができる。
 - 開発のモチベーションは [aviatesk/grad-thesis](https://github.com/aviatesk/grad-thesis) などから知ることができる
 - 日本語で読むことができる

型安定・型不安定の話 (1)

- Julia は JIT コンパイル方式で動作をする
 - Julia は関数の引数に渡された値の**型の情報** を元にコンパイルをする
 - 入力の型から出力の型が決定できれば型安定な実装となり効率の良いコードを生成することができる
 - 入力される値によって出力の型が変わると型不安定になる. 速度が求められる箇所では**型安定なコードを書くのが必須**です.
- 型安定なコードを書くように意識すると
 - 型システムに親しめるようになる
- 型システムに親しめるようになる
 - 型安定なコードを書くように意識する
- 型システムについてはチュートリアル第二部で行います. (このスライドは第一部です)

型安定・型不安定の話 (2)

- [What does "type-stable" mean?](#)
- [Type annotation make JIT compile faster?](#)

上記の質問に対する Stefan Karpinski さんの回答:

No. You do not generally need type annotations on function arguments (except to control behavior via dispatch), nor do you need type annotations in local scope. The place that type annotations are essential for performance is on locations: the fields of structs and the element types of arrays and other data structures.

- 実際 [Argument-type declarations](#) にあるように関数の引数に対して型アノテーションが必要な理由は Dispatch, Correctness, Clarity の3つであって、実行速度の理由で必要とするわけではない。

型安定・型不安定の話 (3)

- @code_xxx 系のマクロで Julia がどのようにコードを理解しているか観測ができる
- @code_xxx 系のマクロの説明は Stack Overflow での議論

[What is the difference between @code_native, @code_typed and @code_llvm in Julia?](#) の解説がわかりやすい.

型不安定な例 (1)

ReLU (rectified linear unit) の例が典型的

```
relu1(x) = x > 0 ? x : 0
```

上記のコードは下記と等価

```
function relu2(x)
  if x > 0
    return x
  else
    return 0
  end
end
```

これは @code_lowered で確認することができる. playground/code/code_comparision のツールを用いて比較ができる.

型不安定な例 (2)

```
relu1(x) = x > 0 ? x : 0
```

x の型 T が `Float64` の場合, x の値によって `Int` 型である `0` を返すかもしれないし T を型とする x 自身を返すかもしれない. この曖昧さが型不安定を引き起こす.

型不安定な例 (2)

```
relu1(x) = x > 0 ? x : 0
```

x の型 T が `Float64` の場合, x の値によって `Int` 型である `0` を返すかもしれないし T を型とする x 自身を返すかもしれない. この曖昧さが型不安定を引き起こす.

処方箋

`0` の代わりに `zero(x)` とする. これは x の型に応じた適切なゼロ(加法単位元)を与えてくれる.

Get the additive identity element for the type of x

```
relu3(x) = x > 0 ? x : zero(x)
```

このようにすると x の入力の型が T だった場合, 常に T を型とする出力を与える実装になる(型安定になる).

型安定な例 (1)

```
relu3(x) = x > 0 ? x : zero(x)
```

入力 x は数学的には実数 \mathbb{R} の要素であることに注意する. これは x の型 T が Julia の `Real` のサブタイプであることを要請する:

```
relu4(x::Real) = x > 0 ? x : zero(x)
relu5(x::T) where {T<:Real} = x > 0 ? x : T(0)
relu6(x::T) where {T<:Real} = x > 0 ? x : zero(T)
```

`relu4`, `relu5`, そして `relu6` は等価

Julia 内部では次のような実装になっている (@less `zero(1.0)` などでも検証せよ):

```
zero(x::Number) = oftype(x,0)
oftype(x, y) = convert(typeof(x), y)
zero(::Type{T}) where {T<:Number} = convert(T,0)
convert(::Type{T}, x::Number) where {T<:Number} = T(x)::T
```

型安定な例 (2)

- @code_llvm relu4(rand()) の結果

```
; @ string:1 within `relu4`  
define double @julia_relu4_2276(double %0) #0 {  
top:  
    %.inv = fcmp ogt double %0, 0.000000e+00  
    %1 = select i1 %.inv, double %0, double 0.000000e+00  
    ret double %1  
}
```

- @code_llvm relu5(rand()) の結果

```
; @ string:2 within `relu5`  
define double @julia_relu5_2278(double %0) #0 {  
top:  
    %.inv = fcmp ogt double %0, 0.000000e+00  
    %1 = select i1 %.inv, double %0, double 0.000000e+00  
    ret double %1  
}
```

型不安定・型安定なコードの比較 (1)

```
using Random

"""
型不安定なコード
"""
function main1(N=10)
    rng = MersenneTwister(0)
    s = 0.0
    arr = [relu1(2rand(rng) - 1) for _ in 1:N] # arr は Vector{Real} になる
    for y in arr
        s += y
    end
    s
end

"""
型安定なコード
"""
function main3(N=10)
    rng = MersenneTwister(0)
    s = 0.0
    arr = [relu3(2rand(rng) - 1) for _ in 1:N] # arr は Vector{Float64} になる
    for y in arr
        s += y
    end
    s
end
```

型不安定・型安定なコードの比較 (2)

ほんのちょっとの気遣いで 10 倍コードが速くなる・ちょっとした怠けで 10 倍遅くなる.

```
julia> using BenchmarkTools
julia> N = 100000;
julia> @benchmark main1($N) # 型不安定
BenchmarkTools.Trial: 1391 samples with 1 evaluation.
Range (min ... max): 3.283 ms ... 7.546 ms | GC (min ... max): 0.00% ... 44.61%
Time (median): 3.403 ms | GC (median): 0.00%
Time (mean ± σ): 3.590 ms ± 595.631 μs | GC (mean ± σ): 4.67% ± 10.10%
Histogram: log(frequency) by time
3.28 ms 6.19 ms <
Memory estimate: 3.83 MiB, allocs estimate: 149983.
julia> @benchmark main3($N) # 型安定
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 214.528 μs ... 9.073 ms | GC (min ... max): 0.00% ... 96.93%
Time (median): 249.382 μs | GC (median): 0.00%
Time (mean ± σ): 335.062 μs ± 635.965 μs | GC (mean ± σ): 24.49% ± 12.41%
Histogram: log(frequency) by time
215 μs 4.45 ms <
Memory estimate: 800.88 KiB, allocs estimate: 14.
```

型不安定・型安定なコードの比較(3)

code_warntype, JET.report_opt など検出する. 対応するマクロもある.

```
using JET
```

```
# 色々警告が出る. REPL だと警告は赤色で表示される
```

```
code_warntype(main1, (Int,))
```

```
@code_warntype main1(10) # (InteractiveUtils.code_warntype(main1, (Base.typesof)(10)))
```

```
report_opt(main1, (Int,))
```

```
@report_opt main1(10) # JET.report_opt(main1, (Base.typesof)(10))
```

```
code_warntype(main3, (Int,))
```

```
@code_warntype main3(10)
```

```
report_opt(main3, (Int,))
```

```
@report_opt main3(10)
```

「型安定・型不安定の話 (2)」の続き

- [Type annotation make JIT compile faster?](#)

上記の質問に対する Stefan Karpinski さんの回答:

No. You do not generally need type annotations on function arguments (except to control behavior via dispatch), nor do you need type annotations in local scope. The place that type annotations are essential for performance is on locations: the fields of structs and the element types of arrays and other data structures.

要するに

- 構造体のフィールド
 - パフォーマンスの観点からは `Vector{Real}` ではなく `Vector{Float64}` とするのが良い
- 配列をはじめとするデータ構造に関しての要素型

構造体: アフィン変換の例 (1)

アフィン変換 $f(x) = Wx + b$ を考える

```
julia> struct Affine
           W
           b
       end

julia> (aff::Affine)(x) = aff.W * x + aff.b # Python での `def __call__` 相当

julia> aff = Affine(rand(2,2), rand(2))
Affine([0.05221655288726834 0.4531635136343529; 0.006660217186564732 0.32819491295200176], [0.16834154818251978, 0.6

julia> aff(x)
2-element Vector{Float64}:
 0.24022734713529842
 0.7082428857823445
```

動いてそう.

構造体: アフィン変換の例 (2)

人間のからすると w は行列, b はベクトルということはわかるが, Julia 側からはわからない:

```
julia> # 前のスライドからの続き
julia> typeof(aff)
Affine # フィールドの型情報が見えない

julia> @code_warntype aff(x)
MethodInstance for (::Affine){::Symbolics.Arr{Num, 1}}
  from (aff::Affine)(x) @ Main REPL[2]:1
Arguments
  aff::Affine
  x::Symbolics.Arr{Num, 1}
Body::Any
1 - %1 = Base.getproperty(aff, :w)::Any
   %2 = (%1 * x)::Any
   %3 = Base.getproperty(aff, :b)::Any
   %4 = (%2 + %3)::Any
   return %4
```

フィールドの型情報が見えないので Any がいっぱい. (パフォーマンスの観点からよくない)

構造体: アフィン変換の例 (3)

- フィールドに型をつければいいでしょ？
- 一旦 REPL を再起動して下記を動かす

```
julia> struct Affine
           W::Matrix{Float64}
           b::Vector{Float64}
       end

julia> (aff::Affine)(x) = aff.W * x + aff.b

julia> aff = Affine(rand(2,2), rand(2))
Affine([0.6711341109848074 0.24382214292980786; 0.6036132680677607 0.13610672620163822], [0.2609069141726297, 0.4285

julia> aff(rand(2))
2-element Vector{Float64}:
 0.9884996898337165
 1.04891268376067
```

動いてそう

構造体: アフィン変換の例 (4)

良さそうに見える

```
julia> @code_warntype aff(rand(2))
MethodInstance for (::Affine){::Vector{Float64}}
  from (aff::Affine)(x) @ Main REPL[2]:1
Arguments
  aff::Affine
  x::Vector{Float64}
Body::Vector{Float64}
1 - %1 = Base.getproperty(aff, :W)::Matrix{Float64}
   | %2 = (%1 * x)::Vector{Float64}
   | %3 = Base.getproperty(aff, :b)::Vector{Float64}
   | %4 = (%2 + %3)::Vector{Float64}
   └─ return %4
```

構造体: アフィン変換の例 (5)

次の使い方をするユーザには適用できない 🙄

```
julia> struct Affine
    W::Matrix{Float64}
    b::Vector{Float64}
end
julia> Wf32 = rand(Float32, 2, 2); bf32 = rand(Float32, 2);
julia> Affine(Wf32, bf32).W |> typeof
Matrix{Float64} (alias for Array{Float64, 2}) # Float32 で計算したいのに...
julia> using CUDA
julia> Wcu = CUDA.rand(2,2); bcu = CUDA.rand(2,2)
julia> Affine(Wcu, bcu).W |> typeof
Matrix{Float64} (alias for Array{Float64, 2}) # GPU のリソース使いたかったのに...
julia> W = OffsetArray(rand(2,2), 0:1, 0:1)
```

アフィン変換は汎用性が高いので多くの場面で"いい感じに"振る舞ってほしい!

構造体: アフィン変換の例 (6)

次のようにしてパラメトリック構造体([Parametric Composite Types](#))によって定義する:

```
julia> struct Affine{T1, T2}
    W::T1
    b::T2
end
julia> (aff::Affine)(x) = aff.W * x + aff.b
julia> Wf32 = rand{Float32, 2, 2}; bf32 = rand{Float32, 2};
julia> aff = Affine(Wf32, bf32);
julia> @code_warntype aff(rand{Float32, 2})
MethodInstance for (::Affine{Matrix{Float32}, Vector{Float32}})(::Vector{Float32})
  from (aff::Affine)(x) @ Main REPL[7]:1
Arguments
  aff::Affine{Matrix{Float32}, Vector{Float32}}
  x::Vector{Float32}
Body::Vector{Float32}
1 - %1 = Base.getproperty(aff, :W)::Matrix{Float32}
   | %2 = (%1 * x)::Vector{Float32}
   | %3 = Base.getproperty(aff, :b)::Vector{Float32}
   | %4 = (%2 + %3)::Vector{Float32}
   └─ return %4
```

構造体: アフィン変換の例 (7)

今回のように 2x2, 3x3 程度の小規模の行列では配列のサイズ情報も含めて最適なコードを生成する StaticArrays.jl を用いた方法を採用すると良い.

```
julia> using StaticArrays
julia> struct Affine{T1, T2}
           W::T1
           b::T2
       end
julia> (aff::Affine)(x) = aff.W * x + aff.b
julia> W = @SMatrix rand(Float32, 2, 2)
2×2 SMatrix{2, 2, Float32, 4} with indices SOneTo(2)×SOneTo(2):
 0.106584  0.369698
 0.182812  0.803011

julia> b = @SVector rand(Float32, 2)
2-element SVector{2, Float32} with indices SOneTo(2):
 0.8716512
 0.4812594
julia> aff = Affine(W, b)
julia> x = @SVector rand(Float32, 2)
julia> aff(x)
```

寄り道 (StaticArrays.jl による計算の様子)

人間が直接 $W[1,1] * x[1] + W[1,2] * x[2]$ のように書き下したのとほぼ同じことをしていることがわかる:

```
julia> # 前のページの続き
julia> @code_typed aff(x)
CodeInfo(
  1 - %1 = Base.getfield(aff, :W)::SMatrix{2, 2, Float32, 4} # W = aff.W
    | %2 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32} # W.data <- 生のデータにアクセス
    | %3 = Base.getfield(%2, 3, false)::Float32 # W.data[3] == W[1,2]
    | %4 = StaticArrays.getfield(x, :data)::Tuple{Float32, Float32} # data = x.data
    | %5 = Base.getfield(%4, 2, false)::Float32 # x.data[2] == x[2]
    | %6 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32} # data = W.data
    | %7 = Base.getfield(%6, 1, false)::Float32 # W.data[1] == W[1,1]
    | %8 = StaticArrays.getfield(x, :data)::Tuple{Float32, Float32} # x.data
    | %9 = Base.getfield(%8, 1, false)::Float32 # x.data[1]
    | %10 = Base.mul_float(%7, %9)::Float32 # W[1,1] * x[1] 相当の計算
    | %11 = Base.muladd_float(%3, %5, %10)::Float32 # W[1,2] * x[2] + W[1,1] * x[1]
    | %12 = StaticArrays.getfield(%1, :data)::NTuple{4, Float32}
  (中略)
  └─ return %30
) => SVector{2, Float32}
```

寄り道 (ConcreteStructs.jl でサボる)

```
julia> struct Affine{T1, T2}
           W::T1
           b::T2
       end
```

における T1, T2 は形式的につけるものであって実装の部分としては本質的でない。実装時に人間が意識したくない。

```
julia> using ConcreteStructs # @concrete マクロを提供する
julia> @macroexpand @concrete struct Affine; W; b; end
```

下記のコードと等価

```
struct Affine{__T_W, __T_b} <: Any
    W::__T_W
    b::__T_b
    function Affine(W::__T_W, b::__T_b) where {__T_W, __T_b}
        return new{__T_W, __T_b}(W, b)
    end
end
```


寄り道 (サボり方)

型を意識しなくて済む.

```
julia> using ConcreteStructs; @concrete struct Affine; W; b; end
julia> using StaticArrays
julia> W = @SMatrix rand(2,2);
julia> b = @SVector rand(2)
2-element SVector{2, Float64} with indices SOneTo(2):
 0.620866754177038
 0.5075755639223726
julia> aff = Affine(W, b)
Affine{SMatrix{2, 2, Float64, 4}, SVector{2, Float64}}([0.21089312982809838 0.4062630292552363; 0.039518291001160444
```

今までの議論から分かるように, サボるにはそれなりの教養が必要.

JET.jl を用いた潜在的なエラーの発見 (1)

```
"""
    sumevens(N::Integer)

1 から N までの範囲の偶数を収集する．収集した値を全て足し，その値を返却する．
"""
function sumevens(N::Integer)
    N ≥ 1 || throw(DomainError(N, "`N` cannot be less than 1."))
    arr = []
    for n in 1:N
        if iseven(n)
            push!(arr, n) # Python での `arr.append(n)` 相当
        end
    end
    return sum(arr)
end
```

```
julia> @assert sumevens(8) == 2 + 4 + 6 + 8 == 20
```

JET.jl を用いた潜在的なエラーの発見 (2)

```
julia> sumevens(1) # おっと？  
ERROR: MethodError: no method matching zero(::Type{Any})
```

- $N = 1$ の時は `arr = []` の次にあるループは実質何もしない. 一番最後で `sum([])` を実行することになる.
- `[]` は `Vector{Any}` を型とする要素数が 0 の配列
 - `Vector{Any}` は任意の値が格納できる. 任意の型に対するゼロ元を定義するのができないので Julia ではエラーを返す
 - `no method matching zero(::Type{Any})` が出るのはこのため

```
# sum(arr) は大雑把に言えば下記のようなことをする
```

```
s = <初期値> # これをどう定義するか？  
for a in arr  
    s += a  
end
```

JET.jl を用いた潜在的なエラーの発見 (3)

JET.jl は `sumevens(1)` を実行しなくてもまずい部分を検出してくれる。

```
julia> using JET; report_call(sumevens, (Int,))  
===== 1 possible error found =====  
sumevens(N::Int64) @ Main ./REPL[1]:13  
└─ sum(a::Vector{Any}) @ Base ./reducedim.jl:994  
    └─ sum(a::Vector{Any}; dims::Colon, kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @ Base ./reducedim.jl:998  
        └─ _sum(a::Vector{Any}, ::Colon; kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @ Base ./reducedim.jl:999  
            └─ _sum(f::typeof(identity), a::Vector{Any}, ::Colon) @ Base ./reducedim.jl:999  
                └─ _sum(f::typeof(identity), a::Vector{Any}, ::Colon; kw::Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}) @ Base ./reducedim.jl:999  
                    └─ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}) @ Base ./reducedim.jl:357  
                        └─ mapreduce(f::typeof(identity), op::typeof(Base.add_sum), A::Vector{Any}; dims::Colon, init::Base.InitialValue, itr::Base.Iterator{A}, itrargs::Base.Iterators.EltIterator{A}) @ Base ./reducedim.jl:357  
                            └─ _mapreduce_dim(f::typeof(identity), op::typeof(Base.add_sum), ::Base.InitialValue, A::Vector{Any}, ::Colon) @ Base ./reducedim.jl:357  
                                └─ _mapreduce(f::typeof(identity), op::typeof(Base.add_sum), ::IndexLinear, A::Vector{Any}) @ Base ./reducedim.jl:357  
                                    └─ mapreduce_empty_iter(f::typeof(identity), op::typeof(Base.add_sum), itr::Vector{Any}, ItrEltType::Base.Iterator{A}, itrargs::Base.Iterators.EltIterator{A}) @ Base ./reducedim.jl:357  
                                        └─ reduce_empty_iter(op::Base.MappingRF{typeof(identity), typeof(Base.add_sum)}, itr::Vector{Any}, ::Base.Iterator{A}) @ Base ./reducedim.jl:357  
                                            └─ reduce_empty(op::Base.MappingRF{typeof(identity), typeof(Base.add_sum)}, ::Type{Any}) @ Base ./reducedim.jl:357  
                                                └─ mapreduce_empty(::typeof(identity), op::typeof(Base.add_sum), T::Type{Any}) @ Base ./reducedim.jl:367  
                                                    └─ reduce_empty(::typeof(Base.add_sum), ::Type{Any}) @ Base ./reducedim.jl:347  
                                                        └─ reduce_empty(::typeof(+), ::Type{Any}) @ Base ./reducedim.jl:338  
                                                            └─ zero(::Type{Any}) @ Base ./missing.jl:106  
                                                                └─ MethodError: no method matching zero(::Type{Any}): Base.throw(Base.MethodError(zero, tuple(Base.Any...)))
```

JET.jl を用いた潜在的なエラーの発見 (4)

処方箋は次のとおり. `arr = Int[]` とすればよい:

```
function sumevens(N::Integer)
    N ≥ 1 || throw(DomainError(N, "`N` cannot be less than 1."))
    arr = Int[] # この行を修正した
    for n in 1:N
        if iseven(n)
            push!(arr, n)
        end
    end
    return sum(arr)
end
```

`sum(arr)` は大雑把に言えば下記のようなことをする

```
s = 0 # Int 型のゼロ元
for a in arr
    s += a
end
```

`Int` 型の値を格納する配列に対し, 空の配列の `sum` は 0 とするのが自然だから `sum(arr)` が 0 として計算できる.

JET.jl 周り学習リソース (JuliaCon)

JET.jl の紹介

レクチャー動画

高速化周り学習リソース(JuliaCon)

佐藤さんによる Julia高速化の常識・非常識 @ Bio”Pack”athon2022#12

Cthulhu.jl

- クトゥルフ と発音するらしい
- ローカル変数がどのような型になっているかを観察できる
- 誤解を恐れずに言えば Debugger.jl の型バージョン

BenchmarkTools.jl

ベンチマークをとることができる

```
julia> @benchmark main1($N) # 型不安定な実装
BenchmarkTools.Trial: 1410 samples with 1 evaluation.
Range (min ... max): 3.257 ms ... 6.721 ms | GC (min ... max): 0.00% ... 39.38%
Time (median): 3.367 ms | GC (median): 0.00%
Time (mean ± σ): 3.540 ms ± 588.671 μs | GC (mean ± σ): 4.80% ± 10.34%
Histogram: log(frequency) by time
3.26 ms 5.97 ms <
Memory estimate: 3.83 MiB, allocs estimate: 149983.

julia> @benchmark main3($N) # 型不安定な実装
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
Range (min ... max): 215.560 μs ... 10.235 ms | GC (min ... max): 0.00% ... 97.30%
Time (median): 249.598 μs | GC (median): 0.00%
Time (mean ± σ): 338.071 μs ± 647.998 μs | GC (mean ± σ): 24.88% ± 12.44%
Histogram: log(frequency) by time
216 μs 4.59 ms <
Memory estimate: 800.88 KiB, allocs estimate: 14.
```

Profile.jl/ProfileView.jl/ProfileSVG.jl

- どの部分が時間がかかっているかを調べることができる.
- 先ほどの Cthulhu.jl と連携することもできる

Replay.jl

- REPL での作業を自動化する.
- [ドキュメントはこちら](#)
 - hyrodium さんありがとうございます

Pluto.jl 入門

Pluto.jl について

- Julia が動作するノートブック
 - ソースコード: <https://github.com/fonsp/Pluto.jl>
 - 使い方: <https://github.com/fonsp/Pluto.jl/wiki>
- ユーザが書いたコードが更新されるとセル間の依存関係を自動で解決
- パッケージの依存関係が一つのファイル内に記録される
 - 動作の再現がしやすい(=他者と共有しやすい)
- PlutoUI.jl と連携し簡易 UI を構築することができる
- 頑張ると [Introduction to Computational Thinking](#) のようなリッチな教育資源を構築することができる

Pluto.jl の使い方

インストール

```
$ julia -e 'using Pkg; Pkg.add("Pluto")'
```

起動

デフォルトのポート番号は 1234

```
$ julia -e 'using Pluto; Pluto.run()'
```

Docker の Julia 公式イメージから出発すると次のように起動することができる:

```
$ docker run --rm -it -v $PWD:/work -w /work \  
-p 1234:1234 \  
julia:1.9.2 \  
julia -e 'using Pkg; Pkg.add("Pluto"); using Pluto; Pluto.run(host="0.0.0.0")'
```

Demo

- [ソースコードはこちら](#)
- [Lévy C curve](#)
- [反復関数系](#)
- [RandomLogos.jl](#)

豆知識

ローカルにある自作パッケージを Pluto で動かす場合

- `julia --project=@. -e 'using Pluto; Pluto.run()'` で Pluto を起動
- 下記のセルを追加

```
begin
  Pkg.activate(Base.active_project())
end
```

- [terasakisatoshi/Sacabambaspis.jl](#)
- 詳しいことは [Pluto's built-in package management](#) を読むと良い

Pluto.jl 周り学習リソース (JuliaCon)

Pluto.jl の紹介

開発者らの最近のレクチャー動画

VS Code (1)

- [julia-vscode/julia-vscode](https://github.com/julia-vscode/julia-vscode)

VS Code (2)

VS Code 1.80.0 から VS Code の統合ターミナル上で画像を表示できるようになった:

```
using ImageInTerminal
using TestImages
m = testimage("mandril_color")
```

```
using FileIO, Sixel, Plots
gr()
buf = IOBuffer()
show(buf, MIME("image/png"), plot(sin, size=(500, 300)))
buf |> load |> sixel_encode
```

```
using OpenCV
using ImageCore: normedview, colorview, RGB
using ImageInTerminal

img_bgr = OpenCV.imread("sin.png")
img_rgb = OpenCV.cvtColor(img_bgr, OpenCV.COLOR_BGR2RGB)
# Julia のエコシステムが理解できるデータ構造に変換する
jlimg = colorview(RGB, normedview(img_rgb))
# メモリレイアウトの関係上、空間方向の座標が反転していることに注意。転置する。
jlimg'
```

VS Code (3)

- @testitem によって部分的にテストを実行可能
 - 機能 A はテストが通った
 - 機能 B のデバッグとテストに集中したい
- [JuliaTesting/ReTestItems.jl](#)
- [terasakisatoshi/Sacabambaspis.jl](#)

まとめ

まとめ

- Julia言語の導入, REPL 周りのエコシステムを紹介した
- ここで紹介しきれなかったことはいっぱいある
- 思ったよりも便利な機能があると思います
 - 皆さんの感想を教えてください
- エコシステムが個人や集団からなるコミュニティベースで作られていることがわかる
- Julia言語を多くの人に試してもらい作る側の立場に立てる方が増えれば幸いです

以上